



ulm university universität
uulm

OpenConfigurator - A Practical Approach to Configurator Implementation

Vivian Ulrich Steller

Diplomarbeit an der Universität Ulm
Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

13. August 2012

1. Gutachter:	Prof. Dr. Manfred Reichert
2. Gutachter:	Prof. Dr. Franz J. Hauck
Betreuer:	Dipl.-Inf. Rüdiger Pryss

Abstract

Selling customizable products, tailored to customers demands becomes an increasingly important business opportunity in highly competitive, saturated markets. Enterprises pursuing product customization, employ product configurators to support the *configuration* of customized product variants. Thereby, product configuration systems efficiently facilitate the integration of the customer into the enterprise's value chain and reduce the complexity induced by the manufacturing of customizable products.

Since configurators encapsulate the complete product knowledge, including manifold constraints, implementing them technically in a sustainable way, is challenging and complex. The adequate modeling of configurable products is decisive for the system's maintainability. Therefore, the modeling capabilities must support the precise, correct and compact, yet human readable and verifiable definition of configuration knowledge.

The researched methodology presented in this work, called *OpenConfigurator*, implements a framework for realizing custom product configurators. OpenConfigurator defines a conceptual modeling language to describe configurable products as Java classes, annotated with configuration meta-data. The generic API offered by the framework, allows to instantiate the custom, domain-specific product model, while maintaining the configuration's consistency during the specification process. The framework's capabilities are demonstrated by the implementation of a generic mobile configurator application.

The methodology shows how modern Java EE technologies including JPA, Bean Validation and CDI are used, to simplify the development of configurators. Leveraging standardized technologies, the introduced approach is easy to learn. Moreover, the resulting configurator incorporates a highly flexible, extensible architecture, that strongly fosters maintainability.

Table of Contents

1. Introduction	1
1.1. Motivation	4
1.2. Mission Statement	4
1.3. Context and Scope	6
1.4. Challenges	8
1.5. Thesis Structure	8
2. Product Customization	13
2.1. From Products to Product Configurations	14
2.1.1. Products and Components	14
2.1.2. Variants and Variant Management	15
2.1.3. Customized Products and Product Configurations	16
2.1.3.1. Customizable Areas	17
2.1.3.2. Comparison: Custom-Made Products, Variant Series Prod- ucts, and Customized Products	18
2.1.3.3. Product Configurations	19
2.2. From Classical Production to Product Customization	20
2.2.1. Per-Order/One-of-a-Kind Production	21
2.2.2. Series Production	22
2.2.3. Variant Production	22
2.2.4. Mass Production	23
2.2.5. Product Customization Strategies	23
2.2.6. Mass Customization	25
2.2.7. Strategy Comparison	27
2.3. Product Customization Implementation	28
2.3.1. Mass Customization Achievement	28
2.3.2. Product Customization Value Chain	31
2.3.3. Manufacturing Aspects	33
2.3.3.1. Product and Production Related Prerequisites	33
2.3.3.2. Product Architecture	33
2.3.3.3. Modularity	35
2.3.3.4. Order Fulfillment Strategies	38
2.3.3.5. Production Process Split	39
2.3.3.6. Other Manufacturing Process Related Aspects	41
2.3.4. Economical Aspects	41
2.3.4.1. Market and Marketing Related Prerequisites	41
2.3.4.2. Mass Customization Cost-Efficiency Overview	42
2.3.4.3. Profit Gain Through Product Customization	43
2.3.4.4. Cost Reduction Through Mass Customization	44
2.3.5. Marketing Aspects	46
2.3.5.1. The Buying Process (Customer Perspective)	47
2.3.5.2. The Selling Process (Supplier Perspective)	47
2.3.5.3. The Interaction Process of Product Customization	49
2.4. Application Areas and Examples	50
2.5. Summary	52
3. Configurators	53
3.1. Product Models	54
3.1.1. Components: Structural Decomposition	55
3.1.2. Attributes: Component Characteristics	58
3.1.3. Constraints: Domain Restrictions	60
3.2. Product Configuration	62
3.2.1. Characteristics of Product Configuration	62
3.2.2. Product Configuration Processes	64
3.2.2.1. The Global Configuration Process	64
3.2.2.2. The Configuration Process as a Transformation Process	65
3.2.2.3. The Interactive Configuration Process	67

3.3. Product Configurators	68
3.3.1. Tasks and Requirements	70
3.3.1.1. Tasks of a Configurator	71
3.3.1.2. Additional Requirements	73
3.3.2. Features	73
3.3.2.1. Information	74
3.3.2.2. Specification / Recommendation	75
3.3.2.3. Communication	77
3.3.2.4. Integration and Administration	78
3.3.2.5. Other Features	79
3.3.2.6. Development	80
3.3.3. Categorization	80
3.3.3.1. Application Context	83
3.3.3.2. System Environment	84
3.3.3.3. Modeling Capabilities	87
3.3.3.4. Configuration Characteristics	88
3.3.3.5. Implementation Aspects	96
3.3.4. Benefits	98
3.4. Summary	99
4. Methodology and Conceptualization	101
4.1. The OpenConfigurator Methodology	101
4.1.1. Main Characteristics	102
4.1.2. Configurator Development	104
4.1.3. Other Approaches to Configuration	106
4.1.4. Advantages of OpenConfigurator	107
4.2. Modeling Approach	108
4.2.1. Object-Oriented Product Modeling	109
4.2.2. Product Models in Java	111
4.2.3. Providing Meta-Data with Java Annotations	112
4.3. The Generic Configuration Model	116
4.3.1. Elements of the Generic Configuration Model and Meta Model	119
4.3.2. Model Mapping	120
4.3.3. Facets	121
4.4. Modeling Concepts	123
4.4.1. Structure Modeling	124
4.4.1.1. Components	124
4.4.1.2. Attributes	126
4.4.1.3. Parts	128
4.4.2. Product Modeling	130
4.4.3. Configuration Modeling	132
4.4.3.1. Specification Methods	133
4.4.3.2. Configuration Attributes	139
4.4.3.3. Configurable Parts	142
4.4.4. Data Modeling	143
4.4.4.1. Domains	143
4.4.4.2. Defaults	154
4.4.5. Constraint Modeling	156
4.4.5.1. General Constraint Characteristics	157
4.4.5.2. Supported Constraints	159
4.5. Configuration Procedure	160
4.5.1. The Object-Oriented Configuration Procedure	160
4.5.2. Example Configuration Procedure	163
4.5.3. Configuration Agenda	168
4.6. Summary	174
5. Technical Architecture and Implementation	177
5.1. Technologies	177
5.1.1. Java	177

5.1.2. Java Persistence API (JPA)	178
5.1.3. Bean Validation	179
5.1.4. Contexts and Dependency Injection for Java EE (CDI)	179
5.2. Architectural Overview	183
5.2.1. Multi-Tier Architecture	184
5.2.2. Client Configuration Interface	185
5.2.3. Internal Implementations	187
5.2.4. Extension and Integration SPI	188
5.2.5. Plugins and Connectors	191
5.3. API Usage: Working with OpenConfigurator	192
5.3.1. Bootstrapping and Starting a Configuration Session	192
5.3.2. Performing Configuration	194
5.3.2.1. Navigating through the Generic Configuration Model	194
5.3.2.2. Accessing Meta-Data and State Information	195
5.3.2.3. Specifying Options	195
5.3.2.4. Working with Facets	196
5.3.2.5. Working with the Configuration Agenda	200
5.3.3. Configuration Submission and Further Processing	202
5.4. SPI Usage: Extending and Integrating OpenConfigurator	203
5.5. Summary	204
6. Evaluation and Validation	207
6.1. Case Study: Bike Configurator for the iPad	207
6.1.1. MyCustomBike Inc. - the Custom Bike Company	207
6.1.2. MyCustomBike Inc.'s Product Range	208
6.1.3. The Bike Domain Model	209
6.2. The Generic Mobile Configurator Client	211
6.2.1. Technology	211
6.2.2. Architecture	212
6.2.3. User Interface	214
6.3. Configuration Procedures	216
6.3.1. Creating a New Bike Configuration	217
6.3.2. Configuring the Bike Frame Component	219
6.3.3. Triggering and Resolving a Configuration Error	220
6.3.4. Selecting Wheels	221
6.3.5. Adding an Equipment	222
6.3.6. Submission and Checkout	223
6.4. Validation	224
6.5. Summary	225
7. Summary and Outlook	227
7.1. Discussion	227
7.1.1. Methodology Suitability	227
7.1.2. Implementation Characterization	230
7.1.3. Strengths and Weaknesses	233
7.1.3.1. Strengths	234
7.1.3.2. Weaknesses	236
7.2. Outlook	237
7.2.1. CSP/Constraint Solver Integration	238
7.2.2. Recommendation Integration	239
7.3. Conclusion	240
A. Constraints	243
B. Example Domain Model: Bike	251
B.1. Component Specifications	251
B.2. Source Code	253
C. OpenConfigurator API/SPI	271
Bibliography	311

List of Figures

1.1. Market Trends: From Custom-Made to Customized Products	3
1.2. Context of this Work	7
1.3. Thesis Structure	11
2.1. Customizable Areas	17
2.2. Comparison of Custom-Made Products, Variant Series Products, and Customized Products	19
2.3. Overview of Common Production Processes	21
2.4. Customer Involvement Strategies	24
2.5. Main Principles of Mass Customization	26
2.6. Overview of the Necessary Conditions for Achieving Mass Customization	30
2.7. Value Chain of Product Customization Pursuing Enterprises	31
2.8. General Types of Modularity	35
2.9. Types of Modularization for Platform Designs	37
2.10. Order Fulfillment Strategies	39
2.11. Make-to-Stock and Make-to-Order Manufacturing Scenarios	40
2.12. Influencing Factors for Mass Customization's Efficiency	43
2.13. Interaction Process in the Context of Product Customization	46
2.14. Marketing Mix for Product Customization	48
3.1. Development Process for Custom Products	54
3.2. Basic Generic Meta Model for Configurable Products	55
3.3. Compositional Structure of a Bike	56
3.4. Structural Configuration Decisions	58
3.5. Global Configuration Process	64
3.6. Configuration as a Transformation Process	66
3.7. Configuration Steps	66
3.8. Interactive Configuration Process	67
3.9. Product Configurator as Key Enabler for Mass Customization	69
3.10. Tasks of a Configurator	71
3.11. Overview of the Main Features of a Configuration System	74
3.12. Morphological Box of Configurators	82
3.13. Different Integration Scenarios for Configuration Systems	86
3.14. Integration Levels of Configuration Systems	87
3.15. General Specification Approaches	89
3.16. Relationship Between Configuration Type and Configuration Decisions	90
4.1. The Fundamental Idea behind the OpenConfigurator Approach	103
4.2. Overview of OpenConfigurator Methodology	104
4.3. Overview of Other Methodologies	106
4.4. Example Object-Oriented Product Domain Model	110
4.5. Using Annotations to Map Domains Models to the Generic Meta Model	114
4.6. Using Annotations for Domain Definition	114
4.7. Using Annotations for Declarative User Interface Mapping	116
4.8. The Generic Configuration Model as "Backbone" of the Configuration Process.....	116
4.9. Role of the Generic Configuration Model and its Relationships	118
4.10. UML Class Diagram of the Generic Configuration Model and Related Descriptors	119
4.11. Using Annotation to Control Facets	122
4.12. Runtime Representation of Components	125
4.13. Runtime Representation of Attributes	127
4.14. Runtime Representation of Parts	129
4.15. Component Selection Schema	134
4.16. Incremental Component Selection	135
4.17. Component Configuration Schema	137
4.18. Component Construction Schema	138
4.19. Comparison of Specification Methods	139
4.20. Comparison of Attribute Types	140

4.21. Morphological Box for Domains	144
4.22. Runtime Representation of a Component Type Domain	145
4.23. Component Level and Attribute Level Domains	146
4.24. Domain Definition Concepts	148
4.25. General Structure of OpenConfigurator Constraints	157
4.26. The Object-Oriented Configuration Procedure	161
4.27. Abstract Specification Process for Configurable Products	162
4.28. Configuration Domain Model for Customizable Bikes (Excerpt)	163
4.29. Step 1: Initialized Model and Agenda	164
4.30. Step 2: Bike Type Specified	165
4.31. Step 3: Frame Attributes Specified	166
4.32. Step 4: Wheels Specified	166
4.33. Step 5: Unspecific Equipment Added	167
4.34. Step 6: Equipment Specified	167
4.35. Step 7: Configuration Completed	168
4.36. Configuration Model Result Object	168
4.37. Configuration Process as Screenflow	169
4.38. Configuration Process Customization	169
4.39. Relationship between Configuration Decisions, Tasks, and the Generic Configuration Model	170
4.40. Agenda Tasks Type Hierarchy	171
4.41. Example Task Hierarchy	173
5.1. Architecture Overview	183
5.2. Implementors	189
6.1. Components of a Bike	209
6.2. UML Class Diagram for the Bike Domain Model	210
6.3. Generic Mobile Configurator Architecture	213
6.4. Mobile Configurator Wireframe	214
6.5. Hierarchical Agenda Navigation	215
6.6. Exemplary Screenshot of the Generic Configurator	216
6.7. Initial Configurator State after Creating a New Configuration	217
6.8. Configuration State after Type Selection	218
6.9. Specification Dialog of the Frame Component	219
6.10. Visualization of Constraint Violations	220
6.11. Wheels Selection Dialog	221
6.12. Plural Part Specification	222
6.13. Configuration Submission	223
7.1. Classification of OpenConfigurator within the Morphological Box for Configurators	231

List of Tables

4.1. Mapping between Object-oriented Concepts and Product Modeling Concepts	109
4.2. Mapping between the Java Concepts and the Generic Configuration Model	121
4.3. Structure Modeling Annotations	124
4.4. Component State Variables	125
4.5. Component Events	126
4.6. Attribute State Variables	128
4.7. Attribute Events	128
4.8. Part State Variables	129
4.9. Part Events	130
4.10. Product Modeling Concepts	130
4.11. Configuration Modeling Concepts	132
4.12. Type Domain Annotations	149
4.13. Implicit Attribute Domains	150
4.14. Attribute Domain Annotations	150
4.15. Part Domain Annotations	152
4.16. Decision Default Behavior	154
4.17. Default Value Definition Annotations	154
4.18. Configuration Agenda Tasks	171
5.1. Global Events Emitted via the CDI Event Bus	190
A.1. Logical/Arithmetical Comparison Constraints	244
A.2. Cardinality Constraints	245
A.3. Compatibility Constraints	246
A.4. Incompatibility Constraints	248
A.5. Complex Constraints	249

List of Examples

4.1. Example JavaBean Domain Model	111
4.2. Exemplary Annotation Definition and Usage	113
4.3. Using Annotations to Define Constraints	115
4.4. Usage of the @Component Annotation	125
4.5. Usage of the @Attribute Annotation	127
4.6. Usage of the @Part Annotation	128
4.7. Usage of the @Product Annotation	132
4.8. Modeling Selectable Components	134
4.9. Modeling Configured, Parameterized Components	136
4.10. Modeling Constructed Components	138
4.11. Usage of the @Calculated Annotation	142
4.12. Use of the @Configured Annotation	143
4.13. Implicit and Explicit Definition of Type Domains	149
4.14. Implicit and Explicit Definition of Attribute Value Domains	151
4.15. Implicit and Explicit Definition of Part Value Domains	153
4.16. Implicit and Explicit Definition of Default Values	155
4.17. Constraint Annotation Usage	156
4.18. Constraint Annotation Usage Methods	158
5.1. SPI Usage Example	203
7.1. Modeling Customizable Products with OpenConfigurator Concepts	228

1

Introduction

A Bit of History

Handcrafted work characterized the production in pre-industrial times. Custom-made products were developed and manufactured for customer-specific demands. These hand-craft activities required specialized skills and detailed know-how that was only implicitly available to the craftsman. This knowledge was trained and passed down to generations [Lindemann2006a, p. 2].

While in some business areas craft manufactures still exist, the replacement of human work by machinery marked the beginnings of the industrialization in the mid 19th century. An increasingly automated production, the division of labor and continuous production with the help of assembly lines as well as other principles of mass production helped enterprises in the early 20th century like Ford to great success. Singular product and process innovations lead to quite productive manufacturing processes and lowered costs, which not only raised the companies' profits, but generally increased the mass purchasing power through heightened incomes. An increased production performance was the ultimate goal and the main driver for production processes for a long period of time. The market conditions of those times fit perfectly with what mass production is made for (cp. [Holthöfer2001, p. 9]):

1. Producing and selling *large amounts of items*,
2. for *homogeneous markets*,
3. with *stable demand*,
4. over a *long period of time*.

The Market Shift

However, an important shift in this market situation happened in the previous decades: "[...] The main condition that have ensured a successful mass production, namely stability and demand homogeneity are no longer available and do not coin the actual picture of the business environment [today]" [Blecker2005, p. 10]. In a wide variety of markets, the potentials of mass production have been exhausted, in some of them almost completely (cp. [Reichwald2009, pp. 23], [Lindemann2006a], [Holthöfer2001, pp. 9], [Piller2006]):

Increasingly saturated markets lead to buyer markets. In previously existing seller markets, the suppliers dictated what was sold at what prices. However, nowadays it's more and more the consumer that has the power to influence prices and functionality (often referred to as *customer empowerment*). The consequence of this is that large amounts of items aren't marketable anymore and in general a higher fluctuation of demand can be observed.

Globalization leads to tougher competitive conditions. More suppliers for the same or similar products result in higher price and innovation pressure for each supplier, but especially for those not producing in low-wage countries.

Influence of information- and communication technology lead to higher information transparency. The evolution of information technology allow much simpler and faster comparison of products and prices. Ratings for products and suppliers together with customer recommendations are often thoroughly considered by customers today. The emergence of large communities of users offer additional sources for information to the customer. Consequences of the increased market transparency are lowered risks and decreased prices from a consumer's perspective, and more keen competitive conditions from a supplier's point of view.

Innovation and technological progress lead to shorter development times and product life cycles. In general, faster innovation and technological progress can be observed, which results in more fluctuating product ranges. Today, enterprises are required to quickly and flexibly react to market changes [Reichert2012].

Increased customer demands lead to heterogeneous markets. Today's customers are both more cost-conscious and demanding. They request products precisely fitting their requirements. In the area of capital goods, customized production factors and machinery, that are highly adapted to meet the specialities of the customer's value creation activities, formed valuable competitive advantages for years. In consumer markets

- changes in people's professional environment,
- socio-demographic developments resulting in more prosperity (increased income, more spare time, higher education),
- an intensified relationship to adventure- and design-orientation, and
- a new awareness of quality and functionality

lead to an increased demand in customer tailored products. Consumers often want to differentiate themselves through individually designed products.

Enterprises recognized those market developments and realized, that not a product or technology alone, but the ability to *sell* those products come to the fore. Especially customer orientation and customer loyalty are two of the most important success factors.

As a consequence, companies begun diversifying their product range by producing **variants** [Hallerbach2010]. The basic principle of variant series production is a compromise between high productivity and sufficient satisfaction of individual customer needs. This compromise is achieved by reducing the amount of work required to fulfill customer wishes to the greatest possible extend, e.g., by combining pre-developed / pre-produced components.

However, addressing a wide variety of customer demands using this strategy has its limitations [Lindemann2006a, p. 2]. As Pine and Gilmore formulated: "Fundamentally, customers do not want choice; they just want exactly what they want." [Pine1999, p. 76]. In other words, they want fully **customized products**.

Figure 1.1, "Market Trends: From Custom-Made to Customized Products" illustrates this transition schematically (cp. [Lindemann2006a, pp. 2]).

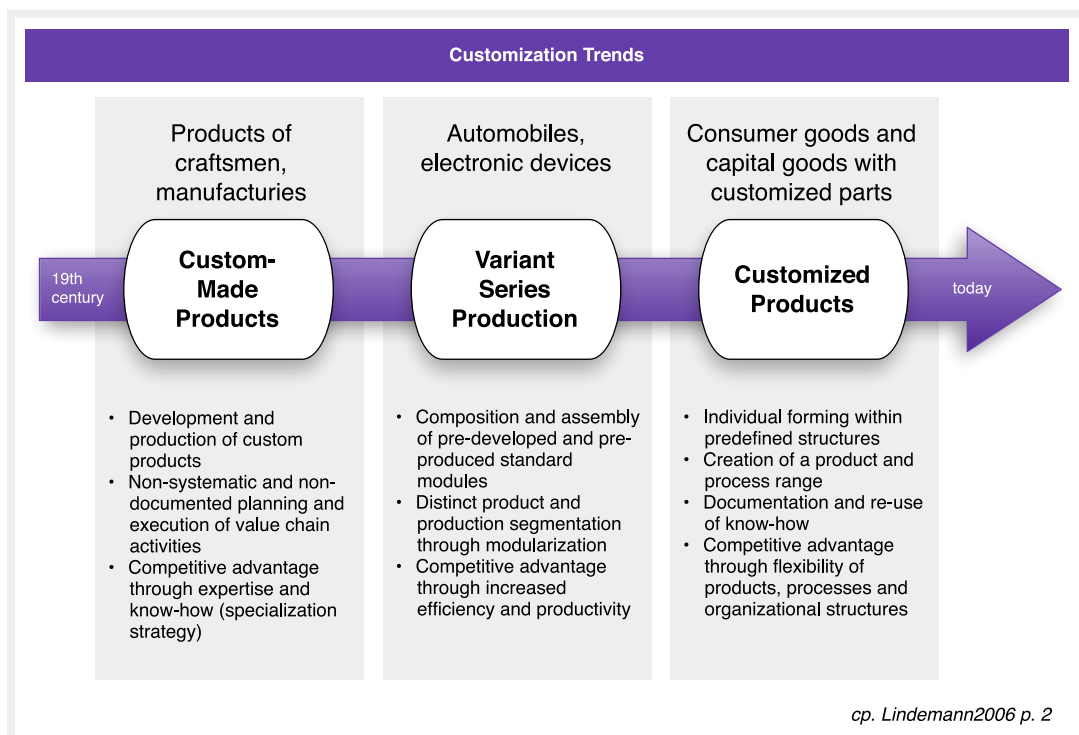


Figure 1.1. Market Trends: From Custom-Made to Customized Products

The production of customized goods requires a shift in numerous activities related to the value chain all over the company [Knuplesch2010]. Suppliers are demanded to create flexible product structures, processes and production techniques as well as to establish agile organizational structures that allow optimal reaction on today's market demands in order to achieve competitive advantages [Dadam2009][Reichert1998]. The acquisition of knowledge about customers and their preferences increasingly gain importance [Mundbrod2012]. All those activities need to be backed with well-integrated, efficient information and communication systems that support the entire value chain of a company: from product development, production and distribution to sales and after-sales tasks. The manufacturing of products according to individual customer needs is referred to as **product customization** [Blecker2005, p. 2].

Business Today

Today's enterprises more and more face increasing demands on product quality and individuality by their customers [Lohrmann2012]. In many business areas, consumers prefer custom-tailored products over mass produced, one-of-a-kind goods. Fulfilling these demands efficiently and at affordable prices is a big challenge for manufacturing companies. However, the capability to produce and offer customized products, that exactly meet the customer's needs, become an important success factor and a promising business opportunity simultaneously. In fact, due to high market competition and price pressure, offering customer-tailored products as a differentiation strategy is often a necessity rather than a sole management decision.

Selling and producing customized products is challenging, because the customer needs to directly interact with the supplier for the purpose of specifying his requirements on the manufactured good. Not only does this involvement of the customer complicate the manufacturer's business processes, but also bears the risk that individually specified requirements cannot be fulfilled and the customized product variant isn't producible after all.

In order to avoid these difficulties and to efficiently integrate the customer into the company's value chain, modern enterprises employ **product configurators**, also called *prod-*

uct configuration systems. With the help of product configurators, the customer's requirements on the to be produced product can be captured, while ensuring validity of the so defined product configuration in real-time. Configurators have several key characteristics, that make them unique among other business applications within the enterprise. Product configurators...

- are powerful, 24/7 hours available marketing and sales tools
- reduce required sales personnel with expert knowledge, particularly for complex products
- reduce time (and money) required to create sales quotations and thus increase sales "throughput" [Lanz2010]
- avoid specification errors and make product variability more transparent
- can be used to generate specification documents, like sales quotations, product specifications, bill of materials, work plans, etc. fully automated

For these reasons, product configuration systems are valuable business tools for any company practicing product customization. They incorporate a huge amount of product knowledge, including available product variants, constraints and other product related logic.

1.1. Motivation

Especially for complex application domains, defining and maintaining such configuration knowledge as well as implementing the actual configuration system is a difficult task. Even with only a few product attributes, the variability and amount of resulting product variants can quickly explode. Therefore, suppliers restrict the available variants by formulating certain constraints, sometimes for technical, other times for marketing reasons. Defining, implementing and maintaining a product's variability and constraints as well as the overall system in an effective, simple, well-understandable and verifiable manner, is thus, eminently important.

Nevertheless, many of today's configuration solutions are still implemented "from scratch", as one-off applications build for a singular purpose or use case only. Products and product variability is often modelled without following a standardized, well defined procedure. This results in configuration process knowledge being intermixed with product information, effectively leading to maintainability problems. Moreover, many configuration vendors employ proprietary technologies and concepts, making learning and development of configuration knowledge bases a time consuming, difficult and cost intensive task. Even worse, these problems hinder the resulting configurator from being updated, extended, integrated or being otherwise enhanced. Consequently, configurators often do not yield the potential they could be used for.

Granted that sophisticated configurators are a complex application domain, developing a configuration system that fulfills the mentioned attributes and scales well with the to be configured product's complexity, is - as we believe - an unreasonably complicated issue yet. We argue, that applying modern programming paradigms and approaches could strongly simplify the development of advanced configurators, which are well testable, integrable and extensible. Thereby, in general, the employed approach can be a lot easier to learn and much more productive after all.

1.2. Mission Statement

In this work, we aim to develop a state-of-the-art framework for creating custom configurators, which are utilized for the specification of simple as well as complex customizable products. By establishing a unified, standard modelling approach for configurator applications, that utilizes modern technologies and patterns, not only the benefit-cost ratio for configurator development projects is strongly increased but also value-added features can be implemented once for any kind of configuration problem.

We feel that the emergence of new standards and technologies in the Java space (and especially in the area of Java EE) strongly simplify application development in general, but specifically in the area of configurator applications. These technologies create new, unique opportunities for building flexible configurator applications. In fact, we see our attempt as a novel approach, that combines advantages on various stages.

Our ultimate goal is to provide the average Java developer a tool, that allows him to quickly implement high quality configurators fostering scalability, extensibility, integrability and maintainability. While focussing developers to work with the framework, we primarily target the creation of configurators for medium to highly complex configuration domains, where verifiable correctness of configuration results is inevitable. This is why the introduced conceptualization itself and the development approach promoted by it is strongly designed with testability in mind.

Concrete Objectives

The methodology for developing configurators as defined in this work can be characterized by the following attributes:

Model-driven. The configurator developer solely models the configuration problem and attaches meta-data to it, the rest of the application is dynamically generated at runtime.

Generic. The methodology and the framework can be applied to arbitrary product domains: the system's completely implementation is independent of the modeled domain.

Standards based. The methodology is designed to be compliant with or build on existing, widely accepted standards where possible. This not only applies to technologies (e.g., Java EE), but also on practically applied development procedures (e.g., source version control support).

Considering non-functional aspects. The non-functional aspects are central aspects of the methodology and are thus focussed in particular, including:

- **Ease-of-learning and ease-of-development.** The practical applicability of the framework for average, non-expert Java developers is a major driver for decisions taken in this work.
- **Scalability.** The framework should be able to cover small, medium-sized and large projects equally, from simple to complex application domains. Thereby, the development and maintenance effort may not disproportionately grow with the project's complexity.
- **Extensibility and flexibility.** Being extensible and flexible for changes to be incorporated in the future is seen as a fundamental aspect, since requirements for today's applications get more and more demanding.
- **Integration and interoperability.** Similarly to the previous point, integration and interoperability aspects play an essential role in today's business software landscape. Especially, this counts for configurators, which link different other IT systems.
- **Testability and maintainability.** The ability to verify the correct behavior of the developed configurators is inevitably important for complex configurators, as configuration errors quickly become very costly. Moreover, the long-term maintainability of the developed applications is crucial for the ongoing success of such a system.

Concretely, in this work, we'll do the following:

1. **Develop a modeling approach for configuration problems.** During this work, we conceptualize a modeling approach that can be used to describe configuration problems in an

object-oriented fashion, so called *configuration domain models*. We show how the approach is realized within the context of modern Java programming frameworks. Specifically, the modeling approach features the following characteristics, which will be described in detail later in this work:

- domain specific
 - object-oriented / component-based
 - declarative meta-data (constraints, domains)
 - plain Java, well integrated with Java EE¹, particularly: JPA² (used for accessing configuration data / domain values) and Bean Validation (used for constraint definition and validation)
2. **Develop a generic configurator framework.** Based on the designed modeling approach, we conceptually design and implement a framework that allows to create valid instances (configurations) from the configuration domain models. The presented generic configurator can be characterized by the following facts:
 - the framework "interprets" the domain model and associated meta-data
 - it features a generic, domain-independent API³
 - it provides a low-level configuration API and a high-level task-based API for performing product configuration
 - it's highly extensible through the provided SPI⁴
 - is technically fully Java EE 6 web profile based (CDI⁵, JPA, Bean Validation)
 3. **Implement a mobile configurator for bikes.** For evaluation purposes, we will visually design and realize an example configurator for bikes. Effectively, we'll, thereby, implement a generic configurator client (based on the developed configuration framework) for mobile devices, that can be backed with arbitrary configuration domain models. The concrete characteristics of the realized solution are:
 - it's web-based and primarily targets mobile devices such as Apple's iPad, but can be accessed by desktop web browsers as well
 - example domain model: bikes. The model can be easily shared with other applications as it's implemented as a simple JavaBean class model
 - concrete technologies involved: HTML5/CSS/JavaScript, Vaadin, CDI and our configuration API

The framework introduced in this work is named *OpenConfigurator*. The name shall emphasize the *openness* of the framework and its flexibility. Consequently, throughout the work, we will mostly refer to the developed methodology and/or its implementation as "OpenConfigurator methodology" or "OpenConfigurator approach".

1.3. Context and Scope

The development of a generic configuration framework requires a deep understanding of the fundamentals of configuration, namely *product customization*, the business discipline practiced by companies offering product configurators. By analyzing product customization from various perspectives, we'll be able to identify important requirements to be fulfilled by the conceptualization and implementation. Learning about the business context, in which configurators are applied, also helps to understand the overall role of these tools within an enterprise. Figure 1.2, "Context of this Work" illustrates the context of this work:

¹Abbrev. *Java Enterprise Edition*

²Abbrev. *Java Persistence API*

³Abbrev. *Application Programming Interface*

⁴Abbrev. *Service Provider Interface*

⁵Abbrev. *Contexts and Dependency Injection*

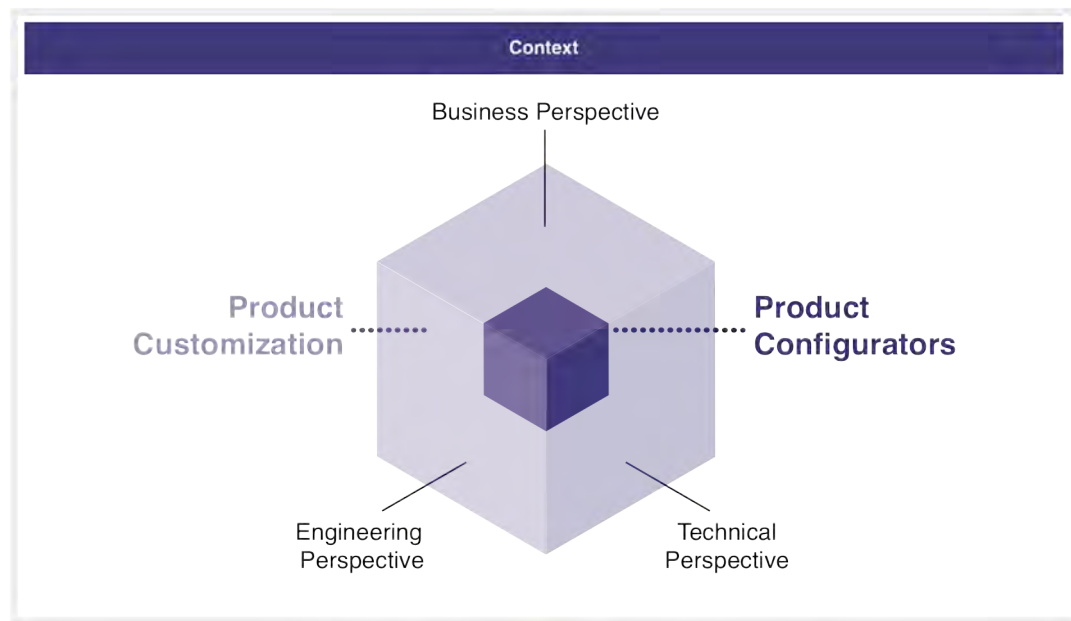


Figure 1.2. Context of this Work

So, *product configurators* are understood as a specific topic within the broader scope of *product customization*, which builds the global context of our work. We will look at both from three different perspectives: the *business perspective*, the *engineering perspective*, and the *technical perspective*. The latter one, however, will be focused primarily.

The targeted audience of this work is three-fold:

Managers. On the one hand, in this work, business managers will find a comprehensive overview about the topics of product customization as a business strategy and product configurators as enablers for that strategy. For them, the business topics of Chapter 2, *Product Customization* and Chapter 3, *Configurators* will be of primary interest.

Solution architects/engineers. Configurator solution architects or engineers with specific knowledge in the targeted domain, on the other hand, will gain from this work by learning a practical approach to modelling configuration knowledge bases and realizing a custom product configurator. They will particularly be interested in the engineering related topics of Chapter 2, *Product Customization*, Chapter 3, *Configurators* and Chapter 4, *Methodology and Conceptualization*.

Developers. Configurator developers and integrators will find a detailed introduction to the architecture and implementation of product configuration systems. Developers should read at least Chapter 4, *Methodology and Conceptualization*, Chapter 5, *Technical Architecture and Implementation* and Chapter 6, *Evaluation and Validation*.

By serving these three quite different groups of readers together, we argue that our work provides a unique overview of the entire topic of configurator realization. In fact, we argue that the work at hands, bridges a huge gap between plain theory and strategic concepts on the one hand, and the practical implementation detailed down to code lines, on the other hand.

Filling that gap within an acceptable amount of pages, we had to narrow the overall scope of the work. The thesis' scope can be defined as follows:

- provide a comprehensive overview to the topic of product customization and product configuration in particular, for all stakeholders involved in configurator realization projects (addressed by Chapter 2, *Product Customization* and Chapter 3, *Configurators*).

- define a meaningful (yet not necessarily complete) conceptualization for modeling real-world configurable products. It is sufficient, if the conceptual foundation is detailed enough to be evaluated with regard to its practical applicability (addressed by Chapter 4, *Methodology and Conceptualization*).
- provide a prototypically implementation of the framework that allows to assess, whether the approach is feasible or not. The implementation doesn't aim to be feature complete but endeavours to show that the methodology can be implemented technically (addressed by Chapter 5, *Technical Architecture and Implementation*).
- proof the applicability, appropriateness and the technical feasibility of the approach by demonstrating a real-world use case. While the implemented example should be reasonable and is meant to serve the purposes of showing particular functionalities, a feature complete implementation is not in the scope of this work (addressed by Chapter 6, *Evaluation and Validation*).
- discuss and assess the realized approach and show future usage as well as extension possibilities (addressed by Chapter 7, *Summary and Outlook*).

1.4. Challenges

We claim that OpenConfigurator introduces a unique approach to deal with configuration problems. During our research we didn't find any solution that allows implementing custom configurators the way OpenConfigurator aims to do. In fact, during researching, conceptualizing and implementing on the framework, we identified manifold challenges:

New technologies, "from scratch" implementation. The technologies combined in OpenConfigurator are simply too new that existing configurator solutions could have been using them already. Unfortunately, the consequence of this is that we need to implement OpenConfigurator "from scratch", that is, from the core component model to the user interface. Consequently, the work presented in this thesis doesn't aim to be complete. Instead, OpenConfigurator is a "proof of concept", a prototypical implementation of a configurator framework. Nevertheless, we spend much effort into implementing things aiming at a high quality.

The gap between the complex domain of product configuration and ease-of-development. The main challenge for OpenConfigurator is to bridge the gap between the complex domain of configuration software and the ease-of-development requirement demanded by application programmers. We want to create a framework that allows an average Java EE application programmer to easily implement real-world configurators of various complexity, without forcing him to learn new technologies, new programming languages or new implementation paradigms he isn't used to. Instead, we try to combine existing, well-known technologies and implementation concepts to create configurator applications providing a steep learning-curve and a unique developer experience.

Of course, we realize that this mix of technologies to solve configuration problems changes the way of covering configuration problems. In fact, the framework doesn't aim to be the "all catching", single and best solution for all configuration problems. However, existing configuration solutions often induce a high complexity even for simple configuration problems and make product configurators difficult to maintain. We think, a developer friendlier, more extensible and much easier to maintain solution would be appropriate.

1.5. Thesis Structure

Having motivated our work (*why we do it*) and described the main goals (*what we'll do*) to be accomplished by this work, including the major challenges influencing it, we will describe our solution strategy (*how we do it*), next:

Chapter 2, Product Customization. In the second chapter, we will *analyze the business area of "product customization"* in detail. We will introduce product and process related basics and discuss the implementation of product customization strategies from both an engineering and business perspective. Specific questions relevant to the overall theme answered by this chapter include:

- What are customizable products and how are they described?
- What's special about the production and selling of customizable products and how are these specifics addressed?
- What role do configurators play within a product customization scenario?

The main tenor of this chapter is to understand companies pursuing product customization and the way they are working. The knowledge of the enterprises that employ product configurators, is essential for the conceptualization and implementation of a configurator framework.

Chapter 3, Configurators. This chapter basically *analyzes the requirements on configuration systems* and explains their functionality in detail. We transfer the knowledge about customizable products, acquired in the previous chapter, into the technical world by discussing product configuration models. Then, we will describe product configuration processes from various perspectives, before investigating product configurators as software tools in great detail. A presentation of benefits rounds up the last section. Potential questions, answered by this chapter include:

- How are customizable products represented within configuration systems?
- What happens during configuration in detail?
- What are product configurators precisely and what functionality do they offer?
- What are the benefits of configurators?

Having studied this chapter, the reader should be familiar with configuration tools. Moreover, he should realize the complexity of these systems and should basically understand the scope of configurators in terms of their functionalities.

Chapter 4, Methodology and Conceptualization. Chapter 4 introduces our own methodology to the implementation of custom configurators, called *the OpenConfigurator approach*. We are going to *explain our methodology* as follows: first, we will characterize our methodology in general and present the main ideas behind its conceptualization. Then we'll disclose fundamental aspects of our modeling approach and describe the most relevant concepts from a high-level perspective. Next, we'll describe the available modeling capabilities in full detail, which includes the modeling of structural aspects, product information, configuration semantics, data access and model constraints. Finally, we will discuss the configuration procedure, as intended by our framework, by walking through a concrete configuration example. By reading this chapter, you'll receive answers to the following questions:

- What are the fundamental ideas behind the OpenConfigurator approach and what makes this approach special?
- What are the main tasks of the OpenConfigurator framework and how does it accomplish these conceptually?
- How can customizable products be modeled with OpenConfigurator's concepts?
- How does the framework transform custom domain models into a concrete configuration process?

Chapter 5, Technical Architecture and Implementation. In Chapter 5, we are going to *explain the architecture and implementation of the runtime component* of the OpenConfigurator framework. We will discuss the main technologies the framework bases on, along with their relationship to our implementation. Then, we'll describe OpenConfigurator's architecture in

detail before showing practically, how the framework provided API and SPI is used. In this chapter, you'll find answers to the following questions:

- What are the most important technologies with regard to OpenConfigurator and how do these technologies relate to the framework?
- How do configurators build on top of the framework look like, technically? And how is the framework's internal architecture organized?
- How can configuration activities be performed with the framework, concretely?
- How can the framework be extended?

Effectively, this chapter answers one of the most important questions: *how does it all work, technically?*

Chapter 6, *Evaluation and Validation*. The second to last chapter presents a case study, that *demonstrates the practical application of our methodology*. We'll develop an exemplary domain model for customizable bikes and implement a custom configurator for Apple's iPad. Thereby, we'll realize a generic, HTML5 based mobile configurator based on the OpenConfigurator framework and the Vaadin technology. Chapter 6 will answer the following questions:

- More specifically: how can a real-world customizable product be modeled using OpenConfigurator's modeling concepts?
- How can an HTML5 based, generic mobile configurator be implemented on top of the framework?
- How does a custom configurator, realized with OpenConfigurator look like visually?
- Does OpenConfigurator address practice relevant, project related issues adequately?

Again, ultimately, this chapter provides an answer to the question: *how is the OpenConfigurator methodology applied practically?*

Chapter 7, *Summary and Outlook*. In the last chapter, we are going to *conclude this work* by recapitulating, how OpenConfigurator aims to satisfy the earlier stated requirements and motivations for building custom configuration systems. Moreover, we will characterize the implemented framework from a high level perspective and discuss various strengths and shortcomings. Then, an outlook to future development ideas follows. Finally, we'll conclude the work by providing a short resumé. These are the questions you can expect to be answered by the last chapter:

- Does OpenConfigurator's modeling approach address the modeling requirements for configurable products adequately?
- How can the framework implementation be characterized?
- What are the strengths and weaknesses of the system?
- How can the framework be improved in the future?

Figure 1.3, "Thesis Structure" provides an overview of the chapters, that make up this work.

Introduction 1	Motivation 1.1	Mission Statement 1.2	Context and Scope 1.3	Challenges 1.4	Thesis Structure 1.5
Product Customization 2	From Products to Product Configurations 2.1	From Classical Production to Product Customization 2.2	Implementing Product Customization 2.3	Application areas and examples 2.4	
Configurators 3	Product Models 3.1	Product Configuration 3.2	Product Configurators 3.3		
Methodology and Conceptualization 4	The OpenConfigurator Methodology 4.1	Modeling Approach 4.2	The Generic Configuration Model 4.3	Modeling Concepts 4.4	Configuration Procedure 4.5
Technical Architecture and Implementation 5	Technologies 5.1	Architectural Overview 5.2	API Usage: Working with OpenConfigurator 5.3	SPI Usage: Extending and Integrating OpenConfigurator 5.4	
Evaluation and Validation 6	Case Study: Bike Configurator for the iPad 6.1	The Generic Mobile Configurator Client 6.2	Configuration Procedures 6.3	Validation 6.4	
Summary and Outlook 7	Discussion 7.1	Outlook 7.2	Resumé 7.3		
Appendix	Constraints A	Example Domain Model: Bike B	OpenConfigurator API/SPI documentation C		

Figure 1.3. Thesis Structure

Each chapter will end up with a short summary, highlighting the most important facts. If you're in a hurry, you may want to skip an individual chapter and just read its summary to get a basic idea of what's going on. This helps to understand subsequent chapters.

Now, let's begin with the theory.

2

Product Customization

In order to successfully establish a configuration system within an enterprise, the company's business strategy (including management concepts and production strategies) must be adapted to deal with configurable products. In general, we refer to a business strategy that involves user interaction as part of the value creation chain as *product customization* strategy. According to Blecker et al. "product customization can be defined as producing a physical good or a service that is tailored to a particular customer's requirement" [Blecker2005, p. 11] .

In this chapter, we will take a closer look at product customization from a business / manufacturing point of view while subsequent chapters will focus on technical aspects. We'll first introduce some basic terms related to customizable products. Then, we'll cover production processes widely used in manufacturing companies today. These production processes include:

- one-of-a-kind production,
- series production,
- mass production, and
- mass customization or product customization in general.

Next, we'll focus on product customization in particular and discuss manufacturing related aspects such as modularization and economical aspects including benefits of customer-tailored production. Finally, we'll cover the interaction process between supplier and customer during sales. Examples of application areas that frequently employ customization strategies end up this chapter.

The aim of this chapter is to provide well-grounded background knowledge about the topic of product customization. The terms and concepts presented in this chapter are not only meant to establish a common understanding and vocabulary across different stakeholders involved in configurator projects. Instead, the topics discussed throughout this chapter can be considered fundamental for evaluating the applicability and benefits offered by product customization strategies.

2.1. From Products to Product Configurations

In this section, we'll introduce several terms and definitions related to product customization and manufacturing in general.

2.1.1. Products and Components

Products

The term "product" is used in many different contexts including mathematical, chemical, commercial and others, which is why several definitions exist. In the sense of its commercial meaning, which is beyond the technical definition, the only relevant definition for our purposes, a *product* refers to "the result of a manufacturing process or an offered service"¹. However, to provide a more precise definition it's useful to examine the term from different perspectives.

From an engineering perspective, a product can be seen as a ready-made respectively sellable, physical or non-physical commodity, that has been manufactured using the elementary factors work, equipment and raw materials [Bieniek2001]. According to Krug [Krug2010], the commodity (manufactured item) mentioned in this definition refers to "a self-contained, functioning object as a manufacturing result that consists of a number of groups and/or parts" according to DIN 6789. On the one hand, this explanation already shows that a product is a result of a production process (which is covered in more detail in Section 2.2, "From Classical Production to Product Customization"). On the other hand, the description of a manufactured item provided by the DIN norm is quite close to the technical perspective definition that is primarily referred to throughout this work (cp. [Krug2010, p. 6]):

Product A *product* is characterized through a number of *attributes* and consists of multiple *parts (components)*.

Furthermore, Krug establishes the following rules for the use of the term "product" in his work, which we also consider to be valid in ours: a product

- is used as a synonym for good, ware or service,
- is the result of a production process,
- has a serviceable or a dissipating character,
- can be of substantial (physical) or insubstantial (non-physical) nature,
- can be categorized along different dimensions and
- services as integral instrument to fulfill customer needs.

From a production point of view, products are sometimes referred to as *assemblies*. A product family describes "a group of different products which are created from a common set of components (modules) and which have a number of common characteristics" [Hvam2008, p. 31].

Components

Products are composed of *elements* and *sub-assemblies*². These parts, that make up a product, are often referred to as *components* or *modules*. Precisely, a component can be defined as follows:

Component/Module (physical) A *component/module* refers to a part of a product (or sub-part of another part), which is characterized by attributes and which may consist of any number of sub-parts (sub-components / sub-modules).

¹From <http://de.wiktionary.org/wiki/Produkt>, last accessed July 29th, 2012.

²See http://www.fml.mw.tum.de/fml/index.php?Set_ID=320&letter=P&b_id=4246437B-4641-3737-422D-303536332D34, last accessed July 29th, 2012.

Based on the previous definition, the terms "element" and "sub-assembly" can be explained as follows: a *composite component* (sub-assembly) refers to a component that is assembled of other components, that is, a component containing at least one sub-component. In contrast, a component without any further divisible sub-component is considered an *atomic component* (elements). In contrast to sub-assemblies, elementary components cannot be deconstructed without physically breaking them.

The fundamental principle behind the decomposition of a product into individual components is *modularization*, which is described in more detail in Section 2.3.3.3, "Modularity".

2.1.2. Variants and Variant Management

Variants

According to DIN 199-1, a *variant* basically describes an object of similar form or functionality usually with a high degree of identical elements or sub-assemblies [Schönsleben2000]. These objects at least share similarities regarding geometry, material or technology, and result from the combination of different values for characteristic attributes. The group of variants derived from a particular core product make up a so called *product line*³.

Product variants result from product architectures supporting variety. We will discover different product architectures in more detail in Section 2.3.3.3, "Modularity". For now, it's useful to distinguish two types of variety (cp. [Piller2003a, p. 223], [Anderson1996, p. 45], [Child1991, p. 55], [Hildebrand1997, p. 75]):

External variety. Refers to the number of variants perceived by the customer. Variant producers and customizers usually try to maximize the degree of external variety.

Internal variety. Refers to the number of variants that the manufacturing process and all internal operations are faced with. The degree of internal variety dictates the number of different tasks performed by the company and, thus, ultimately determines the complexity that the manufacturer needs to handle. Consequently, variant producers and customizers try to minimize the degree of internal variety.

Variant Management

The discipline of development, design and modeling of a product range, along with its variants, is called *variant management*⁴. Companies introduce product variants to better meet customer requirements, that is, to increase the benefit that a customer gains from a product. In terms of classical variant management strategies (so called *variant series production*), the users' needs are usually determined through market research and the results are incorporated into the manufacturer's product structure by clustering the product range so that as much customers' needs as possible can be matched. The variability of the product range is thus pre-determined and entirely designed during product development without further customer involvement. This allows stable production, sales and "one-way" communication processes: once the product structure has been settled, the procedures and processes can be rolled out throughout the company steadily.

Companies implementing variant management are required to deal with an increased (inner) complexity: adding variety to the product range introduces challenges on every stage of the product's lifecycle. During construction, product parts must be designed to be variable and compatibility across several modules in different layouts. Furthermore, the production processes need to be adapted to allow creating several variants, which not only requires multiple versions of design and production specifications (work lists, bill-of-materials), but also

³See http://www.fml.mw.tum.de/fml/index.php?Set_ID=320&letter=V&b_id=3742337B-4131-4535-322D-353443452D34, last accessed July 29th, 2012.

⁴See http://www.fml.mw.tum.de/fml/index.php?Set_ID=320&letter=V&b_id=3742337B-4131-4535-322D-353443452D34, last accessed July 29th, 2012.

demands a deeper understanding of the company's product range from those employees that assemble the variants. Not only (pre-/)production activities need to deal with an increased complexity through variant management: in post-production, sales and after-sales activities variability needs to be addressed, too. This includes handling multiple product versions in quality assurance processes, communication of variants in marketing (including retailers), creation of variant-aware product documentation and training materials as well as coping with a larger diversity of spare parts in service activities. Of course, all these additional tasks generated from variant management raise development, production and transaction costs.

From a customer perspective, a diversified product range has both advantages and disadvantages. On the one hand, a customer *can choose* the product that best matches his needs from multiple alternatives. On the other hand, he *has to match* his needs to the product range himself, which often requires more knowledge about the product structure in order to find the best choice. If the supplier offers too many variants, the selection process consumes much time and effort, often leading to frustration and customer refusal. Conversely, if the supplier offers too few variants, the customer needs to make concessions, in case the product doesn't match his necessities optimally, which can result in negative buying experiences and product rejection after all (cp. [Piller2001]). Another disadvantage in this case may be the extended price charged by the supplier that result from product variability and its associated complexity. It's the task of variant management to balance between external and internal variance in order to realize an optimal, cost-benefit oriented product range.

As mentioned earlier, companies employ several modularization and standardization techniques in order to reduce complexity generated by product variety. However, Lindemann et al. [Lindemann2006a, p. 8] argue that two fundamental problems of variant diversity are still neglected:

High efforts for development of complete variant spectrums. Often enterprises spend huge amounts of time and resources to develop and produce complete variant spectrums although only 5% to 15% of variants gets frequently sold after all. About 80% of the variants are rarely shipped with a rate less than 1%. Consequently, a high amount of complexity induced by product variants is created for nothing.

Non-optimal customer needs satisfaction. Customer needs are often not met adequately by simply introducing numerous amounts of product variants. Thus, the induced complexity by product variety not necessarily results in increased customer benefit.

2.1.3. Customized Products and Product Configurations

As described in the previous section, there are several reasons for the emergence of variants. Variants defined exclusively by the manufacturer are called *manufacturer specific variants*, while *customer specific variants* result from requirements defined by a particular customer [Holthöfer2001, p. 5]. Scheer [Scheer2006] differentiates three types of products in relation to the degree of customer involvement (cp. [Krug2010, p. 9]):

Supplier-oriented products. The customer has no direct influence on the design of the product and cannot customize it in any way.

Customer-centric products. While the customer can adapt the product to his specific needs, the scope of customization options is restricted.

Customer-oriented products. These kinds of products can be freely customized to fulfill customer requirements.

In general *customization* refers to the ability of adapting a product to individual customers' needs⁵. *Customized products* describe bundles of physical products and services that include

⁵From <http://de.wikipedia.org/wiki/Customizing>, last accessed July 29th, 2012.

both standard and individualized (customized) components [Lindemann2003]. Adaption in the context of product customization means the modification of parameters, structures and behaviors of a product so that the products' characteristics comply with the individual preferences of a customer that manifest themselves in specific customer as well as usage attributes [Piller2001] [Pulm2004].

2.1.3.1. Customizable Areas

Especially for consumer products, a common classification of those product attributes that are considerably influenced by customer preferences (also known as general customization options, cp. [Piller2003b]) are:

- functionality
- fit (form, layout, dimensions)
- design (style)

While these categories adequately support the characterization of rather simple consumer products (e.g., t-shirts), for complex products or capital goods (e.g., cars, machineries etc.) a more detailed categorization is reasonable. Lindemann et al. introduce the following *customizable areas of a product* that can be identified during product structure development (cp. [Lindemann2006a]):

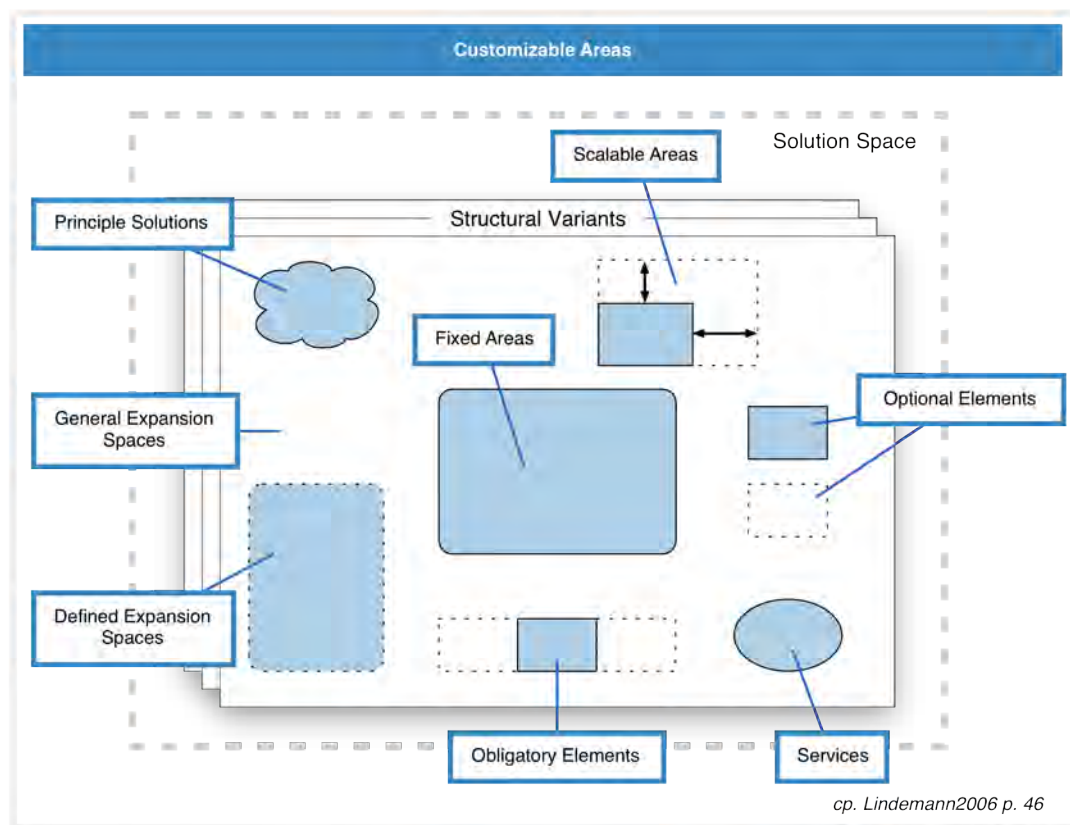


Figure 2.1. Customizable Areas

The identified section in Figure 2.1, "Customizable Areas" are:

Fixed areas. Unmodifiable core of a product structure that implements the basic functionality and structure provided by the product. Modifications of these areas would result in extensive structural and functional changes of the overall product, which cannot be implemented efficiently within a customer-specific manufacturing process.

Beyond the fixed area, Lindemann et al. describe the following variable areas.

Obligatory and optional alternatives. With the help of obligatory ("must-have") and optional ("can-have") alternatives, product customization can be achieved through selection of components similar to the selection process in variant series production. While *obligatory elements* describe essential, indispensable items of the product structure that must be selected by the customer in one or the other specificity (e.g., the engine of a vehicle), *optional alternatives* describe structural elements that may be added depending on the customer demand (e.g., additional equipment such as a seat heating).

Scalable areas. With scalable areas customizable elements can be modeled that can be freely modified by the customer within predefined constraints and rules such as performance parameters or dimensions. The value space of scalable areas is continuous.

Principle solutions. Principle solutions designate areas whose customization capabilities base on a predefined mechanism of action, that is, the customer can choose between different concrete alternatives that all implement the same solution principle (e.g., different types of switches, cover plates or functional elements such as digital vs. analog displays). Opposed to scalable areas, for principle solutions no mathematical rule that constraints the alternatives exists. Predefined solution principles significantly reduce the time and effort required during product adaption.

Services. Another, well established approach, to complement physical goods is the addition of services (e.g., insurance, extra warranty, etc.). These services can be often customized alike.

Defined and general expansion spaces. Finally, defined and general expansion spaces describe highly customizable areas of the product structure. *Defined expansion spaces* have been explicitly designed to support free composition using predefined design possibilities (e.g., the tools to perform the design process). While variabilities in this area are foreseen during construction, the exact specific customer requirements are not or not entirely known.

General areas of expansion are those areas that would in general allow customization. However, their adaption possibilities have not been considered during construction yet. If necessary, changes in general areas can be implemented in a customer specific manner, but require further planning and validation.

With the help of these areas of customization, various structural variants of products of any complexity can be designed, which is why these areas form fundamental requirements for our modeling approach introduced in Chapter 4, *Methodology and Conceptualization*. The set of all structural variants forms the complete *solution space* implemented by a product range.

2.1.3.2. Comparison: Custom-Made Products, Variant Series Products, and Customized Products

Depending on the production type and the degree of customer involvement (see Figure 2.4, "Customer Involvement Strategies" in Section 2.2.5, "Product Customization Strategies"), we can differentiate *custom-made products*, *variant series products* and *customized products*. Figure 2.2, "Comparison of Custom-Made Products, Variant Series Products, and Customized Products" lists the general characteristics of these types of products according to [Lindemann2006a, p. 10]:

Custom-Made Products	Variant Series Products	Customized Products
Developed "from scratch"	Selection of pre-produced modules	Configuration of pre-defined modules and customer specific adaption in defined customization areas
Products developed exactly according to customer specification	Often developed and produced ahead of customer inquiries	Product's structure is pre-developed
Customer-specific design in arbitrary product areas	Emergence of numerous variants that are never built	Adaption of the product to meet customer requirements is carried out on a per-order basis
Very low lot size, one-of-a-kind production	Non-transparent appearance of costs, cross-subsidization of "exotic" variants with low lot sizes	Production systems and processes are designed for flexibility
Very cost and time intensive	Increased diversity variant raises combinatorial product and process complexity	Prices and delivery times are comparable to those of series products Mainly structural product and process complexity needs to be mastered

Figure 2.2. Comparison of Custom-Made Products, Variant Series Products, and Customized Products

The main difference between the variant series production discussed above and production of customized products is the handling of complexity: while variant management strategies try to reduce the diversity of elements and combinations, in the case of individualized products, diversity shall not be lowered but instead be addressed with flexible product, process and management techniques. In contrast to entirely custom-made products, however, adaptations in customized products are restricted to certain areas of the product that are specifically optimized for modification [Lindemann2006a, p. 11].

2.1.3.3. Product Configurations

For the rest of this work we will particularly focus on customized products. The adaption of components to an individual customer's requirements is performed within a customer-specific development process that involves the collection and transformation of customer requirements into concrete customized components [Lindemann2006a, p. 9]. We refer to the act of collection and transformation of customer requirements into manufacturable, concrete product specifications as *product configuration process*.

The result of the product configuration process is a concrete **product configuration**. A product configuration can be defined as a product specification, that consists of a set of interconnected components with specific attributes, which is valid in regard to certain constraints. These constraints ensure that the configured product remains manufacturable in both a physically and an economically profitable way. In terms of manufacturing, a product configuration corresponds to a concrete variant of the product, however, we use the term product configuration to signify such a variant that has been specified within a customization process.

Product configurations are created by software tools that are called **product configurators** (configuration systems) which are the main subject of this work. Throughout the customer-integrated development process and in particular, the interactive selling process, product configurators form an essential tool used for the communication and specification of customer requirements (see Section 2.3.5.3, "The Interaction Process of Product Customization"). Piller [Piller1998, p. 9] describes product configurators as design tools that align cus-

tomer needs with the capabilities of the supplier. Without these tools, users wouldn't find suitable solutions for their specific problems due to the enormous complexity that originates in the diversity of the product range and the large number of feasible combinations (see the advantages and disadvantages of variants mentioned in Section 2.1.2, "Variant Management"). Product configurators help customers to quickly find exactly that combination of components that provides the highest possible benefit for their particular needs. In Chapter 3, *Configurators* we will take a detailed look on these specification tools.

Having now introduced the notion of *products* and *customizable products* in particular, we will take a look at *production processes* employed to build the same.

2.2. From Classical Production to Product Customization

The term *production* describes the process of combining and transforming production factors in order to manufacture goods. The results of this process is called *product* (see Section 2.1, "From Products to Product Configurations") [Domschke2005].

"Classical" production processes apply to the creation of regular, non-customizable goods. As mentioned in Chapter 1, *Introduction*, these production processes have a long history and have been evolved steadily. Until now, they're quite wide-spread and applied in all kinds of business areas. Knowing them helps to understand how companies (particularly in the manufacturing industry) perform their daily business. This, in turn, is a necessary requirement for understanding their difficulties, demands and potential motivations as well as limitations for moving into product customization.

In the following, we will examine the "classical" production processes (see [Krug2010, pp. 13]) with respect to their applicability in product customization scenarios, which will be described in more detail afterwards. We will also take a look on mass customization, a dedicated form of product customization and finally compare these strategies with each other.

Figure 2.3, "Overview of Common Production Processes" provides an overview of the different production processes and highlights important relationships.

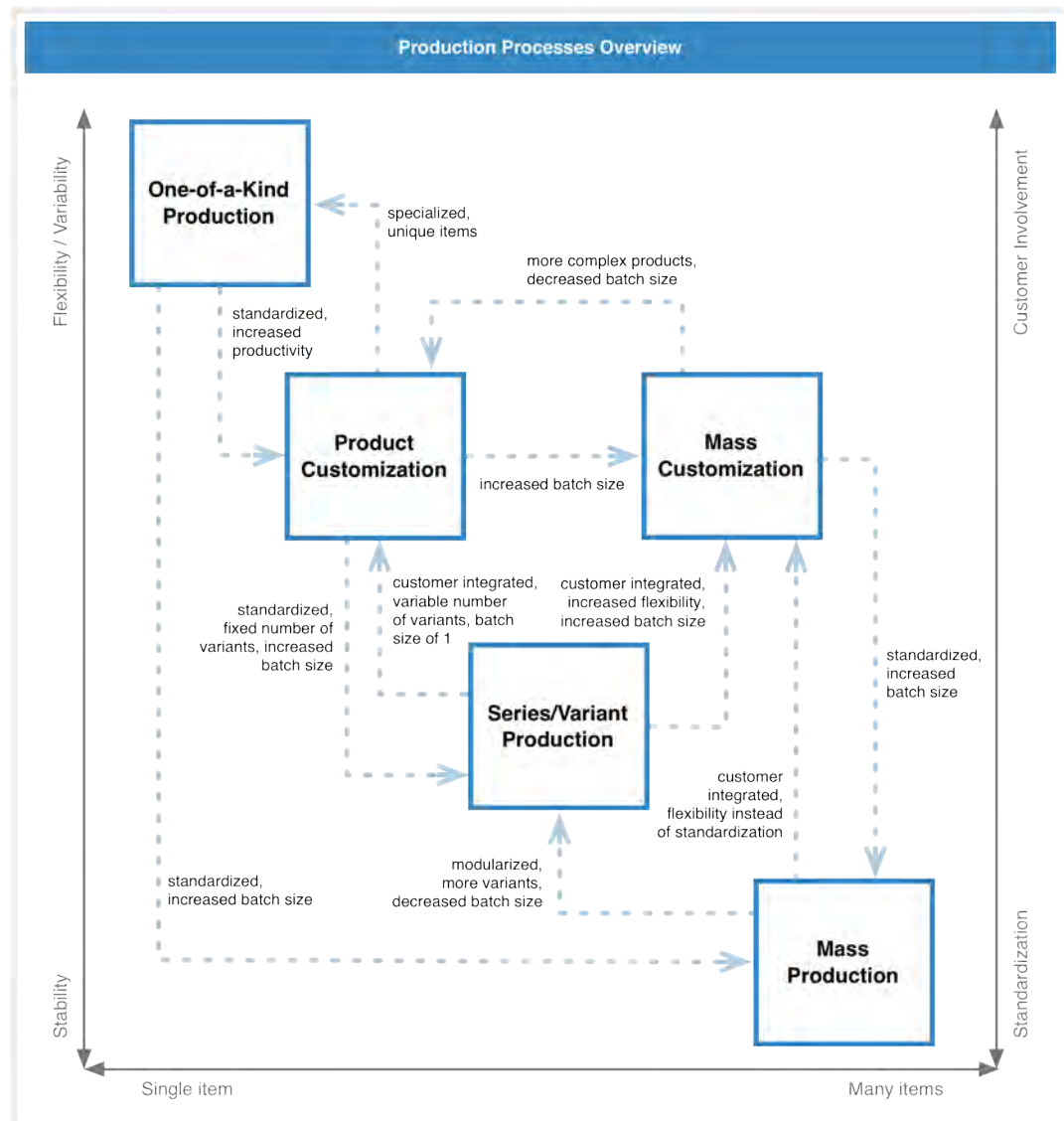


Figure 2.3. Overview of Common Production Processes

2.2.1. Per-Order/One-of-a-Kind Production

In *one-of-a-kind production* processes (also called *per-order* or *job production*) a single, unique item is manufactured. The produced item is not manufactured for the anonymous market, but instead for an individual customer on a per-order basis⁶. This allows the supplier to precisely react on the customer's desires and to fulfill non-standard requirements, which requires flexible production systems.

According to Krug [Krug2010], the one-of-a-kind production type cannot be supported by product configurators adequately due to the high degree of individualization. Although product configurators are equally applicable to complex products (see [Kratochvíl2005]), they're not intended to specify unique products. Furthermore, disadvantages of one-of-a-kind production techniques such as high production costs per item and low degree of automation make this kind of production less suitable for configuration. Per-order production is used, for instance, in the shipbuilding industry.

⁶See <http://de.wikipedia.org/wiki/Einzelfertigung>, last accessed July 29th, 2012.

2.2.2. Series Production

Series production (also referred to as *batch production* or *charge production*) target the manufacturing of multiple items of different lot sizes. It is characterized by the parallel or sequential, uninterrupted production of multiple, more or less uniform items (a series, batch) on one and the same production line. It's a production type with an either small or large, nevertheless, limited number of repetitions. Once the production of a single series has been finished, the production of the subsequent series starts⁷.

In series production only minor changes of materials without changes in the composition of components themselves are feasible. The per-item costs are significantly lower compared to one-of-a-kind productions as a result of the increased capacity in combination with optimized production plants (fixed cost degression). Though, they are still higher than those in mass production environments, due to the costs required to switch between series (process changeover costs).

Although the degree of automation is on a mean level, the flexibility of series production is very limited [Abels2004], which is why Krug argues that series production is not appropriate for product customization purposes [Krug2010, pp. 13]. Examples for this kind of production type can be found in supplier companies for the automobile industry, such as tire producers.

2.2.3. Variant Production

While in series production solely uniform items are produced in a single batch, in *variant production* items with slightly altered functionality, geometry, processing or design can be manufactured in the same batch⁸. Hence, variant production targets the production of a larger diversity of items.

In classical variant production scenarios, the customer is not involved during the manufacturing process. Instead he chooses and purchases the product from either the supplier directly or one of its retailers. Examples of variant production include, for instance, the production of t-shirts in different variants.

As mentioned by Krug [Krug2010, pp. 13], the applicability of variant production for product customization purposes depends on the point in time when the final composition of the variant happens (see Section 2.3.3.4, "Order Fulfillment Strategies"). In case the composition is performed by the customer *prior manufacturing*, the production process is considered product customization compatible. Here, true (*variant*) *configuration* is applied. The customer *is* involved in the manufacturing process before the product is assembled. Otherwise, if the customer can merely choose a specific variant from a set of already assembled items *after manufacturing*, Krug does not consider the manufacturing process product customization compatible. Here, the customer *is not* involved in the manufacturing process before the product is assembled. The latter describes (*variant*) *selection* rather than true product configuration.

Strictly speaking, *variant selection* cannot be considered a discipline of *product customization*, which requires the customer being involved in the manufacturing process prior to product assembly. Nevertheless, we argue that beneath *configuration tasks* for the specification of to-be-produced items, even complex *selection tasks* for choosing an already produced item delivered from stock are valid use cases for *product configurators* as well.

⁷See <http://de.wikipedia.org/wiki/Serienfertigung>, last accessed July 29th, 2012.

⁸See <http://de.wikipedia.org/wiki/Sortenfertigung>, last accessed July 29th, 2012.

2.2.4. Mass Production

The highest specificity of production concepts is provided by *mass production*, where uniform items are manufactured in very large lot sizes. Thereby, a fixed production process is continuously repeated on preset production lines. Mass production is characterized by⁹:

Devison of labor. Increased efficiency and productivity through job specialization, particularly by the devison of executive and planning activities. However, monotonously and repetitively performed activities lead to decreased satisfaction and motivation of employees.

Standardization. The standardization of products allows unification of processes. This leads to stabilized, highly optimized work procedures.

Focus on production techniques. Detailed analysis and planning of applied production techniques in order to produce goods with optimal resource utilization, while maintaining constant quality.

Assembly line production. The utilization of an assembly line during production ensures a constant working speed, which results in highly decreased production durations per item and thus to enormous efficiency enhancements. Assembly lines were first introduced large-scale by Henry Ford in the early 20th century and enabled productivity increases around 500%. The extremely successful model "T" was this way produced in only 2 hours 35 minutes in total opposed to previously 12 hours 8 minutes.

Hierarchical organization. The demand on control and monitoring of production requires hierarchical structures with professional managers in order to guarantee optimal business performance. Issuing of detailed instructions along strict hierarchical structures goes back to Frederick Winslow Taylor's elaborations on "Scientific Management" (often referred to as "Taylorism") which was also introduced in the early 20th century.

Vertical integration. In order to ensure continuous operation of the assembly line, bottle-necks on both sides procurement and distribution are avoided by strong integration of vertical activities.

Low costs and prices. Economies of scale (benefits, resulting from the procurement and production of large amounts of items) and economies of scope (benefits, resulting from specialized production of uniform items) allow production at minimum costs, which in turns lower the prices of the resulting products.

The main goal of mass production "is to develop, manufacture, market and deliver goods and services at prices which are low enough to where nearly everyone is able to afford them." [Blecker2005, pp. 9]. Due to the high degree of automation, mass production does not only provide the highest possible efficiency, but also reduces the per-item costs to a minimum. On the other hand, mass production systems lack flexibility regarding individual customer needs, which is why this production type is not suitable for product customization. Examples for mass produced goods include sugar and fuel.

2.2.5. Product Customization Strategies

Blecker et al. define *product customization* as "producing a physical good or a service that is tailored to a particular customer's requirements" [Blecker2005, p. 11]. The goal of customization is to increase the value of a product perceived by the customer. Compared to a mass produced product, a customized product increasingly fulfills the need of the customer (cp. [Svensson2001, p. 1]).

⁹See <http://de.wikipedia.org/wiki/Massenfertigung>, last accessed July 29th, 2012.

From a business point of view, product customization can be considered a strategy that features differentiation on products rather than solely on prices. A differentiation strategy tries to distinguish the products of a company from those of competitors, in order to achieve competitive advantages (i.e. higher customer retention and lowered price consciousness) and to increase profits (i.e. higher attainable prices) through product uniqueness [Scheer2006, p. 7]. Companies pursuing product customization offer individualized products to their customers, often but not necessarily at premium prices (up to 10-15% more compared to mass produced goods).

In practice, customized products are manufactured in a continuum between *standardization* and *customization*. While standardization targets the cost efficient production (i.e. achieving a cost position through cost reductions, see Section 2.3.4.4, "Cost Reduction Through Mass Customization"), customization aims at fulfilling customer's desires in order to increase the customer perceived value (product differentiation). Through product differentiation, companies usually expect profit gains (see Section 2.3.4.3, "Profit Gain Through Product Customization") [Scheer2006, p. 12].

Prominently, customization strategies are characterized by the **integration of the customer** into the manufacturer's value chain. Depending on the point in time the customer is involved, one can dissect business strategies as follows:

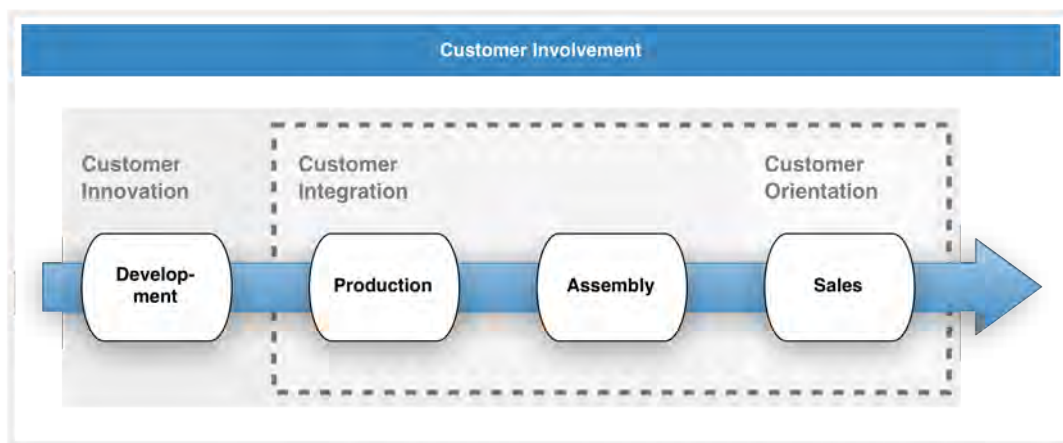


Figure 2.4. Customer Involvement Strategies

Customer Orientation. Fundamentally, the customer is put at the center of the company's strategic focus. The sales activities by the company are mainly driven by customer demands and dedicated customer relationship management (CRM) processes have been established. Though, the customer itself is not involved in the value-adding activities of the value chain (see also [Scheer2006, p. 8]).

Customer Integration. The customer is involved in the production and/or assembly processes of the value chain for the purpose of the creation of customer-tailored products. He is integrated into the process during the sales process in terms of the precise specification of his requirements on the individually built product. While the product architecture is flexible enough to support customized products, the overall solution space offered by the company is pre-defined and fixed.

Customer Innovation. The company involves the customer within its value chain to the greatest possible extend: by including him into product development activities, it is the customer who defines the products offered and sold by the company large scale. In this case, the solution space offered by the supplier is variable and determined in accordance with its consumers instead of being dictated by the management. This business strategy is referred to as "Open Innovation" [Reichwald2009].

In customer involved strategies and especially product customization scenarios, the relationship to the customer plays a central role in the company's business activities: according to Blecker et al. "an optimal understanding of customer needs is a necessary requirement for the success of the strategy." [Blecker2005, p. 3] The customers must be seen as partners in the value creation process that provide valuable input for development and production. Toffler coined the term "prosumers" for this kind of consumers [Toffler1984, p. 275]. The information gathered during sales and especially during the specification processes are continuously aggregated in order to establish a durable, personalized relationship to the customer (cp. [Piller2003a, pp. 208]). In the literature, this aspect is referred to as *learning relationships* [Piller2003a, p. 154]. In general, a strong customer-relationship is considered a key factor for establishing long-term competitive advantages, which is why detailed information about consumer behavior is more valuable than ever before.

2.2.6. Mass Customization

The concept of mass customization emerged in the late 1980s and the term was coined for the first time by Davis in his book "Future Perfect" in 1987 [Davis1987]. By end of 1992, the publication of Pine's book "Mass Customization: The New Frontier in Business Competition" [Pine1992] dramatically increased the popularity of the concept among managers and academics [Blecker2005, p. 40].

Mass customization signifies a business strategy, that aims at fulfilling individual customer needs with near mass production efficiency and prices [Pine1992]. While "plain" product customization as described in the previous section does not necessarily imply a focus on the costs perspective, mass customization does.

So, on the one hand the concept of mass customization describes a *hybrid business strategy*, that aims to fulfill both competitive strategies of product and price differentiation at the same time (against the position of Porter who argues that both goals cannot be reached simultaneously [Porter1980]) [Piller2003a]: "The challenge that manufacturing companies have to face is to provide individualized products and services by maintaining a high costs' efficiency." [Blecker2005, p. 2].

On the other hand mass customization can be seen as a *production concept* that tries to combine the benefits of mass production with a high degree on individuality regarding specific customer requirements [Reichwald2009]. It can be placed as independent production type between one-of-a-kind, variant and mass production: mass customization unifies their advantages by supplementing standardized processes in large parts of the value chain by customer-specific activities (see Section 2.3.2, "Product Customization Value Chain"). From a customer perspective, this lead to an individualized product portfolio [Piller2003a, p. 207].

A generally accepted definition of the term has been formulated by Piller: Mass customization is defined as

- the production of products and services for a relatively large market,
- where each item exactly meets the different needs of every single customer,
- at prices, that correspond to those of comparable standard goods, produced with mass production systems.

The information gained during the customization process serve the purpose of establishing durable, individual relationships to each customer. [Piller1998, p. 65]

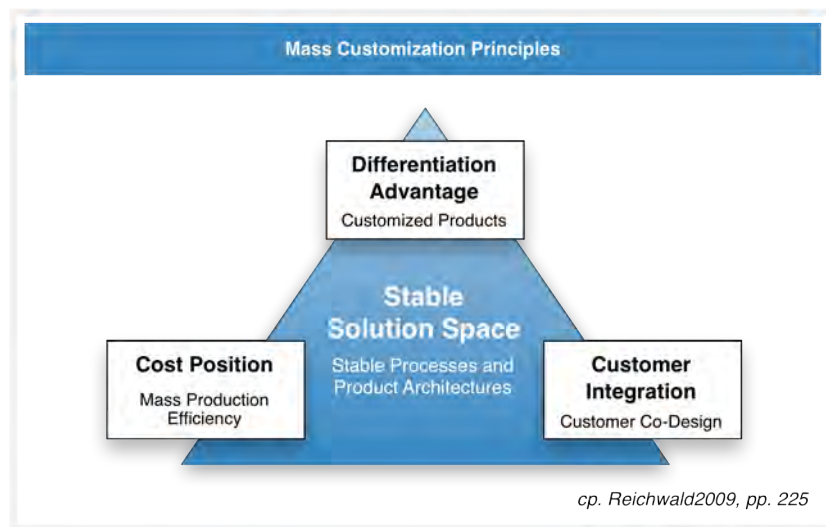


Figure 2.5. Main Principles of Mass Customization

The main principles of mass customization (see Figure 2.5, "Main Principles of Mass Customization") according to Reichwald and Piller are (cp. [Reichwald2009, pp. 225]):

Customer Integration. Central element of mass customization is the *integration of the customer* into the supplier's value chain in terms of co-design activities. The strong interaction between supplier and customer during this co-design process, that aims to specify the individual product within the stable solution space, is significant to mass customization.

Differentiation Advantage. By offering customized products, an enterprise pursuing mass customization can gain competitive advantages due to a *product based differentiation*. Customization possibilities related to an individual fit, functionality or design increase the customer recognized benefit of a product.

Cost Position. In general, the participation of the customer during the production process raises complexity and thus leads to increased transaction/production costs. However, the involvement of the customer and the information elicited during the specification process can be turned into new cost reduction potentials, so called "economies of integration" (see Section 2.3.4.2, "Mass Customization Cost-Efficiency Overview"). Also, consumers are often willing to purchase customized products at premium prices which corresponds to the increased benefit offered by those products. All in all, mass customization strategies can be implemented with near mass production efficiency but perform much better than one-of-a-kind productions.

Stable Solution Space. The key to accomplish the aforementioned potentials is a modular, flexible product architecture, that allows the customization of product parts within the boundaries of a *stable solution space*. In contrast to customer innovation approaches, the stable solution space is developed by the company without customer involvement. This allows to establish stable processes, which help to efficiently manufacture customized goods large-scale.

According to Blecker et al. "providing customers with individualized products at affordable prices is the main goal of mass customization." [Blecker2005, p.2] In summary, the efficiency of mass customization can be explained by the fact that mass customization restricts the customization abilities to the significant parts of the product, that make up the customer perceived value. These restrictions allow stable manufacturing processes that support the realization of economies of scale (as in mass production) and economies of scope (as in variant production) and simultaneously enable other economical advantages by utilizing in-depth knowledge about the companies' customers (cp. [Piller2003a, pp. 191], [Holthöfer2001, p.

10)). Furthermore, the combination of a modular product architecture, stable processes in production and last but not least the utilization of efficient, modern information and communication systems throughout the enterprise's value chain, allow "satisfying the customer's individual needs with near mass production efficiency" [Blecker2005, p. xxi].

In Section 2.3, "Product Customization Implementation" we will describe in detail, what it means for a company to move into product or mass customization practically. There, we will cover manufacturing, economical and marketing related aspects.

2.2.7. Strategy Comparison

Piller elaborates on the differences of mass customization and other production strategies, namely one-of-a-kind, variant production and mass production in [Piller2003a, pp. 207]. In our opinion, the arguments provided by Piller not only count for mass customization, but in general for product customization strategies, which is why we present a summary of Piller's findings in the following.

Compared to **one-of-a-kind production**, a product customization strategy is just *not* characterized by attributes typical for job shop productions, like:

- per-order, from scratch calculation of offerings
- high requirements on flexibility on all levels of production
- individual planning of all production processes
- specific creation of production documents (bill of materials, work plans, etc.)

Instead, a customization strategy builds on a pre-defined, stable yet flexible modular product architecture that allows the customer to freely configure the few, nevertheless essential components that make up the individual product value from a customer perspective. Piller speaks about "standardization of individualization" and as we will see in Section 2.3.4, "Economic Aspects", it is exactly this standardization, that enables the cost efficient performance of customization strategies. Customers can define their custom products within the limitations of the product architecture, which is designed to be efficiently manufacturable: not only the different modules can be produced at low costs, but also bill of materials, work and assembly plans as well as other specification documents are automatically created. Furthermore, product customization strategies target larger markets than those of one-of-a-kind productions.

Anonymous **variant production** aims to offer the largest possible diversity of variants in order to address customer demands. In contrast, product customization targets the exact fulfillment of customer expectations through the precise collection of customer requirements and the subsequent manufacturing of *customized* product variants (cp. [Holthöfer2001, p. 11]). In case of variant production the customer has to find the product variant, that best matches his needs on his own, while in case of product customization it is the company that provides the customer a product that *exactly* meets the customer's expectations. This way, possibly difficult, unsatisfactory decisions for one or against another variant is avoided. Additionally, in product customization scenarios, the final product variants are not produced prior to the reception of a customer order. Though, parts of it may well be pre-produced (we will discuss this in more detail in Section 2.3.3, "Manufacturing Aspects"). Moreover, the actual manufacturing or assembly of the product is postponed unless detailed knowledge about the customer's demand has been communicated. In terms of variant series production, the variants are pre-produced large scale, based on prospects determined through market research. The manufactured goods are then either distributed to retailers or put into stock. Although this allows quick order processing, putting items to stock holds additional risks of not selling all of them due to demand deviations.

Mass production strategies concentrate on highest production efficiency and the efficient handling of enterprise resources in terms of lean production. Fixed and stable processes

across the entire value chain are the main enablers for the cost-efficiency of mass production systems. Mass customization strategies, in turn, focus on the fast and all-embracing reaction on specific customer needs regarding production activities. In fact, the nature of mass customization is characterized by the efficient management of frequent changes, varying requirements and turbulent market conditions. According to Piller, mass customizers should turn their attention to the reduction of the complexity, that results from customer-individual production (cp. [Piller2003a, p. 208-210]).

We've now learned a lot about production processes in theory. In order to successfully establish a product customization strategy within an organization practically, different prerequisites need to be addressed by the company: ranging from a modular product architecture, over flexible production processes to the implementation efficient communication systems, aka product configurators, to elicit customer requirements. We will take a look at the most important aspects in the remainder of this chapter starting with a macro perspective of the value chain employed by mass customizers.

2.3. Product Customization Implementation

Customization strategies target the development of flexible product structures and order fulfillment processes optimized for *customer adaption*. These product structures and processes allow the fast and cost-efficient reaction on individual customer needs.

The implementation, respectively the shift into a product customization strategy, is a challenge for companies of any size. Important aspects for the successful implementation of product customization strategies are (cp. [Lindemann2006a, p. 11]):

- a step-wise development of the product and process range
- development of a product structure, that is optimized regarding customization possibilities
- a divided development process that comprises an upstream design phase followed by a customer-specific product adaption phase
- an individualized interaction with the customer during the buying process along with the integration of the customer into the order fulfillment process
- utilization of flexible product and process structures for manufacturing of customized components

2.3.1. Mass Customization Achievement

In the literature, various categorizations of strategies to implement product or mass customization practically, can be found ¹⁰. Piller explains a 2-dimensional categorization of strategies in [Piller2003a]¹¹:

Soft Customization. In terms of soft customization, adaption doesn't affect the manufacturing processes themselves. Consequently, these processes can be fully standardized. Instead, products contain *built-in customization options* and adaption is performed independent from the manufacturer, either by retailers or the consumers themselves. Due to the fact that customers (except in the case of service customization) are not required to interact directly with the producer, complexity within the manufacturer's value chain can be kept low.

In particular, there Piller mentions the following, concrete soft customization strategies:

¹⁰For an overview and discussion see [Blecker2005], pp. 12.

¹¹Translations partly taken from [Blecker2005], p. 17.

- **Self customization.** Development and production of products with built-in customization options, that can be adapted by the customers themselves.
- **Point-of-delivery customization.** Delivery of standardized products that are customized by retailers.
- **Service customization.** Offering of customized services complementing standardized products.

Hard Customization. In case of *hard customization*, variety results from customized activities executed during production. Customers are directly involved in the production process and influence the built up product deeply. Consequently, this approach requires a tighter customer-supplier interaction in order to avoid costly specification errors and misunderstandings, which may lead to unsatisfactory results. These could quickly damage the manufacturer's reputation sustainably.

Piller distinguishes the following hard customization strategies:

- **Customization-Standardization-Mix.** Either the first step (material processing) or the last step (assembly, finishing) of the value creation process are customized within the factory. All other steps are standardized.
- **Modular product architectures.** Customized goods are built from standardized, interconnected and among each other compatible modules.
- **Flexible customization.** Using flexible manufacturing systems for production of fully customized products at batch sizes of one without higher costs.

In practice, often more than one strategy, respectively a mixture of these, is compiled and implemented. In fact, a modular product architecture forms the basis of most product customization strategies pursued in the industry. It's frequently accompanied by service customizations.

Due to the fact that moving to product customization strategies may likely result in significantly more complex processes, manufactures often start by implementing a soft customization strategy and step-wise integrate hard customization.

Moreover, a company should carefully consider, whether necessary product, production and market conditions are met before even start to shift to product customization. Blecker et al. introduce a comprehensive framework covering the main conditions for achieving mass customization [Blecker2005, pp. 30]. Figure 2.6, "Overview of the Necessary Conditions for Achieving Mass Customization" provides an overview of the necessary prerequisites and conditions to be met by an enterprise in order to successfully pursue mass customization.

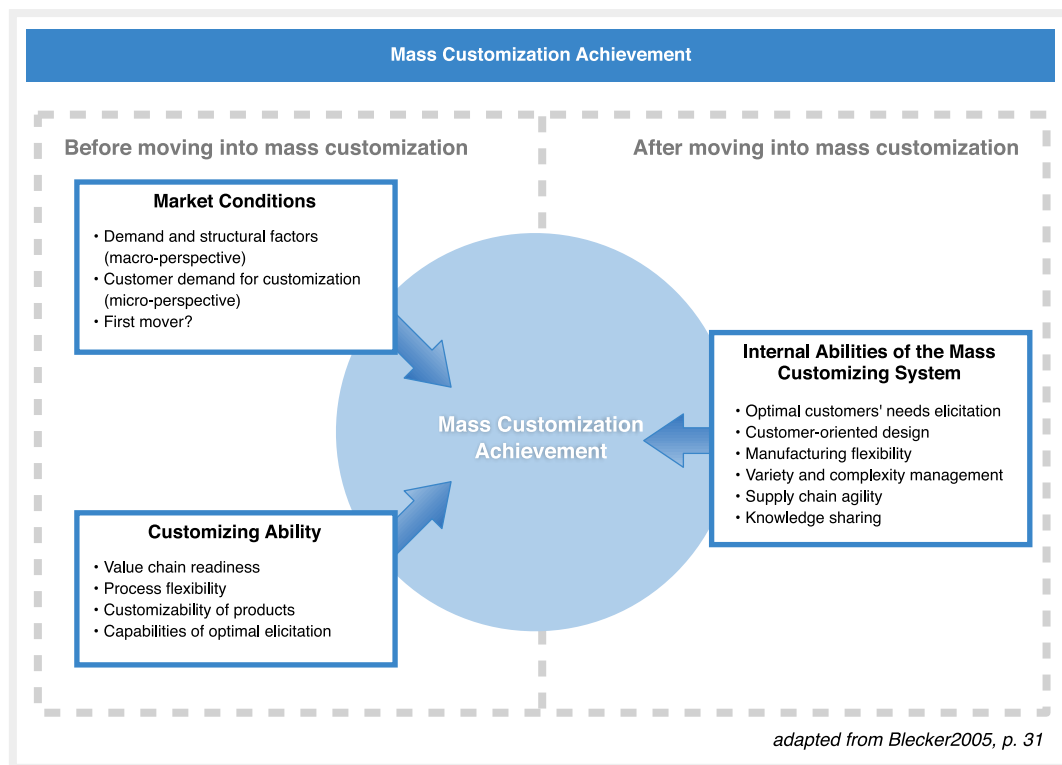


Figure 2.6. Overview of the Necessary Conditions for Achieving Mass Customization

Before moving into mass customization several **market conditions** have to be evaluated. We will discuss these in Section 2.3.4, “Economical Aspects”.

Related to the **customizing ability** the *readiness of the value chain* is concerned with the question whether the network including the company, its suppliers, distributors and retailers is able to deal with customization in general. We will discuss the value chain in detail in Section 2.3.2, “Product Customization Value Chain” and describe concrete strategies for integrating the customer in Section 2.3.3.4, “Order Fulfillment Strategies”. In order to offer a large diversity of product variants, the *production processes* need to be laid out flexibly, which will be discussed in more detail in Section 2.3.3.5, “Production Process Split” and Section 2.3.3.6, “Other Manufacturing Process Related Aspects”. Beyond the processes, also the *product range* itself needs to be ready for customization. Customizable products have already been discussed in Section 2.1.3, “Customized Products and Product Configurations” and we will further investigate that topic in Section 2.3.3.2, “Product Architecture” and Section 2.3.3.3, “Modularity”. *Capabilities of optimal needs elicitation* relate to the ability of a company to gather the requirements on a customized product from the consumer. This is the point, where product configurators are involved. Effectively, product configurators are the tools that actually drive the customer's requirements specification. We will discuss them in great detail in Chapter 3, *Configurators*.

Regarding the **internal abilities of a mass customizing system**, the topic of *variety and complexity management* has already been discussed in Section 2.1.2, “Variant Management”. *Supply chain agility* deals with the management of the “uncertainty that is triggered by unforeseen requirements” [Blecker2005, p. 40]. *Customer-oriented design* determines, whether a product is actually designed to suite customers' needs. We will cover that topic in Section 2.3.3.2, “Product Architecture”. *Manufacturing flexibility* will be mentioned in Section 2.3.3.6, “Other Manufacturing Process Related Aspects”. Finally, *knowledge sharing* describes the ability of an enterprise to quickly and efficiently transfer the knowledge about the customers' desires and preferences, gathered during the interaction phase, across the value chain.

In the next section, we will provide a global overview of the product customization value chain. We'll then go into manufacturing related details, economical aspects and finally discuss consequences for the marketing activities of a company establishing product customization. We'll, thereby, focus on hard customization strategies only.

2.3.2. Product Customization Value Chain

Product customization strategies are mainly characterized by a two-split value chain: the value-adding activities are divided into *standardized* (customer order neutral) and *individual* (customer order related) parts as depicted in Figure 2.7, "Value Chain of Product Customization Pursuing Enterprises".

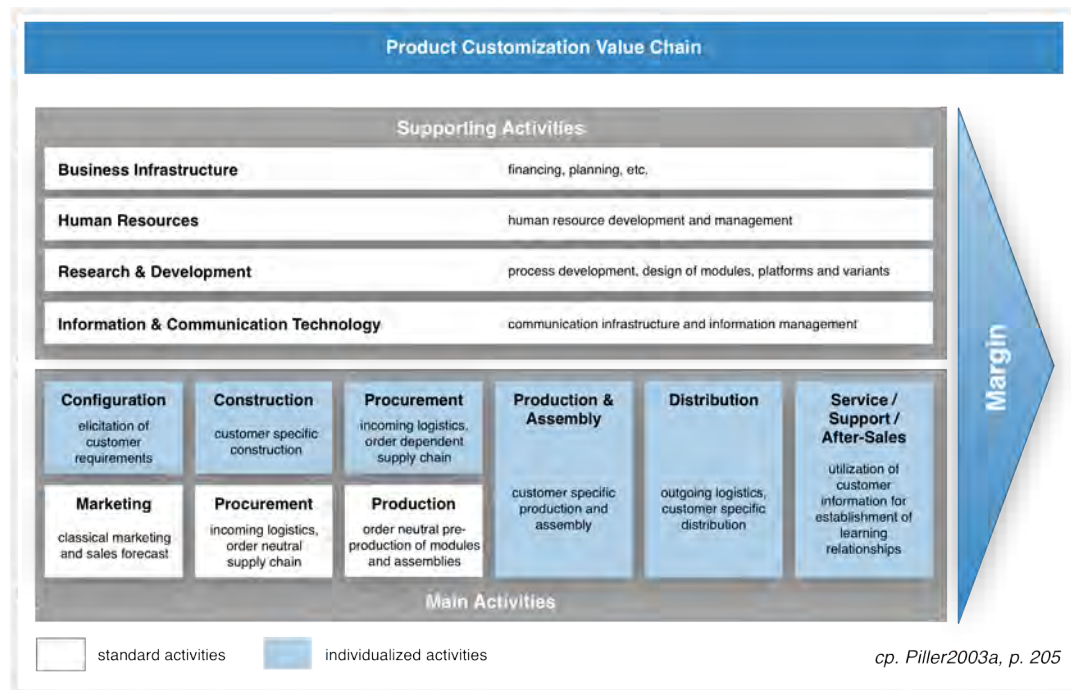


Figure 2.7. Value Chain of Product Customization Pursuing Enterprises

The presented figure taken from Piller is based on a diagram of the value chain introduced by Michael E. Porter in 1980 [Porter1980]. *Supporting activities* are those activities that indirectly affect the goods produced by the company. In contrast, *main activities* are the parts of the value chain that are directly concerned with good fabrication. *Margin* describes the profit gained through value creation, that is, the difference between outcome (revenue) and the input (resources) of the process. Each enterprise executes its own value chain. The interrelated value chains of different enterprises within the same branch is called *supply chain*¹².

As stated earlier, product customization strategies affect nearly all activities within an enterprise, including the **supporting activities**. From a management perspective (*business infrastructure*), a shift to product customization often implies a change of the entire product portfolio, with customizability being an outstanding feature of the product range. Not only marketing activities must communicate customization options as unique selling points (USPs). Also the companies' executives need to adjust business processes and management tasks in a much more agile manner, opposed to classical hierarchical organizational structures (relates to *human resources*).

Furthermore, significant changes in *research and development* (R&D) activities are required, in order to successfully pursue product customization: these activities do not only tackle

¹²See <http://de.wikipedia.org/wiki/Wertsch%C3%B6pfungskette>, last accessed July 29th, 2012.

standardized, monolithic products that can be designed and planned in their entirety. Instead R&D activities need to deal with modularized, flexible product architectures, that offer manifold customization options in order to support diversified, yet cost-efficient manufacturable product variants. The design of the so called *generic product architecture* (see Section 2.3.3.2, “Product Architecture”) involves the definition of modules with compatible interface, which enable customers to easily configure custom variants fulfilling their needs. Limited customization options, the careful consideration of using standardized components opposed to customizable ones and the use of standardized modules with built-in customization options can be seen as techniques to balance between the additional value offered by customizability and cost-efficient production. In terms of product customization, the goal of R&D departments is to design a product range, that exhibits low internal variety on the one hand, and which supports widely stable production processes on the other hand. Likewise, it is the goal of a product architecture to meet customers “ideal points” regarding most sales relevant product attributes (cp. [Piller2003a], pp. 270).

Related to *information and communication technology*, product customization is even more demanding. The design of the generic product architecture requires sophisticated computer aided design (CAD) tools. Flexible production systems are controlled with computer integrated manufacturing (CIM) systems. The integration of the customer into the value chain as well as the establishment of long-term customer relationships requires mature customer relationship management (CRM) tools. Finally, the customized sales process needs to be backed with intelligent computer added selling (CAS) systems, such as product configurators. Additionally, all these systems need to be well integrated with existing enterprise software, like enterprise resource planning (ERP) and product data management (PDM) platforms for efficient data exchange. We will take a closer look at the role of configurators as sales supporting tools and their role within the system landscape in Section 2.3.5, “Marketing Aspects” and Chapter 3, *Configurators*.

The **main activities** of the value chain in a product customization scenario can be divided into activities that require direct customer interaction (*individual activities*) and those that do not (*standard activities*). Depending on the exact stage of customer involvement (the so called *customer order decoupling point*, see Section 2.3.3.4, “Order Fulfillment Strategies”) the activities highlighted in the figure above are performed in higher or lower intensity. In almost all cases, in order to efficiently fabricate customized products and to shorten delivery times, the product architecture features standard parts, that are pre-produced to stock independently from customer orders. In this case classical marketing and sales forecast techniques are employed to prognosticate the demand on modules which are frequently used in customized products. Production specifications including construction drawings, bill of materials and work plans have already been finalized in previous R&D efforts. The sales predictions and production specifications form the basis in subsequent procurement and production processes related to the manufacturing of these “ready-made” modules. Any other activity of the value chain is triggered by specific customer orders and each activity is based on information supplied by the customer.

The elicitation of customer requirements is subject to the **configuration step**, which is an integral part of the entire sales process (we will take a closer look at the sales process in Section 2.3.5, “Marketing Aspects”). During this very first step, the product configuration system is utilized to precisely collect the customer's desires and to design the customized product exactly matching the customer's needs. While any subsequent activity involves customer specific information, the interaction with and communication to the customer is limited to the configuration phase (except after-sales activities).

The result of the configuration phase is a fixed and stable product configuration, which does not require any further negotiation with the customer but instead can be considered a complete order specification. The fact that the product configuration remains stable for the rest of the production process is a key factor for executing the customized value chain with high performance and for optimizations in production activities. Additionally, it's one of the main differences to the significantly less efficient one-of-a-kind production processes, where spec-

ification changes are often allowed during execution of the value chain (so called *change requests*).

While product configurations can (most of the time) be automatically transformed into production specifications, in rare cases, customized components require a customer-tailored *construction step*, which usually drastically delays delivery times. After the production specification phase has been completed, a customer order triggers the *procurement process*, but just in case the final product cannot be assembled with modules from stock or modules produced in-house.

The production of the latter ones and the final assembly of all modules according to the product specification is subject to the *production and assembly step*. The resulting product is shipped to the customer through the customer selected *distribution channel*. The last step of the value chain, the *service and after-sales activities* can be considered long-term product support activities. After delivery of the customized product, the customer information collected during the configuration phase is used to establish a long-term relationship to the customer including subsequent personalized product support offerings and individualized services, such as frequent maintenance and repair offerings.

2.3.3. Manufacturing Aspects

2.3.3.1. Product and Production Related Prerequisites

In order to successfully pursue product customization the enterprise must address some product and production related challenges. On the one hand, the company's product range must be designed for customization, that is, the *product architecture* has to provide customization options that support building custom product variants. Due to the fact that a flexible product architecture, which allows diverse variations to be configured, leads to increasingly complex production and management processes, techniques to lower complexity induced by product customization have to be established. The most important technique in this regard, from a manufacturing perspective, is *product modularity*.

On the other hand, the company must adjust its production processes to cope with customer involvement: the company must decide the point of the value chain when the customer is to be involved in the creation of the customized product. In other words, the company must choose from one of the existing *order fulfillment strategies*. Finally, another important milestone towards mass customization is to *find the optimal degree of pre-produced versus on demand created goods*.

We will cover these issues one by one in the following.

2.3.3.2. Product Architecture

The (*generic*) *product architecture* is an abstract description of the company's product range. According to Blecker et al. "the main purpose of a product architecture is to define the product building blocks by specifying what they do and how they interface with each other" [Blecker2005, p. 164]. This definition can even be extended when product architectures are described in terms of product models. Schwarze defines a product model as an abstract representation or description of a product or family of products, describing the structure and all facts, objects, concepts and properties that are relevant in any lifecycle phase of the same [Schwarze1996, p. 33]. Thereby, a product model may encompass multiple perspectives [Andreasen1997], [Hvam2008, p. 36]:

Product structure. The product structure considers what the product consists of, that is, it describes how products are built up and which parts they consist of (*decomposition structure*). Furthermore, the structural perspective comprises *relationships* and possibly *constraints* between specific parts of the product.

Product functions and properties. The functional perspective describes what products can do (their *function*) and characterize them according to *properties* such as weight, dimensions, surface, strength, price and so on. *Solution principles* can be modeled in order to fulfill certain product functions (see also Section 2.1.3.1, “Customizable Areas”).

Product life cycle properties. From the moment a product originates as a need recognized by the consumer, a product goes through a number of phases such as design, production, assembly, transportation, installation, use, service, disposition and recycling. For each of these phases, specific information may evolve that must be accessible at the point the product enters the particular phase, which is why these *life cycle related properties* can be considered as part of the product model.

Variation and family structures. In order to offer variety and to allow configuring different variants of a product, the product model must contain customization options. In Section 2.1.3.1, “Customizable Areas” we already discussed different adaption possibilities. In general, we base customizability on the fact that the product architecture is *modular* and variable components can be selected as is (*selection*), altered in terms of parameter changes (*customization/configuration*) or constructed from scratch (*construction*).

As stated earlier, during research and development activities, the product architecture is developed with respect to two influencing factors: customer requirements and efficient manufacturability. Companies strive for offering a suitable variety (high degree of external variety) while maintaining a suitable commonality (low degree of internal variety, see Section 2.1.2, “Variants”) in order to keep the complexity of internal processes minimal (cp. [Hvam2008, p. 143]). Again, limited customization options, though strongly aligned with the customers' demands, are a key factor of mass customization's efficiency.

Hvam et al. elaborate on the analysis of a company's product range and the development of the product architecture (see [Hvam2008, p. 139]). They base their procedure on three important areas of theory (cp. [Hvam2008, p. 147]):

System theory. As the basis for modeling the components of the product architecture, system theory can be used. System theory is based on the distinction between *function* (what the product *can*) and *structure* (what the product *is*). Furthermore, system theory considers the interaction of a system with its environment, that is, it considers the *input* consumed and the *output* produced by the system.

Object-oriented modeling. The object-oriented modeling methodology offers lots of valuable mechanisms for precisely describing product architectures. With the help of classes and objects, attributes and methods, generalization-specialization relations and associations between objects and classes entire product families can be designed in an extremely compact manner. Object-oriented analysis not only considers the *part-of structure* and the *kind-of structure* of product components, but also considers *behavior* in terms of method definitions.

Multi-structuring. Techniques of multi-structuring can be utilized the model relevant product aspects from different perspectives. Hvam et al. consider at least the three viewpoints practical (cp. [Hvam2008, p. 150]):

- **Customer view.** Focusses on the products' functions, properties and structures from a customer point of view.
- **Engineering view.** Focusses on the relationship between a products' functions and structures in terms of solution principles from an engineering perspective.
- **Production view.** Focusses on a product's detailed structure and its life-cycle properties related to production and assembly.

The product range analysis and the development of a generic product architecture helps to identify a product program's readiness for customization. During analysis activities within

the company, often cleanup and standardization efforts take place, in case the product model cannot be described easily or when the resulting model is too complex for the company or the customer to understand. Also, modularization possibilities in previously monolithic products can be identified during attempts to describe products and their components.

2.3.3.3. Modularity

"Product modularity enables the manufacturing of a large number of product configurations by simultaneously taking the advantage of the economies of scale and scope. [...] [Modularity] enables not only the ability to put the 'mass' in mass customization, but also to configure the products according to the customer's requirements. [...] Product modularity is considered to be a necessary requirement [for mass customization]." [Blecker2005, pp. 163]

In general, *modularization* refers to the decomposition of a complex object into separate parts, so called *modules*, with lower complexity. Nilles further characterizes product modules as follows¹³ (cp. [Nilles2002, p. 127]):

- A product module is a subsystem with lower complexity than the overall system of which the module is a part.
- A product module is a closed functional unit.
- A product module is a spatially closed unit.
- A product module has a well-defined and obvious interface.

While in the literature various types of modularization are described¹⁴, Hvam et al. provide a comprehensive set of main modularity types. They are summarized in Figure 2.8, "General Types of Modularity" [Hvam2008, p. 30-31]:

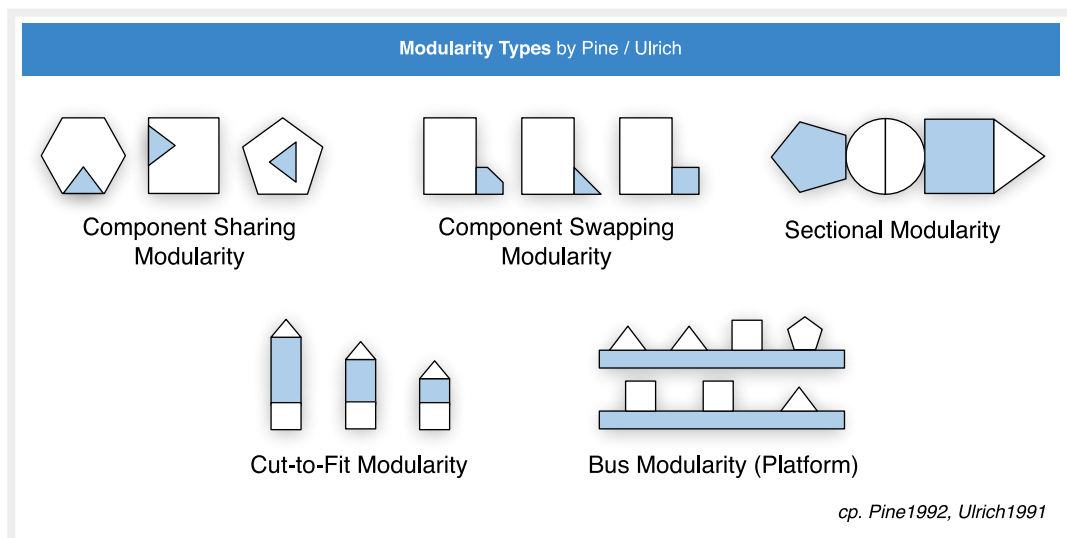


Figure 2.8. General Types of Modularity

Component sharing modularity. In case of *component sharing modularity*, a standardized module is shared across different product families. For instance, a car manufacturer may use one and the same car radio in various car families.

Component swapping modularity. Within a single product family, *component swapping modularity* is applied, if a component can be replaced by other component variants with the

¹³Cited from [Blecker2005, p. 164]

¹⁴See [Blecker2005, pp. 164] and [Kratovich2005, pp. 84] for overviews of various types of modularization.

same interface. For example, the cover of a mobile phone can often be replaced by another "themed" cover.

Sectional modularity. The *sectional modularity* type can be seen as the "Lego" brick modularity. Any components with matching interfaces can be attached to each other. A modular pipe system can be taken as an example for sectional modularity, where tubes and pipe switches can be freely combined as long as their endings are compatible.

Cut-to-fit modularity. *Cut-to-fit modularity* is applied, when the dimensions of parametric components can be adapted to suit the customer's requirements. Examples include the custom-fit clothings or customized windows with individual dimensions.

Bus modularity (also: platform modularity). Another popular modularity type is *bus modularity*, where different product variants can be induced by mounting various components with compatible interfaces to one and the same platform. Thereby, the number of attached components may vary. As an example, the mother board of a computer can be mentioned, which allows to connect different devices such as DVD or hard drives to it.

With these types of modularity, that are usually intermixed within a single architecture, a largely diverse, flexible and modular product range can be modeled.

Depending on the degree of similarity and the type of modularity, one can differentiate between the following architectures:

Model range. Products of a *model range* (type series) solely vary in size but otherwise share the same functionality, same production technique and the same materials.

Model kit. *Model kits* (building blocks) allow the creation of variants by combining modules with different forms or layouts but otherwise compatible interfaces [Kolb2012].

Platforms. *Platform* designs have a basic module in common and variability results from extension of the basic module with additional ones [Lindemann2006a, p. 44].

Individual Products. *Individual products* are entirely independent objects that do not share a common construction concept [Schönsleben2000]

The concepts of model ranges, model kits, platforms and in general modularization and standardization techniques are considered established strategies to cope with the increased complexity faced in variant management activities [Schuh2001].

Platform designs, can be further differentiated by their type of modularization (cp. [Piller2003a, p. 259-260])¹⁵:

Generic modularization. Composition of products with a *fixed number of standardized components*, that itself may vary in their characteristics. The core of these products builds the common platform.

Quantitative modularization. Also based on a common platform, product variants are composed by attaching a *varying number of standardized components* to it.

Custom modularization. Composition of products based on a platform with a *fixed or varying number of either standardized or customized components*, that is, build according to specific customer requirements (custom components).

Free modularization. Describes the composition of a *fixed or varying number of standardized or customized components*, which are *not based on a common platform* as the product's foundation.

¹⁵Note that Piller uses the terms *model kit* and *platform designs* equivalently.

Figure 2.9, “Types of Modularization for Platform Designs” illustrates these different platform architectures.

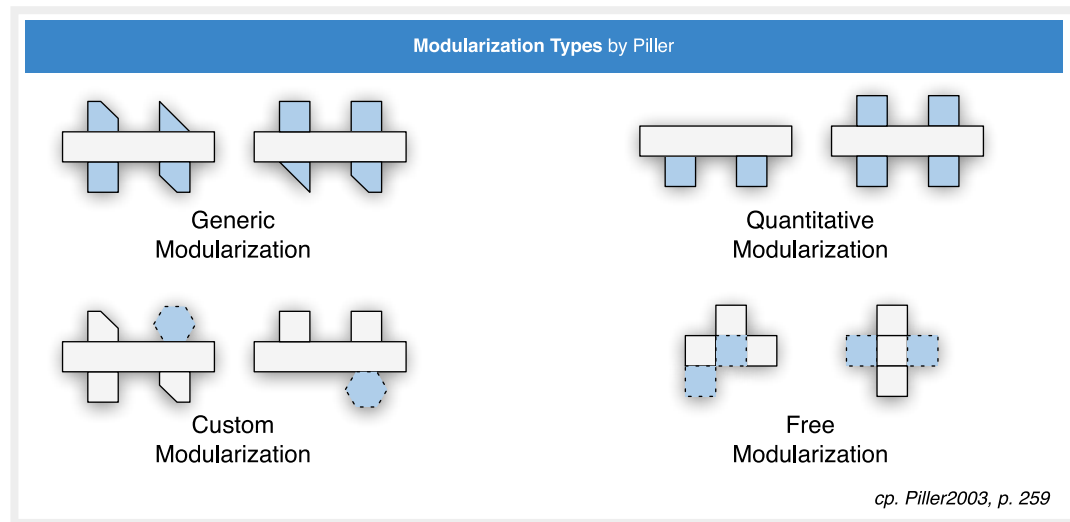


Figure 2.9. Types of Modularization for Platform Designs

The Role of Modularization

Modularization can be considered a fundamental concept for the efficient implementation of product customization. According to Blecker, "[...] [modularity] is a very relevant concept that enables the reduction of product complexity and the achievement of economies of scale, the economies of scope and the economies of substitution by simultaneously ensuring a high level of product variety." [Blecker2005, p. 5]

In this context, the **main benefits of modularization** related to mass customization are, that modules support building a large variety of products while, at the same time, the complexity of both the individual components and the production processes to fabricate customized products can be reduced. Modularization is beneficial across the entire product customization value chain: related to marketing and sales, the company can offer a wide variety of customizable products. In terms of research and development activities, product innovations can be accelerated by re-using large parts of the product and only advancing performance or sales relevant modules. In production, work can be distributed and decentralized. Modularization not only enables stabilized production processes but also shortens production lead times, since large parts of the production process can be parallelized. Also, modularization can lead to an increased product quality, due to the fact that components can be independently tested in isolation, early in the production process. If most modules of the product have been pre-produced (see Section 2.3.3.5, “Production Process Split”) and solely customized parts are manufactured upon customer order reception, the delivery times for the resulting product are strongly reduced. Products built up from closed modules not only simplify product assembly, but also support maintenance and repair activities during product use. Additionally, they allow customization even during the product's lifetime, e.g., components can be updated / replaced by newer ones in order to increase the product's performance (cp. [Blecker2005, p. 168-169], [Piller1998, pp. 194]).

Even though the advantages of modularization outweigh its disadvantages, some **limitations** can be relevant from certain points of view. First and foremost, the development of a modular product range can be cost-intensive compared to the development of integral systems. Additional costs incur for designing the carry-over of parts (initial design costs), for testing those modules in different variant scenarios (testing costs) and for searching components to be re-used (search costs, which can be considered negligible if appropriate software

systems are in use). Also management costs and required coordination efforts increase when the number of modules grows drastically. However, the simpler a modular product architecture is, the easier can competitors imitate the company's product design (cp. [Piller1998, p. 197]).

As stated by Blecker et al. modularity is just one of several system attributes that must be considered when designing product architectures and trade-offs between e.g., integrity, up-datability and modularity must be balanced [Blecker2005, p. 169].

2.3.3.4. Order Fulfillment Strategies

An essential decision when implementing product customization is the definition of the exact point in time when the customer is being integrated into the value chain. This point is referred to as *customer order decoupling point*. Basic customer involvement strategies have already been shown in Section 2.2.5, "Product Customization Strategies". Precisely, the following strategies of order fulfillment/customer integration can be differentiated [Reichwald2002]¹⁶:

Match-to-Order (MtO). Product customization performed by the sales department. Mapping of standard products to the requirements profile of a customer.

Bundle-to-Order (BtO). Product customization performed by the sales department. Composition of standard products to a "profit bundle" that corresponds to the customer's profile.

Assemble-to-Order (AtO). Product customization is performed during the final assembly of the product, which is carried out with standard components taken from stock.

Make-to-Order (also Fabricate-to-Order, FtO). Product customization applies to the production stage of the value chain. Components fabrication takes place on a per-order basis.

Develop-to-Order (also Engineer-to-Order, EtO). Product customization applies to the product development. Most extensive form of customization, merely used in capital good productions.

Figure 2.10, "Order Fulfillment Strategies" illustrates the different order fulfillment strategies¹⁷.

¹⁶Translated from [Piller2003a, p. 248].

¹⁷In the style of [Hvam2008, p. 26].

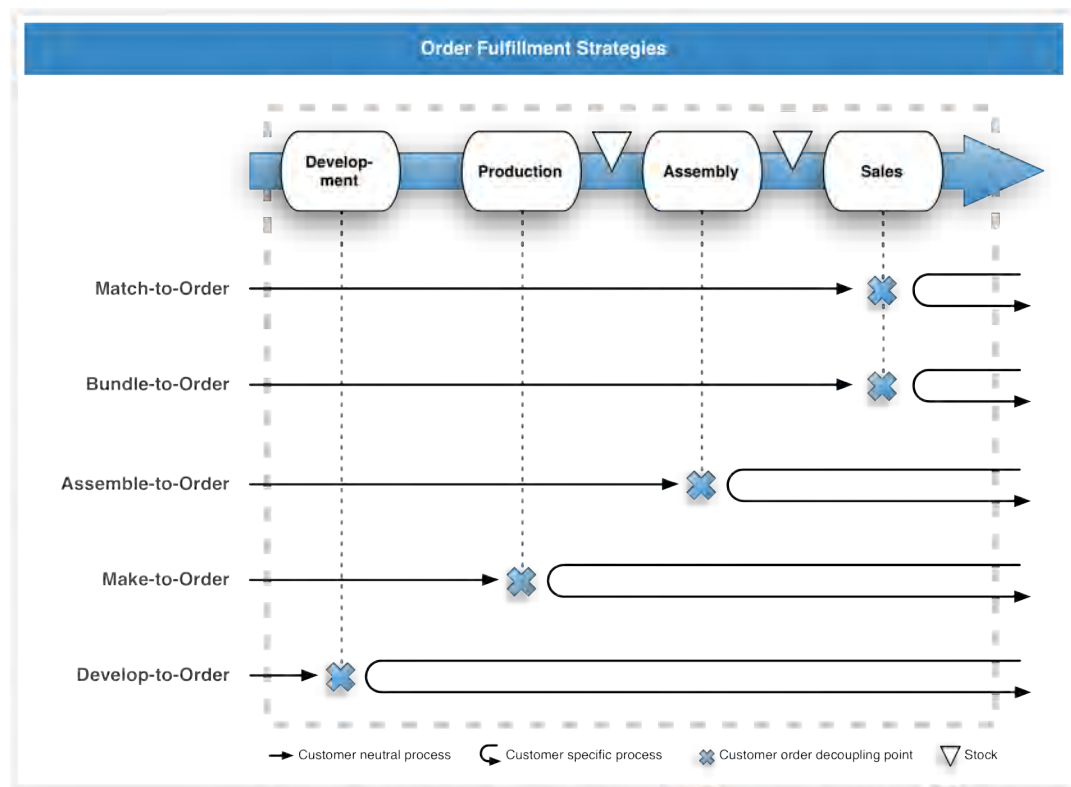


Figure 2.10. Order Fulfillment Strategies

In case of *Match-to-Order* and *Bundle-to-Order*, end products are put to stock within the retailer's facilities, before they're delivered to the customer. In an *Assemble-to-Order* scenario, components or intermediate products are stored within the producer's facilities. Finally, in case of *Make-to-Order* and *Develop-to-Order* no storage is involved at all, since components are continuously manufactured and assembled to the final product, which is subsequently shipped to the customer without interruption (cp. [Krug2010, p. 12]).

2.3.3.5. Production Process Split

In essence, customized products result from customer specific assembly of *standard modules* and *customized modules* and it is the customer order decoupling point that splits the value chain into a customer order neutral part and a customer specific part.

Within the *customer order neutral part* standard modules, that are used in the majority of product variants, are developed and pre-produced. In this part, processes can be highly automated and stabilized, which allows cost-efficient manufacturing of large parts of the product. The production of custom modules and the customer specific assembly of the final product are subject to the *customer specific part* of the value chain (see Section 2.3.2, "Product Customization Value Chain"). As fabrication of customized modules ultimately depends on the customer requirements elicited during the configuration phase, the customer specific part is not triggered before customer order reception. [Piller2003a, pp. 230]

According to Piller, the **bisection of the value chain** is an essential prerequisite for the successful reduction of the overall planning and control complexity induced by customer specific production. The goal of the distinction between customer order neutral and customer specific part is to identify all production steps, that can be executed independently from customer orders and to plan them as such. These steps can be scheduled with much higher flexibility and thus greatly reduce the production planning complexity as a whole [Piller2003a, p. 230-231]. The planning task itself can be split into independent sub systems, that can be represented by two control cycles [Piller2003a, p. 231]:

Customer order neutral regulator circuit. This circuit controls the production orders for parts, modules and variants independently of customer orders. Thus, within this circuit, a produced item cannot yet be related to a specific customer order.

Customer order dependent regulator circuit. Triggers production orders immediately upon reception of a customer order. All subsequent steps are thus directly related to one specific customer order.

Again, the customer order decoupling point delimits both regulator circuits. The decision when the one circuit ends and when the other one starts therefore corresponds to the decision on the *optimal degree of pre-production*, which should be carefully considered. Essentially, it's the decision about the optimal ratio between standardized and customized activity performed by a mass customizer. Piller describes two alternative scenarios for manufacturing, depicted in Figure 2.11, "Make-to-Stock and Make-to-Order Manufacturing Scenarios" [Piller2003a, p. 231-233]:

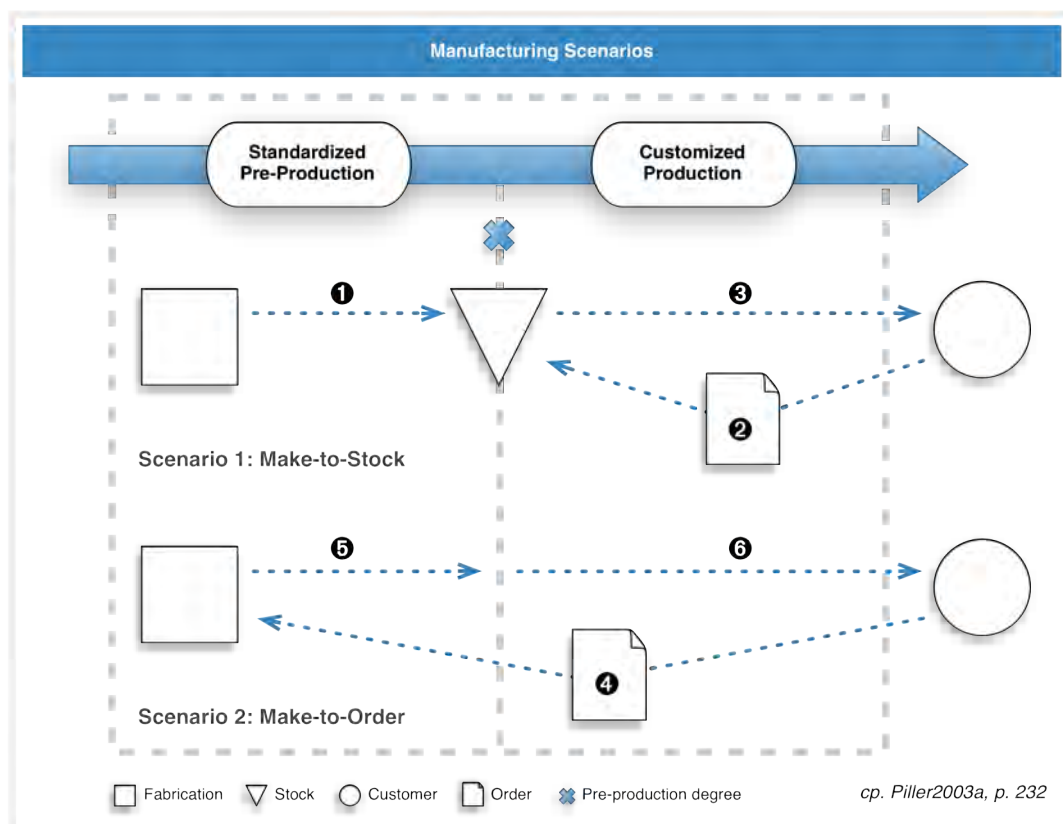


Figure 2.11. Make-to-Stock and Make-to-Order Manufacturing Scenarios

Scenario 1: Make-to-Stock (MTS). In the first scenario, the product components are pre-produced in a customer neutral manner and placed into stock ❶. Upon customer order reception ❷, the pre-produced components are retrieved from stock again and further manipulated and/or directly assembled to the final product according to the customer's specification before being handed over to the customer ❸.

Depending on the degree of pre-produced components, efficient, mass production like processes and machinery can be used for the manufacturing of standardized modules. While stocking increases both inventory costs and the risk that the stored modules never get sold, the delivery time of the final product can be reduced significantly.

Scenario 2: Make-to-Order (MTO). In the second scenario, the pre-production of components does not start before, but is triggered immediately after a customer order has been

received ④. The manufactured components are directly assembled to the final customized products ⑤ without putting them into storage in between. They're directly delivered to the customer ⑥.

With the help of this so called *customer pull strategy*, inventory costs and stocking risks as well as wastage is avoided. Standardized, repetitive and stable production steps, that are defined in a customer neutral way, strongly reduce the planning complexity and allow the efficient manufacturing of the product's parts. Nevertheless, sufficient capacities in pre-production and a quick response time of the overall system are important prerequisites in this scenario.

Practically, the determination of the optimal degree of pre-production depends on (cp. [Homburg1996, pp. 661], [Piller2003a, p. 233]):

- technical criteria,
- on the costs for the intermediate storage of semi-finished products,
- the delivery time accepted by consumers,
- the accuracy of the prospects about component demand, and
- on the costs for process changeover.

2.3.3.6. Other Manufacturing Process Related Aspects

Modularity and the bisection of the value chain can be seen as essential complexity reduction techniques in the areas of engineering, management and (high level) production planning. However, to support mass customization other techniques related to the production itself, assembly, procurement and delivery have to be considered (see [Lindemann2006a, p. 63-87]).

In production, for instance, manufacturing systems such as CNC universal machines, which perform all required processing tasks (e.g., drilling, rotating, milling and bending) within a single step, on one and the same machine, offer huge potentials for flexibility. These systems not only avoid unproductive process changeover and transition times, but also improve the overall product quality by an increased positional accuracy. Similar techniques such as *Soft Tooling* (uninterrupted execution of various processing tasks with a single tool) and *Automatic Fabrication* (fully automated casting, forming, rotating, etc. of CAD planned models with special materials; mainly used for tool production) strongly involve laser technologies to enable highly flexible and efficient manufacturing (cp. [Piller2003a, pp. 305])¹⁸.

Moreover, flexible systems in procurement, transportation, production planning and control backed with computer aided manufacturing (CAM) tools may be involved in order to increase production efficiency in product customization scenarios (see [Lindemann2006a, pp. 151]).

In summary, the increased proliferation of technology and extensive automation, enabled by modern computer integrated manufacturing (CIM) systems, nowadays allow the efficient manufacturing of highly customizable goods at affordable prices.

2.3.4. Economical Aspects

Having looked at aspects related to the manufacturing of customized goods in the previous section, we want to discuss the economical prerequisites for and implications of product customization in this section.

2.3.4.1. Market and Marketing Related Prerequisites

"Competitive advantage fundamentally grows out of the value a firm is able to create for its buyers that exceeds the firm's cost of creating it. Value is what buyers are willing to pay, and

¹⁸For examples of additional technologies refer to [Lindemann2006a, p. 89].

superior value stems from offering lower prices than competitors for equivalent benefits or providing unique benefits that more than offset a higher price" [Porter1998, p. 3]¹⁹ As stated in Section 2.2.5, "Product Customization Strategies", mass customization characterizes a *hybrid business strategy*, which aims to provide additional value due to customer-tailored products (differentiation position), at prices comparable to those of mass products (cost position).

From an economical perspective, however, certain prerequisites have to be examined in order to profitably implement mass customization (see *Market conditions* in Figure 2.6, "Overview of the Necessary Conditions for Achieving Mass Customization", cp. [Blecker2005, p. 31-33]). On the one hand, *demand and structural factors* have to be examined:²⁰

Product related. Product lifecycle lengths, technological environment, necessity vs. luxury, vulnerability to substitutes.

Customer related. Requirements stability and certainty, price, quality and style consciousness.

Market related. Demand stability, market homogeneity vs. heterogeneity, saturation level, buyer vs. seller market, economic cycle dependence.

Competition related. Competitive environment, price competition vs. product differentiation.

Basically, the less the market's demand can be responded to with mass production instruments, e.g., high market heterogeneity, instable demand on a largely saturated buyer market, the higher is the probability of success when moving from mass production towards mass customization.

On the other hand, *customer demand for customization* seems a natural necessity for product customization. The willingness of the customer to (possibly) pay premium prices and to accept longer delivery times are important factors in this context. As stated by Porter, in the end, it is the customer perceived, additional value of a good induced by product customization, that decides about the customer's acceptance.

Finally, another important market related factor may be the *first mover advantage*: as Kotha pointed out, it will be beneficial for the supplier's image, if it is the first company offering customized products within a segment where previously only standard products were sold [Kotha1996, p. 447]. "Even when competitors enter the mass custom segment afterwards, they will find it difficult to prevail, especially when customers consider the first mover as the leader and best supplier of individualized products" [Blecker2005, p. 33].

2.3.4.2. Mass Customization Cost-Efficiency Overview

Even when the market related conditions for mass customization seem promising (refer to Section 1, "The Market Shift"), a company considering to move into mass customization has to carefully balance the monetary benefits and risks of this challenging business strategy upfront. Throughout this chapter, we have shown, that mass customization may likely imply numerous integral modifications to the company's business model, including the product structure, processes, machinery and infrastructure. These modifications come at their price, but also offer huge potentials for additional profits.

While in the literature, various schemas for discussing the costs and benefits of mass customization can be found²¹, in this section, we will present the version from Lindemann et al. [Lindemann2006a, pp. 166].

¹⁹Cited from [Blecker2005, p. 11].

²⁰The factors presented are derived from Pine's market turbulence map introduced in [Pine1992, p. 66].

²¹For example in [Reichwald2009, pp. 241] or [Piller2003a, pp. 169].

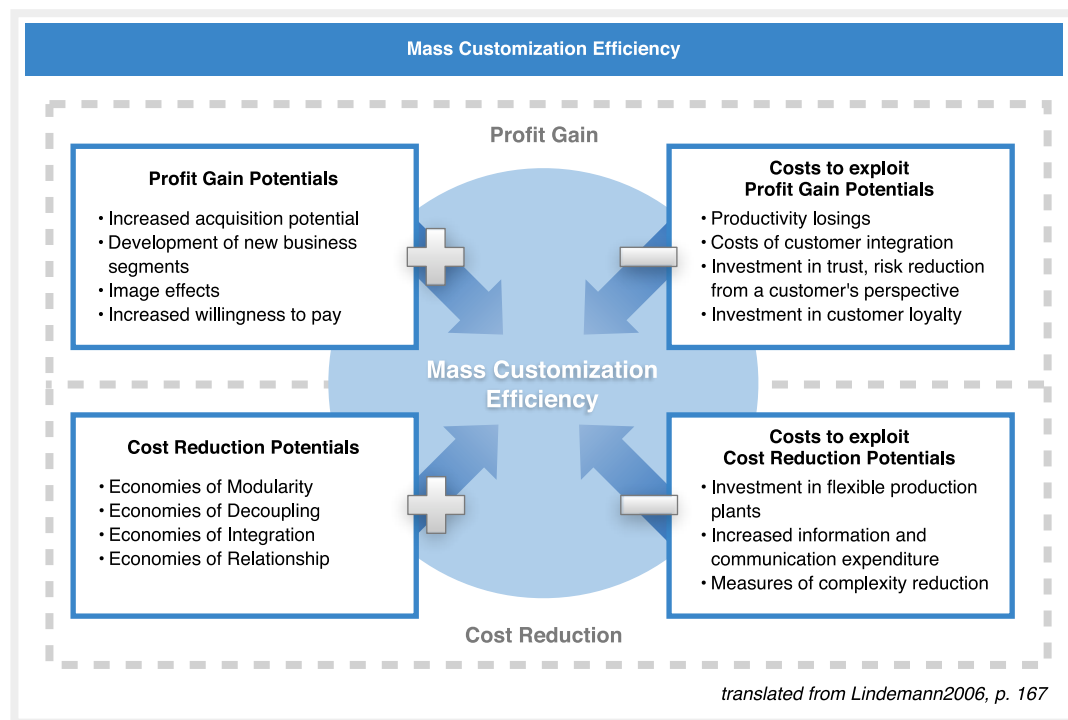


Figure 2.12. Influencing Factors for Mass Customization's Efficiency

Figure 2.12, “Influencing Factors for Mass Customization's Efficiency” gives an overview of the factors, that influence the profitability of mass customization. We will discuss these factors in more detail shortly.

In general, profit gains are accompanied by costs to exploit these potentials. Likewise, cost reduction potentials can only be freed up, when certain investments, that induce additional costs, are made. In the long-run, a product customization strategy is only profitable, if the costs to exploit additional profits and cost reductions do not exceed the actual value generated by these potentials.

2.3.4.3. Profit Gain Through Product Customization

In terms of **profit gain potentials**, the customization options of a product can be marketed as unique selling points (USPs) of a good, which offers increased acquisition potentials for sales people. A flexible, modular product architecture accelerates the development of new products by re-using existing components. With these new products, additional business segments can be occupied faster and more easily. Also the additional value added by customer-tailored product variants may not only result in positive image effects due to a heightened customer satisfaction, but also increase the willingness to pay premium prices for individualized goods.

To exploit these benefits however, the enterprise must anticipate productivity losings due to decreased lot sizes and consequently missing mass depression effects (economies of scale). Although, various techniques can be employed to reduce those losses (refer to Section 2.3.3, “Manufacturing Aspects”), anonymous series production systems are still more efficient than those directly involving the customer. In fact, the integration of the customer into the value chain leads to significantly more complex processes, which results in overall higher *production costs*. Additionally, *transaction costs*²², which are those non-value-adding costs, that arise from selling the product on the market, considerably grow as well. The direct interaction between the supplier and the customer, required to transfer of the customer's knowl-

²²The distinction between production and transaction costs has been coined by Picot in [Picot1982].

edge regarding his requirements (Reichwald and Piller call these *sticky information*, see [Reichwald2009, p. 241]) is an important cost driver in this regard. As we will see in Section 3.3.4, “Benefits”, product configurators are an eminently important tool to reduce such costs by automating the sales process from a supplier's perspective. Another issue, a product customizing company has to deal with, is *uncertainty*: as customized goods cannot be tangibly experienced during the sales process, customers often feel uncomfortable with what they're spending money for. Thus, the company must spend additional effort in order to promote trust and to lower the customer perceived risk regarding the quality and functionality of the configured product, e.g., by providing an exceptional buying experience and a highly professional overall appearance. Moreover, potentials of product customization can notably be more exhausted, if the supplier fosters a long-term relationship to the customer: on the one hand, re-using the sticky information elicited from the customer during previous order processes can considerably ease and shorten subsequent sales activities. On the other hand, the previous expenditures in trust and risk reduction pay out with each additional purchase. In short, investments in long sustaining customer relationships increase customer loyalty. [Lindemann2006a, pp. 166][Reichwald2009, pp. 240]

2.3.4.4. Cost Reduction Through Mass Customization

Beyond opportunities for profit gains, mass customization also offers **cost reduction potentials**. In the literature, these cost reductions are frequently explained with different *economies* [Lindemann2006a, p. 168-172]:

Economies of Modularity. Modularity of products and processes as fundamental principle of product customization (refer to Section 2.3.3.3, “Modularity”) is a considerable source of cost saving potentials. By distinguishing standard modules from customized modules, mass degression and specialization effects (*Economies of Scale*) as well as synergetic effects (*Economies of Scope*) can be harnessed: standardized modules can be pre-produced in high volume and with an increased variety on component level, often otherwise unused resource pools and facilities can be employed.

Economies of Decoupling. Another substantial characteristic of product customization processes is the bisection of the production cycle into a customer neutral and a customer specific part (refer to Section 2.3.3.5, “Production Process Split”). Although customer integration is an important factor for additional costs induced by product customization, the decoupling of both processes can also be a source of cost savings. Especially, when a true *customer pull strategy* is pursued by leveraging a *Make-to-Order* fulfillment policy, resources can be utilized in a strongly target-oriented way. Wastage as well as the risk to launch a product that “flops” is avoided by shipping only those products, that have been specifically requested by customers. In short, *economies of decoupling* describe those cost savings, that result, when an enterprise performs its value adding activities in an exceptionally accomplished manner, by utilizing up-to-date and precise information about the customer's demands [Lindemann2006a, p. 169].

Economies of Integration. During the product configuration step (see Section 2.3.2, “Product Customization Value Chain”), detailed information about the customer's requirements are gathered. This information can be used to enhance the efficiency of the value adding processes within the enterprise. This profitable utilization of information is subsumed with the term *economies of integration*. Aggregated know-how about the customer can be used to even better adapt the product portfolio to the market demands, especially, when the firm offers standardized products beneath to the customized ones. The direct customer interaction can supplement or even entirely supersede cost intensive market research activities, thereby, concentrating research and development activities on desired product variants. As described above, the improved knowledge about the customers can be used to develop new business segments, but can also be beneficial to gain new customers by proposing better matching product variants right from the beginning of the product evaluation phase (e.g., Amazon²³

²³See <http://www.amazon.com/>, last accessed April 20th, 2012.

provides the "Customers Who Bought This Item Also Bought..."-feature for exactly the same purpose). Likewise, the overall quality of consultancy can be enhanced.

Economies of Relationship. In the case of product customization, a tight interaction between customer and supplier during the sales process is virtually inevitable. This not only eases the elicitation of customer and market related information, but in addition offers new opportunities to increase customer loyalty. The gained know-how can significantly ease interaction between customer and supplier in further purchases, thereby, strongly reducing transactional costs. Particularly, when the compiled information was complex or hard to determine (e.g., a 3D model of the customers body, scanned with expensive equipment at a retail store), customers tend to repeatedly purchase a good from the same supplier, provided that they are satisfied with the previously bought product. An intensified customer retention is the consequence, which, as another positive side effect, builds an even higher market entry barrier for competitors. Also, an increased customer loyalty eases cross-selling opportunities by simultaneously lowering marketing and acquisition costs.

Again, in order to exploit the cost reduction potentials explained above, *additional production and transaction costs* have to be anticipated. On the one hand, a number of investments in various areas of the company's value chain are required, which result in increased fix costs. On the other hand, variable costs grow due to the overall increased complexity (e.g., planning, production and distribution complexity) induced by customer integration.

Additional *production costs* arise from required investments in modular product and process architectures and flexible production plants, that efficiently support the manufacturing of customized goods. The fix cost drivers span from expenses for research and development activities in order to design a modular product range, over expenditures for coping with the increased complexity in planning, control and distribution, up to flexible production plants. Thus, investments in both hardware (e.g., CNC²⁴ universal machines) and modern software infrastructure (ERP²⁵, CAD²⁶ and PDM/PLM²⁷ software, production related systems like CIM²⁸ including MRP²⁹ and production planning and control software) are required. Further variable, production related costs result from efforts for planning and coordinating the manufacturing process of customized components and their customer specific assembly.

Transaction costs grow due to the fact that the overall information and communication activities are intensified. Opposed to mass producers, which sell their products through various channels with the help of retailers, companies pursuing product customization mostly interact directly with the customer. Beyond common pre-sales, sales and after-sales tasks, the elicitation of customer requirements are time consuming and costly activities. Additional transaction costs are thus required for installing a product configuration system, that supports the specification process and helps the customer to easily formulate his requirements (see also Section 3.3.4, "Benefits" for a more detailed analysis of the efficiency of a configuration system). Along with a product configurator, the company must spend efforts in reducing the customer perceived risk related to the uncertainty about the customized product's quality and functionality, which also increases transaction costs.

Finally, another cost driver are measures to complexity reduction. In order to cope with the huge variability offered by product customization, complexity management techniques must be employed. Those measures include, but are not limited to, the installation of appropriate information systems, that support the management of product variants as well as the targeted planning of the variety of products.

As the previous elaborations have shown, product customization strategies not only provide potentials for profit gains but even offer cost reduction opportunities. Consequently, if an

²⁴Abbrev. Computerized Numerical Control

²⁵Abbrev. Enterprise Resource Planning

²⁶Abbrev. Computer Aided Design

²⁷Abbrev. Product Data Management/Product Lifecycle Management

²⁸Abbrev. Computer Integrated Manufacturing

²⁹Abbrev. Material Requirements Planning

enterprise manages to exhaust these potentials, a hybrid business strategy, such as mass customization, can be profitable by all means.

In this section, we described the potentials for additional profit gains and cost reductions from a high level perspective. For the purpose of this work, the qualitative view is sufficient. For a more quantitative evaluation attempt, refer to Reichwald and Piller's calculation example provided in [Reichwald2009, pp. 263].

2.3.5. Marketing Aspects

To complete the picture on implementing product customization, we want to close up with marketing related aspects. Particularly, we will take a look at the sales relevant processes of product customization. In this context, we will see that product configurators play an essential role, which is why we characterize those processes in more detail, here.

As already discussed in Section 2.3.2, “Product Customization Value Chain” (see Figure 3.1, “Development Process for Custom Products”), the value creation process involves both the customer and the supplier. In fact, the buying process of customized products is characterized by a strong and intensive interaction between both parties.

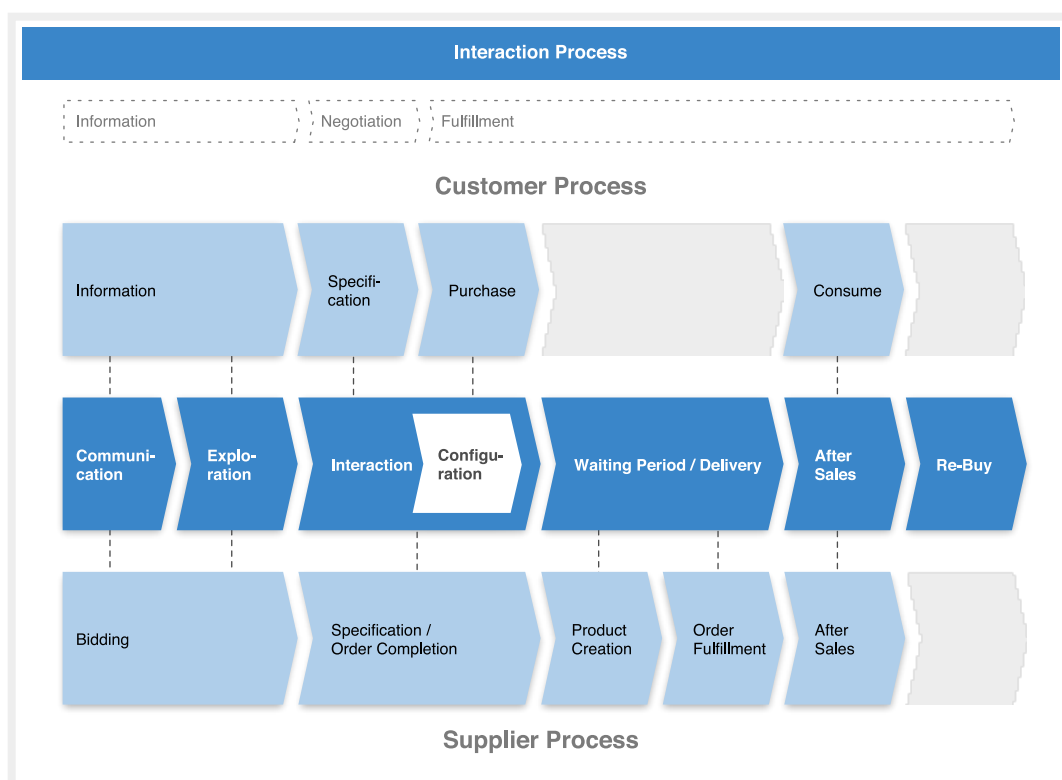


Figure 2.13. Interaction Process in the Context of Product Customization

In the following, we will take a look at the *buying process* (customer perspective) on the one hand, and at the *selling process* (supplier perspective) on the other hand. This helps us, to characterize the resulting *interaction process* in detail. Figure 2.13, “Interaction Process in the Context of Product Customization” demonstrates the relationship between the customer's and the supplier's process.

Let's begin with the buying process.

2.3.5.1. The Buying Process (Customer Perspective)

The *buying process* comprises all steps that a consumer performs in order to satisfy his needs [Scheer2006, p. 13]. Starting with the identification of a particular need, which may be triggered by the consumer himself (e.g., due to experience) or activities performed by an entity, e.g., marketing / advertisement), a customer initiates the buying process.

It begins with the *information phase*, during which the customer seeks information to satisfy his needs. The customer usually evaluates multiple alternative products from various suppliers depending on

- personal (e.g., cognitive skills),
- social (e.g., age),
- task-related (e.g., number of available alternatives),
- financial (e.g., buying power),
- situational (e.g., cognition), as well as
- information (e.g., forms of information), and
- problem related (e.g., product knowledge) factors.

Additionally, buyer independent factors such as

- economical conditions (e.g., political situation),
- social environment (e.g., norms), and
- specifics related to the particular consume situation (e.g., ambient temperature)

influence the customer.

After the information seeking phase, the consumer starts to further specify his need in terms of a concrete demand. During the *specification phase*, the customer selects a particular product type and defines the requirements on the product in question.

The *buying phase* starts, when the customer decides to buy the good: a decision, which may lie between planned / rational, limited, experienced and impulsive / spontaneous boundaries [Leckner2006, p. 28p].

After purchasing the product and the waiting period until the product is delivered, the *consume phase* begins. In case of customized goods, this is the first time, the customer can compare whether the purchased product really matches his initial needs (cp. [Scheer2006, p. 13-18]).

2.3.5.2. The Selling Process (Supplier Perspective)

On the opposite side, the *selling process*, begins with the *bidding phase* (also called *pre-sales*). The main tasks of pre-sales is to generate interest in the company's offerings and to acquire sales opportunities. The marketing activities at this stage are mainly driven by the company's defined **marketing mix** (see Figure 2.14, "Marketing Mix for Product Customization"). Reichwald et al. extend the well known "4 Ps", which comprise the policies of *product*, *price*, *promotion* and *place*, with two important instruments related to product customization, namely *configuration* and *customer relationship management* [Reichwald2006b, pp. 20]³⁰:

³⁰The following list has partly been translated from [Scheer2006, p. 19].

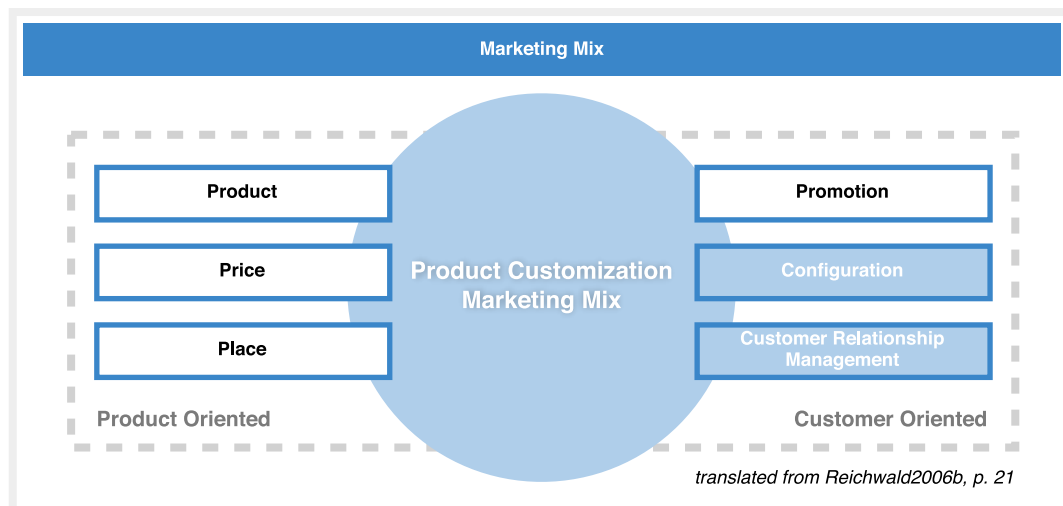


Figure 2.14. Marketing Mix for Product Customization

Product. Defines the presentation of a product's or a product program's features in terms of innovation, variation, differentiation and elimination of similar products on the market. Product customizers particularly describe the customization options offered by the product as unique selling points (USPs).

Price. Compromises the pricing and other related conditions. Especially, the ratio between a product's benefit and its monetary as well as non-monetary reward is being marketed. In case of customized products, customers are often willing to pay premium prices.

Place (also Distribution). Addresses the organization and the management of the transfer of the product from the supplier to the customer through various distribution channels. In terms of product customization, the fact that selling a product requires mostly a direct interaction between the supplier and the customer (except in *point-of-sales customization* or *self customization*, see Section 2.3.1, "Mass Customization Achievement"), requires special considerations.

Promotion (also Communication). Describes additional measures for presenting and distributing information about the supplier and its product range among the consumers. Related to product customization, the differentiation position may require extra marketing efforts, especially for market newcomers. However, the differentiation can also lead to an increased perception within the targeted clientele without additional activities. Regarding advertisements, the additional complexity induced by the specification process as well as the special role, that the customer owns during the value creation process, must be addressed.

Configuration. The product configuration system is the central tool that **facilitates the interaction between supplier and customer**. In fact, the customer actually communicates solely with the configurator itself, which is in charge of representing the enterprise in front of the customer. Beyond its specification capacity, it's also the task of the configurator to communicate the supplier's competencies. Optimally, it even advocates enthusiasm among the customers. In summary, the configuration system is considered one of the most important marketing instruments as part of the product customizer's marketing mix.

Customer relationship management. Another vital aspect of product customization is the establishment of *learning relationships* (see Section 2.2.5, "Product Customization Strategies"), which, in turn, requires a thorough management of customer relationships. Again, due to the direct interaction with the products' recipients during the configuration process, the particular customer relationship can be maintained in an exceptionally good quality. Given that the information systems employed offer corresponding capabilities, personalized commu-

nication, tailored to each customer, the so called *1:1 marketing* (see [Scheer2006, p. 21]), can be facilitated.

Having acquired the customer's interest in the bidding phase, the customer is supported during the *specification* of his requirements by the configurator. The complexity of the configuration process and the customization intensity is determined by the point in time when the customer is integrated into the value chain (see Section 2.3.3.4, "Order Fulfillment Strategies") and the degree of customizability (see Section 2.1.3, "Customized Products and Product Configurations").

Upon finishing the specification, the supplier complements the product configuration with a corresponding quote including a final price and a preliminary delivery date.

Finally, the order submission initiated by the customer completes the order process. After purchase, the customer specific *production* of the customized good, i.e. the fulfillment process described in Section 2.3.3.4, "Order Fulfillment Strategies", is triggered. Depending on the negotiated conditions, additional services (e.g., installation) are carried out during the *fulfillment process*. Finally, the *after sales* activities are concerned with customer care, including maintenance, repair, training and eventually product recycling (cp. [Scheer2006, p. 19-25]).

2.3.5.3. The Interaction Process of Product Customization

Now, that we have seen the buying as well as the selling process from both the customer's and the supplier's perspectives, we want to discuss the **interrelated interaction process** in between. Notably, we want to highlight product customization specifics here, instead of repeating the explanations from above.

Related to the *communication* in the customer-supplier interaction process, it is worth mentioning, that in case of customizable products, due to an increased complexity of the product (opposed to equivalent standard products), a higher demand on information exists. Within the communication phase, the supplier has to explain the tasks, that the customer has to perform in order to successfully specify a customized product and customization options should be precisely described. Furthermore, the added value offered by those options must be a prominent subject of communication. Since the customized product cannot be viewed or physically touched in advance, additional efforts have to be undertaken to reduce the customer perceived uncertainty and to produce trust into the offered capabilities (cp. [Reichwald2006a, p. 118]).

It is the task of the configurator to present the company itself, as well as the its product portfolio, to the customer. During the *exploration phase*, the customer occupies himself intensively with the customizable product. He explores the different configuration options in order to evaluate whether the configurator, i.e. the product architecture, is capable of optimally satisfying his needs. In this context, the usability and the user experience promoted by the configurator is of dominant interest.

The transition from the exploration to the **interaction phase**, which is the key phase of the process, is fluent. At this stage, the customer must be actively supported to translate his needs into specific product attributes, such as appearance, dimensioning and functionality. Product configurators are the software tools to accommodate the specification support. The configuration process is explained in detail in Section 3.2, "Product Configuration". During configuration, the tool continuously collects information about the customer and his preferences. The data is stored in the user's profile and forms the basis for future learning relationships. Moreover, the configurator has several other functions, that will be discussed in detail in Section 3.3, "Product Configurators".

Although the configuration process may last only minutes, it can span several hours or even days until the customer submits the configuration and places an order. The interaction phase usually closes up with the payment of the product. The result of the configuration process is a *product configuration* (see Section 2.1.3.3, "Product Configurations"), that is, a specification

of the product including all characteristic values, that correspond to the custom product. The product specification is used in any subsequent value creation steps, which are triggered immediately.

Next, the supplier starts with the manufacturing of the customized good according to the product specification. While production is in progress, the customer is *waiting* until the product is delivered. In terms of customized products, the filling of the waiting period (e.g., by regularly reporting the order fulfillment status to the customer, the so called *order tracking*), is essential to reduce the customer perceived uncertainty, which also affects his overall satisfaction heavily.

Once the custom product has been shipped to the client, he starts with actually using it. During the consume phase, the supplier must still support the customer with *after sales* activities and should try to retain contact to him. As an example, the supplier might want to request feedback from the customer to verify, whether he's satisfied with the tailored product. Moreover, due to the direct relationship, a personalized communication with the customer should be established in order to foster customer loyalty. This includes the ability to access the customer's profile and information about previous orders, including custom product specifications, in response to customer inquiries (e.g., through a call center).

Finally, the elicited data during the interaction phase can be used (sometimes even periodically, e.g., in case of dissipating products) to trigger a *re-buy phase*. In repeating interaction cycles, existing knowledge about the customer's preferences can and should be used to speed up and ease the configuration process. In this sense, a long-term relationship to the customer can be realized.

The in-depth discussion of the main processes involved during sales activities for product customization closes up our detailed explanation on how to implement mass customization. In the remainder of this work, we will solely focus on configurators as important tools for realizing the interaction process. Especially, we'll discuss their technical implementation aspects. Before, however, we'll finally give a short overview of application areas of product customization.

2.4. Application Areas and Examples

Product customization appears in a large diversity of application areas. While physically tangible products have been stressed throughout this chapter, product customization applies to non-tangible services and software equally. From B2C³¹ to B2B³² markets, customization is used within a variety of industries³³:

- **Accessories.** Jewelries, rings, bags and more with personalized forms, designs and signatures.³⁴
- **Apparel.** Individually designed clothings with customizable fit, material and appearance.³⁵
- **Automobile & Vehicles.** The majority of car manufacturers allows customers to customize their new car using a product configurator online.³⁶ Also, many commercial vehi-

³¹Abbrev. Business to Consumer

³²Abbrev. Business to Business

³³The following list has been extracted from <http://www.configurator-database.com/>, last accessed July 30th, 2012.

³⁴E.g., Stein Diamonds [<http://www.steindiamonds.com/design-your-own/>], Der Trauring Juwelier [<http://www.dertrauringjuwelier.de/trauringkonfigurator/>], Berlin Bag [http://www.berlinbag.com/marketplace/customize_product], last accessed July 30th, 2012.

³⁵E.g., Spreadshirt [<http://www.spreadshirt.de/t-shirt-gestalten-und-bedrucken-C59>], Hemdschneider [<http://www.hemdschneider.de/meinhemd.php>], last accessed July 30th, 2012.

³⁶E.g., VW [<http://www.volkswagen.de/de/CC5.html>], BMW [http://www.bmw.de/de/de/general/configurations_center/configurator.html], Mercedes-Benz [http://www.mercedes-benz.de/content/germany/mpc/mpc_germany_website/de/home_mpc/passengercars/home/new_cars/]

cle manufacturers support the customization of their offerings according to the customer's needs by utilizing a modular product architecture.³⁷ Other vehicle industries including motorcycles, bicycles, luxury yachts and others are build around a modular core which offers customization opportunities.³⁸

- **Beauty & Health.** Personalized perfumes, make-up, cremes, soaps, shampoos, vitamins and more.³⁹
- **Electronics & Media.** Since computer's have been designed modular almost from the beginning, many hardware manufacturers today offer the customization of their systems.⁴⁰ Other multimedia brands provide personalization offerings in terms of functional adaptations, too.⁴¹
- **Food.** From beer, cakes, chocolate or ice cream up to tea, wine - even meat - can be customized.⁴²
- **Footwear.** Personalized shoes with custom fit and design.⁴³
- **House & Garden.** Windows and doors are common customized products, furniture, entire houses or gardens are more exotic examples for the capabilities of product customization.⁴⁴
- **Industry.** From industrial machineries, machine parts to customized gases a huge variety of customizable products is available.⁴⁵
- **Services.** Also service companies from the financial sector or assurances offer customer-tailored offerings.⁴⁶

Even fully individual products, build from custom 2D or 3D models can be configured and ordered online⁴⁷. The variety of customizable goods ranges from simple products up to highly complex systems.

In principle, every product, that is compound of modules, owns customization potential. Though, in the end, it is the customer who decides whether individualization is demanded or not.

model_overview_configurator.flash.html], Toyota [http://www.toyota.de/cars/new_cars/configurator-index.tmx], Renault [http://www.renault.de/renault-modellpalette/renault-pkw/index.jsp], Ford [http://www.ford.de/Pkw-Modelle], last accessed July 30th, 2012.

³⁷E.g., Scania [http://www.scania.de/] (according to [Hvam2008, p. 144]), Setra [http://www.setra.de/de/beratung-kontakt/designcenter.html], last accessed July 30th, 2012.

³⁸E.g., Harley-Davidson [http://customizer.harley-davidson.com/], Bausatzrad [http://www.bausatzrad.de/], Hanse Yachts [http://www.hanseyachts.com], last accessed July 30th, 2012.

³⁹E.g., My Parfuem [http://www.myparfuem.com/Parfuem.html], e.l.f. [http://www.elf-kosmetik.de/de/all/elements/make-up-box-konfigurator/], Haircare4Me [http://www.haircare4me.de/], Purmeo [http://www.purmeo.de/], last accessed July 30th, 2012.

⁴⁰E.g., Apple [http://store.apple.com/de/configure/MD322D/A?], Dell [http://configure.euro.dell.com/dell-store/config.aspx?oc=d0023203&c=de&l=de&s=dhs&cs=dedhs1&model_id=inspiron-one-2320], last accessed July 30th, 2012.

⁴¹E.g., Loewe [http://www.loewe.tv/int/products/individual/personalization.html], last accessed July 30th, 2012.

⁴²E.g., Chocomize [http://www.chocomize.com/personalized-chocolate-bars], DeineTorte.de [http://www.deinetorte.de/], WurstMixx [http://www.wurstmixx.de/bestellung.php], last accessed July 30th, 2012.

⁴³E.g., Mi Adidas [http://shop.miadidas.de/miadidas/], Nike iD [http://store.nike.com/de/de_de/?l=shop,nikeid], last accessed July 30th, 2012.

⁴⁴E.g., Audena [http://www.audena.de/], Fensternorm [http://www.fensterorm.com/], Regnauer [http://www.regnauer.de/hausbau/vitalhaeuser/haus-konfigurator/], last accessed July 30th, 2012.

⁴⁵E.g., Linde Gase [http://www.linde-gase.de/produkte/spezialgase/konfigurator.html], ThyssenKrupp Elevators [http://www.ceteco.biz/de/], last accessed July 30th, 2012.

⁴⁶E.g., Huk24 [http://www.huk24.de/], last accessed July 30th, 2012.

⁴⁷E.g., Formulor [http://www.formulor.de/], last accessed July 30th, 2012.

2.5. Summary

In this chapter, we've presented fundamental knowledge related to the global context of our work: product customization.

We introduced important terms related to the subject, such as *products*, *components* and *variants*, to establish a common understanding of the vocabulary used throughout the work. Next, we had a look at *variant management* and the complexity induced by this approach. Finally, at the end of the first section, we explained the notions of *customizable products* and *product configurations* and showed in general "what actually can be customized" by defining a list of customizable areas. This list will be an important driver for the modeling concepts worked out in Chapter 4, *Methodology and Conceptualization*.

While focussing the *product perspective* in the first section, the following section focusses the *process perspective*. We explained classical production processes such as *one-of-a-kind production*, *series production*, *variant production* and *mass production* in order to demonstrate the differences to the later introduced *product customization* strategy. We identified product customization as a strategy, that is capable of fulfilling the increasing demand for customized products and motivated in Chapter 1, *Introduction*. We stressed the involvement of the customer into the value creation process as the most important implication of any product customization strategy. Furthermore, we distinguished *mass customization* as a special case of product customization that not only focusses on *product differentiation*, but also considers the *cost position* in order to offer customized goods at prices comparable to those of standard products. At the end of this section, compared product customization with other production strategies. Having read this section, the reader should be able to answer the question "what product customization actually is" and "what it is used for".

In Section 2.3, "Product Customization Implementation", we elaborated on the topic of "how to implement mass customization". We discussed in detail the *product customization value chain*, which shows significant differences to the value chain of a mass producer, since the main activities are divided into *standard activities* (executed customer neutral) and *individual activities* (executed specifically for a customer). As the split of the value chain - a consequence of customer involvement - significantly increases the overall process' complexity, we discussed methods and techniques for managing this complexity. In this context, we described *modularity* and a *modular product architecture*, as well as the bisection of the production process into a *customer neutral part* and a *customer specific part* as the most important factors. The details about economic aspects showed, that there are huge potentials for profit gains on the one hand, and cost reduction opportunities on the other hand. We argued that a company exhausting those potentials will be able to pursue product customization in a profitable way. Finally, the section on marketing aspects disclosed important insights of the interaction process between customer and supplier. Within this interaction process, configurators play a prominent role as they're used to elicit the customer's requirements on the tailored products beyond communicating the supplier's product portfolio and capabilities.

We closed up with this chapter by presenting manifold areas of application for product customization.

In the next chapter, Chapter 3, *Configurators*, we will concentrate on *product configurators*.

3

Configurators

In Chapter 2, *Product Customization*, the business strategy for offering customized products has been described in detail. We identified the *integration of the customer into the product creation process* as crucial aspect of product customization. Specifically, the customer is involved in the *configuration phase* of the value chain (see Section 2.3.2, “Product Customization Value Chain”), which aims at eliciting the customer's requirements on the tailored product. Without these information, the customized product cannot be build. Therefore, the configuration, respectively the requirements specification phase, plays a prominent role in the entire process. Its result is a detailed specification of the product, that optimally fulfills the customer's needs.

In order to efficiently perform the specification process, the configuration phase must be supported by information technology: **product configurators** are software tools, that help the customer (or a sales representative) to transform individual needs into specific attributes of the customized product. Thus, configurators are a central element within the sales process.

This chapter concentrates on configurators and can be seen as the (information) technical point of view on product customization. At first, we will describe the act of product configuration and place it in the context of product customization. Then, we will describe product configurators in detail, covering both general features and requirements as well as technical topics.

Configuration in the Context of Product Customization

In product customization, modularity and the resulting modular product architecture are the foundations of customizable goods. Particularly, the *limitation* of customization options to the most important properties of a product from a customer perspective, enable the efficient manufacturing of such goods. Through the precise planning and definition of customizable areas within the product architecture (see Section 2.1.3.1, “Customizable Areas”), and especially by disallowing arbitrary customizations, a *stable solution space* is established. The fast and cost efficient manufacturability of the resulting products without expensive reconstructions can be guaranteed. During the configuration phase, the customers configure their customized product within the stable solution space, by selecting components and specifying the values of parametric attributes. Thus, as can be seen in Figure 3.1, “Development Process for Custom Products”, the development of customized products can be seen as a two-split process [Piller2003b]:

Customer neutral development phase. Within the first phase, the enterprise develops an optimized, stable solution space in terms of a modular product architecture (see Section 2.3.3.2, “Product Architecture”), which is described as a **product model**. The customization options build into the product architecture represent the capabilities of the manufacturer offered to its customers. Characteristic for this part of the development process is, that it's customer neutral, that means, the customer is not involved in this phase yet. Moreover, at this stage, no products have been manufactured at all.

Customer specific development phase. During the second phase, the customer (or a representative) configures the product within the range of the solution space, thus performs the actual **product configuration** (in terms of an action). Upon order submission, the resulting *product configuration* (in terms of a subject) is transformed into the final specification of the custom product, which forms the basis for subsequent fulfillment processes, including the good manufacturing.

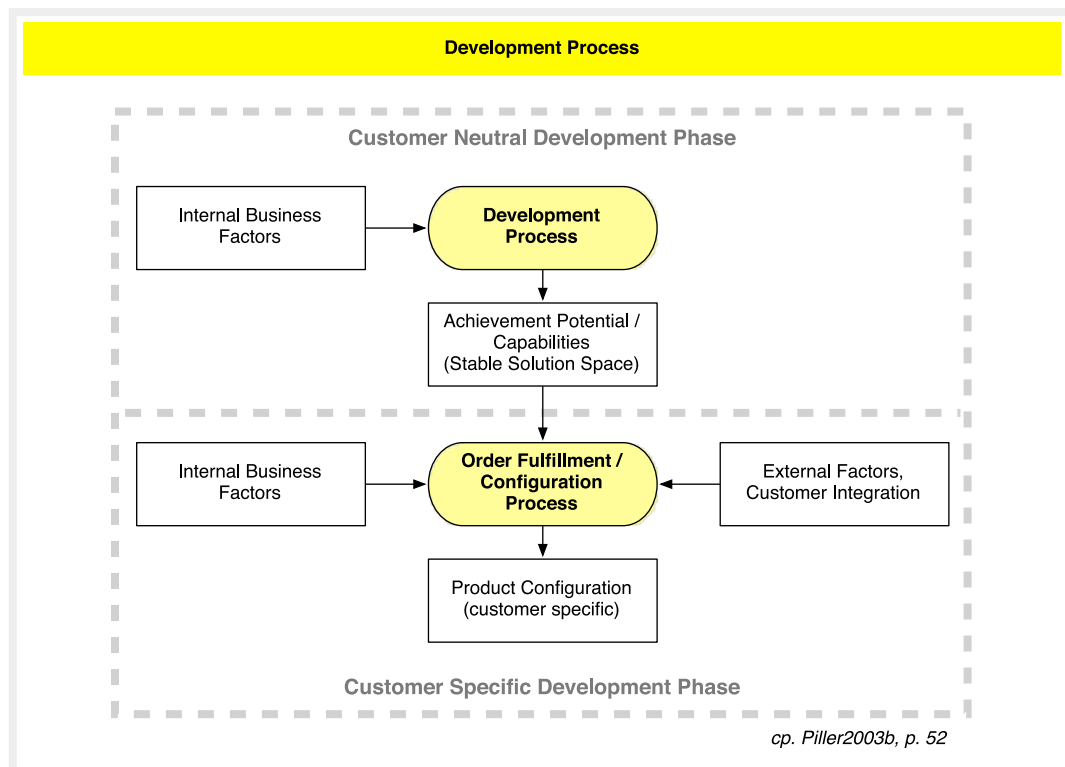


Figure 3.1. Development Process for Custom Products

In other words, during the development process (customer neutral development phase), the supplier designs a *product model* that determines the configuration options presented to the user. Within the customer specific development phase, the *product configuration* aims at precisely specifying the customized product. Particularly, this happens within the *configuration process* driven and supported by a **product configurator**. In the following we will describe these terms and the concepts behind them in detail.

3.1. Product Models

The **product model** is the digitalized, formal description of the product range (see Section 2.3.3.2, “Product Architecture”) and forms the basis for the configuration process. Within the product model, the *compositional structure* inferred by **modularization** efforts (discussed in Section 2.3.3.3, “Modularity”) and the sales relevant, customer perceivable *product characteristics* are recorded. Particularly, the **customizable areas** of the product (described in Section 2.1.3.1, “Customizable Areas”) are represented within the product model. Additionally,

constraints, that stem from physical, technical or marketing related factors, may be defined to express restrictions of the solution space.

While numerous product characteristics, including detailed technical attributes, can be captured in the product model (see Section 2.3.3.2, “Product Architecture”), for configuration purposes mostly only those attributes, that concern the functionality, form and appearance are relevant to the customer. The granularity, that is, the level of detail encompassed in the product model, must be carefully considered to balance between ease of configurability and expressiveness of customer requirements [Renneberg2010, p. 81].

Basic Meta Model for Generic Product Modeling

There exist various approaches to product configuration, that foster different methods of product modeling (we will discover several of them in Section 3.3.3.5, “Implementation Aspects”, “Configuration approaches”). Nevertheless, the description of products by means of

- *structure (components)*,
- *attributes*, and
- *constraints*

is common to most of them. These concepts build up a **basic meta model** to describe product models (see Figure 3.2, “Basic Generic Meta Model for Configurable Products”). The meta model elements can be used to describe a highly diverse set of products, originating from different product domains, which is why it's called *generic* product meta model.

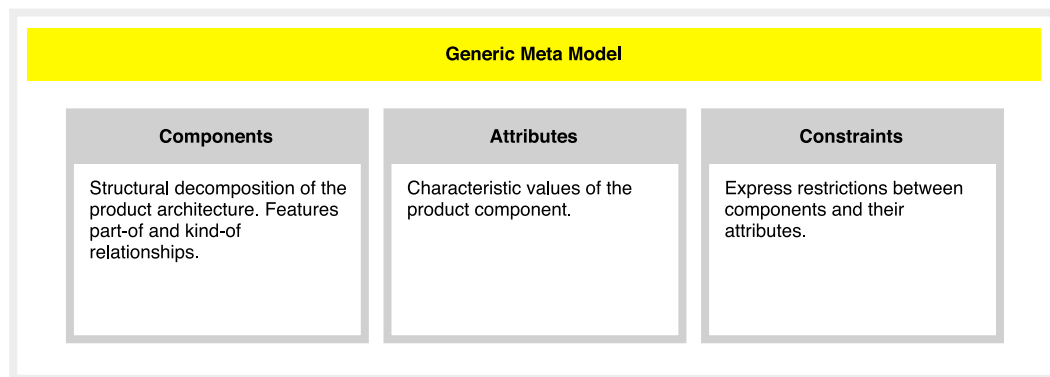


Figure 3.2. Basic Generic Meta Model for Configurable Products

Product Models in the Context of Configuration

Used in the context of product configuration, from these model elements, various **configuration decisions** (also referred to as *configuration options*) can be derived. In the following sections, we will discuss each element in more detail and describe the relevant configuration decisions more precisely.

The meta model as well as the configuration decisions form the theoretical basis for the conceptual framework described later in our methodology: exactly these concepts and options will be supported by our framework. Thus, we'll come back to these in more detail in Chapter 4, *Methodology and Conceptualization*.

In the end, the product model comprises all required information for a generic product configurator to generate valid configurations of a customizable product.

3.1.1. Components: Structural Decomposition

With the help of **components** (also called *modules*), the *structure* of a product is described. Components encapsulate closed physical or functional units. They can be related to each other in order to express *classification* (kind-of) and *containment* (part-of) hierarchies.

In short, components can be characterized as follows:

Identifier/Type. Components are identified by a unique (qualified) name. This name also manifests the component's type.

Attributes. To further characterize components and distinguish different variations of these, a component defines attributes (see Section 3.1.2, “Attributes: Component Characteristics”).

Relationships. A component can be associated to others by different means:

- **Kind-Of.** Identifies a component to be a specialization or a generalization of another type of component; also known as “is a”-relationship.
- **Part-Of.** Designates a component to be a part of another component. This way, a compositional (containment) hierarchy can be modeled. Again, also known as “has a”-relationship, *aggregation* or *composition*.
- **Association.** Is a type of relationship, that can be further attributed to indicate a special kind of association: e.g., the *uses* relationship indicates that a component makes use of another object, but the targeted component must not necessarily be a part of this object only. Likewise, a *dependency* relationship signals, that a component depends on the existence of another one.

Multiplicity. If a component is part of another component, a multiplicity can be provided in order to specify the number of objects participating in the association.¹

Components in the Context of Configuration

When describing product structures in the context of product configuration, the compositional structure is of primary interest. Most often, the part-of structure is depicted as tree-like structure (see Figure 3.3, “Compositional Structure of a Bike”): while the root represents the final product, the nodes correspond to components. An intermediate node designates an *aggregated component*, that consists of multiple other (sub-) components. A leaf node describes a *primitive component*, which is not further divisible.

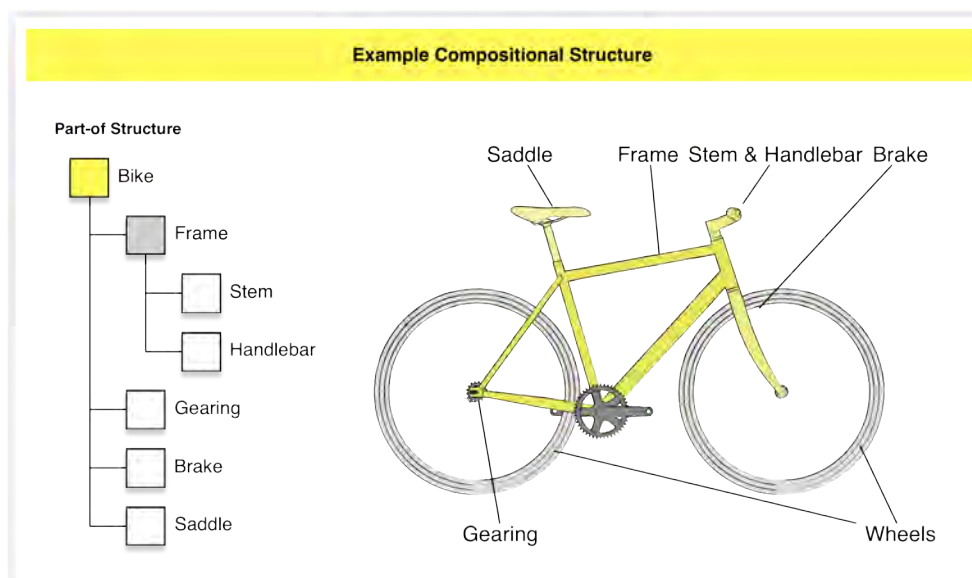


Figure 3.3. Compositional Structure of a Bike

¹In the following we use the notation $[n..m]$, where n represents the included lower bound and m the included upper bound. A fixed number of n items is notated as $[n]$.

Configuration Decisions Related to the Compositional Structure

Related to the structural aspects of a product, during the configuration process feasible decisions are enumerated in the following **list of structural configuration options** (for exemplary illustrations see Figure 3.4, “Structural Configuration Decisions”)²:

- i. **Component type decision.** Since a component may appear in a *kind-of* relationship, it may be possible to use a sub-type of the stated component at the given decision point. Thus, in this case, the user must specify the exact type to use.
- ii. **Component quantity decision.**
 - A. *Optional component*: If the multiplicity of a component is $[0..1]$, the component is optional and may be specified at the user's choice ❶ ❷ ❸ (cp. Figure 3.4, “Structural Configuration Decisions”).
 - B. *Mandatory component*: If the multiplicity of a component is $[1]$, the component is mandatory and must be specified ❹-❺ ❸ ❹.
 - C. *Multiple components*: If the multiplicity of a component is $[n..m]$, where $n \geq 0$, $m > n$, the user must specify the exact quantity of components involved. The quantity must be greater or equal than n and less or equal than m . In this case, two approaches can be differentiated ❹-❺:
 - a. a *single* component is inserted *multiple* times ❺, or
 - b. *multiple* components are inserted *single or multiple* times ❻.
- iii. **Component variety decision.**
 - A. *Fixed component*: In certain situations, there may only exist a single component instance available for a decision. We say the decision's option is *fixed*, since the user has no alternatives to choose from. This means for optional components, that the user may choose, whether *exactly that* component instance may be part of the product or no such component will be part of the product at all ❷.
 - B. *Alternative components*: Allows the user to choose from different alternative component instances ❸.
- iv. **Component customization decision.**
 - A. *Selected/parameterized component*: The user can select a predefined component instance, which may, however, define parameterized attributes ❶-❸.
 - B. *Constructed component*: Allows the user to construct a new component instance ❹.

²See also [Renneberg2010, p. 83-84], [Lindemann2006b], [Lindemann2006c], who define a subset of these options.

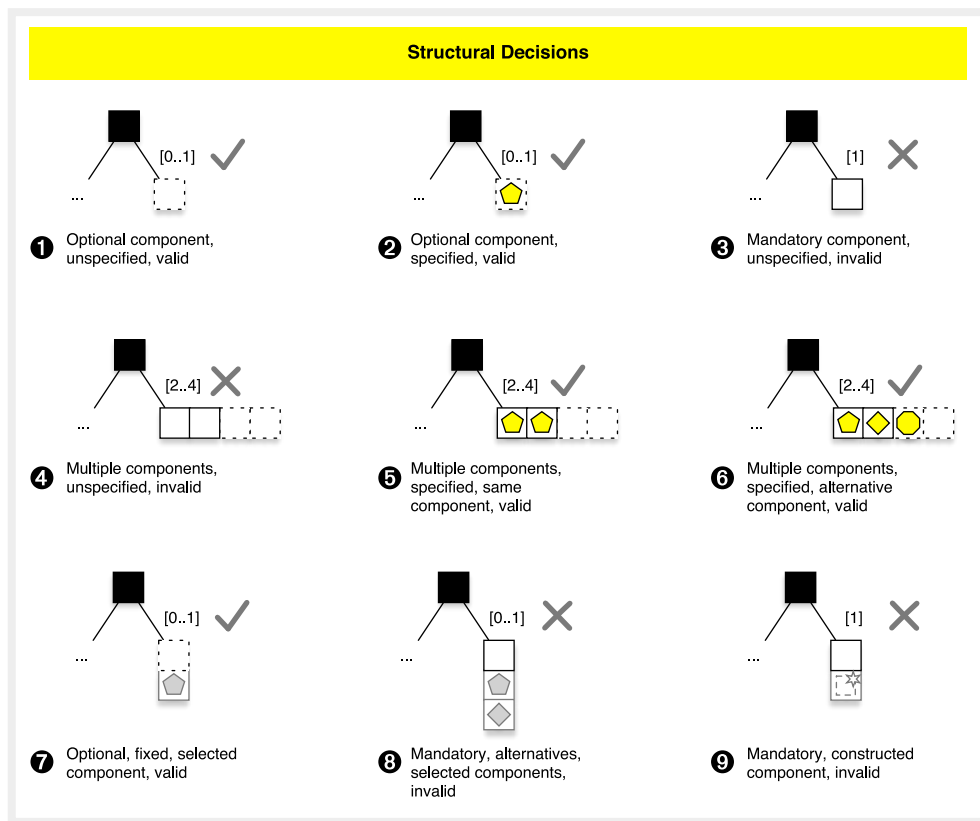


Figure 3.4. Structural Configuration Decisions

Within our conceptualization, structural aspects will be covered in Section 4.4.1, “Structure Modeling”. There we will discuss various examples of different component structures and decisions.

3.1.2. Attributes: Component Characteristics

Attributes (also referred to as *properties*, *options* or *degrees of freedom*³) are used to describe components in more detail. For the purposes of configuration, only the value-adding characteristics of a product are relevant and should be included in the product model. Variants of a component primarily result from component instances, that differ in various attribute values.

An attribute is characterized as follows:

Name. The name that identifies the attribute within the scope of a component.

Type. The value type of a component, e.g., number, string, symbol (enumerated value), etc.; must be compatible with the attribute's domain (see below).

Multiplicity. Also attributes may specify a multiplicity. Although, in most cases the cardinality is [1], for attributes, the following multiplicity alternatives may be relevant⁴:

³In our opinion the name “degree of freedom” as used, for instance, in [Renneberg2010], is an inadequate substitution term for “attribute”, since not only attributes can be customized. Instead, a *degree of freedom* can also denote a decision point related to the component structure, e.g., the cardinality of a component.

⁴See also [Jørgensen2003], who introduces a subset of these multiplicities.

- **AtLeast(n).** The selection of *n* and more values is allowed. *n* is the lower bound.
- **AtLeastOne.** The selection of 1 or more values is allowed (special case of *AtLeast*, where the lower bound is fixed to 1).
- **AtMost(m).** The selection of zero to *m* values is allowed. *m* is the upper bound.
- **AtMostOne.** The selection of one value is permitted but not mandatory (special case of *AtMost*, where the upper bound is fixed to 1).
- **OneOf.** Exactly one value is mandatory and must be selected (combination of *AtLeastOne* and *AtMostOne*).
- **Optional.** Either true or false (special case of *OneOf* for boolean domains).
- **AnyOf.** Any number of values from the domain may be selected.
- **Exactly(n).** The exact number of selected values must be selected (combination of *AtLeast* and *AtMost*, where the lower and the upper bound are fixed to *n*).

Domain. Contains the values, which can be assigned to the attribute. Domains can be characterized across different dimensions:

- **Boolean domains.** Contains exactly the boolean values true and false.
- **Symbolic/Literal domains.** Domain values are discrete and represented by a symbol, e.g., blue.
- **Numeric domains.** Domain values are linear or discrete numbers, e.g., floating points 2.5 or natural numbers 3.
- **String domains.** Domain values are character strings, e.g., "King Louis II."
- **Enumerated domains.** Contain a bound, enumerable set of values, e.g., {blue, red, green}.
- **Discrete domains.** Contain a limited, though not enumerable number of values, e.g., a natural number between 0 and 1500, in short [0-1500].
- **Linear domains.** Contain an unlimited number of values, e.g., a floating point number between 0 and 1, in short [0.0-1.0].
- **Interval domains.** Allow the specification of intervals, e.g., {[10-15]}.

Attributes in the Context of Configuration

Related to configurability, one can differentiate various types of attributes:

Regular attributes. Are attributes that cannot be customized, but which may vary depending on the selected variant of a component (e.g., the price of a component). The domain of a regular attribute may contain multiple, different values.

Constant attributes. Cannot be customized and is fixed to a specific value independently of the selected variant (e.g., the material attribute of a component may be fixed to the value "aluminium", if all variants of a component are build from that material). The domain of a constant attribute contains a single, fixed value.

Calculated attributes⁵ Contain values that are calculated from other attribute values (e.g., the weight of a component is calculated from the dimensions and the material of a component). The domain of a calculated attribute depends on the calculation.

⁵Also known as *performance variables*, see [Businger1993, p. 15].

Customizable/Variable attributes⁶ Allow the user to choose between different values. However, the value space of the attribute is discrete and bound (e.g., the user may choose the color of a component from "blue", "red" and "green"). The domain of a customizable attribute contains multiple, but limited number of enumerable values.

Parameter attributes. Represent linear, possibly unbound variable attributes that the user must specify (e.g., the length of a tube component). The domain of a parameter attribute contains an unlimited or not enumerable number of values.

We sometimes refer to *variable (component) attributes* by the means of customizable attributes and / or parameter attributes simultaneously.

Configuration Decisions Related to Component Characteristics

The following *attribute configuration options* are feasible:

v. Attribute quantity decision.

- A. *Optional attribute*: When the multiplicity of an attribute is `AtMostOne` (or equivalent), the attribute is optional and its value may be specified at the user's choice.
- B. *Mandatory attribute*: When the multiplicity of an attribute has `AtLeastOne` semantics, the attribute is mandatory and a value must be specified.
- C. *Multi-value attribute*: When the multiplicity of an attribute is greater than 1, the user must specify the exact quantity of attribute values. In visual configuration scenarios, however, the quantity of values is in most cases implicitly defined by simply specifying multiple values.

vi. Attribute valuation decision.

- A. *Selected value*: In case of customizable attributes, the user selects a value from the attribute's domain.
- B. *Custom value*: For parameterized attributes, the user enters a custom value in the given format.

Within our conceptualization, component characteristics in terms of attributes will be covered in Section 4.4.1.2, "Attributes". Again, we will discuss various examples of different component characteristics and related decisions in that section.

3.1.3. Constraints: Domain Restrictions

Regardless of the configuration approach, in the product model, restrictions will be modeled in order to exclude product variants, that cannot or shall not be build for technical or marketing reasons. So, while the definition of components and attributes (particularly, the domains of those) can be seen as *widening* the solution space, with the help of restrictions, complex relationships can be expressed, that effectively *narrow* the solution space (see also Section 4.4.4.1, "Domain Definition" in this context).

Renneberg identifies the following types of restrictions [Renneberg2010, p. 84-85]:

Boolean constraints. Allow the definition of boolean expressions that must hold true for a configuration to be consistent. In other words, when a configuration *satisfies* all constraints defined within the product model, that is, all expressions evaluate to `true`, the configuration is said to be *valid*. Using constraints, restrictions can be formulated *declaratively*.

⁶Sometimes referred to as *decision variables*, see [Businger1993, p. 15].

In practice, usually first-order logic is used to describe constraints. In this context, attributes represent the variables that occur in the logical terms. Again, constraints can be differentiated across various dimensions:

- **Logical constraints.** Solely contain variables, constants and logical operators, such as logical and, or, not, == (equals), != (unequal), <, <=, >, >=, etc.
- **Arithmetic constraints.** May also contain arithmetic operators such as +, -, /, *, % (modulo) and others.
- **Unary constraints.** Reference only one variable.
- **Binary constraints.** Reference two variables.
- **N-ary constraints.** Reference more than two variables.

Relational constraints. With relational constraints, valid value combinations (also referred to as *feasible tuples*) for a set of variable attributes can be intensionally defined (see Section 4.4.4.1, “Domain Definition”).

Rules. A rule is a logical term in the form of If <condition> Then <action>. With the help of these terms, a rule system can be realized, that combines both logical and procedural knowledge. Upon matching and evaluating the condition to true, which may happen each time the user submits a decision, the given action is executed. The action may perform state changes, which possibly triggers other rules to fire.

Constraints in the Context of Configuration

In terms of product configuration scenarios, constraints are used to ensure the validity of a specified configuration/variant.

As stated by Renneberg, in the context of model based configuration, *constraints* should be used in favor of *rules* for various reasons (cp. [Renneberg2010, p. 85], [Jørgensen2003], [Eizaguirre2008]):

- With constraints, the definition of dependencies is easier to understand and more clear, due to the fact that procedural and logical knowledge is separated.
- Constraints can be checked in an arbitrary order. In contrast, the sequence of rule evaluations is significant and affects the result. This is why, the interrelations between rules are more difficult to analyse.
- Rule systems are reactive systems, which makes it technically harder to deduct knowledge about valid attribute valuations *in advance*.

Configuration Decisions Related to Constraints

In an advanced configuration scenario, even *constraint related configuration options* can be feasible:

vii. Constraint decision

- A. **Optimization variables and constraint weighting** With the help of constraints, variables can be demarcated to be optimized within a configuration. As an example, a constraint can be defined to minimize the price of a product. A recommendation algorithm may use this information to optimize the configuration by proposing component instances with minimal prices. Additionally, when multiple optimization goals are defined, constraints can be weighted in order to compromise diverging targets. The constraints used for these purposes are sometimes referred to as *soft constraints*.

- B. Constraint relaxation** Another use case for constraint related decisions are expert configuration scenarios, where the configuration systems allows the expert user to relax certain constraints. As an example, an advanced configuration system that deals with tube systems might constrain the tube length to a certain limit for easier manufacturing. However, for special designs, an expert may be allowed to specify extra sized tube components by relaxing the length constraint.

In our conceptualization, we'll consider both boolean and relational constraints, but for the reasons mentioned above, dismiss rules. We will cover the exact semantics of constraints in Section 4.4.5, "Constraint Modeling", where we will point to various concrete examples, too.

We've now discovered the theoretical concepts behind product models. In the context of model-based product configuration (see Section 3.3.3.5, "Implementation Aspects"), these models are the most important sources of information available to the configuration system. Effectively, they provide the concepts to represent real, physical or immaterial products within a software application.

While we discussed product modeling on a theoretical basis here, in Chapter 4, *Methodology and Conceptualization* we are going to conceptualize a concrete modeling approach, that must encompass, respectively realize, the concepts worked out in this section. Hence, this section can be seen as the *requirements analysis* for the subsequent conceptualization.

Similarly, we will describe the process of product configuration on a rather theoretical level in the next section.

3.2. Product Configuration

In this section we will take a look at **product configuration** as a process (respectively as a course of action). We described product configurations (as a subject) already in Section 2.1.3, "Customized Products and Product Configurations" from a static, high level perspective. Here we will concentrate on the dynamic, information technical point of view.

3.2.1. Characteristics of Product Configuration

The act of *configuration* can be seen as design task between *selection* and *construction*. In this context, the distinction between supplier-centric, customer-oriented and customer-centric products introduced in Section 2.1.3, "Customized Products and Product Configurations" is relevant [Scheer2006, p. 32]:

Selection. In the case of *supplier-centric products*, the customer can solely search and select a specific product variant. All product attributes are fixed and their values cannot be customized. The solution space offered by the company is limited and relatively small.

Configuration. In terms of *customer-oriented products*, certain product attributes can be altered, while others are fixed. The value spaces of these customizable attributes (so called *parameters*) are usually pre-defined and limited. Nevertheless, during configuration, the customer chooses the exact value depending on his demands, which results in a tailored product. Due to the limited options, the solution space remains stable. From a supplier perspective, product development has been finished.

Construction. While customer-oriented products provide limited customization options, *customer-centric* products can be nearly arbitrarily adapted. In effect, the constructive task performed by the customer during the specification process enlarges the solution space, which therefore becomes virtually unlimited. The customer is actively involved in the product development process and the actual fulfillment process corresponds to a Develop-to-Order strategy (refer to Section 2.3.3.4, "Order Fulfillment Strategies").

In the context of product customization, the task of configuration is to specify the structure and concrete attribute values of a product, while the correctness of the model must be maintained, in order to ensure the manufacturability of the customized product.

Product configuration can thus be defined as follows:

Product Configuration Product configuration describes the composition of a product out of pre-defined components (so called *selection* and *combination*) and the specification of values for variable component attributes (so called *parameterization*). Thereby, the correctness of defined consistency rules (so called *constraints*) must be maintained. Configuration options result from selection, combination and parametrization of the product, restricted by constraints (cp. [Scheer2006, p. 41]).

In more detail, Scheer characterizes configuration as follows [Scheer2006, p. 40]:

- During configuration, pre-defined components can be composed, that is, selected and combined within the range of the configuration possibilities. Those configuration possibilities are restricted with the help of constraints.
- A component is described in terms of interfaces, attributes (including possible domain values) and constraints.
- During an active configuration, neither can concrete component instances be modified nor can new component instances be created (in the physical sense). Also, previously unforeseen component combinations cannot be applied.⁷
- The configuration result describes the entirety of selected components and their compositional relationship.

Characterized this way, a configuration task differs from a (constructive) design task in a way, that during construction, new component instances, combinations or attribute values can be added. In case of configuration, however, these operations are disallowed to maintain manufacturability without expensive re-designs.

In our conceptualization, introduced in Chapter 4, *Methodology and Conceptualization*, the distinction between *selection*, *configuration* and *construction* is a fundamental concept. Though, we apply these concepts not only on *product level*, but rather on *component level*: for a particular component of the product model it can be defined, whether the component is specified through:

Component Selection. The particular component variant is *selected* from the available set of those items. In this case, the customer in fact cannot add a new component variant nor provides the component any parameterization options.

Component Configuration. A base component is chosen but can further be specified through parameterization.

Component Construction. The customer can construct entirely new component variants within the range of the supplied configuration restrictions.

We will describe these (so called *specification methods*) in more detail in Section 4.4.3.1, “Specification Methods”.

While in this section, we characterized product configuration in general, in the next section, we will discover configuration processes in detail.

⁷In essence, by formulating these restrictions, Scheer complies with the in Section 2.2.6, “Mass Customization” identified requirement, that the solution space for customizable products should remain *stable*, in order to be able to manufacture them efficiently.

3.2.2. Product Configuration Processes

Product configuration processes can be described from numerous perspectives and on various levels of detail. In the literature, we found the following approaches, which we consider being useful to understand configuration from a process point of view (see [Maher1990, p. 49-50], [Businger1993, p. 18-19], [Scheer2006], [Reichwald2006a, pp. 123]):

Global Configuration Process. Describes the act of configuration from a high-level, macro perspective.

Configuration as Transformation Process. Describes the transformation of a (variable) product model into a concrete product configuration instance on a rather technical, lower level.

Interactive Configuration Process. Describes configuration from an interaction point of view, as perceived by the user of a configuration system.

In the following, we will cover these different perspectives one by one.

3.2.2.1. The Global Configuration Process

In case of *self-configuration*, the customer takes decisions regarding the given configuration options autonomously, that is, without assistance by some sales personnel. In this sense, all online product configurators (which we focus primarily in this work) can be considered to realize self-configuration.

We termed the self-configuration process characterized by Scheer in detail [Scheer2006, p. 42] with the name **global configuration process**, as it provides a macro perspective on configuration processes. Figure 3.5, “Global Configuration Process” illustrates the process.

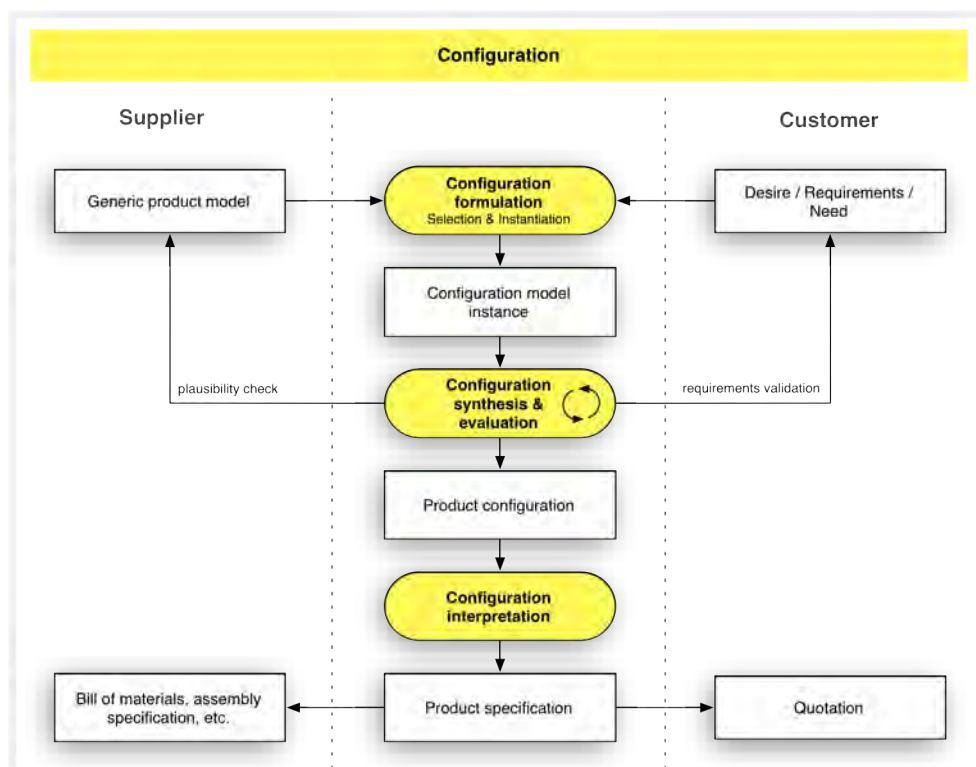


Figure 3.5. Global Configuration Process

Starting point. *Starting point* for the configuration is the customer, who wants to configure a custom product in order to satisfy a certain need. The customer owns an idea about the solution, which is accompanied by the customizable product offered by the supplier.

Configuration model. The configurable product is represented by a *generic product model* (also called *generic product structure* or *configuration model*), which defines the product components along with their attributes and domains (commonly referred to *product model elements*), their structure and related constraints. A specific instance of the generic product model corresponds to a *customer specific product model* (which is termed *configuration model instance* accordingly). The customer specific product model doesn't contain constraints itself, but rather holds the specifications of selected product components that are *consistent* with the defined constraints.

Configuration process. The (*actual*) *configuration process* comprises the selection and parametrization of components according to the given constraints. During the configuration process, the customer's requirements are transformed into a customer specific product model, that is, a concrete product composed of customized components. The transformation process includes the following phases⁸:

- **Configuration formulation.** Involves the identification and interpretation of customer requirements as well as their translation into goals and restrictions for certain configuration options. Concretely, during the configuration formulation phase, a generic product model is selected and instantiated into an initial customer specific product model. Additionally, global requirements are applied to the model in terms of domain restrictions. Moreover, default attribute values are set.
- **Configuration synthesis and evaluation.** Iterative sub process that aims to incrementally specify configuration options and subsequently validate the model against the constraints (*plausibility check*) and user requirements (*customer feedback*).
- **Configuration interpretation.** Deduction of final performance attributes, such as prices and delivery times, as well as generation of detailed specifications, including sales quotation, bill-of-materials and assembly descriptions, from the customer specific product model.

Result. The *result* of the configuration process is a customer specific product configuration in terms of a detailed product specification, that fulfills the customer's requirements and is consistent with the defined constraints.

The detailed definition of the process by Scheer, allows a precise dissociation of otherwise often interchangeably used words: the global, complex *configuration process* including all three phases is referred to as *configuration*. The technical software system that supports the full process is called *configuration system*. The subsystem of the configuration system that conducts the *configuration synthesis* is understood as (*actual*) *configurator*. The corresponding process supported by the configurator is called (*actual*) *configuration process* [Businger1993, p. 18-19].

Nevertheless, for simplicity reasons, those terms are often used interchangeably.

3.2.2.2. The Configuration Process as a Transformation Process

In essence, the product configuration process describes a **transformation process**: the generic product model, representing all possible variations of a customizable product, is incrementally transformed into a specific product configuration.

Figure 3.6, "Configuration as a Transformation Process" depicts the relationship between generic product model and a specific product configuration graphically.

⁸cp. [Maher1990, p. 49-50], [Businger1993, p. 18-19], [Scheer2006]

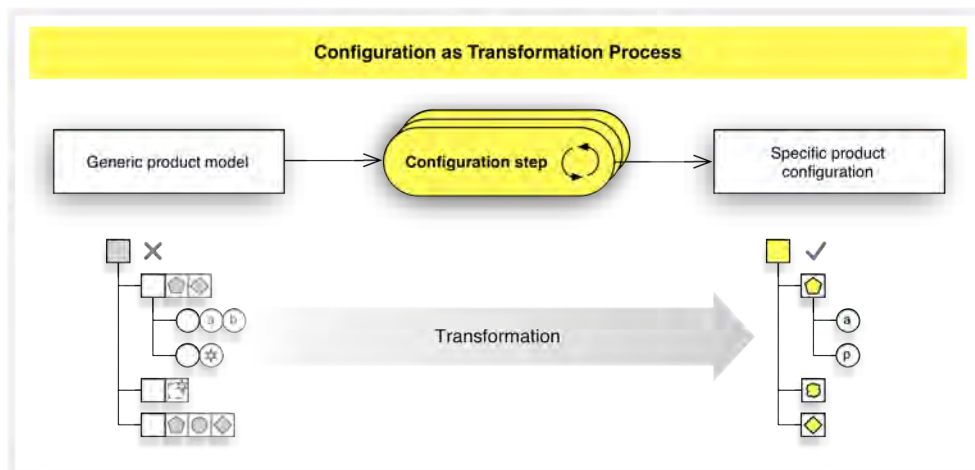


Figure 3.6. Configuration as a Transformation Process

The transformation process consists of multiple *configuration steps*. In particular, there is one configuration step for each configurable option (respectively configuration decision) of the generic product model. To recall the configurable options identified in Section 3.1, “Product Models”, the different kinds of decisions possibly be taken by the customer are:

- i. Component type decision
- ii. Component quantity decisions
- iii. Component variety decisions
- iv. Component customization decisions
- v. Attribute quantity decisions
- vi. Attribute valuation decisions
- vii. Constraint decisions

During each configuration step, the user specifies a value for the configuration option. Thereby, he further details the custom product's specification.⁹ Figure 3.7, “Configuration Steps” shows the internals of a configuration step, which are described below:

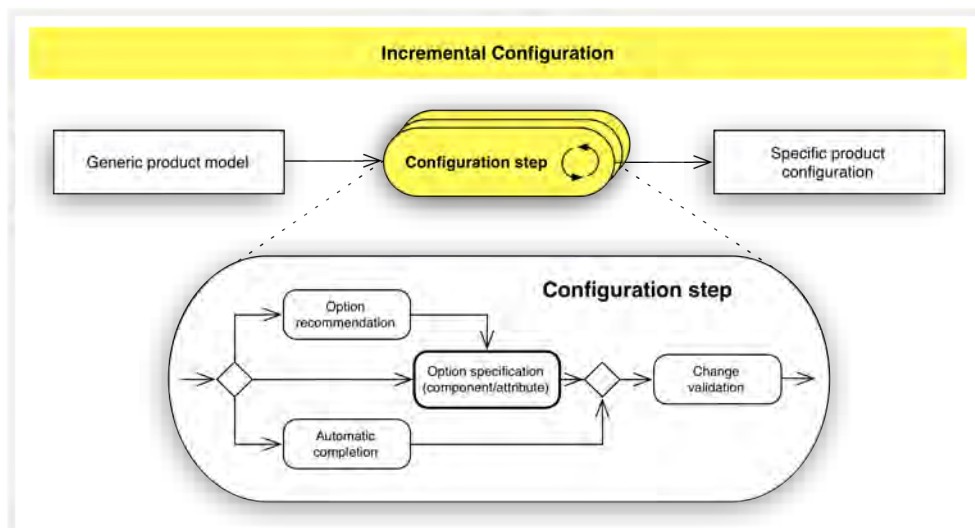


Figure 3.7. Configuration Steps

⁹In visual, interactive configuration systems, many decisions are combined within a single interaction step, which is why the configuration steps usually do not directly correspond to the application's screen flow. Also, some configuration decisions are even implicitly taken or multiple decisions are taken at once. This may happen, for instance, with the help of graphical component editors.

Option specification. As described above, the user specifies the value of a configurable option. For example, he selects a specific value from the domain of an attribute, chooses a pre-defined component or modifies the quantity of an already identified one.

Option recommendation. The configuration system might implement a recommendation feature that proposes a specific value for a configuration option. To generate a recommendation, configuration systems may implement different filter methods that realize algorithms known from recommender systems [Leckner2004]. The user must manually initiate the recommendation process and may choose to adopt or discard the proposed value. Also, the application of a *default value* for a configurable component or attribute can be considered a value proposal (see also Section 3.3.2.2, “Specification / Recommendation” and Section 7.2.2, “Recommendation Integration” for details).

Automatic completion. In advanced configuration systems, the user might choose to automatically complete an unfinished configuration or parts of it. For instance, the user specifies the most relevant configuration options of a component and leaves the rest of the configuration decisions up to the recommendation system. Moreover, in case the domain of an attribute is reduced to a single possible value due to the propagation of constraints, the attribute can be automatically specified with the remaining value.

Change validation. After the submission of a specific value modification, the verification of the configuration's consistency is triggered. Assuming that constraint violations occurred, the system may signal the error to the user or even intervene by rejecting the applied value entirely.

The configuration process is finished, when all constraints are satisfied and the product configuration sufficiently fulfills the customer's requirements.

All decisions taken by the user are recorded in the configuration result, a (*valid*) *product configuration*. The product configuration tracks components, their relationships and chosen attribute values. It can be stored and re-edited at a later stage and is transformed into a concrete product specification or related specification documents upon customer order.

3.2.2.3. The Interactive Configuration Process

Having described the configuration process from a macro (*global configuration process*) and a micro technical perspective (*transformation process*), we finally want to give an overview of the very same process from an interaction point of view, as perceived by the user: the *interactive configuration process* (see Figure 3.8, “Interactive Configuration Process”).

In essence, the process described below realizes the **configuration step** identified as part of the customer-supplier interaction in Section 2.3.5, “Marketing Aspects” cp. [Reichwald2006a, pp. 123].

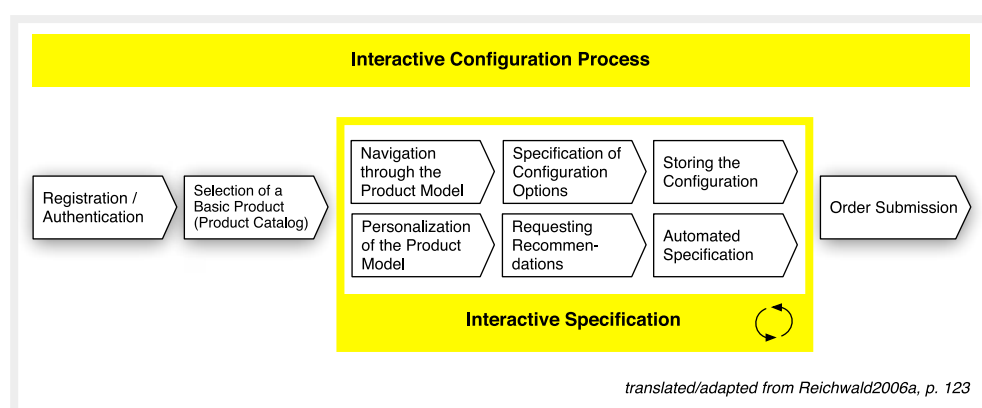


Figure 3.8. Interactive Configuration Process

The interactive process at some point in time involves the *registration* or *authentication* of the user. The authentication is an important step to identify the user and to elicit and store customer specific information. In fact, the registration of the user is the initial starting point for the realization of a learning relationship. Nevertheless, the configuration process itself can be performed anonymously and registration may be performed upon saving the configuration or submitting an order, thus after the actual product specification.

The transition from the exploration phase to the interaction phase of the interaction process is established by *selecting a basic product* from the product catalog for further customization. At this point, the **interactive specification process** begins. From a high level perspective, this cyclic process includes the following activities:

Navigation through the product model. The user may navigate through the product model in order to explore and specify the configuration options. While some configuration systems enforce a specific order (*sequential access*) of configuration items, others allow randomly specifying the different configuration parameters (*random access*).

Personalization of the product model. In advanced configuration systems, the configuration model can be personalized, e.g., by requesting customer preferences and mapping them to model constraints. For instance, the configurator may ask the user to specify his favorite color and pre-select the stated color in any subsequent color related decisions.

Specification of configuration options, Requesting recommendations, Automated specification. The essential activity of the interaction process, the specification of values for configuration options, has been described in detail in Section 3.2.2.2, "The Configuration Process as a Transformation Process". There we also mentioned, that the specification action can be supported by the configuration system by generating recommendations or automatically filling values.

Storing the configuration. Another important step is the persistence of the configuration. The specification of a complex product may last several hours. Hence, the process is often temporarily interrupted by the customer and resumed at a later point in time. To accommodate this scenario, the configuration system should be able to capture and persist the exact configuration state and restore it later on.

The interactive specification phase ends when the customer successfully specified a valid product configuration and places an order. At this point, the specification of the custom product is finished and the start of the supplier's internal fulfillment process is triggered.

It is the interactive configuration process that is technically supported by a *configurator*, which helps the customer to perform the activities discussed above. In the next section, we will take a look at these configuration systems in detail.

3.3. Product Configurators

In this section, we will concentrate on product configurators as software tools. We will again recap their role within a product customization scenario and the describe tasks and requirements applying to them. Next, we will work out a set of features, realizing these tasks. A comprehensive categorization of configuration systems will finally not only describe the features in more detail, but will also stand as a sophisticated comparison matrix for different configuration approaches.

The Role of Configurators within Product Customization

Product configuration systems are a key enabler for mass customization [Bourke2000]. They play a fundamental role in the interaction process between supplier and customer by supporting the "design process" of a customized product, especially in the context of electronic commerce [Rogoll2004].

As illustrated in Figure 3.9, “Product Configurator as Key Enabler for Mass Customization”, the product configurator plays a central role in the product customization scenario. It can be seen as the common interface between customer, supplier and the supplier's offered product range.

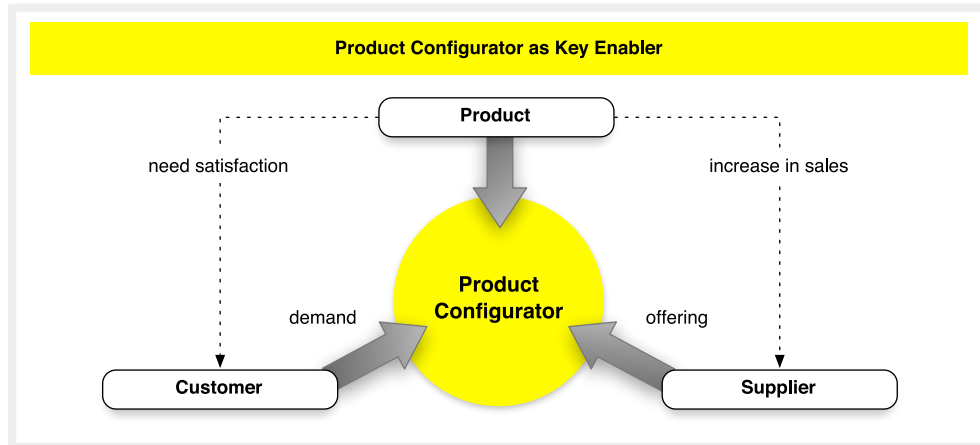


Figure 3.9. Product Configurator as Key Enabler for Mass Customization

Into the direction of the customer, the configurator presents the supplier's product range and overall capabilities. In fact, in the case of self-configuration, the configurator acts as a sales utility in replacement of a human sales employee and thus represents the entire company on behalf of the supplier. The configurator must not only communicate the product's features adequately, but instead should motivate the user to explore, configure and after all *buy* a product. Thus, the appearance, usability, interactivity, and use of multimedia-based content are important factors to mediate a virtual buying experience to the user [Reichwald2006b].

On the other hand, the configurator enables the communication of product requirements from the customer to the supplier. It thus allows the mass customizer to access the *sticky information* (see Section 2.3.4.3, “Profit Gain Through Product Customization”, i.e. [Reichwald2009, p. 241]) owned by the consumer.

Origins of Product Configuration Tools

During the past 20 years, configuration systems evolved from various areas of IT. Originating fields of application include [Reichwald2006b, p. 32]:

CAD systems. *Engineering driven configurators* were introduced as part of computer aided design tools. The focus of these configurators, which are primarily used offline by engineers and sales experts, is the detailed technical specification of physical products and their technical visualization.

ERP systems. Often configuration software is introduced as part of enterprise resource planning suites. They're also used mostly by sales engineers. *Production driven configurators* support the automated generation of bill of materials and work plans. Tightly integrated with the ERP system, quotes can be generated containing detailed pricing information and delivery times. Upon order submission, production orders are automatically scheduled.

E-Commerce. With the emergence of electronic commerce in the mid 1990s, purely *sales driven configurators*, that targeted the consumer directly over the world wide web, evolved. E-commerce configurators are embedded within the company's corporate website and integrate seamlessly with other sales tools such as online shops or CRM software. In this regard, the user experience and the usability in terms of "ease of configuration" are exceedingly relevant aspects.

In practice, today mixed forms of the above types of configurators are common. In the context of our work, we primarily focus on sales configurators used by end users or field sales staff over the world wide web. However, we pay attention to integration aspects related to CAD and ERP use cases.

Definition of Product Configurators

"A *product configurator* is a tool which supports the product configuration process so that all the design and configuration rules, which are expressed in a product configuration model, are guaranteed to be satisfied. The configurator simplifies the manufacturing process by assuring that all orders received are possible to build. Interactive configurator tools can support quick and flexible customization by giving immediate and accurate information about the available combinations of options" [Hedin1998, p. 107].

Similar to product catalogs, product configuration systems are software systems applied in the area of computer aided selling (CAS). While *product catalogs* solely support search and selection of pre-defined products, *product configurators* additionally allow the individual combination and parametrization (configuration) of components in order to design a custom product.

These definitions already provide a basic idea on what a configurator is. However, in order to implement a framework for creating custom configurators, a more detailed definition of possible and required features must be worked out. We'll summarize the tasks of configurators and the requirements on such systems in the following.

3.3.1. Tasks and Requirements

Configurators are complex systems that must fulfill lots of tasks and requirements. Again, a configurator is a design tool for specifying custom products that aims to ease and automate a mass customizer's sales process. Thus, the core task of a product configurator is the presentation of components, attributes and attribute values defined in the product model and the capturing of composition and parametrization decisions. Thereby, the system must guard the consistency of the configured product [Scheer2006, p. 27].

However, there are more responsibilities to be taken by product configuration systems. In the literature, manifold lists of tasks and requirements are available¹⁰. Nevertheless, in our opinion these descriptions often mix different aspects or terms and hence do not provide a clear picture on the feature set commonly implemented by configuration systems.

We try to provide an aggregated, comprehensive description of the main tasks and high-level features of a configuration system here. We use the following interpretations of the terms *task*, *feature* and *requirement* as a basis:

Task. Describes an ability or a responsibility, that the configurator must fulfill, i.e. explains *why* the configurator does something.

Requirement. Describes a condition, that further characterizes *how* the configurator performs a task. When implementing a feature, the requirements related to that feature must be considered.

Feature. Describes a functionality of the configurator, that realizes a task, i.e. explains *what* the configurator does to fulfill a task.¹¹

¹⁰See [Piller2003b, pp. 26], [Rogoll2004, pp. 26], [Scheer2006, p. 33-34, p. 49-50], [Reichwald2009, pp. 282], [Krug2010, pp. 24]

¹¹To provide a short example: the configurator should inform the customer about the supplier's products (*task*). It does so by providing a product browser (*feature*), that allows to explore the product range and display product information. The information presented should be aided by multimedia features to provide an optimal, sales encouraging user experience (*requirement*).

3.3.1.1. Tasks of a Configurator

We identified four main task areas, for which Figure 3.10, “Tasks of a Configurator” summarizes the most important responsibilities. We describe them in more detail, together with additional requirements, below. The findings in this section have been aggregated from various literature sources, including [Rogoll2003, pp. 23/37], [Scheer2006, pp. 33-34/49-50], [Krug2010, pp. 24].

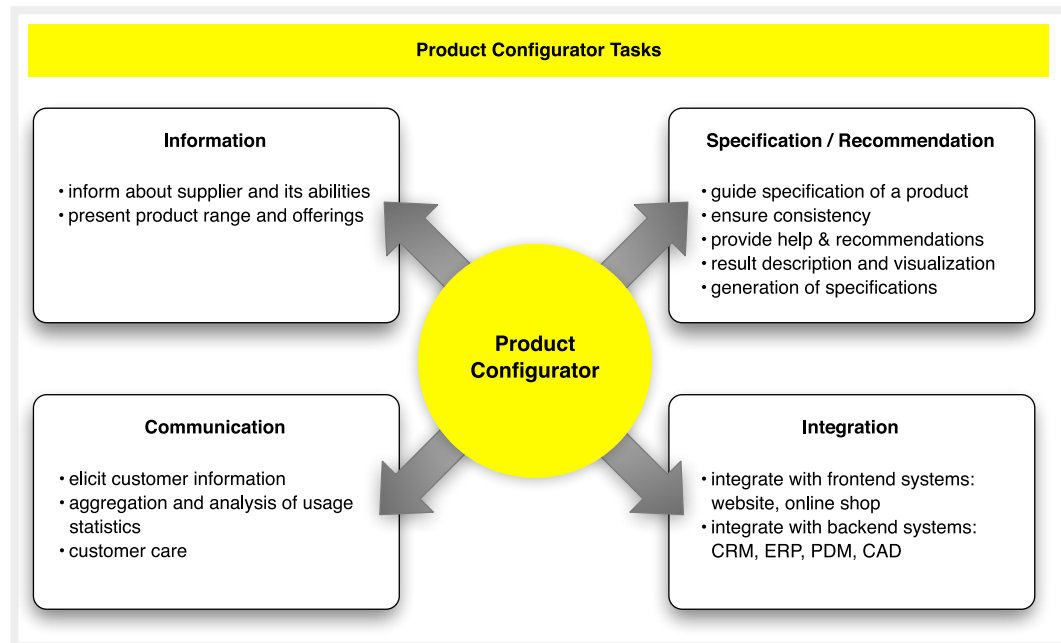


Figure 3.10. Tasks of a Configurator

Information

The configurator is responsible for providing information about the supplier and its capabilities, including detailed product and pricing information.

The information should be comprehensive, presented clearly, aesthetically appealing and is optimally backed with multimedia content. Sophisticated product and supplier information can build trust and significantly reduce the customer's perceived risk about the configured, not yet physically tangible product.

Specification / Recommendation

The main task of the configurator is to support the customer during the specification of the custom product, thereby, automating the supplier's selling and consulting process. Starting with the selection of a base product, through the presentation of configuration options and reception of customer decisions, the configurator should guide the customer through the configuration process, while not necessarily imposing a fixed decision sequence. Additionally, the system should support the customer by providing context sensitive help information, option recommendations or automatic completion methods upon customer request. In parallel, the configuration's consistency must be continuously checked. The result should be explained at least textually, include up-to-date pricing information and, if possible, should indicate delivery times. Optimally the configured product is also graphically visualized to give the customer immediate feedback about his doings. The final configuration should be detailed enough to let the configurator generate a sales quote, bill-of-materials and other production relevant specifications.

While these are the basic tasks related to product configuration, there are additional requirements on how these tasks should be fulfilled: first of all, the produced configurations must

be complete and correct. To reduce the overall complexity, the configurator must ease the configuration process as much as possible and allow the customer to configure a product in real-time. This optimally includes the a priori restriction of selectable options to those that result in valid product configurations. Offering options that lead to inconsistent or implausible specifications cause frustration on the customer's side and may even let the customer abandon the configuration process entirely. In this context, also the user-friendly, customer oriented design of the process, as well as low latency and quick response times are of significant importance. At best, this leads to a "flow" experience, motivating and engaging the user to participate in the co-design process with enthusiasm (cp. [Rogoll2003]). Furthermore, the user should be able to control the configuration process by skipping, postponing or changing configuration decisions. For the specification of complex products, which may require several minutes or even hours, it's important to establish a mechanism to store, pause, load and resume the configuration process at a later point in time.

Communication

The configurator has another, very important responsibility: it must realize the entire communication activities during the selling process on behalf of the supplier (see Section 2.3.5.2, "The Selling Process (Supplier Perspective)"). A configurator can be understood as a major sales supporting tool, that realizes a complete, customer facing information system. In particular, the configurator must elicit or request information about the customer in order to adequately respond to his requirements. Moreover, the configuration software should optimally support the collection, aggregation and analysis of usage statistics in order to continuously align the company's offerings to the market's demand. The gathered information about the customer and his requirements are used to build a customer profile, which is the basis of subsequent, personalized communication activities in the sense of a learning relationship between supplier and customer.

Often, online users avoid revealing personal information, such as personal data, individual preferences or billing information. Especially, in the context of product customization, whose success strongly depends on these information, it is essential for the supplier to build trust and communicate competency. The thorough handling of customer data, including an unrestricted admission to adhere to privacy policies is important. A personalized, customer-tailored communication with the user as well as the ability of the customer to personalize the configuration process and its interface, positively influences trust and the identification of the customer with the company and the configured product.

Integration

To successfully deploy product customization, the configuration system must be well integrated into the companies business processes and other IT¹² systems. Not only must the configurator support the automated generation of sales quotes and optimally other production relevant specification documents, such as bill-of-materials, work plans, assembly descriptions and detailed product specifications. Instead, it's also the task of the configurator to integrate with other customer-facing systems, such as the corporate website and the online store to enable immediate customer orders. After receiving a purchase order, the configuration software must feed the configuration result and related data into other backend systems, such as ERP systems for order fulfillment, CRM for storing customer information and PDM¹³ systems for archiving custom product variants. During the configuration process, the graphical display of the product may require the integration of CAD systems to provide detailed technical drawings or other elaborate visualizations (e.g., 3D models).

From a supplier perspective, it's desired that the configuration software seamlessly integrates with the other systems of the IT landscape to establish a smooth, error-free and fully automated specification and sales process.

¹²Abbrev. Information Technology

¹³Abbrev. Product Data Management

3.3.1.2. Additional Requirements

While primarily the *task-related* aspects of information, specification, communication and integration described above are relevant from a business perspective, there are other very important requirements from both the customer and supplier point of view.

From a **customer perspective**, the *ease of configuration* is of primary interest. The complexity reduction of the configuration process even for complex products and the realization of an attractive user experience demands a great deal of usability. The ability of the customer to control the configuration process as well as the feeling of being able to actively design the configured product, creates enthusiasm and motivates for purchasing the product.

On the **supplier's side**, there are additional requirements related to the *administration* of the product configuration system. In general, the configuration software should be independent of the modeled product. This implies a separation of product and application logic and consequently improves the overall *maintainability* of the system, which is one of the most important requirements after all. Regarding the development of the product configurator, the modeling language used to describe the configurable product must be flexible and powerful enough to adequately represent the company's problem domain. During operation, the maintainability of the configurator's knowledge base is a crucial aspect. The configuration system should allow to easily incorporate new product variations and other information.

Implementing the configurator as an online application accessible over the world wide web, that centrally stores the configuration knowledge, enables distributed, collaborative work. In this context, the implementation of durable, persistent sessions are a challenge when realizing the configurator as a web application. Other **technical requirements** are an overall high system stability and fault tolerance in case of network problems. Integration and extension capabilities may become important aspects in the long run.

3.3.2. Features

In the literature, the specification of functionalities in terms of concrete product features of configurators is missing. Instead, the authors only provide a high level overview of the tasks of a configuration system as described in the previous section.

In the following, we will try to provide a basic set of features commonly implemented by configuration system and match them with the task and requirements stated above.

Figure 3.11, "Overview of the Main Features of a Configuration System" gives an overview of the features explained below.

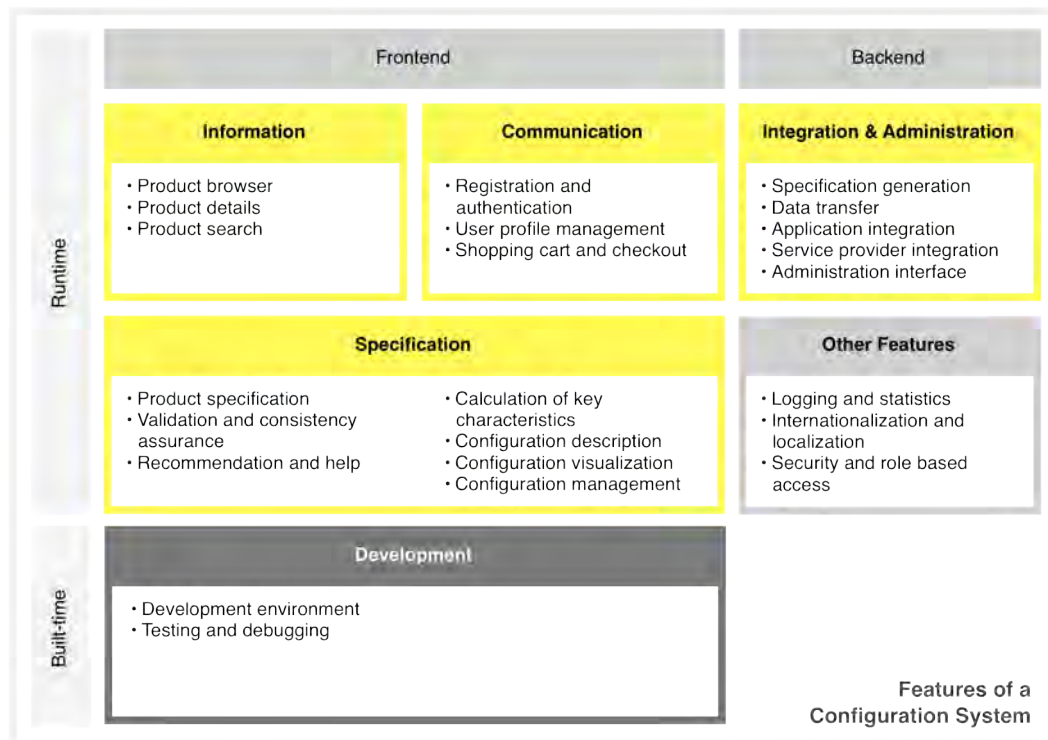


Figure 3.11. Overview of the Main Features of a Configuration System

We group the task specific features according to the categories introduced in the last section, although some of them might be relevant in other categories, too. Moreover, it's useful to define the different global scopes of the features:

Runtime. Encompasses all features that are relevant, when the application is deployed in production.

- **Frontend.** Contains features relevant to the customer during the product configuration process.
- **Backend.** Describes those features related to the operation and administration of the configurator.

Development. Comprises features and tools used during development of the configurator.

Usually, the frontend scope is meant when we talk about the *features of a configurator*, while we mean both the runtime and the development scope, that is, the full system, when talking about the *features of a configuration system*.

3.3.2.1. Information

The following features are the main ones addressing information related tasks and requirements, as described in Section 3.3.1.1, "Information".

Product browser. The *product browser* allows to navigate through the company's offered product space. In effect, this feature corresponds to a basic *product catalog*. From within the product browser, the user selects a base product or product type to start configuration from. Usually, the navigational structure is organized by product families, model range and individual product types. Individual product modules may additionally be described in parallel to the product family structure.

Product details. The product browser features an area to display *product details*. In this area, elaborate product descriptions, specifications, images and other related information including pricing details are presented. An important aspect of the product details area is the product visualization part. In the case of invariant products or components the visualization usually is realized with images, 2D drawings or 3D models. Otherwise, when a configured product is displayed, the visualization is generated according to the user's specification. We will describe the visualization of configured products in Section 3.3.3.4, "Configuration Characteristics", "Visualization".

Product search (optional). An optional extension to the product browser is a *search or filtering functionality*, that allows to query the product space using different criteria. Not only during the selection of the base product, but also when selecting a specific component during the configuration process, a search may ease the process of finding the right component significantly. A search feature improves the configuration experience for advanced users and drastically speeds up the process for experts.

3.3.2.2. Specification / Recommendation

In order to fulfill the specification and recommendation tasks and requirements discussed in Section 3.3.1.1, "Specification / Recommendation", a typical configurator provides the following features.

Product specification (dialog). The essential feature of a configurator is the *product specification (dialog)* component. Having selected a base product in the browser, control is transitioned to the specification dialog, which is in charge of executing the configuration process, that is, requesting all required decisions from the user and storing them in the configuration model. In general, one can differentiate two approaches for realizing the configuration dialog:

- **Form/option list based specification.** Basically, allows the specification of the custom product by presenting textual decisions in terms of form fields (drop-down lists, radio groups, check boxes, input fields, etc.) or option lists (stacked lists from which the customer selects a single item per list). Also visual controls, such as sliders, graphical buttons and others are possible. In contrast, the interactive specification of values within the visualization is not covered. Form based specification is especially useful for the specification of *obligatory elements*, *optional elements*, *principle solutions* and *services* (see Section 2.1.3.1, "Customizable Areas").
- **Visually interactive specification.** Supports the interactive, graphical product specification within the product rendering or a graphical variant of the same (e.g., a 2D perspective of the 3D visualization). Especially, when the product comprises *defined expansion spaces* or *scalable areas* (see Section 2.1.3.1, "Customizable Areas"), interactive specification can be applied reasonably. The specification in this case is supported by custom design tools, that usually realize *point and click* as well as *drag and drop* behavior. They're mostly implemented for specific use cases only.

A configuration process may even combine both variants of specification depending on the part of the product.

Furthermore, we can distinguish two basic forms of process control as realized by the configuration dialog:

- **Pre-defined, sequential control flow.** The navigational path taken by the user to specify the configurable options of a product is pre-defined by the system. In this case, the configurator sequentially presents the options one by one, possibly even hiding subsequent decisions entirely. However, although the control flow is pre-defined, the application must not necessarily disallow random access completely: the user can point options earlier and possibly also later in the process freely and then return to the regular control flow. A pre-

defined, sequential flow offers great guidance possibilities and signifies a simplification of the process for non-expert users.

- **Random access control flow.** Expert users, however, usually prefer a more compact, randomly accessible form of specification. The specification dialog must ensure quick navigability through the product model and must not constraint the order of option specifications.

We will describe these control flows in more detail in Section 3.3.3.4, "Configuration Characteristics", "Process scheme".

Validation and consistency assurance. The configuration system must implement a feature to *validate and ensure consistency* of the configuration. On the one hand, the configuration dialog can solely present valid options according to the current configuration state (**a priori consistency**). On the other hand, some decisions cannot be restricted to contain only valid options a priori due to logical or technical reasons. E.g., a string text field may be restricted to contain 60 characters, listing any possible combination of characters is of course neither feasible nor technically possible with sufficient performance. In such cases, validation must occur *after* the user specified the value and conflicts must be signaled accordingly (**ex post validation**).

Note that in the case of advanced implementations of a priori consistency, which effectively constrains the configuration decision's solution domain, it might still be possible to select a disabled value. In that situation, the configuration system must *repair* the configuration, i.e. recursively backtrack and recommend a change-set that satisfies all constraints including the given value. Repair operations are considered an advanced use case.

Moreover, the employed consistency algorithm should optimally not only implement **local consistency**. That means, a given value is valid to the locally defined constraints, disregarding whether a valid solution for the entire configuration problem exists, that contains that value. Instead, it should optimally ensure **global consistency**: if the user chooses a specific option, the configurator *guarantees* that with each particular decision the configuration is *completable* without errors. Obviously, global consistency is harder to implement technically.

Recommendation and help (optional). During the configuration process, the configurator may give advice to the user who's possibly overstrained to take a decision. In this case, offering *recommendation and help techniques* should be considered. One can differentiate between active and passive forms of support.

Active support is triggered or performed intrinsically by the system. For instance, the system may automatically recommend an option based on a user's previous choice or due to his personal preferences. The application of default values and the application of change-sets as a result of repair operations (see above) can also be considered a form of active support. Though, the automated selection of options (except in case of default values) should be signaled to and confirmed by the user to convey controllability – an important usability aspect.

In contrast, *passive support* means, that the user himself initiates the recommendation process, requests help or looks up additional information on a specific option. The provisioning of context sensitive help for configuration decisions, e.g., in terms of tool tips, is considered the minimum requirement for the implementation of a basic help functionality.

Calculation of key characteristics. In parallel to the configuration process, the system should *calculate key attributes* of the configured product, such as price, delivery time, weight, total dimensions or application specific values, and appropriately present them to the user as part of the configuration description.

Related to the display of *pricing information* for selected components, it's reasonable to distinguish two different strategies: total price display or premium price display. For each selectable component, *total price display* shows the full price of the configured product as if the component was part of the configuration. In terms of *premium price display*, only the ad-

ditional charge compared to a default price is displayed beneath the selectable option (cp. [Polak2008]).

The calculation of key characteristics, especially pricing and delivery time, are important factors for the buying decision from a customer point of view. Nevertheless, often only an approximate value can be compiled during the configuration process. For example, the delivery time cannot be provided definitely unless the configured item is scheduled for production, that is, unless the configuration has been fed to the PPS¹⁴ system.

Configuration description. An essential part of the product configuration dialog is the *description of the configuration*, which is presented in tabular and/or textual form. It lists selected components, important attributes and key characteristics. The configuration description must be detailed enough to explain the configuration approach and must immediately be synchronized with the configuration upon changes. It should precisely reflect the customer's choices, because the user will review his decisions based on the configuration description prior submitting an order of the configured product.

Configuration visualization. Due to the fact that in terms of custom products, the customer cannot tangibly experience the configured item, the correct and realistic *visualization of the product configuration* plays a decisive role. The graphical visualized product can be seen as a surrogate for the real product, which considerably influences the configuration experience and the overall buying decision [Rogoll2003, p. 58-59]. For a more detailed discussion on visualization options, refer to Section 3.3.3.4, "Configuration Characteristics", "Visualization".

Configuration management (optional). Another useful feature from the customer point of view is the option to persistently save a configuration and load it again at a later point in time. This is especially relevant for complex products, that require several minutes or even hours to be configured. The *configuration management* feature allows the user to create multiple configurations and is completed with the ability to delete unused ones.

3.3.2.3. Communication

The next features, are merely used to implement the communication related tasks and requirement described in Section 3.3.1.1, "Communication".

Registration and authentication. An important aspect of product customization is the direct relationship between customer and supplier. In order to identify a particular customer, a *registration and authentication feature* must be implemented.

During registration, which may be conducted either before or after the actual configuration process, the customer initially provides personal data to the supplier. The system stores this data in the user's profile. In subsequent configuration processes, it is sufficient for the user to authenticate himself with his credentials in order to match his identity to the earlier created profile. After being authenticated, all customer specific information elicited during configuration are stored with the user's profile.

User profile management. As the customer provides personal data to the system during registration and configuration, he must be able to *manage his user profile data* at a later stage. This includes dropping his account entirely, which should result in all customer specific data being removed from the system. Optimally, all elicited information can be accessed through the user profile management interface. This feature strongly mediates information transparency. Consequently, it can be considered an essential one to build trust. This, in turn, is a major prerequisite for a long term customer-supplier relationship. The stored user profile is the basis for a personalized communication with the customer.

Shopping cart and checkout (optional). Optimally, the customer cannot only configure custom products, but also directly order the configured items. To support this, order capabilities by means of a *shopping cart and checkout functionality* are required. In practice, these

¹⁴Abbrev. Production Planning and Scheduling

features may be provided by an external e-commerce application, the configuration system is integrated with.

A shopping cart can be used to store one or multiple configured items until the user's session expires or even persistently, which requires the user to be registered and logged in. To finally allow the customer placing an order, the configurator must additionally provide a *checkout mechanism*, which again integrates with another external provider that realizes on-line payment.

This basically completes the category of frontend related features. The remaining functionalities of a configuration system target the backend respectively the development environment for configurators.

3.3.2.4. Integration and Administration

The following are the main features that realize the tasks and requirements presented in Section 3.3.1.1, "Integration".

Specification generation. One of the most important features of the configuration system with respect to the automation of the sales process is the *generation of specification documents*. In fact, the main purpose of installing a configurator is to ease and automate the creation of such documents. Thereby, specification errors are reduced significantly and quotation creation, which is otherwise considered a routine work on the supplier's side, is speed up immensely.

Moreover, during the product's lifecycle a huge variety of specification documents needs to be created at different stages, including (see [Hvam2008, p. 18]):

- **Identification of need.** Sales quotes, order / product specifications.
- **Product design.** Drawings, lists of parts, strength calculations.
- **Production preparation.** Bill of materials (BOM), list of operations, process description, setup instructions.
- **Planning.** Production / work plan.
- **Production.** Registered use of time and materials, quality data.
- **Delivery.** Transport specification, assembly instructions.
- **Use.** User manual, service manual.
- **Disposal.** Specification of destruction.

Especially sales quotes, product specifications, drawings, bill of materials and work plans are of primary interest to be generated from product configurations. The resulting documents are the basis for subsequent activities in the order fulfillment process and are thus of high importance.

Data transfer (push). The information technical integration of follow-up activities is achieved by feeding the data from the configuration system into other information systems of the enterprise's system landscape. Hence, the configuration system must incorporate features to *transfer of customer and configuration data* to customer facing systems as well as supplier internal systems upon order submission, such as:

- **Customer related systems.** Corporate website / portal, e-commerce platform, CRM system, service portal, etc.
- **Production related systems.** ERP, PPS, PDM, logistics platform, etc.

As it is the configuration system, that initiates the data transfer, effectively a *data push approach* is realized. In state-of-the-art environments, communication with other systems is established by webservice calls, leveraging XML¹⁵ as data format. Nevertheless, a wide variety of other data formats for communication with business systems exist, for instance [Krug2010, p. 35]:

¹⁵Abbrev. eXtensible Markup Language

- EDIFACT (Electronic Data Interchange For Administration, Commerce and Transport),
- openTRANS (open standard to support data exchange in business transactions),
- cXML (Commerce eXtensible Markup Language), or
- xCBL (XML Common Business Library).

Application integration (pull, optional). Modern information systems also provide an interface to support the *integration with other applications*. By providing a webservice interface or an application programming interface (API), other systems can access the configuration system directly, which effectively realizes a *data pull approach*. The operations offered through these interfaces may range from methods to access customer and configuration specific data to operations, that allow full programmatic control over the configuration process.

Service provider integration (pull/push, optional). Beyond an interface for external applications to integrate with the configurator, modern systems also provide a service provider interface (SPI) that allows to connect to other information systems. Through the SPI the configurator accesses underlying data sources, while read and write operations, respectively pull and push access, are possible. In the context of product configuration, integration with the following systems are reasonable:

- **Databases.** To store configurator internal data.
- **PDM, CMS¹⁶, DMS¹⁷, asset management systems.** To access product specific data and additional product-related documents or assets including supplier related information.
- **CAD systems.** To create technical drawings and other product visualizations.
- **ERP, PPS systems.** To retrieve pricing information and for the calculation of delivery times.
- **CRM systems.** To access customer information.
- **Other technical systems.** Such as single sign-on (SSO) solutions or directory services to avoid authentication with multiple systems and to access user specific information, e.g., permissions, preferences etc.

Administration interface (optional). In order to access and manage submitted configurations, customer information, product data and usage statistics, the configuration system's backend may provide an *administration interface*. The administration interface allows to access these informations using a secured, graphical user interface.

In case the configuration system doesn't come with an administrative interface, the data must be transferred to other backend systems directly, using data transfer mechanisms or through SPIs.

3.3.2.5. Other Features

Beyond these task specific features, today's systems usually provide a number of other features, relevant across multiple aspects of the application. These *horizontal features* include, but are not limited to the following:

Logging and statistics (partly optional). Throughout the configuration process, the configurator should *log information about the customer's decision* for different purposes within a configuration journal or log file. On the one hand, the configuration journal can be used to track decisions, which is a necessary requirement for repair operations. On the other hand, the journal can be used to explain the configuration and may offer valuable hints about the customer's behavior from a business perspective. Also, the logging of usage statistics is required for subsequent analysis and data mining activities.

Internationalization and localization (optional). In times of globalization, products are often sold in multiple countries around the world. By incorporating *internationalization and*

¹⁶Abbrev. Content Management System

¹⁷Abbrev. Document Management System

localization features, different languages as well as country specific circumstances can be taken into account adequately. This concerns not only product related information in particular, but also the user interface of the configuration system in general.

Security and role based access (partly optional). Related to the configurator's frontend, advanced systems may allow different users to access a different level of detail according to their role assigned by the system. For instance, an expert user might see more / other options to choose from. Or he might see additional sources of information, while a regular user is only allowed to access standard information. This is usually implemented by a role based access model, that integrates with the authentication mechanism. In general, registration, authentication and in some cases even the configuration process itself must be secured, e.g., using data encryption mechanisms.

Concerning the application's backend, different roles may allow different permission levels. For example, a certain permission level may permit access to already created configurations for sales people but may forbid modifications to product information, such as prices, which are subject to sales managers only.

3.3.2.6. Development

Finally, there are features not directly related to the functioning of configurators, but related to their *development*. We want to present the two most important ones from our perspective here:

Development environment (optional). Due to the fact that a configuration system mostly is not realized like a "traditional" standard software, in practice, often a dedicated *environment for the development and maintenance of the configurator and its knowledge base* is required. Such a tool helps configurator developers to model the product range, define constraints, incorporate product data into the application, design the configuration process, style the configurator's interface and much more.

Testing and debugging tools (partly optional). As part of the development environment or integrated within the configuration software, the system may provide *testing and debugging facilities*. Although not necessarily required, with automated tests the correctness of the configurator and its knowledge base can be continuously verified. In case of defects or unexpected application behaviors, debugging tools can help to identify the cause of the error. Debugging capabilities are required in almost all configurator implementation projects, regardless of the product domain's complexity.

This finishes our explanation of configuration systems from a task respectively feature perspective. As one can observe from the number of features discussed (23 in total), implementing a sophisticated configuration system from scratch isn't a trivial, quickly achievable thing. Reasonably, a configurator is implemented based on a well-designed, well-tested and well-integrated platform that provides the majority of features out-of-the-box. On top of this platform, manifold features can be realized for numerous use cases at once, provided that the configuration platform is implemented in a generic manner. This saves time, effort and money. Our framework, introduced in Chapter 4, *Methodology and Conceptualization*, aims to implement such a generic configuration system platform for exactly those reasons.

3.3.3. Categorization

Before we plunge into the details of our approach, though, we will discuss various categories, that configuration systems are usually compared against during product evaluations. While the existence of a feature solely states that a certain functionality is *available*, the categorization identifies *how*, respectively *in which manner*, the particular feature is implemented. Thus, identifying and discussing these categories gives an idea of the solution space for configurator implementations in general.

Identifying these categories doesn't only allow to compare configuration systems with each other, but also helps to describe the implementation status a single system. So, the categorization allows to precisely characterize a configuration systems capabilities.

In the literature, many taxonomies, explaining various categorizations, can be found¹⁸. In most cases, the concept of a *morphological box*¹⁹ is used to display the variety of characteristics.

We will provide a detailed categorization of configuration systems, that has been compiled from all categories provided by the six literature sources stated above, in this section. We also utilize a morphological box to compactly cover the enormous diversity of different capabilities. The resulting taxonomy is shown in Figure 3.12, "Morphological Box of Configurators".

For easier perception, we partition the categories into five sections:

Application context. Concentrates on the application area and the business context of configuration systems. Moreover, the application context provides a high level overview of *why* and *where* the particular configurator is respectively can be employed.

System environment. Contains categories to characterize the environment of the configuration system in which it is embedded to. This includes a characterization of the system's overall architecture and main integration aspects.

Modeling capabilities. Gives a basic overview of modeling features provided by the configuration system, to design the customizer's product range.

Configuration characteristics. Covers categories to precisely describe the specification related aspects of a configuration system. From a business perspective, the product specification is the most important task of a configurator. Due to the large amount of items in this section, it's further divided:

- **Configuration approach.** Characterizes the overall configuration approach and the specification capabilities from a high level perspective.
- **Interaction characteristics.** Contains categories to describe the interaction approach employed in the configurator.
- **Configuration procedure.** Discusses the support for specification activities provided by the configuration system in detail.
- **Presentation Categorizes.** Describes alternatives related to the presentation of the configuration and other visualization features.

Implementation aspects. Provides details about the implementation and general configuration approach.

We will describe the characteristics in detail below, pointing to related features introduced in the previous section where appropriate.

¹⁸See [Blecker2004, p. 26-30], [Leckner2006, p. 49-52], [Scheer2006, p. 51-57], [Reichwald2006b, p. 30-33], [Renneberg2010, p. 75-80], [Krug2010, p. 39-60].

¹⁹The idea of morphological boxes goes back to Zwicky, who introduced them for the first time for the structuring and efficient presentation of ideas [Zwicky1966].

Application Context									
Product nature	immaterial product				material product				
Business area	business-to-business (B2B)				business-to-consumer (B2C)				
Site of operation	internal				external				
					point-of-sales (POS)		web		
Target audience	staff				customers				
Integration context	sales		development/engineering		production		hybrid		
Sales support	supplement				replacement				
Underlying order-fulfillment strategy	assemble-to-order		build-to-order			develop-to-order			
System Environment									
System interaction	offline				online				
					thin client		fat client		
System organization	centralized				distributed				
Integration scenario	none/standalone		embedded			integrated			
Integration level	not integrative		data integrative			data and application integrative			
			provider			consumer			
Target device	desktop computer/laptop		touch device/tablet			mobile phone/smart phone			
Modeling Capabilities									
Modeling capabilities	fixed areas	obligatory/ alternative elements	optional elements	scalable areas	principle solutions	services	defined expansion spaces	general expansion spaces	
Level of detail	limited					unlimited			
Product model evolution support	none			limited			full		
Configuration Characteristics									
Configuration Approach									
Specification approach	product-centric/structure-oriented			customer-centric/function-oriented			hybrid/combined		
Configuration capabilities	module configuration (generic modularization)		mixed configuration (quantitative modularization)		adaptive configuration (custom modularization)		design configuration (free modularization)		
Interaction Characteristics									
Configuration procedure	batch				interactive				
Process scheme	pre-defined/fix	structure-oriented	user-oriented		data-driven	case-driven	adaptive		
Dialog style	text based				graphical				
	questionnaire		forms						
		single screen	screen sequence	hierarchi- cal					
Configuration Procedure									
Starting point	blank configuration		pre-defined base configuration			custom existing configuration			
Solution strategy	bottom-up				top-down				
Option display/consistency enforcement	unrestricted, no consistency enforcement			limited restriction, partial consistency enforcement			fully restricted, strong consistency enforcement		
Validation	none	singular, at configuration end			repeatedly		continuously		
Specification support	active		passive / re-active						
	defaults	automatic completion		help		recommendations			
	non-personalized	personalized		context-insensitive	context-sensitive				
Presentation									
Configuration description	textual/tabular			graphical			both		
Visualization	static/primitive				dynamic				
Degree of interactivity	none		limited			extensive			
Point in time of visualization	delayed		real-time, continuously						
Rendering method	pre-produced images		compound images		rendered images/graphics		virtual reality scenes		
Level of detail	low				high				
Price calculation and display	simple		complex						
	no calculations/accumulations		calculated/accumulated		total prices	price premiums	both/combined		
Implementation Aspects									
Implementation approach	custom development	application module		standard software		refinement (reference system)		platform/framework	
Universality	specialized				universal				
Configuration approach	implicit techniques			explicit techniques / model based				combined techniques	
	decision-free based	rule based	case based	decision-table based	resource based	constraint based	component based	object-oriented	knowledge based
								hybrid	

Figure 3.12. Morphological Box of Configurators

3.3.3.1. Application Context

Product nature. In general, it's quite relevant what types of products are subject to configuration and whether the configuration system supports these types: the *product nature* distinguishes *non-tangible, immaterial products* like services and *material, physical products*. Also combinations of both are possible, in the sense that a customizable, physical product may well be supplemented by configurable service accomplishments. The support for different product types ultimately depends on the modeling capabilities provided by the configuration system, see Section 3.3.3.3, "Modeling Capabilities".

Business area. Also the *business area* to be served by the company strongly influences the configurator and the overall configuration process. In *business-to-business (B2B)* scenarios, it can be assumed that configurator users own specific domain knowledge. The utilization of technical vocabulary and a more technical presentation is appropriate in this case. In contrast, in the context of *business-to-consumer (B2C)* transactions, a more sophisticated user experience and an easy to overlook and follow configuration process, that doesn't require sophisticated domain knowledge is important.

Site of operation. Likewise, the *site of operation* has an impact on the configurator's design: application's solely used *enterprise internal* are usually used more frequently by the same users. Here, the efficiency of the interface compared to its appearance is more important.

A system used *externally*, faces customers directly or at least indirectly, which is why an aesthetically appealing interface is more relevant. External systems can be further divided into those that are used at the *point-of-sales (POS)*, that is, at retailer's store, or those that are virtually accessed through the world wide web. In the former case, sales staff can assist the user during the configuration process (the user is then indirectly facing the application), while in the latter case, the users are left by oneself (in direct contact with the system), which may require the availability of additional support options. See the section on "Specification support" in Section 3.3.3.4, "Configuration Characteristics" for details on how this support can be achieved.

Target audience. The previous point already showed, that it's of interest who's the *targeted audience* of the configurator application. *Staff members*, that frequently access a tool, have other demands and requirements on efficiency, design and usability than *customers*, using a tool only sporadically. As an example, the usability principles²⁰ *interface controllability* (e.g., quick navigation, short cuts) and *interface customizability* (e.g., different views, such as compact list view vs. elaborated detail view) are important from an expert's point of view. However, the principles *self-descriptiveness* (e.g., texts formulated using user's vocabulary) and *suitability for learning* (e.g., sophisticated help texts) have priority for non-experts²¹.

Integration context. The organizational scope of use determines the *integration context*. Whether the configurator is used to support sales, development, production activities or any combination of these is a decisive factor for the design of the configurator and the incorporated product model.

A *sales driven* configurator (typically referred to as "sales configurator" or "quotation configurator") focusses on marketing and selling aspects. The product model usually only features customer perceived product characteristics, that directly relate to the product's functionality. Sales configurators are the common case for configuration systems in e-commerce scenarios. They have a strong relationship to CRM systems.

Configurators used in *development and engineering* have their origins in the field of CAD systems. They're primarily used to ease and automate the design process of product variants,

²⁰For details on usability principles, refer to http://de.wikipedia.org/wiki/EN_ISO_9241#EN_ISO_9241-110_Grunds.C3.A4tze_der_Dialoggestaltung, last accessed May 18th, 2012.

²¹For more details on the relevance of usability principles in relation to user characteristics, see <http://wiki.magnolia-cms.com/download/attachments/21954574/Usability+Principles+in+relation+to+Usage+Context.pdf?version=1>, last accessed May 18th, 2012.

with a special focus on technical aspects. A typical task of an engineering configurator is the generation of technical 2D drawings.

Configurators originating from ERP systems and related software, such as PPS systems, are used to support *production* activities. They support to breakdown configured products into detailed bills-of-material or work plans and thus help to automate the entire order fulfillment process from sales over production to assembly.

In practice, the precise distinction of these systems is often difficult, due to the fact that *hybrid forms* exist, that serve multiple purposes simultaneously.

Sales support. In terms of sales configurators, it's meaningful to further characterize the extend of *sales support*. Particularly, the configuration system can have a supplementing or a replacing role.

A configurator that purely *supplements* the sales process doesn't offer direct ordering and purchase capabilities (see feature *Shopping cart and checkout* in Section 3.3.2.3, “Communication”). This model is frequently used in business areas where products are sold at comparably high prices, such as the automobile industry, real estate and capital goods. While these configurators allow the customer to independently self-configure products, upon submission the resulting configurations are solely transmitted to a retailer. They allow to generate preliminary quotes and thus strongly support the sales process, but in the end, it's the retailer that negotiates the final contract with the customer face-to-face.

On the other hand, configurators with ordering and payment functionalities can fully automate the sales process and thus act as complete *replacement* for other sales activities.

Underlying order-fulfillment process. In the context of process automation, also the *underlying order-fulfillment process* (see Section 2.3.3.4, “Order Fulfillment Strategies”) employed by the product customizer, plays an important role during the overall forming of the configurator. The configurator helps to automate the order-fulfillment process from the point the customer is integrated in the company's value chain up to delivery of the product and sometimes even longer. The earlier the customer is involved, the more elaborate the configuration process and the more precise the configurator generated product specification must be. Specification errors applying to an earlier stage of production are by far more expensive than those applying to a later one. Consequently, companies aim for reducing specification mistakes to a minimum.

In summary, a configurator in a *Build-to-Order* or *Develop-to-Order* scenario is usually more elaborate than a system, that elicits requirements in an *Assemble-to-Order* scenario.

3.3.3.2. System Environment

While the previous section focussed on business related aspects, categories related to the technical environment of a configuration system need to be examined, too.

System interaction. Architecturally, the style of *system interaction* is fundamental. Configurators specifically designed for (*pure*) *offline* use (e.g., distributed on CD/DVD) are static and must encompass the full product data and configuration logic. Nowadays, they're rarely used in B2C markets due to the relatively high distribution costs. However, they are still widespread in application areas, where:

- the configurator integrates complex, locally installed 3rd party applications (e.g., CAD software for rendering 2D graphics),
- if high hardware performance is demanded (e.g., utilization of special graphics card capabilities), or
- if a huge amount of data is involved (e.g., high resolution images, very large component catalogs, etc.).

Online systems are those that are distributed via the internet, intranet or extranet. These configurators can be divided depending on the data synchronization and communication pattern applied.

A *thin client* typically realizes a client-server scenario. Whereas the client solely serves for presentation purposes, the data and logic remains on the server-side. Thus, an intensive interaction between both peers is required. An example for a thin client is a traditional HTML²² web page.

In contrast, a *fat client* not only realizes the presentation layer, but also implements the configuration logic (or at least parts of it). Sometimes, even the data is (eventually temporarily) stored with the client allowing to (temporarily) being completely disconnected from the network. Examples for these systems are modern HTML5²³ applications or "sandboxed" techniques such as Adobe Flash applications, Java Applets / WebStart programs or native mobile apps [Pryss2010].

In either case, online or offline, configurators with order capabilities need to feed final configurations back to the supplier. This may happen immediately or delayed and depending on the network status requires storage capabilities.

System organization. Also strongly related to the application's deployment strategy and its high level interaction pattern, is the *system organization* in terms of (product-) data distribution. The data, the configurator operates on, may be stored on a *central* server or may be *distributed* to the clients. While the former approach requires highly frequent communication (typical for thin clients), in the latter approach, the distributed data storages must be synchronized (fat clients). The synchronization may occur each time the application is started / requested from the server (typical for fat clients, e.g., Adobe Flash, Java Applets / WebStart) or happens periodically by pulling the data from the server using an update mechanism (typical for desktop applications or mobile apps).

Integration scenario. Another vital characteristic of a configuration system is its *integration scenario*. If the configurator is not integrated into another system at all, it is considered a *standalone* system. It comes with its own database, user interface and execution environment.

A configurator that is *embedded* into another system is quite similar to a standalone system, with the exception, that the user interface is integrated within another system (the so called *host system*) and sometimes, the data store is shared. Otherwise, the host application and the configurator are not tightly coupled, technically. For example, when a standalone configurator is made accessible from within an ERP suite or visually embedded into the same, we consider it an embedded configurator.

The most extended integration scenario results, when the configurator is *fully integrated* into another application. In this case, both systems are tightly coupled on code level and the integrating application merely uses the configurator's API / SPI to facilitate configuration or vice versa. For fully integrated configurators, the hosting application provides the presentation logic entirely. Consequently, in this scenario, the configurator cannot be used standalone, but is rather used as an application library, i.e. is a module of a larger application. Figure 3.13, "Different Integration Scenarios for Configuration Systems" illustrates the different scenarios.

²²Abbrev. HyperText Markup Language

²³More precisely: HTML and JavaScript based applications, that utilize local storage features introduced in HTML specification version 5.

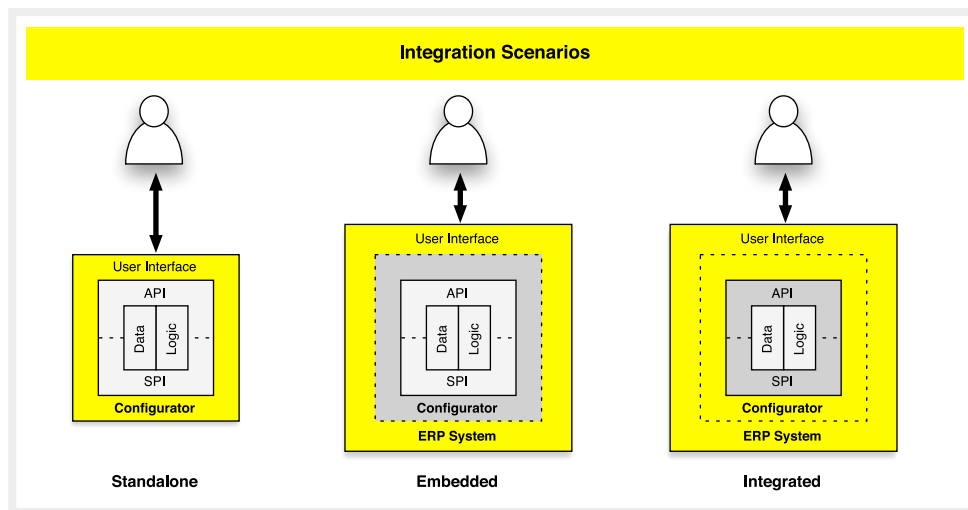


Figure 3.13. Different Integration Scenarios for Configuration Systems

Integration level. The integrative relationship between the configurator and its environment can be further characterized by the *integration level* supplied. As depicted in Figure 3.14, “Integration Levels of Configuration Systems”, we distinguish:

- not integrative,
- data integrative, and
- application integrative.

Not integrative means, that the configurator is a closed system, not providing any kind of interface to other systems. Though the system may itself very well access other applications or *consume* external services, e.g., to transfer configuration results to other backend systems. It merely doesn't *provide* such access points for external applications.

In turn, a *data integrative* configurator not only is capable of *pushing* data to other systems but offers interfaces, that allow applications to *pull* information from the configuration system. This can technically be achieved, for instance, by providing a (read-only) web service endpoint.

Importantly, data integration doesn't involve the execution of application logic, which is significant for an *application integrative* scenario. To facilitate application integration, the configurator may expose its functionality as web service, application programming interface (API) or service provider interface (SPI). A web interface or an API are usually installed in order to access the configurator from another application. In this case, the configurator acts as a *provider* for the external application. An SPI in turn, is offered by the system in order to let the configurator *itself* connect to other systems. For example, a database vendor may provide a driver, that is compatible with the configurators SPI. This allows the configuration system to push/pull data to/from the database. In terms of an SPIs, the configurator mainly acts as a *consumer*.

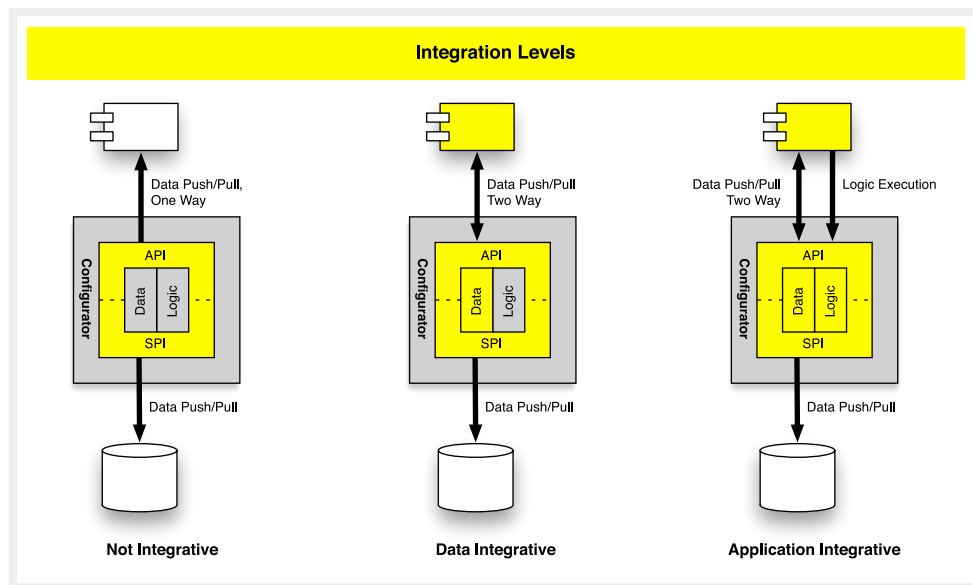


Figure 3.14. Integration Levels of Configuration Systems

Target device. To end up the characterization of the configurator's system environment, it's meaningful to determine the *target device*, on which the application is being deployed. The target device not only has an impact on the layout of the configurator's user interface, but also affects its overall architecture. In our opinion, it's useful to distinguish *desktop computers/laptops*, *touch device/tablets* and *mobile devices*. They differ in their available screen size, but also in their provided computing power, resources, software capabilities and network availability, which are technically important factors: they determine the general conditions of the configuration system and are thus highly relevant for the technical implementation.

3.3.3.3. Modeling Capabilities

Next we want to discuss some aspects, that relate to capabilities requested by configurator developers.

Modeling capabilities. For the developer implementing the configurator, it's important, whether the configurator provides sufficient *modeling capabilities* to represent the customer's problem domain. In Section 2.1.3.1, "Customizable Areas" we discussed a framework for describing customizable products. To recap, the different sections are *fixed areas*, *obligatory/alternative elements*, *optional elements*, *scalable areas*, *principle solutions*, *services*, *defined expansion spaces* and *general expansion spaces*. We argue that this list of sections is comprehensive enough to cover most use cases and that it can be adequately used to characterize the capabilities of a configuration system. Typically, a configuration system will support several of these capabilities.

Level of detail. Beyond the general modeling capabilities, it's relevant whether the configuration system allows to model the required *level of detail*. When a configuration system constrains the degree of recursion of the product structure, that is, restricts the depth of nested components to a certain number, we consider the level of detail as *limited*. Otherwise, the detail level is *unlimited*.

Product model evolution support. Although not solely related to modeling capabilities, the *product model evolution support* is an important issue for configurator maintenance in the long run. This property describes, whether the configuration system supports changes of the underlying product model, the configurator operates on by any means. Moreover, it describes how these modifications are handled, when old configurations are restored, e.g., in

case an end user wants to order spare parts for a custom machinery configured in the past. Either the system has

- no support,
- limited, or
- full support

for such use cases.

No support means the configurator is entirely closed and modifications to the knowledge base require a new version of the configurator being deployed. Old configurations cannot be restored at all or not correctly in the new configurator version. Also, they cannot "per se" be transitioned to the new system but require a custom transformation.

Limited support exists, when at least the knowledge base can be adjusted without requiring a complete new version of the configurator being deployed. This way, old configurations can be reloaded as long as they're compatible with the changes. Nevertheless, when reloading an old configuration, a sometimes desired, but in most cases undesired feature is, that the configuration rules or the price calculations are not the same as in the original configuration.

This issue is not addressed in limited support scenarios, but it is when the configurator facilitates *full support* for product model evolution. Here, the configurator features *versioning* for product models allowing full *reconfiguration* support. This means, old configurations and newer ones can be loaded in parallel in the configurator correctly: while for old configurations the old product model and along with the old constraints are used, for the new configuration the new product model applies. However, full product model evolution support is considered an advanced feature for configuration systems.

3.3.3.4. Configuration Characteristics

The next categories directly relate to the configurator's interface and the overall configuration process implemented.

Configuration Approach

First, the general *configuration approach* can be characterized. It defines *how* configuration is performed from a high level perspective.

Specification approach. In general, the *specification approach* can be designed

- product-centric,
- customer-centric, or
- can be a mixture of both.

Figure 3.15, "General Specification Approaches" depicts these approaches schematically.

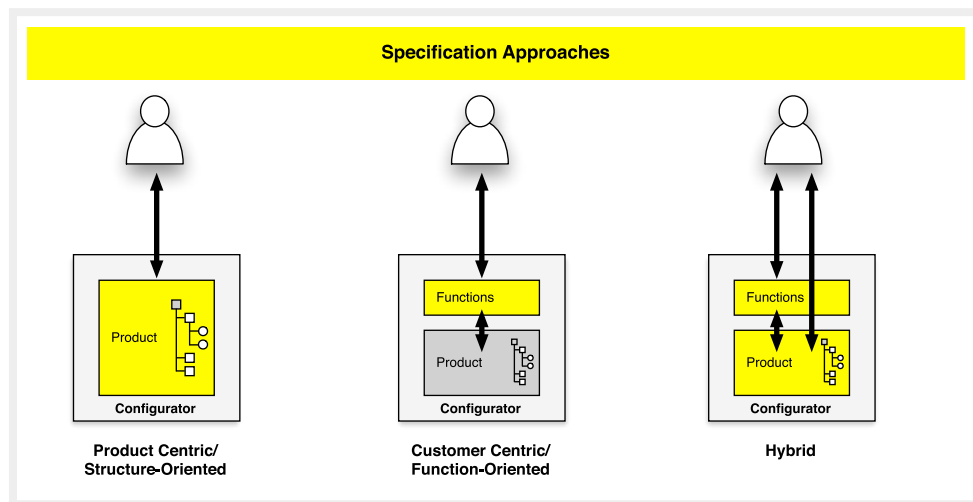


Figure 3.15. General Specification Approaches

In a *product-centric* scenario, the configuration process is organized according to the product's structure, which is why this approach is also coined *structure-oriented*. The user directly specifies the product's components and attributes one by one, that is, he selects values for options, adds components etc. In order to configure a custom product, the user needs to know how his requirements map to the features of the product. Hence, this more technically driven approach demands sufficient domain knowledge to be owned by the customer.

In contrast, in terms of a *customer-centric* approach, the configuration process is structured based on the product's functions. Therefore, this approach is also called *function-oriented* or *need-oriented*. During configuration, the user doesn't specify concrete product characteristics but instead, he merely specifies its functional characteristics. This allows him to map his requirements to functional preferences, which is much easier for the majority of users without sophisticated domain knowledge: it is the configurator, that maps the functional requirements to specific product attributes.

A *hybrid* specification approach, signifies a mixture of function-orientation and structure-orientation. In this case, configuration starts by eliciting specific customer requirements. These requirements are mapped by the configurator to an initial product configuration, which can then be further adjusted by the customer who modifies product parameters directly.

Configuration capabilities. The *configuration capabilities* influence the overall characteristic of the configuration process. Depending on the type of modularization, see Figure 2.9, "Types of Modularization for Platform Designs" in Section 2.3.3.3, "Modularity", we distinguish the following configuration types:

- **Module configuration.** In case the underlying product architecture realizes *generic modularization*, during configuration a fixed number of pre-defined components must be selected and possibly parameterized. For example, the configuration of a car can be considered a type of module configuration.
- **Mixed configuration.** If *quantitative modularization* is implemented in the configurable product architecture, multiple atomic components (see Section 2.1.1, "Components") can more or less unrestrictedly be combined. An example for mixed configuration is implemented in mymuesli.com's cereals configurator²⁴.
- **Adaptive configuration.** Products build up of a *custom modularization* architecture have a defined base structure, but can be adapted to customer needs at certain points. Beyond

²⁴See <http://www.mymuesli.com> [<http://www.mymuesli.com/>], last accessed May 20th, 2012.

pre-defined components, customer-specific components can be selected. Nevertheless, the general extend of customizability is limited.

- **Design configuration.** Offers the most extended form of configuration capabilities. Special design tools allow to construct entirely custom products. The resulting product architecture corresponds to a *free modularization* approach.

Figure 3.16, “Relationship Between Configuration Type and Configuration Decisions” shows the relevant configuration decisions in each scenario. Refer to the respective sections in Section 3.1, “Basic Meta Model for Generic Product Modeling” for a detailed description of the decisions stated.

	Module Configuration	Mixed Configuration	Adaptive Configuration	Design Configuration
Component Quantity Decision				
Optional component	●	●	●	■
Mandatory Component	●	●	●	■
Multiple Components	○	●	●	■
Component Variety Decision				
Fixed components	●	●	●	■
Alternative components	●	●	●	■
Component Customization Decision				
Selected component	●	●	●	■
Constructed component	○	○	●	■
Attribute Quantity Decision				
Optional attribute	●	●	●	■
Mandatory attribute	●	●	●	■
Multi-value attribute	●	●	●	■
Attribute Valuation Decision				
Selected value	●	●	●	■
Custom value	●	●	●	■
Constraint Decision				
Optimization variables and constraint weighting	■	■	■	■
Constraint relaxation	■	■	■	■

● Available ○ Not available ■ Advanced use case

Figure 3.16. Relationship Between Configuration Type and Configuration Decisions

Interaction characteristics

Having identified the general configuration approach, one can describe the *interaction characteristics* of the configuration process more precisely.

Configuration procedure. The (*general*) *configuration procedure*, which describes the way the configurator solves the configuration problem from a high level perspective, can be executed in two ways: batch processing and interactive processing.

Batch processing means, the configurator autonomously solves the configuration task without any further user interaction. At the beginning, the customer states requirements the final

solution must fulfill. Then the configurator automatically calculates one or more, matching results. The customer finally choose the best matching solution.

The second, alternative approach is *interactive processing*. Here, the solution is step-wise worked out in a cooperative interaction process between customer and configurator: while the system repeatedly requests decisions from the user, the customer chooses options that incrementally specify the final product more precisely.

Process scheme. An interactive configuration procedure can be run in different variants. according to the underlying *process scheme* (see [Scheer2006, p. 52]). The process scheme basically defines the sequence in which the configuration decisions are presented. It can either be:

- **Pre-defined/fixed.** The supplier defines a fixed path through the configuration process. He may do so in order to simplify the configuration procedure for the customer or to reduce the complexity of the technical realization of the configuration solution strategy.
- **Structure-oriented.** The process is derived from the product's respectively the product model's structure. For instance, at first, the main product characteristics are determined, then decisions on for the product's sub-components are recursively requested. The resulting procedure is thus driven by the composition structure of the product (see Section 3.1.1, "Components: Structural Decomposition").
- **User-oriented.** The configuration process' structure is designed according to customer demands. For example, the most essential decisions may be requested first, while detailed specification decisions can be performed at a later stage or be automatically completed by the system. The procedure is driven by the functional characteristics of the product.
- **Data-driven.** If the configuration process depends on the current state of the configuration, it can be considered a data-driven process. For instance, the process may terminate immediately in case no other solution for the previously selected options is left.
- **Case-driven.** The configuration process is adapted to similar configuration problems, that have been solved previously. For that purpose, elicited customer requirements are matched with the case database and a specific process scheme is selected.
- **Adaptive.** An adaptive configuration process may be chosen just before or even modified during configuring by the customer himself. This allows customer specific configuration sequences based on his personal preferences.

Dialog style. Independently from the underlying configuration process scheme, configurators may strongly vary depending on the supported *dialog style*. The configuration dialog is responsible for recording the customer's decisions. It can be designed text based or graphically.

A *text based* specification interface uses either a sequence of prose questions (*questionnaire style*) or a *form* that is split across one a *single* or *multiple screens* to elicit the customer information. A single screen means, that all configuration options are presented on one large page. Multiple screens dissect the configuration options to multiple pages. Depending on the process scheme, the form may also be *hierarchically* organized. Hierarchical or sequential screen flows require an additional navigation to step through the different screens. Forms can be technically realized with standard controls like input fields, checkboxes etc. but may also be filled with option lists that have a more textual than technical character. Also, special controls like sliders or image menus are common for improving the user experience.

A *graphical* specification interface means, that configuration is performed directly in the configuration's visualization view using *point-and-click* methods. They are frequently used for the configuration of design-centric, layout-aware products. However, their implementation is also the most challenging from a technical perspective. As an example, the layout of a kitchen can be designed much easier with a visual interface opposed to textual input forms. In contrast, an insurance contract cannot meaningfully be configured graphically.

Configuration procedure

The applied *configuration procedure* (in terms of the configuration process) can be characterized more precisely.

Starting point. The *starting point* of a configuration session may vary according to the configuration scenario: on the one hand a new configuration can start with a *blank, empty configuration* which doesn't contain any pre-configured items. This does not affect the application of default attribute values, though.

On the other hand, the supplier may *pre-define standard or base configurations* to start the configuration process with. Effectively, starting with a partially finished configuration accelerates the configuration process for the customer. It is a common scenario especially for platform based architectures (see Section 2.3.3.3, "Modularity"), where configuration starts with a previously chosen or default basic product and customers solely configure the modules attached to it. For example, this is mostly the case in car configurators, where the user chooses a basic car model and solely specifies motor power, exterior and interior related attributes.

Moreover, the configuration procedure can also be started with an *existing configuration* that has either been previously saved by the customer himself or by another community member²⁵. Starting with an existing configuration requires reconfiguration features implemented in the configurator (see Section 3.3.3.3, "Modeling Capabilities", "Product model evolution support").

Solution strategies. Depending on the starting point of the configuration process, the supplier may also implement different *solution strategies*²⁶. Most commonly, a *bottom-up* approach is taken, where the supplier comes up with an empty configuration and the customer primarily adds the items he wants in addition to the existing ones. In contrast, in a *top-down* scenario, the configuration process starts with the full set of possible components and the customer removes those components, that he doesn't need.

Option display/consistency enforcement. Another essential point from both the customer's and the technical perspective is the form of *option display*. While the sole display of options doesn't sound like a challenging task, it is closely related to the *consistency enforcement* mechanism realized in the configurator, which, in turn, is a challenging issue to implement.

Basically, it's all about whether the configurator restricts the shown options to those that are valid in the current configuration situation or whether it doesn't do so. Assuming that the configurator presents valid options only, it's logically impossible to result in an inconsistent configuration state. We say *the consistency has been enforced* by restricting the options displayed. Technically, implementing consistency enforcement is a challenging task, because numerous constraints may be checked in order to remove all inconsistent domain values. However, in practice, multiple levels of option display / consistency enforcement can be distinguished:

- **Unrestricted/no consistency enforcement.** For a given configuration decision, the configurator presents all options without restriction. An invalid selection by the user is not detected unless the configuration is validated (see "Validation" below). Consistency is thus enforced by no means. Technically spoken, for attributes, the configurator presents the full domain to the user.
- **Limited restriction/partial consistency enforcement.** The configurator *partly* checks the validity of a decision's options prior displaying them. We say *consistency is partially enforced*. Depending on the constraints, that restrict a variable attribute's domain, verifying

²⁵Sharing configurations across a community of users, is part of an idea on cooperative configuration described by Leckner in [Leckner2006].

²⁶In [Krug2010], solution strategies are referred to as "Entscheidungsarten", which essentially describe the same concept.

the validity of a particular domain value may be a difficult, sometimes time-consuming or even infeasible task. Especially, it's hard to enforce consistency across strongly interrelated attributes. In this case, complex logical and mathematical calculations must be performed in order to determine the validity of a given value or value combination.

An alleviated implementation may, for example, only ensure local consistency on a given attribute's domain. That means, only unary or binary constraints (see Section 3.1.3, "Constraints: Domain Restrictions") are evaluated for consistency enforcement purposes. Consider the following example: for a numeric attribute *A* with a domain of {0, 1, 2, 3, 4, 5, 6}, an unary constraint may be defined, that states $A < 4$. In this case the configurator could evaluate the constraint and remove the values 4, 5 and 6 as they would anyway be rejected upon the next validation step. Although local consistency could be enforced, there might be another condition defined on any other variable, which may require that the value of *A* being greater than 0 and consequently the value 0 would also have to be removed. However, this condition not necessarily must be defined explicitly, but instead may be the result of a complex calculation, which is not performed in advance. Consistency is thus enforced only *partially*.

- **Full restriction/strong consistency enforcement.** To continue with the example above: if the value 0 would also have been removed and all remaining values in *A*'s domain {1, 2, 3} are guaranteed to result in consistent configurations, we consider the attribute as fully restricted and consistency as being enforced strongly. In this case, regardless of the selection of the user, the configuration would remain consistent regardless of any subsequent decision, which is why we say it's *a priori consistent*.

Validation. Fundamentally important for a configurator, is the *validation* of the configuration's correctness. While the consistency enforcement mechanism described above is concerned with *a priori* consistency, the validation mechanism checks consistency *ex post*, that is, *after* the customer specified a value. In other words, it checks a specific set of attribute valuations against the constraints defined over that set. An important factor in this regard is *when* the configuration is validated. We differentiate the following validation strategies (cp. [Krug2010]):

- **None, primitive.** The configuration is not validated at all. It's the primitive case and not really reasonable for real world use cases.
- **Singular, at configuration end (result based).** The configuration is validated only once, after the user specified all configuration options. Under certain circumstances this may frustrate the user dramatically, since in case of validation errors, he might be forced to track back to an early point in the configuration process. Thereby, he's possibly forced to dismiss numerous decisions taken after the faulting one, in order to correct the problem.
- **Repeatedly (interval based).** To alleviate the problem described above, the configurator might verify the configuration's consistency periodically at specifically defined points in the process, e.g., upon the specification of a logical unit of the product.
- **Continuously (runtime based).** Optimally, configuration validity is verified continuously, that is, anytime the customer submits a value. This way the user quickly receives feedback and can immediately react on problems that resulted from his latest decision.

As validation is performed *ex post*, respectively after some kind of user interaction, it can be considered a *passive* or *re-active* mechanism. In contrast, consistency enforcement is performed upfront and can thus be seen as an *active* mechanism supporting the customer.

Specification support. A configurator can provide more *specification support* features, both active and passive ones. Active specification support includes the provision of defaults and automatic completion features. Passive support functionality includes help and recommendation techniques.

Defaults are relevant when instantiating an entirely new configuration or adding a new component to it: they are supplier defined, initial valuations for configuration options. By providing defaults, the supplier can communicate what's considered the "standard" option for a decision. In many cases, this may speed up the configuration process drastically, in other cases it might help a customer without personal preferences regarding a specific configuration option, to make a decision.

Automatic completion functionality may additionally speed up the configuration process and thus can be considered a feature to improve customer convenience. It's relevant, for instance, if the user selects a certain option that requires another option to be selected (expressed by a constraint). The second option would be automatically chosen by the configurator on behalf of the customer. Automatic completion may, however, also be triggered explicitly by the customer: once the user has specified his most important requirements on the product, he may ask the configurator to complete the configuration, in order to quickly finish the entire process (for details, refer to the *User-oriented* process scheme in Section 3.3.3.4, "Configuration Characteristics", "Process scheme"). During defaults application or during an auto completion step the configurator can either take personal customer requirements into account, when calculating the chosen option (*personalized*), or don't do so (*not personalized*).

A *help* feature providing additional information to the customer can be understood as passive form of support: the customer must perform an action in order to be assisted. One can distinguish *context-sensitive* help, e.g., a tooltip displayed on a particular attribute that explains the available configuration options, from *context-insensitive* help, e.g., a user manual delivered separately from the configurator.

In advanced configurators, a user can request *recommendations* for specific configuration decisions. The system then proposes the best matching option in the context of the current configuration. The user can either accept or reject the proposed value. There are several methods to calculate reasonable recommendations, e.g., based on the choices, that other users with a similar profile took. These methods are subject to the discipline of recommender systems, which is why we refer to the literature at this point (see [Renneberg2010]).

Presentation

Configurators largely vary in the way information is presented. Thus, in the following, *presentation* related categories shall be discussed.

Configuration description. As already mentioned in Section 3.3.2.2, "Specification / Recommendation", the *configuration description* is an important part of the configuration dialog. Configurations can be described in *textual/tabular* form, which lists all chosen components along with the key attributes and pricing information (see "Price calculation and display" below). Additionally, the configuration description can be supported by *graphical visualizations* (see "Visualization" below). Commonly, configurations are described using *both*, textual and graphical elements.

Visualizations. Configuration *visualizations* can be designed quite differently. *Primitive* approaches do not consider configuration specific attributes. They're simply *static* product images. Thus, in practice, normally *dynamic* visualizations are used for configuration purposes. Their scope of design can be classified by the degree of interactivity, the point in time of visualization, by the method employed to produce them and by their level of detail.

Starting with the **degree of interactivity**, we differentiate:

- **No interaction (static presentation).** The statically rendered view of the configuration doesn't offer any interaction capabilities.
- **Limited interaction (semi-static presentation).** Semi-statically rendered views allow limited interaction, e.g., clicking a part of the generated image to jump to the correspond-

ing component specification dialog. Direct modification of the configured object is not possible.

- **Extended interaction (interactive presentation).** Interactive renditions allow the graphical editing of the configured object. This way, they enable the *visually interactive specification* described above.

Related to the **point in time of visualization** one can differentiate between [Rogoll2003, p. 59-60]:

- **Delayed visualization (at the end of the configuration process).** The configured product is only singularly rendered at the end of the configuration process and visualization is thus *delayed*. This method doesn't really support the configuration process, but may be comparably easier to implement.
- **Realtime visualization ("Step by step").** The visual representation of the product is instantly and continuously updated, as the customer specifies configuration options. While this method provides a better user experience, the implementation is considerably more complex than delayed, one-time rendering.

Methods for rendering visualizations include [Rogoll2003, p. 59-60]:

- **Pre-produced images, sprites.** The visualizations for all product variants have been pre-produced, including all possible combinations. Depending on the selected variant, the correct image is loaded. Obviously, this approach is only practicable for a limited set of combinations. Visualization forms include high quality photo realistic pictures, scribbles or technical drawings. They do not require special software on the client side.
- **Compound pictures.** Multiple pre-produced pictures of components are aggregated within a single image. While the resulting images have more or less photo realistic quality, only the different component variations, but not their combinations, must be pre-produced. By utilization of transparency effects, the technical realization is comparably easy and no additional software is required on the client computer.
- **Rendered pictures.** In terms of rendering, a model or vector graphic of a component or the entire product is dynamically generated for near photo realistic appearance. During the rendering process, the model is overlaid with a texture determined by the configuration. On the one hand, this approach offers great flexibility and allows to efficiently generate a visualization for an arbitrary number of component and attribute variations. On the other hand, the technical requirements on the configurator's image processing system are relatively high.
- **3D models, virtual reality.** The most flexible visualization approach and the one which provides the best user experience is the use of interactive virtual reality scenes. The product is described as a 3-dimensional model and rendered according to the configuration state directly on the client appliance. Whereas in old web browsers²⁷ special software plugins for rendering 3D scenes is required, most modern browsers already implement native support for industry standards such as X3D²⁸. In contrast to rendered pictures, 3D models can be equipped with additional interactive features: the model can be freely rotated or zoomed, parts can be moved realistically and even virtual camera flights showing a product's interior, can be realized.

Finally, configuration visualizations vary in the **level of detail** they provide. A presentation with a *low* level of detail solely shows the final product, hiding detailed visualizations of the components it consists of. A visualization with a *high* level of detail may provide multiple

²⁷We consider web browsers as the primary client application for configurators throughout this work.

²⁸See <http://de.wikipedia.org/wiki/X3D>, last accessed May 14th, 2012.

perspectives of the configured product, different resolutions and additional graphics that show component details.

Price calculation and display. Another important aspect of configuration, especially in terms of configuration of consumer products, is the *calculation and display of prices* (cp. [Polak2008, pp. 11]). In our opinion, it's reasonable to differentiate between simple and complex price calculation methods, because suppliers often have very different price calculation models.

In terms of *simple* price calculation, the configurator either does not provide a special calculation mechanism, e.g., when a fixed price or a specific price function is defined for a product, or alternatively just accumulates the prices of a product's components. Consequently, for displaying purposes, solely the product's final price and the price of each component as defined in the product model is used. This method is considered simple, because the price can be determined from the configuration result without any additional context information.

In contrast, a *complex price calculation* mechanism requires contextual information. Here, we distinguish two flavors: total price and premium price display. In case of a *total price display*, for every option the total price of the product, calculated from the current product's price plus the option's price, is displayed. In contrast, *premium price display* shows the price premium the user would have to pay, if choosing that option, that is, the total price of the product including the option *minus* the current configuration's price. In the latter approach, the displayed price for a component depends on the previously selected, alternative component, which is why we say that contextual information is required to calculate the price. In practice, often *both forms* of price display are used in combination within the same configuration process.

3.3.3.5. Implementation Aspects

In the last section of configurator categorization, we want to discuss some implementation related, technical aspects of configuration systems.

Implementation approaches. First of all, there are several *implementation approaches* to configurators (cp. [Scheer2006, p. 54]):

- **Custom development.** The configurator is implemented from scratch for a single, particular use case. Product knowledge and configuration logic are mostly hard coded within the application.
- **Application module.** The configurator is developed as a module itself or using a module of a larger application, e.g., an ERP software suite. The product knowledge is derived from the host application's data, which is augmented by configuration specific knowledge and functions.
- **Standard software.** The configurator is created using a standard configuration software. The system may be a truly generic configuration system that is capable of mapping any kind of product, or may be a solution for a specific business field (e.g., insurance services, see also "Universality" below). Furthermore, the standard software may deploy a static configurator instance or may provide a "sandbox" that acts as an execution environment for defined customizable product models.
- **Refinement.** An existing reference system, e.g., a research prototype, is refined in order to build a custom product configurator.
- **Platform/framework.** Specific configurator instances are created based on a common platform or around a common framework, which provides services for generically dealing with different product models. A configuration system platform also provides additional tools for managing knowledge bases, product data, configurations, testing and debugging configurators, usage statistics analysis and more.

Universality. Given a specific implementation approach, one can differentiate a configuration system according to its *universality*. A *specialized* configurator is capable of configuring a single product, products of a single company or those of a specific business area. In contrast, a *universal, general-purpose* configurator can handle a variety of product models at it is implemented entirely model based.

Configuration Approach. Finally, a far reaching difference between configuration systems is their underlying *configuration approach*. Depending on the separation of product knowledge and configuration logic, we can differentiate implicit, explicit and combined techniques [Scheer2006, p. 54-57].

Within **implicit techniques** the configuration problem is described *implicitly*, that is, only specific or partial solutions of the entire configuration problem are provided. Furthermore, the same formalism is used to describe both product knowledge and configuration logic, i.e. the product knowledge is defined *implicitly* by the configuration rules [Scheer2006, p. 54] (see also [Sabin1998]):

- **Decision-tree based configuration.** The configuration options are structured as a decision-tree, where each node represents a decision and the node's descendants correspond to the options the user may choose from. During the configuration process the tree is traversed unless a leaf node is reached, which signifies a complete configuration [Rogoll2003, p. 80-81].
- **Rule based configuration.** Rules consisting of a condition and an action part (see "Rules" in Section 3.1.3, "Constraints: Domain Restrictions") are used to model the domain knowledge and control strategies. This approach intermixes product knowledge and the logic stated by experts to solve problems. A configuration solution is determined by letting a rule engine repeatedly match the rules' conditions against the current configuration state and performing the defined actions. Moreover the application of heuristics helps to find matching solutions [Rogoll2003, p. 77-78].
- **Case based configuration.** The basis for configuration are previously completed, archived configurations. During the configuration process a similar solution is searched within the set of archived ones using a *similarity measure*. A matching solution is used as a template for the current configuration problem.

In terms of **explicit configuration approaches**, also known as *model based approaches*, the configuration problem is *explicitly* described as a domain model, i.e. all possible configuration options are represented within a generic product model (see Section 3.1, "Product Models"):

- **Decision-table based configuration.** Configuration is based on decision tables and compatibility matrices. Within *decision tables*, dependencies between product components are modeled, while *compatibility tables* can be used to express feasible combinations and consistency constraints. Solutions are determined by traversing the tables [Rogoll2003, p. 79].
- **Resource based configuration.** Configuration is understood as *resource balancing problem*. Hence, the model contains types, that produce resources and components that consume them. During an iterative process, resource *requirements* are defined by adding *resource consumers*. The resource consumption is *compensated* by adding *resource providers*. The configuration is not completed, unless consumption and production aren't balanced.
- **Constraint based configuration.** The configuration problem is modeled using domain knowledge, that is, available components with corresponding attributes, and control knowledge in terms of constraints. Constraints describe relationships and conditions between components, i.e. attributes of components, and are used to verify the validity of a configuration. Thereby, *propagation* is used to incrementally restrict selectable options until a single valuation of configuration variables remains. The remaining valuation is considered a valid solution to the configuration problem, if all constraints are satisfied [Rogoll2003, p. 78].

The configuration problem can also be seen as a **Constraint-Satisfaction-Problem (CSP)**: if all constraints defined within a problem hold true for a given valuation, a solution to the problem is found.

- **Component based configuration.** A component based configuration approach, also known as *structure-based* approach, describes the configuration problem in terms of components, attributes, ports and functions. During configuration, at first, the attributes of key components are specified. Then, additional components are subsequently added to the configuration according to function requirements and port specifications. Compatibility between components are described using constraints, which are attached to particular components.
- **Object-oriented configuration.** Similar to a component based approach, the configuration problem is described in terms of an object-oriented model. Within the object-oriented model, relationships between classes can be established by the means of association, aggregation and generalization/specialization. While classes correspond to components (component types), objects, i.e. instances of classes, correspond to component variants. During the configuration process, a concrete object instance is searched or instantiated. Configuration restrictions are described as constraints.

Combined techniques can be seen as comprehensive, elaborate approaches that involve multiple other techniques:

- **Knowledge based configuration.** In knowledge based configurators, also known as *expert systems*, product and configuration logic is stored within a dedicated *knowledge base*. The configuration process itself is driven by an *inference machine* operating on the data stored in the knowledge base. Such an expert system allows the user to specify his requirements, and let system guides him through the problem solving process. Multiple problem resolution techniques are applied in a goal-oriented manner until a satisfactory solution is found [Rogoll2003, p. 80].
- **Hybrid techniques.** Approaches, that involve multiple techniques, can also be considered hybrid strategies. In modern, generic configuration systems, combined solution techniques are a common case.

The overview of configuration approaches closes up our detailed categorization scheme. Later in Section 7.1.2, “Implementation Characterization” we will discuss our approach introduced in Chapter 4, *Methodology and Conceptualization* according to this morphological box in order to classify our approach precisely. However, before introducing our methodology to configurator implementation, we'll take a final look at the overall benefits of a configuration system.

3.3.4. Benefits

In Section 2.3.4, “Economical Aspects”, we already discussed several risks and potentials related to product customization in general. Here, we want to shortly highlight benefits, that are directly related to the employment of configurators in such a scenario.

Scheer compiled the following benefits of product configuration from various sources (cp. [Scheer2006, p. 43-44]):

Supplier Perspective

Strategic competition advantages.

- Improved complexity handling of a large diversity of product variants in production and customer near business areas.

- Reduction of product variety through central modelling and optimization.
- Saving and protection of configuration knowledge by transforming the knowledge of employees into a generic product model.
- Improved customer orientation by the direct realization of customer requirements in product specifications, fulfillment of a large variety of different customer desires as well as an improved consulting quality and overall customer retention.
- Mediation of a buying experience.

Productivity potentials.

- Shortening of value adding processes through the integration of product knowledge, assured bills-of-materials and product requirements as well as automated generation of production near specification documents.
- Efficient sales quote creation by reducing the need of clarification and avoiding repeated consultation, systematic elicitation of customer needs and automated deduction of a product specifications.
- Increased amount of individual customer configurations with constant human resources.
- Downsizing of routine work and development of areas of freedom for employees.
- Reduction of special requests by displaying all possible configuration options during decision taking as well as avoidance of invalid orders.

Cost reduction potentials.

- Reduction of costs in value adding areas of the enterprise (i.e. variant, changeover and reworking costs) by ensuring manufacturing feasibility during configuration.
- Reduction of sales and marketing costs (i.e. costs related to quote creation, sales personnel, travel expenses and training).

Customer Perspective

Customer-oriented product configuration.

- Autonomous, interactive specification of customizable products.
- In amount and duration unlimited form of specification.
- Spatially ("from anywhere") and temporally ("at anytime") independent specification.
- Reduced personal commitment and non-binding product specification.
- Multimedia based, virtual product preview and generation of customer-oriented, comparable product descriptions.

These benefits are strong arguments for the establishment of a configuration system within the enterprise. Especially, when the implementation of a configuration system doesn't require sophisticated effort in terms of time and money, the return on investment (ROI) can be increased strongly.

3.4. Summary

In this chapter, we established the theoretical foundation for our conceptualization: we presented and aggregated knowledge from various literature sources and compiled a detailed, technical perspective on configuration systems.

Aligned with the procedure in the previous chapter, at first, we discussed *product models*, which corresponds to the *product perspective*. Product models effectively represent the product range (refer to Section 2.3.3.2, "Product Architecture") within the configuration system. They contain *components*, *attributes* and *constraints* to reflect structural aspects, characteristics and manufacturing restrictions. Particularly in the context of configuration, the *customizable areas* of a tailorable product are of primary interest (see Section 2.1.3.1, "Customizable Areas"). In essence, the product model encompasses the full variability of the product range. Importantly, in this section, we not only extracted a *generic meta model* for representing prod-

uct models, but also identified several *configuration decisions* related to the respective model elements. Both will play a fundamental role for our conceptualization presented in the next chapter (in particular, see Section 4.4, “Modeling Concepts”).

Next, we discussed the act of *product configuration* in detail, which corresponds to the *process perspective* again. We characterized product configuration in general as *design task between selection, adaption (parameterization) and construction*. This characterization will also form an important aspect within our conceptualization, as it basically describes different *specification methods* for customizable components (see Section 4.4.3.1, “Specification Methods”). Then, we looked at configuration processes in detail by discussing both a macro (global configuration process) and a micro perspective. The configuration process was described as a *transformation process*, that converts the generic configuration model into a concrete product configuration. The transformation is driven by the configuration decisions mentioned above. For the last part of this section, we shifted to the customer's perspective and described the *interactive configuration process* from a user's point of view.

The third section focussed on *product configurators* in terms of software tools. We explained the important role of configurators within a product customization scenario, discussed their responsibilities in terms of tasks and requirements and collected a large set of features realizing these. Moreover, we compiled an elaborate taxonomy for the comparison and detailed description of configuration systems from various literature sources. Finally, the presentation of the main benefits of configurators closed up this section.

This chapter ends up the theoretical part of our work. It should have conveyed a deep understanding on configuration processes and configuration systems in general. Chapter 4, *Methodology and Conceptualization*, introduces our approach to the implementation of such systems.

4

Methodology and Conceptualization

In this chapter, we will introduce a novel approach to the realization of product configurators: *the OpenConfigurator methodology*. We argue that this new, developer-friendly way of realizing high quality, sophisticated configurators, strongly supports the practical implementation of product customization.

As described in Section 1.2, “Mission Statement” at the beginning of this work, the establishment of our methodology involves basically two things:

1. The conceptualization of a **modeling language** for the description of product configuration models.
2. The conceptualization and implementation of a **framework**, that is capable of interpreting the described configuration models and turning them into executable configuration processes

While the second issue, the description of the framework implementation, is subject to Chapter 5, *Technical Architecture and Implementation*, this chapter will introduce the OpenConfigurator methodology in general and describe the modeling capabilities in detail. In Chapter 6, *Evaluation and Validation* we will then realize a concrete use case, leveraging both the modeling language and the framework for the implementation of a mobile bike configurator.

Let's begin with the explanation of OpenConfigurator's fundamental ideas.

4.1. The OpenConfigurator Methodology

Again, OpenConfigurator aims to provide a novel approach for implementing configuration systems. While there are numerous systems available on the market today, we argue that most systems over-complicate the development of configurators. Additionally, many systems require to learn new, proprietary technologies for modeling and implementing configuration knowledge. The consequence of this is, that developers face a steep learning curve and a high entry barrier to overcome, before being able to enter the product configuration area.

OpenConfigurator, instead, can be considered as a state-of-the-art configurator implementation approach, leveraging a homogeneous set of modern technologies to accomplish high

expressiveness and high flexibility. Thereby, it remains strongly developer-friendly, extensible and maintainable. In the following, we will characterize our approach more precisely and compare it to existing approaches in the next section.

4.1.1. Main Characteristics

There are some key characteristics, that describe OpenConfigurator's nature from a high level perspective. In summary, OpenConfigurator's configuration approach is:

- model-based,
- object-oriented,
- universal and generic,
- declarative, and
- Java based.

We'll shortly explain these characteristics below.

Model-based. OpenConfigurator configuration approach is *model-based*. That means, the configuration problem is described in terms of a **domain specific model**. The model represents an abstraction of the real world and describes the enterprise's product range along with its restrictions using the application domain's vocabulary. Importantly, the model based approach clearly separates application logic from configuration knowledge, that is, the implementation of the configurator is fully independent of the application's domain logic. This allows evolving both concerns separately, thereby, greatly simplifying development of the configuration logic and strongly improving maintainability. Moreover, a model-based approach is very well suited of being accompanied by a graphical modeling environment, effectively allowing to realize a fully *model-driven development* (MDD) methodology.

Object-oriented. More specifically, OpenConfigurator's modeling approach is build on top of the *object-oriented paradigm*. The mechanisms provided by this modern, widespread programming paradigm can be adequately used to represent product and configuration knowledge (see Section 2.3.3.2, "Product Architecture"). Well known concepts such as generalization, composition and others allow the definition of product structures in an extremely compact manner. Furthermore, encapsulation and information hiding are important aspects in respect to re-use and maintainability. Finally, other important benefits of object-orientation are the availability of a fully integrated development methodology, from *object-oriented analysis* (OOA) over object-oriented design (OOD) to object-oriented programming (OOP), including a rich set of development tools. Last but not least, object-orientation has a strong reputation in the community.

Universal and generic. Another important fact about OpenConfigurator's configuration approach is that it's *universal*. The framework is *not* built for a particular application domain only. Instead, it offers a huge variety of features allowing to model an even larger set of use cases. On the other hand, OpenConfigurator internally maps a particular domain model to a *generic configuration model*, to which the user interface and other services are implemented against. This means, an *application developer*, that aims to implement a configurator on top of the OpenConfigurator framework, can implement nearly arbitrary configuration problems in a *domain specific manner*, while a system integrator, framework programmer or plugin developer aiming at integrating or extending the system, faces a *generic interface*. The huge benefit of this genericness is, that any feature implemented against the generic configuration model can be applied on a large number of configuration problems simultaneously, establishing an "implement it once and for all" way of thinking.

Declarative. The main enabler for the realization of the framework's universality and genericness is the fact, that the modeling approach is *declarative*. While the application specific domain model is designed using object-oriented concepts (i.e. Java language concepts in particular, see below), the model is enriched with configuration specific knowledge using

declarative meta-data annotations (i.e. Java source code annotations). This approach is well known to Java developers having worked with any recent Java technology. Since the meta-data annotations are pre-defined, the framework is capable of interpreting configuration models, that itself utilize the vocabulary of the application domain. Hence, the framework doesn't impose any specific structure of the domain model (e.g., no base classes provided by the framework must be extended). This opens the door for re-using the one and the same domain model in different application contexts.

Java based. Although not necessarily required, the framework is designed and developed entirely *Java based*. That means, it is implemented on top of the Java platform and its supporting technologies. In fact, Java is the single, major technology used for all aspects of configurator implementation: from the definition of the domain model and its constraints, over the implementation of the framework itself, up to the implementation of the user interface, everything is written in plain Java. The huge benefit of this is, that the developer is not forced to learn a new, possibly proprietary programming language, but instead faces a single, homogeneous software technology stack without a "media break". Even more, the strong alignment and integration of OpenConfigurator's development approach with existing, state-of-the-art Java technologies (e.g., JavaBeans, JPA, Bean Validation, etc.) leads to a flat learning curve and enables the average Java developer to get started with the framework quickly.

The Fundamental Idea of OpenConfigurator

The fundamental idea behind our configuration approach is, that:

- **configurable products are represented as (Java) classes,**
- **concrete configurations correspond to instances of those classes (objects)** and consequently,
- the **configuration process** (in the sense of Section 3.2.2.2, "The Configuration Process as a Transformation Process") is understood as the **instantiation of a (Java) class**.

This idea is visualized graphically in Figure 4.1, "The Fundamental Idea behind the OpenConfigurator Approach".

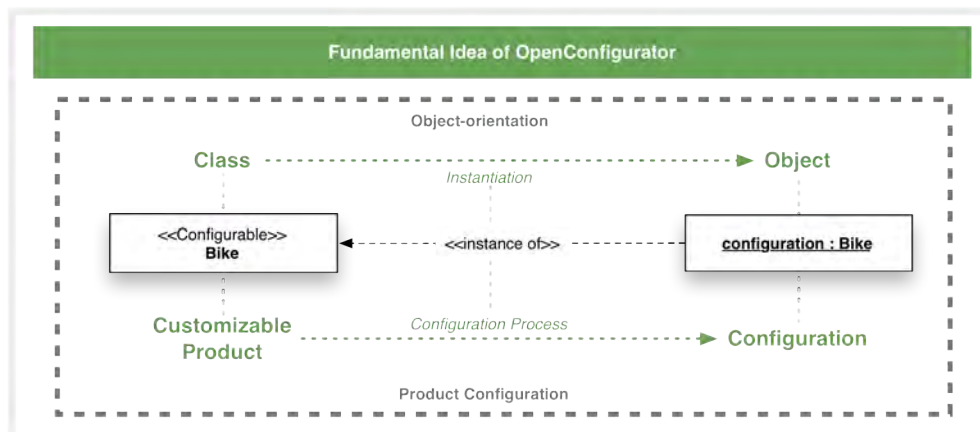


Figure 4.1. The Fundamental Idea behind the OpenConfigurator Approach

The main task of the OpenConfigurator framework, is to realize the configuration process. Or, in other words, to drive the instantiation process of domain model classes. Hence, the result of the configuration process is a regular (JavaBean) instance of a configurable class.

We'll cover all important aspects of this idea in great detail throughout this chapter.

4.1.2. Configurator Development

In order to provide a "big picture" of the configurator development strategy, fostered by our approach, we'll give an overview of our methodology in this section, see Figure 4.2, "Overview of OpenConfigurator Methodology". We will shortly explain the depicted procedure below. In this context, we'll narrow the scope of the conceptualization described in the rest of this work.

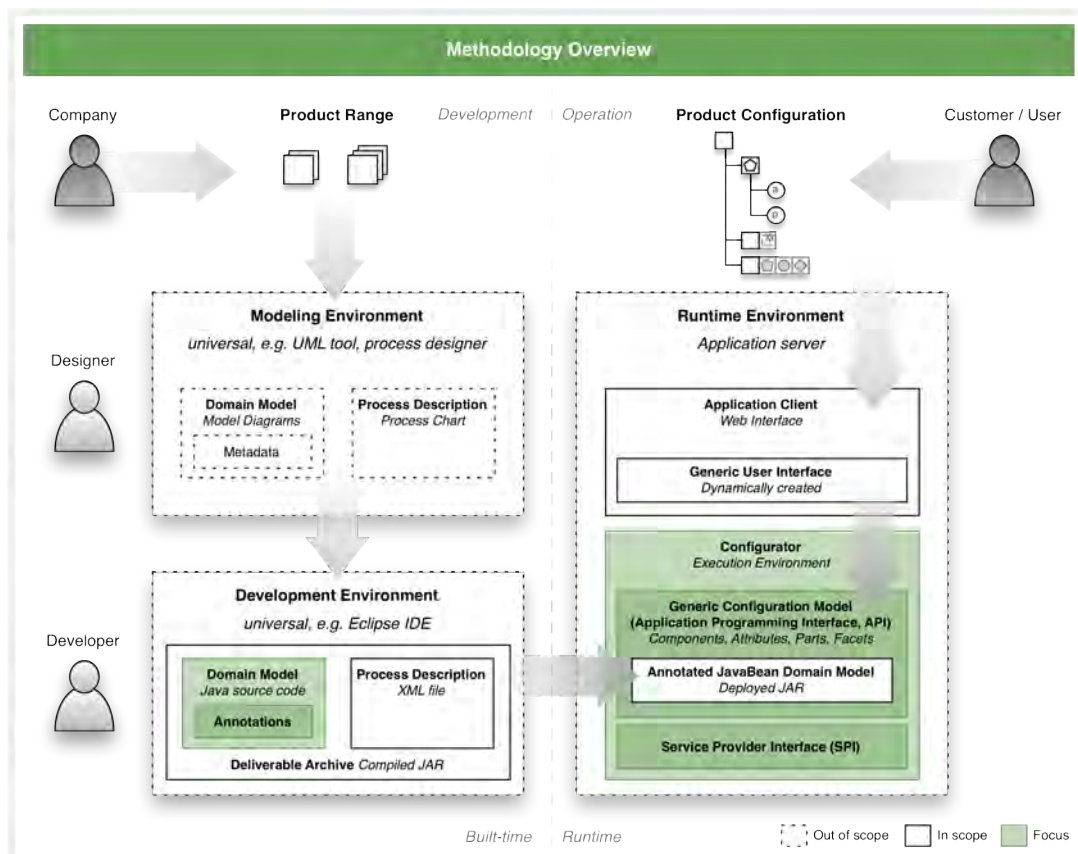


Figure 4.2. Overview of OpenConfigurator Methodology

A configurator development process typically involves multiple roles, including company representatives, designers and developers. It is the company representative that defines the **product range** or parts of it to be covered by the configuration system.

The next step is the **modeling of the application domain** in a computer-readable format. This task is optimally performed by the domain experts themselves, which hold a deep understanding of the domain knowledge. Since they usually do not have programming skills, though, this task is at best supported by software tools, i.e. a modeling environment. While graphical modeling support is out of the scope of this work, in general, the overall methodology is designed to be supported by such modeling tools. OpenConfigurator's object-oriented, model-based nature allows to leverage existing, universal UML tools without compromise. Such tools enable the graphical design of the product range using class diagrams and support the addition of the required meta-data through UML annotations or OCL constraints out-of-the-box (see [Felfernig2000]). The code generation facilities of modern UML tools can then be utilized to generate the Java source code of the domain model, as required

by the OpenConfigurator framework¹. A similar approach can be applied for the customization of the configuration process flow².

In our methodology, the **developer solely implements the product domain model** using Java classes and enriches the code with configuration specific Java annotations, defined by the OpenConfigurator framework. The source code annotations give the product model its semantics in terms of the configuration logic. So, as long as the graphical modelling support is not available, it is the developer that designs the domain model directly using Java classes within his preferred integrated development environment (IDE)³. Powerful tools such as Eclipse⁴ offer sophisticated support for rapidly implementing such domain models. The development experience can additionally be improved by leveraging Java 6's annotation processing facility⁵. It can be used to ensure model validity during source code writing in real-time⁶.

The **deliverable** provided by the developer (aka the development process), is a single archive, solely containing the product model including constraints as compiled Java classes, plus some optional descriptor files.

The architecture foresees, that this single Java archive (JAR) is deployed into the *generic configurator application*, which acts as an **execution environment** for the configuration process.

The model contained within the deployed JAR file is introspected *at runtime* and a **generic, in-memory representation of the configurable domain** is constructed. This in-memory representation is referred to as *generic configuration model* and can be seen as an abstraction layer, that builds the foundation for both the generic user interface and the service provider interface (SPI).

That means, the configurator's user interface is *not* implemented for a particular configuration problem only. Instead, it is a **generic user interface, automatically and dynamically created** on top of (i.e. with information provided by) the generic configuration model. A consequence of this novel approach in the area of configurators is, that, for instance, a user interface client implemented against the generic configuration model is **capable of visualizing and driving arbitrary configuration processes**. The Apple iPad⁷ application client introduced in Chapter 6, *Evaluation and Validation*, is implemented against the generic configuration model, allowing any custom product being configured on the iPad mobile platform. The same strategy works for service providers integrating the framework with other enterprise applications, such as ERP, CAD or PDM systems. All these interfaces interact with generic, abstract representation of the configuration model.

Internally, the generic configuration model manages **concrete instances of the domain model classes**. Hence, at any time, a regular Java object (also referred to as *Plain Old Java Objects*, POJOs) of the domain model can be retrieved from the configurator using the configuration API. This way obtained objects can be used with any other Java technology operating on POJOs, such as JAXB⁸ for XML processing, JAX-WS⁹ for webservice interoperability or JPA¹⁰ for

¹Note that custom transformation rules, that correctly transform UML meta-data into Java source code annotations, would have to be developed in order to complete a fully integrated modeling procedure. However, in our opinion this is generally a feasible task and thus not discussed further in this work.

²Note: the OpenConfigurator framework currently doesn't support the explicit design of the configuration process flow.

³Note: we also consider the design of the product model and its realization as source code as *modeling*. Again, in a future version of OpenConfigurator this "modeling on code level" may be replaced by graphical tools.

⁴See <http://www.eclipse.org/>, last accessed June 21th, 2012.

⁵See <http://docs.oracle.com/javase/6/docs/technotes/guides/apt/index.html>, last accessed June 21th, 2012.

⁶Note that the development of such an annotation processor is also subject to some future work.

⁷See <http://www.apple.com/de/ipad/>, last accessed June 21th, 2012.

⁸Abbrev. Java Architecture for XML Binding, see http://de.wikipedia.org/wiki/Java_Architecture_for_XML_Binding, last accessed June 21th, 2012.

⁹Abbrev. Java API for XML Web Services, see http://de.wikipedia.org/wiki/Java_API_for_XML_Web_Services, last accessed June 21th, 2012.

object-relational persistence (to name just a few). This offers great **integration and interoperability** potentials for configurators implemented on top of the OpenConfigurator framework out-of-the-box.

Additionally, the methodology is designed to foster *single source development*. This means, that the Java domain model is considered the single source of information (excepts the descriptor files) required for the configurator to facilitate product configuration. Together with the DRY¹¹ principle, in our opinion, this greatly improves the developer experience and eases maintainability simultaneously.

Also, it's important to mention, that even though our methodology is *model-based*, it **does not require built-time code generation**. Nevertheless, code generation, in the sense of model-driven development, may be a useful complement to the framework in the future. In the context of this work, however, we will focus on the modeling capabilities required for configuration problem design, as well as the runtime execution environment interpreting the domain model.

4.1.3. Other Approaches to Configuration

Having characterized our approach, it's worth to take a look at competing approaches to configurator implementation, in order to better understand the novelty and advantages of our methodology. Figure 4.3, “Overview of Other Methodologies” provides an aggregated overview of methodologies, that are subject to science or applied in current business projects.

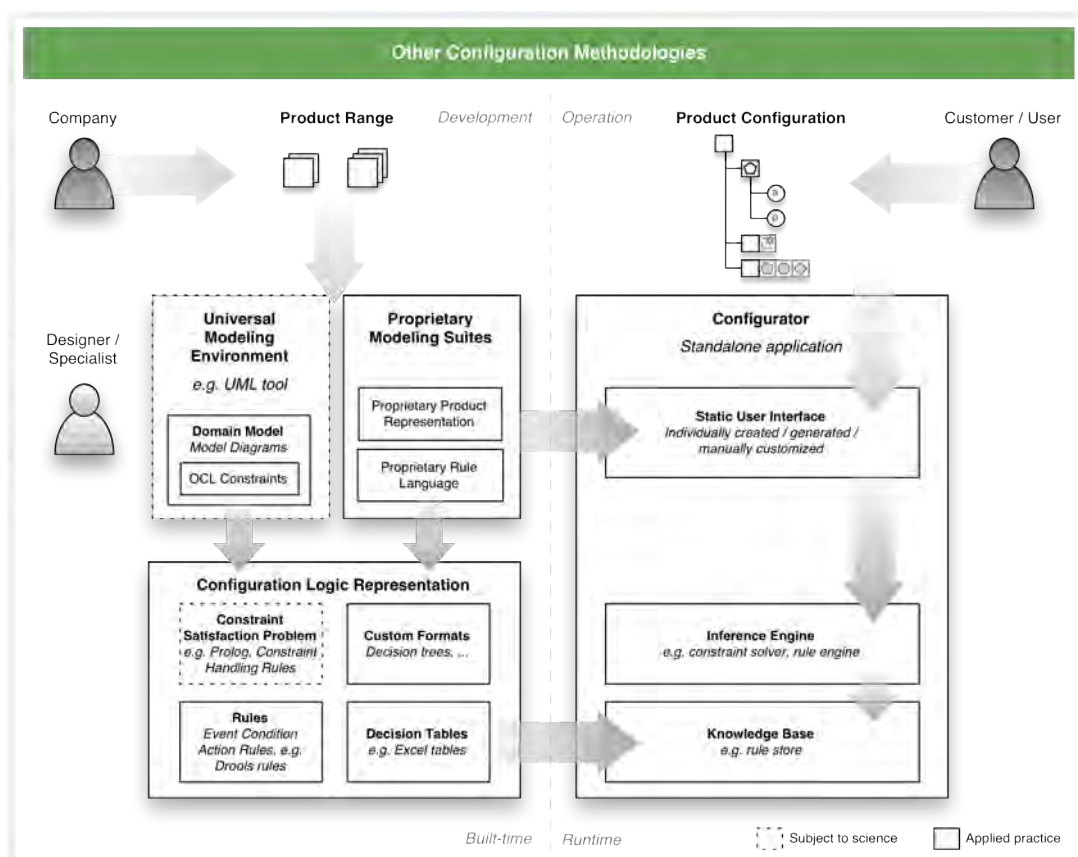


Figure 4.3. Overview of Other Methodologies

¹⁰ Abbrev. Java Persistence API, see http://en.wikipedia.org/wiki/Java_Persistence_API, last accessed June 21th, 2012.

¹¹ Abbrev. Don't Repeat Yourself, see <http://en.wikipedia.org/wiki/DRY>, last accessed August 1st, 2012.

Existing configurator platforms mostly offer a dedicated, graphical modeling environment to be used by specialists to define the configuration logic. Commercial products (cp. [Rogoll2003]) we've seen so far, offer **proprietary modeling suites**, that support the definition of product models using custom, vendor-specific formats as well as proprietary rule definition languages. The consequence of this is a strong **vendor lock-in**, requiring tight relationships to and imposing a strict dependency on the system provider. Particularly, custom developments and extensions to the system, sometimes desired to address specific configuration requirements, may quickly lead to uncapped costs due to the lack of competition, once a particular software supplier has been chosen.

Approaches currently subject to science, on the other hand, are implemented solely in prototype applications. But go one step further: they utilize UML/OCL¹² and related tool chains for designing configuration knowledge bases. These in fact, aim to establish a fully model-driven approach (see [Felfernig2000]).

However, in both cases, the **configuration model is transformed into a dedicated configuration logic representation format at built-/development-time**. These representation formats include rules, decision-tables or custom formats. The employment of truly constrained based approaches, that map the configuration problem to constraint satisfaction problems, interpretable by constraint solvers, has been rarely seen in practice, but is still subject to the science of artificial intelligence, yet. Nevertheless, the built-time transformation into another intermediate knowledge representation, in our opinion, turns out to be a **paradigm shift** and thus a "media break". The average programmer isn't capable of understanding or dealing with such an intermediate format (e.g., Prolog rules, business rules, etc.) within a reasonable amount of time. This is especially relevant, when it comes to errors and debugging or software modifications, integrations and extensions.

In many approaches, the result of the development process is a **monolithic, standalone configurator application**, generated specifically for a single project. This application often features a generated, rather static user interface (e.g., in terms of Java Server Pages, JSPs), which is manually customized upon changes of the knowledge base. The configuration interface interacts with an **inference component**, which itself encapsulates the knowledge base on which it operates.

4.1.4. Advantages of OpenConfigurator

From our elaborations in the previous section, we argue that the approaches available on the market today, face the following problems:

- proprietary modeling concepts and rule definition languages,
- paradigm shifts between multiple configuration problem representations
- built-time generation of a more or less static, monolithic runtime application

These issues ultimately result in:

- a strong vendor lock-in and a dependence on proprietary formats
- a steep learning curve and a lack of developer friendliness
- an overcomplicated, complex configurator development process
- strong limitations regarding extensibility, integrability and interoperability

In contrast, our approach offers the following features:

- a modelling approach fully utilizing the generally accepted, wide-spread, object-oriented programming pattern
- a single paradigm and a homogeneous technology stack based on modern Java concepts, standards and technologies

¹²Abbrev. Unified Modeling Language/ Object Constraint Language.

- a modeling approach strongly aligned with state-of-the-art technologies and development methodologies accepted in the Java space
- a highly extensible, integrable and interoperable execution environment
- a flexible user interface that fully automatic, dynamically adapts to the underlying configuration model

The result of these capabilities are huge potentials, beneficial in various regards:

- a flat learning curve, ease-of-development and an overall developer friendliness
- a quick turnaround due to the possibility to employ standardized, efficient development cycles
- a high return on investment (ROI) due to reusability of the model and dynamic adaptability upon model changes

Having characterized OpenConfigurator's general methodology in this section, including a comparison with other approaches and a final look at its benefits, in the remainder of this chapter we will show how configuration problems can be modeled using our conceptualization.

4.2. Modeling Approach

The foundation of our approach for modeling configuration problems is *object-orientation*¹³. In the sense of the categorization discussed in Section 3.3.3.5, "Implementation Aspects", it can be further characterized as follows:

Model based, domain specific. The configuration problem is modeled separately from the application logic, in a domain specific manner. The configuration model in our methodology is simply a *Java domain model* comprising an arbitrary number of classes conforming to the JavaBean convention¹⁴. With the help of these classes, the customizable product is described including any restrictions.

The object-oriented nature of our approach allows to map structured product architectures (see Section 2.3.3.2, "Product Architecture") one-to-one to the OpenConfigurator required representation of the configurable product.

How configurable products are described on code level concretely, will be described in Section 4.2.2, "Product Models in Java".

Component based, generic. Internally, the Java domain model is transformed and interpreted as (technically we say "wrapped by") a *generic component model*. The applied generic component model, thereby, complies one-to-one with the architecture worked out in Section 3.1, "Product Models", respectively can be seen as an extension of it.

Hence, a customizable product is represented as a set of configurable *components* containing arbitrary nested components (so called *parts*) and customizable (variable) *attributes*. The configuration decisions explained in the respective sections of Section 3.1, "Product Models" equally apply to our model. In fact, they build the basic foundation for our configuration process, too (see Section 3.2.2.2, "The Configuration Process as a Transformation Process" for detailed theoretical basics).

In Section 4.3, "The Generic Configuration Model" we will explain the generic configuration model in explicitly. Section 4.3.2, "Model Mapping" shows, how Java domain models are "transformed" into their generic representation.

¹³Our modeling approach has been strongly influenced by the works of Felfernig et al., who investigated UML as modeling technique for configuration knowledge bases more than 10 years ago [Felfernig2000].

¹⁴See <http://en.wikipedia.org/wiki/JavaBeans>, last accessed August 1st, 2012.

Constraint based. Our approach utilizes *boolean constraints* as well as *relational constraints* (see Section 3.1.3, “Constraints: Domain Restrictions”) in order to express configuration restrictions declaratively.

While Section 4.2.3, “Providing Meta-Data with Java Annotations” basically shows, how meta-data (including constraints) is expressed declaratively, in Section 4.4.5, “Constraint Modeling” we will present the available constraints concretely.

As our methodology combines multiple explicit techniques (see Section 3.3.3.5, “Implementation Aspects”), we consider it as *hybrid* configuration approach. For a full characterization refer to Section 7.1.2, “Implementation Characterization” in Chapter 7, *Summary and Outlook*.

Before presenting the concrete conceptual elements provided by the OpenConfigurator framework, we are going to explain some important modeling basics.

4.2.1. Object-Oriented Product Modeling

As described in Section 2.3.3.2, “Product Architecture”, the basic principles of the object-oriented programming paradigm fit very well for the description of product models. In this sense, the following mapping between the object-oriented and the product modeling world (see Section 3.1, “Basic Meta Model for Generic Product Modeling”) is considered to be straightforward, although two required concepts for product configuration aren’t available directly:

Table 4.1. Mapping between Object-oriented Concepts and Product Modeling Concepts

Object-Oriented Concept	Product Modeling Concept
Class	Product / component type
Generalization / specialization	"Is-a" / "kind-of" relationship
Aggregation / composition	"Contains-a" / "part-of" relationship
Object	Product / component instance, variant
Field (member variable)	Attribute
Member type (<i>incomplete correspondence</i>)	Attribute value domains
Method	Calculated attribute
Type boundaries (<i>incomplete correspondence</i>)	Constraints

Class. With the help of *classes*, product families can be modeled. A single class corresponds to a certain product or component type.

Generalization/specialization. Utilizing the concept of inheritance, *generalization/specialization* relationships can be expressed, which allows to model taxonomies of products and components. E.g., a *Mountain Bike* and a *City Bike* are both specializations of the more general product *Bike* or both a *V-Brake* and an *Hydraulic Brake* are specialized types of a *Brake* component.

Aggregation/composition. *Aggregation* respectively *composition* relationships can be used to model the compositional structure of a class of products. For instance, a *Bike* product can be decomposed into several parts, such as *Frame*, *Handle bar*, *Wheels*, *Gearing*, and so on.

Object. An *object*, an instance of a class, is equivalent to a product instance. It can also be seen as a specific variant of a product. An object encapsulates an internal state exposes methods to the outside world, that operate on or manipulate an instance’s state.

Field (member variable). For that purpose, classes typically define *fields*, respectively *member variables*, which correspond to attributes in the product world. They are used to de-

scribe the characteristics of a class and represent the state of an object, respectively the value of a specific product attribute. For instance, the frame of a bike may have an attribute *Color*, which in the object-oriented world would correspond to a member variable *color*.

Member type. Attributes have a value of a certain type, e.g., boolean, integer, string etc. The *member type* effectively defines the values, that can be assigned to that variable. Thus, the member type can be considered as the attribute's domain (see "Domain" in Section 3.1.2, "Attributes: Component Characteristics"). However, in order to express product models precisely, using member types only to define attribute domains is not sufficient. The values must be definable much more precisely.

Method. *Methods* are used to define functions or procedures related to an object. In the same way, they can be seen as realization of calculated attributes of a product (see Section 3.1.2, "Attributes in the Context of Configuration"). E.g., the *weight* attribute of a bike may be calculated by a method.

Type boundaries. In general, when describing product attributes with object-oriented types, the type's technically imposed *boundaries* can be considered as constraints: effectively, they restrict the value an attribute can take. For instance, if a field has type byte, a value in the range of $-/+255$ can be assigned, but the value 4711 cannot. Thus, the attribute corresponding to that field is (implicitly) constrained by the field's type.

Figure 4.4, "Example Object-Oriented Product Domain Model" shows an example model¹⁵ for the *Bike* domain. While an elaborated version of the domain model is presented in Section 6.1.3, "The Bike Domain Model", we'll use excerpts in various places of this work.

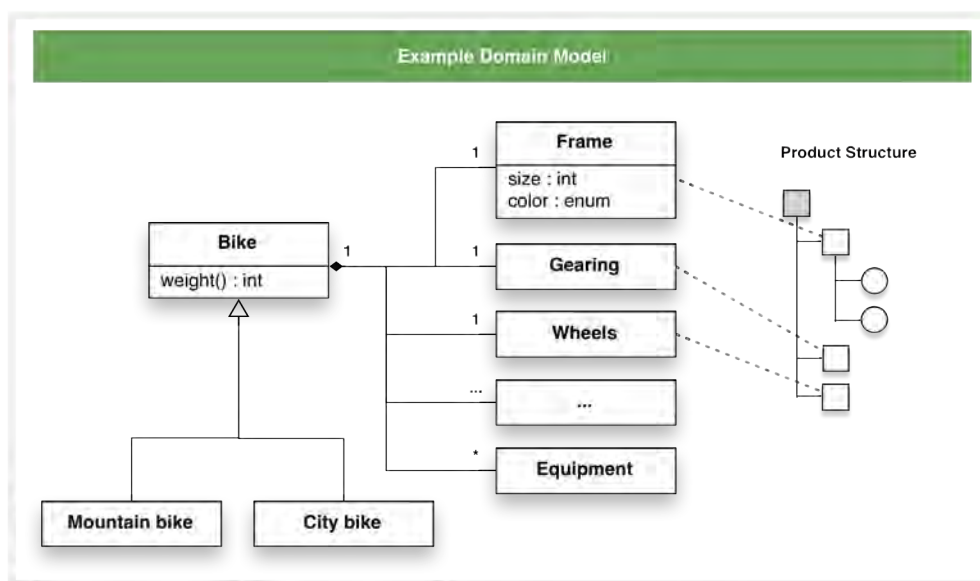


Figure 4.4. Example Object-Oriented Product Domain Model

In summary, with respect to the meta-model presented in Section 3.1, "Basic Meta Model for Generic Product Modeling", object-oriented programming languages like Java provide good support for the majority of required product modeling concepts inherently. However, they do not directly support sophisticated mechanism for modeling *attribute domains* and *constraints*, which is why we need to come up with other techniques to express these. As you'll see shortly, in Section 4.2.3, "Providing Meta-Data with Java Annotations", OpenConfigurator utilizes annotations to fill that gap.

¹⁵We use an UML like notation for domain models throughout this work.

4.2.2. Product Models in Java

Having described product modeling using object-oriented features *in general*, in the following we will focus on modeling products using the Java programming language¹⁶ in particular. In this context, the JavaBean convention¹⁷ can be used for component modeling. Most importantly, the convention standardizes accessors for fields, so called *getter* and *setter* methods. This way it establishes a common way to model the notion of *properties*¹⁸. The following code snippet, adhering to the JavaBean convention, shows the Java source code for the example domain model introduced above¹⁹:

Example 4.1. Example JavaBean Domain Model

```
public abstract class Bike {

    // Fields
    private Frame frame;
    private Fork fork;
    private Wheels wheels;
    private Collection<Equipment> equipments;

    // Methods
    public int weight() {
        // ...
        return calculatedWeight;
    }

    // Accessors
    public Frame getFrame() {
        return frame;
    }

    public void setFrame(Frame frame) {
        this.frame = frame;
    }

    ...
}

public class MountainBike extends Bike { ... }
public class CityBike extends Bike { ... }

public class Frame {

    private int size;
    private Color color;

    // Accessors
    ...
}

public class Fork { ... }
public class Wheels { ... }
public class Equipment { ... }
```

As you can see, the implementation is straightforward: products and components are mapped to classes, while attributes and parts correspond to JavaBean style properties.

¹⁶See <http://www.java.com/>, last accessed May 24th, 2012.

¹⁷See http://en.wikipedia.org/wiki/Java_Beans, last accessed August 1st, 2012.

¹⁸Note: OpenConfigurator in fact only utilizes the properties mechanism defined by the convention, but does not require the domain class implementing the `java.io.Serializable` interface. Although, the implementation of this (empty) marker interface is recommended.

¹⁹Implementations of the `Serializable` interface have been omitted for brevity.

Again, in terms of OpenConfigurator, configuration problems expressed as product models are implemented entirely in Java. For modeling, all common Java language concepts can be used, including:

Types and inheritance. Java's type system provides classes and interfaces. Both can be used to model products and components. While classes can be generalized using the *extends* keyword, a class may implement multiple interfaces using the *implements* keyword (Java's way to overcome multiple inheritance issues). A class, thereby, inherits all properties and methods from its parent class. Likewise, a product inherits all attributes and parts of its supertype. Additionally, *abstract* types can be used and are semantically fully supported by the OpenConfigurator framework.

Primitives, standard types and enums. For the modeling of attributes, the usual Java primitive types (boolean, byte, int, double, etc.), standard types (like String, Integer, and so on) and type-safe enumerations (using the *enum* keyword) are supported.

Arrays, collections and special types. Additionally, attributes can be array-valued (e.g., String[]) or can be of collection types (e.g., Collection<String>). This way, multi-value attributes can be modeled. Used together with custom types (e.g., the Equipment class), a collection such as Collection<Equipment> designates a multi-component part, also referred to as *plural part*. Moreover, there are some special types, such as byte[] or InputStream, that allow the modelling of special properties like binary images.

Methods. For realizing calculated attributes, regular Java methods can be implemented. Within these methods, arbitrary calculations can be performed without restriction. For example, a component may define a method double `getWeight()` to calculate the weight of a product.

Regarding modeling flexibility, the OpenConfigurator framework tries to be minimal intrusive. That means, when designing a configuration domain model, the developer must not extend or implement any pre-defined class or interface provided by the framework. Instead, the domain model can be enriched with Java annotations in order to the control configuration behavior. The annotations are interpreted by the framework at runtime and incorporated into the generic component model. We will discover annotation usage in more detail in the next section.

4.2.3. Providing Meta-Data with Java Annotations

As stated earlier, in terms of OpenConfigurator, the configuration knowledge as well as the control logic, that drives the configuration process, is described *declaratively*. In fact, a significant aspect of our methodology is how meta-data is provided: using Java annotations²⁰. Indeed, Java annotations are the **key concept of our modeling approach**, as they facilitate the mapping between the Java domain model and the generic product meta model, respectively specify the mapping more precisely.

Java annotations are directly embedded into the source code and can be placed on various locations, including types, methods, fields, parameters and others. They allow the specification of elements with optional default values. Furthermore, they can be inspected at compile time and at runtime, provided their retention is specified accordingly²¹. The following program listing, Example 4.2, "Exemplary Annotation Definition and Usage", shows the definition of a custom annotation named `Description`, that can be used on type, method or field level and which can be inspected at runtime. Moreover, the example demonstrates the application of the custom annotation on a class `Frame`:

²⁰See <http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html>, last accessed August 1st, 2012.

²¹The retention is defined using the `@Retention(RetentionPolicy)` meta annotation. See the example annotation definition below.

Example 4.2. Exemplary Annotation Definition and Usage

```
@Retention(RUNTIME)
@Target({ TYPE, METHOD, FIELD })
public static @interface Description
{
    String value() default "";
}

@Description(value = "The base element of a bike")
public class Frame
{
    ...
}
```

While annotations are frequently used for the specification of meta-data in a growing number of Java standards, including the EJB²², JPA²³ and Bean Validation²⁴ specifications, we argue, that using them in the area of product modeling and product configuration in particular is innovative.

Annotation Usage within the OpenConfigurator Methodology

Within the OpenConfigurator framework, annotations are heavily used. In fact, strictly speaking, *modeling* in terms of OpenConfigurator, comes down to "tagging" the domain model with a variety of pre-defined annotations. Thus, when describing the modeling capabilities in Section 4.4, "Modeling Concepts", we'll basically present the available annotations for product and configuration modeling along with their exact semantics and demonstrate their practical application.

To give an idea on how annotations are leveraged concretely, we want to anticipate the *main areas of annotation usage* in the following. Specifically, annotations are used for:

Meta-model mapping. Annotations are used to map the Java domain model to the generic configuration meta model, which will be described in detail in Section 4.3.1, "Elements of the Generic Configuration Model and Meta Model". This way, e.g., a class is identified as configurable item or specific properties of the class are designated as variable attributes, that need to be specified during the configuration process. The framework falls back to sensible default mappings, in case no annotations are present. Hence, annotation usage is made optional where possible, thereby, providing a great deal of developer convenience. Figure 4.5, "Using Annotations to Map Domains Models to the Generic Meta Model" outlines the usage of annotations for meta-model mapping purposes.

²²Abbrev. Enterprise JavaBeans, see <http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html>, last accessed May 24th, 2012.

²³Abbrev. Java Persistence API, see <http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html>, last accessed May 24th, 2012.

²⁴See <http://jcp.org/aboutJava/communityprocess/final/jsr303/index.html>, last accessed May 24th, 2012.

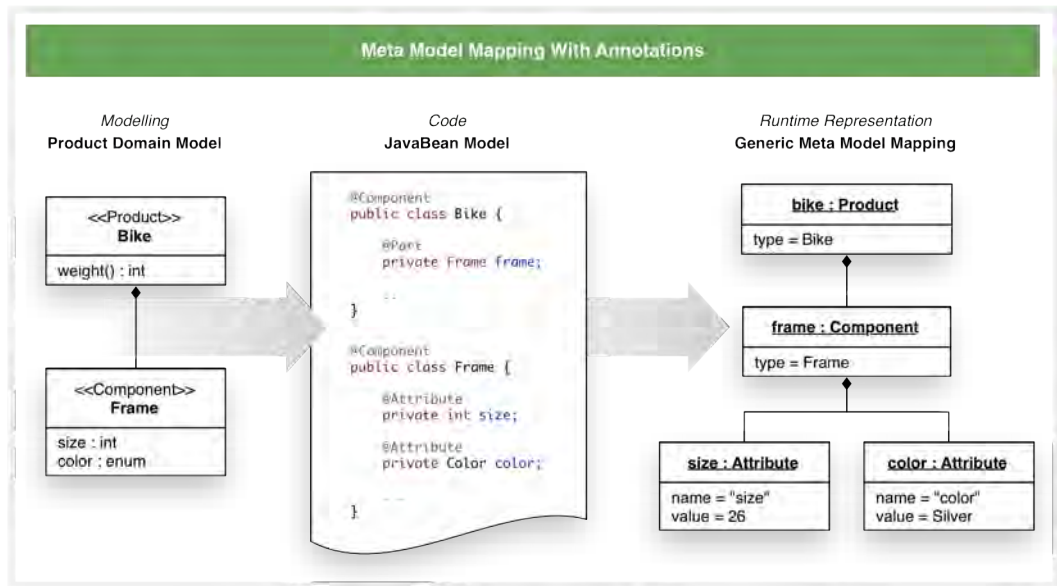


Figure 4.5. Using Annotations to Map Domains Models to the Generic Meta Model

Data provisioning. We stated earlier, that the object-oriented concepts do not provide adequate support for modeling attribute domains, as required for configuration purposes (see Section 4.2.1, “Object-Oriented Product Modeling”). The way OpenConfigurator overcomes this deficit is: using annotations. Beyond meta-model mapping, source code meta-data is used to “attach” data to the domain model, that is, to define the exact domain values of attributes and parts.

Figure 4.6, “Using Annotations for Domain Definition”, illustrates how the domain of an attribute can be narrowed to an enumerated list of values and shows the impact on the configurator's user interface:

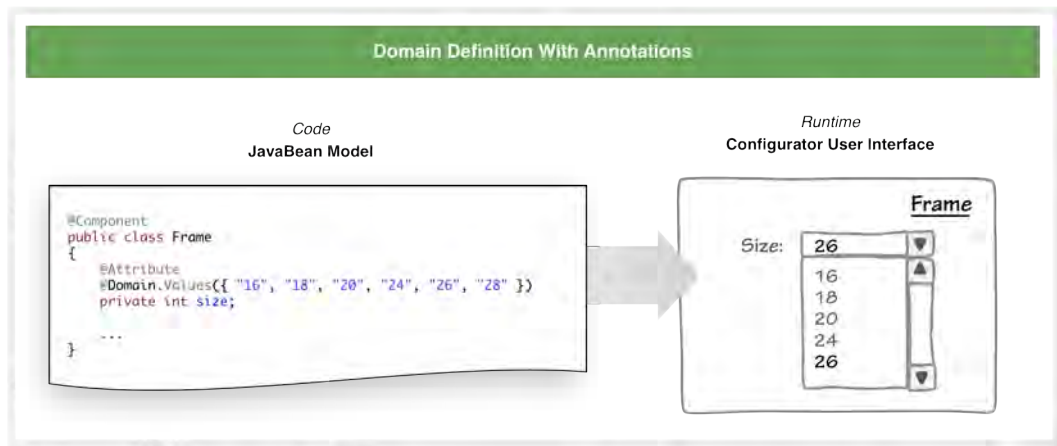


Figure 4.6. Using Annotations for Domain Definition

Constraint definition. Likewise, annotations are used to define constraints within the domain model, thereby, filling the missing gap in the object-oriented programming paradigm. Example 4.3, “Using Annotations to Define Constraints” demonstrates this concept.

Example 4.3. Using Annotations to Define Constraints

```
public class Bike {
    ...

    @Size(max=10)
    private Set<Equipment> equipments;

    ...
}
```

In this case, the quantity of equipment items is constrained to a maximum of 10 items. The definition of constraints in this way has been adapted from JSR²⁵-303²⁶, a specification, that standardizes the definition of constraints on Java classes across various application areas. In fact, OpenConfigurator strives to be compatible with the Bean Validation specification where possible, but needs to extend it in various regards, though.

User interface mapping. With annotations, also the visual appearance of the configurator could be manipulated declaratively (see also Section 4.3.3, “Facets”)²⁷. For example, the developer may specify, that a list of enumerated values is rendered as group of radio boxes instead of a single combo box, which is the default. This is demonstrated in Figure 4.7, “Using Annotations for Declarative User Interface Mapping”.

While it can be argued, that this strategy intermixes user interface aspects with application domain logic, in our opinion the following arguments advocate this feature:

- **Single source development.** In Section 4.1.2, “Configurator Development”, we explained, that OpenConfigurator fosters the concept of *single source development*. In accordance with the DRY principle, this means, that the Java source code should remain the “single source of truth” and contain all information to control the application. If the developer seeks to alter the configuration process’ behaviour, he solely needs to inspect the source code and no additional source files.

Hence, defining user interface mapping behavior using annotations, supports the principle of single source development.

- **Declarative, portable user interface mapping.** Using annotations for user interface (UI) mapping, declaratively describes, how a particular element is visualized on *any* and *not only a single* interface. Thus, the *declarative description* not only avoids code dependencies on a particular UI technology, but even more ensures portability across a variety of different interfaces, such as mobile device, desktop application, web page etc.

Thus, UI mappings defined this way are user interface technology agnostic, which we see as a strong benefit.

²⁵Java Specification Request

²⁶See <http://jcp.org/aboutJava/communityprocess/final/jsr303/index.html>, last accessed May 24th, 2012

²⁷Note that this feature is currently not implemented.

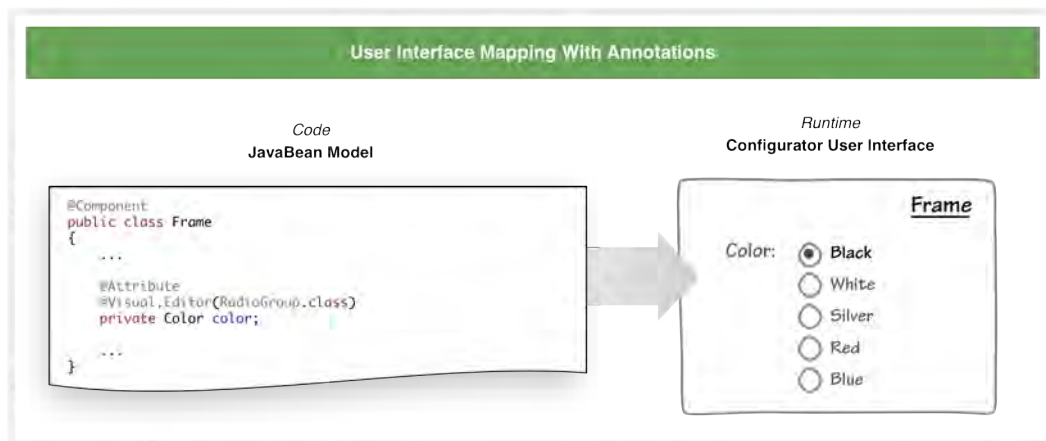


Figure 4.7. Using Annotations for Declarative User Interface Mapping

The aforementioned annotations are only a small extract from the full set of possibilities, which are described in more detail in Section 4.4, “Modeling Concepts”. Basically, annotations are used to **control the entire configuration process**. With their help, the domain model becomes the single source of information for the generic configurator framework.

To realize this, the OpenConfigurator framework extracts the specified meta-data at application runtime. During this process, the *domain specific* product model is transformed into a *domain independent* representation, the so called *generic configuration model*. We will explain this generic model as well as the mapping between both representations in more detail in the next section.

4.3. The Generic Configuration Model

The *generic configuration model*, which can be seen as a **domain independent representation of the configurable product**, is one of the most fundamental concepts of our approach. Described technically, it is a runtime data structure, that builds the “backbone” of the entire configuration process, as it supplies and stores all relevant information related to the current configuration. This relationship is schematically illustrated in Figure 4.8, “The Generic Configuration Model as “Backbone” of the Configuration Process”.

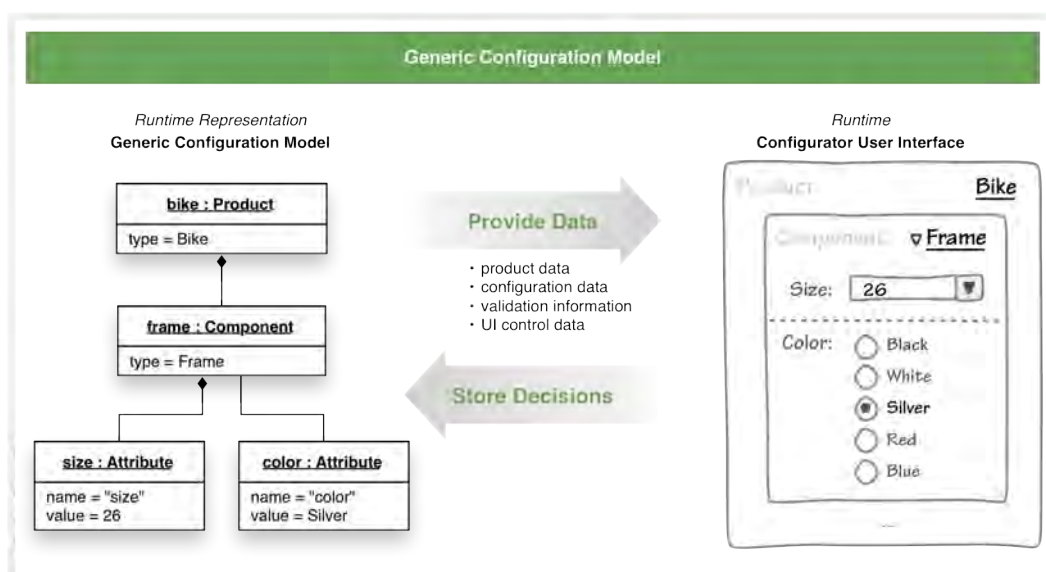


Figure 4.8. The Generic Configuration Model as “Backbone” of the Configuration Process

The generic configuration model has its roots in the product model described Section 3.1, “Product Models”. Essentially, it features the same concepts, such as *components*, *attributes* and *constraints*, but extends the model with *parts* (nested components) and facets (“perspectives” to a component). A detailed description of the internal representation of the configuration model is important for a well understanding of the modeling concepts described later on. Therefore, we will describe the model elements in detail in Section 4.3.1, “Elements of the Generic Configuration Model and Meta Model”. Facets will be subject to Section 4.3.3, “Facets”.

Representations of Configurable Products

In terms of OpenConfigurator, configurable products are described on *different levels*. Moreover, the framework knows *multiple representations* of one and the same configuration, whereas the generic configuration model is one of such representations.

Levels. First of all, we have to distinguish the *type* level from the *instance* level.

- **Type level.** As stated in Section 4.1.1, “The Fundamental Idea of OpenConfigurator”, with our approach, configurable products are described using Java classes. As Java *types* are compiled, they’re considered *static* throughout the lifetime of the application. Hence, the *type level* is concerned with the *static view* of the configuration domain model.
- **Instance level.** On the other hand, concrete configurations correspond to Java objects. At runtime, Java types (configurable products) are *instantiated* in terms of Java objects (concrete configurations). These instances make up the *dynamic* part of the application, respectively the *instance level*.

Again, it’s the task of the OpenConfigurator framework to instantiate a particular Java class, representing a customizable product, during the configuration process. In order to accomplish this, the framework maintains different representations for both the static product types and the dynamic configuration items.

Representations/views. Specifically, the distinguishable representations are:

- **Logical view.** The logic view precisely realizes the main idea of the OpenConfigurator approach: items configured by the container are true instances of the domain model classes. The model is *domain specific*, which was one of the fundamental requirements to be fulfilled by our approach (refer to Section 4.2, “Modeling Approach”).
- **Internal representation.** Another fundamental requirement was *genericness*. For example, genericness applies to the user interface: the configurator developer shall not implement the user interface from scratch, but rather concentrate on modeling the domain logic. The framework provides a *generic user interface* for arbitrary domain models (see Section 4.1.2, “Configurator Development”).

To realize this genericness, the configurator must maintain an *internal representation* of the configured product. Opposed to the logical view, the internal representation has a *generic interface* (which gives the model the name “generic configuration model”) and is thus *domain independent*.

At runtime, the OpenConfigurator continuously synchronizes the logical view with the internal representation²⁸.

Relationships between the Different Levels and Representations

The relationships between these different concepts are depicted in Figure 4.9, “Role of the Generic Configuration Model and its Relationships”.

²⁸Note: currently only a “one-way synchronization” is implemented: changes to the generic configuration model are “pushed” to the managed domain model instances, but not vice versa.

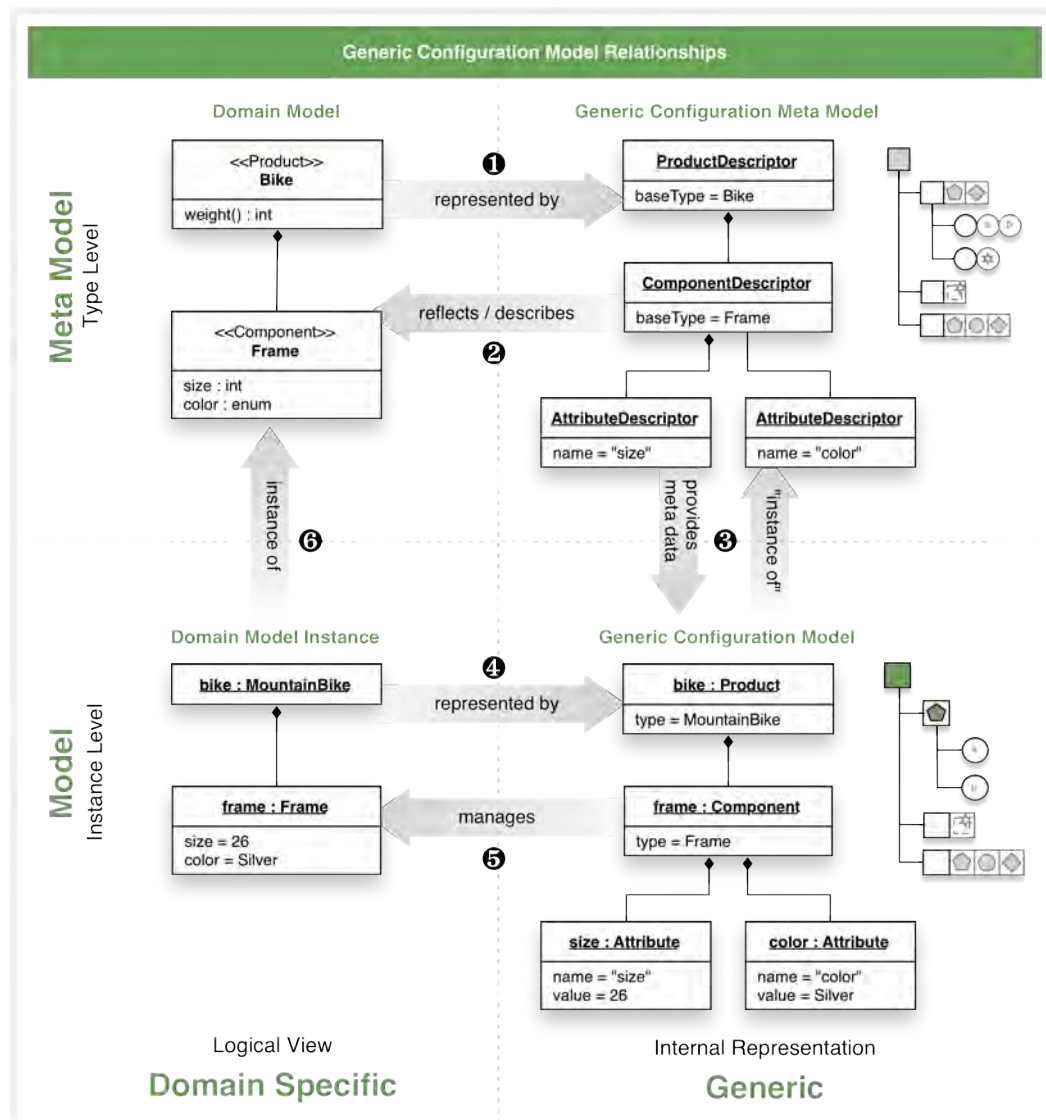


Figure 4.9. Role of the Generic Configuration Model and its Relationships²⁹

The generic configuration model represents the domain independent, *internal representation* of the configuration, which is the view, the configurator itself (and all third party extensions) deal with exclusively. On the other hand, there is the domain specific, *logical view* to the configuration model.

The application specific *domain model*, supplied to the configurator in terms of a Java class model (see Section 4.1.2, “Configurator Development”), is represented at runtime by the *generic configuration meta model* ①. The meta model contains so called *descriptors*, that reflect all relevant, static aspects of the domain model and which describe its decomposition structure ②. It also incorporates any meta data supplemented through model annotations. We refer to the process of transforming the annotated domain model into the generic product meta model as *meta model extraction*.

During the configuration process, a *generic configuration model* is instantiated based on the meta model. Thus, semantically, the generic configuration model is an instance of the meta

²⁹Note that the graphic has been slightly simplified for brevity: `PartDescriptor`s and `Parts` (see Section 4.3.1, “Elements of the Generic Configuration Model and Meta Model” for details) have been omitted and the `ProductDescriptor` doesn't exist exactly as depicted; nevertheless, the figure provides a *semantically correct* overview.

model ③. The meta model can be seen as a static description on the domain model (*type level*). It exists only once in the application.

However, each product instance, is represented by exactly one dynamically modified, corresponding generic configuration model (*instance level*) ④. This model in turn, manages the underlying JavaBean instance, that is, the logic view ⑤.

As you can see, in effect, the configurator just "assists" the user during the instantiation of a particular domain model class ⑥. More precisely, he drives the instantiation process in a way that all constraints, defined within the domain model, are satisfied.

Next, we'll examine the generic configuration model respectively the meta model in more detail.

4.3.1. Elements of the Generic Configuration Model and Meta Model

In the following, we will describe the elements of the generic configuration model and related descriptors more precisely. Figure 4.10, "UML Class Diagram of the Generic Configuration Model and Related Descriptors" provides a simplified UML model of the classes involved.

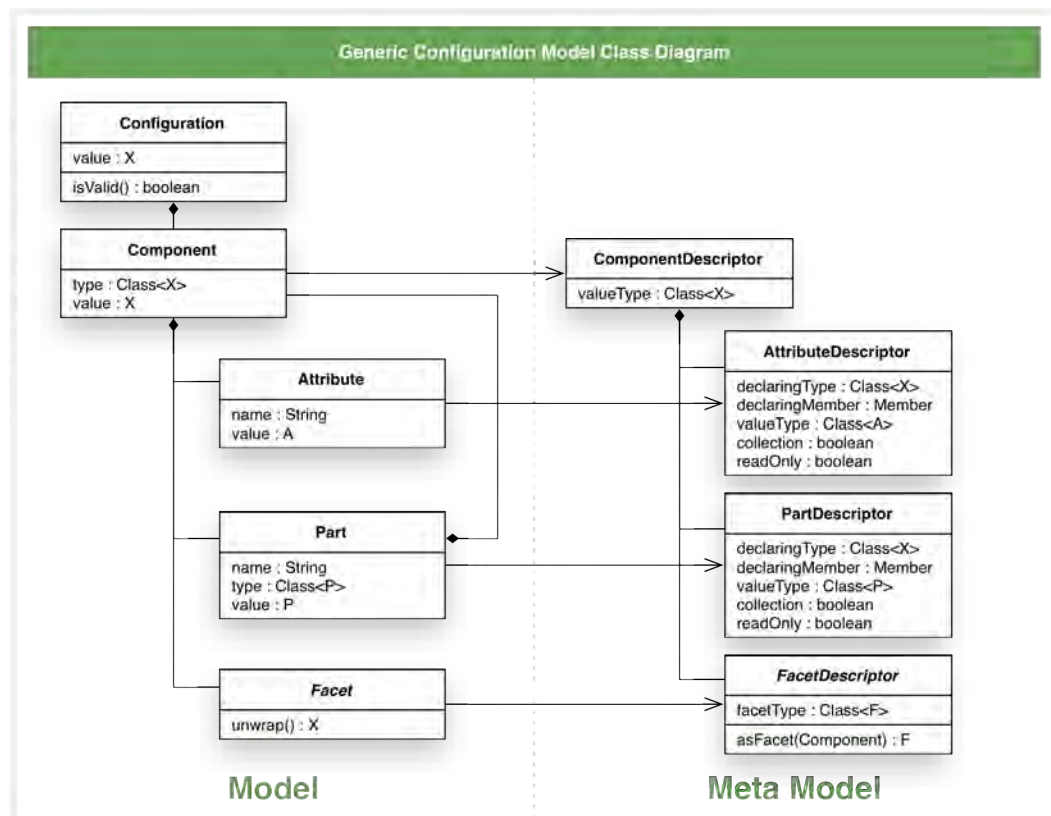


Figure 4.10. UML Class Diagram of the Generic Configuration Model and Related Descriptors

Configuration. The Configuration class is a wrapper, that represents a configuration at runtime. It allows to access the configured instance (logical view) via the `value` attribute. A configuration contains a single configurable component called `rootComponent`.

Component/ComponentDescriptor. The Component class is the central type of the component hierarchy. It represents a configurable product or product component. A component

stores (if already instantiated) an instance of the configured type `X` in its `value` attribute, which is sometimes referred to as *local value*. It aggregates a varying number of `Attribute` and `Part` instances, depending on the configured type. The current type of the maintained product instance is defined by the `type` attribute. Furthermore, a component may provide multiple `Facets` (see below).

A component references the descriptor that exactly matches the type `X`. As the type of an instance may be altered dynamically during configuration, the referenced component descriptor may also change. The attributes and parts are synchronized accordingly.

Attribute/AttributeDescriptor. Represents a single, non-complex property of a configurable object. Opposed to parts, an attribute is not further divisible and thus simply holds a value of the defined property type. The type is often referred to as *value type*.

An attribute references an `AttributeDescriptor`, which holds meta information about the Java class' member, that defines this attribute. While `declaringType` designates the Java class that declares the property, while `declaringMember` identifies the exact member. That is, `declaringMember` either points to a field or a (getter) method. If the property's `valueType` is a collection (in the sense of an instance of Java's collection framework) or an array type, the attribute is considered a *multi-value attribute*, which is reflected by the descriptor's `collectionMemberVariable`. If there's no setter method, but a corresponding getter for one and the same property, the attribute is considered `readOnly`.

Part/PartDescriptor. A `Part` represents a complex property of a configurable object. That is, a part references a sub-component of the given product instance and thus allows to model recursively nested component structures. Due to the fact that a `Part`, in turn, references a `Component` instance, it can be configured as any top level component is, including dynamic type changes. The current part's type is stored in the corresponding type member variable, and (if present) the local value in the `value` variable.

Similar to components, parts reference `PartDescriptor` instances that reflect their current type. The descriptor holds information equivalent to the one of an `AttributeDescriptor` (see above).

Facet/FacetDescriptor. To not overload the component model with manifold information required for configuration, the concept of *facets* has been introduced. A `Facet` encapsulates a certain *aspect* of a component. This way, the core component model can be kept clean and simple, containing only the 3 main types, namely `Component`, `Attribute` and `Part`. Any additional features, such as attribute domains or constraints, are implemented within a particular facet. Examples for facets include:

- **Product facet.** Allows accessing the component in the sense of a product.
- **Configuration facet.** Provides configuration relevant information.
- **Data facet.** Implements a generic data abstraction layer. Allows accessing a component's or attribute's domain.
- **Validation/constraint facet.** Encapsulates constraints and validation relevant aspects.
- **User interface facet.** Stores user interface mapping related information.

Facets usually also require a static descriptor, the `FacetDescriptor`, that describes how the facet relates to the component precisely. We will describe facets in more detail in Section 4.3.3, "Facets".

4.3.2. Model Mapping

We described domain modeling using object oriented techniques and Java in particular in Section 4.2.1, "Object-Oriented Product Modeling" and Section 4.2.2, "Product Models in

Java”. In the last section, Section 4.3.1, “Elements of the Generic Configuration Model and Meta Model”, we introduced the generic configuration model, which is used internally by the configurator. What’s missing yet, is the mapping between both the domain model and its generic representation. In this section, we will describe how the object-oriented concepts are translated into meta model elements.

In short, Table 4.2, “Mapping between the Java Concepts and the Generic Configuration Model” summarizes the mapping:

Table 4.2. Mapping between the Java Concepts and the Generic Configuration Model

JavaBean Domain Model	Types	Annotation	Generic Configuration Model	Description
Class (Java type)	any	@Component	Component	By default, a custom Java class maps to a (configurable) component.
Property (field, getter / setter)	primitives + wrappers, String, Enums, InputStream; plus their array and collection variants	@Attribute	Attribute	All <i>non-complex typed</i> JavaBean properties (attributes in a UML class diagram) are mapped to attributes within the component model.
Property (field, getter / setter)	Object (and any other subclasses) plus their array and collection variants	@Part	Part	All <i>complex typed</i> JavaBean properties (aggregations in a UML class diagram). I.e. all non attribute relevant types, except ignored properties, are considered (configurable) parts.

For the given types, the mapping is established by default without the need of annotating a class’ members. We say it is *implicitly mapped*. If the annotation is present, a property is considered *explicitly mapped*.

The mapping is established within a process called *meta model extraction*, which usually happens once at configurator initialization time (also called *bootstrap*). During that process, the Java domain model is introspected and a corresponding hierarchy of descriptors is recorded. The process also involves the initialization of facet descriptors, which leads to other annotations being processed. We will describe these annotations and their corresponding mappings in extensively in Section 4.4, “Modeling Concepts”.

4.3.3. Facets

The concept of facets is an essential part of the OpenConfigurator modeling approach. As we will see later in Section 5.2.5, “Plugins and Connectors”, facets are a key technique to facilitate the extensibility and flexibility of the overall framework. While the core component model precisely reflects the component structure derived from a JavaBean domain model, facets enrich the model with additional semantics. Technically spoken, a facet implements the *facade*³⁰ design pattern. This pattern abstracts from the underlying model by focussing

³⁰See http://en.wikipedia.org/wiki/Facade_pattern, last accessed May 30th, 2012.

a single aspect. Figure 4.11, “Using Annotation to Control Facets” illustrates the usage of annotations to control different facets of a component.

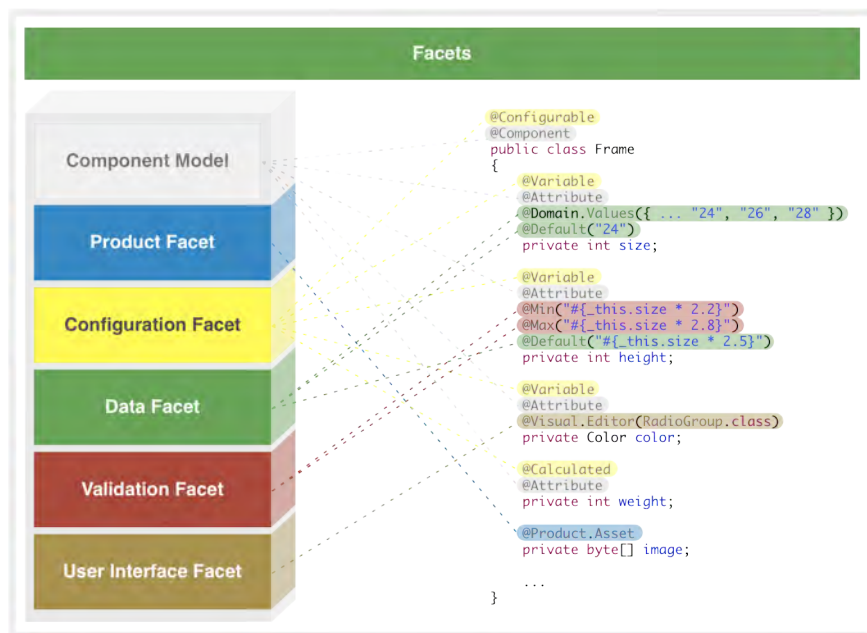


Figure 4.11. Using Annotation to Control Facets

From a high level perspective, with the help of facets, various perspectives onto one and the same product/component can be modeled. In Section 2.3.3.2, “Product Architecture” we described some of these perspectives and identified *multi-structuring* as an essential structuring technique to encompass all relevant model information.

However, by now, OpenConfigurator uses facets for the realization of relatively low level functionalities. Currently it knows the following component facets:

Product facet. Exposes all information of a component, that are part of the *product view*. The product view is an information model, that combines all marketable attributes relevant to the customer. It is used, for instance, to realize the product browser functionality (see Section 3.3.2.1, “Information”).

Configuration facet. Provides a facade for the configuration related functionality of a component and is accessed primarily during the configuration process. For example, the configuration facet contains methods to query all variable attributes of a component, all calculated ones etc. Internally, the it operates on the data facet and the validation facet described below.

Data facet. Allows to control the data model of a component. It is used, for instance, to manage the domain of a component or attribute. Moreover, the data facet abstracts from the backend data storage and thus helps to decouple the configuration logic and the underlying data provider, which may, for example, be a PDM system, JPA accessed relational database or a JCR³¹ based content repository.

Validation facet. Encapsulates model constraints and validation relevant methods. This facet abstracts from the underlying constraint resolution and validation mechanism. For example, constraints compatible with the Bean Validation specification [JSR3032009] can be defined and accessed or a constraint solver, compatible with the Constraint Programming API³² can be integrated (see Section 7.2.1, “CSP/Constraint Solver Integration”).

³¹ Abbrev. Java Content Repository, a high level standard for accessing content data storages, currently specified in version 2.0 by JSR-283, see <http://jcp.org/en/jsr/detail?id=283>, last accessed May 30th, 2012.

³² See <http://jcp.org/en/jsr/detail?id=331>, last accessed May 30th, 2012.

User interface facet³³ Provides an abstraction layer for user interface customizations. Through the UI³⁴ facet, hints on how to render the component within the configuration dialog can be declared as part of the model. For example, the selection style, e.g., "drop down list" vs. "radio group", can be defined. The presentation customizations are defined independently from the concrete view technology (e.g., local SWT³⁵ or JSF³⁶ based view layer) by the OpenConfigurator framework, making configuration models portable across different platforms and presentation frameworks.

Technically important, a facet must not adhere to a particular interface. Hence, 3rd party model information can be integrated transparently. For instance, the JPA meta model (see [JSR3172009, pp. 179]) of a particular element can be made available as a component facet (refer to Section 5.4, "SPI Usage: Extending and Integrating OpenConfigurator" for details).

4.4. Modeling Concepts

Now that we have given an elaborated overview on the main principles and techniques applied in our methodology, it's time to discuss the concrete modeling concepts available.

In general, the modeling concepts introduced here, target to fulfill the requirements discussed in Chapter 3, *Configurators*. Specifically, Section 3.1, "Product Models" provides useful information related to the modeling of product structures, while Section 3.2, "Product Configuration" can be seen as foundation for the realized configuration related functionality.

In this section, we will explain the modeling capabilities of our conceptualization in detail. Particularly, we are going to describe the framework provided meta-data annotations with their exact semantics. Moreover, we'll explain accurately how they influence the configuration process and demonstrate various examples throughout the text.

The section is structured as follows:

Structure Modeling. Section 4.4.1, "Structure Modeling" recapitulates the use of the structural annotations, that allow to **explicitly map JavaBean domain models to the generic configuration model**. Moreover, the concrete model elements, including components, attributes and parts, and their most important concepts are explained.

Having read this section, you'll understand, how the structural aspects of Java domain models are interpreted by the OpenConfigurator framework precisely. Moreover, you can observe, how the generic configuration model, that serves as the basis for the configuration process, is build up by the framework step by step.

Product Modeling. Next, in Section 4.4.2, "Product Modeling", we will discuss annotations used for **modeling the product view** (which corresponds to the *customer view* in Section 2.3.3.2, "Product Architecture"). During the meta model extraction process, the component **product facet** will be populated with the product related annotations (see *Product facet* in Section 4.3.3, "Facets").

Configuration Modeling. Configuration related annotations are subject of Section 4.4.3, "Configuration Modeling". These annotations define:

- which elements (attributes or parts) of product, respectively the Java class, can be customized,
- how these elements are specified during configuration

In effect, the modeling concept introduced in this section allow to define the customizable areas of a product (see Section 2.1.3.1, "Customizable Areas"). With the help of these anno-

³³Note that this feature is currently not implemented.

³⁴Abbrev. User Interface

³⁵Abbrev. Standard Widget Toolkit, see <http://www.eclipse.org/swt/>, last accessed May 30th, 2012.

³⁶Abbrev. Java Server Faces, see <http://www.oracle.com/technetwork/java/javase/javaserverfaces-139869.html>, last accessed May 30th, 2012.

tations, the configurator framework is able to **identify the configuration decisions**, that are necessary to configure an instance of the annotated component class.

After the meta model extraction process, the configuration meta-data modeled with configuration annotations is accessible through the **configuration facet** (see *Configuration facet* in Section 4.3.3, “Facets”).

Data Modeling. Having defined, which elements are customizable and how they are specified, the configuration decisions need to be backed with options, that are presented to the user during the configuration process. The **definition of the domains (options) for configuration decisions** is subject to Section 4.4.4, “Data Modeling”. This section also covers, how **default values** for decisions can be defined.

The data related information is captured by the **data facet** (see *Data facet* in Section 4.3.3, “Facets”).

Constraint Modeling. Finally, Section 4.4.5, “Constraint Modeling” defines, how **domain model restrictions** can be realized our conceptualization. While the full list of available constraints is presented in Appendix A, *Constraints*, this section gives insights about the general constraint definition concepts and provides an overview of supported constraints.

Similar to data related information, the constraint knowledge defined using the constraint annotations discussed here, is accessible through the constraint facet (see *Configuration facet* in Section 4.3.3, “Facets”).

In fact, the sequence of modeling followed in this section, could stand as a general procedure for the development of custom configuration domain models in practice.

Let's begin with the modeling of the structural aspects of a customizable product.

4.4.1. Structure Modeling

Basically, Section 4.3.2, “Model Mapping” has already shown, how product structures are modelled within our methodology. To recap, in order to model product / configuration structures explicitly, OpenConfigurator offers the following annotations:

Table 4.3. Structure Modeling Annotations

Annotation	Description
@Component	Designates a type that is recognized by the framework and treated as a component.
@Attribute	Designates a property as an attribute managed by the framework.
@Part	Designates a property as being a sub-component of a given component.
@Ignore	Designates a property as being ignored by the framework.

With these concepts, a huge variety of products can be described (see Section 3.1, “Product Models”). In the following, we will describe each concept in more detail.

4.4.1.1. Components

Components represent products, assemblies or atomic modules of a product. In the Java domain model they are represented as *classes* (see Section 4.3.2, “Model Mapping”).

By annotating a Java class with @Component, the class is demarcated as a *managed component (type)*. A managed component can be instantiated by the framework's provided API. It is capable of instantiating any Java class, that adheres to the JavaBean convention³⁷. Explicitly

³⁷Note that the OpenConfigurator framework currently doesn't require to implement the `java.io.Serializable` interface as stated in the JavaBean convention, though.

mapping a class with `@Component` allows the framework to detect the component type and extract the given meta-data at application startup.

An example usage of the `@Component` annotation is shown in listing Example 4.4, “Usage of the `@Component` Annotation”.

Example 4.4. Usage of the `@Component` Annotation

```
@Component
public class Product {
    private Assembly assembly;
    ...
}
```

```
@Component
public class Assembly { ... }
```

Runtime Representation

Internally, a managed component *type* is represented by an application scoped instance of a `ComponentDescriptor`, which is instantiated during the meta-data extraction process. Within an active configuration, a managed component instance is represented by a `Component` object. Figure 4.12, “Runtime Representation of Components” illustrates this relationship using an UML object diagram.

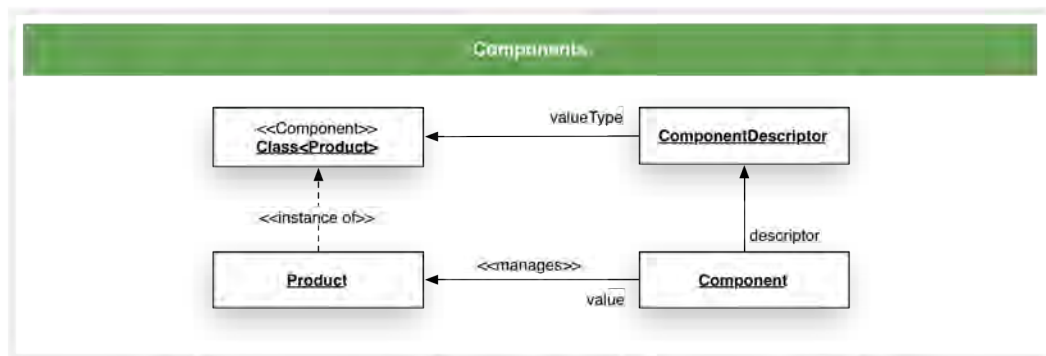


Figure 4.12. Runtime Representation of Components

During configuration, the framework propagates any changes applied on the component to the underlying managed value object (the JavaBean instance), which the component instance represents.

Responsibilities

The component instance has the following responsibilities:

- manage a bean instance of the given value type,
- manage the bean instance's type including dynamic type changes (see below),
- track the managed instance's state, including the following properties:

Table 4.4. Component State Variables

State Variable	Description
abstract	Defines, whether the component's current type is an abstract class.
instantiated	Defines, whether the bean managed by the component has been instantiated already.

State Variable	Description
specified	Defines, whether the user explicitly specified some component property.

- manage referenced attributes and parts; provide convenience methods for accessing them,
- support structural navigation: provide access to the parent component and realize the visitor pattern³⁸
- realize event handling: provide methods to allow the registration of event handlers for the following events:

Table 4.5. Component Events

Event	Description
Change	Triggered upon any component changes (e.g., value changes, type changes, ...).
ValueInstantiation	Fired upon the first instantiation of the managed type.
ValueChange	Triggered when the managed bean changes (e.g., attribute/part changes).
TypeChange	Raised when the underlying bean's type changes (which requires re-creation of the object).
Reset	Fired when the component is reset.

- provide meta-data and facet access: enable accessing the `ComponentDescriptor` corresponding to the component's current type and any registered facets of the component.

Due to the indirection layer introduced by the component model, `OpenConfigurator` not only allows to set properties on not yet instantiated beans (in case the managed type is abstract), but even allows to change the managed type dynamically at runtime. Thereby, the framework must re-instantiate the bean, losing all of its internal, invisible state. Nevertheless, the framework captures and re-instantiates all managed attributes and parts (see below) automatically³⁹. In Section 4.5.2, “Example Configuration Procedure”, we will give an example of a type change.

4.4.1.2. Attributes

The characteristics of a product or assembly are represented by *attributes*. In the Java domain model, an attribute corresponds to a JavaBean property, that is, a field and a pair of getter/setter methods.

Annotating a field (so called *field level access*) or a getter method (*method level access*) with the `@Attribute` annotation, explicitly marks the property as *managed attribute*⁴⁰. By default, all properties of a managed type, with one of the following types (including subtypes of these) are considered attributes:

- `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char` and their object wrappers, `String`, `BigDecimal`

³⁸See http://en.wikipedia.org/wiki/Visitor_pattern, last accessed June 5th, 2012.

³⁹In a future release of `OpenConfigurator`, we might specify callback methods to better support state transitioning during type changes.

⁴⁰Note that both field level and method level access strategies are supported, for a particular property only one of both strategies may currently be used. That means, for an individual property, all `OpenConfigurator` recognized annotations must be placed either on the field, getter or setter method but may not be spread across those three members.

- the array and collection variants of all the above, e.g., `int[]`, `byte[]` or `Collection<String>`
- `InputStream`, `Enum`

An attribute may have a singular value (*singular attribute*) or may have multiple values (*plural attribute*).

The following code fragment, Example 4.5, “Usage of the `@Attribute` Annotation”, shows various attribute definitions.

Example 4.5. Usage of the `@Attribute` Annotation

```
public class Assembly {
    // implicit, singular String attribute, field access
    private String stringAttribute;

    // explicit, singular Integer attribute, field access
    @Attribute
    private Integer intAttribute;

    // explicit, plural String attribute, method access
    @Attribute
    public Collection<String> getMultiValueAttribute() { ... }
    ...
}
```

Runtime Representation

At runtime, the *definition* of a managed attribute is represented as an instance of `AttributeDescriptor`, i.e. either `SingularAttributeDescriptor` or `PluralAttributeDescriptor`, depending on the member's multiplicity. The attribute itself is managed by an instance of `Attribute`, i.e. `SingularAttribute` or `PluralAttribute`. Figure 4.13, “Runtime Representation of Attributes” depicts this relationship.

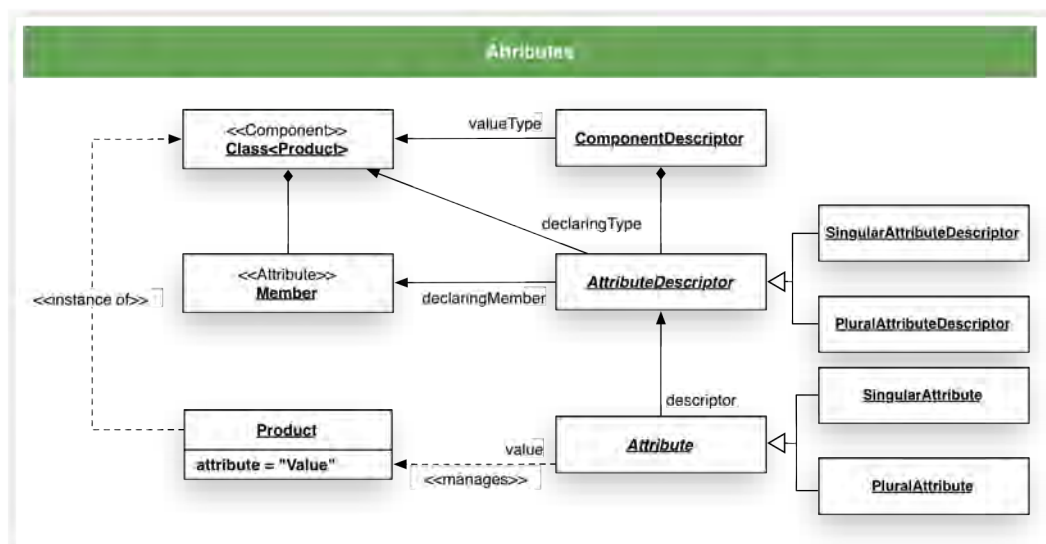


Figure 4.13. Runtime Representation of Attributes

Responsibilities

The `Attribute` instance is mainly responsible for propagating any value changes to the underlying bean property. The full responsibilities of the `Attribute` interface can be summarized as follows:

- manage the value of a bean instance member with the given type

- track the member's state, including the following properties:

Table 4.6. Attribute State Variables

State Variable	Description
instantiated	Defines, whether the managed member's value has been instantiated already.
specified	Defines, whether the user explicitly specified the attribute value.

- support structural navigation: provide access to the parent component
- realize event handling: provide methods to register event handlers for the following events⁴¹:

Table 4.7. Attribute Events

Event	Description
Change	Triggered upon any value changes.

- provide meta-data access⁴²: enable accessing to the `AttributeDescriptor` corresponding to the member's type.

4.4.1.3. Parts

Complex sub-assemblies of a product or assembly are modeled using *parts*. Like attributes, a part corresponds to a JavaBean property, that is, a field and a corresponding getter/setter method pair.

By default, all Object valued bean properties (except those mapped as attributes, see Section 4.4.1.2, “Attributes”) are considered parts. They can be explicitly mapped using the `@Part` annotation. Similar to attributes, parts can have a single value (*singular part*) or can have multiple values (*plural part*). Example part definitions are shown in Example 4.6, “Usage of the `@Part` Annotation”.

Example 4.6. Usage of the `@Part` Annotation

```
public class Assembly {
    // implicit, singular part, field access
    private SubAssembly part;

    // explicit, singular part, field access
    @Part
    private SubAssembly anotherPart;

    // explicit, plural part, method access
    @Part
    public Collection<SubAssembly> getMultiComponentPart() { ... }
    ...
}

public class SubAssembly { ... }
```

Runtime Representation

Again, the *definition* of a managed part is represented by an instance of a `PartDescriptor`, i.e. either `SingularPartDescriptor` or `PluralPartDescriptor`. The part itself, in turn, is

⁴¹Additional events may be specified in the future.

⁴²Note that by now facets only exist on component level. Expanding this concept on attribute level might be a reasonable framework enhancement in the future.

an instance of `Part`, i.e. `SingularPart` or `PluralPart`. In contrast to attributes, which directly manage the JavaBean member's value, a part references the JavaBean property's value in terms of a component. On the one hand, this precisely reflects the natural relationship between two components, on the other hand this has the benefit, that nested components can be treated like regular components.

Figure 4.14, “Runtime Representation of Parts” shows the UML object diagram of a part relationship. Note that in UML diagrams, part relationships are usually modelled as aggregations, respectively compositions.

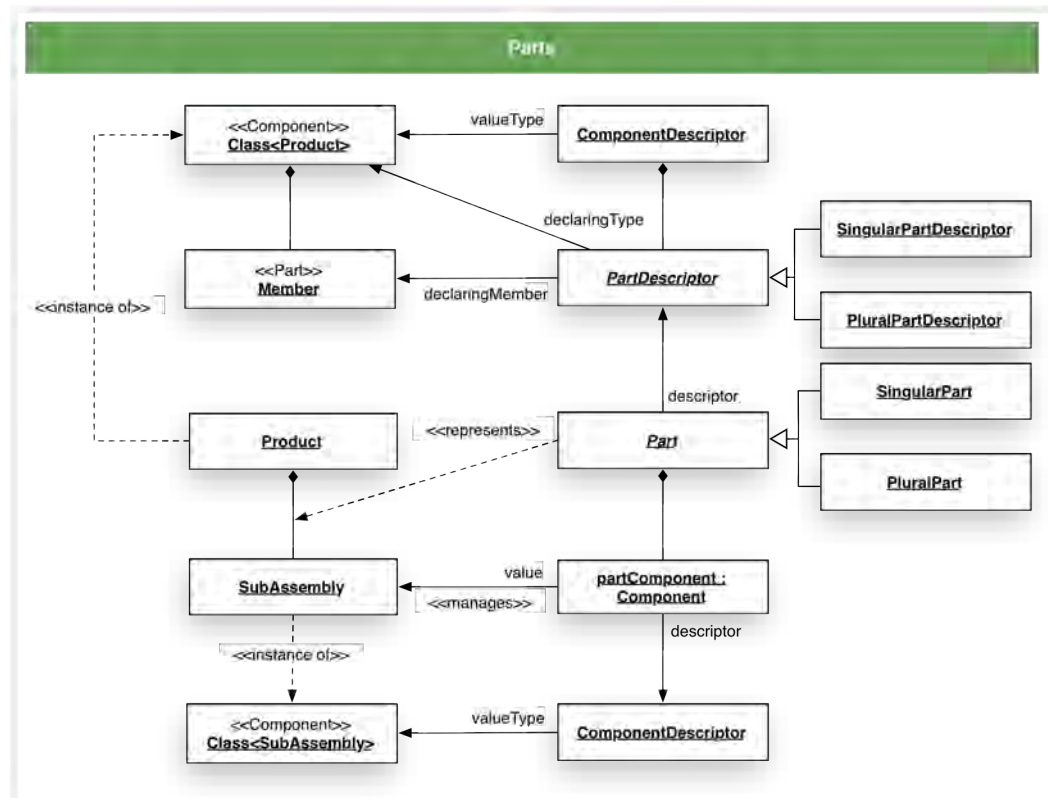


Figure 4.14. Runtime Representation of Parts

Responsibilities

The intermediate `Part` instance is used to control the aggregation relationship between two or more components. It can be seen as a placeholder for the aggregated component(s). The responsibilities of the `Part` interface can be summarized as follows:

- manage access to the nested component instance
- delegate dynamic type changes (see Section 4.4.1.1, “Components”) to the nested component instance
- track the nested component's state, including the following properties:

Table 4.8. Part State Variables

State Variable	Description
instantiated	Defines, whether the component, managed by the part, has been instantiated already.

State Variable	Description
specified	Defines, whether the user explicitly specified some property of the part's nested component.

- support structural navigation: allow accessing the parent component and nested component(s)
- realize event handling: provide methods to register event handlers for the following events:

Table 4.9. Part Events

Event	Description
Change	Triggered upon any changes of the nested component (e.g., value changes, type changes, ...).
ValueChange	Triggered when the nested managed bean changes (e.g., attribute / part changes).

- provide meta-data access⁴³: enable access to the `PartDescriptor` corresponding to the part's current type

Having described the structural modeling concepts in this section, in the next section we will discover, how particular attributes can be accompanied with additional, product related semantics.

4.4.2. Product Modeling

Within our conceptualization, we specify additional annotations to support product modeling more precisely. The following annotations are available:

Table 4.10. Product Modeling Concepts

Annotation	Description
<code>@Product</code>	Designates a type to represent a (physical or non-physical) product.
<code>@Product.Name</code>	Identifies an attribute as the product's name.
<code>@Product.Description</code>	Marks an attribute as product description.
<code>@Product.Attribute</code>	Designates an attribute as product specification attribute. Specification attributes are considered as the "visible", most relevant attributes from a customer perspective. They appear, for instance, in a product catalog. The <code>@Product.Attribute</code> annotation provides the following elements: • <code>label</code> (optional): <code>String</code> providing the speaking name of an attribute • <code>level</code> (optional): <code>int</code> array describing the level of detail of the attribute. We distinguish: • <code>UNDEFINED</code> (ordinal 0)

⁴³Note that the concept of facets is also not available on part level yet. This may change in a future release of OpenConfigurator.

Annotation	Description
	• IDENTIFICATION (ordinal 1) • STANDARD (ordinal 2) • DETAIL (ordinal 4) • SPECIAL (ordinal 8) The handling of these levels is client specific.
<code>@Product.Asset^a</code>	Marks an attribute as provider of a product asset. The attribute must be of type <code>byte[]</code> or <code>java.io.InputStream</code>. The <code>@Product.Asset</code> annotation further provides the following attributes: • name (optional): gives the asset a client-specific name. <code>OpenConfigurator</code> specifies the following built-in names^b: • PRIMARY • PRIMARY_THUMBNAIL • SECONDARY • SECONDARY_THUMBNAIL The name is not required to be unique, that is, a class can provide multiple, e.g., secondary images. The consideration of the name is client-specific and non-normative.
<code>@Product.Price</code>	Identifies an attribute as provider for the product's price. The attribute may have any numeric type, however, <code>BigDecimal</code> is recommended for precise price calculation purposes.^c

^aProviding product visualizations using this annotation is considered a very basic form. We will enhance the visualization capabilities of products in a future version. E.g., we'll allow to specify the encoding format for the returned data stream etc.

^bThe values' prefixes `info.openconfigurator.products.annotation.Product.Asset.` have been omitted for brevity. The API provides equally named static constants for these names.

^cNote that beyond the `@Product.Price` annotation, `OpenConfigurator` doesn't specify any additional price calculation strategy yet. Consequently, within the price attribute's getter method, the developer must implement the full pricing logic manually. We will provide support for more sophisticated strategies in a future version.

Example 4.7, "Usage of the `@Product` Annotation" shows a `JavaBean` definition using `@Product` annotations:

Example 4.7. Usage of the @Product Annotation

```

@Product
public abstract class Bike
{
    @Product.Name
    public String getName() { ... }

    @Product.Description
    public String getDescription() { ... }

    @Product.Price
    public BigDecimal getPrice()
    {
        return new BigDecimal(0, new MathContext(2))
            .add(basePrice)
            .add(frame.getPrice())
            .add(fork.getPrice())
            .add(wheels.getPrice())
            .add(getEquipmentsPrice());
    }

    @Product.Asset(name = PRIMARY)
    public byte[] getImage() { ... }

    @Product.Attribute(label = "Total weight", level = STANDARD)
    public int getWeight() { ... }

    @Product.Attribute(label = "Robustness factor", level = SPECIAL)
    public double getRobustness() { ... }
    ...
}

```

Runtime Representation

The data designated by the annotations introduced above can be accessed at runtime using the *product facet*. This facet provides a product-centric view on a component (see "Product facet" in Section 4.3.3, "Facets"). For a detailed description of the Product facet interface, refer to Appendix C, *OpenConfigurator API/SPI*.

The concepts presented in this sections merely describe static aspects of a product. With them, customizable areas cannot be defined yet. This is subject to the configuration related annotations presented in the next section.

4.4.3. Configuration Modeling

For interactive configuration of customizable products, the framework needs to know which components of the structure are configurable and how specification for these shall proceed. For that purpose, our conceptualization offers a number of annotations to be applied on a product type:

Table 4.11. Configuration Modeling Concepts

Annotation	Description
@Selectable	Component level annotation, indicating that the given component is to be selected from an existing set of available components.
@Configurable	Component level annotation, indicating that the given component is selected

Annotation	Description
	but additionally contains parameterized attributes.
@Constructible	Component level annotation, indicating that custom variants can be created freely.
@Variable	Designates an attribute to be configurable during the configuration process.
@Parameter	Designates an attribute being a freely customizable parameter of the underlying component.
@Calculated	Signaling, that an attribute represents a calculated value.
@Configured	Designates a configurable part of the component.

The first three annotations, @Selectable, @Configurable and @Constructible, are used to define the *specification method*. These methods exactly correspond to the three different strategies applicable for the specification of configuration decisions identified in Section 3.2.1, “Characteristics of Product Configuration”. We will discuss these specification methods in more detail in a moment.

The annotations @Variable, @Parameter and @Calculated apply to attributes and characterize their role during the configuration process. Basically, they are used to model the different attribute types explained in Section 3.1.2, “Attributes: Component Characteristics”.

The last annotation, @Configured, is used to explicitly designate a part being configurable in the context of its parent.

4.4.3.1. Specification Methods

Per component type, the domain model author must define, how an instance of the given type is being specified during the configuration process. He may choose from one of the three available methods @Selectable, @Configurable and @Constructible for a particular component type. By default, if no annotation is present, a component is considered @Configurable.

In order to better understand the different specification methods, it's useful to describe different concrete use cases. Hence, we will introduce each specification method with a concrete example.

Selection

Take for example a car manufacturer offering configurable automobiles, a typical assemble-to-order company (AtO, see Section 2.3.3.4, “Order Fulfillment Strategies”): when configuring a car, the manufacturer won't offer motors with arbitrary performance parameters, for both technical and marketing reasons. Instead, as an ATO company, he will offer specific, pre-defined variants of motors only, of which he holds numerous exemplars in stock. During configuration, the customer *selects* a single component variant and takes it “as is”.

In terms of OpenConfigurator, this would be modeled as follows (see Example 4.8, “Modeling Selectable Components”):

Example 4.8. Modeling Selectable Components

```

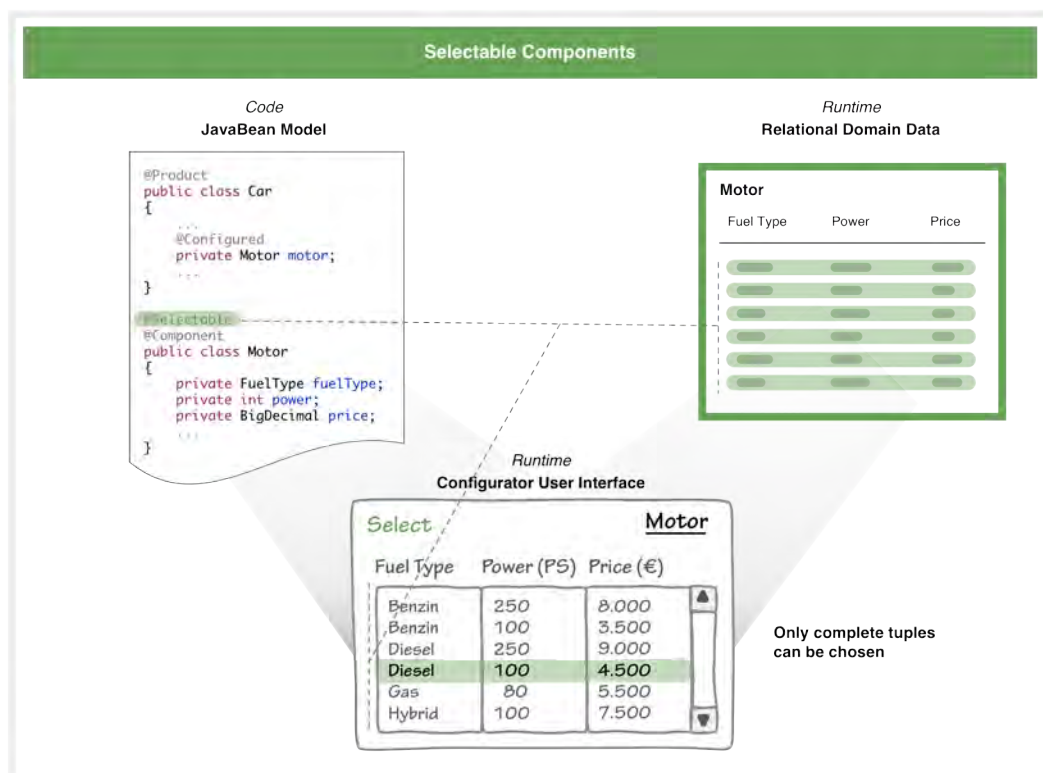
@Product
public class Car
{
    ...
    @Configured
    private Motor motor;
    ...
}

@Selectable
@Component
public class Motor
{
    private FuelType fuelType;
    private int power;
    private BigDecimal price;
    ...
}

```

The `@Selectable` annotation is placed directly on the target component, `Motor` in this case. This makes sense, since the car manufacturer solely produces pre-manufactured motor variants and wants to enforce selection of a particular variant, whenever a motor is referenced within a configurable product.

Technically spoken, the user selects a **tuple of attributes** instead of choosing arbitrary combinations of values for individual attributes. Figure 4.15, “Component Selection Schema” illustrates this:

**Figure 4.15. Component Selection Schema**

As whole tuples and not particular attribute values are selected, the domain must be defined on component level (*component level domain*) using the `@Domain.Query` annotation (see Section 4.4.4.1, “Domains” for details).

Provided that the underlying data store (e.g., a relational database) identifies the tuple by a unique identifier (ID), for the configurator it is sufficient to store the exact type (Motor in this case) and the tuple ID with the configuration.

Note that although `@Selectable` is used, signaling tuple selection mode, the visual interface for choosing that tuple may be displayed "incrementally", as Figure 4.16, "Incremental Component Selection" indicates:

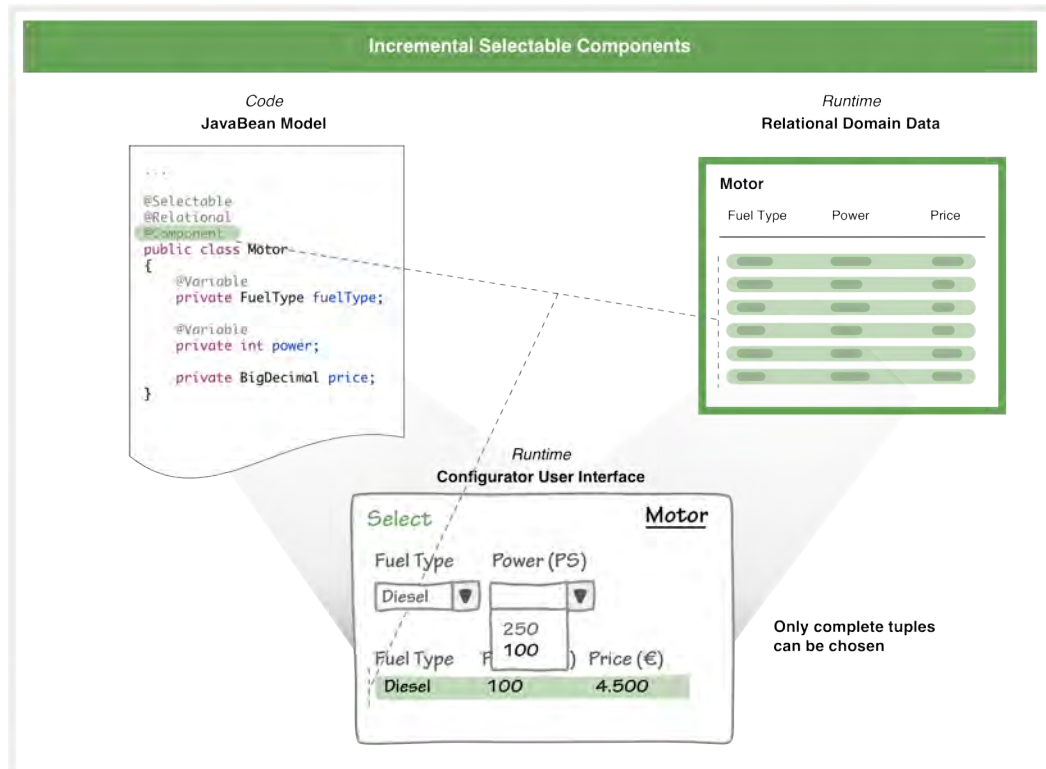


Figure 4.16. Incremental Component Selection

In this case, the `Motor` component would additionally be constrained using the `@Relational` annotation (see Section 4.4.5, "Constraint Modeling") and the variables, that the user can influence directly, are annotated `@Variable` (see Section 4.4.3.2, "Configuration Attributes"). This ensures, that the attribute domains are reduced in relation to each other: e.g., when the user specifies `Diesel` as `FuelType`, the `Power` attribute's domain is shrunk to the values 250 and 100 automatically, as only tuples with these values remain. This is done regardless of the order of specification. That means, when the user specified the `Power` attribute to be 80 first, the configurator would have automatically fixed the `FuelType` attribute to the value `Gas`, as this way selected gas motor is the only one available with a `Power` of 80. The reduction of domain values in this manner, is an example for *constraint propagation* (see "Constraint based configuration" in Section 3.3.3.5, "Implementation Aspects").

A selectable component must not contain parameterized attributes, i.e. attributes annotated `@Parameter`. If the selection of a base product and subsequent parameterization of the chosen component is required, *Configuration*, as discussed in the next section, must be used as specification method.

Configuration

OpenConfigurator offers the ability to mark parameterized components as being `@Configurable`. As stated in Section 3.2.1, "Characteristics of Product Configuration", *Configuration*

can be seen as a specification method between *Selection* and *Construction*. In fact, the specification of a component often involves an initial selection of a base component variant along with a subsequent parameterization activity (which could be considered as a construction task as well, see below).

An example use case for a configurable, parameterized component is a window: a window manufacturer may offer several base window types that determine the general forming, used materials etc. Concrete windows, however, are built customer specific, that is, the company doesn't pre-define the dimensions of the product upfront. Instead, width and height are defined as parameters in the product model and the decision about the concrete dimensions is left to the customer. In reality, though, the company would certainly restrict the maximum dimensions to a certain size using @Max constraints (see Section 4.4.5, "Constraint Modeling").

In terms of OpenConfigurator, a Java domain model describing such a window product could look like the one shown in Example 4.9, "Modeling Configured, Parameterized Components":

Example 4.9. Modeling Configured, Parameterized Components

```
@Configurable
@Product
public class Window
{
    @Variable
    private Form form;

    @Variable
    private Material material;

    @Parameter
    private int width, height;
    ...
}
```

Since the Window type contains the @Parameter annotated attributes width and height, it must be annotated @Configurable (or alternatively @Constructible, see below). During configuration, the user first selects a base product variant of type Window, as he does in the case of *Selection* described above, too. Technically, this again corresponds to the selection of an existing *tuple of attributes* (here a 2-tupel containing a specific form and material value). Then, he specifies the additional attributes declared as parameters (here width and height) to complete the component specification task.

Figure 4.17, "Component Configuration Schema" illustrates this concept graphically:

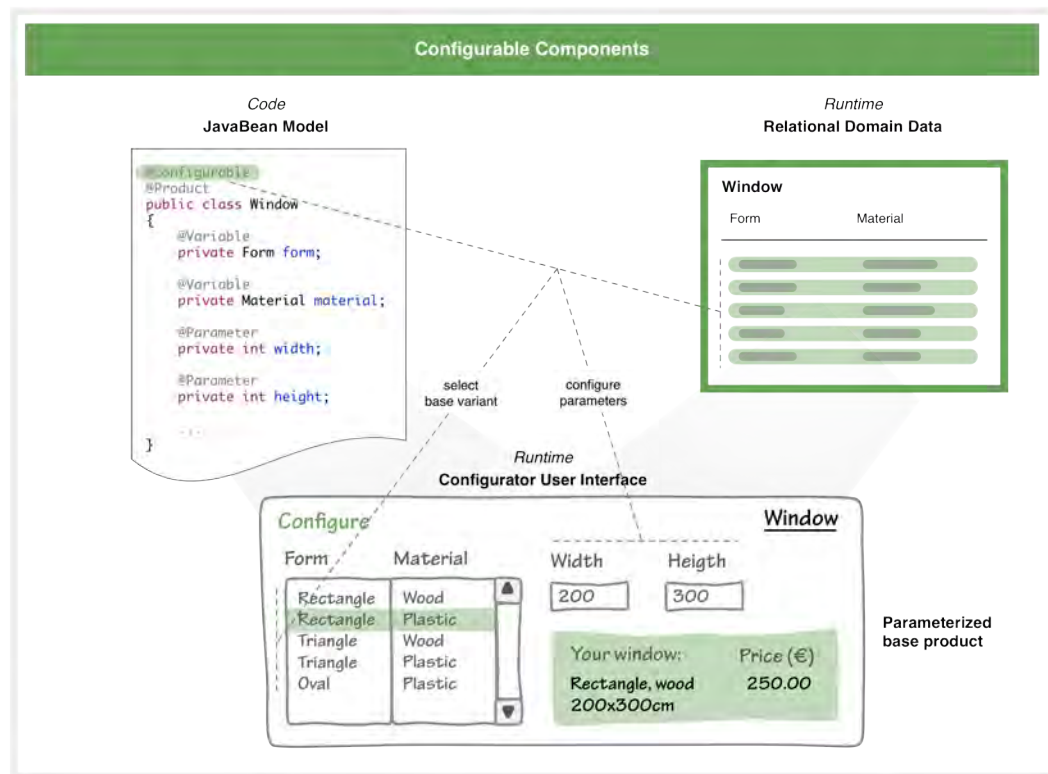


Figure 4.17. Component Configuration Schema

For parameterized components, often both types of domains, component level and attribute level domains (see Section 4.4.4.1, "Domains"), are used: the component level domain definition demarcates the selectable tuple of the component, while attribute domain specifications define the value space for the parameters.

The configurator must store the selected base product tuple's ID along with all specified parameters with the configuration. The `@Relational` annotation may be used in connection with the `@Configurable` annotation, but the relation must not involve any `@Parameter` annotated attribute, though (see Section 4.4.5, "Constraint Modeling" for details).

Construction

Another type of products are entirely customized goods. Particularly, companies pursuing built-to-order strategies (BtO, see Section 2.3.3.4, "Order Fulfillment Strategies"), often offer products, that include freely definable components. For these components, no base component exists and virtually any combination of attribute values is manufacturable.

For example, this is the case for kitchens: while the individual units of a kitchen themselves belong to a particular product line and are configurable components in itself, the overall kitchen can be modeled as a constructable product, that doesn't exist as is.

Using OpenConfigurator, a kitchen domain model could be defined as follows, see Example 4.10, "Modeling Constructed Components".

Example 4.10. Modeling Constructed Components

```

@Constructible
@Product
public class Kitchen
{
    @Parameter
    private int length, width, height;

    @Configured
    private Countertop countertop;

    @Configured
    private List<Unit> floorUnits, wallUnits;
    ...
}

```

As a *Construction* component, the kitchen class solely owns parameter attributes and configured components, which itself may be selectable, configurable or constructible, in the same way.

Basically, an @Constructible component may also cover @Variable attributes, but no regular ones. During specification, the user must specify all parameters and choose values for all non-optional variable attributes. As there is no upstream selection activity, regular attributes wouldn't ever get assigned, which is why they're forbidden entirely.

In the *Construction* scenario, the user does not select any attribute tuples, but instead specifies each attribute individually. Thus, domains may solely be defined on attribute level (*attribute level domains*).

Figure 4.18, “Component Construction Schema” explains the constructive specification graphically:

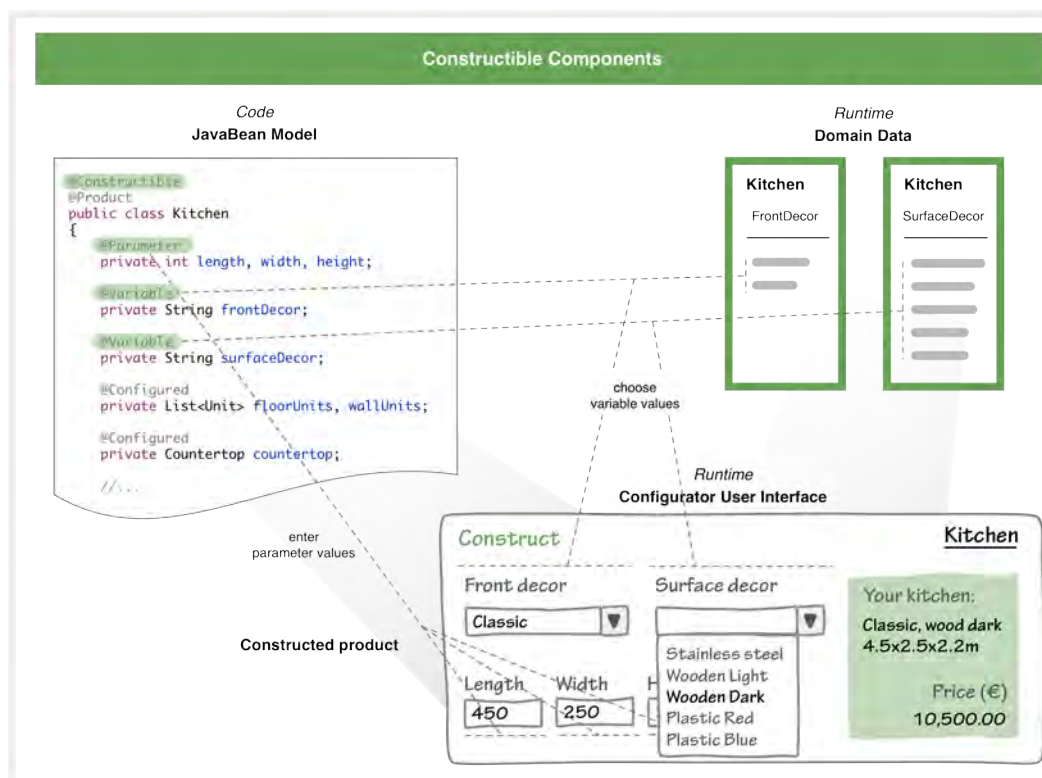


Figure 4.18. Component Construction Schema

For constructed components, the configurator stores each particular attribute value instead of a tuple ID.

Figure 4.19, “Comparison of Specification Methods” shows a comparison matrix of the three specification methods *Selection*, *Configuration* and *Construction*, that summarizes their differences. The features mentioned in the table will be explained more detailed in subsequent sections.

	Selectable	Configurable	Constructible
	Tuple selection	Tuple selection + Attribute specification	Attribute Specification
Attribute Behavior Supported			
Regular attribute	●	●	○
Variable	●	●	●
Parameter	○	●	●
Default Attribute Behavior			
Regular attribute	●	○	○
Variable	○	●	○
Parameter	○	○	●
Domain Type Supported			
Component level (tuple)	●	●	○
Attribute level (value)	○	●	●
Constraint Support			
Relational	●	●	○
Persisted State			
Tuple ID	●	●	○
Individual values	○	●	●
● Supported ● Limited support ○ Not supported			

Figure 4.19. Comparison of Specification Methods

While all three specification methods have some intersections regarding their behavior, we separated them conceptually for an improved semantical description of the product model. Also, the specification methods differ in their default behavior related to the treatment of attributes in case these are not annotated. By this means, it's often obsolete to annotate attributes entirely, which greatly simplifies modeling and supports code readability simultaneously.

4.4.3.2. Configuration Attributes

Attributes are used to precisely characterize component variants.

In the previous section's examples, we already used attribute annotations and mentioned their purposes roughly. In this section, we want to discuss the available configuration attribute annotations, namely `@Variable`, `@Parameter` and `@Calculated`, in more detail.

Figure 4.20, “Comparison of Attribute Types” summarizes the four different types of attributes graphically. We will come back to this comparison matrix as we discuss the different concepts.

	Regular	Variable	Parameter	Calculated
	Invariable, indirectly specifiable or internal attributes	Directly specifiable attributes, bound domain	Directly specifiable attributes, unbound domain	Invariable attributes, automatically calculated
Specification Method Supported				
Selection	●	●	○	●
Configuration	●	●	●	●
Construction	○	●	●	●
Domain Type Supported				
Component level (tuple)	●	●	○	○
Attribute level (value)	○	●	●	○
Constraints				
Restrictable	●	●	●	○
Expression targets	●	●	●	○
Other Supported Capabilities				
Automatic completable	●	●	○	●
Solvable	●	●	○	○
● Supported ● Limited support ○ Not supported				

Figure 4.20. Comparison of Attribute Types

Defaults and Regular Attributes

When a particular attribute is not annotated with one of the given configuration annotations at all, one of the following two options applies:

1. **Default behavior application.** In case also *none of the other attributes* declared within the same class is annotated, the default behavior as defined in Figure 4.19, “Comparison of Specification Methods” applies. That is, by default, attributes of `@Selectable` components are treated as regular attributes, the ones of `@Configurable` components are considered `@Variable` attributes and those of `@Constructible` components are treated as `@Parameters`.
2. **Regular attribute.** If *at least one of the other attributes* is annotated with a configuration attribute annotation, a non-annotated attribute is considered as *regular attribute*.

A *regular attribute* can be used to either model internal aspects of a product (e.g., an internal value used for price calculations) or can be used to represent invariable component characteristics. Anyway, regular attributes cannot be specified directly by the user. However, they can only be changed indirectly, when the user selects a different tuple.

Consider, for instance, the price field of the `Motor` component used in Figure 4.16, “Incremental Component Selection”, which is a regular attribute. Depending on the selected tuple,

the price is automatically adjusted. In this case, the user cannot specify the tuple by choosing a particular price.

Regular attributes can itself be constrained and referenced as targets in constraint expressions (see Section 4.4.5, “Constraint Modeling”).

Variable Attributes

Attributes, that may be manipulated by the user during configuration, must be annotated `@Variable` (the default behavior for configurable components). Importantly, variable attributes must have a *bounded domain*, that is, their domain values must be enumerable (see Section 4.4.4.1, “Domains” for details). Otherwise, the framework raises an exception.

During the configuration process, the configurator client presents some kind of editable component for manipulating the variable attribute's value. For attributes with locally defined domains, this editable component may be one of the standard user interface components, such as an input field, a drop-down list, a checkbox, radioboxes etc. However, it may as well be a tabular list for domains defined on component level.

Like regular attributes, variables can be constrained and referenced as targets in constraint expressions. Furthermore, due to their bounded domain, the configurator can resolve configuration problems featuring variable attributes and can automatically complete their values.

An example for a variable attribute is the `material` attribute from Example 4.9, “Modeling Configured, Parameterized Components”.

Parameter Attributes

Not all attributes have bounded domains. For example, a simple `String`, `long`, `double`, `float` or `int` typed attribute without any additional constraint is considered unbounded⁴⁴. To support unbounded domains, `OpenConfigurator` allows attributes to be annotated `@Parameter`.

Parameters are treated in a special way within the framework at various points: for example, they cannot be considered during the solving process of constraint satisfaction problem (CSP)⁴⁵ and they cannot be automatically assigned by the framework. Furthermore, their domain cannot be defined on component level, since providing domains in terms of concrete tuples inherently result in enumerable, bounded domains. Otherwise, they share the same characteristics than variable attributes.

Typical examples for parameters are the dimension properties `length`, `width` and `height` shown in Example 4.10, “Modeling Constructed Components”.

Calculated Attributes

Product models often require attributes, that perform some kind of calculation. Meaningful examples include price fields of configurable components, which, for instance, calculate prices depending on the product's dimensions.

Example 4.11, “Usage of the `@Calculated` Annotation” shows an example of a calculated price attribute:

⁴⁴Note that, strictly speaking, these types *are* bounded: there's a maximum string length, a maximum `long` (`Long.MAX_VALUE`) and `int` value (`Integer.MAX_VALUE`) and also a maximum `double` precision. However, enumerating their values isn't feasible with sufficient performance, which is why they're semantically considered to be unbounded.

⁴⁵The feature of automatically generating solutions by transforming the configuration into a constraint satisfaction problem (CSP) and solving it is not further discussed but may be subject to future research work, see also Section 7.2.1, “CSP/Constraint Solver Integration”.

Example 4.11. Usage of the @Calculated Annotation

```
@Configurable
@Product
public class Window
{
    @Parameter
    private int width, height;
    private BigDecimal pricePerSquareMeter = new BigDecimal(125.5);

    @Calculated
    @Price
    public BigDecimal getPrice()
    {
        final int sqm = width / 100 * height / 100;
        return new BigDecimal(1, new MathContext(2))
            .multiply(new BigDecimal(sqm))
            .multiply(pricePerSquareMeter);
    }
    ...
}
```

Technically, OpenConfigurator must not cache the computed values within the generic model instance, which is why domain model authors are encouraged to annotate such attributes `@Calculated`. This signals the framework that the value may dynamically change at any point in time, that is, whenever a property accessed during the computation changes.

Moreover, an important characteristic of calculated attributes is, that their results cannot be accessed before the underlying managed bean instance has been instantiated. An exception to this rule are calculated attributes defined using Java's `static` modifier. These can be accessed without an instance of the same class.

Naturally, for calculated attributes the domain is not specified explicitly, due to its unbounded size. Furthermore, since OpenConfigurator doesn't know which properties influence the computation, calculated attributes must not be constrained or used as targets in constraint expressions. If a constraint referencing a calculated property would be violated, the framework could not provide any hint on how to resolve the violation⁴⁶.

4.4.3.3. Configurable Parts

Finally, OpenConfigurator provides the `@Configured` annotation to mark parts as being relevant for configuration explicitly. For parts, the same default behavior applies as it does for attributes: if at least a single property is annotated `@Configured`, only annotated properties are considered. Otherwise, if none of the part properties is annotated, they're all considered `@Configured` by default.

Example 4.12, "Use of the `@Configured` Annotation" shows one configured part motor and one regular, invariable part gearbox:

⁴⁶In a future version of OpenConfigurator we might employ another annotation that provides exactly this information. For instance, a code fragment like `@PropertyDependency({ "width", "height", "pricePerSquareMeter" })` could be used to specify that the calculation depends on the properties `width`, `height` and `pricePerSquareMeter`. This way, the framework could provide messages like "Change width and/or height of X in order to resolve the constraint violation".

Example 4.12. Use of the @Configured Annotation

```

@Product
public class Car
{
    ...
    @Configured
    @Part
    private Motor motor;

    @Part
    private Gearbox gearbox;
    ...
}

```

Any component type, that is referenced as @Configured part of another component, must define one of the specification methods @Selectable, @Configurable or @Constructible as described in Section 4.4.3.1, “Specification Methods”.

We've now seen, how customizable components are modeled structurally and how customizable areas (realized as attributes or parts) can be defined. From these definitions, OpenConfigurator is able to extract the list of **configuration decisions**, required to configure an instance of the given type. We'll cover an example in detail in Section 4.5, “Configuration Procedure”.

In the next section, we will show, how the OpenConfigurator supports the definition of domains. That is, how the options for the particular configuration decisions can be defined.

4.4.4. Data Modeling

By now, we solely defined the customizable product's structure along with its adaptable areas. In principle, a model defined this way can already be used as the domain model for a working configurator. The user could choose arbitrary values for the different attributes and could configure any part at will (assuming the part uses the specification method other than @Selectable).

However, this behavior is rarely demanded in real-world use cases: in Section 2.2.6, “Mass Customization”, we identified the **limited, stable solution space** as a key factor for the successful implementation of mass customization. That means, companies in practice wouldn't allow the customer to configure arbitrary product variants. Instead, they'd restrict the solution space by specifying the values, a particular performance attribute can take. Or they would want to define the concrete variants of a component, the user can choose from, when selecting a part. Hence, the configurator developer must be able to **define or restrict the domains for a given configuration decision**.

In the next section, we'll show how you can define domains for the decisions determined from the configuration domain model. The then following Section 4.4.4.2, “Defaults” will demonstrate, how you can designate a specific domain value as the default value for a particular element. Finally, Section 4.4.5, “Constraint Modeling” introduces constraints, used to restrict the domains' value spaces.

4.4.4.1. Domains

Again, so far we showed, how Java language constructs enriched with a custom set of annotations can be used to model *product structures* and *configuration behavior*. However, an essential aspect of product modeling was still missing: the modeling of **product variation**.

For instance, in Example 4.8, “Modeling Selectable Components” we described a Car product containing a Motor component characterized by the attributes fuelType, power and price.

We did not specify **valid values** and **value combinations** for neither the Motor component itself, nor its three attributes. As explained in Section 2.1.2, “Variants” though, it are particularly the *attribute values*, that make up the different variants of one and the same product.

To recapitulate, here's the source code fragment from the Car component model again:

```
@Product
public class Car
{
    @Configured
    private Motor motor;
    ...
}

@Selectable
@Component
public class Motor
{
    private FuelType fuelType;
    private int power;
    private BigDecimal price;
    ...
}
```

Now, when configuring an instance of the Car class, how can the configurator know, what Motors are available? What options / values for the Motor attributes should it offer to the user to choose from?

The answer is short: the developer must back the model with *data*. In particular, he must design the components' and attributes' **domains**.

Domains, however, have manifold characteristics and can be defined in numerous ways (see also item *Domains* in Section 3.1.2, “Attributes: Component Characteristics”). Figure 4.21, “Morphological Box for Domains” gives an overview of possibilities relevant to the Open-Configurator framework.

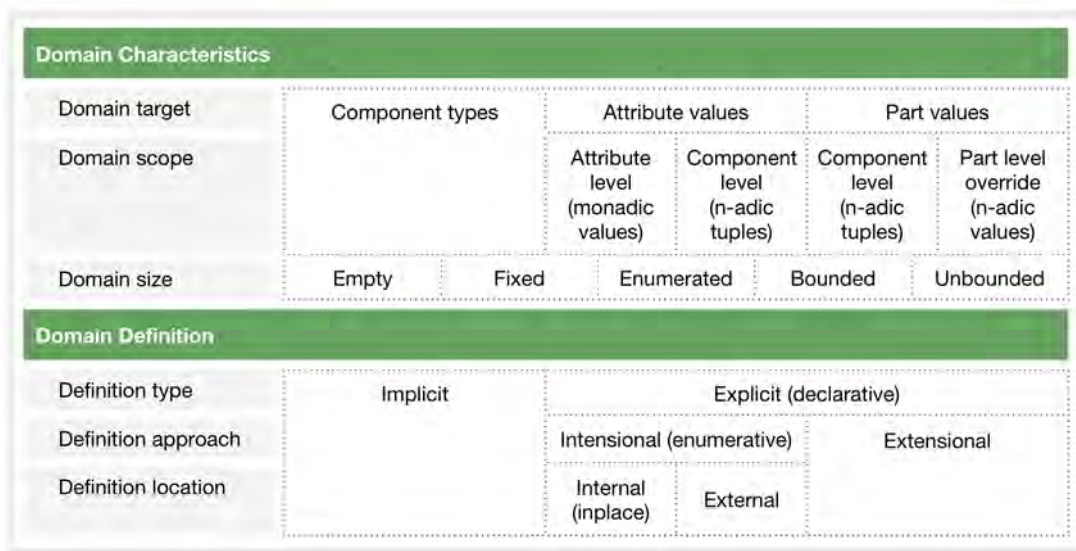


Figure 4.21. Morphological Box for Domains

Domain Characteristics

First of all, we need to distinguish certain **domain characteristics**. For configuring object-oriented models, we have to differentiate various *domain targets*. Basically, all decisions defined

in Section 3.1, “Basic Meta Model for Generic Product Modeling”, must be backed with domains. However, some of them cannot be modeled explicitly.

Component type domains. During configuration, the user sometimes must choose a particular component type, as defined in Section 3.1.1, “Components: Structural Decomposition”. To back this decision with data, *component type domains*, containing type identifiers, are required.

For instance, in Section 4.5.2, “Example Configuration Procedure” we will see, that a **Bike** can be configured with one or multiple **Equipment** components. When adding an **Equipment**, the user must choose between a **Lock** or a **Basket** item. The **specify type task** thus must be backed with a component type domain, containing the values `Lock.class` and `Basket.class`.

Figure 4.22, “Runtime Representation of a Component Type Domain” illustrates this, including the runtime representation of a corresponding domain object:

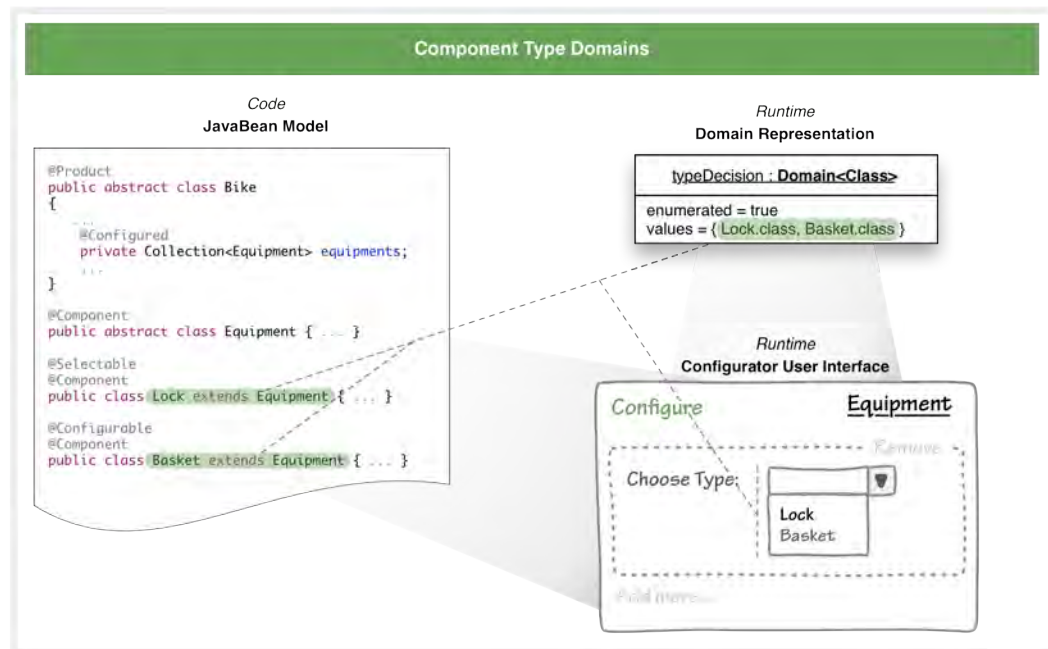


Figure 4.22. Runtime Representation of a Component Type Domain

Of course, a type decision is only needed, when multiple subtypes of a given component exist. For a configured part with an abstract type (e.g., **Equipment**), the component type domain by default contains all non-abstract subtypes of the given type (e.g., **Lock** and **Basket**). Otherwise, OpenConfigurator offers an annotation to restrict the available types to an enumerated set, see Section 4.4.4.1, “Component Type Domain Definition” below.

Attribute value domains. The most common case for configuration decisions is the specification of attribute values. These decisions must be backed with *attribute value domains*. In general, it's useful to differentiate two domain scopes:

- **Attribute level (monadic values).** Domain definitions may have attribute scope, that is, the definition solely defines the values of a dedicated attribute. Hence, a list of singular values is specified.
- **Component level (n-adic tuples).** Otherwise, domains may be defined on component level, providing values for all its n attributes. Here, the definition specifies a table, or more

precisely: multiple lists of values (one for each attribute), that are extracted from the provided tuples.

Figure 4.23, “Component Level and Attribute Level Domains” illustrates the difference between both domain scopes:

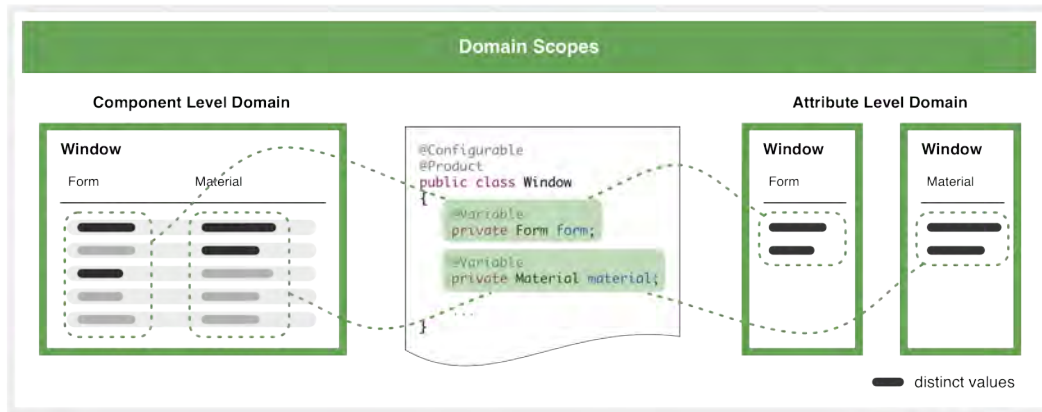


Figure 4.23. Component Level and Attribute Level Domains

In both cases, only distinct values are picked up into the attribute's domain.

Part values domains. Another common case for configuration decisions is the specification of parts. Similar to attribute decisions, these decisions must be backed with *part value domains*.

The domains for part values can be defined in terms of tuples in two ways:

- **Component level (n-adic tuples).** The domain defined on component level, that is, within the component definition, is used whenever the component is referenced as part of another component. It can be seen as the default domain for the component.
- **Part level override (n-adic tuples).** The domain for part components can also be overridden on part level, allowing only a particular subset of component instances to be used for a certain part.

We will provide an example for both definition methods below.

However, the definition of part value domains is optional in certain situations. It depends on the part's *specification method*:

- **Selection.** If the part component is annotated `@Selectable`, a part domain *must* be specified on component or part level explicitly.
- **Configuration.** If the part component is annotated `@Configurable`, a part domain *may* be specified on component or part level. If the domain is not specified though, the component may not contain regular attributes. Instead all attributes must be either declared as variable (`@Variable`), parameter (`@Parameter`) or calculated attributes (`@Calculated`).
- **Construction.** If the part component is annotated `@Constructible`, a part domain *must not* be specified, neither on component nor on part level.

Domains for other decisions. The domains for the other decisions mentioned in Section 3.1, “Basic Meta Model for Generic Product Modeling” cannot be modeled explicitly, but are implicitly modeled by the developer. For instance, the *component customization decision* is implicitly defined by the specification method used (see Section 4.4.3.1, “Specification Methods”).

Also, the *component variety decision* depends on the cardinality of the domain (see "Domain size" below): if there are multiple items contained in the domain, the user must choose a particular one (*alternative components*). If there is only a single one, the user has no choice (*fixed component*).

Nevertheless, candidates for explicit domains are possibly the *quantity decisions*, but currently our conceptualization doesn't foresee explicit definitions for these. Instead, the cardinality of multi-value attributes or multi-component parts is managed internally by the framework and may be restricted by extensional constraints only (see Section 4.4.5, "Constraint Modeling").

Domain size. Technically, an important aspect is the *domain size*, which strongly influences the configurator's computing performance. The domain size may be described as:

- **Empty.** An empty domain may result from constraint propagation, that is, all values have been removed from the domain due to the application of constraints (see "Constraint based configuration" in Section 3.3.3.5, "Implementation Aspects"). If the attribute is non-optional, the configuration problem cannot ever be solved and previous decisions must be backtracked in order to find a valid solution.
- **Fixed.** When only a single value remains within a domain, we say it is fixed to that particular value. The configurator automatically selects fixed domain values and doesn't require the user to specify the attribute explicitly (as no alternative options remain anyway).
- **Enumerated.** The domain values are stored as individual values. A domain is considered enumerated (also called *finite*), if it contains less than a certain *threshold amount* of values. Assuming the configuration problem solely contains enumerated domains, the configurator can (at least in theory) calculate all possible solutions in finite time. By continuously retaining only those values in a domain, that are part of a solution, the configurator can avoid decision backtracking entirely (see [Runte2006, pp. 57]).
- **Bounded.** The domain's values are limited to a lower and an upper bound value. While mathematically, the amount of values are countable, the domain size is usually too large to allow real-time propagation in interactive configuration processes.
- **Unbounded.** The domain doesn't specify an upper and/or lower bound value or the values are mathematically not countable (e.g., real numbers). Unbound domains cannot be propagated during interactive configuration.⁴⁷

An unbounded domain of a variable x may become bounded, when a constraint like $x < 10000$ is posted. A bounded domain, in turn, may become enumerated, when a constraint like $x \leq t$ is posted, where t is the threshold value, or when the domain's values are intensionally enumerated (see "Domain definition" below).

Domain Definition

The OpenConfigurator framework offers several ways to express **domain definitions**. Before describing concrete annotations, we will give an overview of the available definition methods first.

For every component type/attribute the domain is defined *implicitly* by its Java data type. For instance, the domain of an attribute extracted from a `boolean` field implicitly contains the values `true` and `false`.

⁴⁷Note that although computation technically, the Java data types `integer` (32 bit, 2^{32} values), `long` (64 bit, 2^{64} values), `float` (32 bit single precision floating point) and `double` (64 bit double precision floating-point) have inherent bounds (e.g., `Integer.MAX_VALUE`), in absence of further restrictions they're considered as being unbounded throughout this work.

A domain can also be defined *explicitly* by the developer using meta-data annotations, though. Using this *declarative* approach, domains are described either *intensionally* or *extensionally*:

Intensional (enumerative). An *intensional* definition means, that the exact values available are *enumerated*. Here, we can distinguish, whether the model developer lists the values *inplace* or defines them *externally*.

Inplace means, the developer embeds the list of values directly into the Java model, using a source code annotation (e.g., `@Domain("black", "white", "silver", "red", "blue")`). *Externally* means, he points to an *external* representation (e.g., by stating a query using the `@Domain.Query` annotation).

Enumeratively described domains are by definition *enumerated* (in the sense of the *domain size* described above).

Extensional. Defining domains *extensionally* means, that constraints are used to limit the implicitly specified value space of an attribute. For example, an unary constraint such as `@Max(1000)` states, that the upper bound of a domain is (at maximum) 1000. Theoretically, also binary and n-ary constraints (see Section 3.1.3, “Constraints: Domain Restrictions”), which reference more than one variable, restrict the domain of an attribute. However, in practice, propagating such constraints quickly becomes a complex operation, which is why these constraints are mostly validated a posteriori, hence, after the customer submitted a value.⁴⁸

Effectively both definition methods, intensional and extensional annotations, restrict the attribute domain's value space, which is why both can be considered *domain constraints* in the narrower sense. Throughout this work, however, we refer to intensional annotations as **domain annotations** and to extensional annotations as **constraint annotations**. Both concepts can be used in parallel to precisely define the allowed values for an attribute.

Figure 4.24, “Domain Definition Concepts” summarizes the domain definition concepts.

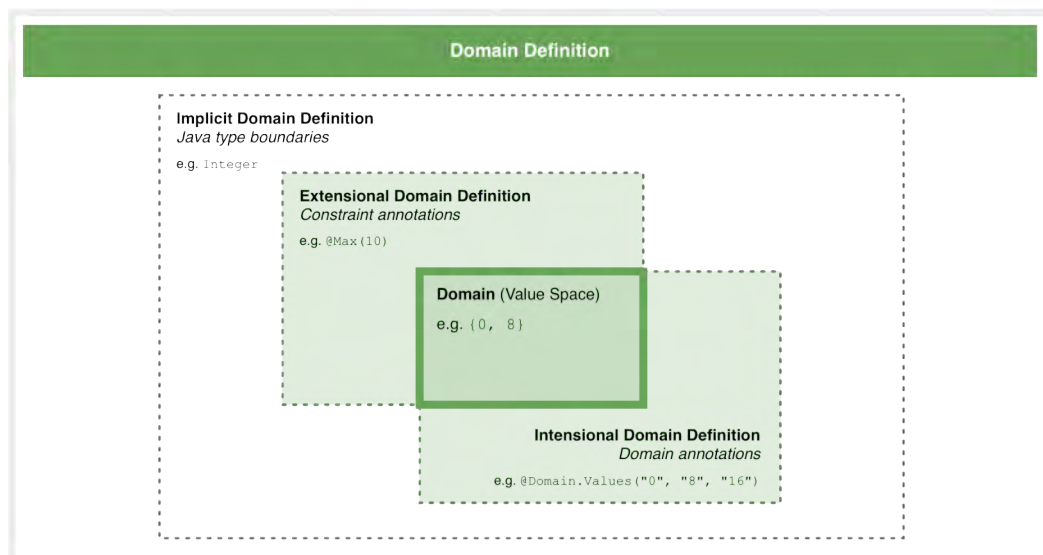


Figure 4.24. Domain Definition Concepts

As can be seen in the previous figure, the ultimate **value space of a domain** is determined by the *intersection of the implicit domain, extensional domain and intensional domain*.

⁴⁸Note that the OpenConfigurator framework currently doesn't even propagate unary constraints. Instead, only intensional domain annotations are considered while building up the domains' value spaces. All extensional constraints are checked after user input.

Also remember, that an attribute's domain may change at runtime, as the user may change dependent variables, add components with additional constraints or explicitly remove domain values.

Now it's time to describe the domain annotations along with the implicit domain definition behavior concretely. Constraint annotations are discussed in Section 4.4.5, “Constraint Modeling”.

Component Type Domain Definition

For component type decisions related to the specification of configurable parts, OpenConfigurator applies the following behavior:

Implicit type domain. The implicit *type domain* of a part corresponds to the (*concrete*) *type closure* of the part's base type. Hence, by default, all non-abstract subtypes of the given part type are contained in the type domain.

Explicit type domain annotations. The implicit type domain can be *narrowed* by applying the following annotations on the part:

Table 4.12. Type Domain Annotations^a

Annotation	Characteristic	Description
@TypeDomain	intensional, internal	Enumerates the possible types of a part. The listed types must extend the part's base type.

^aAdditional annotations to control the type domains are planned for a future version of OpenConfigurator.

Example 4.13, “Implicit and Explicit Definition of Type Domains” demonstrates the different component type definition methods, including the use of the @TypeDomain annotation.

Example 4.13. Implicit and Explicit Definition of Type Domains

```
public abstract class Bike
{
    ...
    @Configured
    public Collection<Equipment> getEquipments() { ... } ❶
    ...
}

public class CityBike extends Bike { ... }

public class MountainBike extends Bike
{
    @Configured
    @TypeDomain({ Lock.class }) ❷
    @Override
    public Collection<Equipment> getEquipments() { ... }
}

public abstract class Equipment { ... }
public class Lock extends Equipment { ... }
public class Basket extends Equipment { ... }
```

- ❶ The Bike class defines the equipment part without an explicit TypeDomain annotation. Thus, the domain falls back to the implicitly defined one.

Consequently, when configuring a CityBike, the type domain for the equipment part contains all non-abstract subtypes of Equipment: Lock and Basket.

- ❷ Within the MountainBike class, the equipment part is *redefined* (by overriding the getEquipments() method), narrowing the type domain to the Lock class by using the @TypeDomain annotation.

Thus, when configuring a MountainBike, a user can only configure Lock equipments.

Attribute Value Domain Definition

OpenConfigurator applies the following behavior for attribute domains:

Implicit attribute value domain. The implicit domains for component attributes are created depending on the attribute's Java type. The following mapping is assumed:

Table 4.13. Implicit Attribute Domains

Types ^a	Domain Characterization
boolean	Boolean domain, enumerated, values: true and false
byte, short	Numeric domain, bounded
integer, long, float, double	Numeric domain, unbounded
char, String	Character /String domain, unbounded
Enum (derivatives)	Symbolic domain, enumerated, values: Enum.values()
byte[], InputStream	Binary domain, unbounded

^aThe primitive object wrappers `Boolean`, `Integer`, etc. as well as their array equivalents (if not stated otherwise) belong to the group of their primitive equivalents, `boolean`, `int`, etc. and are not listed explicitly.

Explicit attribute value domain annotations. The implicit attribute value domain can be narrowed by applying the following annotations on the corresponding Java member (**attribute level domains**), respectively on the Java type declaring the member (**component level domains**):

Table 4.14. Attribute Domain Annotations^a

Annotation	Characteristic	Description
@Domain (attribute level)	intensional, internal	Enumerates the possible values of an attribute. The listed types are represented as Strings and must be coercible to the respective attribute type using the target type's <code>valueOf(String)</code> methods. The @Domain annotation can be used on properties only (field / setter method).
@Domain.Query (component level)	intensional, external	Allows to specify a query as the annotation's value. The query is interpreted by the underlying data provider (see "Data provider" in Section 5.2.2, "Services"). If no value is specified, the framework automatically queries all objects with the given type. By specifying the annotation's name element, the provider is requested to execute the corresponding <i>named query</i> (see [JSR3172009, p. 351]). The @Domain.Query annotation can be used on component classes and on part properties (field / setter method).

^aAdditional annotations to control the attribute value domains are planned for a future version of OpenConfigurator.

Effectively, the `@Domain.Query` annotation allows to **externalize configuration options into a database**, thereby, significantly easing the configurator knowledge base maintenance.

When annotations on both component level and attribute level are specified, OpenConfigurator intersects the resulting domains.

Example 4.13, “Implicit and Explicit Definition of Type Domains” demonstrates the use of the attribute domain annotations for the Bike, Wheels, Fork and Frame component.

Example 4.14. Implicit and Explicit Definition of Attribute Value Domains

```
@Selectable
@Domain.Query ❶
public class Wheels
{
    private String manufacturer;
    private double treadDepth;
    private BigDecimal price;
    ...
}

@Selectable
@Domain.Query("select f from Fork f where f.special = false") ❷
public class Fork
{
    private boolean suspension;
    private double suspensionStrength;
    private BigDecimal price;
    private boolean special;
    ...
}

@Configurable
public class Frame
{
    @Variable
    @Domain({ "16", "20", "24", "26", "28" }) ❸
    private int size;

    @Parameter
    private double height; ❹

    @Variable
    private Color color; ❺
    ...
}
```

- ❶ For the `Wheels` component, the attribute value domain is defined on component level by applying the default `@Domain.Query` annotation. The underlying data provider thus queries all components of type `Wheels` and collects the distinct values for the attribute domains (`manufacturer`, `treadDepth` and `price`).
- ❷ For the `Fork` component both component level and attribute level domain definitions are applied. The `@Domain.Query` annotation is evaluated by the data provider, which executes the specified query. Possibly, the returned tuples do not define the `suspensionStrength` attribute, which is annotated separately (otherwise the results are intersected).
- ❸ The attribute value domains for the configurable type `Frame` are defined on attribute level. For the `size` attribute the possible values are specified explicitly, using the enumerative `@Domain` annotation.
- ❹ The parameter attribute `height` is implicitly mapped to a numeric, unbounded double domain.

- ⑥ The color attribute's domain is implicitly mapped to an enumerated domain containing the values of the Java enum type `Color`.

Part Value Domain Definition

For part values, `OpenConfigurator` applies the following domain control behavior:

Implicit part value domain. If the part value domain is not explicitly defined according to the rules below, `OpenConfigurator` applies a default behavior depending on the nested component's specification method:

- **Selection.** If the nested component is `@Selectable`, `OpenConfigurator` behaves as if the part was annotated `@Domain.Query` (without attributes).
- **Configuration.** If the nested component is annotated `@Configurable`, the framework checks, whether the component contains regular attributes or variable attributes without attribute level domain definitions. If it does, it behaves as if the part was annotated `@Domain.Query` (without attributes). Else, the framework solely takes attribute level domain definition into account.
- **Construction.** The framework solely takes attribute level domain definition into account.

Explicit part value domain annotations. The implicit domain can be replaced by applying the following annotations on either the component class or the part property⁴⁹:

Table 4.15. Part Domain Annotations^a

Annotation	Characteristic	Description
<code>@Domain.Query</code> (component level)	intensional, external	Allows to specify a query as the annotation's value. The query is interpreted by the underlying data provider (see "Data provider" in Section 5.2.2, "Services"). If no value is specified, the framework automatically queries all objects with the given type. By specifying the annotation's name element, the provider is requested to execute the corresponding <i>named query</i> (see [JSR3172009, p. 351]). The <code>@Domain.Query</code> annotation can be used on component classes and on part properties (field/setter method).

^aAdditional annotations to control the part value domains are planned for a future version of `OpenConfigurator`.

Effectively, this leads to the following algorithm:

1. If `OpenConfigurator` detects a `@Domain.Query` annotation on the part's property, it evaluates it and returns the resulting domain.
2. Otherwise, if the part property is not annotated, but instead, the component class identified by the part's base type is annotated with `@Domain.Query`, `OpenConfigurator` uses this annotation to create the domain.

⁴⁹Note: if multiple `@Domain.Query` annotations are specified, most specific one is used. For instance, imagine an implicit annotation has been defined, one as component class annotation and one on the part property. In this case the part property's annotation would be used. The other annotations are simply ignored instead of being intersected. See the description of the applied algorithm further down this section.

3. If neither the part property nor the component class is annotated, OpenConfigurator falls back to the implicit behavior.

Note that OpenConfigurator supports component level domains only for `@Selectable` or `@Configurable` parts (cp. Figure 4.19, “Comparison of Specification Methods”). For `@Constructible` part components steps 1 and 2 of the algorithm above are effectively omitted, i.e. such annotations are simply ignored.

Example 4.15, “Implicit and Explicit Definition of Part Value Domains” shows the part specific usage of domain annotations:

Example 4.15. Implicit and Explicit Definition of Part Value Domains

```
@Selectable
@Domain.Query ❶
public class Wheels
{
    private String manufacturer;
    private double treadDepth;
    private BigDecimal price;
    ...
}

public abstract class Bike
{
    @Configured
    public Wheels getWheels() { ... } ❷
    ...
}

public class CityBike extends Bike { ... }

public class MountainBike extends Bike
{
    @Configured
    @Domain.Query("select w from Wheels w where w.treadDepth > 0.5") ❸
    @Override
    public Wheels getWheels() { ... }
    ...
}
```

- ❶ As the `@Selectable` component `Wheels` is annotated `@Domain.Query`, all `Wheels` instances are queried from the underlying data store, whenever `Wheels` is referenced as part. This is the default behavior for `@Selectable` components. Consequently the `@Domain.Query` annotation could have been omitted entirely.
- ❷ The `Bike` product references `Wheels` as a part, but does not re-define the domain. When a `CityBike` instance is configured, OpenConfigurator uses the `@Domain.Query` annotation of the `Wheels` class.
- ❸ For `MountainBike` instances, the `wheels`' part is *redefined* (by overriding the `getWheels()` method), narrowing the domain to all tyres with a `treadDepth` greater than 0.5, as defined by the `@Domain.Query`.

In this section, we've seen in detail, how domain values for all relevant elements of the generic configuration model can be defined.

This way, practical use cases can be implemented conveniently: the developer can specify the domain values for specific attributes, that is, the options for the corresponding decisions, directly within the domain model using annotations. Alternatively, he externalizes the option values into a database and solely specifies the correct queries within the model. The latter approach allows to easily change configuration behavior: an additional option can be incorporated into the configurator by simply inserting another row into the database.

In practice, it's often desired to specify a specific *default value* (default option) for a specific decision. The next section discusses, how these defaults can be defined.

4.4.4.2. Defaults

As active specification support feature (see *specification support* in Section 3.3.3.4, “Configuration Characteristics”), OpenConfigurator allows the definition of *defaults* using Java annotations. The concept of *defaults* in this context, applies to all decision types identified in Section 3.1, “Basic Meta Model for Generic Product Modeling” and can be understood as the automated pre-selection of a particular value from the corresponding domain without user intervention.

A *default* is applied, whenever a component, part or attribute is added to the configuration. OpenConfigurator applies one of the following actions with regard to defaults:

Implicit default behavior. For the various decision types, OpenConfigurator implements the default behavior as defined in Table 4.16, “Decision Default Behavior”:

Table 4.16. Decision Default Behavior

	Decision	Default Behavior	Explicitly Definable ^a
i.	Component type decision	None. The user must specify the exact value.	No
ii.	Component quantity decision	The cardinality of the initial property value, if any, is used to determine the default quantity. Otherwise, the quantity is set to 0.	No
iii.	Component variety decision	None. The user must specify the exact value.	Yes
iv.	Component customization decision	None. Depends on specification method.	Yes, indirectly.
v.	Attribute quantity decision	The cardinality of the initial property value, if any, is used to determine the default quantity. Otherwise, the quantity is set to 0.	No
vi.	Attribute valuation decision	The initial property value, if any, is used as default value. Otherwise, the user must specify the exact value.	Yes

^aExplicit default modeling capabilities for those decisions, marked as not explicitly definable, may be introduced in a future version of OpenConfigurator.

Explicit default value definition annotations. The categorization of definition approach introduced for domain definition equally applies to the definition of defaults, which is why a detailed discussion is not presented here again. In short, OpenConfigurator offers the following annotations for default value definition:

Table 4.17. Default Value Definition Annotations^a

Annotation	Characteristic	Description
@Default (attribute level)	intensional, internal	Allows to define a one or multiple default values, depending on the attribute's cardinality. The listed types are represented as Strings and must be convertible to the respective attribute type using the target type's <code>valueOf(String)</code> methods. The <code>value</code> attribute accepts expression values.

Annotation	Characteristic	Description
		The <code>@Default</code> annotation can be used on attribute properties only (field/setter).
<code>@Default.Query</code> (component level)	intensional, external	Allows to specify a query as the annotation's value attribute, which is executed by the underlying data provider. If no query is specified, the framework automatically queries all objects with the given type. By specifying the annotation's name attribute, the provider is requested to execute a named query. The executed query's result should contain a single tuple only, otherwise the first returned tuple is used as default. The value attribute accepts expression values. The <code>@Default.Query</code> annotation can be used on component classes and on part properties (field/setter method).

^aAdditional annotations to control attribute value domains are planned to be included in a future version of OpenConfigurator.

The following rules also apply:

- For *mandatory attributes or parts* only: whenever a particular domain gets fixed to a single value, that value is pre-selected and cannot be modified. This special case of default application is referred to as *auto completion*.
- Initial property values can only be determined for *non-abstract component types*, for technical reasons. Hence, the annotations for explicit default value definition discussed below, should be preferred (see the last definition in the example below).
- A default value must be contained within the respective domain and must not violate any defined constraints.
- For *parts with multiple default annotations* only: in order to determine the correct annotation to use, the algorithm described in Explicit part value domain annotations applies analogously.

Example 4.16, “Implicit and Explicit Definition of Default Values” demonstrates the usage of `@Default` annotations:

Example 4.16. Implicit and Explicit Definition of Default Values

```
public class MountainBike extends Bike
{
    @Configured
    @Domain.Query("select f from Frame f where f.type = 'MTB'")
    @Default.Query("select f from Frame f where f.type = 'MTB' +
        " and f.default = true") ❶
    @Override
    public Frame getFrame() { ... }
    ...
}

@Configurable
public class Frame
{
```

```
@Variable
@Domain({ "16", "18", "20", "24", "26", "28" })
@Default("24") ❷
private int size;

@Parameter
@Default("#{_this.size * 2.5}") ❸
private double height;

@Variable
private Color color = Color.BLACK; ❹
...
}
```

- ❶ For MountainBike instances, the frame part's default value corresponds exactly to the frame instance stored within the database, that has type set to MTB and default set to true, as stated by the corresponding query.
- ❷ For non-MountainBike instances, that don't otherwise specify a default query on the part property, attribute level defaults apply. For the size attribute, for instance, the value 24 is used as default.
- ❸ The @Default annotation of the height parameter shows the use of an expression to dynamically calculate the default value depending on another value.
- ❹ The color attribute defaults to the given initial property value, which is Color . BLACK. If Frame was an abstract type, using the equivalent notation @Default ("BLACK") would be recommended.

We've now discussed all data related aspects relevant for modeling of configurable products. Most importantly, we have explained, how the solution space is described *implicitly* by the Java data types and how it can be defined more precisely using *domain annotations*.

In the next section, we will show, how the solution space can be further restricted using *constraint annotations*.

4.4.5. Constraint Modeling

Basically, assigning model restrictions using constraints is one of the most important aspects of configuration modeling and product modeling. OpenConfigurator allows to define model constraints using Java annotations, inspired by the Bean Validation specification [JSR3032009].

In fact, OpenConfigurator is designed to be compatible with the Bean Validation specification, allowing to reuse any defined JSR-303 compatible constraint for configuration purposes. Importantly, this allows the configurator developer to easily define arbitrary new constraints. As long as they are defined in accordance with the Bean Validation specification (see [JSR3032009, pp. 4]), OpenConfigurator automatically considers them during the configuration process.

Consider Example 4.17, “Constraint Annotation Usage”, which demonstrates the usage of the JSR-303 defined @Max constraint to limit the maximum value of a JavaBean property:

Example 4.17. Constraint Annotation Usage

```
@Configurable
public class Window
{
    @Parameter
    @Max(200)
    private int width, height;
    ...
}
```

After the user specified some value for the width or height attribute, OpenConfigurator checks any defined constraint for these attributes by invoking their associated *validator*. For instance, for the `@Max` constraint, the Bean Validation implementation provides a validator, that applies to `int` values and which verifies, whether the given attribute value is consistent with the specified value. In this case, width and height must have values less than or equal to 200.

In its current version, the Bean Validation specification solely specifies *unary* constraints (see Section 3.1.3, “Constraints: Domain Restrictions”) applicable on property level. However, the specification also standardizes, how custom bean level constraints can be defined.

4.4.5.1. General Constraint Characteristics

For configuration purposes, unary constraints, as provided by JSR-303, are not sufficient to model real-world configuration restrictions adequately. Consequently, OpenConfigurator introduces numerous *binary* and even *n-ary* constraints to fulfill the most common configuration requirements. Before describing the available annotations in detail, we will discuss some general characteristics first.

Most constraints provided by the framework have a structure equivalent to the one depicted in Figure 4.25, “General Structure of OpenConfigurator Constraints”:

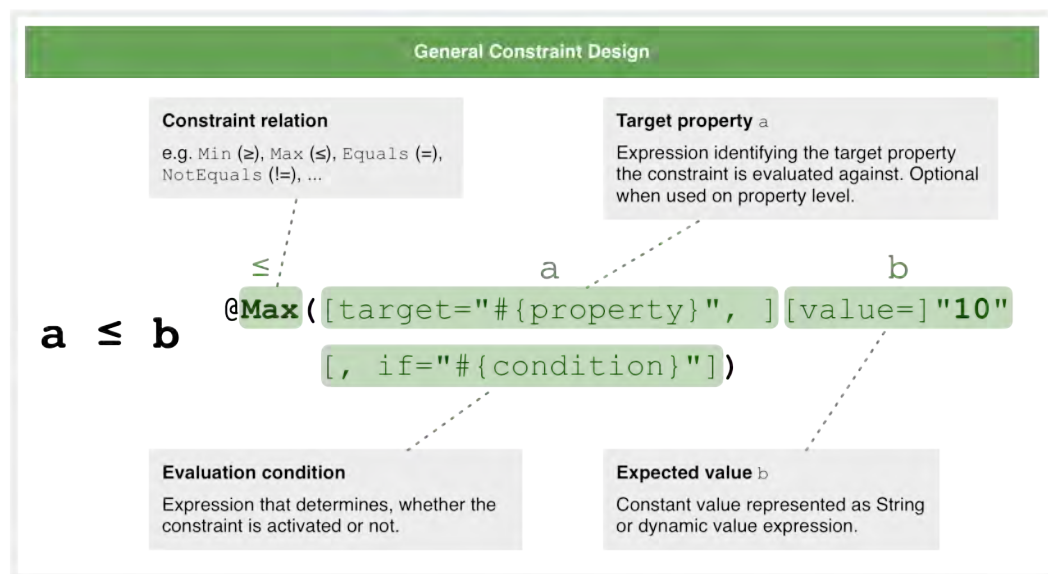


Figure 4.25. General Structure of OpenConfigurator Constraints

The annotation name identifies the *constraint relation*. OpenConfigurator implements the common mathematical binary comparison relations. The *target property* identifies the attribute, whose value is being evaluated. This annotation element is optional, if the annotation is being used on property level (field / getter method). In the following, the target property is referred to using the symbol a . The annotation's value element represents the *expected value* of the evaluated property. While a static value can be specified as `String` (effectively leading to the constraint being unary), in most cases a value expression is used, pointing to a second property (binary constraint). We refer to the expected value using the symbol b .

Constraint Usage

Basically, the constraints declared in this way can be used in two flavors, namely as *property level* and as *bean level* annotations, as Example 4.18, “Constraint Annotation Usage Methods” shows:

Example 4.18. Constraint Annotation Usage Methods

```

@Constructible
@Max(target = "#{_this.length}", value = "#{_this.roomLength}") ❶
public class Kitchen
{
    @Parameter
    private int roomWidth, roomLength, roomHeight;

    @Parameter
    @Max("#{_this.roomWidth}") ❷
    private int width;

    @Parameter
    private int length;

    @Parameter
    @Max("250") ❸
    private int height;
    ...
}

```

Constraint usage on property level. Mostly, a constraint is applied on *property level*, that is, an attribute, which acts as target, is annotated with the constraint annotation directly (see ❷ and ❸). In this case, quite often the expected value expression points to another variable effectively expressing a binary relationship (see ❷)⁵⁰.

The above example could be interpreted as follows: "The kitchen width's maximum value corresponds to the roomWidth". The other @Max constraint ❸ states a static upper bound of 250 for the kitchen's height.

Constraint usage on bean level. Another approach, semantically equivalent to the one above, is the application of a constraint on *bean level*, plus specifying the target attribute explicitly ❷. However, this approach is considered less readable and should optimally be avoided⁵¹.

Additionally, there's yet another potential approach of specifying the maximum value relationship between length and roomLength: using the bean level @ScriptAssert⁵² constraint annotation:

```

@ScriptAssert(lang="javascript",
              script="_this.length =< _this.roomLength")

```

However, the implicit constraint expression is much harder to process technically, since the developer may specify arbitrary complex scripts in there. Without a complex compilation of the script, this approach does *not* allow, for instance:

- to map a constraint violation to a particular attribute.
- to automatically provide useful help or error messages.
- to propagate the constraint, that is, to filter domain values *a priori*⁵³.

⁵⁰Note that due to a technical restriction of the Bean Validation specification and its reference implementation (Hibernate Validator), which is used within the OpenConfigurator framework, using expressions on property level doesn't currently work. For this reason, binary constraints cannot be expressed on property level at the moment. See <https://hibernate.onjira.com/browse/BVAL-240> and specifically <https://hibernate.onjira.com/browse/BVAL-237>, last accessed June 14th, 2012.

⁵¹Unfortunately, it's currently the only approach supported in a standardized manner, see the previous footnote.

⁵²The @ScriptAssert annotation is part of Hibernate Validator as of version 4.1, see <http://docs.jboss.org/hibernate/validator/4.1/api/org/hibernate/validator/constraints/ScriptAssert.html>, last accessed June 14th, 2012.

⁵³As stated in Section 4.4.4.1, "Domain Definition", processing constraints during domain construction is not currently implemented. Solely domain annotations are evaluated during domain creation.

Conditional Evaluation

The built-in constraint annotations also provide a `condition` element, which takes a boolean expression. If the expression evaluates to `true`, the constraint is *active* and processed during configuration validation. If it is `false`, it's considered *inactive* and consequently not processed.

The usage of the `condition` element allows modeling conditional constraint behavior flexibly. Logically, conditional constraint evaluation corresponds to *implication*. Consider the following constraint as an example:

```
@Max(target="x", value="y", condition="a == b")
```

Written as a predicate logical expression, this constraint correspond to: $a \equiv b \Rightarrow x \leq y$.

Again, the concepts introduced so far apply to any built-in constraint provided by the OpenConfigurator framework but must be re-implemented for any custom constraint.

We will describe the concrete constraints, shipped with the OpenConfigurator framework, in the next section.

4.4.5.2. Supported Constraints

Beyond the unary constraints provided by Bean Validation (`@NotNull`, `@Null`, `@AssertTrue`, `@AssertFalse`, `@Max`, `@DecimalMin`, `@Min`, `@DecimalMax`, `@Digits`, `@Size`, `@Pattern`, `@Past`, `@Future`, `@Valid`, for their exact semantics, see [JSR3032009, p. 101-112]), that merely take static arguments, OpenConfigurator provides a number of different constraint annotations.

All these built-in annotations support dynamic values using expressions, which allows to use them as unary or binary constraints in the same way. Also, all annotations support conditional evaluation.

The following list gives an overview of the available constraint annotations. For the full list, accompanied with usage examples, refer to Appendix A, *Constraints*:

Logical/arithmetical comparison constraints. In this category, OpenConfigurator offers constraints such as `@Equals`, `@NotEquals`, `@Min` and `@Max`, which correspond the operations $a = b$ (i.e. the equals method for Java objects), $a \neq b$, $a \geq b$ and $a \leq b$.

Cardinality constraints. Restrict the cardinality of collections, including exact cardinality definitions using `@Cardinality`, lower bounds using `@Cardinality.Min` and upper bounds using `@Cardinality.Max`.

Mathematically, this corresponds to $|A| = b$, $|A| \geq b$ and $|A| \leq b$.

Additionally, there are two additional cardinality annotations, which are frequently used, namely `@Optional` (equivalent to `@Cardinality.Min(0)`) and `@Required` (equivalent to `@Cardinality.Min(1)`). These can be used for non-collection properties, too.

Compatibility constraints. Express compatibility relationships by describing the set of compatible items (positive formulation). We distinguish type compatibility (`@Compatible.Type`) and compatibility based on a condition (`@Compatible.Matches`), both on single element and collection level (`@Compatible.ElementType`, `@Compatible.ElementMatches`).

Mathematically, type constraints can be described as follows: $\text{typeOf}(a) \in \{x \mid x = b \vee x \in \text{subTypeOf}(b)\}$ for singular elements, re-

spectively $\forall e \in \text{elementOf}(a) : \text{typeOf}(e) \in \{x \mid x = b \vee x \in \text{subTypeOf}(b)\}$ for collections. Matching constraints correspond to $\text{condition}(a) = \text{true}$, respectively $\forall e \in \text{elementOf}(a) : \text{condition}(e) = \text{true}$.

Incompatibility constraints. Express incompatibility relationships by describing the set of incompatible items (negative formulation). They can be seen as the inverse versions of compatibility constraints. They're frequently used for cases, where it's simpler or more appropriate to describe excluded items. In other respects, they share the same semantics with their positive equivalents: type incompatibility (`@Incompatible.TypeNot`, `@Incompatible.ElementTypeNot`) and conditional compatibility (`@Incompatible.MatchesNot`, `@Incompatible.ElementMatchesNot`).

Complex constraints. Additional, more complex constraints include `@Satisfies` and `@Relational`. Both of them are component level constraints applied on Java types. The former one is equivalent to the aforementioned `@ScriptAssert` constraint. It can be used to evaluate arbitrary scripts, that evaluate to true or false. The latter one, `@Relational`, imposes a relational dependency across the attributes of a component. The individual tuples, that form the relation, must be provided by a component level domain annotation (see Section 4.4.4, "Data Modeling").

Again, refer to Appendix A, *Constraints* for examples of using these constraints.

With the declaration of constraints, that are used restrict the solution space of a configuration problem, our conceptualization of a modeling language for configurable products is complete.

We are going to demonstrate these modeling capabilities in Chapter 6, *Evaluation and Validation*, where a complete case study for customizable bikes will be realized.

4.5. Configuration Procedure

Continuing the pattern from the previous chapters, discussing various topics from both a product and a process perspective, we described the *static, product related* modeling capabilities in the section before. Now, we will shift to the *process view* again: we'll demonstrate, how the configuration domain model, as modeled in the previous section, is turned into a configuration process. In other words, we are going to describe, how the *configuration decisions* are extracted from the domain model and in what way they are used to *drive the configuration process*.

In short, the **configuration procedure**, as realized by the OpenConfigurator framework will be discussed. While describing a general object-oriented configuration process in the first section, the second part will walk through a concrete example.

4.5.1. The Object-Oriented Configuration Procedure

Figure 4.26, "The Object-Oriented Configuration Procedure" gives an overview of the configuration process realized by the OpenConfigurator framework.

Basically, the configuration procedure must facilitate the **configuration process** as discussed in Section 3.2.2.1, "The Global Configuration Process". To recap that process, now with the understanding of object-oriented product modeling, the steps can be interpreted as follows:

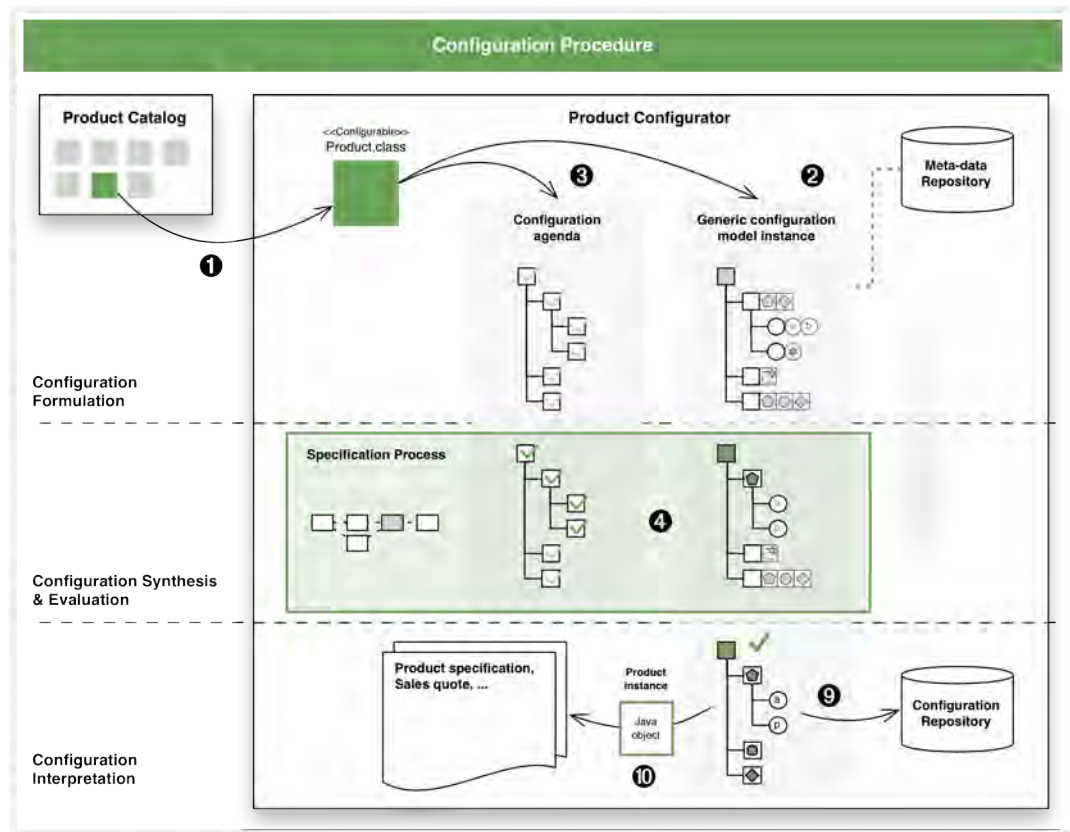


Figure 4.26. The Object-Oriented Configuration Procedure

Configuration formulation⁵⁴

Assuming the user starts a new configuration, the initial step in the configuration process is the selection of the base product, which is to be configured ❶. In terms of our object-oriented configuration approach, this means, that the user must **select a configurable product type**, that is, a Java class of the domain model. We will refer to the selected type as *configured type*.

Technically, the framework looks up a descriptor from the meta-data repository and **instantiates a new generic configuration model instance** based on that descriptor ❷. Moreover, the configurator **initializes the configuration agenda** ❸, which contains the *specification tasks*, the user has to perform. The specification tasks basically correspond to the configuration, decisions that have been discussed in Section 3.1, “Basic Meta Model for Generic Product Modeling”. Thus, the configuration agenda can be seen as an hierarchical data structure, that reflects the *decision model* of a configuration. Additionally, it backs the configuration process by tracking a list of already taken and remaining decisions.⁵⁵ With the generic configuration model instance and the configuration agenda being initialized, the actual specification activities can begin.

⁵⁴Note that OpenConfigurator, at the current stage, solely supports product-centric/structure-oriented configuration (see Section 3.3.3.4, “Configuration Characteristics” respectively Section 7.1.2, “Implementation Characterization”), which makes elicitation and mapping of customer requirements as part of the configuration formulation obsolete.

⁵⁵Currently, the actual decision sequence is also derived from the configuration agenda, which is why OpenConfigurator can be considered to implement a *structure-oriented process scheme* (see Section 3.3.3.4, “Configuration Characteristics” respectively Section 7.1.2, “Implementation Characterization”).

Configuration synthesis and evaluation

Next, the user iteratively **specifies configuration decisions** based on the configuration agenda ④. The specification process is described graphically in Figure 4.27, “Abstract Specification Process for Configurable Products”.

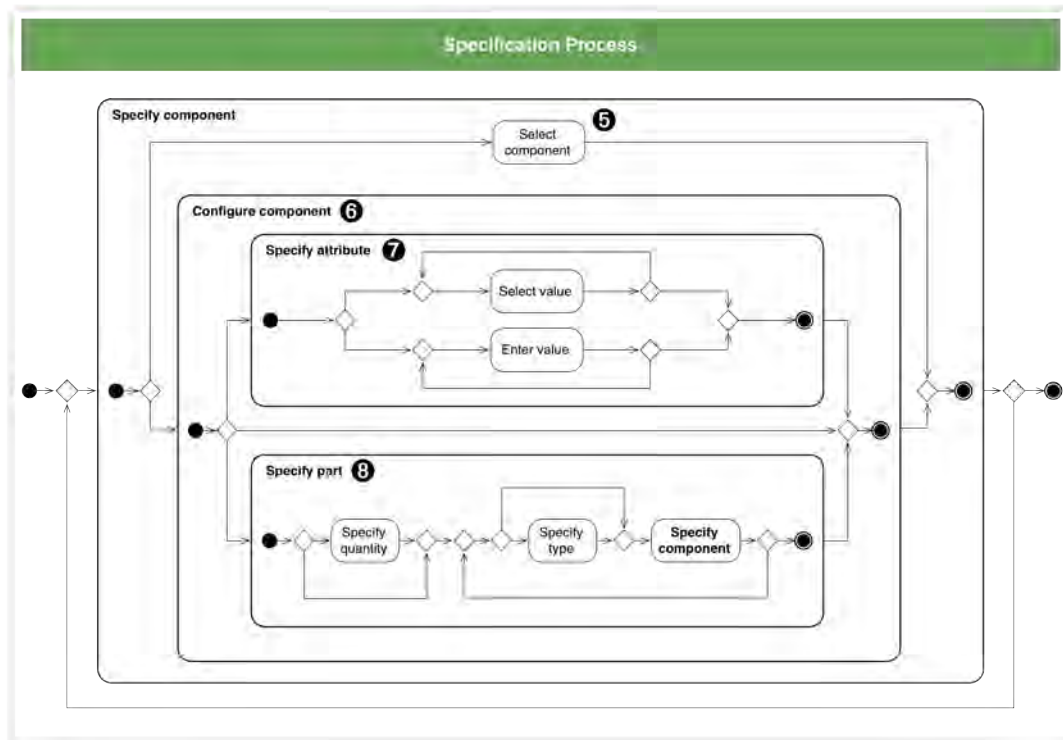


Figure 4.27. Abstract Specification Process for Configurable Products

Component specification. The process incorporates most of the decisions identified in Section 3.1, “Basic Meta Model for Generic Product Modeling”. For the *specification of a particular component*, the configurator offers the user, to either:

- *select* an existing component instance ⑤. The configurator will do so, for components using the specification method @Selectable (see Section 4.4.3.1, “Selection”).
- or asks him to *configure/construct* a new instance ⑥. This is the case for components using the specification methods @Configurable or @Constructible (see Section 4.4.3.1, “Configuration” respectively Section 4.4.3.1, “Construction”).

In the first case, the user simply chooses an instance queried from the database and is done. In the second case, the user configures a custom instance, which is not yet stored in the database. That means, he **specifies all variable attributes** of a component (annotated @Variable or @Parameter) ⑦ and **specifies any configurable part** (annotated @Configurable) ⑧:

Attribute specification. Depending on the attribute's cardinality and domain type, the user possibly repeatedly:

- *selects a value*, in case the domain is enumerated, or
- *enters a value*, if the attribute's domain is bounded or unbounded (see “domain size” in Section 4.4.4.1, “Domain Characteristics”).

If the domain value is fixed, the single possible value is used. Moreover, in case the domain is empty and the attribute has been marked @Required, a configuration error is indicated.

Part specification. For any configurable part, the user first needs to *specify the quantity*, if the part in question is a plural part (see Section 4.4.1.3, “Parts”). Then he *specifies the type* of the part's nested component, but only if multiple alternatives are available, that is, the type domain of the nested component contains multiple elements (see “Component type domains” in Section 4.4.4.1, “Domain Characteristics”). Finally, he recursively specifies the nested components. Depending on the part's quantity, the user may repeat this process unless any cardinality constraints are violated (see “Cardinality constraints” in Section 4.4.5.2, “Supported Constraints”).

The actual configuration process is finished, when the user specified all components and validation succeeds.

Configuration Interpretation

As stated in Section 4.3, “The Generic Configuration Model”, the generic configuration model instance exposes a valid configuration as an **instance of the configured type**, that is, the result of the configuration is a regular Java object. While the configurator automatically stores the configuration along with any protocol information in the configuration repository ⑨, the host application can perform arbitrary operations with the resulting object. This includes, for instance, **transformations into the respective specification documents** ⑩.

To illustrate the configuration procedure, at this point, we want to provide a more elaborated example. We will explain and illustrate the configuration procedure for a bike step-by-step in the next section.

4.5.2. Example Configuration Procedure

Consider the domain model depicted in Figure 4.28, “Configuration Domain Model for Customizable Bikes (Excerpt)” as exemplary configuration model. Note that we stripped most attributes and sub-components for brevity, as the example shall concentrate on the key concepts. For the full example, including source code, refer to Chapter 6, *Evaluation and Validation* respectively Appendix B, *Example Domain Model: Bike*.

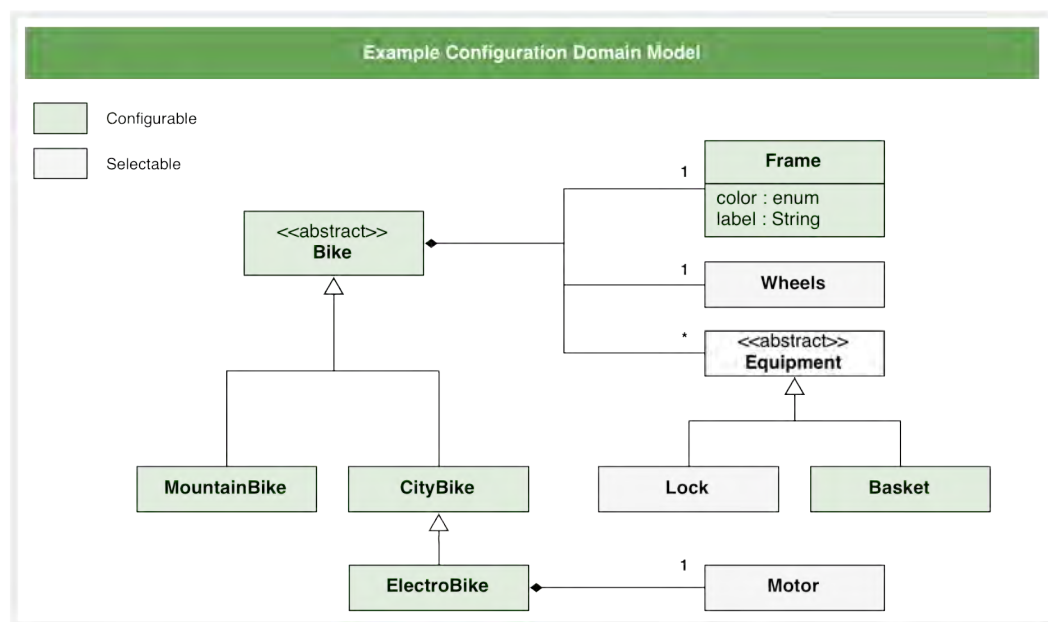


Figure 4.28. Configuration Domain Model for Customizable Bikes (Excerpt)

The model consists of a Bike class, that itself consists of a Frame component, wheels and any number of Equipments. While Bike is an abstract type, MountainBike, CityBike and

ElectroBike are concrete specializations. An ElectroBike additionally aggregates a Motor component. Also Equipment is an abstract entity having Lock and Basket as concrete specializations.

The classes on the left hand side, Bike, MountainBike, CityBike and ElectroBike are considered *product types*. They're annotated @Product. The ones on the right hand side, Frame, wheels, Lock, Basket and Motor are *component types* annotated @Component. While all product types are marked @Configurable, that is, the user may create custom instances of them, all component types except Frame are @Selectable. That means, the user will only be able to select an existing component instance for these, without being able to further parameterize them. Solely the frame part of the bike can be parameterized in this example. For the abstract class Equipment no specification method is defined, because its subtypes themselves define, whether they are selectable (Lock) or configurable (Basket).

Now, the user chooses to configure a custom bike by selecting the Bike product type from the catalog. Respectively, the configuration is started with the abstract Bike class being the configurable type. The framework looks up the meta-data descriptor (an instance of ComponentDescriptor, see Section 4.3.1, "Elements of the Generic Configuration Model and Meta Model") for the Bike class and initializes a generic configuration model instance plus an instance of the configuration agenda accordingly.

The initial state of the agenda as well as the model is depicted in terms of an UML object diagram in Figure 4.29, "Step 1: Initialized Model and Agenda".

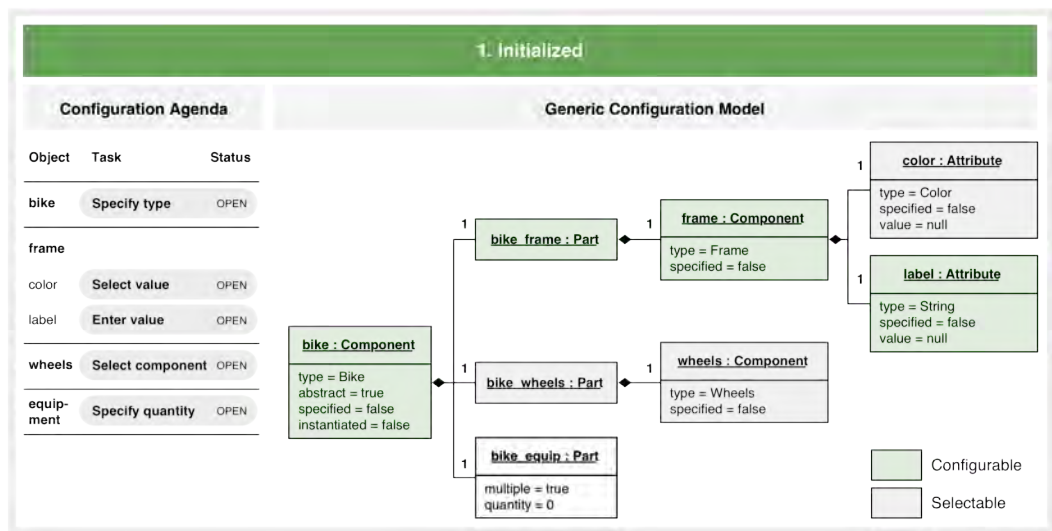


Figure 4.29. Step 1: Initialized Model and Agenda

The component model contains Component, Part and Attribute instances as defined by the mapping in Section 4.3.2, "Model Mapping". Since nothing has yet been explicitly specified by the user, all specified properties of these elements are set to false (see Table 4.4, "Component State Variables" in Section 4.4.1.1, "Responsibilities"). Moreover, the abstract flag of the bike component is set to true, indicating that no concrete type has been chosen for that component yet. Consequently, bike cannot yet be instantiated as reflected by the corresponding instantiated flag. Be aware of the fact, that for all parts with an exactly specified cardinality (in this case all parts except the equipment), the nested components are initialized upfront.

The configuration agenda contains decisions for each initialized component, part and attribute instance. Again, these decisions correspond to the decision types identified in Sec-

tion 3.1, “Product Models”. Notice that while the agenda pretends to impose a fixed decision sequence (from top to bottom), the user can specify the decisions in an arbitrary order. For example, he must not start by specifying the bike's type, but may instead start by entering a value for the label attribute. In fact, the user can even specify an attribute of a not yet instantiated bean: for instance, imagine bike had an attribute manufacturer. In this case the user could specify the manufacturer right before selecting a concrete type for the bike. In “regular Java” this wouldn't be possible: you had to instantiate a concrete class *before* setting any property value.

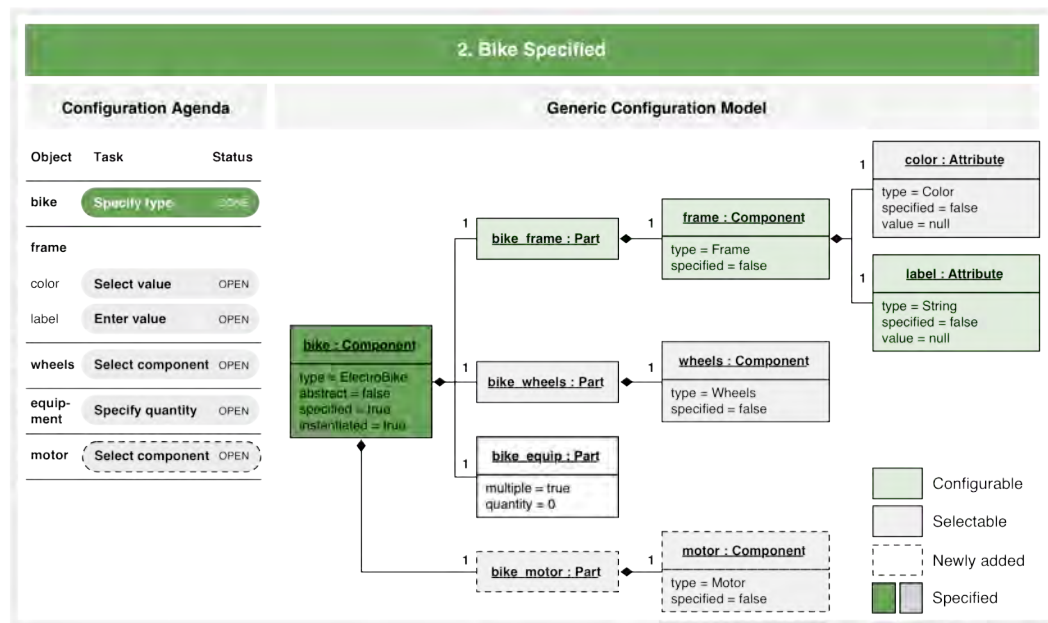


Figure 4.30. Step 2: Bike Type Specified

Nevertheless, for the *component type decision* of bike, the user chooses `ElectroBike` as the bike's type. The resulting state is visualized in Figure 4.30, “Step 2: Bike Type Specified”. The framework immediately updates the model and adds the motor part, including its nested component instance, to the configuration. Internally, `OpenConfigurator` recognizes that `ElectroBike` is non-abstract and immediately instantiates the underlying `JavaBean` instance, managed by the bike component. The component is marked as being specified.

Moreover, the configuration agenda is automatically synchronized with the configuration model: the *Specify type* task is marked as done and a *Select component* task is added for the motor component.

Next, we assume the user specified the frame component's attributes (*attribute valuation decisions*): for the `color` attribute he selected `Silver` from the attribute's domain. The domain values, corresponds to the values of the `Color` enumeration, a custom Java enum type. For the `label` attribute he entered the value `MyBike`.

As depicted in Figure 4.31, “Step 3: Frame Attributes Specified ”, again the configuration agenda is synchronized with the configuration model's state.

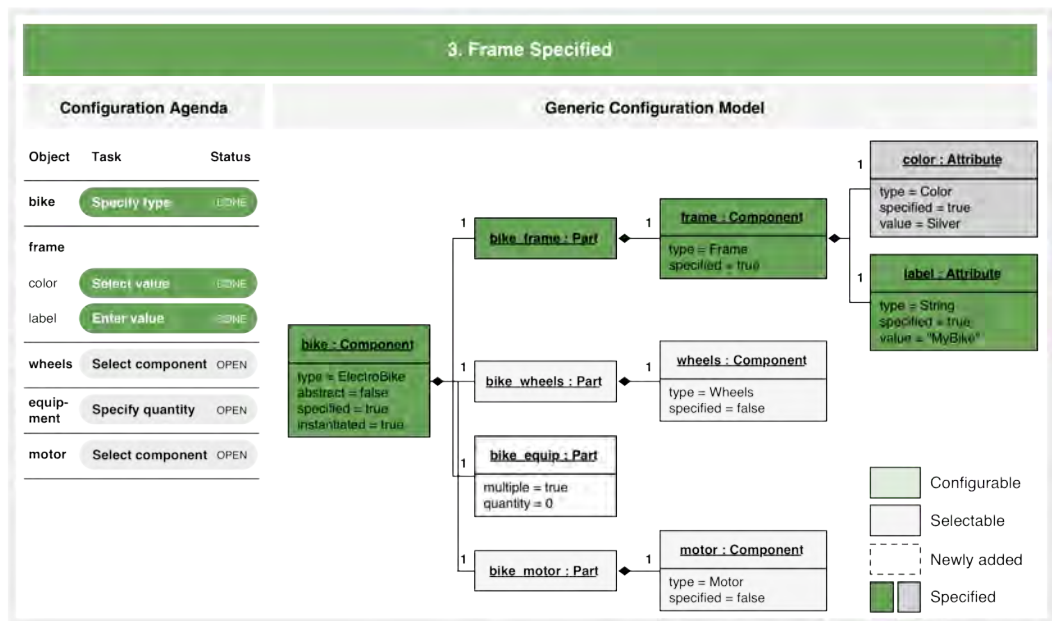


Figure 4.31. Step 3: Frame Attributes Specified

In step 4, the user specified the wheels part by selecting an existing component from the wheels' domain (*component variety decision*). For example, this might have been pre-defined slick tyres.

The resulting state is shown in Figure 4.32, "Step 4: Wheels Specified".

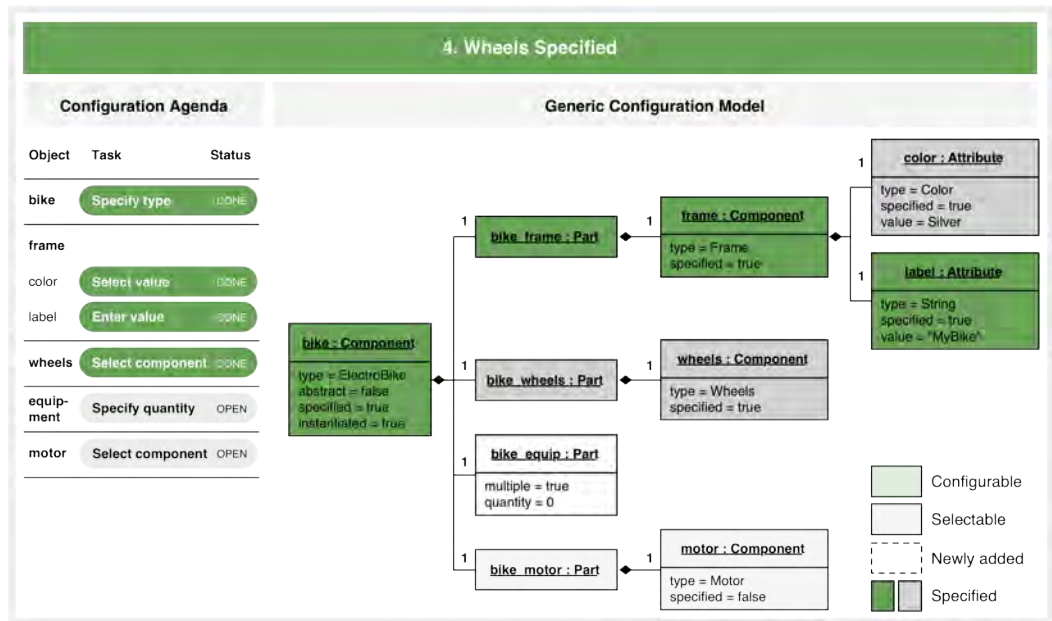


Figure 4.32. Step 4: Wheels Specified

Next, the user wants to add some equipment: a bike lock. As described in Section 4.5.1, "The Object-Oriented Configuration Procedure" "Configuration synthesis and evaluation", this is a two-step process: in the first step, the user increases the quantity of the equipment part (*component quantity decision*). The configurator automatically instantiates a new component, equip1, and adds it to the configuration model. The type of the newly added component is left abstract.

As can be seen in Figure 4.33, “Step 5: Unspecific Equipment Added”, the agenda is synchronized accordingly.

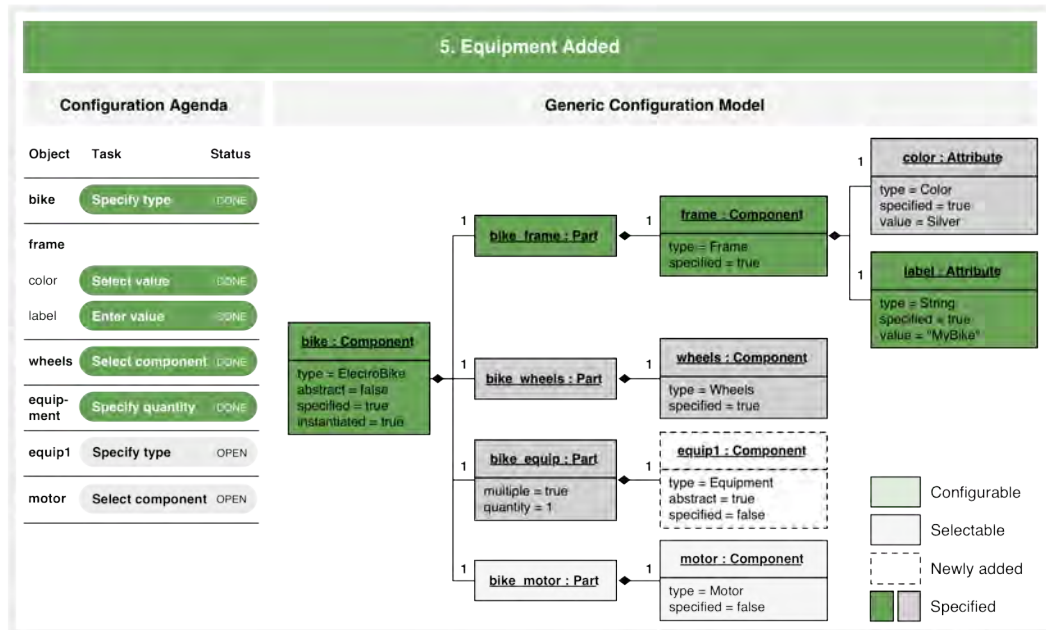


Figure 4.33. Step 5: Unspecific Equipment Added

In the second step of the equipment part specification task, the user chooses Lock as the concrete type of the equip1 component (*component type decision*) and selects an instance from the Lock classes' domain (*component variety decision*).

The updated state is shown in Figure 4.34, “Step 6: Equipment Specified ”.

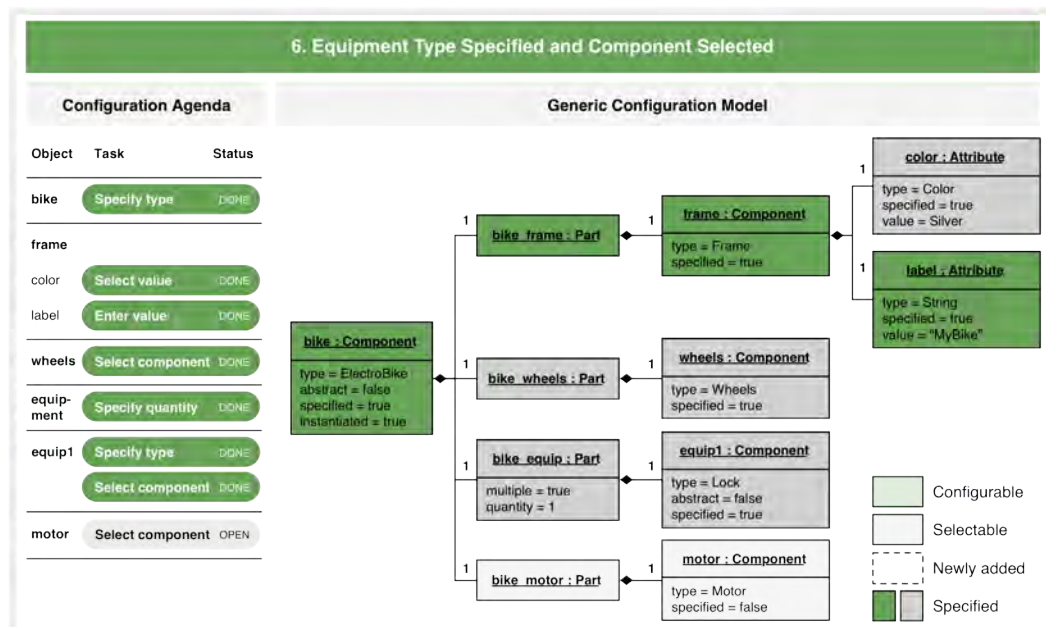


Figure 4.34. Step 6: Equipment Specified

Finally, the user specifies the motor component (*component variety decision*), which completes the configuration. Since all components, parts and attributes now have been specified and no additionally defined constraints have been violated, the configuration is *complete* and *valid*.

The final configuration state is depicted in Figure 4.35, “Step 7: Configuration Completed”.

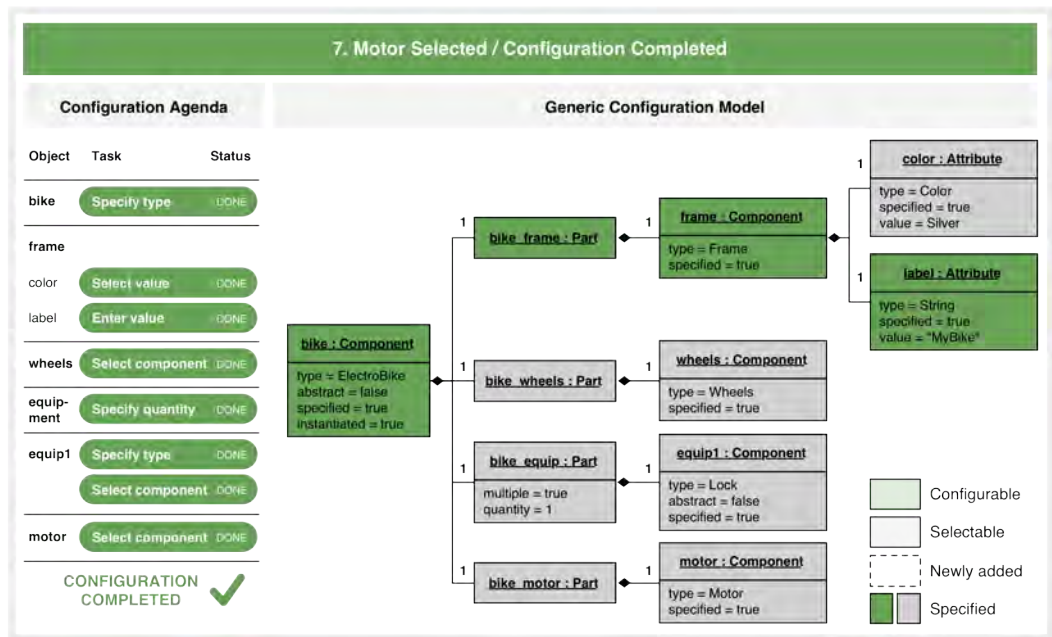


Figure 4.35. Step 7: Configuration Completed

The configuration is persisted to the configuration repository. The result, a regular JavaBean instance of type `ElectroBike` is shown in Figure 4.36, “Configuration Model Result Object”. The instance is fed to post-processing modules, that may, in turn, generate specification documents.

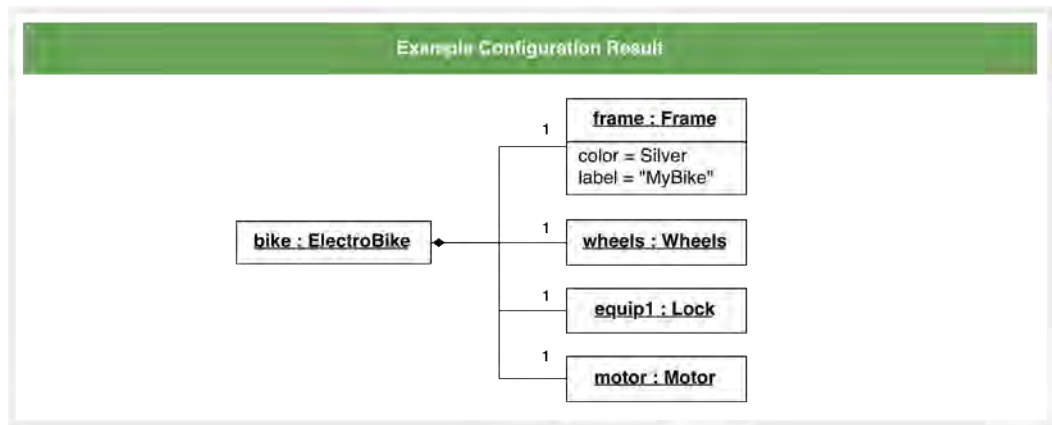


Figure 4.36. Configuration Model Result Object

This completes our example walk-through. We've seen how configuration proceeds within an object-oriented model and how the `OpenConfigurator` framework handles the configuration state internally. In the following section, we'll finally take a closer look at the *configuration agenda*.

4.5.3. Configuration Agenda

Configuration Process versus Configuration Agenda

In general, a *process* can be defined as a *course of action*. Moreover, a *deterministic process* describes a course of action, where each activity depends on its predecessors. A process has a characteristic structure, but not necessarily implies a pre-defined order of activities.

However, in the context of this work, when talking about the **configuration process**, we usually *do imply*, that there is a more or less pre-defined sequence, in which *configuration activities* are executed. In this sense, the configuration process is comparable to the *screenflow* of the configurator, see Figure 4.37, “Configuration Process as Screenflow”.

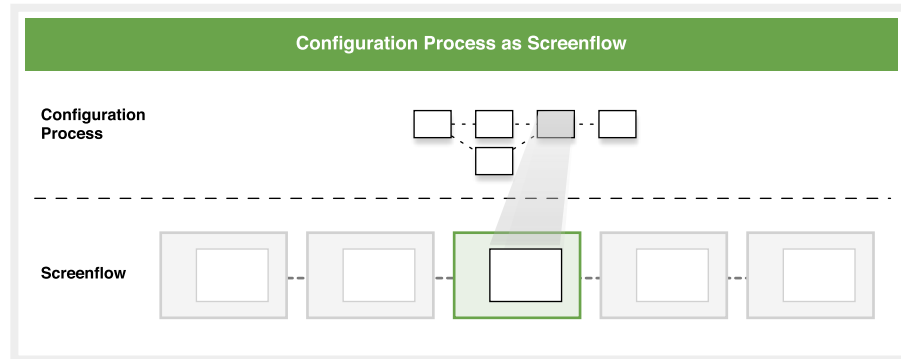


Figure 4.37. Configuration Process as Screenflow

While OpenConfigurator does not currently implement a way to *customize the configuration process* (screenflow), an extension to the framework that allows to declaratively describe the configuration process is planned for a future version.

The **configuration agenda**, in turn, solely defines the **decision model of a configuration** process. While not implying any decision sequence, the agenda model captures, *what* decisions need to be taken during the configuration process but *not when* these decision occur.

In essence, the aforementioned **configuration process customization** feature would be realized as a *mapping of agenda items to different process steps* (screens). This concept is shown in Figure 4.38, “Configuration Process Customization”.

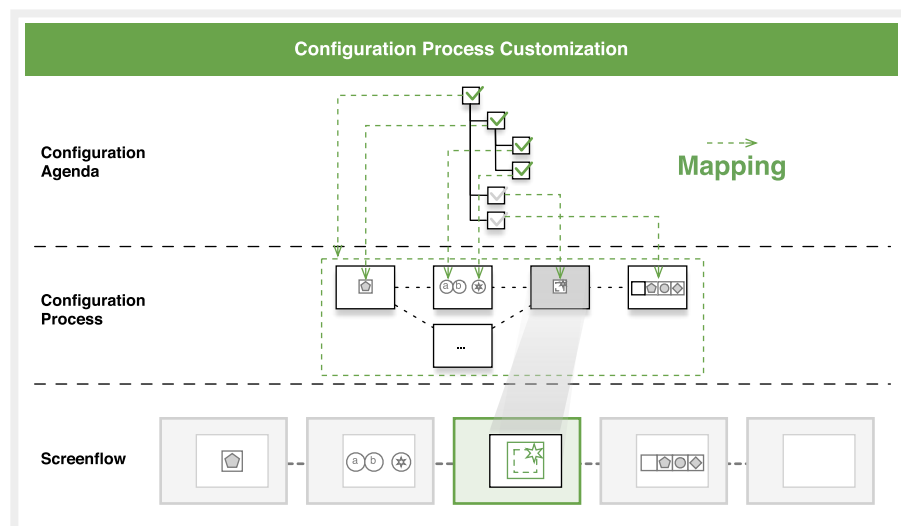


Figure 4.38. Configuration Process Customization

Configuration Agenda Characteristics

Again, the *configuration agenda* realizes the decision model of a configuration. It is an hierarchical structure composed of **configuration tasks**. Hence, the configuration agenda can also be seen as the *task based view of a configuration*.

A *configuration task*, in turn, represents a particular *configuration decision*, that must be taken in order to complete the configuration procedure. This relationship is depicted in Figure 4.39, “Relationship between Configuration Decisions, Tasks, and the Generic Configuration Model”.

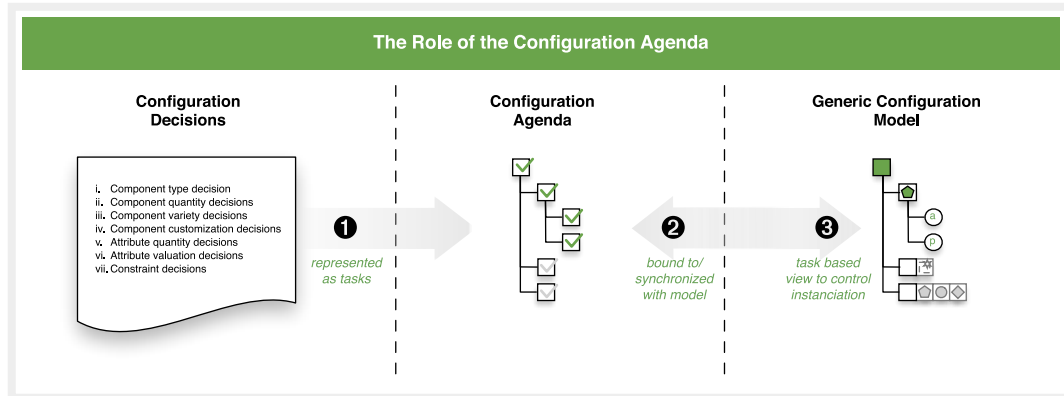


Figure 4.39. Relationship between Configuration Decisions, Tasks, and the Generic Configuration Model

- ❶ The configuration agenda represents the configuration decisions identified in Section 3.1, “Product Models” in terms of configuration tasks.
- ❷ The configuration tasks are extracted from the generic configuration model and subsequently synchronized to reflect the state of the configuration.
- ❸ Effectively, the configuration agenda provides a task based view to control the generic configuration model.

We’ve identified all relevant configuration decisions related to object-oriented product models in the respective sub-sections of Section 3.1, “Product Models”. We’ll discuss the mapping of configuration decisions to configuration tasks below.

Configuration tasks are characterized as follows. A task:

- is *bound* to a particular element of the generic configuration model,
- may be a *composite task* that contains *nested tasks*,
- provides a *domain* that is related to the corresponding decision (e.g., it maintains the domain for a component type decision),
- is marked as *complete*, if a particular value from the domain has been assigned (*specified*) and that value is *valid*,
- may become *obsolete*, if the represented decision does not provide any alternatives to choose from (in other words, the domain containing only a single value is *fixed*, see “Domain size” in Section 4.4.4.1, “Domain Characteristics”).

Notably, at runtime, the OpenConfigurator framework automatically extracts the configuration agenda from the generic configuration model and synchronizes the tasks’ states with the models continuously. Technically, this is realized with the help of OpenConfigurator’s sophisticated event infrastructure (see “Generic Configuration Model” in Section 5.2.2, “Client Configuration Interface”).

Agenda Task Types

The Task's type hierarchy is shown in Figure 4.40, "Agenda Tasks Type Hierarchy":

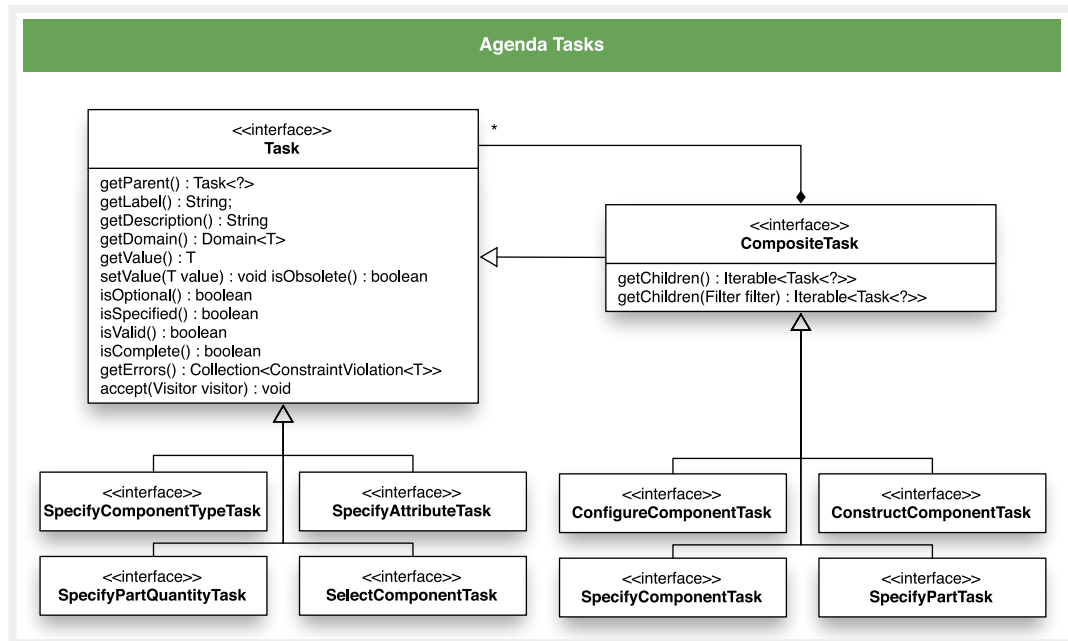


Figure 4.40. Agenda Tasks Type Hierarchy

The parameterized Task interface can be seen as "convenience interface", that aggregates methods from the Data and the Validation facet. The type parameter T corresponds to the model element's type, to which the particular task is bound. A SpecifyComponentTypeTask<T>, for instance, is bound to a Component<T>, whereas SpecifyAttributeTask<T> is bound to an Attribute<T>. The binding is represented by the same named interface Binding<T>, which all stated subtypes of Task and CompositeTask implement.

We will discuss the concrete mapping between configuration decisions, tasks and the generic configuration model in the following.

Model to Task Mapping

The agenda tasks defined by OpenConfigurator are mapped as described in Table 4.18, "Configuration Agenda Tasks". The mapping is basically derived from the process described in Section 4.5.1, "Configuration synthesis and evaluation".

Table 4.18. Configuration Agenda Tasks

Task	Decision	Generic Configuration Model Binding	Domain	Description
SpecifyComponentTask	No direct correspondence.	Component<C>	Domain<C>	Composite task to instantiate a component.
SpecifyComponentTypeTask	Component type decision (i.)	Component<C>	Domain <Class<? extends C>	Task to specify the type of a component.

Task	Decision	Generic Configuration Model Binding	Domain	Description
SelectComponentTask	Component customization decision (iv.A.)	Component<C>	Domain<C>	Task to select a component instance.
ConfigureComponentTask	Component customization decision (iv.A.)	Component<C>	Domain<C>	Composite task to instantiate a configurable component.
ConstructComponentTask	Component customization decision (iv.B.)	Component<C>	Domain<C>	Composite task to instantiate a constructible component.
SpecifyAttributeTask	Attribute quantity decision (v.) and attribute valuation decision (vi.)	Attribute<?, A>	Domain<A>	Task to specify the value of an attribute.
SpecifyPartTask	No direct correspondence.	Part<?, P>	Domain<P>	Composite task to specify the value of a part.
SpecifyPartQuantity-Task	Component quantity decision (ii.C.)	Part<?, P>	Domain <Integer>	Task to specify the quantity of a type.

The configuration decisions not mapped directly, are realized by additional constraints. For instance, whether a part is mandatory or optional is defined by the corresponding `@Required` respectively `@Optional` constraint annotation (see Section 4.4.5, “Constraint Modeling”).

Agenda Task Hierarchy

`CompositeTasks` reference arbitrary subtasks, effectively leading to tree-like *task hierarchy*. `OpenConfigurator` builds up of the configuration agenda, by recursively traversing the generic component model (using the visitor facility, see “Structural navigation support” in Section 4.4.1.1, “Responsibilities”) and applying the following behavior depending on the element's type:

Component processing. For each component found in the generic configuration model `OpenConfigurator` creates a corresponding `SpecifyComponentTask` in the agenda, which contains the following nested tasks:

- a single `SpecifyComponentTypeTask`, and
- if the specification method defined on the specified type is `@Selectable`, a single `SelectComponentTask`, or
- if the specification method defined on the specified type is `@Configurable`, a single `ConfigureComponentTask`, or
- if the specification method defined on the specified type is `@Constructible`, a single `ConstructComponentTask`

Attribute processing. For each variable attribute (`@Variable` or `@Parameter`) of a traversed `@Configurable` or `@Constructible` component, `OpenConfigurator` creates a `SpecifyAttributeTask` as a nested task of the corresponding `ConfigureComponentTask` respectively `ConstructComponentTask`.

Part processing. For each `@Configured` part of a traversed `@Configurable` or `@Constructible` component, `OpenConfigurator` creates a `SpecifyPartTask` as a nested task of the corresponding `ConfigureComponentTask` respectively `ConstructComponentTask`.

Depending on the processed part's cardinality, `OpenConfigurator` does:

- **Singular part.** For a singular part, a `SpecifyComponentTask` bound to the part's nested component is added to the `SpecifyPartTask`, that corresponds to the part.
- **Plural part.** For a plural part, a `SpecifyPartQuantityTask` bound to the part is added to the `SpecifyPartTask`. Moreover, for each nested component of the part, a corresponding `SpecifyComponentTask` is added to the `SpecifyPartTask`.

Figure 4.41, “Example Task Hierarchy” depicts an exemplary task hierarchy for the configuration domain model discussed in Section 4.5.2, “Example Configuration Procedure”.

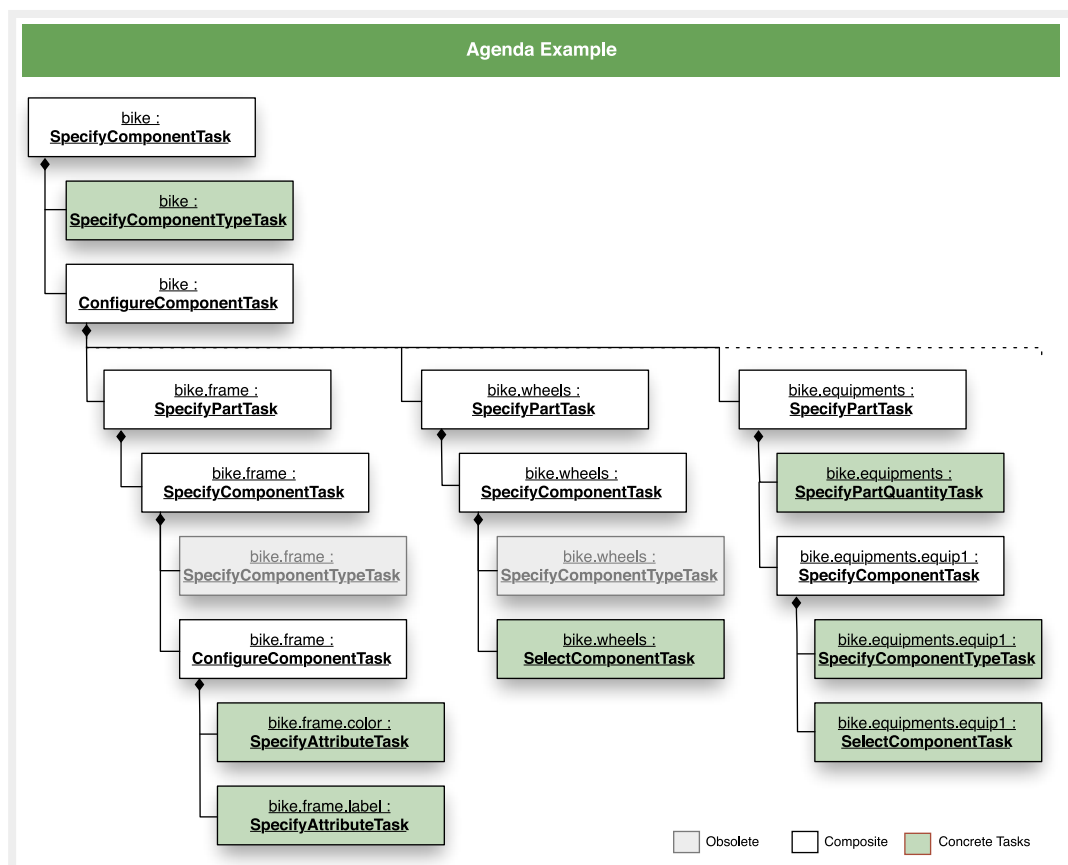


Figure 4.41. Example Task Hierarchy⁵⁶

The Agenda instance, containing the different Tasks can, for instance, be used by the configurator UI to navigate and control the configuration process.

⁵⁶Note that the object identifiers used in the example signify the model element to which the particular task is bound, which is why those names are not unique.

Task State Transitions

Importantly, the agenda reflects the state of the configuration: once all tasks have been completed, a valid instance of the configured product type has been specified (within the generic configuration model).

In general, a task is marked as *complete*, if the corresponding element of the generic configuration model is assigned with a value that doesn't violate any constraint.

A task may also become *obsolete*, that is, the user cannot take any further action related to the task. This is the case, when the task's associated domain contains only a single value and the task is non-optional. In this case, OpenConfigurator specifies the task automatically, by assigning the remaining value to the target element.

Moreover, a nested task becomes complete or obsolete, if its parent is completed respectively marked as obsolete.

We will discover, how the agenda is used concretely in Section 5.3.2.5, "Working with the Configuration Agenda". However, before we shift to the technical implementation of the OpenConfigurator framework in the next chapter, we'll shortly recapitulate this chapter again.

4.6. Summary

In this chapter, we introduced the OpenConfigurator approach for developing custom configurators. Particularly, we provided an overview of our methodology and explained our conceptualization in terms of its concrete modeling capabilities and the intended configuration procedure in detail. Let's shortly summarize the main facts again.

Within the first section, we provided a high-level overview of the OpenConfigurator approach. In general, it can be characterized as:

- model-based,
- object-oriented,
- universal and generic,
- declarative, and
- Java based.

Beyond these fundamental aspects, we identified the basic idea behind OpenConfigurator's way of realizing product configuration. The idea has been stated by three facts:

1. Customizable products are represented by object-oriented classes.
2. Custom product variants/configurations correspond to instances of these classes.
3. Consequently, the configuration process equals the instantiation process of a configurable type.

This idea makes the main task of the OpenConfigurator framework plausible: the framework must incrementally realize the instantiation of a product configuration on behalf of the user, stating configuration decisions. Thereby, configurable products are described in terms of custom, domain-specific Java class models. Meta-data attached to these domain models via Java annotations are interpreted by the framework at runtime and drive the configuration process.

We explained the targeted configurator development approach applied in practical implementation projects in detail and compared it to other configuration approaches. Most importantly, we stressed the fact, that in terms of OpenConfigurator, the developer solely needs to model the configurable product range and deploy the same to the generic configurator. The configurator acts as an execution environment for these domain models. It is capable

of interpreting the meta-data annotations and transforming the model automatically into a configuration process. The configuration process, in turn, is visualized by a generic user interface, which may be any a client ranging from desktop facing applications to mobile apps.

Finally, we explained the resulting advantages of our approach in the first section.

In the next section, we characterized our approach of modeling configurable product domains in general. We explained the role of the object-oriented concepts for product modeling and discussed the use of Java annotations for providing meta-data roughly.

The main subject of section three was the generic configuration model, which is a fundamental concept of the OpenConfigurator framework: it is the domain-independent representation of the configurable product at runtime. This model not only provides the "backbone" of the configuration process, but also accomplishes OpenConfigurator's promised extensibility and flexibility. Facets have been explained and mentioned as a key concept in this regard, too.

Then we concentrated on the concrete modeling capabilities defined within our conceptualization. In summary, our modeling language, that builds on the product modeling aspects identified in Section 3.1, "Product Models", encompasses the following main areas:

- **Definition of product structures.** With the help of **Java language constructs** (classes, fields and methods), the structure of a configurable product (or product family) is described.
- **Definition of product information.** The **@Product** and **similar annotations** are used to identify product related attributes in the decomposition structure of a domain model.
- **Definition of configuration behavior.** The customizable areas of the product, that must be specified by the customer during the configuration process, can be identified using **configuration related annotations**. With these annotations, effectively the possible set of *configuration decisions* is defined⁵⁷. Moreover, the configuration annotations state, *how* the configuration decisions can be made.
- **Definition of product variation.** In order to describe manufacturable variants of a product more precisely, the developer can back the domain model with additional product data. For this purpose, the **data related annotations** are employed. They define the *domain values* of attributes and parts, respectively specify the *available options for the configuration decisions*. The data may be embedded into the Java source code or may be externalized into a database.
- **Definition of product restrictions.** The choices, the user can take during product configuration, can be further restricted by adding **constraint annotations** to the model. This way, the configurator developer can realize manufacturing or business related restrictions imposed on the product range.

With the help of these concepts, sophisticated customizable products can be modeled, including both physical products and immaterial services. We will discover a concrete example in Chapter 6, *Evaluation and Validation*.

The last section of this chapter, focussed the process view again: we described, how the configurable product model is turned into a concrete configuration procedure. We examined this procedure in detail by walking through an elaborated example. This way we showed, how the configuration decisions identified in Section 3.1, "Product Models" are applied practically and how the transformation process described in Section 3.2.2.2, "The Configuration Process as a Transformation Process" is realized in terms of OpenConfigurator.

⁵⁷The possible set means, that by applying configuration annotations, all feasible configuration decisions are determined. However, it may be the case, that some configuration decisions are dismissed, due to the lack of option alternatives, which are effectively controlled with domain annotations and constraints.

In the next chapter, we will discover, how OpenConfigurator's runtime execution environment is implemented.

5

Technical Architecture and Implementation

In this chapter, we will take a look at OpenConfigurator's technical architecture and implementation. We will highlight the most important technologies, architectural concepts and APIs.

The previous chapter introduced OpenConfigurator's methodology in general and presented concrete concepts for modeling configurable products. In this part, we will focus the realization of the runtime *execution environment*, that is used to actually perform the configuration process based on the deployed product domain model.

5.1. Technologies

As stated in Section 4.1, “The OpenConfigurator Methodology”, OpenConfigurator is a purely Java based framework for implementing product configurators. While the modeling capabilities discussed in the previous chapter more or less exclusively build on core Java constructs (classes, interfaces, annotations, etc.), OpenConfigurator has been designed to interplay well with a number of other technologies. These technologies will be described in the following, highlighting the most relevant aspects related to our framework for each of them.

5.1.1. Java

Of course, first and foremost, the Java technology itself is relevant for the framework. We already mentioned the importance of Java's object-oriented nature in Section 4.2, “Modeling Approach”. In summary, the most relevant aspects of Java related to OpenConfigurator are:

Object-orientation. Java is a fully object-oriented (OO) language and implements all major features of the OO paradigm, including (singular) inheritance, polymorphism, abstraction, encapsulation, information hiding and more. Particularly, these concepts are important for the modeling of configuration knowledge in our object-oriented approach as described in detail Section 4.2, “Modeling Approach”.

Annotations and annotation processing. A main building block of our modeling approach is the specification of meta-data using Java *annotations* in order to express configura-

tion semantics. In fact, Java annotations, which have been introduced in Java version 5 (formerly 1.5), are heavily used within the OpenConfigurator framework and constitute a major technique of our conceptualization (see Section 4.2.3, “Providing Meta-Data with Java Annotations”). They are also applied in all of the technologies mentioned below in manifold ways.

Moreover, the *annotation processing tool* (APT) introduced in Java 6 is highly relevant: the APT library is an extension to the Java compiler, that has been created to perform compile-time processing of source code annotations. Applications for APT include code generation and source code validation. While not yet utilized within the OpenConfigurator framework, in the future, APT can be used to verify the correct utilization of source code annotations provided by our conceptualization. Correctness in this context doesn't solely include the *syntactical* usage of annotation, which is checked by the compiler out-of-the-box, but especially their *semantics*, which can only be verified by some APT extension.¹

Reflection. As most of the “magic” implemented by OpenConfigurator happens at runtime, the utilization of *reflection* is essential to the OpenConfigurator technology. Particularly, the built-up of the generic configuration model based on annotated JavaBean classes as well as the dynamic instantiation of the these beans during the configuration procedure strongly depend on Java's reflection API. Using this API, the model itself and the meta-data provided through annotations is extracted and introspected at runtime.²

Type safety and generics. OpenConfigurator tries its best to provide an easy to use, type-safe API for configuring JavaBean instances. In fact, most framework classes and interfaces make extensive use of generics in order to avoid unsafe type casts and to enable built-time source code validation using the Java compiler.³

5.1.2. Java Persistence API (JPA)

While OpenConfigurator in general is persistence provider agnostic⁴, it has been specifically designed to integrate well with the Java Persistence API for object-relational persistence purposes. As a well accepted standard in the Java community, JSR-317⁵, which currently features version 2.0, offers great opportunities when used in conjunction with OpenConfigurator. Most prominently, the following capabilities are important in regard to our framework:

Declarative persistence meta-data. Similar to our approach, the JPA specification primarily utilizes Java annotations to enrich JavaBean domain models with persistence related information. It allows to annotate POJOs with meta-data that dictates the persistence provider on how to map particular Beans to and from an SQL⁶ based relational data mode. Thereby, the persistence layer implementation is strongly simplified and simultaneously an abstraction from any concrete database technology or vendor is achieved.

Query facility. JPA offers strong querying capabilities for directly retrieving regular JavaBean objects from the database without any manual conversion. While primarily the string based interface to the query API is currently utilized by OpenConfigurator's data related query annotations (see Section 4.4.4.1, “Attribute Value Domain Definition”), also the type

¹For more information on annotations and APT, see <http://docs.oracle.com/javase/6/docs/technotes/guides/language/annotations.html> and <http://docs.oracle.com/javase/6/docs/technotes/guides/apt/GettingStarted.html>, last accessed June 22th, 2012.

²See <http://docs.oracle.com/javase/6/docs/technotes/guides/reflection/index.html>, last accessed June 22th, 2012.

³More details on generics can be found at <http://docs.oracle.com/javase/6/docs/technotes/guides/language/generics.html>, last accessed June 22th, 2012.

⁴The contract between OpenConfigurator and the underlying data provider is realized by the `DataProvider` service interface (see “Data provider” in Section 5.2.2, “Services”). While OpenConfigurator comes with an implementation of that service for JPA, another implementation, e.g., based on the Java Content Repository API defined by JSR-283 (see <http://jcp.org/en/jsr/detail?id=283>) could be plugged in easily.

⁵See <http://jcp.org/aboutjava/communityprocess/final/jsr317/index.html>, last accessed June 22th, 2012.

⁶Abbrev. Structured Query Language

safe criteria query API introduced in version 2 of the specification, may offer great potentials for future versions of the framework.

Persistence meta-data facility. Another novelty introduced in JPA 2 is the meta-model API, that allows other frameworks to retrieve meta information about persistent entities programmatically. That means, without having to introspect JPA annotations manually. While not currently employed by the OpenConfigurator framework, the persistence meta-model could be used to make certain configuration related annotations obsolete, following the DRY principle⁷. As the design of OpenConfigurator's meta-model facility has been greatly influenced by JPA's meta-model API, both models are expected to interplay very well with each other.

Again, JPA is currently considered the primary data provider for making product data available to configuration models. In particular, JPA is involved whenever domain query annotations (see Section 4.4.4.1, "Attribute Value Domain Definition"), are evaluated: queries specified in those annotations are directly passed over to the JPA query facility.

5.1.3. Bean Validation

The Bean Validation specification is one of the more recent Java standards. It has been specified as JSR-303⁸ within the Java community process. Bean Validation aims to provide a "unified way of declaring and defining constraints on an object model"⁹ and to validate these object models at runtime using a validation engine. In the context of the OpenConfigurator framework, the following characteristics of Bean Validation are important:

Declarative constraints. Bean Validation specifies an approach to define constraints on Java objects declaratively using annotations. In fact, the way constraints are defined in OpenConfigurator has been strongly inspired by the Bean Validation specification. The framework also tries to remain compatible with the specification where possible, allowing to re-use custom constraint annotations, that are defined in accordance with JSR-303, to be used as is for configuration purposes.

Constraint validation framework. The Bean Validation specification not only standardizes the way constraints and their corresponding validators can be defined, but also provides a unified API to validate objects against the stated constraints.

Constraint meta-data facility. Similar to the meta-data facility provided by JPA, also JSR-303 defines an API for the introspection of constraints defined on a particular model class.

In summary, OpenConfigurator uses Bean Validation for the definition of constraints and adapts the validation engine for verifying the consistency of object properties. However, as Bean Validation does not exactly match all requirements of our framework, sometimes custom extensions to the standard, especially related to its meta-data facility, have to be employed.

5.1.4. Contexts and Dependency Injection for Java EE (CDI)

Not directly related to the modeling of configuration knowledge, the Contexts and Dependency Injection (CDI) specification, defined under JSR-299¹⁰, forms a key technology for the

⁷For example, properties of domain model classes annotated with JPA's `@Id` annotation could be ignored by the framework automatically, since they are mostly irrelevant in the context of configuration. Currently, these attributes need to be annotated `@Ignored`, though.

⁸See <http://jcp.org/aboutJava/communityprocess/final/jsr303/index.html>

⁹Cited from <http://beanvalidation.org/>, last accessed June 22th, 2012.

¹⁰See <http://jcp.org/aboutJava/communityprocess/final/jsr299/index.html>, last accessed August 4th, 2012.

implementation of the OpenConfigurator framework. The following features are of special relevance for our purposes:

Dependency injection. CDI defines a programming model, that features a declarative dependency injection mechanism to realize "loose coupling with strong typing"¹¹. Dependency Injection (DI) frees a particular object from looking up other objects it depends on. Instead, the dependencies are declared using annotations (simply by applying the `@Inject` annotation on, for instance, a field member of the class) and the DI container is responsible for performing object lookup and instance injection. Moreover, the mechanism implemented in CDI also is concerned with the lifecycle of the injected objects by binding them to different contexts, invoking lifecycle methods and so on.

Effectively, dependency injection decouples objects, which has a number of advantages:

- Improved testability.
- Better extensibility.
- Simpler, more readable code.

The following code snippet shows how dependency injection and contextual management is applied using CDI:

```
@ApplicationScoped ❶
public interface ValidationProvider { ... }

public class JSR303ValidationProvider
    implements ValidationProvider { ... }

@SessionScoped ❷
public class ConfigurationSessionImpl
    implements ConfigurationSession
{
    @Inject ❸
    private ValidationProvider validationProvider;

    ... ValidationResult<T> validate(Configuration<T> configuration)
        throws ConfiguratorException
    {
        return validationProvider.validate(configuration);
    }
    ...
}
```

The `ConfigurationSession` class in the example above is a *session scoped* component ❷, which means, that during the lifetime of an HTTP¹² session only a single instance of this class exists. Similarly, `ValidationProvider` is defined as *application scoped* component ❶. Consequently, during the lifetime of the application only a single instance of that object is created and all clients share the same instance. When injecting the dependencies into the `ConfigurationSession` ❸, the CDI container resolves the `ValidationProvider` dependency to the `JSR303ValidationProvider` type and looks up an instance of that class within the context associated to the application scope.

Through the utilization of dependency injection, lots of boilerplate code can be avoided and in general a cleaner, simpler software architecture can be realized.

Event notification system. CDI also provides a type-safe *event notification system*, effectively implementing a publish/subscribe feature similar to the whiteboard pattern known in

¹¹Cited from <http://docs.jboss.org/weld/reference/latest/en-US/html/part-1.html>, last accessed June 22th, 2012.

¹²Abbrev. HyperText Transport Protocol

the OSGi¹³ world (see [OSGi_Alliance2004]). Special about CDI's event framework is the full decoupling of event producers and event subscribers, which don't even require a compile time dependency between each other. Solely, a reference to the event type itself (which in CDI corresponds to the *event payload*) is common to both. The emitted and observed events can be further characterized using *qualifier annotations*.

The following code listing shows an example of how the CDI event bus can be utilized:

```
public class Payload {} ❶

public class Sender
{
    @Inject @SomeQualified ❷
    private Event<Payload> event;

    public void emitEvent()
    {
        event.fire(new Payload()); ❸
    }
}

public class Recipient
{
    public void onEvent(@Observes @SomeQualified Payload event)
    { ... } ❹
}
```

In the example above, the `Payload` class acts as event object ❶. The `Sender` class injects a CDI provided `Event` object ❷, that represents the interface to the CDI event bus. The injection point is further substantiated using a custom qualifier annotation `@SomeQualified`. The sender emits an event by calling the `fire` method of the injected `Event` object, passing an instance of the `Payload` class as a parameter ❸. The recipient, in turn, declares the `onEvent` method as callback function for `Payload` events, qualified with `@SomeQualified`, by annotating its parameter with the CDI provided `@Observes` annotation ❹. Now, when `Sender` fires the event, the CDI container invokes the `onEvent` method, passing along the `Payload` object created by the sender.

Note that `Recipient` never references `Sender` directly and vice versa. In fact, `Recipient` observes *any* `Payload` event emitted by *any* sender that fired such an event (assuming their qualifiers match), which is a significant difference to the traditional observer pattern¹⁴. Here, the recipient registers itself with the sender explicitly. Due to this fact, we refer to the CDI provided event bus as *global event system*. Likewise, the latter one is sometimes referred to as *local event system*, as the sender maintains a reference to all recipients *locally*.

Other features. CDI implements various other features, that are quite useful to implement flexible, clean software infrastructures. These additional features include:

- **Interceptors.** Method invocation *interceptors* allow AOP¹⁵-like decoupling of technical concerns from the business logic¹⁶. Interceptors are declaratively applied on certain objects with *any* type.

A prominent use case for interceptors is the implementation of a security layer. In this context, a security interceptor, can guard method invocations by checking a user's permissions prior passing control to the actual, intercepted method¹⁷.

¹³Formerly Open Services Gateway initiative

¹⁴See http://en.wikipedia.org/wiki/Observer_pattern, last accessed June 22th, 2012.

¹⁵Abbrev. Aspect Oriented Programming

¹⁶See <http://docs.jboss.org/weld/reference/latest/en-US/html/interceptors.html>, last accessed June 22th, 2012.

¹⁷For an example security interceptor implementation based on CDI, refer to <http://www.warski.org/blog/2010/03/simple-security-interceptor-in-weldjsf2/>, last accessed August 4th, 2012.

- **Decorators.** With *decorators* CDI allows business concerns being compartmentalized¹⁸. Opposed to interceptors, decorators are implemented against a *specific* type. Therefore, they are aware of the objects they decorate with additional functionality.

A meaningful use case for decorators, also in the context of OpenConfigurator, is the realization of currency conversion functionality¹⁹. Strictly, currency conversion is not business logic, but rather a cross-cutting concern, required multiple areas of an application. Therefore, its implementation should be externalized from, for instance, price calculation methods and implemented as a decorator instead. Depending on the user's locale, the decorator automatically converts all prices into the corresponding currency.

- **Portable extensions.** With the help of *portable extensions*, the CDI programming model can be extended. This allows third party frameworks to deeply integrate with the CDI technology, disregarding of the concrete CDI implementation in use.

The OpenConfigurator framework implements a CDI extension, in order to integrate itself into the CDI infrastructure. This allows, for instance, to observe archive scanning during container startup. During this scanning process, OpenConfigurator discovers configurable types on the Java classpath.

OpenConfigurator primarily leverages CDI's dependency injection facility and its event infrastructure as global event bus. Both concepts are key corner stones of its implementation and influenced the overall design eminently. Concretely, CDI is considered the major driver for OpenConfigurator's built-in *flexibility*, *extensibility*, *maintainability* and *testability*, because of two important aspects:

Dependency injection. Every single framework implementation class obtains its dependencies via `@Inject` annotations, aka the CDI container. This means:

- the implementation classes are decoupled from their dependencies, relying solely on their interface without being concerned with their lifecycles (maintainability)
- the implementation classes or their dependencies can be easily swapped and replaced by other implementations, even depending on the deployment scenario (e.g., integration tests, enterprise versions, etc.) using built-in CDI features (flexibility, extensibility, testability)
- the implementation classes can be easily (unit) tested in isolation, passing *mock objects*²⁰ as their dependencies (testability)

Event infrastructure. Many managed framework objects (e.g., framework components like Configurator or ConfigurationSession, model objects like Component, Attribute or Part, etc. discussed in more detail below) have a well defined lifecycle, that is mirrored by CDI events. For example, when a new configuration session begins, OpenConfigurator emits an ConfigurationSession event²¹ qualified with the `@Initialization` lifecycle qualifier. This has the following advantages:

- the initialization process for complex instances can be spread across several classes, that observe the corresponding events (flexibility)
- the initialization process can be adapted without changing the instance's internal behavior (maintainability)
- plugins can hook into the initialization process, providing custom extensions to the framework at a various places (extensibility)

¹⁸See <http://docs.jboss.org/weld/reference/latest/en-US/html/decorators.html>, last accessed June 22th, 2012.

¹⁹Example derived from <http://www.theserverside.com/news/2240016831/Part-3-of-dependency-injection-in-Java-EE-6>, last accessed August 4th, 2012.

²⁰See http://en.wikipedia.org/wiki/Mock_object, last accessed June 23th, 2012.

²¹More precisely, `ConfigurationSessionImplementor`, the framework internal representation of a `ConfigurationSession`, is used as event payload. More information on implementors see Section 5.2, "Architectural Overview".

In fact, the dependency injection and event features provided by CDI enable a state-of-the-art, modern software architecture, which supports the important aspects of flexibility, maintainability, testability and extensibility out-of-the-box.

While CDI's additional features (interceptors, decorators, etc.) are powerful tools for framework and application developers, too, at the current stage OpenConfigurator only foresees their use in a future version, but doesn't employ them concretely yet. In principle, however, all framework objects can be intercepted or decorated using CDI features as their lifecycles are managed entirely by the CDI container.

Having introduced the main technologies used within our implementation and their relationship to the OpenConfigurator framework, we can now describe our framework's overall architecture in more detail.

5.2. Architectural Overview

In this section, we will provide details on OpenConfigurator's internal architecture. While not going into technical details showing concrete interfaces, we will give a high level overview of the main concepts realized. Figure 5.1, "Architecture Overview" serves as a map for this section, that shows the architecture's "big picture".

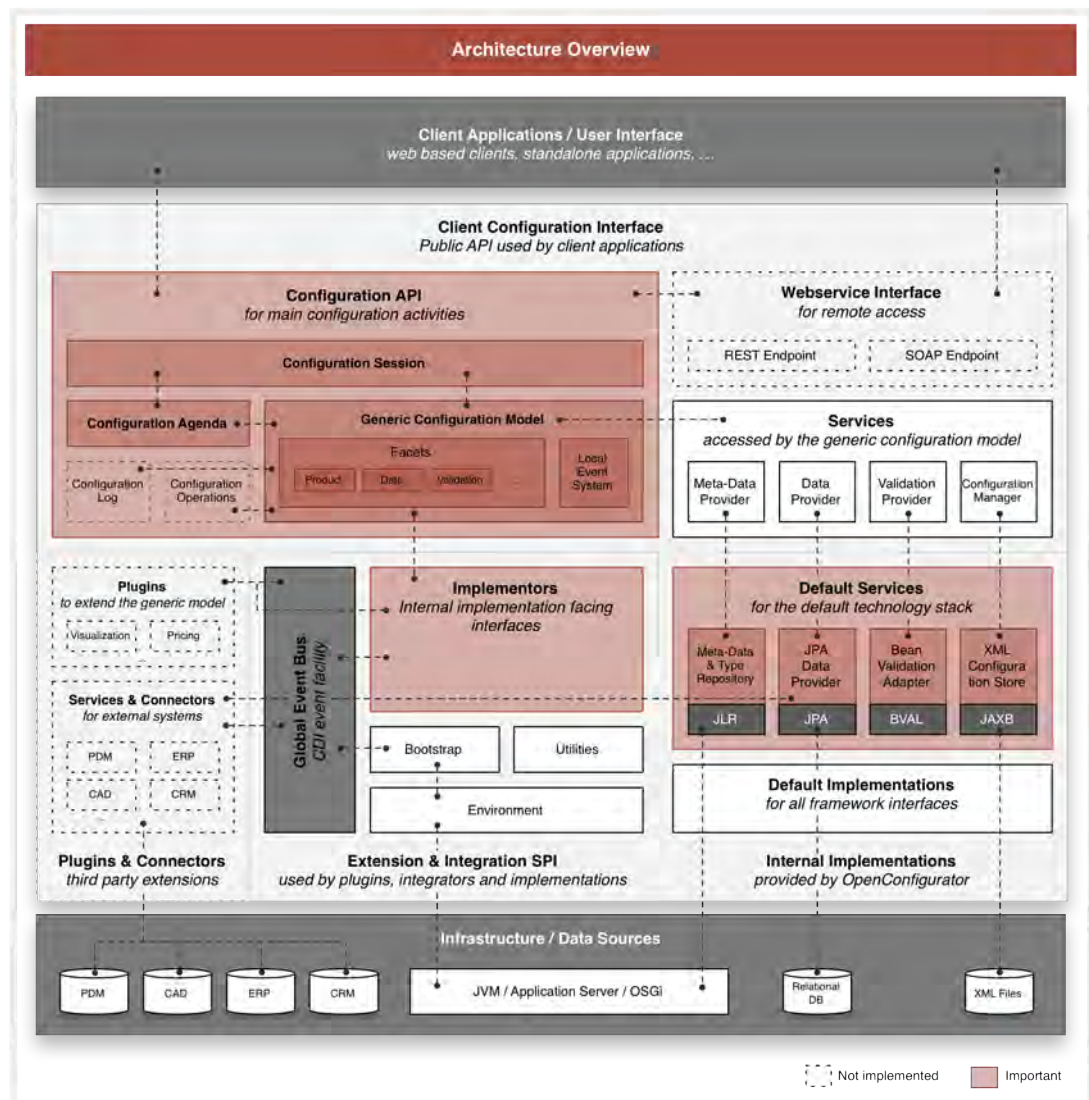


Figure 5.1. Architecture Overview

5.2.1. Multi-Tier Architecture

First of all, like many modern applications today, OpenConfigurator targets a **three-tier application architecture**²² to be used in a client-server scenario. In our context, the tiers can be described as follows:

Presentation/client tier. The presentation tier encapsulates the display of the configuration process within the client application. Typical application types targeted by OpenConfigurator are thin, web-based applications, that may either be deployed standalone (e.g., like the mobile web client based on the Vaadin technology²³ presented in Chapter 6, *Evaluation and Validation*) or which may be embedded within the company's corporate website. Alternatively, the OpenConfigurator framework can also be integrated into other Java applications (e.g., based on the Eclipse Rich Client Platform²⁴).

Client applications either access the framework's provided configuration interface (see below) directly using the *configuration API*, or indirectly using the *webservice interface*²⁵.

Logic tier. The OpenConfigurator framework primarily implements the logic tier required for configurator applications. This layer is further subdivided into four different parts:

- **Client Configuration Interface.** The *client configuration interface* forms the public API used by client applications to configure JavaBean instances using OpenConfigurator. It comprises the *configuration API* for in-JVM²⁶ access, a *webservice interface* for remote access and a *services* sublayer, which is accessed by the configuration API.
- **Extension and Integration SPI.** The *extension and integration SPI*²⁷ is primarily relevant for framework implementors, plugin developers or application integrators. The SPI provides access to the OpenConfigurator internals and is used to extend the framework or integrate it with other systems.
- **Internal Implementations.** The *internal implementations* part of OpenConfigurator contains default implementations for all interfaces defined by the framework. Particularly, it contains default implementations for the services layer.
- **Plugins & Connectors.** Finally, OpenConfigurator is designed to comprise a sublayer featuring *plugins and connectors*²⁸, which is meant to be filled by third party implementors, that want extend the framework's functionality in certain regards.

Data tier. Like any other modern business application today, OpenConfigurator employs a dedicated *data tier* for the retrieval/storage of data from/to external systems. In terms of OpenConfigurator, relevant external systems include relational databases, as primary data source for product data, and business systems such as PDM, ERP, CRM and CAD software. We also count the underlying Java platform infrastructure as provider for certain resources, such as type information or Java classpath resources.

The three-tier application architecture builds the foundation of OpenConfigurator's overall implementation. In the following sections, we will take a closer look at the logic tier components provided by the framework.

²²See http://en.wikipedia.org/wiki/Multitier_architecture, last accessed June 25th, 2012.

²³See <http://vaadin.com/>, last accessed June 25th, 2012.

²⁴See http://wiki.eclipse.org/index.php/Rich_Client_Platform, last June 25th, 2012.

²⁵Note: the webservice interface is not yet implemented.

²⁶Abbrev. Java Virtual Machine

²⁷Abbrev. Service Provider Interface

²⁸Note: currently no plugins or connectors are implemented.

5.2.2. Client Configuration Interface

The most important interface to the OpenConfigurator framework is the *client configuration interface*, which in turn features the configuration API and a couple of services as main building blocks.

We will provide examples on how to use the different concepts described here in Section 5.3, “API Usage: Working with OpenConfigurator”.

Configuration API

The *configuration API* comprises a set of Java interfaces, that allow client applications to interact with the configurator framework in order to create concrete product configurations programmatically. An instance of the `Configurator` class is the main entry point to the configuration API. The API knows the following main concepts:

Configuration Session. The *configuration session* (represented by the Java interface `ConfigurationSession`) encapsulates the runtime state information of a single configuration process. Whenever a user starts the configuration of a new product or reloads an existing configuration, a corresponding session must be established first. That means, a configuration session wraps an *active* configuration instance at runtime. While configuration instances are mostly persistent entities within the system, whose lifetime may be unlimited, the configuration session's lifecycle is always bound to the HTTP session or conversation scope (if available) within a Java EE web application context.

The `ConfigurationSession` interface provides access to the configuration agenda and the generic configuration model (see below). It can be obtained from the `Configurator` interface or via dependency injection.

A `ConfigurationSession` is a managed object, internally provided by the `ConfigurationSessionImplementor` interface, for which OpenConfigurator emits lifecycle events (`@Initialization`, `@Finalization`) over the CDI event bus.

Configuration Agenda. The configuration session holds an instance of `ConfigurationAgenda`. The *configuration agenda* encapsulates process related information by tracking the specification state for each component, attribute or part of the configured instance (see also Section 4.5.3, “Configuration Agenda”).

Furthermore, the configuration agenda provides a *task based view* to the configuration process: it allows the client to request, whether and which decisions need to be taken in order to complete the configuration process. Also, validation errors are signaled in the configuration agenda. In short, the configuration agenda can be seen as a high level, task based API to perform product configuration.

Generic Configuration Model. As discussed in Section 4.3, “The Generic Configuration Model”, the *generic configuration model* is “the heart” of the OpenConfigurator framework. It maintains the complete state of a configuration instance and provides a low level API to manipulate managed instances. Moreover, it centralizes and aggregates all relevant information related to a particular configuration and its components within a single, consistent component model.

The generic configuration model is represented by the parameterized interfaces `Configuration`, `Component`, `Attribute` and `Part` (see Section 4.4.1, “Structure Modeling”). The `Configuration` object provides access to the *root component* of a configuration model and can be obtained from the session interface. A `Component<T>` implementation manages an instance of type `T`, that is, an object of the configured type. Component model elements are managed objects, internally represented by the interfaces `ConfigurationImplementor`, `ComponentImplementor`, `AttributeImplementor` and `PartImplementor`. For these imple-

mentors OpenConfigurator emits lifecycle events (@Preparation, @Initialization, @Finalization) over the CDI event bus. Internally, these model objects are constructed using a `ModelFactory`, which implements the *factory method* design pattern²⁹.

With the help of *facets*, different views/perspectives to one and the same component can be obtained. As discussed in Section 4.3.3, “Facets”, OpenConfigurator provides some standard facets out-of-the-box, including the *product*, *data* and *validation* facet³⁰. Basically, a facet operates solely on the component data, but may also access external systems. For instance, a visualization facet could be implemented, that accesses a CAD system for rendering the product display. Facets are not required to implement any pre-defined interface and can be registered by third-party extensions at will (see Section 5.2.5, “Plugins and Connectors” and Section 5.4, “SPI Usage: Extending and Integrating OpenConfigurator”).

As already mentioned in Section 4.4.1, “Structure Modeling”, the generic configuration model features a (*local*) *event system*. That is, components, attributes and parts support the registration of observers for certain events. These events are emitted by the particular elements upon state changes, for example, when an attribute's value changes or a component's type is altered. Typically, this event facility is used by other subsystems, such as the user interface, the configuration log (see below) or by facets and extensions to trigger some processing. Opposed to the *global event infrastructure* provided by CDI (see Section 5.2.4, “Global Event Bus”), event listeners must be registered explicitly with the particular elements. Usually, listener registration happens at element initialization time, which in turn can be observed by listening to the global @Initialization event.

Configuration Log³¹ Within the *configuration log*, the configurator maintains a chronologically ordered list of actions performed during the configuration process. The log can be used to implement a truly explanatory component and is a prerequisite for sophisticated conflict resolution strategies, that involve decision backtracking. It tracks all option specifications, that a user manually performed or which have been automatically performed by the system, including value, type and cardinality changes. The configuration log entries therefore correspond to instances of the types `ValueChange`, `TypeChange`, and so on.

Configuration Operations³² While the generic configuration model allows direct, synchronous manipulation of the configuration state, the *configuration operations* API supports a command based³³ interface to the system. Using configuration commands, such as `SetValue`, `SetType` etc. bulk operations, including numerous actions at once, can be executed in an asynchronous manner. This is particularly useful to implement backtracking algorithms or the webservice interfaces (see Section 5.2.2, “Webservice Interface”).

Services

During configuration, the generic configuration model accesses functionality provided by dedicated service facilities. At the current stage, these *services* include³⁴:

Meta-Data Provider. The `MetadataProvider` interface designates the service, that is used to query component related *meta information*, that is, the `ComponentDescriptor`, `AttributeDescriptors` and `PartDescriptors` for a particular Java type of the domain model (see Section 4.3.1, “Elements of the Generic Configuration Model and Meta Model”).

Data Provider. The `DataProvider` service interface is accessed by the data facet in order to access the underlying data model during the configuration process. Specifically, it is used

²⁹See http://en.wikipedia.org/wiki/Factory_method_pattern, last accessed June 26th, 2012.

³⁰Note that the *UI facet* is not currently implemented.

³¹The *configuration log* is not currently implemented, but will be added in a future version of OpenConfigurator.

³²The *configuration operations* interface is not currently implemented, but will be added in a future version of OpenConfigurator.

³³That is, based on the *command* design pattern, see http://en.wikipedia.org/wiki/Command_pattern, last accessed June 26th, 2012.

³⁴Note that we will describe their default implementations later on, in Section 5.2.3, “Internal Implementations”.

to resolve tuple domains, when the `@Domain.Query` annotation is employed to model the configurable type (see Section 4.4.4.1, “Attribute Value Domain Definition”).

Validation Provider. The `ValidationProvider`, in turn, is involved during constraint checking, which is performed by the validation facet. It encapsulates access to the validation subsystem.

Configuration Manager. Another important service is the `ConfigurationManager`, which provides functionality to load and store `Configuration` instances. Also, the *configuration manager* is used to submit configurations for further processing by other subsystems upon completion of the specification process. During submission, the `@Completed Configuration` event is emitted over the CDI event bus. This event can be observed by third party extensions to feed the finalized configuration into other external enterprise systems, such as PDM, ERP or CRM solutions. Hence, the configuration manager forms the central hub between the configurator and other systems.

Webservice Interface

OpenConfigurator's architecture is designed to be accompanied by a *webservice interface* for remote client communication. The webservice can be implemented as REST³⁵ endpoint or as a SOAP³⁶ based interface. Although the complex state loaded into memory during the configuration process is perfectly addressed using HTTP³⁷-sessions, which apparently violate the statelessness axiom postulated by the REST architectural style, we argue that a REST-like approach based on resources and a uniform interface would be a reasonable complement to the OpenConfigurator framework. In general, we see OpenConfigurator's service oriented, loosely coupled architecture well-suited to be published as a SOAP based webservice endpoint in a future release.

5.2.3. Internal Implementations

While the *client configuration interface* explained in the previous section, is a public interface accessed by client applications directly, the framework also contains a number of internal classes hidden to clients. These are discussed in the following.

Default Implementations

OpenConfigurator provides *default implementations* for all framework provided interfaces. These classes implement the basic behavior of the framework and can be seen as the “flesh” between the architectural skeleton. As the system obtains all instances from the CDI container in order to support dependency injection, any concrete implementation can be replaced using CDI's *alternatives* mechanism depending on the deployment scenario (see [JSR2992009, p. 17]).

Default Service Implementations

More importantly though, OpenConfigurator provides *default service implementations* that integrate major standards from the Java EE³⁸ space, namely JPA, Bean Validation and JAXB. These default service implementations include:

Java Reflection based Meta-Data Provider. OpenConfigurator's primary source for configuration meta-data are annotations added to the custom Java domain model. Therefore, the

³⁵Abbrev. *RE*presentational State Transfer, for more information on REST as an architectural style see [Fielding2000, pp. 76]

³⁶Abbrev. Simple Object Access Protocol

³⁷Abbrev. *HyperText* Transport Protocol

³⁸Abbrev. *Java* Enterprise Edition

framework's default `MetadataProvider` implementation uses Java's reflection features, to introspect the domain model types and to subsequently extract configuration related meta-data from them.

A future extension to the annotation based meta-data discovered by reflection could be an XML based meta-data layer. Optional XML descriptor files are still considered a common alternative to Java annotations in many Java standards today.

Additionally, `OpenConfigurator` features a `TypeRepository` implementation, that can be requested to discover available subtypes of a certain type. The discovery of a type's full type closure including its subtypes is a feature, that is not provided by the Java virtual machine out-of-the-box.

JPA based Data Provider. By default, the `OpenConfigurator` framework assumes a Java Persistence API based backend to retrieve configuration relevant product information from. Particularly, JPA's JPQL³⁹ is the primary query language assumed for specifying component tuples, respectively attribute domain values using the `@Domain.Query` annotation. The query statements provided by these annotations are evaluated and executed by the JPA data provider. Currently, `OpenConfigurator` uses JPA 2.0.

Bean Validation based Validation Provider. As mentioned earlier, `OpenConfigurator`'s constraint definition approach is compatible with the one provided by the Bean Validation specification. In fact, `OpenConfigurator` relies on the JSR-303 provided infrastructure for the validation of constraints. Currently, `OpenConfigurator` is implemented against Bean Validation 1.0.

JAXB based Configuration Manager. `OpenConfigurator`'s default `ConfigurationManager` implementation is capable of persisting complete or partial configurations using a *generic XML format*. Additionally, if the underlying JavaBean domain model contains JAXB annotations, that specify its mapping to XML, a completed configuration can be exported using the so called *domain specific XML format*. Both XML serialization mechanisms rely on the JAXB 2.2.

5.2.4. Extension and Integration SPI

Another major part of `OpenConfigurator`'s architecture is the *extension and integration service provider interface (SPI)*, which is used by the internal implementation, plugins and integrators to realize and extend the system or integrate it in different deployment scenarios.

Implementors

The central elements of the SPI are the so called *implementors*. An implementor represents the implementation facing interface of a particular framework type. It contains methods, internally required by the framework to realize the desired behavior. Most importantly, an implementor decorates a framework interface with initializer methods. With the help of these initializer methods, the instances' construction process can be flexibly designed. Figure 5.2, "Implementors" visualizes the role of implementors within the framework using the `Component` interface as an example.

³⁹Abbrev. Java Persistence Query Language

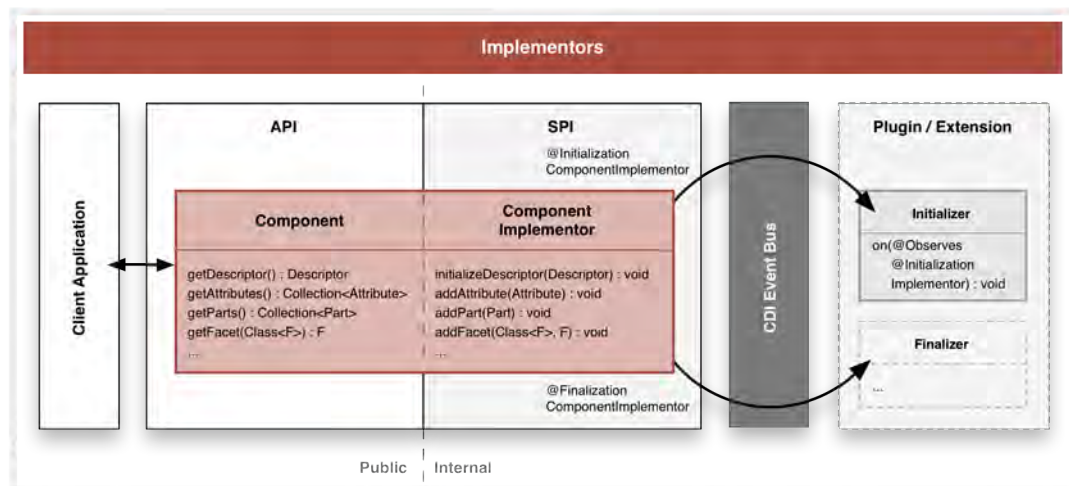


Figure 5.2. Implementors

Using this architecture, client applications can solely access the API provided **Component** type, while internal services see a more sophisticated representation. Particularly, internal implementations are able to modify a component's state during initialization. In contrast, the public API features read-only methods only, making the component itself immutable after construction.

Technically, the implementation utilizes the global CDI event bus (see Section 5.2.4, “Global Event Bus”), by emitting an `@Initialization` event, using the **ComponentImplementor** object as event object/payload. **Initializer** types may observe this event and add, for instance, an additional facet to the partially initialized component through the SPI provided **ComponentImplementor** facility.

In summary, the implementors not only provide a “back door” for OpenConfigurator's implementations to modify the internal state of the model elements, but also allow plugins and extension to influence their initialization (and finalization) behavior. The implementor pattern is applied to numerous concepts throughout the framework, which can profit from these capabilities.

Global Event Bus

Another integral part of the SPI is the *global event bus* facilitated by CDI's event system (see Section 5.1.4, “Contexts and Dependency Injection for Java EE (CDI)”). The global event bus is a key enabler for OpenConfigurator's extensibility and flexibility, as third-party plugins can observe the various events fired throughout the configuration process. Table 5.1, “Global Events Emitted via the CDI Event Bus” provides an overview of the global events fired:

Table 5.1. Global Events Emitted via the CDI Event Bus

Event / Payload	Qualifiers	Description
Framework events		
BootstrapImplementor		Emitted during configurator startup, as an environment neutral event. Allows integrate with the bootstrap process.
ComponentDescriptorImplementor, AttributeDescriptorImplementor, PartDescriptorImplementor	@Preparation, @Initialization	Component meta model element lifecycle events.
ConfiguratorImplementor	@Initialization, @Finalization	Configurator lifecycle events.
ConfigurationSessionImplementor	@Preparation, @Initialization, @Finalization	Configuration session lifecycle events.
ComponentImplementor, AttributeImplementor, PartImplementor	@Preparation, @Initialization, @Finalization	Component model element lifecycle events.
Model events		
Component.Event, Attribute.Event, Part.Event		Local generic configuration model events passed over to the global event bus.
Attribute, Part	@Added, @Removed	Events emitted upon type changes.
Configuration events		
Configuration	@Created, @Stored, @Loaded, @Completed, @Validated, @Submitted	Configuration lifecycle and status events.

We can divide these events into the following categories:

Framework events. Signal the lifecycle of core framework objects. They are primarily used internally by the implementation. The bootstrap related event can be used to integrate with the bootstrap process (see Section 5.2.4, “Bootstrap, Environment and Utilities”), for example, to register additional configurable types.

Model events. Are fired during the lifetime of an active configuration. They correspond one-to-one to the events emitted within the local event system of the generic configuration model. For a complete list of fired events, see the individual subsections in Section 4.4.1, “Structure Modeling”. They’re primarily interesting for client applications, respectively their user interface.

Configuration events. Notify about important process relevant actions, such as the submission of a configuration. These events may be observed, for instance, by connector plugins to queue finalized configuration instances for further processing by other external systems (see Section 5.2.5, “Plugins and Connectors”).

Bootstrap, Environment and Utilities

As mentioned earlier, OpenConfigurator is primarily meant to be used in Java EE based web applications. Specifically, it targets *Java EE 6 web profile* environments, where the Servlet API

3.0, CDI 1.0, JPA 2.0, Bean Validation 1.0 and JAXB 2.2 are available by default. Based on this technology stack, OpenConfigurator implements a default bootstrap strategy, that is triggered automatically when the framework's JAR file is deployed as part of a web application. During bootstrap, OpenConfigurator discovers configurable types as well as any extensions to the framework and finally instantiates a singleton Configurator instance, which itself acts as a factory for ConfigurationSessions.

For the purpose of loading resources and the retrieval of other context specific properties, OpenConfigurator provides an injectable Environment instance that abstracts from the underlying deployment scenario. In a future version of the framework their might exist different Environment implementations, for example, abstractions for:

- *Java EE web application environments* (default), also referred to as WAR⁴⁰ deployments
- *Java SE environments*
- *OSGi environments*

The aforementioned environments are quite different related to class and resource loading behavior, issues that may be addressed and unified by the diverse environment implementations.

Finally, OpenConfigurator provides a number of utilities for framework developers. For instance, it provides a facility for easier introspection of annotated elements, offers an extended type discovery API and numerous other tools frequently required by third-party libraries.

5.2.5. Plugins and Connectors

In the previous section, we've shown various mechanisms and entry-points, that allow to modify, extend or complement the framework's features. Notably, the global event bus and the implementors facility can be mentioned in this context.

In this regard, typical additions to the framework include:

Plugins. Provide extensions to the generic configuration model in terms of *facets*. For example, a meaningful extension to the configuration model could be a *visualization* facet, which is capable of rendering sophisticated graphics for visualizing the configuration. Other reasonable plugins include a detailed price calculation mechanism to be added as a dedicated facet to the component model. The addition of a facet can be realized by simply registering it within a callback method for the @Initialization ComponentImplementor event, using the ComponentImplementor.addFacet(Class<F>, F) method.

Connectors. Allow connecting OpenConfigurator to various other backend systems, including PDM, ERP, CRM and CAD applications. As an example, a submitted configuration can be passed to the ERP and CRM systems to schedule them for further processing by sales employees. This could be achieved by writing a class, that provides a callback method observing the global @Submitted Configuration event.

Using these extension mechanisms, the OpenConfigurator framework with its compact core component model could be applied in manifold usage scenarios, and is not limited to configuration use cases only. The iPad application client introduced in Chapter 6, *Evaluation and Validation*, for instance, features a (yet basic) catalog like product browser based on the generic component model and the product facet.

Having discussed the most important architectural aspects of the OpenConfigurator framework, in the next sections, we are going to demonstrate how the provided API and SPI are used.

⁴⁰ Abbrev. Web Archive

5.3. API Usage: Working with OpenConfigurator

In the previous section, we gave an overview of the global architecture of the OpenConfigurator framework. In this section, we want to show, how the public API provided by the framework can be used (on code level) to perform configuration tasks.

The process described in this section, involves three steps:

1. Bootstrapping the configurator framework and obtaining a configuration session
2. Performing configuration
3. Submitting the configuration and further processing of the configuration result

We will discuss these steps one-by-one in the following.

5.3.1. Bootstrapping and Starting a Configuration Session

Consider the following scenario: imagine we're writing a JUnit⁴¹ based integration test for verifying the behavior of our custom Bike configurator introduced in Chapter 6, *Evaluation and Validation* (for the full domain model refer to Appendix B, *Example Domain Model: Bike*). In order to gain the power of CDI within JUnit, we're utilizing the Arquillian⁴² test framework that supports injection into test cases⁴³. This makes JUnit a *managed environment*, where OpenConfigurator (if available on the classpath) can be accessed simply via dependency injection.

Bootstrapping OpenConfigurator in a Managed Class

As stated above, when using OpenConfigurator within a class that supports dependency injection (like the Arquillian run JUnit test case used in the following, a Servlet 3.0 class or any other CDI bean), bootstrapping OpenConfigurator is completely transparent:

```
@RunWith(Arquillian.class)
public class ConfiguratorIntegrationTest
{
    @Inject
    private Configurator configurator; ❶

    @Test
    public void testConfigurator()
    {
        // do something... ❷
    }
}
```

Obtaining a reference to the entry point of the OpenConfigurator framework is simply done by injecting a `Configurator` into your managed class ❶. OpenConfigurator takes care of bootstrapping its infrastructure internally. After initialization, the configurator is ready for operation ❷.

As part of the bootstrap process, OpenConfigurator emits global events with the following payloads (in the given sequence):

1. `BootstrapImplementor`
2. `@Initialization ConfiguratorImplementor`

⁴¹See <http://junit.org/>, last accessed June 27th, 2012.

⁴²See <http://www.jboss.org/arquillian>, last accessed June 27th, 2012.

⁴³The technology stack comprising JUnit and Arquillian is used for any integration test in OpenConfigurator.

Bootstrapping OpenConfigurator in a Non-Managed Class

If your target class for some reason is not a CDI injection aware type, you can manually bootstrap the OpenConfigurator framework:

```
public class ConfiguratorStandaloneTest
{
    @Test
    public void testConfigurator()
    {
        Configurator configurator =
            new OpenConfigurator().initialize(); ❶
        // do something...
    }
}
```

You do this by creating a new `OpenConfigurator` instance and calling the `initialize` method ❶.

For the rest of this work, we assume injection is available and the code given in the program listings is run as part of a managed class.

Obtaining a Configuration Session

The next thing to do is obtaining a `ConfigurationSession` object, that wraps a specific Configuration instance at runtime:

```
@Inject
private ConfigurationSession<Bike> session; ❶

ConfigurationSession<Bike> session =
    configurator.createSession(Bike.class); ❷

// start configuration
... ❸

session.load(id); ❹

Bike bike = new MountainBike();
session.restore(bike); ❺

// continue with configuration
...
```

Again, there are two ways to obtain a `ConfigurationSession` instance. First, one can use injection ❶. Alternatively, a new, typed session instance can be created using a `Configurator` instance's `createSession` factory method ❷.

From this point, configuration can start (see Section 5.3.2, “Performing Configuration”). One can either start with a blank configuration ❸, load a previously persisted configuration by ID ❹ or restore a session for a given JavaBean instance ❺.

During session creation, `OpenConfigurator` instantiates the generic configuration model and initializes a configuration agenda accordingly. Additionally, it emits the following events (in the given sequence):

1. @Preparation `ConfigurationSessionImplementor`
2. @Preparation `Component / Attribute / Part`, for each element of the model
3. @Initialization `Component / Attribute / Part`, for each element of the model
4. @Initialization `ConfigurationSessionImplementor`

For the full documentation (JavaDoc) of the `ConfigurationSession` and related interfaces, refer to Appendix C, *OpenConfigurator API/SPI*.

5.3.2. Performing Configuration

Having bootstrapped the configurator and obtained a configuration session instance, the actual specification activities can be performed using the OpenConfigurator API. Configuring involves the following activities:

- Navigating through the generic configuration model
- Accessing meta-data and state information
- Specifying options
- Working with facets
- Working with the configuration agenda

While the first two activities can be considered low level operations supported by the core generic model, the configuration agenda builds a high level API to configuration, which itself uses facets to operate on the underlying components.

5.3.2.1. Navigating through the Generic Configuration Model

The configuration API offers various methods to navigate through the configuration model, as shown in the following listing:

```
// retrieve the root component
Configuration<Bike> configuration = session.getConfiguration(); ❶
Component<Bike> bike = configuration.getRootComponent(); ❷

// get all attributes
Collection<Attribute<Bike,?>> attributes = bike.getAttributes(); ❸

// get a specific attribute
Attribute<Bike, Integer> weight = bike.getAttribute("weight"); ❹
...

// get all parts
Collection<Part<Bike,?>> parts = bike.getParts(); ❺

// get a specific part
Part<Bike, Frame> bikeFrame = bike.getPart("frame"); ❻

// access the component associated to a part
Component<Frame> frame = ((SingularPart<Bike, Frame>) bikeFrame)
    .getPartComponent(); ❼

// alternatively, use a shortcut method of the Component interface
Component<Frame> frame = bike.getPartComponent("frame"); ❽
...

assertTrue(frame.getParent().equals(bike)); ❾
```

First of all, the `Configuration` instance is obtained from the current session ❶. As stated earlier, the `Configuration` object makes up the persistent part of a configuration, while `ConfigurationSession` and `Agenda` (see Section 5.3.2.5, “Working with the Configuration Agenda”) are solely created at runtime for an active configuration. A `Configuration` contains a single root element, that can be retrieved using the `getRootComponent` method ❷.

The `Component` interface provides a methods to access all attributes ❸ or a specific attribute identified by its name ❹, which equals the name of the corresponding JavaBean member.

Similar to the attribute related methods, the Component interface contains methods to retrieve a collection of all parts ❶ or a particular part identified by name ❷. In order to get the nested component referenced by a particular part, two methods can be used:

- either the SingularPart interface's `getPartComponent` method (respectively `getPartComponents` for `PluralPart` instances), which requires casting the part to its more specific type, or
- the shortcut method `getPartComponent` (respectively `getPartComponents` for plural parts) provided by the Component interface, that avoids the type cast.

The parent component of a part's inner component can be retrieved using the `getParent` method.

Note that all returned objects, respectively their types (`Configuration<C>`, `Component<C>`, `Attribute<C, A>` and `Part<C, P>`), are parameterized, generic types. This enables typed access to the configuration API without the need to perform casts.

For the full documentation (JavaDoc) of `Configuration`, `Component`, `Attribute` and `Part` as well as their related interfaces, refer to Appendix C, *OpenConfigurator API/SPI*.

5.3.2.2. Accessing Meta-Data and State Information

The generic configuration model element classes additionally offer various methods to identify the current state of a particular element. Although we only show the methods of the Component interface, most of them equally apply to `Attribute` and `Part` as well:

```
// get the descriptor of an element
ComponentDescriptor<C> elementDescriptor = element.getMetadata(); ❶

// whether this element or any of its sub-elements
// has been specified in some way by the user
boolean specified = element.isSpecified(); ❷

// get the current JavaBean type of the component
Class<? extends C> currentType = element.getType(); ❸

// whether the current type is abstract
boolean notInstantiable = element.isAbstract(); ❹

// whether the element has been instantiated,
// aka the underlying bean has a non-null value
boolean instantiated = element.isInstantiated(); ❺
```

With the help of the element's `getMetadata` method, the meta-model object containing descriptive information about the same can be accessed ❶. The `isSpecified` flag indicates, whether some attribute, part or the value of a component has been explicitly defined by the user ❷. The methods `getType` ❸ and `isAbstract` ❹ inform about a component's current type and whether this type is abstract or not. If the type of a component is non-abstract, the JavaBean instance managed internally by the component is automatically instantiated. The `isInstantiated` method indicates whether the instantiation already happened ❺ (see Section 4.4.1.1, “Responsibilities” and Section 4.5.2, “Example Configuration Procedure”).

5.3.2.3. Specifying Options

Probably the most important methods of the configuration API are those, that are used to specify configuration options. With their help, user decisions are incorporated into the generic component model. In the following program listing we show, how the decisions described

in Section 4.5.2, “Example Configuration Procedure” are captured with the configuration API:

```
// specify the type of a component
bike.setType(ElectroBike.class); ❶

// specify attribute values
Component<Frame> frame = bike.getPartComponent("frame");
Attribute<Frame, Color> color = frame.getAttribute("color");
color.setValue(Color.SILVER); ❷
frame.setAttribute("label", "MyBike"); ❸

// specify part values
Wheels slickTyres = ... // get slick tyres instance ...
bike.getPart("wheels").setValue(slickTyres); ❹

PluralPart<Bike, Equipment> bikeEquipment =
    (PluralPart<Bike, Equipment>) bike.getPart("equipment");
bikeEquipment.setCardinality(1); ❺

Component<Equipment> equip1 =
    ((Indexed<Component<Equipment>>) bikeEquipment).get(1);
equip1.setType(Lock.class);
Lock lock = ... // get lock instance
equip1.setValue(lock);

Component<Motor> motor = bike.getPartComponent("motor"); ❻
motor.setAttributeValue("fuelType", FuelType.DIESEL);
motor.setAttributeValue("power", 200);
```

With the `setType` method of the `Component` interface, the type of the underlying `JavaBean` is specified ❶. The type literal passed as method argument must be a subtype of the component's base type `C`, that is, a subclass of `Bike` in this example. Again, once a component is specified to have a non-abstract type, the component model instantiates the underlying `JavaBean`. Thus, in the preceding example, an instance of `ElectroBike` is created immediately after `bike.setType(ElectroBike.class)` is invoked.

Next, the two attributes "color" and "label" of the frame component are specified using `Attribute`'s `setValue` method ❷ and the shortcut method `setAttributeValue` of the `Component` interface ❸.

Similar to attributes, part values can be specified in multiple ways. The one using the `Part` interface's `setValue` method is shown in ❹. A multi-component part's quantity can be specified using the `setCardinality` method of the `PluralPart` interface, which is a subtype of `Part` ❺. Finally, the code listing above shows, how a subcomponent can be specified on attribute level ❻.

Internally, `OpenConfigurator` applies the given changes to the managed `JavaBean` instance, respectively delays the changes until the domain object has been instantiated. Furthermore, it emits the model events described in Section 4.4.1, “Structure Modeling” on both the local and the global event bus.

5.3.2.4. Working with Facets

Until now, we only considered functionality provided by the core component model API. Particularly for configurator applications, however, additional information provided by the different facets is indispensable.

These information are accessible through the different facets (see Section 4.3.3, “Facets”). We're going to describe their use next.

The Product Facet

The Product facet provides a product perspective onto a particular component. The following program listing shows an exemplary use of some methods of the Product facet:

```
Component<Bike> bike = ...
Product<Bike> bikeProduct = bike.getFacet(Product.class); ❶

String description = bikeProduct.getDescription(); ❷
Collection<Attribute<Bike, ?>> characteristicAttributes =
    bikeProduct.getAttributes(Attribute.IDENTIFICATION); ❸
```

First, a reference to the Product facet is obtained ❶. The getDescription method returns the value of the component attribute annotated with @Product.Description ❷. Here, the advantage of a faceted view on a component in relation to developer ease-of-use becomes obvious: the API client must not manually filter all attributes for a @Product.Description annotation, but instead simply invokes a method of the Product interface, that encapsulates this functionality behind a simple, easy to use API.

For the full documentation (JavaDoc) of the Product facet interface refer to Appendix C, *OpenConfigurator API/SPI*.

The Configuration Facet

With the help of the Configuration facet, configuration specific information about a component and its attributes or parts can be determined:

```
Component<Frame> frame = ...
Configuration<Frame> frameConfiguration =
    frame.getFacet(Configuration.class); ❶

SpecificationMethod specificationMethod =
    frameConfiguration.getSpecificationMethod(); ❷
assertTrue(specificationMethod.equals(CONFIGURABLE));

Attribute<Frame, Color> color = frame.getAttribute("color");
AttributeType colorAttributeType =
    frameConfiguration.getAttributeType(color); ❸
assertTrue(colorAttributeType.equals(AttributeType.VARIABLE));

Collection<Attribute<Frame, ?>> parameters =
    frameConfiguration.getAttributes(AttributeType.PARAMETER); ❹
assertTrue(parameters.contains(frame.getAttribute("label")));

Component<Bike> bike = ...
Configuration<Bike> bikeConfiguration =
    bike.getFacet(Configuration.class);
PartType wheelsPartType =
    bikeConfiguration.getPartType(bike.getPart("wheels")); ❺
assertTrue(wheelsPartType.equals(PartType.CONFIGURED));
```

As usual, the configuration facet view of a component is obtained using the Component interface's getFacet method ❶. The Configuration facet interface provides methods retrieving all conceptual, configuration related information about a component, as described in Section 4.4.3, "Configuration Modeling".

The getSpecificationMethod allows to determine the default specification method of a configurable type, that is, either SELECTABLE, CONFIGURABLE or CONSTRUCTIBLE. The value is represented by the Java enumerated type SpecificationMethod ❷.

Similarly, the attribute types REGULAR, VARIABLE, PARAMETER and CALCULATED are represented by the AttributeType enumeration and can be requested for a particular attribute us-

ing the `getAttributeType` method ⑤. The Configuration facet interface also provides the method `getAttributes(AttributeType)` to retrieve a collection of all attributes of a given type ④.

The `PartType` enumeration contains the type-safe literals `REGULAR` and `CONFIGURED` to distinguish non-configurable from configurable parts. As the Configuration facet interface provides methods for attributes, it provides the methods `getPartType(Part)` and `getParts(PartType)` for checking a part's type and retrieving parts of a particular type respectively.

For the full documentation (JavaDoc) of the Configuration facet interface refer to Appendix C, *OpenConfigurator API/SPI*.

The Data Facet

Through the Data facet, API clients gain access to the data, that relates to a particular component. Specifically, the interface enables the inspection of the *domains* and *defaults* of a particular component, as described in Section 4.4.4, "Data Modeling". We'll discuss the domain related methods first:

```
Component<Bike> bike = ...
Data<Bike> bikeData = bike.getFacet(Data.class); ❶

Domain<Class<? extends Bike>> bikeTypeDomain =
    bikeData.getComponentTypeDomain(); ❷
assertTrue(bikeTypeDomain.isEnumerable() &&
    bikeTypeDomain.getSize() == 3 &&
    bikeTypeDomain.contains(MountainBike.class) &&
    bikeTypeDomain.contains(CityBike.class) &&
    bikeTypeDomain.contains(ElectroBike.class));

Component<Frame> frame = bike.getPartComponent("frame");
assertTrue(frameData.getDomainScope().equals(PROPERTY)); ❸

Attribute<Frame, Color> color = frame.getAttribute("color");
Domain<Color> colorDomain =
    frameData.getAttributeValueDomain(color); ❹
assertTrue(colorDomain.isEnumerable() &&
    colorDomain.getDomainType().equals(SYMBOLIC));

for (Color definedColorValue : Color.values())
    assertTrue(colorDomain.contains(definedColorValue)); ❺

Attribute<Frame, String> label = frame.getAttribute("label");
Domain<String> labelDomain =
    frameData.getAttributeValueDomain(label); ❻
assertTrue(!labelDomain.isEnumerable() &&
    !labelDomain.isBounded() &&
    labelDomain.getDomainType().equals(LITERAL));

Part<Bike, Wheels> wheels = bike.getPart("wheels");
Domain<Wheels> wheelsDomain =
    bikeData.getPartValueDomain(wheels); ❼
assertTrue(wheelsDomain.isEnumerable(10));

Iterable<Wheels> wheelsValues = wheelsDomain.getValues(); ❽
...
```

Again, an instance of the Data facet for the Bike component is obtained using the `Component.getFacet(Class)` method ❶. First of all, we're inspecting the *type domain* of the component using the `getComponentTypeDomain` method ❷. The Bike class' type closure on-

ly consists of a few classes, which is why the domain is enumerable and contains exactly the non-abstract types extending `Bike`, namely `MountainBike`, `CityBike` and `ElectroBike`.

Next, we're inspecting the *attribute domains* of the `Frame` component. Since `Frame` itself is annotated `@Configurable` and individual properties are annotated with `@Domain` annotations, respectively fall back on their implicitly defined domains (see Example 4.14, "Implicit and Explicit Definition of Attribute Value Domains"), the domain scope is considered to be *attribute level*, as indicated by the `DomainScope.PROPERTY` enumerated value ❸. A specific domain can be retrieved using the `getAttributeValueDomain(Attribute)` method, which returns a parameterized `Domain` object ❹. The domain of the `color` attribute is an enumerable, symbolic domain and contains all values of the `Color` enumerated type ❺. The `label`'s domain, in turn, is an unbounded, literal domain.

Finally, part value domains are managed by the `Data` facet. Using the `getPartValueDomain(Part)` method of the `Data` facet interface, a `Domain` object representing a part's domain can be obtained ❷. The domain in this case is build from queried tuples of the given type, as shown in Section 4.4.4, "Data Modeling". If the domain is enumerable with respect to a given threshold value, 10 in this case, the concrete domain values can be determined using the `getValues` method of the `Domain` object ❸.

Beyond the inspection of domains, the `Data` facet allows to retrieve information about *default values*, as defined in Section 4.4.4.2, "Defaults":

```
Component<Bike> bike = ...
Data<Bike> bikeData = bike.getFacet(Data.class);
Class<? extends Bike> defaultBikeType =
    bikeData.getDefaultComponentType(); ❶

Component<Frame> frame = bike.getPartComponent("frame");
Data<Frame> frameData = frame.getFacet(Data.class);
Attribute<Frame, Double> height = frame.getAttribute("height");
Double defaultHeightValue =
    frameData.getDefaultAttributeValue(height); ❷

Component<Wheels> wheels = bike.getPartComponent("wheels");
Data<Wheels> wheelsData = wheels.getFacet(Data.class);
Wheels defaultWheelsValuePartLevel =
    bikeData.getDefaultPartValue(wheels); ❸
Wheels defaultWheelsValueComponentLevel =
    wheelsData.getDefaultComponentValue(); ❹
```

Equivalent to domains, default values exist for component type decisions as well as attribute and part value related decisions. The default type for a component can be requested using the `Data` interface's `getDefaultComponentType` method ❶. For attributes, the default value can be determined with the `getDefaultAttributeValue(Attribute)` method ❷. Likewise, a part's default value can be obtained using the `getDefaultPartValue(Part)` method ❸. If not defined on part level, that is, a `@Default.Query` annotation is not present on the part's `JavaBean` member, the default value can alternatively be defined by putting the `@Default.Query` annotation on the target component type. In this case, the actual value can be obtained using the `getDefaultComponentValue` method of the target component's data facet ❹.

The full documentation (JavaDoc) of the `Data` facet interface as well as the `Domain` interface can be found in Appendix C, *OpenConfigurator API/SPI*.

The Validation Facet

Finally, during the configuration process, constraint checking is performed with the help of the `Validation` facet. The use of that facet interface is shown in the following listing:

```
Component<Bike> bike = ...
Validation<Bike> bikeValidation =
    bike.getFacet(Validation.class); ❶

// obtain constraints defined on component level
Collection<Constraint<Bike>> componentConstraints =
    bikeValidation.getComponentConstraints(); ❷

// obtain constraints defined for a particular attribute
Attribute<Bike, String> label = bike.getAttribute("label");
Collection<Constraint<String>> labelConstraints =
    bikeValidation.getAttributeConstraints(label); ❸

// obtain constraints defined for a particular attribute
Part<Bike, Wheels> wheels = bike.getPart("wheels");
Collection<Constraint<Wheels>> wheelsConstraints =
    bikeValidation.getPartConstraints(wheels); ❹

// perform validation
bikeValidation.validate(); ❺

// retrieve constraint violations
for (ConstraintViolation violation :
    bikeValidation.getViolations()) ❻
{
    log.message("Constraint " + violation.getConstraint() +
        " on target " + violation.getTarget() +
        " is not satisfied.");
}
```

Again, the Validation facet is obtained using the Component's `getFacet(Class)` method ❶. The Validation interface provides various methods to inspect the constraints defined for a component: the `getComponentConstraints` method returns all Constraint instances defined on component level ❷. The `getAttributeConstraints(Attribute)` method, in turn, allows to retrieve all constraints defined on a particular attribute ❸, while the `getPartConstraints(Part)` method returns all constraints defined for a specific part ❹.

The collections returned by these methods contain parameterized Constraint instances, each representing the state of a particular annotation-defined domain model restriction. Through the Constraint interface, a restriction can be validated using the `Constraint.validate()` method. However, client applications do not need to invoke these methods explicitly, since the default implementation of OpenConfigurator's validation subsystem continuously performs validation in the background, that is, whenever a value of the domain model changes. Otherwise, when automatic validation is turned off or not supported by an alternative implementation, validation may be triggered explicitly using the Validation facet's `validate` method ❺.

Finally, constraint violations, if any, can be obtained using the `getViolations` method ❻. In order to identify the exact model element or constraint, that caused the violation, the `ConstraintViolation` interface offers the methods `getTarget`, respectively `getConstraint`.

The full documentation (JavaDoc) of the Validation facet interface as well as its related interfaces `Constraint` and `ConstraintViolation` can be found in Appendix C, *OpenConfigurator API/SPI*.

5.3.2.5. Working with the Configuration Agenda

The core component model API, including the related facet interfaces, provide low-level access methods to configuration activities. OpenConfigurator also provides a high-level, task

based interface on top of that API. It's accessed through an instance of Agenda, which, in turn, can be obtained from the current ConfigurationSession:

```
ConfigurationSession<Bike> session =
    configurator.createSession(Bike.class);
Agenda<Bike> agenda = session.getAgenda(); ❶

Iterator<Task<Object>> openTasks = agenda.selectItems(
    new Filter() ❷
    {
        public boolean select(Task<?> task)
        {
            return !(task instanceof CompositeTask) &&
                !task.isComplete(); ❸
        }
    }).iterator();

// simplified configuration process
while (openTasks.hasNext()) { ❹
    Task<Object> task = openTasks.next();
    Domain<Object> domain = task.getDomain();

    if (domain.isEnumerable())
    {
        Object chosenValue = letUserSelectValue( ❺
            task.getDescription(),
            domain.getValues());
        task.setValue(chosenValue);
    }
    else if (domain.isBounded())
    {
        Object enteredValue = letUserEnterValue( ❻
            task.getDescription(),
            domain.getValueType(),
            domain.getLowerBound(),
            domain.getUpperBound());
        task.setValue(enteredValue);
    }
    else
    {
        Object enteredValue = letUserEnterValue( ❼
            task.getDescription(),
            domain.getValueType());
        task.setValue(enteredValue);
    }
}

assertTrue(session.getConfiguration().isComplete()); ❽
```

First, we obtain an instance of Agenda from the current ConfigurationSession ❶. Next, we select ❷ all non-composite (see Section 4.5.3, “Agenda Task Types” below), not yet completed tasks ❸ using Agenda's filtering method.

Then, we apply a very simplified configuration process: we loop over all remaining tasks and extract each task's domain for further inspection ❹. Depending on the domain's size, we

- let the user select a particular value from the enumerated domain ❺,
- let him enter a value from a bounded interval ❻, or
- let him enter an arbitrary value matching the given type ❼.

This process is repeated until all tasks have been resolved. In this case, we can assert, that the configuration is completed ❸.

Once the specification of the configurable instance is complete, the configuration can be submitted for further processing. This is discussed in the next section.

5.3.3. Configuration Submission and Further Processing

In the previous section, we've seen how configuration itself and related activities, such as domain data retrieval and validation, can be performed. Now, when a configuration is completed, it may be submitted by the user in order to be processed by other backend systems.

How these tasks are supported by the OpenConfigurator API is shown in the following listing:

```
ConfigurationSession<Bike> session = ... ❶

// assert configuration validity and completeness
Configuration<Bike> configuration = session.getConfiguration();
assertTrue(configuration.isValid() &&
            configuration.isComplete()); ❷

// persist configuration
session.store(id); ❸

// submit configuration for further processing
session.submit(); ❹

// close session
session.close(); ❺
```

To finalize the configuration process, the current `ConfigurationSession` instance must be obtained via injection ❶. Next, clients usually verify, whether the configuration is valid, using the `isValid` method, and check if all options have been specified using `isComplete` ❷.

If both methods return `true`, that is, the configuration is complete and doesn't violate any constraints, it is stored under a given ID ❸ and submitted for further processing ❹. Both calls emit events qualified with `@Stored` respectively `@Submitted`, using the current `Configuration` as a payload.

The `@Submitted Configuration` event may be observed by third-party extensions/plugins to feed the resulting domain model `JavaBean` instance to an external system:

```
public class ERPConnector
{
    @Inject
    private ERP erp;

    public void process(
        @Observes @Submitted Configuration<Bike> configuration) ❶
    {
        Bike configuredBike = configuration.getValue(); ❷
        erp.order(bike);
        ...
    }
}
```

In the example above, the `ERPConnector` observes any submitted bike configurations ❶. Upon method invocation, triggered by configuration submission, the configured `Bike` instance

is extracted using the `Configuration` instance's `getValue` method ❷. The returned `JavaBean` instance is a regular domain model object, that can be used in arbitrary subsequent operations. In this case it is passed to the fictitious `ERP.order(Bike)` method.

Finally, the session is closed using the `ConfigurationSession.close()` method ❸. The call to the `close` method allows `OpenConfigurator` to establish required cleanup behavior.

This ends up our basic API walk-through. It's up to third-party extensions and plugins to further process the configured, validated `JavaBean` instance. In the next section, we'll discover, how such extensions and plugins can be implemented on top of `OpenConfigurator`'s SPI.

5.4. SPI Usage: Extending and Integrating OpenConfigurator

As mentioned in Section 5.2, “Architectural Overview”, `OpenConfigurator` features a highly flexible, extensible application architecture enabled by the CDI programming model. In this section, we'd like to provide a simple example to demonstrate the extension possibilities of the framework. We'll not go into more specific details of the SPI though.

Example Plugin: the JPA Meta Model Facet

As an example use case, we will implement a plugin, that exposes the JPA meta model of a given configurable type as a facet of the corresponding `ComponentDescriptor`. Consider the code below:

Example 5.1. SPI Usage Example

```
import javax.enterprise.event.Observes;
import javax.inject.Inject;
import javax.persistence.EntityManagerFactory;
import javax.persistence.metamodel.ManagedType;

import configurator.spi.Initialization;
import configurator.spi.metadata.ComponentDescriptorImplementor;

/**
 * A plugin that exposes the JPA metamodel of a managed component
 * type as a facet of that component.
 */
public class JPAMetaModelFacetPlugin ❶
{
    @Inject
    private EntityManagerFactory emf; ❷

    public <C> void registerFacet(
        @Observes @Initialization
        ComponentDescriptorImplementor<C> implementor) ❸
    {
        if (emf == null)
            return;

        Class<C> componentType = implementor.getComponentType();

        ManagedType<C> managedType =
            emf.getMetamodel().managedType(componentType); ❹

        if (managedType != null)
            implementor.addFacet(ManagedType.class, managedType); ❺
    }
}
```

- ❶ The plugin is a regular Java class, that doesn't need to implement any OpenConfigurator specific interface.
- ❷ Within a plugin, dependency injection can be used as almost anywhere in the OpenConfigurator framework. In this concrete example, the plugin defines a dependency on the EntityManagerFactory provided by the JPA container.
- ❸ By observing the ComponentDescriptorImplementor initialization event (see Section 5.2.4, “Global Event Bus”), the plugin integrates itself into the initialization process of the component descriptor. Typically, this event is emitted during the meta model extraction process, that is, on framework startup. The object passed as a method argument corresponds to the implementor, which, in turn, represents the component descriptor currently being initialized (see Section 5.2.4, “Implementors”).
- ❹ After determining the Java type of the component descriptor being initialized, the corresponding ManagedType is retrieved from the JPA 2 meta model.
- ❺ The returned ManagedType (if any) is registered as a facet of the component descriptor using the parameterized addFacet(Class<F> type, F facet) method. Note that ManagedType doesn't implement any OpenConfigurator specific interface and is simply identified by its Java type.

As long as the plugin implementation is part of the configurator application's classpath, OpenConfigurator, respectively the CDI container, automatically notifies the plugin, when a ComponentDescriptor is being initialized.

The following code snippets demonstrates, how the facet can be used:

```
Component<JavaBean> component =
    session.getConfiguration().getRootComponent();
ComponentDescriptor<JavaBean> descriptor = component.getMetadata();

if (descriptor.hasFacet(ManagedType.class))
{
    ManagedType<JavaBean> facet =
        descriptor.getFacet(ManagedType.class); ❶

    PersistenceType persistenceType = facet.getPersistenceType(); ❷
    // ...
}
```

We obtain the facet using the ManagedType.class as an identifier ❶. As an example, we request the persistence type (entity, embeddable, mapped superclass, etc.) of the JavaBean component ❷.

Admitted that this is a quite simple example, it shows, how third party extensions can integrate themselves into the initialization process of framework managed types.

Meaningful examples for extensions include the integration of PDM systems or other external software, such as ERP or CAD systems, that complement the configuration model (or meta model) with additional information.

5.5. Summary

In this chapter, we described the architecture and implementation of OpenConfigurator's execution environment. Effectively, this generic runtime infrastructure is used to turn arbitrary JavaBean domain models, enriched with annotations introduced in Section 4.4, “Modeling Concepts”, into configuration processes.

In the first section of this chapter, we discussed the technology stack OpenConfigurator is based on. We discussed various technologies and highlighted their most important features with respect to our use case, including:

- Java, including annotations and APT,
- JPA,
- Bean Validation, and
- CDI.

Particularly, we identified *Context and Dependency Injection (CDI)* as fundamental programming model behind the framework's architecture, which was subject to the next section.

Architecturally, we characterized OpenConfigurator as multi-tier based application, that can be used in client-server scenarios. Moreover, we stated, that the framework implementation primarily provides the *logic tier components* within that architecture. The main logic tier parts realized by OpenConfigurator are:

Client configuration interface. The *client configuration interface* primarily comprises the configuration API and services, accessed by the generic configuration model implementation. The interface is used by client applications in order to realize, respectively control, configuration processes.

Internal implementations. OpenConfigurator comes with a number of default services and other implementations of framework provided types. The default service implementations realize the integrations with various other Java EE technologies, including JPA, Bean Validation and JAXB.

Extension and Integration SPI. In order to allow extending the framework and integrating it within different deployment contexts, OpenConfigurator provides an *extension and integration SPI*. The main building blocks of the SPI are the implementors facility and the global event bus realized by the CDI event sub-system.

Plugins and Connectors. Finally, OpenConfigurator foresees to be accompanied by additional plugins and connectors, that implement additional functionality on top of the generic configuration model and the extension and integration SPI.

The practical usage of the configuration API and the framework provided SPI, was subject to the following sections. We explained, that the main steps of working with the OpenConfigurator configuration interface include:

1. Obtaining a configuration session for a particular configurable type from the configurator.
2. Performing the actual configuration process until the product specification is complete and valid.
3. Submitting the configuration and further processing.

We demonstrated, that the actual configuration activities (navigation, option specification, validation, etc.) can be accomplished by utilizing the low-level generic configuration model API together with the integrated facets. Alternatively, we showed, that the high-level, task bases agenda API can be used to specify configurable products more conveniently.

Finally, we discussed a concrete example to extend the OpenConfigurator framework.

In the next chapter, Chapter 6, *Evaluation and Validation*, we will introduce a complete case study implementing a bike configurator for the iPad mobile device. The developed configurator, will be backed by a generic mobile configuration client, which is, in turn, build on top of the client configuration interface.

6

Evaluation and Validation

In the previous chapters, we introduced the OpenConfigurator modeling approach as well as the framework's runtime engine, that is used to perform configuration tasks. In this chapter we will evaluate both topics by realizing a sample configurator application based on our methodology. The evaluation allows us to validate, whether the configurator framework meets the requirements identified earlier in an adequate manner.

Specifically, in this chapter, we will:

- introduce an exemplary use case "Bike configuration" and develop a sample domain model, including data and constraints¹, and
- implement a generic mobile configurator client application for the iPad mobile device, that runs the sample domain model.

With these steps we want to demonstrate, that:

- the conceptual framework is adequate to model real world configuration problems, and
- the configurator provided API is sufficient to implement a generic, web-based mobile configurator client on top of it.

We will first describe the case study in some more detail and then move over to the generic configurator client. Finally, we'll show some exemplary specification procedures before validating the overall configurator realization approach.

6.1. Case Study: Bike Configurator for the iPad

As an example use case, we chose to implement a subset of a bike configurator for the fictitious company MyBike Inc..

6.1.1. MyCustomBike Inc. - the Custom Bike Company

Imagine MyCustomBike Inc. being a new market competitor that primarily seeks to fill the niche for modern, stylish pedal electric bicycles (also known as *pedelecs*²). Not only does My-

¹The sample product model has been inspired by Renneberg's bike example, see [Renneberg2010, pp. 265].

²See [http://de.wikipedia.org/wiki/Pedelec], last accessed July 10th, 2012.

CustomBike Inc. aim to provide alternative styled bikes, they also want to offer various customization options to make every single sold cycle a unique product. As it is a young company, with yet relatively low revenue, the management tries to keep any costs, including expenditures for marketing and sales, as low as possible.

An online configurator, that allows customers to explore and self-configure their custom traditional bikes and pedelecs offers great opportunities for MyCustomBike Inc. to increase sales by simultaneously saving personnel costs.

Additionally, MyCustomBike Inc. targets a younger to medium aged audience (people between ~20 to ~40), which is style affine and prefers custom-made or even hand-made products, manufactured in nearby companies, over mass-produced goods from a globalized enterprise. Also, the targeted audience is expected to be quite open-minded to new media and modern technologies, including new ubiquitous computing devices, such as smart phones or tablet computers.

MyCustomBike Inc. decided to implement a state-of-the-art, web-based, mobile bike configurator primarily focussing on mobile devices such as the Apple iPad in order to accommodate the characteristics of the targeted customer group adequately. Developing for such devices not only fulfills the ongoing "hype" for mobile appliances, but also enables a great buying experience for customers, when used by sales personnel in traditional retail stores. Moreover, at the current stage, there doesn't exist a single configurator for such devices in this application domain, which further emphasizes and communicates the uniqueness of MyCustomBike Inc.'s business approach.

6.1.2. MyCustomBike Inc.'s Product Range

The product range of our fictitious company not only includes specialized pedelecs, but also covers traditional bikes with a customizable styling. The main bike categories offered, are:

- Racing bikes
- Mountain bikes (MTB)
- City bikes
- Electro bikes (pedelecs)

Each bicycle may be customized in terms of its size, color, labeling and equipment. Basically, the following components make up an individual bike (see Figure 6.1, "Components of a Bike"):

- Frame
- Gearing
- Brake
- Cranks & Pedals
- Stem & Handlebar
- Wheels
- Seat post & Saddle
- Motor & Battery
- Equipment

The exact specifications of existing component variants, that are used within the example implementation, can be found in Section B.1, "Component Specifications".

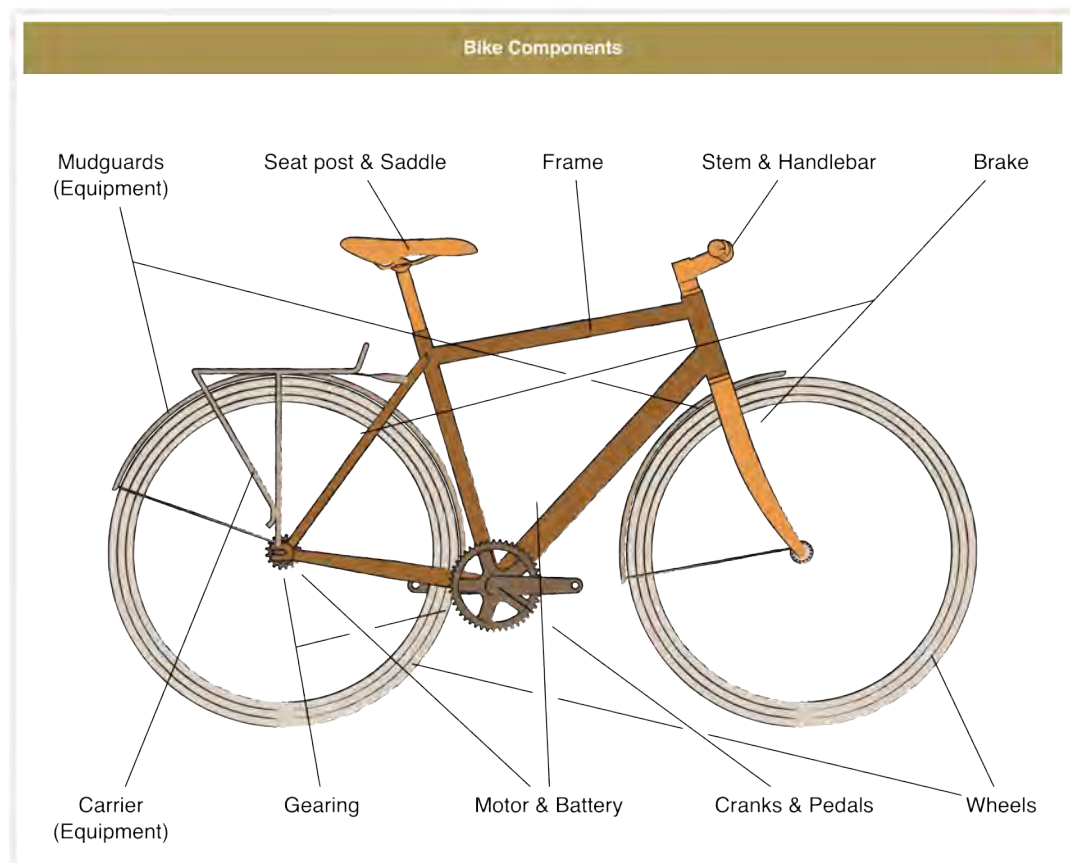


Figure 6.1. Components of a Bike³

6.1.3. The Bike Domain Model

Figure 6.2, “UML Class Diagram for the Bike Domain Model” shows the bike domain model for our example configurator application.

The complete source code of the example can be found in Section B.2, “Source Code”.

The domain model makes use of the following OpenConfigurator concepts:

Specification methods.

- **Selectable.** Most parts of the Bike product use the specification method `@Selectable`, including `SeatpostSaddle`, `StemHandlebar`, `Brake`, `Gearing`, `CranksPedals`, `Wheels`, `Equipments` and `MotorBattery`.
- **Configurable.** The Bike's `Frame` part represents an `@Configurable` component.

Attribute types.

- **Regular.** All components' description and price attributes are regular ones, that is, invariable attributes.
- **Variable.** If not stated otherwise, all remaining attributes are variables.
- **Parameter.** The `Frame` component features a single, freely definable `String` parameter, called `label`.
- **Calculated.** The weight attribute of the Bike product as well as the `Frame`'s size attribute are calculated from other values and are thus read-only.

³Technical drawing copied with permission from electrolyte.cc, see <http://www.electrolyte.cc/>. All rights reserved.

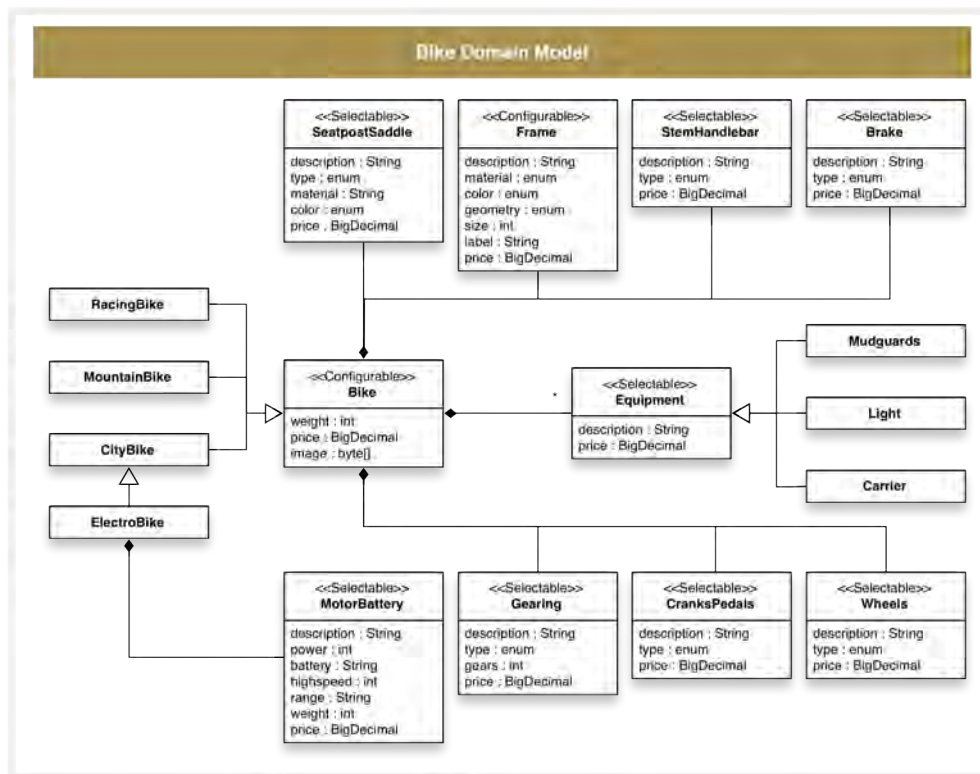


Figure 6.2. UML Class Diagram for the Bike Domain Model

Domain scopes.

- **Component-level domains.** All selectable components make use of component level domains defined using the `@Domain.Query` annotation.
- **Attribute-level domains.** The configurable component `Frame` uses both a component level `@Domain.Query` annotation and implicitly defined domains on attribute level in terms of type-safe Java enumerations.

Domain types.

- **Component type domains.** While the example model's class hierarchy is rather flat, there are two component type domains containing more than a single value. Namely, the `Bike` type domain (containing the values `RacingBike`, `MountainBike`, `CityBike` and `ElectroBike`) and `Equipment` (with subtypes `Mudguards`, `Light` and `Carrier`). Hence, the specification of the bike component as well as the one of equipment require a user decision during configuration.
- **Attribute value domains.**
 - *Symbolic:* There are many properties within the model, such as `Frame.material`, `Frame.color`, `Brake.type` etc. that use Java enumerations as data type. Enumerations are automatically mapped to symbolic domains.
 - *Numeric:* The `Gearing.gears` attribute is an example for a numeric (enumerated) domain.
 - *Literal:* The `label` attribute of the `Frame` component represents an (unbounded) literal domain.

Domain sizes.

- **Fixed.** There are a number of attribute value domains as well as tuple domains, that are fixed to a singular value by definition. That is, their domain contains only a single distinct value. Examples include `MotorBattery.power`, `MotorBattery.maximumSpeed` and the `Light` and `Carrier` equipments.

- **Enumerated.** Most domains of the example bike model have a limited number of elements. Among all properties modeled with Java enum types, which are implicitly enumerated (e.g., `Frame.color`, `Frame.material`, `SeatpostSaddle.type` etc.), all other domains except the `Frame.label` parameter attribute are enumerated. This is due to the fact that component level `@Domain.Query` annotations are used.
- **Unbound.** The `Frame.label` signifies an unbounded domain as it may be assigned arbitrary String values.

Defaults.

For all selectable components, a default tuple is selected via `@Default.Query` annotations. Additionally, for the Bike subtypes `MountainBike` and `RacingBike` a dedicated default is defined for the `wheels` part. For the configurable component `Frame`, attribute level defaults are used.

Constraints.

- **Relational.** All selectable components are annotated with the `@Relational` constraint, to restrict possible value combinations to the available relational tuples.
- **Required/Optional.** Within the Bike product, the minimum quantity is defined for each part using either the `@Required` or `@Optional` annotation. As most components are `@Selectable`, a more detailed specification of their attributes with one of both annotations is obsolete, since the selection of a particular tuple assigns a value to all remaining attribute anyway. However, for the `@Configurable` `Frame` component, all `@Variable` attributes are annotated `@Required`.
- **Size (Bean Validation).** The literal `Frame.label` attribute is annotated with `@javax.validation.constraints.Size` to constrain the maximum length of the string entered.
- **Conditional Min/Max.** The `Gearing.gears` attribute has been annotated with conditional `@Min/@Max` constraints, to explicitly model the relationship between the gearing type and the speeds available.

Product data.

- **Description.** All description attributes are mapped to product descriptions using the `@Product.Description` annotation.
- **Price.** The components' price attributes are annotated `@Product.Price`.
- **Asset.** The `Bike.image` attribute is used to render a visual representation of the product. Hence, it's annotated `@Asset`.

The concepts stated above will be visualized in Section 6.3, “Configuration Procedures”, when we discuss various example configuration activities.

6.2. The Generic Mobile Configurator Client

Before going into further details of the Bike configurator, we will shortly discuss the technologies and general architecture of the generic configurator for the iPad, which was developed as part of this case study.

6.2.1. Technology

For implementing the generic configurator client for the iPad mobile device, we chose to create an HTML5 and JavaScript based web application with mobile and touch support. This way, we avoided a vendor lock-in by not implementing it natively for the iOS⁴ operating system. Instead the application can be accessed on any other mobile device as well. Thus we serve multiple devices with one and the same code base.

⁴See <http://www.apple.com/de/ios/>, last accessed August 5th, 2012.

The main enabling technologies to facilitate the user interface of our mobile application client are Vaadin and the Vaadin Touchkit add-on.

Vaadin

Vaadin is a powerful technology that can be used to create HTML⁵ based, fully AJAX⁶ driven user interfaces (UIs) for complex web applications. Special about Vaadin is, that it allows to implement the UI entirely in (server-side) Java and doesn't require writing a single line of HTML or JavaScript in order to create sophisticated web applications. Internally, Vaadin builds on the Google Web Toolkit (GWT)⁷, a technology that basically compiles regular Java source code into highly optimized JavaScript code, that runs in the web browser.

Hence, Vaadin is a server-side technology, that completely hides the stateless nature of the HTTP protocol from the developer. It allows to implement user interfaces in a manner known from other popular desktop UI technologies like Java Swing⁸ or SWT⁹. Particularly, it provides a comprehensive set of UI widgets, a sophisticated component model as well as a flexible layout and theming mechanism. Opposed to GWT's solely *client-side* UI model, which is compiled from Java source code, Vaadin implements a complete *server-side* UI component model and manages the communication with the client-side widgets transparently. Vaadin this way avoids transmitting entire data models to the client. Instead, only the visual components' state changes and visible parts of the data models are transferred over the wire. This dramatically simplifies web application UI development for the mainstream Java developer, as he doesn't need to cope with remote procedure calls (RPCs) and the transmission of state between client and server at all.¹⁰

Vaadin Touchkit

Moreover, Vaadin provides more than 170 add-ons¹¹, one of them being the Vaadin Touchkit¹², an add-on providing HTML5 based user interface components themed for the Apple mobile devices iPhone and iPad. With the help of the Vaadin Touchkit a browser based, touch enabled user interface with a native look & feel can be implemented in Java, leveraging the full power of the Java EE platform (and particularly the CDI technology, see Section 5.1.4, "Contexts and Dependency Injection for Java EE (CDI)").

For the OpenConfigurator framework, Vaadin and the Vaadin Touchkit add-on offer a straightforward way to bring the Java based, generic configuration API to the world of mobile devices. As both technologies are based on the Java platform, there's no technology break. Moreover, the fact that the UI component model of a running Vaadin application is completely held on the server within the user's session and the communication between the Vaadin UI and the OpenConfigurator framework happens directly through the configuration API, makes the implementation of a separate (stateful) service layer superfluous.

Let's have a look at the resulting architecture, now comprising Vaadin and OpenConfigurator components.

6.2.2. Architecture

The Vaadin/HTML5 based mobile user interface completes the architecture presented in Section 5.2, "Architectural Overview" by filling the yet not discussed presentation tier. Alto-

⁵Abbrev. HyperText Markup Language

⁶Abbrev. Asynchronous JavaScript And XML, for more information see [http://de.wikipedia.org/wiki/Ajax_\(Programmierung\)](http://de.wikipedia.org/wiki/Ajax_(Programmierung)), last accessed July 15th, 2012.

⁷See <https://developers.google.com/web-toolkit/>, last accessed July 15th, 2012.

⁸See [http://en.wikipedia.org/wiki/Swing_\(Java\)](http://en.wikipedia.org/wiki/Swing_(Java)), last accessed July 15th, 2012.

⁹Abbrev. Standard Widget Toolkit, see <http://www.eclipse.org/swt/>, last accessed July 15th, 2012.

¹⁰For more in-depth information about Vaadin, its architecture and available features as well as their use, we recommend reading the freely available "Book of Vaadin", see <https://vaadin.com/book/> last accessed July 15th, 2012.

¹¹See <https://vaadin.com/directory>, last accessed July 15th, 2012.

¹²See <http://vaadin.com/touchkit>, last accessed July 15th, 2012.

gether, this leads to a 4-tier client-server architecture, which is shown in Figure 6.3, “Generic Mobile Configurator Architecture”:

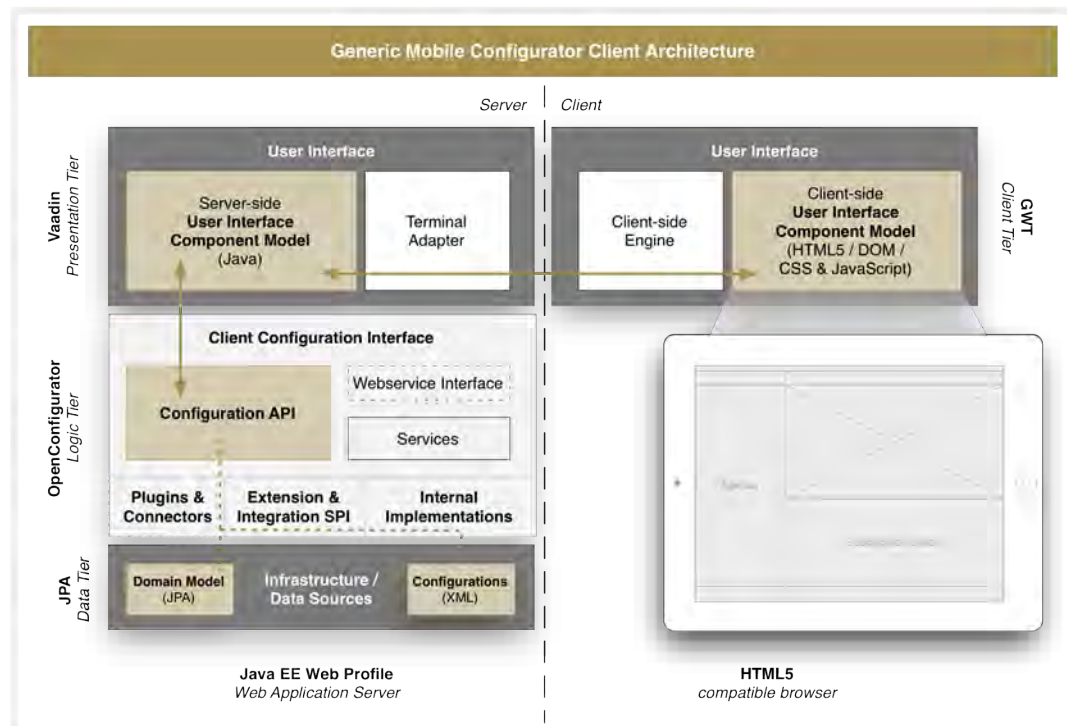


Figure 6.3. Generic Mobile Configurator Architecture

Data Tier. The *persistence layer* is made up of the aforementioned *domain model*, which is enriched with OpenConfigurator and JPA annotations. With the help of the JPA annotations, product data is retrieved from a relational database such as MySQL¹³. The partial and complete configurations, in turn, are stored as XML data models.

Logic Tier. The *application layer* is provided by the OpenConfigurator framework. In the context of the mobile configurator client based on Vaadin, the *Configuration API* of the *Client Configuration Interface* is of primary interest.

Presentation Tier. In contrast to regular 3-tier architectures for desktop applications, the *view layer* of our web-based mobile configurator comprises a server-side and a client-side part. To distinguish both parts, we refer to the former one as *presentation tier* and the latter one as *client tier* representing the fourth layer.

The presentation tier realized by the Vaadin technology contains the *server-side user interface component model* and the so called *terminal adapter*. The UI component model holds the state of the complete application UI within the user's session and is made up of Java objects. The terminal adapter facilitates the communication between client and server and synchronizes any state changes between both peers.

The Vaadin based presentation components directly interact with the Configuration API.

Client Tier. The client-side of the view layer is implemented with the help of the Google Web Toolkit (GWT). It features the *client-side engine*, which encapsulates the communication with the terminal adapter and which realizes the forwarding of any user interface events triggered by user interactions to the server-side UI component model. Moreover, the actual UI is realized using HTML5, DOM¹⁴ CSS¹⁵ and JavaScript.

¹³See <http://www.mysql.com>, last accessed July 16th, 2012.

¹⁴Abbrev. Document Object Model

¹⁵Abbrev. Cascading Style Sheets

The server-side part of the application, which is build entirely on top of the Java platform and can be executed on any Java EE 6 Web Profile compliant web application server, such as JBoss Application Server 7.1¹⁶. The client-side part can be run with any HTML5 compatible web browser, which is available on any modern mobile and desktop device today.

6.2.3. User Interface

Having discussed the mobile client's architecture in the last section, we now want to focus on the actual user interface of the generic mobile configurator application. Figure 6.4, “Mobile Configurator Wireframe” outlines the general layout of the configurator app:

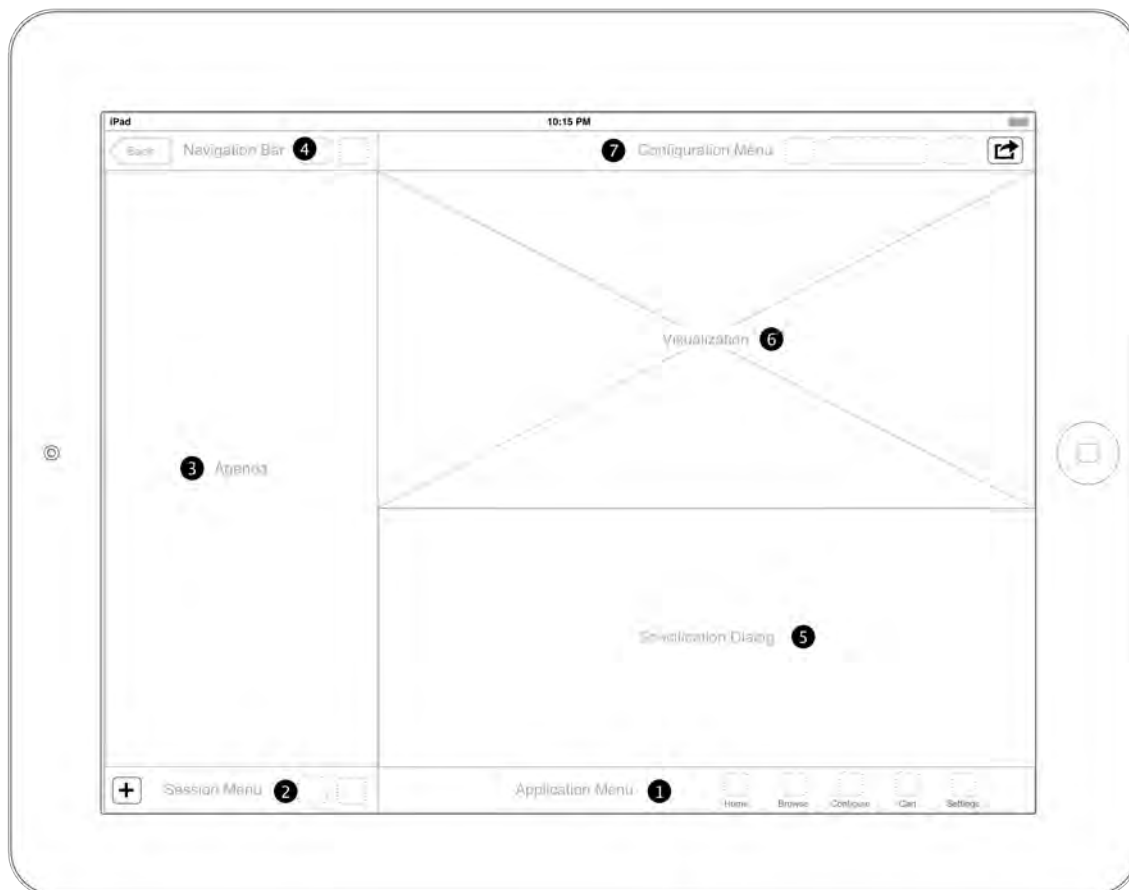


Figure 6.4. Mobile Configurator Wireframe

- ❶ **Application Menu.** The *application menu* allows to navigate between the main application parts, including:
- **Home.** Provides general information about the company offering the configurator. Also provides information on how to use the application.
 - **Browse.** The browser tab represents the built-in product and component catalog. This allows the customer to get an overview of all available component variants.
 - **Configure.** The main configuration tab contains the actual product specification dialog. In the following, we'll particularly focus on this tab.
 - **Cart.** The shopping cart provides an overview of configured and submitted items. From the cart tab a binding order can be placed. The shopping cart is equivalent to those of traditional online shopping systems.
 - **Settings.** The settings tab contains any application specific configuration settings.

¹⁶See <http://www.jboss.org/jbossas>, last accessed July 16th, 2012.

- ② **Session Menu.** The *session menu* is basically used to control the configuration session. It allows to create new configurations, save the currently running one and lets the user load previously stored configurations.
- ③ **Configuration Agenda.** The *configuration agenda* view represents the configuration process navigation. It allows to navigate through the entire model and displays the tasks for each element, that is present in the current configuration. Moreover, the agenda view shows status information about each particular item, whether it has been specified, contains validation errors or is completed. As the name indicates, the view is backed by the current session's Agenda object (see Section 5.3.2.5, “Working with the Configuration Agenda”).

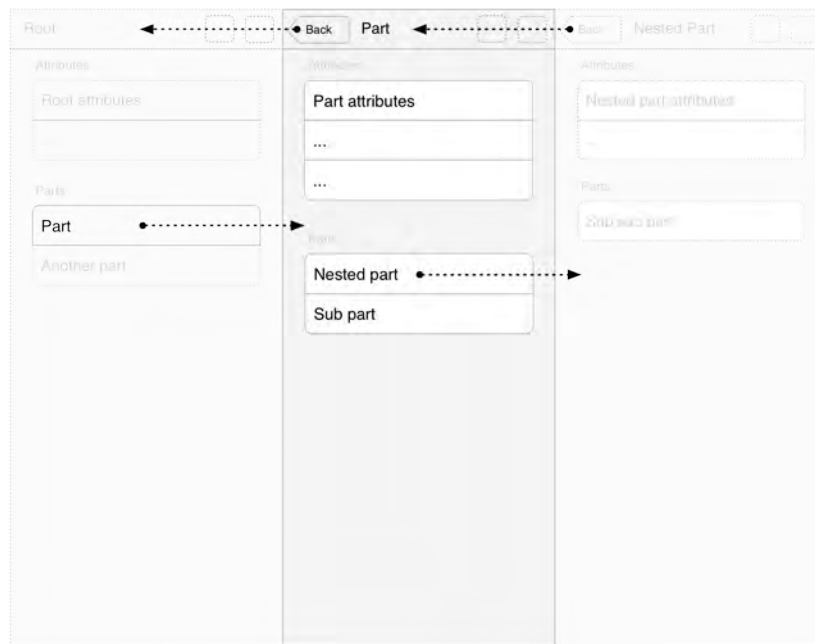


Figure 6.5. Hierarchical Agenda Navigation

The configuration agenda is a hierarchical data structure. Therefore, also the agenda view is organized as a stacked hierarchical navigation: for each part, that itself is a composite component, another navigation deck is added to the hierarchy. Each deck lists the standard product attributes as well as the particular component's nested parts. Figure 6.5, “Hierarchical Agenda Navigation” illustrates the navigational concept of the configuration agenda.

- ④ **Navigation Bar.** The *navigation bar*, which is placed above the agenda view, provides additional navigation controls: the *back* button goes back to the parent component, similar controls allow to move to the next/previous not-yet-completed task. Finally, the navigation bar contains an indicator about the validation state of the current configuration. If validation errors occur, an icon is displayed showing the number of constraint violations found. A click on the icon expands a menu, that allows to navigate to the different faulty tasks.
- ⑤ **Specification Dialog.** The actual configuration decisions are captured by the *specification dialog*. The specification dialog shows all product related information of a particular component and contains a tabular representation of the component's attributes. Variable attributes can be edited inline.
- ⑥ **Visualization.** Assuming the product provides a visual representation, the visualization view can be displayed above the specification dialog. Whether the product image produces customized visualizations, that include configuration decisions, depends on the domain model's method annotated with `@Product.Asset`, which must accommodate those decisions if needed.

- ⑦ **Configuration Menu.** The *configuration menu* allows to access additional information about the current configuration. For instance, the menu permanently displays the actual price of the resulting configuration. Moreover, an information button lists all configured components in a flat table (implementing the *Configuration Description* feature, see Section 3.3.2.2, “Specification / Recommendation” for details). Finally, the configuration menu provides an action button, that allows to perform further operations with the completed configuration, such as submitting the configuration for purchase, viewing generated product specification documents or sharing the configuration with others via e-mail.

Figure 6.6, “Exemplary Screenshot of the Generic Configurator” shows an exemplary screenshot of the generic mobile configurator.

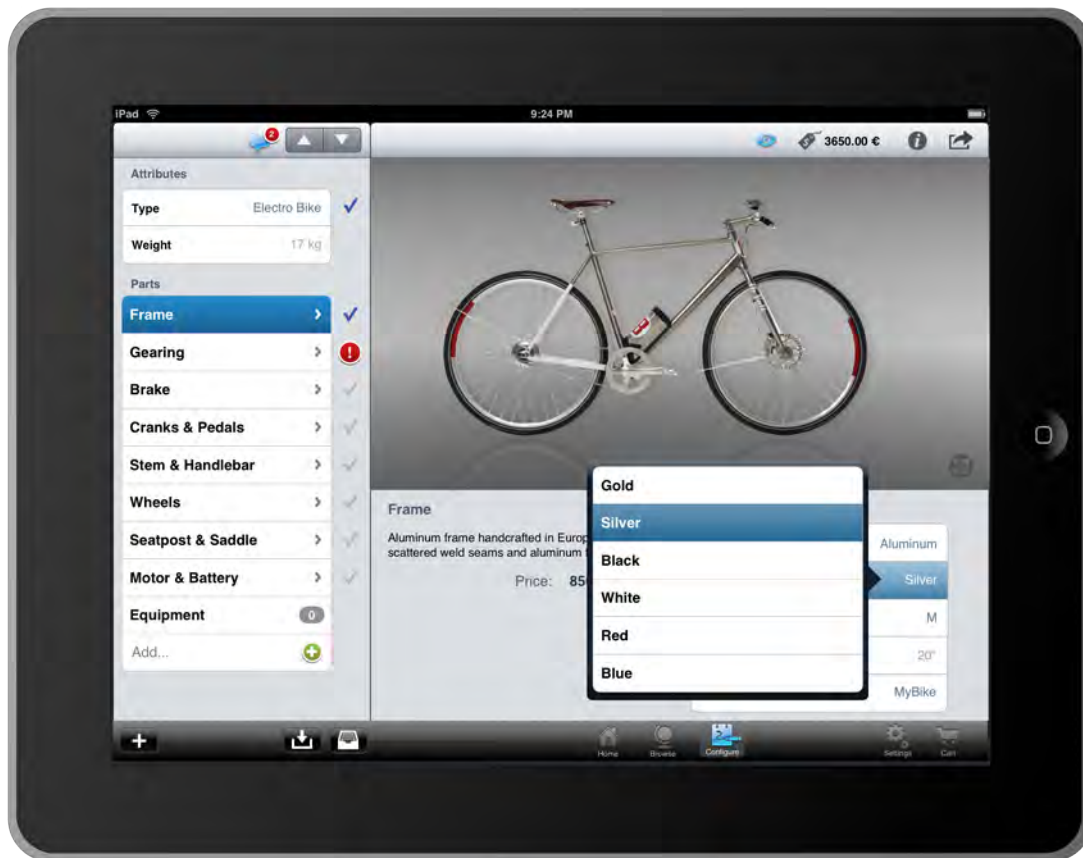


Figure 6.6. Exemplary Screenshot of the Generic Configurator

6.3. Configuration Procedures

We finally want to show some example configuration procedures within the Bike configurator. We selected the following scenarios, which will be described in more detail below:

- Creating a new bike configuration and selecting the concrete bike type
- Configuring the bike frame component
- Triggering and resolving a configuration error
- Selecting wheels
- Adding an equipment
- Submission and check out

6.3.1. Creating a New Bike Configuration

The configurator app initially starts with a fresh session. Figure 6.7, “Initial Configurator State after Creating a New Configuration” shows the initial state of the configurator app. In order to create a new configuration the user clicks the "Add" button of the session menu ❶. If the configurator's domain model would feature multiple configurable products, that is, classes annotated `@Configurable @Product`, a drop-down menu would appear that lets the user choose the product to configure. However, as the Bike class is the only type annotated alike, a new `ConfigurationSession` instance is created.

Within the `ConfigurationSession` object the actual `Configuration` is instantiated and the complete generic configuration model gets constructed. Furthermore, all defaults are applied and the model is initially validated. Simultaneously, the Agenda for the new configuration is set up, reflecting all decisions to be taken by the user in terms of configuration tasks. The agenda view is populated with the Agenda object ❷ and application control is moved to the first configuration task, that is, the specification of the Bike component's concrete type ❸.

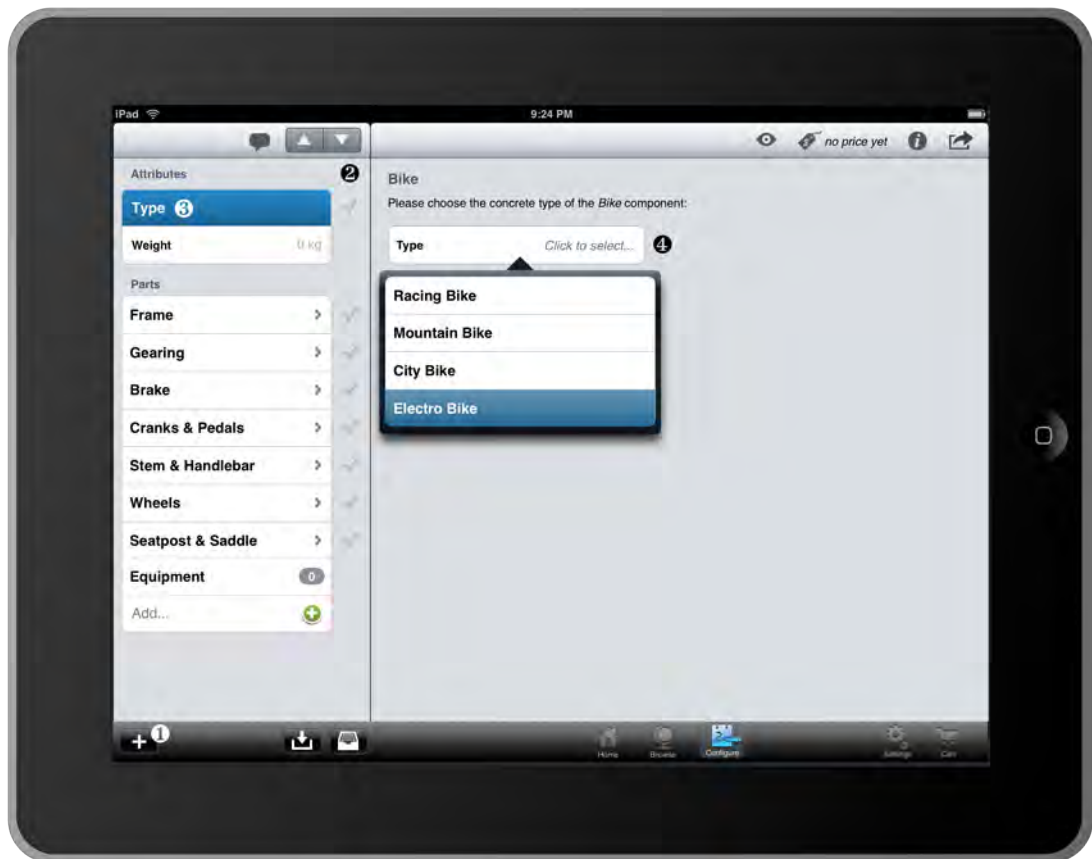


Figure 6.7. Initial Configurator State after Creating a New Configuration

The user chooses `Electro Bike` as type of the root component ❹. Figure 6.8, “Configuration State after Type Selection” shows the updated state.

Now, that a concrete bike type has been selected, the configurator instantiates the underlying domain model `JavaBean` and is capable of invoking the methods `bike.getweight`, `bike.getImage` and `bike.getPrice`. Hence, the user interface displays the values returned by these methods accordingly:

- the initial weight is shown ❶,
- the domain model class renders the initial bike image ❷, and
- the price is displayed ❸.

Also note the addition of the "Motor & Battery" part in the configuration agenda ④. It has been added since the `ElectroBike` provides a `MotorBattery` typed property. This is as an extension to the set of parts collected from the `Bike` supertype (see Figure 6.2, "UML Class Diagram for the Bike Domain Model").

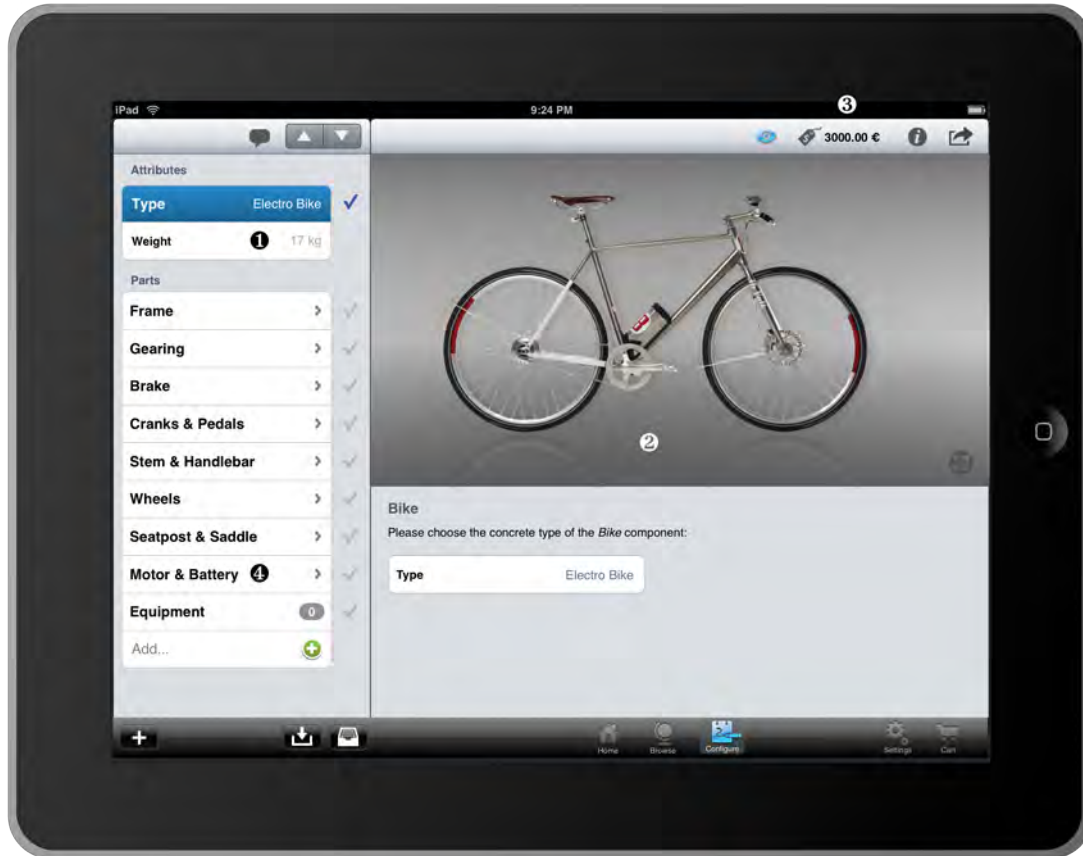


Figure 6.8. Configuration State after Type Selection

6.3.2. Configuring the Bike Frame Component

Figure 6.9, “Specification Dialog of the Frame Component” shows the specification dialog for the Frame component, a part with specification method `@Configurable`. While the user could proceed with the specification of an arbitrary part of the bike, he steps forward using the navigation control ❶ and selects the “Frame” component to be specified next.

The specification area shows the frame component's attributes and invites the user to specify the individual variables, which have been preset with default values ❷. Note that the configurator greys out any non-configurable attributes, such as the regular and calculated ones ❸.

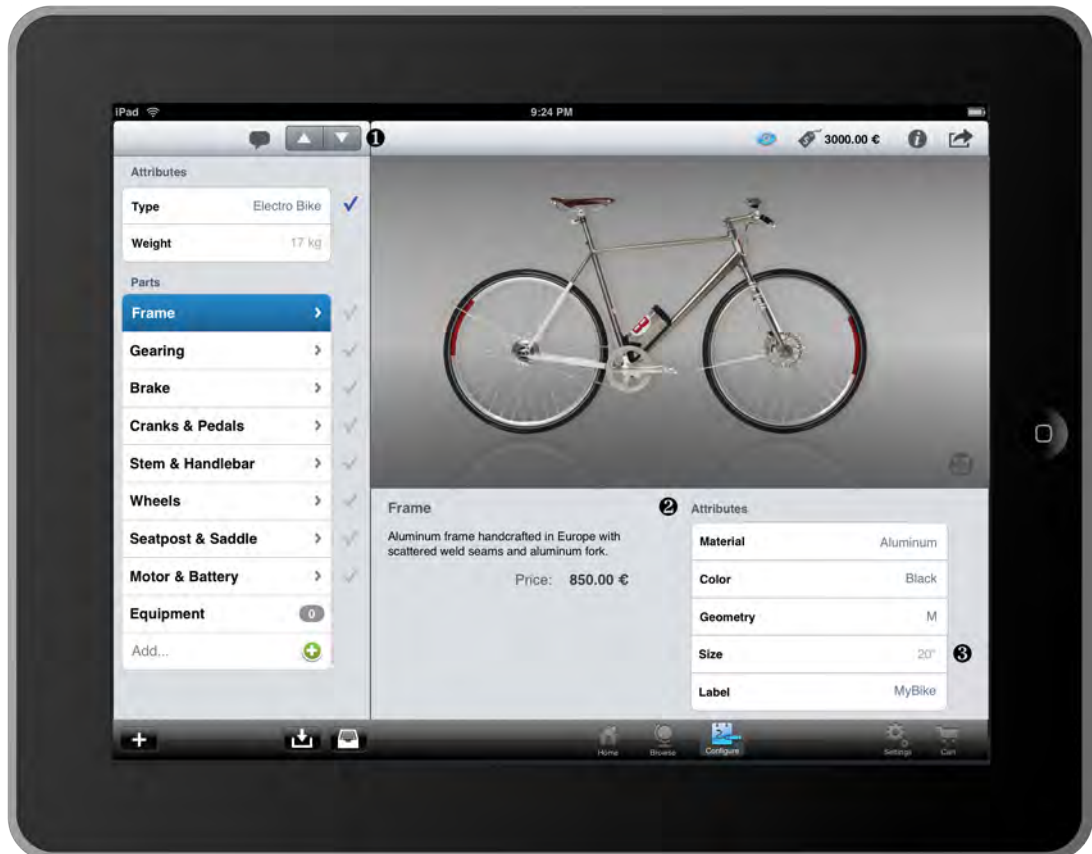


Figure 6.9. Specification Dialog of the Frame Component

6.3.3. Triggering and Resolving a Configuration Error

Let's assume the user entered an invalid value for the Frame component's label attribute: for instance, a value that exceeds the string length limit of 24 characters (as defined by the @Size(max=24) constraint annotation on the label property's getter method). The submission of the value triggers a constraint violation, which is detected immediately.

Figure 6.10, “Visualization of Constraint Violations” depicts the user interface in the faulty state:

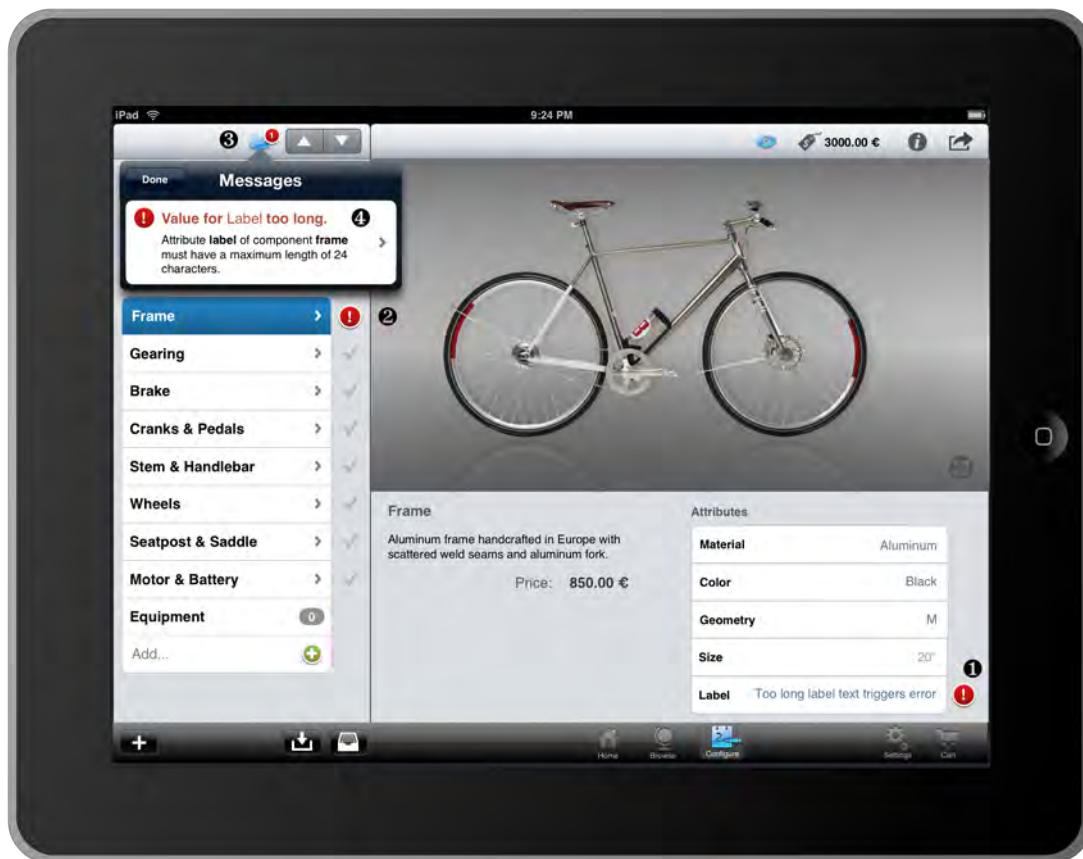


Figure 6.10. Visualization of Constraint Violations

The faulty attribute ❶ as well as the part containing the error ❷ are tagged with constraint violation symbols (exclamation marks). Furthermore, the message symbol in the navigation bar indicates that a validation error occurred ❸. A click on the symbol opens a pop-up dialog explaining the error in detail and provides the option to directly navigate to the faulty element ❹. The user can then correct the error accordingly.

6.3.4. Selecting Wheels

The wheels part of the bike is an example for a component with specification method @Selectable. For selectable components, the user must choose a particular tuple from the component's domain.

Figure 6.11, “Wheels Selection Dialog” shows the specification dialog for such a SelectComponentTask:



Figure 6.11. Wheels Selection Dialog

The configuration agenda indicates, that wheels is one of the few remaining, not yet specified components ❶.

The tuple domain of the wheels component is represented in tabular form ❷. Currently, the second row is selected and used as the bike's wheels instance ❸. Thus, the wheels part is specified and the user may navigate to any of the remaining unspecified parts.

6.3.5. Adding an Equipment

Next, we want to show, how plural parts, such as the bike's equipment part, can be specified within the generic configurator app.

Figure 6.12, “Plural Part Specification” shows the user interface's state after an equipment has been added:

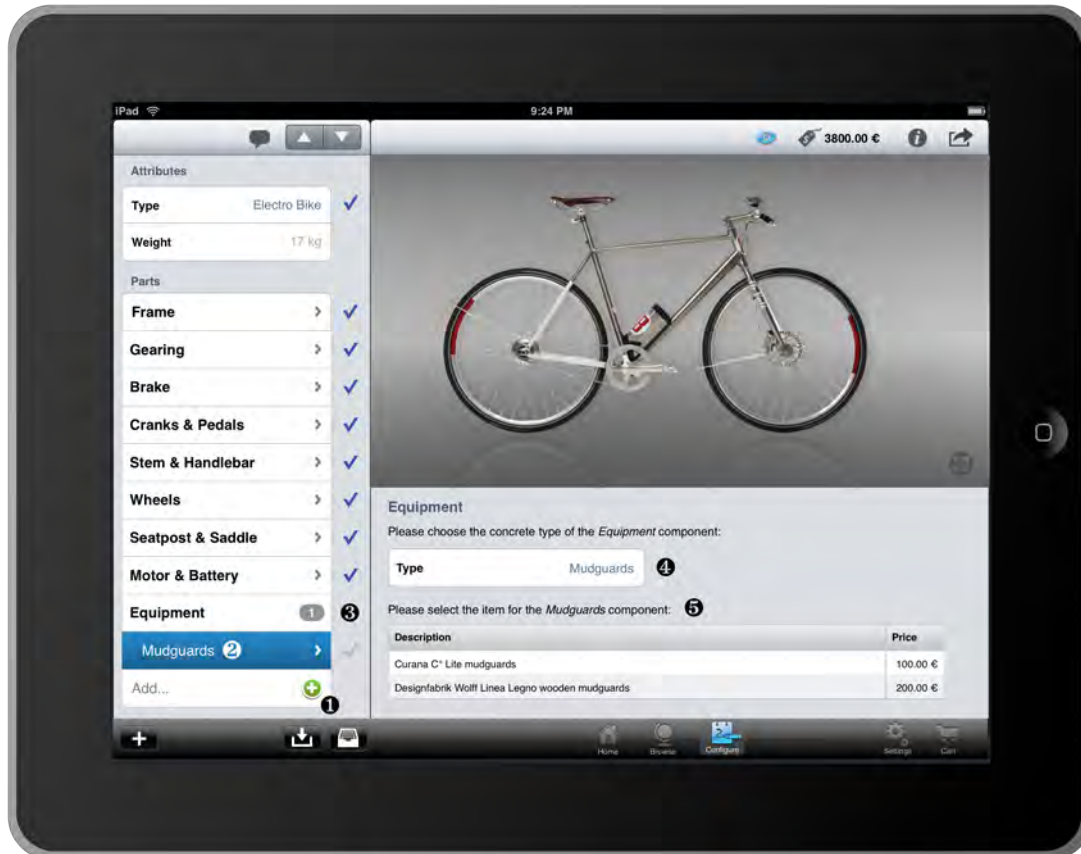


Figure 6.12. Plural Part Specification

First, the user clicks the "Plus" button to add another equipment item ❶. This action corresponds to the editing of the equipment part's related `SpecifyPartQuantityTask`. The yet unspecified equipment item is added to the configuration and the agenda is synchronized accordingly ❷. Also note that the quantity is shown next to the Equipment part element ❸.

Since the domain model knows three different Equipment subtypes, namely Mudguards, Light and Carrier, the user must specify the type of the newly added equipment item first ❹. Assuming he chooses Mudguards, the specification dialog is updated and the `SelectComponentTask` for the Mudguards component is rendered as a data table ❺.

After the selection of a particular item, the new equipment is fully specified and the configuration can be completed.

6.3.6. Submission and Checkout

Now, that all required items have been specified and no validation errors remain, the user may decide to order the configured item.

Figure 6.13, “Configuration Submission” shows how the user can submit the specified product for further processing.



Figure 6.13. Configuration Submission

After clicking the "Action" button of the configuration menu ❶, a pop-up dialog opens, displaying options on how to proceed with the current configuration ❷. As long as the configuration is valid, the user may:

- add the configuration to the shopping cart,
- view the specification of the configuration as a PDF¹⁷ file,
- export the configuration as XML file, or
- share the configuration via e-mail or other social platforms.

Assuming the user clicks the "Add to cart" button ❸, the configured product is put into the shopping cart, whereas the corresponding cart icon in the application menu indicates the number of added items ❹. From the shopping cart view, the user may checkout and order the configured items.

This ends up our generic configurator app walk-through. In the following section, we'll finally validate the overall configurator implementation approach from a high level perspective.

¹⁷Abbrev. Portable Document Format

6.4. Validation

OpenConfigurator's overall goal is to provide a framework, that eases the development of custom configurators. At this point, we want to reflect some outcomes of the implementation approach taken to implement the bike configurator for the iPad mobile device. We'll discuss the results more detailed in the following.

OpenConfigurator's modeling concepts are capable of representing real-world configuration problems. In this chapter, we've shown, that the modeling approach introduced by OpenConfigurator can be used to model a real-world configuration problem. Again, the modeling approach is mainly characterized by the object-oriented design and the provisioning of additional meta-data through source code annotations. For the development of the entire configurator application, solely plain Java knowledge as well as a rudimentary understanding of technologies like JPA and JAXB are necessary prerequisites. Due to this fact, we think, that the overall approach is easy to learn and relatively easy to apply in practical projects, even for non-expert Java developers.

Readable domain model, easily understandable by domain experts. Moreover, the OpenConfigurator approach fosters separation of concerns and allows to cleanly and flexibly model the configuration domain using the domain experts' vocabulary. Thus, the model ultimately remains understandable and manageable also by non-programmers. In the end, we think, that this ensures the maintainability of the system in the long run.

Rapid application development and tooling support. As for the realization of a custom configurator only the plain JavaBean based domain model needs to be developed, along with a one-time project setup effort, we argue that the overall development approach is extremely fast and efficient. Setting up a basic, running configurator application with a custom domain model and some constraints not even requires hours but instead can be done in just a few minutes. We believe, that the OpenConfigurator approach this way reaches the highest level of productivity compared to other configurator implementation methodologies, thereby, providing the same level of expressiveness and implementation strength if not more.

Additionally, the development could be speed up even more with a more sophisticated tool support. Nevertheless, already now, the built-in capabilities of modern integrated development environments (IDEs) can be fully utilized, when implementing configuration knowledge bases, including continuous syntax checking, code completion and highlighting, code templates, refactoring capabilities, UML modeling and code generation features as well as much more. Also, as the configurator's underlying domain model is plain Java, the source code can be shared and versioned within a source code management (SCM) system like Subversion¹⁸ or Git¹⁹. This enables team working possibilities out-of-the-box and integrates well with generally accepted project development processes employed today.

Configuration API enables implementation of generic configurators. The generic mobile configurator implementation shown in this chapter hypothesis, that the OpenConfigurator framework provided APIs are elaborate enough to implement generic configurator clients. In our opinion, we were able to demonstrate, that the introduced concepts and features support the implementation of usable, sophisticated user interfaces and greatly support the configuration of customizable products.

In summary, we argue that the OpenConfigurator methodology as well as the implemented framework are a quite practical, efficient approach to configurator implementation. Even more, we think that the framework addresses the claimed requirements and motivations for configurators in a suitable manner.

¹⁸See <http://subversion.apache.org>, last accessed July 19th, 2012.

¹⁹See <http://git-scm.com>, last accessed July 19th, 2012.

6.5. Summary

In this chapter, we evaluated the methodology introduced by OpenConfigurator with the help of a concrete case study. The case study targets the realization of a mobile bike configurator for the fictitious company MyCustomBike Inc..

After shortly motivating the example by describing MyCustomBike Inc.'s business scenario, we defined the company's product range, which was subsequently modeled using an UML class diagram. The developed Java source code (see Appendix B, *Example Domain Model: Bike*) practically demonstrated, how the conceptual elements, worked out in Chapter 4, *Methodology and Conceptualization*, can be used to model the domain of customizable bikes. Thereby, we covered many OpenConfigurator provided modeling concepts.

Next, we introduced the generic mobile configurator client, that was developed as part of the evaluation. Although implemented as a regular HTML, CSS and JavaScript based web application, the generic configurator is meant to be run on Apple's iPad mobile device primarily. We placed the realized configurator client, featuring a user interface based on the Vaadin technology, within our framework architecture and this way completed the overview from Section 5.2, "Architectural Overview". This section closed by describing the application's user interface more precisely.

Then, we walked through an example configuration procedure and presented these in detail. The individual configuration steps were accompanied by various screenshots of the final application.

Finally, we discussed the overall realization of the case study and validated, whether and in what manner, the approach addresses important practical aspects relevant in configurator development projects.

In the next, final chapter, we will conclude this work and provide an outlook on future development areas.

7

Summary and Outlook

To conclude, we want to discuss the results of this work by comparing them to the earlier stated requirements and motivations. Moreover, we will finally characterize our approach and provide our opinion on various strengths and weaknesses of the methodology.

Before closing up with a short resumé, we will present some ideas for further investigation and developments, that might be subject to future works.

7.1. Discussion

In this section, we want to critically reconsidering our conceptualization in its entirety. In this context, we want to examine the following questions:

- **Methodology suitability.** Do the conceptual elements of the developed OpenConfigurator framework allow to model customizable products adequately?
- **Implementation characterization.** How does our implementation compare to others?

Let's begin by discussing the first question.

7.1.1. Methodology Suitability

In Section 2.1.3, “Customized Products and Product Configurations” we identified the characteristics of customizable products in detail. Based on an in-depth literature research, we worked out three different product types, some general and eight specific areas of customization for defining configurable products. Again, we want to point out, how OpenConfigurator's modeling concepts address these customization options concretely.

First, the three **different product types**, namely *supplier-oriented*, *customer-centric* and *customer-oriented products*, can be realized with OpenConfigurator concepts as follows:

Supplier-oriented products. Are interpreted as products or components, which cannot be adapted by the customer himself. In terms of our conceptualization, they can be modeled

using `@Selectable` product classes. During the configuration process, a component class annotated alike can solely be specified by the customer through the selection of a particular product variant, respectively by selecting a specific instance of the corresponding class.

Customer-centric products. Provide certain customization options with a restricted scope. In our conceptualization, these products are modeled as `@Configurable` product classes, which may consist of arbitrary `@Configured` parts and `@Variable` or `@Parameter` annotated attributes. The customization options can be limited *ex ante* (that is, prior user decision) using `@Domain` annotations or *ex post* with the help of constraints.

Customer-oriented products. Are freely customizable and correspond to `@Constructible` product types in our approach. As the configured variants strongly vary and the number of different attribute value combinations is virtually unlimited, there are no concrete variants maintained in the supplier's database. Instead the products are modeled as parameterized entities. Hence, such product classes primarily contain `@Parameter` annotated attributes.

In our conceptualization the **general customization options**, namely *functionality*, *fit* and *design*, are modeled using object-oriented concepts, that is, using classes and relationships such as association and generalization.

The specific **areas of customization**, as defined in Section 2.1.3.1, “Customizable Areas”, can be expressed as follows:

Example 7.1. Modeling Customizable Products with OpenConfigurator Concepts

```
@Configurable
@Product
@Domain.Query
public class CustomizableProduct ❶
{
    private FixedArea regularPart; ❷
    private String regularAttribute; ❸

    @Required
    @Configured
    private ObligatoryElement constrainedConfiguredPart; ❹

    @Optional
    @Variable
    private OptionalElement variableAttribute; ❺

    @Parameter
    @Max(100)
    private int parametricAttribute; ❻

    @Configured
    @TypeDomain({ SolutionA.class, SolutionB.class })
    private Principle configuredAbstractTypedPart; ❼

    @Configured
    private Service configuredPart; ❽

    @Configured
    private Shape constructiblePart; ❾

    // ... getters & setters
}
```

```

@Selectable
public abstract class Principle { ... }
public class SolutionA extends Principle { ... }
public class SolutionB extends Principle { ... }

@Constructible
public class Shape
{
    @Configured
    private Collection<Vertex> vertices;

    // ... getters & setters
}

@Configured
public class Vertex
{
    @Parameter
    @Max(1000)
    private int x, y;

    // ... getters & setters
}

```

Customizable products. Are basically modeled as classes ❶. In terms of OpenConfigurator a product is treated as a composite component. The concept of a customizable (top-level) product is distinguished semantically from (inner) components solely by annotating them with @Product.

Fixed areas. Are modeled as regular, invariable parts ❷ or attributes ❸, that are valued during component tuple selection. The user cannot specify these regular parts or attributes explicitly.

Obligatory and optional elements. Are modeled as configured parts, respectively variable attributes, accompanied by either a @Required ❹ or @Optional ❺ constraint annotation.

Scalable areas. Can be expressed using parametric attributes, defined using the @Parameter annotation ❻. Mostly, these attributes' values are restricted by constraint annotations in order to limit the solution space.

Principle solutions. Are meant to be defined using the object-oriented concept of generalization. The specification of a principle solution corresponds to the selection of a part component ❼. Hence, the principle is modeled as an abstract class, which is implemented/extended by the different, feasible solutions. Modeled this way, the choice of a concrete solution corresponds to a type decision. The options available for this type decision, in turn, can be controlled with type domain annotations.

Services. Are modeled as regular @Configured components.

Defined expansion spaces. Can also be expressed with the help of @Constructible components. Imagine the customer shall be able to define a custom shape as part of the configuration process. To model this, a configured part is defined in the model ❽. The part is of type Shape, which is designated as @Constructible. The shape itself consists of an unlimited number of configured vertices, which itself define their coordinates as parametric attributes.

For the specification of such a custom constructible component in a convenient way, a special editing control would have to be implemented in the client application, though.

General expansion spaces. By definition, *general expansion spaces* cannot be expressed using *any* formal approach. Otherwise they could have been described as defined expansion spaces. Thus, OpenConfigurator doesn't aim to provide any support for these.

Ultimately, in the sense of OpenConfigurator, different variants of a customizable product correspond to different instances of the configurable class.

The aforementioned elaborations show, that in general, customizable products can be modeled using OpenConfigurator's conceptual elements.

7.1.2. Implementation Characterization

In Section 3.3.3, “Categorization”, we compiled a comprehensive taxonomy for the characterization of configuration methodologies. To make the OpenConfigurator approach comparable to other methodologies, in the following, we will characterize the framework regarding the defined categories.

Figure 7.1, “Classification of OpenConfigurator within the Morphological Box for Configurators” shows a morphological box highlighting OpenConfigurator's capabilities and possible extension areas. Note that we also take the prototypically implemented generic mobile configurator client from Chapter 6, *Evaluation and Validation* into consideration for the purpose of this categorization.

Application context. Regarding the *application context*, the OpenConfigurator's modeling approach can be used to model immaterial as well as physical products for both B2B and B2C markets. Depending on the complexity, the so developed configurators can be used for internal purposes and external sales purposes simultaneously. Due to some missing features, such as more elaborate help and recommendation capabilities, as well as the missing possibility to customize the configuration process, especially for complex products, the configurator is considered to target expert users (staff members) only. Moreover, at the current stage, the framework is considered to be used in the context of sales activities, although engineering and production support are intended and foreseen extension possibilities. Regarding sales support, the framework is currently meant to supplement other tools, as the shopping cart functionality is not yet part of the implementation. Related to the underlying order-fulfillment process, we claim that with OpenConfigurator custom configurators for any strategy can be implemented without compromise.

System environment. Related to the *system environment*, the OpenConfigurator framework can be characterized as follows: at the current stage, the framework has only been used in thin, web-based configuration clients. However, in principle, offline support is available as well, though no such application has been realized so far. For the web-based implementation, the data is stored on the server, effectively implementing a centralized approach. The resulting configurators, built with the framework, can be used standalone and the framework provided APIs and SPIs can be used for integration scenarios, where the configurator acts as both data provider and consumer. The implemented prototype targets mobile devices, but is otherwise implemented as regular HTML5 application, which allows the configurator application being accessed with a regular web-browser, too (although with some look & feel shortcomings). With some layout customizations, the configurator could also easily be adapted for mobile phone use.

Application Context										
Product nature	immaterial product					material product				
Business area	business-to-business (B2B)					business-to-consumer (B2C)				
Site of operation	internal					external		web		
Target audience	staff					customers				
Integration context	sales		development/engineering			production		hybrid		
Sales support	supplement					replacement				
Underlying order-fulfillment strategy	assemble-to-order			build-to-order			develop-to-order			
System Environment										
System interaction	offline					online				
System organization	centralized					thin client		fat client		
Integration scenario	none/standalone			embedded			integrated			
Integration level	not integrative			data integrative			data and application integrative			
Target device	desktop computer/laptop			touch device/tablet			mobile phone/smart phone			
Modelling Capabilities										
Modeling capabilities	fixed areas	obligatory/alternative elements	optional elements	scalable areas	principle solutions	services	defined expansion spaces	general expansion spaces		
Level of detail	limited					unlimited				
Product model evolution support	none			limited			full			
Configuration Characteristics										
Configuration Approach										
Specification approach	product-centric/structure-oriented			customer-centric/function-oriented			hybrid/combined			
Configuration capabilities	module configuration (generic modularization)			mixed configuration (quantitative modularization)		adaptive configuration (custom modularization)		design configuration (free modularization)		
Interaction Characteristics										
Configuration procedure	batch					interactive				
Process scheme	pre-defined/fix	structure-oriented		user-oriented		data-driven		case-driven	adaptive	
Dialog style	text based					graphical				
	questionnaire		forms							
		single screen	screen sequence		hierarchical					
Configuration Procedure										
Starting point	blank configuration			pre-defined base configuration			custom existing configuration			
Solution strategy	bottom-up					top-down				
Option display/consistency enforcement	unrestricted, no consistency enforcement			limited restriction, partial consistency enforcement			fully restricted, strong consistency enforcement			
Validation	none		singular, at configuration end			repeatedly		continuously		
Specification support	active		passive / re-active							
	defaults		automatic completion		help		recommendations			
	non-personalized		personalized		context-insensitive		context-sensitive			
Presentation										
Configuration description	textual/tabular			graphical			both			
Visualization	static/primitive					dynamic				
Degree of interactivity	none					limited		extensive		
Point in time of visualization	delayed					real-time, continuously				
Rendering method	pre-produced images		compound images		rendered images/graphics		virtual reality scenes			
Level of detail	low					high				
Price calculation and display	simple					complex				
	no calculations/accumulations			calculated/accumulated		total prices		price premiums		
						both/combined				
Implementation Aspects										
Implementation approach	custom development		application module		standard software		refinement (reference system)		platform/framework	
Universality	specialized					universal				
Configuration approach	implicit techniques			explicit techniques / model based				combined techniques		
	decision-tree based	rule based	case based	decision-table based	resource based	constraint based	component based	object-oriented	knowledge based	
								hybrid		
Applicable With extension Not applicable										

Figure 7.1. Classification of OpenConfigurator within the Morphological Box for Configurators

Modeling capabilities. As shown in the Section 7.1.1, “Methodology Suitability”, regarding the *modeling capabilities*, the conceptualization of OpenConfigurator supports all important concepts. It allows to model configuration problems with an unlimited level of detail by nesting an arbitrary number of parts. Regarding product model evolution, the approach suffers from the same problems as other compiled domain models do: a change of the product structure requires recompilation of the model and a redeployment of the application [Künzle2011]. Moreover, in special cases, data transformations must be applied to accomplish model changes. Nevertheless, OpenConfigurator's approach allows to dynamically alter the configuration behavior by modifying the underlying knowledge base, which is accessed by the data provider [Künzle2011]. For example, adding another tuple to a particular component domain solely requires to insert the new variant into the corresponding database table. The appropriate configuration dialog controls are automatically updated. This allows straightforward maintenance of the configurator for the majority of changes. We, thus, consider OpenConfigurator's domain model evolution capabilities as limited.

Configuration characteristics. Next, we'll discuss the *configuration characteristics* supported by OpenConfigurator, starting with the *configuration approach*: currently, the framework singularly supports product-centric, structure-oriented configuration, while a function-oriented approach is considered a feasible extension of the system in a later version. In general, the OpenConfigurator's core component model can be accompanied with arbitrary meta-data annotations and facets, which easily allows to extend the conceptualization with additional modeling concepts for the realization of functional or resource-oriented semantics. Moreover, we argue that the model in its current state already supports any type of modularization and configuration types by offering different specification methods (namely selection, configuration and construction).

Regarding *interaction characteristics*, OpenConfigurator (respectively the prototype client) implements an interactive, structure-oriented, adaptive configuration process: the configuration agenda, which allows to navigate through the configuration process sequentially or randomly, reflects the product structure and adapts itself upon model changes. For instance, depending on the domain size of a particular part, the agenda schedules certain decisions or marks them as obsolete. These adoptions, however, could still be improved in a more user-centric or data-driven manner in a future version of the framework. The currently implemented dialog style can further be characterized as textual, form-based and hierarchical, whereas a sequential navigation can be performed using the corresponding navigational controls. In general, it's imaginable the framework's API is also used to back graphical configuration processes, but this needs to be proofed in some future work though.

The *configuration procedure* realized by an OpenConfigurator based tool can be designed flexibly: as starting point, a blank (no default annotations required), a pre-defined (default annotations required) or an existing configuration can be used. Depending on the usage of default annotations, one can consider the implemented procedure as top-down oder bottom-up approach.

Regarding option display, OpenConfigurator currently features a partial consistency enforcement strategy. Respectively, the framework can be considered to implement limited domain restriction capabilities: with domain annotations, the value space of an attribute's or component's domain can be limited a priori. However, currently, constraint annotations are *not* considered during the construction of value domains, but instead only used to validate user input ex post. Hence, constraints are only relevant, *after* the user selected a particular domain value.

Implementing strong consistency enforcement is one of the most important areas of investigation for future developments of the framework (see Section 7.2, “Outlook”). Limited consistency enforcement strategies reduce the overall user experience, because some specification errors are not avoided upfront. However, our framework aims to detect constraint violations as early as possible by continuously performing model validation. This way, a user's

decision is immediately validated upon value submission and specification errors are reported promptly.

OpenConfigurator currently also lacks support for sophisticated help and recommendation features. Their implementation is subject to future work as well. Nevertheless, with defaults and automated configuration completion, the framework provides basic, active, yet not-personalized specification support.

Related to *presentation capabilities*, OpenConfigurator features a textual, tabular configuration description. The rendering of graphical visualizations is currently left to the domain model types themselves, which may provide a `@Product.Asset` annotated method returning arbitrary image information. In this area, there's certainly lots of room for improvement and the framework will likely provide at least some utilities to implement more advanced rendering features in future release. The same counts for price calculation capabilities, which is currently implemented on a very basic level, too: prices need to be calculated by the domain model instances themselves, within an `@Product.Price` annotated method.

Implementation aspects. Finally, the framework can be compared to others in terms of *implementation aspects*. Clearly, as the name "OpenConfigurator framework" already indicates, the implementation approach can be characterized as platform/framework based. Moreover, OpenConfigurator supports arbitrary annotated domain models and can thus be considered generic and universal. Regarding the configuration approach, the framework can be categorized as a model based, combined approach, featuring fully object-oriented, component based and partly constraint based techniques. We do not yet consider our approach as fully constraint based, since the configuration process doesn't currently involve the interaction with a dedicated constraint solver. As stated above, this integration is subject to some future work (see Section 7.2, "Outlook" for details).

The characterization of our configuration methodology within the morphological box for configurators helps to measure the status and direction followed by the OpenConfigurator implementation. Moreover, it displays substantial areas of improvement for future works. We will discuss two of such potential future research areas in Section 7.2, "Outlook", namely the integration of constraint solving techniques and the integration of recommendation features into the OpenConfigurator framework.

7.1.3. Strengths and Weaknesses

Finally, we want to critically reconsider the strengths and weaknesses of the OpenConfigurator methodology. We'll first discuss the most relevant strengths from our point of view and then explain potential weaknesses. For both parts we differentiate between the following aspects:

Conceptualization. Items that relate to the general conceptual approach taken by OpenConfigurator.

Technical. Issues that relate to the practical realization of configurators with the methodology.

Implementation. Describes issues that result from the way the methodology is implemented.

Our argumentation is build up as follows: for each issue, we mention the fact or claim and discuss the consequence to it.

7.1.3.1. Strengths

We argue that the most important strengths of our methodology are:

Conceptualization

Unified paradigm & accepted technologies. Simply said, "Everything is Java": solely building on the Java platform and companion technologies, our methodology is constructed around a single, homogeneous technology stack. The conceptual framework as well as the implementation are designed to integrate well with existing Java technologies and concepts. Object-oriented programming, plain Java, annotations, Java EE technologies such as JPA, CDI and Bean Validation are keywords, that characterize this aspect.

These generally accepted techniques and concepts are a fundamental to the OpenConfigurator framework and influence its ease of learning capabilities significantly.

Model based & domain specific modeling. The conceptual approach is entirely model based and supports domain specific modeling. The fully object-oriented domain models are implemented using Java's strong type system and enriched with source code annotations. This enables build-time error detection. While universal, graphical modeling tools like UML suites are available and applicable to our methodology already, improved support for graphical configuration domain modeling and validation is not yet implemented, though.

Nevertheless, the domain specific modeling capabilities provided by our conceptualization allows domain experts to express configuration knowledge in the target domain's language and using their own vocabulary. We argue that this eases development and long term maintainability dramatically.

Separation of concerns: configuration problem & solution process. With the help of declarative constraints and other meta-data added through annotations, our methodology enforces a clear separation of concerns: the configuration problem knowledge is designed completely independent from the solution process, that is, the concrete configuration procedure. Moreover, the configuration model designer is not even concerned with the algorithm, that determines how the configuration problem is solved during the specification process, at all, but instead solely describes the available solution space. The framework fills the missing gap, by automatically transforming the configuration domain model into a corresponding configuration process.

The consequence of this is a strongly simplified development procedure and yet again an improved maintainability.

Practice oriented. Our conceptualization builds on object-oriented principles, a methodology which is generally accepted being suitable to model real-world use cases compactly, precisely and well understandable. Moreover, our approach allows to define specific configuration knowledge with both intensional and extensional concepts (see Section 4.4.4.1, "Domain Definition"). We argue that this way our conceptualization is capable of capturing practical configuration problems adequately and that it is flexible enough to meet a particular project's requirements exactly.

After all, this practical orientation of our methodology makes the development of configurator applications a straightforward, simple process. Again, maintainability benefits from this fact, too, since additional product variants can be added without code modification once the base product model has been established.

Technical

Highly efficient. Modeling and implementing real-world product models with object-oriented principles is in most cases an uncomplicated issue. Due to the fact, that with our methodology, the development of custom configurators boils down to the task of implement-

ing the domain model, we see our approach as a highly efficient one. An example has been demonstrated in Chapter 6, *Evaluation and Validation* with the bike domain.

The high productivity leads to strongly reduced development costs and shortened implementation times, thereby, increasing the overall development experience greatly.

Development friendly. Moreover, we argue, that our approach is tooling-, testing- and debugging-friendly. For Java a huge variety of development tools are available, such as intelligent IDEs¹ or test environments. These tools provide powerful features for the development, refactoring, testing and debugging of Java applications and domain models. Since everything in OpenConfigurator is plain Java, all these features can be utilized during configurator development, too. For example, this allows to create regular JUnit² regression tests for verifying correct configuration behavior continuously or enables the usage of the IDE's debugger to find bugs within a configuration. That said, the OpenConfigurator framework currently doesn't provide a dedicated test harness, which could be used to simplify the implementation of configuration tests. This feature will be addressed in a future version.

With OpenConfigurator, developers can use the tools and techniques they're already familiar with. They do not have to learn new ones. Ultimately, this eases learning and development greatly, enhances maintainability and shortens the overall project development times in the same way.

Implementation

Universal. The OpenConfigurator framework is generic. That means, it is developed entirely domain independent. Even more, the framework can be considered minimal intrusive as it doesn't impose any particular structure on the domain model. The fundamental enablers for this to work are Java annotations, that provide all necessary meta-data to configure arbitrary domain models.

Consequently, the OpenConfigurator framework can be used for a huge variety of use cases, which makes learning the technology even more attractive, when applied for multiple projects.

Flexible & extensible. Furthermore, the implementation of the framework builds on modern technologies, most notably CDI³, which fosters a highly flexible, loosely coupled architecture inherently.

We claim that this strongly eases the customization of our software in particular usage scenarios and makes the system both extensible and future-proof in the long run. This, in turn, decreases the overall investment risks greatly.

Developer friendly. Finally, we argue that the OpenConfigurator's provided type-safe APIs are comparably easy to understand and that they succeed in hiding the general configuration domain's complexity.

We believe, this lowers the entry barrier into the complex topic of configurators for mainstream Java developers extremely. In fact, it unleashes new potentials for the application of these powerful sales tools.

¹Abbrev. Integrated Development Environments

²See <http://www.junit.org/>, last accessed July 22nd, 2012.

³Abbrev. Contexts and Dependency Injection, see Section 5.1.4, "Contexts and Dependency Injection for Java EE (CDI)".

7.1.3.2. Weaknesses

Although OpenConfigurator's methodology and implementation has been researched in detail, the novelty of our approach and time constraints lead to some shortcomings, that should be addressed in future works. We will explain these issues from our perspective and discuss their consequences in the following.

Conceptualization

High level of abstraction. Building on object-oriented principles and solely requiring developers to define configuration domain models with declarative constraints poses a high level of abstraction from the actual developer's job: implementing a configurator. While this strong abstraction level is a fundamental enabler for OpenConfigurator's ease of use, ease of learning and its overall productivity, the developer might feel uncomfortable with this approach by presuming that it lacks developer control and that too much "magic" happens in the background.

In fact, the framework rigorously frees the developer from tedious, difficult tasks. Thereby, it imposes a currently fixed algorithm, that controls the configuration process. In a future version of the framework, we might provide utilities and features, that allow more control over the configuration process, e.g., a declarative configuration process definition or screen flow.

Risk to scatter knowledge. A key concept of our methodology is the declarative definition of configuration constraints using annotations, which are directly applied on source code level. One might argue, that spreading the configuration logic across the entire domain model, instead of defining rules and constraints in a single place, scatters the knowledge.

While we personally strongly favor this concept, postulating a "single source of truth" approach (the domain model), we agree that alternative ways to define configuration constraints can be beneficial in certain situations. For example, when the configuration domain model gets very large and a huge amount of constraints need to be incorporated, it may certainly be useful to separate the "business rules" from the "business model". Therefore, the support for alternative meta-data sources for constraint knowledge, specifically using XML, are planned for a future version of the OpenConfigurator framework.

Technical

Addresses developers. Our methodology allows to define configuration problems using Java domain models. Without dedicated tool support, which is not currently present, the task of engineering a domain model for the OpenConfigurator framework, is still to be executed by programmers rather than end-users without programming experience. This means, that creating a configurator using our methodology, cannot be performed using a WYSIWYG⁴ style, rich user interface yet. Instead, a developer writes source code and compiles the domain model.

Nevertheless, we argue, that *writing* a configuration domain model is a lot faster compared to *clicking* it, especially, when it comes to define complex structures and constraints. The exceptionally good tool support for writing source code, as provided by state-of-the-art development environments, is a strong reason for this. Moreover, we believe, that the possibility to verify the correctness of the configurator at build time using regression tests, is inevitable for complex application domains. Assuming that a test harness utility provided by OpenConfigurator is available at some point in the future, the possibility to implement configuration tests and to let them run automatically by a continuous integration (CI) system, pays out quite fast.

Limited real-time knowledge base modification support. The fact that OpenConfigurator currently requires coded, compiled domain models to operate on, has a technical draw-

⁴Abbrev. What You See Is What You Get

back: structural changes to the product model cannot be incorporated dynamically at run-time. Instead, a redeployment of the model has to take place.

However, there are two things to note about this issue: first, the implementation of dynamic reloading functionality for domain models is feasible yet quite tedious. This is why this feature has been left to be implemented in some future work. Second, the configurator's knowledge base *can* be changed dynamically at runtime, but changes are limited to the data backing the configuration model. Structural changes, e.g., like adding an attribute or removing a constraint are currently impossible without application redeployment/restart. We believe, this is an acceptable limitation for the moment, but must be addressed in a future version of the framework.

Implementation

Partly incomplete. Possibly the most important issue of OpenConfigurator is, that it's still in the very early stages: the implementation is partially incomplete. While the core concepts have been realized, some specific features haven't been implemented yet. Following an exemplary list of outstanding issues:

- **Missing features.** The configuration process cannot be customized yet. Instead, it is generated automatically. Also, more sophisticated visualization support and additional price calculation methods may be implemented.
- **Missing conceptual elements.** A dedicated concept for function-oriented configuration is still missing.
- **Missing constraints.** For example, an *all-different-type* constraint, that implements semantics like "all elements of a collection must have a different type" hasn't been implemented yet.
- **No constraint solver/true inference component integration.** Currently, OpenConfigurator doesn't transform the configuration problem into a true, classical CSP⁵ solved by a dedicated constraint solver. Thus, constraint propagation isn't currently implemented fully.

Particularly, the last issue is an important one, since currently we cannot guarantee, that a specific decision sequence taken by the user, results in a valid configuration⁶. While our implementation detects the inconsistency and forbids to submit a configuration containing errors, the user could have been prevented from taking such decisions in advance. By the integration of a true constraint solver component, this usability deficit could be avoided. We will outline a possible solution to the integration of a constraint solver in Section 7.2, "Outlook".

Granted that there exist some shortcomings, we strongly believe that the advantages of our approach outweigh its drawbacks by far. Though, we also accept that different approaches may be beneficial in other scenarios. Finally, the viability of the framework in practical projects has not yet been shown on a grand scale.

7.2. Outlook

Finally, we want to point out some possible future development areas of the OpenConfigurator framework. We will first present some basic ideas and then take a look at two concrete issues, namely the integration of constraint solvers and recommendations.

In the current status, OpenConfigurator provides a basic foundation for developing custom configurators. We evaluated various literature sources and theories behind product customization and compiled a core model for capturing and solving configuration problems.

⁵Abbrev. Constraint Satisfaction Problem

⁶See Section 3.3.3.4, "Configuration Characteristics", "Option display/consistency enforcement" for details.

Throughout the work, we showed various areas for further research and development:

Features. In Section 3.3.2, “Features”, we gave an extensive, feature-based overview of configurators. While our prototypical generic mobile configuration client introduced in Chapter 6, *Evaluation and Validation*, can already be considered a useful tool, many features discussed theoretically haven't been implemented yet. Thus, depending on concrete requirements, one could extend the OpenConfigurator implementation with specific functionalities provided by this section.

Classification. Moreover, in Section 3.3.3, “Categorization”, we discussed various axes that can be used to approximate the capabilities of a configurator. In Section 7.1.2, “Implementation Characterization”, we then characterized our approach with regard to the given categories. This characterization showed the framework's current capabilities and provides concrete ideas for further investigation.

We believe, that the following three topics are the most prominent ones, that should be addressed in future works:

- **CSP/Constraint solver integration.** As already mentioned in Section 7.1.3.2, “Weaknesses”, the transformation of the object-oriented configuration model into a CSP and its resolution using a dedicated constraint solver, would allow to predict resolvability of a configuration problem. We will investigate this topic in Section 7.2.1, “CSP/Constraint Solver Integration”.
- **Recommendations.** Incorporating automatically generated, personalized recommendations into the configuration process would be a valuable feature from a business point of view. Full featured recommendations aren't yet standard for today's configuration systems, which is why such a feature could stand as a unique selling point (USP) for our configuration solution. We will describe an approach to recommendation integration, as researched by Renneberg in [Renneberg2010], in Section 7.2.2, “Recommendation Integration”.
- **Visualizations.** Another valuable extension to our current configuration system are advanced visualization capabilities. Especially for end user facing product configurators, the visual appearance and aesthetically appealing product presentation are very important success factors [Reichert2010].

Let's discuss the first two ideas in more detail.

7.2.1. CSP/Constraint Solver Integration

Constraint programming (CP) is an established research topic in the larger domain of artificial intelligence (AI). Describing and solving constraint satisfaction problems (CSPs) has a long history and has been widely discussed in the literature⁷. The application of CSPs in the area of configuration was also subject to many research efforts in the last decade and is still subject to research today⁸.

A *constraint satisfaction problem (CSP)* can be formally described as a triple (V, D, C) , where V is a set of variables, D a set of domains associated with the variables and C a set of constraints. A *constraint* is understood as a relation over a set of variables. A particular valuation of a constraint's variables using values from the variable's domain can either satisfy or violate the constraint.

The task of a *constraint solver* is to find an assignment of all variables, so that all defined constraints are satisfied. The resulting assignment is called a *solution* to the CSP (see also

⁷See, for instance, [Kumar1992], [Barták2005].

⁸See [Stumptner1997], [Sabin1998], [John1999], [Veron1999], [Felfernig2001], [John2002], [Aldanando2002], [Li2005], [Schneeweiss2011].

[Runte2006, pp. 50]). There are many techniques and algorithms known to efficiently find solutions to CSPs, one of them being *constraint propagation* (see [Tack2009]).

Basically, configuration problems match the semantics of CSP very well: within a configuration process a customer takes decisions on various *variables*, while choosing the available options from their *domains*. The decisions taken by the user are further restricted by a set of *constraints*.

However, in reality the mapping between configuration problems (in our sense) to traditional CSPs is more complex, due to (cp: [Ilog2001], [Brown2008]):

- **The structural nature of our configuration models.** *Composition, generalization and instantiation* relations are concepts not known to traditional CSPs.
- **The dynamic nature of configuration models.** In our configuration approach, the set of variables and constraints may dynamically change at runtime depending on the user's decision.

While specific variants of CSPs exist to alleviate these problems, such as *Structural Constraint Satisfaction Problems (SCSP)* [Nareyek1999] or *Dynamic Constraint Satisfaction Problems (DCSPs)*⁹ exist, the application of those approaches in practical configurator applications is still a complex task.

An approach to the integration of constraint solvers into configuration systems is described by Runte in [Runte2006]. Runte not only presents a comprehensive collection of background information on constraint solving, but also describes a concrete architecture to apply multiple constraint resolution strategies to configuration problems.

OpenConfigurator has been carefully designed with the application of constraint solving strategies in mind. However, time constraint didn't allow us to integrate a particular constraint solver into the system yet. In our opinion, such an extension, perhaps in combination with the recently approved JCP standard "JSR-331: Constraint Programming API"¹⁰, would be an interesting and powerful addition to our framework. This way, established algorithms to find correct solutions of constraint related problems could be incorporated. Moreover, in the end, if OpenConfigurator would be accompanied by a constraint solver under the surface, the complex domain of constraint programming could be made accessible to the mainstream Java developer transparently.

7.2.2. Recommendation Integration

In his dissertation, Renneberg describes an adaptive, modular recommender system for generic product configurators, which can be used as an advisory component, complementary to the configurators core functionality [Renneberg2010].

Discussing various personalization¹¹ techniques, Renneberg identifies *automatically generated recommendations* as effective utilities for successfully addressing common problems faced by customers during the highly sales relevant product configuration phase. These problems include excessive demands due to the high degree of choice (also referred to as "mass confusion"¹²) and the lack of preferences of the customer related to certain product attributes (cp. [Scheer2006, p. 45-47]). Additionally, the overall complexity of the product or the missing domain knowledge of the customer can be mentioned as reasons for customer initiated purchase cancellations. All these could be reduced by recommendation techniques.

Whereas recommender systems are an actively discussed topic in computer science, which are widely employed in real world application domains (especially on e-commerce sites,

⁹See http://en.wikipedia.org/wiki/Constraint_satisfaction_problem#Dynamic_CSPs, last accessed July 23rd, 2012.

¹⁰See <http://jcp.org/en/jsr/detail?id=331>, last accessed July 23rd, 2012.

¹¹Scheer describes "personalization" as *customer oriented interaction* [Scheer2006, p. 12].

¹²cp. [Teresko1994, p. 48]

such as Amazon.com¹³), their application in commercial configurator systems is limited and not widespread. The reason for this, according to Renneberg, is the dynamic nature of configurators as well as their inherent complexity: static, singular recommendation techniques do not sufficiently satisfy the requirements for configurator systems.

Based on the motivations above, Renneberg describes an *adaptive* recommendation extension for generic, model-based product configurators in the context of e-commerce, that is capable of automatically generating recommendations with the help of so called *filter modules*. These filter modules can implement a huge variety of recommendation techniques based on collectively aggregated usage data and user profiles. Examples of filter modules include:

- history based recommendation algorithms, which evaluate the customer's historic purchase orders,
- collaborative algorithms, that evaluate the entirety of purchases from all customers, or
- optimizing algorithms, which generate their recommendations based on, e.g., the components' prices.

In order to implement the adaptiveness and to support *hybrid recommendation techniques*, that is, the combination of multiple recommendation strategies, filter modules can be flexibly composed in terms of templates. Renneberg also describes the execution environment for these templates, the so called *ReCoKit*¹⁴ component, along with its interface to the configurator, in detail.

The OpenConfigurator framework has purposely been designed to be complemented by features such as the recommendation solution described above. We argue, that the sophisticated configuration model employed in the framework, fulfill the requirements stated in Renneberg's work (cp. [Renneberg2010, pp. 80, 150-152]) and that the integration of ReCoKit is possible. Moreover, we think, that the OpenConfigurator framework would considerably profit from an advisory component. The ultimate feasibility of the integration of Renneberg's approach has to be proven in a future project.

7.3. Conclusion

Product configurators are powerful sales and design tools, that offer manifold chances and benefits for enterprises, pursuing product customization. Though, configuration systems are complex software applications and their implementation up to now has been a challenging task.

This thesis aimed to develop a methodology for implementing custom product configurators, that strongly simplifies the technical realization of configuration systems. The results of this work are a universal conceptual language for modeling customizable products and a framework, which is capable of turning the configurable product models into executable configuration processes. The overall approach has been given the name "OpenConfigurator".

To accomplish the defined goals, we analyzed the area of product customization, which can be seen as the global business context, in which configurators are used (Chapter 2, *Product Customization*). Moreover, we identified the main theories and concepts behind configurators, based on an in-depth literature research (Chapter 3, *Configurators*). This helped us gathering the concrete requirements, that have to be addressed by our methodology.

The fundamental idea behind our methodology described in Chapter 4, *Methodology and Conceptualization* is, that customizable products are modeled as object-oriented classes. In this sense, the product configuration process corresponds to the instantiation of a configurable product class. The configured object, which is the result of the instantiation process, represents a specific, customer tailored product variant.

¹³<http://www.amazon.com/>, last accessed April 25th, 2012.

¹⁴acronym for Recommendation Construction Kit

In order to realize this idea, we had to accomplish to tasks:

1. Conceptualize an object-oriented modeling language for describing configurable products.
2. Implement a framework, that controls the instantiation process of configurable product type, thereby, incorporating the customer's requirements and decisions.

Consequently, based on the description of customizable products, we derived a modeling approach for expressing configurable product architectures in terms of custom domain models.

These domain-specific models, realized as Java classes, accompanied with Java annotation based meta-data, capture the all configuration relevant information, including:

- Product Structure (classes and their relationships)
- Product Information (product annotations)
- Product Customizability (configuration annotations)
- Product Data (domain and default annotations)
- Product Restrictions (constraint annotations)

Based on the theoretical work and the feature set defined in Chapter 3, *Configurators*, we established an architecture for the framework's execution environment in Chapter 5, *Technical Architecture and Implementation*. The designed framework turns the aforementioned domain-specific model into a concrete configuration process. Thereby, the domain specific model is transformed into a domain-independent, generic configuration model, which represents the configured product at runtime. Based on this generic configuration model, the provided APIs realize additional features, relevant in the context of configuration: it implements the retrieval of product from a database (using the JPA technology) and supports the validation of product constraints (using the Bean Validation standard).

Within a concrete case study, presented in Chapter 6, *Evaluation and Validation*, we showed how the conceptualization and the framework can be applied practically for the implementation of a mobile configurator for customizable bikes.

We believe, that OpenConfigurator establishes a strongly productive methodology for the practical realization of product configurators, fostering maintainability, testability and extensibility in the same way. Even more, by building on technologies and concepts widely accepted in the developer community, it eases learning and development greatly.

The OpenConfigurator framework (potentially marketed as a commercial product) is intended to be used in real-world configurator applications. Simultaneously, we plan to release it under a popular open source license, in order to make it accessible for future research and development efforts: we'd welcome, if it forms the basis for future ideas and applications, not necessarily limited to configuration use cases. OpenConfigurator has been designed for flexibility, customizability and extensibility.



Constraints

The constraints provided by the OpenConfigurator are explained on the following pages.

Table A.1. Logical/Arithmetical Comparison Constraints

Constraint Annotation	Relation (pseudo code)	Description	Example
@Equals	a.equals(b)	Value of a is equal to value of b.	@Equals(value="#{_this.b}", condition="#{_this.unicolor}") Color a; Color b; boolean unicolor;
@NotEquals	not a.equals(b)	Value of a is not equal to value of b.	@NotEquals("#{_this.b}") Color a; Color b;
@Min	a >= b	Value of a is equal to or greater than value of b.	@Min("#{_this.b}") int a; int b;
@Max	a <= b	Value of a is equal to or less than value of b.	@Max("10") int a;

Table A.2. Cardinality Constraints

Constraint Annotation	Relation (pseudo code)	Description	Example
@Cardinality	$ a == b$	Size of collection a is equal to value of b.	@Cardinality(5) Collection<Item> items;
@Cardinality.Min	$ a >= b$	Size of collection a is equal to or greater than value of b.	@Cardinality.Min(1) Collection<Item> items;
@Cardinality.Max	$ a <= b$	Size of collection a is equal to or less than value of b.	@Cardinality.Max("#{_this.available}") Collection<Item> items; int available;
@Optional	$ a >= 0$	Element may but must not be present, respectively collection may but must not contain an element. Note that elements are by default optional, which is why this annotation is used rather for readability only.	@Optional Collection<Equipment> equipment;
@Required	$ a >= 1$, i.e. $a \neq \text{null}$	Element must be present, respectively collection must contain at least one element.	@Required Frame frame;

Table A.3. Compatibility Constraints

Constraint Annotation	Relation (pseudo code)	Description	Example
<code>@Compatible.Type</code>	<code>typeOf(a) = b</code>	Type <code>b</code> is assignable from type of value of <code>a</code>. In other words, <code>a</code>'s type is <code>b</code> or a subtype of it. Most often this annotation is used with a conditional expression. ❶ Only <code>CompatiblePart</code> instances can be used as <code>part</code>. <code>Part</code> and <code>IncompatiblePart</code> cannot.	<pre> class Part { ... } class CompatiblePart extends Part { ... } class IncompatiblePart extends Part { ... } class Product { @Compatible .Type(CompatiblePart.class) ❶ Part part; } </pre>
<code>@CompatibleElementType</code>	all <code>e</code> in <code>a</code>: <code>typeOf(e) = b</code>	For all elements <code>e</code> in collection <code>a</code>, type <code>b</code> is assignable from type of <code>e</code>. In other words, all <code>es</code> in <code>a</code> are of type <code>b</code> or a subtype of it. Most often this annotation is used with a conditional expression. ❶ Only <code>CompatiblePart</code> instances can be used as <code>part</code> instances. <code>Part</code> and <code>IncompatiblePart</code> cannot.	<pre> class Part { ... } class CompatiblePart extends Part { ... } class IncompatiblePart extends Part { ... } class Product { @Compatible .ElementType(CompatiblePart.class) ❶ Collection<Part> parts; } </pre>

Constraint Annotation	Relation (pseudo code)	Description	Example
<code>@Compatible</code> . <code>Matches</code>	<code>condition(a) = true</code>	Value of <code>a</code> matches condition. <code>a</code> is passed as variable <code>"_"</code> to condition. ❶ <code>MatchingPart</code> instances can be used as <code>part</code> instance. <code>NotMatchingPart</code> instances cannot.	<pre> class Part { abstract boolean matches(); } class MatchingPart extends Part { boolean matches() { return true; } } class NotMatchingPart extends Part { boolean matches() { return false; } } class Product { @Compatible.Matches("#{_}.matches()}") ❶ Part part; } class Product { @Compatible .ElementMatches("#{_}.matches()}") ❶ Collection<Part> parts; } </pre>
<code>@Compatible</code> . <code>ElementMatches</code>	<code>all e in a: condition(e) = true</code>	For all elements <code>e</code> in collection <code>a</code>, <code>e</code> matches condition. The current element <code>e</code> is passed as variable <code>"_current"</code> to condition. The collection <code>a</code> itself is passed as <code>"_"</code>. ❶ <code>MatchingPart</code> instances can be used as <code>part</code> instances. <code>NotMatchingPart</code> instances cannot.	<pre> class Product { @Compatible .ElementMatches("#{_}.matches()}") ❶ Collection<Part> parts; } </pre>

Table A.4. Incompatibility Constraints

Constraint Annotation	Relation (pseudo code)	Description	Example
<code>@IncompatibleTypeNot</code>	<code>typeOf(a) not b</code>	Type b is not assignable from type of value of a. In other words, a's type is not b nor any subtype of it. Most often this annotation is used with a conditional expression.	<pre> class Part { abstract boolean matches(); } class MatchingPart extends Part { boolean matches() { return true; } } class IncompatiblePart extends Part { boolean matches() { return false; } } class Product { @IncompatibleTypeNot(IncompatiblePart.class) Part part; } </pre>
<code>@IncompatibleElementTypeNot</code>	<code>all e in a: typeOf(e) not b</code>	For all elements e in collection a, type b is not assignable from type of e. In other words, all es in a are not of type b nor any subtype it. Most often this annotation is used with a conditional expression.	<pre> class Product { @Incompatible .ElementTypeNot(IncompatiblePart.class) Collection<Part> parts; } </pre>
<code>@IncompatibleMatchesNot</code>	<code>not condition(a) = true</code>	Value of a does not match condition. a is passed as variable " _ " to condition.	<pre> class Product { @Incompatible .MatchesNot("#{_.matches()}") Part part; } </pre>
<code>@IncompatibleElementMatchesNot</code>	<code>all e in a: not condition(e) = true</code>	For all elements e in collection a, e does not match condition. The current element e is passed as variable " _current " to condition. The collection a itself is passed as " _ "	<pre> class Product { @Incompatible .ElementMatchesNot("#{_.matches()}") Collection<Part> parts; } </pre>

Table A.5. Complex Constraints

Constraint Annotation	Relation (pseudo code)	Description	Example
@Satisfies	eval(script) = true	Bean-level annotation. Asserts that the evaluation of the script given in value returns true.	<pre>@Satisfies("#{_this.x < _this.y}") class Product { int x; int y; }</pre>
@Relational	exists(a, R)	Bean-level annotation. Asserts that the given instance a is contained in relation R.	<pre>@Selectable @Relational @Domain.Query class Product { int x; int y; }</pre>



Example Domain Model: Bike

B.1. Component Specifications

This section lists the exact specification data¹ and source code of the example discussed in Chapter 6, *Evaluation and Validation*.

¹The specification data has kindly been provided by electrolyte.cc, see <http://www.electrolyte.cc/>, last accessed July 18th, 2012.

Bike Components						
Frame						
Description	Material	Color	Geometry	Size	Price (EUR)	
Titanium custom made frame with titanium fork	Titanium	Golden Silver Black	XS S M L	(calculated from geometry)	800	
Aluminum frame handcrafted in Europe with scattered weld seams and aluminum fork.	Aluminum	White Red Blue			850	
Gearing						
Description		Type	Gears		Price (EUR)	
Classical single-speed gearing		Single-speed	1		350	
Modern hub gearing		Hub gear	3 5 7 8 9 11 14		600	
Brake						
Description		Type			Price (EUR)	
Shimano Deore V-brake		V-Brake			350	
Cross brake lever with Shimano 105 brake		V-Brake			420	
Trickstuff Cleg 2 disc brake		Disk-Brake			500	
Cranksets & Pedals						
Description		Type			Price (EUR)	
Shimano Alfine crankset with 45 teeth; Wellgo Lu C17 pedals		Classic			320	
Shimano Alfine crankset (45 teeth) with Wellgo Lu C19 pedals		Regular			380	
Limited Shimano Dura Ace Carbon crankset with up to 45 teeth		Deluxe			450	
Seat post & Saddle						
Description		Type	Material	Color	Price (EUR)	
Titanium seatpost with Brooks Titan Swallow saddle.	Regular		Titanium / Leather	Brown	300	
Syntace P6 aluminium seatpost with Selle San Marco Regal saddle.	Trekking		Aluminum / Synthetic	Black	150	
Syntace P6 carbon seatpost with Tune Carbon saddle	Race		Carbon	Black	400	
Stem & Handlebar						
Description		Type			Price (EUR)	
Tune stem, ti handlebar with Brooks leather grips		Classic			350	
Syntace Force 149 stem with Syntace Duraflite handlebar		Regular			250	
Syntace Force 139 stem with Syntace Stratos bullhorn handlebar		Sportive			380	
Wheels						
Description		Type			Price (EUR)	
Mavic rims with Shimano front hub; Schwalbe Marathon Supreme tyres and safety quick release skewer		Urban			150	
Mavic rims with Son deluxe Dynamo, Schwalbe Marathon Supreme tyres and safety quick release skewer		Urban			200	
Mavic rims with Shimano front hub; Conti Cross tyres and safety quick release skewer		Cross			220	
Mavic CXP 33 rims with Tune front hub, Schwalbe Ultremo DD (25 mm) tyres and Tune time trial quick release skewer		Race			300	
Motor & Battery						
Description	Power (Watt)	Battery	Max. Speed (km/h)	Range (km)	Weight (kg)	Price (EUR)
250 watt hub motor powered by a 36V 2,6Ah battery	250	Li-Ion 36V 2,6Ah	25	15	5	1000
Powerful all-round motor with larger dimensions	250	Li-Ion 36V 11Ah	25	30-90	12	1200
Equipment						
Kind	Description					Price (EUR)
Mudguards	Curana C° Lite mudguards					100
Mudguards	Designfabrik Wolff Linea Legno wooden mudguards					200
Light	Supernova E3 front and tail light					80
Carrier	Aluminum carrier					100

B.2. Source Code

Class Bike

```
package example;

import java.math.BigDecimal;
import java.util.Collection;

import configurator.configuration.annotation.Calculated;
import configurator.configuration.annotation.Configurable;
import configurator.product.annotation.Product;
import configurator.product.annotation.Product.Asset;
import configurator.product.annotation.Product.Attribute;
import configurator.product.annotation.Product.Price;
import configurator.validation.annotation.Optional;
import configurator.validation.annotation.Required;

@Configurable
@Product
public abstract class Bike
{
    private Frame frame;
    private Gearing gearing;
    private Brake brake;
    private CranksPedals cranksPedals;
    private StemHandlebar stemHandlebar;
    private Wheels wheels;
    private SeatpostSaddle seatpostSaddle;
    private Collection<Equipment> equipment;

    private int weight;
    private byte[] image;
    private BigDecimal price;

    @Required
    public Frame getFrame()
    {
        return frame;
    }

    public void setFrame(Frame frame)
    {
        this.frame = frame;
    }

    @Required
    public Gearing getGearing()
    {
        return gearing;
    }

    public void setGearing(Gearing gearing)
    {
        this.gearing = gearing;
    }

    @Required
    public Brake getBrake()
    {
        return brake;
    }
}
```

```
public void setBrake(Brake brake)
{
    this.brake = brake;
}

@Required
public CranksPedals getCranksPedals()
{
    return cranksPedals;
}

public void setCranksPedals(CranksPedals cranksPedals)
{
    this.cranksPedals = cranksPedals;
}

@Required
public StemHandlebar getStemHandlebar()
{
    return stemHandlebar;
}

public void setStemHandlebar(StemHandlebar stemHandlebar)
{
    this.stemHandlebar = stemHandlebar;
}

@Required
public Wheels getWheels()
{
    return wheels;
}

public void setWheels(Wheels wheels)
{
    this.wheels = wheels;
}

@Required
public SeatpostSaddle getSeatpostSaddle()
{
    return seatpostSaddle;
}

public void setSeatpostSaddle(SeatpostSaddle seatpostSaddle)
{
    this.seatpostSaddle = seatpostSaddle;
}

@Optional
public Collection<Equipment> getEquipment()
{
    return equipment;
}

public void setEquipment(Collection<Equipment> equipment)
{
    this.equipment = equipment;
}

@Calculated
@Product.Attribute(level = Attribute.STANDARD)
public int getWeight()
```

```

    {
        return weight;
    }

    @Asset
    public byte[] getImage()
    {
        return image;
    }

    @Price
    public BigDecimal getPrice()
    {
        return price;
    }
}

```

Class Brake

```

package example;

import java.math.BigDecimal;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

import configurator.configuration.annotation.Selectable;
import configurator.configuration.annotation.Variable;
import configurator.data.annotation.Default;
import configurator.data.annotation.Domain;
import configurator.model.annotation.Ignore;
import configurator.product.annotation.Product.Description;
import configurator.product.annotation.Product.Price;
import configurator.validation.annotation.Relational;

@Relational
@Domain.Query
@Default.Query("select b from Brake b where b.type = 'V_BRAKE'")
@Selectable
@Entity
public class Brake
{
    public static enum Type
    {
        V_BRAKE,
        DISC_BRAKE
    }

    private long id;
    private String description;
    private Brake.Type type;
    private BigDecimal price;

    @Ignore
    @Id
    @GeneratedValue
    public long getId()
    {
        return id;
    }
}

```

```
    @Description
    public String getDescription()
    {
        return description;
    }

    @Variable
    public Brake.Type getType()
    {
        return type;
    }

    public void setType(Brake.Type type)
    {
        this.type = type;
    }

    @Price
    public BigDecimal getPrice()
    {
        return price;
    }
}
```

Class Carrier

```
package example;

public class Carrier extends Equipment
{
}

```

Class CityBike

```
package example;

import configurator.configuration.annotation.Configurable;
import configurator.product.annotation.Product;

@Configurable
@Product
@Product.Name("City Bike")
public class CityBike extends Bike
{
}

```

Class CranksPedals

```
package example;

import java.math.BigDecimal;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

import configurator.configuration.annotation.Selectable;
import configurator.configuration.annotation.Variable;
import configurator.data.annotation.Default;
import configurator.data.annotation.Domain;
import configurator.model.annotation.Ignore;

```

```

import configurator.product.annotation.Product.Description;
import configurator.product.annotation.Product.Price;
import configurator.validation.annotation.Relational;

@Relational
@Domain.Query
@Default.Query("select c from CranksPedals c where c.type = 'Regular'")
@Selectable
@Entity
public class CranksPedals
{
    public static enum Type
    {
        Classic,
        Regular,
        Deluxe
    }

    private long id;
    private String description;
    private CranksPedals.Type type;
    private BigDecimal price;

    @Ignore
    @Id
    @GeneratedValue
    public long getId()
    {
        return id;
    }

    @Description
    public String getDescription()
    {
        return description;
    }

    @Variable
    public CranksPedals.Type getType()
    {
        return type;
    }

    public void setType(CranksPedals.Type type)
    {
        this.type = type;
    }

    @Price
    public BigDecimal getPrice()
    {
        return price;
    }
}

```

Class ElectroBike

```

package example;

import configurator.configuration.annotation.Configurable;
import configurator.product.annotation.Product;
import configurator.validation.annotation.Required;

```

```
@Configurable
@Product
@Product.Name("Electro Bike")
public class ElectroBike extends CityBike
{
    private MotorBattery motorBattery;

    @Required
    public MotorBattery getMotorBattery()
    {
        return motorBattery;
    }

    public void setMotorBattery(MotorBattery motorBattery)
    {
        this.motorBattery = motorBattery;
    }
}
```

Class Equipment

```
package example;

import java.math.BigDecimal;

import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.MappedSuperclass;

import configurator.configuration.annotation.Selectable;
import configurator.data.annotation.Domain;
import configurator.model.annotation.Ignore;
import configurator.product.annotation.Product.Description;
import configurator.product.annotation.Product.Price;
import configurator.validation.annotation.Relational;

@Relational
@Domain.Query
@Selectable
@MappedSuperclass
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Equipment
{
    private long id;
    private String description;
    private BigDecimal price;

    @Ignore
    @Id
    @GeneratedValue
    public long getId()
    {
        return id;
    }

    @Description
    public String getDescription()
    {
        return description;
    }
}
```

```

    }

    @Price
    public BigDecimal getPrice()
    {
        return price;
    }
}

```

Class Frame

```

package example;

import java.math.BigDecimal;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Transient;
import javax.validation.constraints.Size;

import configurator.configuration.annotation.Calculated;
import configurator.configuration.annotation.Configurable;
import configurator.configuration.annotation.Parameter;
import configurator.configuration.annotation.Variable;
import configurator.data.annotation.Default;
import configurator.data.annotation.Domain;
import configurator.model.annotation.Ignore;
import configurator.product.annotation.Product.Description;
import configurator.product.annotation.Product.Price;
import configurator.validation.annotation.Optional;
import configurator.validation.annotation.Relational;
import configurator.validation.annotation.Required;

@Relational(properties = { "description", "material", "price" })
@Domain.Query
@Configurable
@Entity
public class Frame
{
    public static enum Material
    {
        Titanium,
        Aluminum
    }

    public static enum Color
    {
        Gold,
        Silver,
        Black,
        White,
        Red,
        Blue
    }

    public static enum Geometry
    {
        XS(16),
        S(18),
        M(20),

```

```
L(22);

private int size;

private Geometry(int size)
{
    this.size = size;
}

public int getSize()
{
    return size;
}
}

private long id;
private String description;
private Frame.Material material;
private Frame.Color color;
private Frame.Geometry geometry;
private String label;
private BigDecimal price;

@Ignore
@Id
@GeneratedValue
public long getId()
{
    return id;
}

@Description
public String getDescription()
{
    return description;
}

@Required
@Variable
@Default("Aluminum")
public Frame.Material getMaterial()
{
    return material;
}

public void setMaterial(Frame.Material material)
{
    this.material = material;
}

@Transient
@Required
@Variable
@Default("Black")
public Frame.Color getColor()
{
    return color;
}

public void setColor(Frame.Color color)
{
    this.color = color;
}
}
```

```

    @Transient
    @Required
    @Variable
    @Default("M")
    public Frame.Geometry getGeometry()
    {
        return geometry;
    }

    public void setGeometry(Frame.Geometry geometry)
    {
        this.geometry = geometry;
    }

    @Transient
    @Size(max = 24)
    @Calculated
    public int getSize()
    {
        if (geometry != null)
            return geometry.size;

        return 0;
    }

    @Transient
    @Optional
    @Parameter
    @Default("MyBike")
    public String getLabel()
    {
        return label;
    }

    public void setLabel(String label)
    {
        this.label = label;
    }

    @Price
    public BigDecimal getPrice()
    {
        return price;
    }
}

```

Class Gearing

```

package example;

import java.math.BigDecimal;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

import configurator.configuration.annotation.Selectable;
import configurator.configuration.annotation.Variable;
import configurator.data.annotation.Default;
import configurator.data.annotation.Domain;
import configurator.model.annotation.Ignore;

```

```
import configurator.product.annotation.Product.Description;
import configurator.product.annotation.Product.Price;
import configurator.validation.annotation.Max;
import configurator.validation.annotation.Min;
import configurator.validation.annotation.Relational;

@Relational
@Domain.Query
@Default.Query("select g from Gearing g where g.type = 'HubGear' and g.gears = 7")
@Selectable
@Entity
public class Gearing
{
    public static enum Type
    {
        SingleSpeed,
        HubGear
    }

    private long id;
    private String description;
    private Gearing.Type type;
    private int gears;
    private BigDecimal price;

    @Ignore
    @Id
    @GeneratedValue
    public long getId()
    {
        return id;
    }

    @Description
    public String getDescription()
    {
        return description;
    }

    @Variable
    public Gearing.Type getType()
    {
        return type;
    }

    public void setType(Gearing.Type type)
    {
        this.type = type;
    }

    @Max(value = "1", condition="#{_this.type == 'SingleSpeed'}")
    @Min(value = "2", condition="#{_this.type == 'HubGear'}")
    @Variable
    public int getGears()
    {
        return gears;
    }

    public void setGears(int gears)
    {
        this.gears = gears;
    }
}
```

```
@Price
public BigDecimal getPrice()
{
    return price;
}

}

Class Light

package example;

public class Light extends Equipment
{

}

Class MotorBattery

package example;

import java.math.BigDecimal;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

import configurator.configuration.annotation.Selectable;
import configurator.configuration.annotation.Variable;
import configurator.data.annotation.Domain;
import configurator.model.annotation.Ignore;
import configurator.product.annotation.Product.Description;
import configurator.product.annotation.Product.Price;
import configurator.validation.annotation.Relational;

@Relational
@Domain.Query
@Selectable
@Entity
public class MotorBattery
{
    private long id;
    private String description;
    private int power;
    private String battery;
    private int maximumSpeed;
    private String range;
    private int weight;
    private BigDecimal price;

    @Ignore
    @Id
    @GeneratedValue
    public long getId()
    {
        return id;
    }

    @Description
    public String getDescription()
    {
        return description;
    }
}
```

```
@Variable
public int getPower()
{
    return power;
}

public void setPower(int power)
{
    this.power = power;
}

@Variable
public String getBattery()
{
    return battery;
}

public void setBattery(String battery)
{
    this.battery = battery;
}

@Variable
public int getMaximumSpeed()
{
    return maximumSpeed;
}

public void setMaximumSpeed(int maximumSpeed)
{
    this.maximumSpeed = maximumSpeed;
}

@Variable
public String getRange()
{
    return range;
}

public void setRange(String range)
{
    this.range = range;
}

public int getWeight()
{
    return weight;
}

public void setWeight(int weight)
{
    this.weight = weight;
}

@Price
public BigDecimal getPrice()
{
    return price;
}

}
```

Class MountainBike

```
package example;

import configurator.configuration.annotation.Configurable;
import configurator.data.annotation.Default;
import configurator.product.annotation.Product;

@Configurable
@Product
@Product.Name("Mountain Bike")
public class MountainBike extends Bike
{
    @Default.Query("select w from Wheels w where w.type = 'Cross'")
    public Wheels getWheels()
    {
        return super.getWheels();
    }
}
```

Class Mudguards

```
package example;

public class Mudguards extends Equipment
{
}
```

Class RacingBike

```
package example;

import configurator.data.annotation.Default;
import configurator.product.annotation.Product;

@Product.Name("Racing Bike")
public class RacingBike extends Bike
{
    @Default.Query("select w from Wheels w where w.type = 'Race'")
    public Wheels getWheels()
    {
        return super.getWheels();
    }
}
```

Class SeatpostSaddle

```
package example;

import java.math.BigDecimal;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

import configurator.configuration.annotation.Selectable;
import configurator.configuration.annotation.Variable;
import configurator.data.annotation.Default;
import configurator.data.annotation.Domain;
import configurator.model.annotation.Ignore;
import configurator.product.annotation.Product.Description;
import configurator.product.annotation.Product.Price;
```

```
import configurator.validation.annotation.Relational;

@Relational
@Domain.Query
@Default.Query("select s from SeatpostSaddle s " +
               "where s.type = 'Regular' and s.color = 'Black'")
@Selectable
@Entity
public class SeatpostSaddle
{
    public static enum Type
    {
        Regular,
        Trekking,
        Race
    }

    public static enum Color
    {
        Brown,
        Black
    }

    private long id;
    private String description;
    private Type type;
    private String material;
    private Color color;
    private BigDecimal price;

    @Ignore
    @Id
    @GeneratedValue
    public long getId()
    {
        return id;
    }

    @Description
    public String getDescription()
    {
        return description;
    }

    @Variable
    public SeatpostSaddle.Type getType()
    {
        return type;
    }

    public void setType(SeatpostSaddle.Type type)
    {
        this.type = type;
    }

    @Variable
    public String getMaterial()
    {
        return material;
    }

    public void setMaterial(String material)
    {

```

```

        this.material = material;
    }

    @Variable
    public Color getColor()
    {
        return color;
    }

    public void setColor(Color color)
    {
        this.color = color;
    }

    @Price
    public BigDecimal getPrice()
    {
        return price;
    }
}

```

Class StemHandlebar

```

package example;

import java.math.BigDecimal;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

import configurator.configuration.annotation.Selectable;
import configurator.configuration.annotation.Variable;
import configurator.data.annotation.Default;
import configurator.data.annotation.Domain;
import configurator.model.annotation.Ignore;
import configurator.product.annotation.Product.Description;
import configurator.product.annotation.Product.Price;
import configurator.validation.annotation.Relational;

@Relational
@Domain.Query
@Default.Query("select s from StemHandlebar s where s.type = 'Regular'")
@Selectable
@Entity
public class StemHandlebar
{
    public static enum Type
    {
        Classic,
        Regular,
        Sportive
    }

    private long id;
    private String description;
    private Type type;
    private BigDecimal price;

    @Ignore
    @Id
    @GeneratedValue

```

```
    public long getId()
    {
        return id;
    }

    @Description
    public String getDescription()
    {
        return description;
    }

    @Variable
    public StemHandlebar.Type getType()
    {
        return type;
    }

    public void setType(StemHandlebar.Type type)
    {
        this.type = type;
    }

    @Price
    public BigDecimal getPrice()
    {
        return price;
    }
}
```

Class Wheels

```
package example;

import java.math.BigDecimal;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

import configurator.configuration.annotation.Selectable;
import configurator.configuration.annotation.Variable;
import configurator.data.annotation.Default;
import configurator.data.annotation.Domain;
import configurator.model.annotation.Ignore;
import configurator.product.annotation.Product.Description;
import configurator.product.annotation.Product.Price;
import configurator.validation.annotation.Relational;

@Relational
@Domain.Query
@Default.Query("select w from Wheels w where w.type = 'Urban'")
@Selectable
@Entity
public class Wheels
{
    public static enum Type
    {
        Urban,
        Cross,
        Race
    }

    private long id;
```

```
private String description;
private Type type;
private BigDecimal price;

@Ignore
@Id
@GeneratedValue
public long getId()
{
    return id;
}

>Description
public String getDescription()
{
    return description;
}

@Variable
public Wheels.Type getType()
{
    return type;
}

public void setType(Wheels.Type type)
{
    this.type = type;
}

@Price
public BigDecimal getPrice()
{
    return price;
}
}
```



OpenConfigurator API/SPI

Interface OpenConfigurator

```
package configurator;

public interface OpenConfigurator
{
    /**
     * TODO emit BootstrapImplementor event TODO emit
     * ConfiguratorImplementor event
     */
    public Configurator initialize();
}
```

Interface Bootstrap

```
package configurator.bootstrap;

import configurator.Configurator;
import configurator.spi.Initialization;

/**
 * Bootstraps the configurator instance. The bootstrap sequence:
 * <ol>
 * <li>invoke {@link #initialize()}</li>
 * <li>obtain {@link Configurator} instance by calling
 * {@link #getConfigurator()}</li>
 * </ol>
 */
public interface Bootstrap
{
    /**
     * Triggers initialization of a configurator instance. Emits a
     * {@link Configurator} {@link Initialization} event.
     */
    void initialize() throws BootstrapException;
}
```

```
        * Obtain the initialized configurator instance.
        */
    Configurator getConfigurator();
}
```

Interface Configurator

```
package configurator;

import configurator.model.Configuration;

/**
 * The main entry point to the configurator API. Allows to create
 * configuration sessions. TODO emit events: @Preparation
 * ConfigurationSessionImplementor, @Preparation
 * Component/Attribute/Part, @Initialization Component/Attribute/Part,
 * @Initialization ConfigurationSessionImplementor
 */
public interface Configurator
{
    <T> ConfigurationSession<T> createSession(Class<T> componentType);

    <T> ConfigurationSession<T> restoreSession(
        Configuration<T> configuration);
}
```

Interface ConfigurationSession

```
package configurator;

import java.io.Serializable;

import configurator.model.Configuration;
import configurator.task.Agenda;

/**
 * (c) 2012 Copyright by Vivian Steller
 */

/**
 * Manages the runtime state of a configuration.
 *
 * @author Vivian Steller
 * @since 1.0
 */
public interface ConfigurationSession<T> extends Serializable
{
    Class<T> getConfiguredType();

    Configuration<T> getConfiguration();

    Agenda<T> getAgenda();

    void store(String id);

    void load(String id);

    void restore(Configuration<T> configuration);

    void restore(T instance);

    void submit();

    void close();
}
```

```
}
```

Interface Configuration

```
package configurator.model;

/**
 * TODO rename to Model<T>
 */
public interface Configuration<T>
{
    /**
     * Allows traversal of the configuration.
     */
    Component<T> getRootComponent();

    /**
     * Validates the configuration.
     */
    boolean isValid();

    /**
     * Whether all components, attributes and parts contained within the
     * configuration have been specified.
     */
    boolean isComplete();

    /**
     * Allows accessing the underlying configured object.
     */
    T getValue();
}
```

Interface Element

```
package configurator.model;

public interface Element<T>
{
    Component<?> getParent();

    T getValue();

    void setValue(T value);
}
```

Interface Component

```
package configurator.model;

import java.util.Collection;

import configurator.event.Events;
import configurator.event.Listener;
import configurator.metadata.ComponentDescriptor;

/**
 * Represents a component of type C. This interface allows
 * you to generically manage instances of JavaBean style classes.
 * Implementations automatically take care of instantiation, state as
 * well as dynamic type changes.<br/>
 * <br/>
 * While C is considered the base type for this component's
 * value, the actual value can have any subtype of C. See
```

```
* {@link #getType()} for more information.<br/>
* <br/>
* Additionally, the interface supports registering of observers for
* various events that indicate changes to the component's state.<br/>
* <br/>
* Implementations of this class do <i>not</i> have to be implemented
* thread-safe but serializable.<br/>
* <br/>
* <i>Modeling note: methods are designed to be idempotent</i>
*
* @param <C>
*         type represented by this component
*/
public interface Component<C> extends VariableTyped<C>, Facetted,
    Element<C>
{
    /**
     * Descriptor containing metadata about the current component's type
     * (returned by {@link #getType()}). As the component's type can
     * dynamically change at runtime, also the descriptor returned by
     * this method varies. However, the descriptor always reflects the
     * currently set type of this component which may be any subtype of
     * <code>C</code>.<br/>
     * <br/>
     * The returned {@link ComponentDescriptor} is <i>immutable</i>.
     *
     * @param <T>
     *         type corresponding to the value's current type
     *         (returned by {@link #getType()})
     * @return the {@link ComponentDescriptor} associated to the current
     *         type
     */
    <T extends C> ComponentDescriptor<T> getMetadata();

    /**
     * Gets the base type of the value managed by this component.
     *
     * @return the component value's base type
     */
    Class<C> getBaseType();

    /**
     * Gets the current type of the value managed by this component.
     *
     * @param <T>
     *         the value's current type
     * @return the component value's current type
     * @see #setType(Class)
     */
    <T extends C> Class<T> getType();

    /**
     * Sets the type of the value managed by this component. The passed
     * type must be a subclass of this component's base type
     * <code>C</code> and may be declared abstract.<br/>
     * <br/>
     * The component supports switching between any type for that the
     * rules above apply. Especially these type changes are supported:
     * <ul>
     * <li><i>down the type hierarchy</i>: setting the type to a subtype
     * of the component's current type (returned by {@link #getType()})</li>
     * <li><i>up the type hierarchy</i>: setting the type to a supertype
     * of the component's current type (returned by {@link #getType()})
     */
}
```

```

* as long as the supertype is a subtype of the base type
* <code>C</code></li>
* <li><i>horizontally in the type hierarchy</i>: setting the type
* to another subtype of the component's base type which is not
* necessarily a subtype or supertype of the component value's
* current type (returned by {@link #getType()})</li>
* </ul>
* The component dynamically adds or removes attributes and parts
* from the component depending on the type change. Further, upon a
* type change the {@link #getMetadata()} method returns the
* descriptor of the newly set type so that the new type is
* reflected.<br/>
* <br/>
* Calling this method triggers the following events:
* {@link TypeChanged}, {@link Changed}
*
* @param <T>
*         the value's type to set
* @param type
*         the type of the component's value to set
*/
<T extends C> void setType(Class<T> type);

/**
 * Whether the current component's type is instantiatable or
 * abstract.
 *
 * @return <code>true</code> if the current type is abstract
 *         (non-instantiatable)
 */
boolean isAbstract();

/**
 * Whether the component is instantiated or not. Instantiated means
 * that the component manages an underlying bean instance of type
 * <code>C</code> (or a subtype thereof).
 *
 * @return <code>true</code> if the component is instantiated
 * @see #getValue()
 * @see #setValue(Object)
 */
boolean isInstantiated();

/**
 * Whether the user did specify any property value of this component
 * or any of it's sub-components (parts). This method basically
 * returns <code>true</code> if any <code>set*</code>-method, such
 * as {@link #setAttributeValue(String, Object)},
 * {@link #setPartValue(String, Object)}, {@link #setValue(Object)}
 * or {@link #setType(Class)} has been invoked.
 *
 * @return <code>true</code> if the user specified any of this
 *         component's properties
 */
boolean isSpecified();

/**
 * Accesses the instance managed by this component. The bean might
 * have been automatically instantiated during the configuration
 * process or alternatively must have been explicitly set by the
 * user using the {@link #setValue(Object)} method.
 *
 * @return the underlying bean instance or <code>null</code> if the

```

```
*          component has not been instantiated yet
* @see #setValue(Object)
*/
C getValue();

/**
 * Explicitly sets the underlying bean instance. The component's
 * attributes and parts are populated with the given value in order
 * to reflect the bean instance's state.<br/>
 * <br/>
 * Calling this method triggers the following events if changes to
 * the component's current state are detected: {@link ValueChanged},
 * {@link TypeChanged}, {@link Changed}
 *
 * @param value
 *         the value to populate this component with
 */
void setValue(C value);

/**
 * If this component is a part of another component this method
 * returns the containing component otherwise <code>null</code>.
 *
 * @return the component containing this component if present,
 *         otherwise <code>null</code>
 */
Component<?> getParent();

/**
 * Resets the component's state. The component's type is reset to
 * the base type <code>C</code>. The managed instance is set to
 * <code>null</code>.<br/>
 * <br/>
 * Calling this method triggers the following events: {@link Reset},
 * {@link Changed}
 */
void reset();

// F A C E T S

/**
 * Checks whether the given facet type is defined for the component.
 *
 * @param facetType
 *         the facet type to check availability for
 * @return <code>true</code> if the component can be viewed as the
 *         given facet type
 */
boolean hasFacet(Class<?> facetType);

/**
 * Obtains the facet view of the given type for this component. The
 * returned facet <b>must be</b> cached by the implementation so
 * that multiple invocations to this method always return the same
 * instance.
 *
 * @param facetType
 *         the type of the facet
 * @return the component as facet of the given type
 */
<F> F getFacet(Class<F> facetType);

// A T T R I B U T E S
```

```

/**
 * Gets the (singular and plural) attributes of this component. The
 * attributes returned correspond to the ones that are defined by
 * the component's current type (returned by {@link #getType()}).<br/>
 * <br/>
 * The returned collection is <i>unmodifiable</i>.
 *
 * @return the attributes of the component
 */
Collection<Attribute<C, ?>> getAttributes();

/**
 * Gets a specific (singular or plural) attribute of this component.
 * If the component's current type (returned by {@link #getType()})
 * doesn't contain the specified attribute an exception is thrown.
 *
 * @param <A>
 *         the type of the attribute's value
 * @param attributeName
 *         the name of the attribute to retrieve (may denote a
 *         singular or plural attribute)
 * @return the attribute with the given name
 * @throws ModelException
 *         if the component's type doesn't contain an attribute
 *         with the given name
 */
<A> Attribute<C, A> getAttribute(String attributeName)
    throws ModelException;

/**
 * Gets the value of a specific singular attribute of this
 * component. If the component's current type (returned by
 * {@link #getType()}) doesn't contain the specified attribute or
 * the given attribute name doesn't denote a singular attribute an
 * exception is thrown.<br/>
 * <br/>
 * This method is a shortcut for: <code>
 * this.getAttribute(attributeName).getValue()
 * </code>
 *
 * @param <A>
 *         the type of the attribute's value
 * @param attributeName
 *         the name of the attribute whose value to retrieve
 *         (must denote a singular attribute)
 * @return the named attribute's value
 * @throws ModelException
 *         if the component's type doesn't contain a singular
 *         attribute with the given name
 * @see SingularAttribute#getValue()
 */
<A> A getAttributeValue(String attributeName) throws ModelException;

/**
 * Sets the value of a specific singular attribute of this
 * component. If the component's current type (returned by
 * {@link #getType()}) doesn't contain the specified attribute or
 * the given attribute name doesn't denote a singular attribute an
 * exception is thrown.<br/>
 * <br/>
 * This method is a shortcut for: <code>
 * this.getAttribute(attributeName).setValue(value)

```

```
* </code>
*
* @param <A>
*         the type of the attribute's value
* @param attributeName
*         the name of the attribute whose value to retrieve
*         (must denote a singular attribute)
* @throws ModelException
*         if the component's type doesn't contain a singular
*         attribute with the given name
* @see SingularAttribute#setValue(Object)
*/
<A> void setAttributeValue(String attributeName, A value)
    throws ModelException;

/**
 * Gets the values of a specific plural attribute of this component.
 * If the component's current type (returned by {@link #getType()})
 * doesn't contain the specified attribute or the given attribute
 * name doesn't denote a plural attribute an exception is thrown.<br/>
 * <br/>
 * This method is a shortcut for: <code>
 * this.getAttribute(attributeName).getValue()
 * </code>
 *
 * @param <A>
 *         the type of the attribute's value
 * @param attributeName
 *         the name of the attribute whose value to retrieve
 *         (must denote a plural attribute)
 * @return the named attribute's values
 * @throws ModelException
 *         if the component's type doesn't contain a plural
 *         attribute with the given name
 * @see PluralAttribute#getValue()
 */
<A> Collection<A> getAttributeValues(String pluralAttributeName)
    throws ModelException;

/**
 * Sets the values of a specific plural attribute of this component.
 * If the component's current type (returned by {@link #getType()})
 * doesn't contain the specified attribute or the given attribute
 * name doesn't denote a plural attribute an exception is thrown.<br/>
 * <br/>
 * This method is a shortcut for: <code>
 * this.getAttribute(attributeName).setValue(value)
 * </code>
 *
 * @param <A>
 *         the type of the attribute's value
 * @param attributeName
 *         the name of the attribute whose value to retrieve
 *         (must denote a plural attribute)
 * @throws ModelException
 *         if the component's type doesn't contain a plural
 *         attribute with the given name
 * @see PluralAttribute#setValue(Object)
 */
<A> void setAttributeValues(String pluralAttributeName,
    Collection<A> values) throws ModelException;

/**
```

```

* Resets a specific (singular or plural) attribute of this
* component. If the component's current type (returned by
* {@link #getType()}) doesn't contain the specified attribute an
* exception is thrown.<br/>
* <br/>
* This method is a shortcut for: <code>
* this.getAttribute(attributeName).reset()
* </code>
*
* @param attributeName
*         the name of the attribute to reset (may denote a
*         singular or plural attribute)
* @throws ModelException
*         if the component's type doesn't contain an attribute
*         with the given name
* @see Attribute#reset()
*/
void resetAttribute(String attributeName) throws ModelException;

// P A R T S

/**
* Gets the (singular and plural) parts of this component. The parts
* returned correspond to the ones that are defined by the
* component's current type (returned by {@link #getType()}).<br/>
* <br/>
* The returned collection is <i>unmodifiable</i>.
*
* @return the parts of the component
*/
Collection<Part<C, ?>> getParts();

/**
* Gets a specific (singular or plural) part of this component. If
* the component's current type (returned by {@link #getType()})
* doesn't contain the specified part an exception is thrown.
*
* @param <P>
*         the type of the part's value
* @param partName
*         the name of the part to retrieve (may denote a
*         singular or plural part)
* @return the part with the given name
* @throws ModelException
*         if the component's type doesn't contain an part with
*         the given name
*/
<P> Part<C, P> getPart(String partName) throws ModelException;

/**
* Gets the referenced component of a specific singular part of this
* component. If the component's current type (returned by
* {@link #getType()}) doesn't contain the specified part or the
* given part name doesn't denote a singular part an exception is
* thrown.<br/>
* <br/>
* This method is a shortcut for: <code>
* ((SingularPart) this.getPart(partName)).getPartComponent()
* </code>
*
* @param <P>
*         the type of the part's value
* @param partName

```

```
*          the name of the part whose component to retrieve (must
*          denote a singular part)
* @return the named part's referenced component
* @throws ModelException
*          if the component's type doesn't contain a singular
*          part with the given name
* @see SingularPart#getPartComponent()
*/
<P> Component<P> getPartComponent(String partName)
    throws ModelException;

/**
 * Gets the value of a specific singular part of this component. If
 * the component's current type (returned by {@link #getType()})
 * doesn't contain the specified part or the given part name doesn't
 * denote a singular part an exception is thrown.<br/>
 * <br/>
 * This method is a shortcut for: <code>
 * this.getPart(partName).getValue()
 * </code>
 *
 * @param <P>
 *          the type of the part's value
 * @param partName
 *          the name of the part whose value to retrieve (must
 *          denote a singular part)
 * @return the named part's value
 * @throws ModelException
 *          if the component's type doesn't contain a singular
 *          part with the given name
 * @see SingularPart#getValue()
 */
<P> P getPartValue(String partName) throws ModelException;

/**
 * Sets the value of a specific singular part of this component. If
 * the component's current type (returned by {@link #getType()})
 * doesn't contain the specified part or the given part name doesn't
 * denote a singular part an exception is thrown.<br/>
 * <br/>
 * This method is a shortcut for: <code>
 * this.getPart(partName).setValue(value)
 * </code>
 *
 * @param <P>
 *          the type of the part's value
 * @param partName
 *          the name of the part whose value to retrieve (must
 *          denote a singular part)
 * @throws ModelException
 *          if the component's type doesn't contain a singular
 *          part with the given name
 * @see SingularPart#setValue(Object)
 */
<P> void setPartValue(String partName, P value)
    throws ModelException;

/**
 * Gets the referenced components of a specific plural part of this
 * component. If the component's current type (returned by
 * {@link #getType()}) doesn't contain the specified part or the
 * given part name doesn't denote a plural part an exception is
 * thrown.<br/>
```

```

* <br/>
* This method is a shortcut for: <code>
* ((PluralPart) this.getPart(partName)).getPartComponents()
* </code>
*
* @param <E>
*         the type of the part's collection elements
* @param partName
*         the name of the part whose component to retrieve (must
*         denote a plural part)
* @return the named part's referenced components
* @throws ModelException
*         if the component's type doesn't contain a plural part
*         with the given name
* @see PluralPart#getPartComponents()
*/
<E> Collection<Component<E>>
    getPartComponents(String pluralPartName)
        throws ModelException;

/**
 * Gets the values of a specific plural part of this component. If
 * the component's current type (returned by {@link #getType()})
 * doesn't contain the specified part or the given part name doesn't
 * denote a plural part an exception is thrown.<br/>
 * <br/>
 * This method is a shortcut for: <code>
 * this.getPart(partName).getValue()
 * </code>
 *
 * @param <E>
 *         the type of the part's collection elements
 * @param partName
 *         the name of the part whose value to retrieve (must
 *         denote a plural part)
 * @return the named part's values
 * @throws ModelException
 *         if the component's type doesn't contain a plural part
 *         with the given name
 * @see PluralPart#getValue()
 */
<E> Collection<E> getPartValues(String pluralPartName)
    throws ModelException; // shortcut
                           // for
                           // getPart(partName).getValue()

/**
 * Sets the values of a specific plural part of this component. If
 * the component's current type (returned by {@link #getType()})
 * doesn't contain the specified part or the given part name doesn't
 * denote a plural part an exception is thrown.<br/>
 * <br/>
 * This method is a shortcut for: <code>
 * this.getPart(partName).setValue(value)
 * </code>
 *
 * @param <E>
 *         the type of the part's collection elements
 * @param partName
 *         the name of the part whose value to retrieve (must
 *         denote a plural part)
 * @throws ModelException
 *         if the component's type doesn't contain a plural part

```

```
*           with the given name
* @see PluralPart#setValue(Object)
*/
<E> void setPartValues(String pluralPartName, Collection<E> values)
    throws ModelException;

/**
 * Resets a specific (singular or plural) part of this component. If
 * the component's current type (returned by {@link #getType()})
 * doesn't contain the specified part an exception is thrown.<br/>
 * <br/>
 * This method is a shortcut for: <code>
 * this.getPart(partName).reset()
 * </code>
 *
 * @param partName
 *         the name of the part to reset (may denote a singular
 *         or plural part)
 * @throws ModelException
 *         if the component's type doesn't contain an part with
 *         the given name
 * @see Part#reset()
 */
void resetPart(String partName) throws ModelException;

/**
 * Accepts a component hierarchy visitor and invokes its methods for
 * any attribute and part found in this component (in
 * <em>breadth-first</em> order). Recursively calls
 * {@link #accept(Visitor)} on any of the part's components.
 *
 * @param visitor
 *         the visitor to invoke
 */
void accept(Visitor visitor);

// L I S T E N E R S

/**
 * Registers a listener for the {@link Changed} event.
 *
 * @param listener
 *         the listener to register
 * @see Changed
 */
void addChangeListener(ChangeListener<C> listener);

/**
 * Unregisters a listener for the {@link Changed} event.
 *
 * @param listener
 *         the listener to unregister
 * @see Changed
 */
void removeChangeListener(ChangeListener<C> listener);

/**
 * Registers a listener for the {@link ValueInstantiated} event.
 *
 * @param listener
 *         the listener to register
 * @see ValueInstantiated
 */
```

```

void addValueInstantiationListener(
    ValueInstantiationListener<C> listener);

/**
 * Unregisters a listener for the {@link ValueInstantiated} event.
 *
 * @param listener
 *         the listener to unregister
 * @see ValueInstantiated
 */
void removeValueInstantiationListener(
    ValueInstantiationListener<C> listener);

/**
 * Registers a listener for the {@link ValueChanged} event.
 *
 * @param listener
 *         the listener to register
 * @see ValueChanged
 */
void
    addValueChangeListener(
        ValueChangeListener<C> valueChangeListener);

/**
 * Unregisters a listener for the {@link ValueChanged} event.
 *
 * @param listener
 *         the listener to unregister
 * @see ValueChanged
 */
void removeValueChangeListener(
    ValueChangeListener<C> valueChangeListener);

/**
 * Registers a listener for the {@link TypeChanged} event.
 *
 * @param listener
 *         the listener to register
 * @see TypeChanged
 */
void
    addTypeChangeListener(TypeChangeListener<C> typeChangeListener);

/**
 * Unregisters a listener for the {@link TypeChanged} event.
 *
 * @param listener
 *         the listener to unregister
 * @see TypeChanged
 */
void
    removeTypeChangeListener(
        TypeChangeListener<C> typeChangeListener);

/**
 * Registers a listener for the {@link Reset} event.
 *
 * @param listener
 *         the listener to register
 * @see Reset
 */
void addResetListener(ResetListener<C> listener);

```

```
/**
 * Unregisters a listener for the {@link Reset} event.
 *
 * @param listener
 *         the listener to unregister
 * @see Reset
 */
void removeResetListener(ResetListener<C> listener);

/**
 * Event listener for the {@link Changed} event.
 *
 * @param <C>
 *         type of the changed component's value
 * @see Changed
 * @author Vivian Steller
 * @since 1.0
 */
interface ChangeListener<C> extends Listener
{
    void handle(Changed<C> event);
}

/**
 * Event listener for the {@link ValueInstantiated} event.
 *
 * @param <C>
 *         type of the instantiated value
 * @see ValueInstantiated
 * @author Vivian Steller
 * @since 1.0
 */
interface ValueInstantiationListener<C> extends Listener
{
    void handle(ValueInstantiated<C> event);
}

/**
 * Event listener for the {@link ValueChanged} event.
 *
 * @param <C>
 *         type of the changed value
 * @see ValueChanged
 * @author Vivian Steller
 * @since 1.0
 */
interface ValueChangeListener<C> extends Listener
{
    void handle(ValueChanged<C> event);
}

/**
 * Event listener for the {@link TypeChanged} event.
 *
 * @param <C>
 *         base type of the component's value whose type has
 *         changed
 * @see TypeChanged
 * @author Vivian Steller
 * @since 1.0
 */
interface TypeChangeListener<C> extends Listener
```

```

    {
        void handle(TypeChanged<C> event);
    }

/**
 * Event listener for the {@link Reset} event.
 *
 * @param <C>
 *         type of the reset component's value
 * @see ValueInstantiated
 * @author Vivian Steller
 * @since 1.0
 */
interface ResetListener<C> extends Listener
{
    void handle(Reset<C> event);
}

// E V E N T S

// Change
/**
 * Event indicating that the component's state changed.<br/>
 * <br/>
 * This event is fired upon any component state changes, e.g. value
 * changed, type changed etc. Usually other events such as
 * {@link ValueChanged} or {@link TypeChanged} have been fired
 * before.<br/>
 * <br/>
 * Change detection is implemented transitively, that is, if one of
 * the component's attributes or parts changes, the change is
 * propagated to the component and the respective event is fired.
 *
 * @param <C>
 *         base type managed by the component
 * @author Vivian Steller
 * @since 1.0
 */
static class Changed<C> extends
    Events.Changed<Component<C>, ChangeListener<C>>
{
    public Changed(Component<C> source)
    {
        super(source);
    }

    protected void call(ChangeListener<C> listener)
    {
        listener.handle(this);
    }
}

// Value Instantiation
/**
 * Event indicating that the component's value has been
 * instantiated.<br/>
 *
 * @param <C>
 *         base type managed by the component
 * @author Vivian Steller
 * @since 1.0
 */

```

```
static class ValueInstantiated<C>
    extends
        Events.ValueInstantiated<Component<C>, C,
            ValueInstantiationListener<C>>
{
    public ValueInstantiated(Component<C> source, C newValue)
    {
        super(source, newValue);
    }

    protected void call(ValueInstantiationListener<C> listener)
    {
        listener.handle(this);
    }
}

// Value Change
/**
 * Event indicating that the component's value changed.<br/>
 * <br/>
 * Change detection is implemented transitively, that is, if one of
 * the component's attributes or parts changes, the change is
 * propagated to the component and the respective event is fired.
 *
 * @param <C>
 *         base type managed by the component
 * @see Component#getValue()
 * @author Vivian Steller
 * @since 1.0
 */
static class ValueChanged<C> extends
    Events.ValueChanged<Component<C>, C, ValueChangeListener<C>>
{
    public ValueChanged(Component<C> source, C oldValue, C newValue)
    {
        super(source, oldValue, newValue);
    }

    protected void call(ValueChangeListener<C> listener)
    {
        listener.handle(this);
    }

    protected static <C> ValueChanged<C> create(
        Component<C> source,
        C oldValue, C newValue)
    {
        return new ValueChanged<C>(source, oldValue, newValue)
        {
            {
            };
        }
    }
}

// Type Change
/**
 * Event indicating that the component's type changed.<br/>
 *
 * @param <C>
 *         bae type managed by the component
 * @see Component#getType()
 * @author Vivian Steller
 * @since 1.0
 */

```

```

static class TypeChanged<C> extends
    Events.TypeChanged<Component<C>, C, TypeChangeListener<C>>
{
    public TypeChanged(Component<C> source,
        Class<? extends C> oldType, Class<? extends C> newType)
    {
        super(source, oldType, newType);
    }

    protected void call(TypeChangeListener<C> listener)
    {
        listener.handle(this);
    }
}

// Reset
/**
 * Event indicating that the component has been reset.<br/>
 *
 * @param <C>
 *         base type managed by the component
 * @see Component#reset()
 * @author Vivian Steller
 * @since 1.0
 */
static class Reset<C> extends
    Events.Reset<Component<C>, ResetListener<C>>
{
    public Reset(Component<C> source)
    {
        super(source);
    }

    protected void call(ResetListener<C> listener)
    {
        listener.handle(this);
    }
}

// V I S I T O R

/**
 * Visitor interface that is used to implement the visitor design
 * pattern on component hierarchies.
 *
 * @author Vivian Steller
 * @since 1.0
 */
static interface Visitor
{
    /**
     * Invoked by {@link Component#accept(Visitor)} for each
     * component in the hierarchy.
     *
     * @param component
     *         the component to be visited
     */
    void visit(Component<?> component);

    /**
     * Invoked by {@link Component#accept(Visitor)} for each
     * attribute in the hierarchy.
     *

```

```
        * @param attribute
        *         the attribute to be visited
        */
        void visit(Attribute<?, ?> attribute);

    /**
     * Invoked by {@link Component#accept(Visitor)} for each part in
     * the hierarchy.
     *
     * @param part
     *         the part to be visited
     */
    void visit(Part<?, ?> part);
}
}
```

Interface Attribute

```
package configurator.model;

import configurator.event.Events;
import configurator.event.Listener;
import configurator.metadata.AttributeDescriptor;

/**
 * Represents an attribute of type A. This interface allows
 * you to generically manage simple typed properties of
 * JavaBean-style instances.<br/>
 * <br/>
 * Opposed to {@link Part}s, attributes are "atomic" and not further
 * divisible, which means that their value has to be set "at once".
 * Further more, again in contrast to {@link Part}s whose value's type
 * may vary at runtime, the type of the attribute's value is fixed.<br/>
 * <br/>
 * There exist two subtypes of this interface {@link SingularAttribute}
 * and {@link PluralAttribute} to distinguish between non-collection
 * valued and collection valued attributes.<br/>
 * <br/>
 * Additionally, this interface supports registering of observers for
 * various events that indicate changes to the attribute's state.<br/>
 * <br/>
 * Implementations of this class do not have to be implemented
 * thread-safe but serializable.<br/>
 * <br/>
 * <i>Modeling note: methods are designed to be idempotent</i>
 *
 * @param <C>
 *         base type of the component containing the attribute
 * @param <A>
 *         type of the value managed by this attribute
 * @see SingularAttribute
 * @see PluralAttribute
 */
public abstract interface Attribute<C, A> extends Element<A>
{
    /**
     * Descriptor containing metadata about the attribute.<br/>
     * <br/>
     * The returned {@link AttributeDescriptor} is immutable.
     *
     * @param <T>
     *         type that declares the attribute
     * @param <A>
     */
}
```

```

    *           type of this attribute's value
    * @return the {@link AttributeDescriptor} associated to the
    *           attribute
    */
<T extends C> AttributeDescriptor<T, A> getMetadata();

/**
 * Name of the attribute.
 *
 * @return the name of the attribute
 */
String getName();

/**
 * Component that this attribute is attached to.
 *
 * @return the component containing this attribute
 */
Component<C> getParent();

/**
 * Whether the attribute value is instantiated (non-
 * <code>>null</code>) or not.
 *
 * @return <code>true</code> if the attribute is instantiated
 * @see #getValue()
 * @see #setValue(Object)
 */
boolean isInstantiated();

/**
 * Whether the user did specify the value of this attribute.
 *
 * @return <code>true</code> if the user specified the value of this
 *           attribute
 */
boolean isSpecified();

/**
 * Accesses the value managed by this attribute. The value must have
 * been explicitly set by the user using the
 * {@link #setValue(Object)} method or the containing component's
 * type is instantiated and provides a default value for the
 * attribute.
 *
 * @return the current attribute's value or <code>>null</code> if the
 *           attribute's value has not been set yet
 * @see #setValue(Object)
 */
A getValue();

/**
 * Sets the value of the attribute. <br/>
 * Calling this method triggers the following events if changes to
 * the attribute's state are detected: {@link Changed}
 *
 * @param value
 *           the attribute value to set
 */
void setValue(A value);

/**
 * Resets the attribute's state. The attribute value is set to

```

```
* <code>null</code> or the default value (refer to
* {@link #getValue()}).<br/>
* <br/>
* Calling this method triggers the following events:
* {@link Changed}
*/
void reset();

// L I S T E N E R S

/**
 * Registers a listener for the {@link Changed} event.
 *
 * @param listener
 *         the listener to register
 * @see Changed
 */
void addChangeListener(ChangeListener<C, A> listener);

/**
 * Unregisters a listener for the {@link Changed} event.
 *
 * @param listener
 *         the listener to unregister
 * @see Changed
 */
void removeChangeListener(ChangeListener<C, A> listener);

/**
 * Event listener for the {@link Changed} event.
 *
 * @param <C>
 *         base type of the component containing the attribute
 * @param <A>
 *         type of the changed attribute's value
 * @see Changed
 * @author Vivian Steller
 * @since 1.0
 */
interface ChangeListener<C, A> extends Listener
{
    void handle(Changed<C, A> event);
}

// E V E N T S

// Change
/**
 * Event indicating that the attribute's state changed.
 *
 * @param <C>
 *         base type of the component containing the attribute
 * @param <A>
 *         type of the changed attribute's value
 * @author Vivian Steller
 * @since 1.0
 */
class Changed<C, A> extends
    Events.Changed<Attribute<C, A>, ChangeListener<C, A>>
{
    public Changed(Attribute<C, A> source)
    {
        super(source);
    }
}
```

```

        }

        protected void call(ChangeListener<C, A> listener)
        {
            listener.handle(this);
        }
    }
}

```

Interface Part

```

package configurator.model;

import configurator.event.Events;
import configurator.event.Listener;
import configurator.metadata.PartDescriptor;

/**
 * Represents a part relationship between a component of type
 * <code>C</code> and one or more other components of type
 * <code>P</code>. This interface allows you to generically manage
 * <i>complex typed</i> properties of JavaBean-style instances.<br/>
 * <br/>
 * Opposed to {@link Attribute}s, parts are "complex" and may itself
 * contain other attributes or parts (or more precisely the referenced
 * part component can do so). The value of a part may explicitly set or
 * may be automatically instantiated by the framework while the user
 * specifies the concrete type, specific attribute or sub-part values.
 * Further more, in contrast to {@link Attribute}s whose value's type is
 * fixed, the part's type may vary at runtime according to the rules
 * defined by {@link Component#setType(Class)}.<br/>
 * <br/>
 * There exist two subtypes of this interface {@link SingularPart} and
 * {@link PluralPart} to distinguish between non-collection valued and
 * collection valued parts.<br/>
 * <br/>
 * Additionally, this interface supports registering of observers for
 * various events that indicate changes to the part's state.<br/>
 * <br/>
 * Implementations of this class do <i>not</i> have to be implemented
 * thread-safe but serializable.<br/>
 * <br/>
 * <i>Modeling note: methods are designed to be idempotent</i>
 *
 * @param <C>
 *     base type of the component containing the part
 * @param <P>
 *     base type of the value managed by this part
 * @see SingularPart
 * @see PluralPart
 * @author Vivian Steller
 * @since 1.0
 */
public abstract interface Part<C, P> extends Element<P>
{
    /**
     * Descriptor containing metadata about the part. Although the part
     * component's type can vary at runtime, the descriptor returned by
     * this method remains constant as it represents information about
     * the part itself and not information about the component
     * referenced by this part.<br/>
     * <br/>
     * The returned {@link PartDescriptor} is <i>immutable</i>.
     */

```

```
*
* @param <T>
*           type that declares the part
* @param <P>
*           base type of this part's value
* @return the {@link PartDescriptor} associated to the attribute
*/
<T extends C> PartDescriptor<T, P> getMetadata();

/**
 * Name of the part.
 *
 * @return the name of the part
 */
String getName();

/**
 * Component that this part is attached to.
 *
 * @return the component containing this part
 */
Component<C> getParent();

/**
 * Whether the part value is instantiated (non-null) or
 * not.<br/>
 * <br/>
 * This method delegates to the referenced part component(s).
 *
 * @return true if the part is instantiated
 * @see #getValue()
 * @see #setValue(Object)
 */
boolean isInstantiated();

/**
 * Whether the user did specify the value of this part or any of the
 * part component's attributes or parts.<br/>
 * <br/>
 * This method delegates to the referenced part component(s).
 *
 * @return true if the user specified any of this
 *         part's properties
 */
boolean isSpecified();

/**
 * Accesses the instance(s) managed by this part. The bean might
 * have been automatically instantiated during the configuration
 * process or must have been explicitly set by the user using the
 * {@link #setValue(Object)} method. Alternatively, the method
 * returns a default value if the containing component's type is
 * instantiated and provides a default value for the part<br/>
 * <br/>
 * This method delegates to the referenced part component(s).
 *
 * @return the underlying bean instance or null if the
 *         part has not been instantiated yet
 * @see #setValue(Object)
 */
P getValue();

/**
```

```

    * Explicitly sets the underlying bean instance. The part
    * component(s)' attributes and parts are populated with the given
    * value in order to reflect the bean instance's state.<br/>
    * <br/>
    * Calling this method triggers the following events if changes to
    * the part's current state are detected: {@link ValueChanged},
    * {@link Changed} and the events triggered by calling this part
    * component(s)' {@link Component#setValue(Object)} method.<br/>
    * <br/>
    * This method delegates to the referenced part component(s).
    *
    * @param value
    *         the value to populate this part with
    */
void setValue(P value);

/**
 * Resets the part's state. The part component(s)' type is reset to
 * their base type <code>P</code>. The managed instance is set to
 * <code>null</code> or the default value (refer to
 * {@link #getValue()} ).<br/>
 * <br/>
 * Calling this method triggers the following events:
 * {@link Changed} and the events triggered by calling this part
 * component(s)' {@link Component#reset()} method. <br/>
 * This method delegates to the referenced part component(s).
 */
void reset();

// E V E N T S

/**
 * Registers a listener for the {@link Changed} event.
 *
 * @param listener
 *         the listener to register
 * @see Changed
 */
void addChangeListener(ChangeListener<C, P> listener);

/**
 * Unregisters a listener for the {@link Changed} event.
 *
 * @param listener
 *         the listener to unregister
 * @see Changed
 */
void removeChangeListener(ChangeListener<C, P> listener);

/**
 * Registers a listener for the {@link ValueChanged} event.
 *
 * @param listener
 *         the listener to register
 * @see ValueChanged
 */
void addValueChangeListener(ValueChangeListener<C, P> listener);

/**
 * Unregisters a listener for the {@link ValueChanged} event.
 *
 * @param listener
 *         the listener to unregister

```

```
* @see ValueChanged
*/
void removeValueChangeListener(ValueChangeListener<C, P> listener);

/**
 * Event listener for the {@link Changed} event.
 *
 * @param <C>
 *     base type of the component containing the part
 * @param <P>
 *     base type of the changed part's referenced component
 * @see Changed
 * @author Vivian Steller
 * @since 1.0
 */
interface ChangeListener<C, P> extends Listener
{
    void handle(Changed<C, P> event);
}

/**
 * Event listener for the {@link ValueChanged} event.
 *
 * @param <C>
 *     base type of the component containing the part
 * @param <P>
 *     base type of the changed part's referenced component
 * @see ValueChanged
 * @author Vivian Steller
 * @since 1.0
 */
interface ValueChangeListener<C, P> extends Listener
{
    void handle(ValueChanged<C, P> event);
}

// Change
/**
 * Event indicating that the part's state changed.<br/>
 * <br/>
 * This event is fired upon any part state changes which corresponds
 * to the state changes of this part's referenced component(s).<br/>
 * <br/>
 * Change detection is implemented transitively, that is, if one of
 * the part component(s)' attributes or parts changes, the change is
 * propagated to the part and the respective event is fired.
 *
 * @param <C>
 *     base type of the component containing the part
 * @param <P>
 *     base type of the changed part's referenced component
 * @author Vivian Steller
 * @since 1.0
 */
class Changed<C, P> extends
    Events.Changed<Part<C, P>, ChangeListener<C, P>>
{
    public Changed(Part<C, P> source)
    {
        super(source);
    }

    protected void call(ChangeListener<C, P> listener)
```

```

        {
            listener.handle(this);
        }
    }

    // Value Change
    /**
     * Event indicating that the part's value changed.<br/>
     * <br/>
     * Change detection is implemented transitively, that is, if one of
     * the part component(s)' attributes or parts changes, the change is
     * propagated to the part and the respective event is fired.
     *
     * @param <C>
     *         base type managed by the component
     * @param <P>
     *         base type of the changed part's referenced component
     * @see Part#getValue()
     * @author Vivian Steller
     * @since 1.0
     */
    class ValueChanged<C, P> extends
        Events.ValueChanged<Part<C, P>, P, ValueChangeListener<C, P>>
    {
        public ValueChanged(Part<C, P> source, P oldValue, P newValue)
        {
            super(source, oldValue, newValue);
        }

        protected void call(ValueChangeListener<C, P> listener)
        {
            listener.handle(this);
        }
    }
}

```

Interface ComponentDescriptor

```

package configurator.metadata;

import java.util.Map;

import configurator.model.Facетted;

/**
 * Holds meta data for a configurable type <code>C</code>.
 */
public interface ComponentDescriptor<C> extends Facетted
{
    /**
     * The type that this descriptor describes.
     *
     * @return the Java class that is described by this component
     *         descriptor
     */
    Class<C> getValueType();

    /**
     * Attributes of <code>C</code>. Maps attribute names to the
     * respective descriptors.<br/>
     * <br/>
     * The returned map is <i>unmodifiable</i>.
     */
}

```

```
    * @return the map of attributes of <code>C</code>
    */
    Map<String, AttributeDescriptor<C, ?>> getAttributes();

    /**
     * Parts of <code>C</code>. Maps part names to the respective
     * descriptors.<br/>
     * <br/>
     * The returned map is <i>unmodifiable</i>.
     *
     * @return the map of parts of <code>C</code>
     */
    Map<String, PartDescriptor<C, ?>> getParts();

    boolean hasFacet(Class<?> facetType);

    <F> F getFacet(Class<F> facetType);
}

```

Interface AttributeDescriptor

```
package configurator.metadata;

import java.lang.reflect.AnnotatedElement;
import java.lang.reflect.Member;
import java.util.Collection;

import configurator.model.Typed;
import configurator.util.ValueDelegate;

/**
 * Holds meta data of an attribute of component <code>C</code>. The
 * attribute's value is of type <code>A</code>.
 *
 * @param <C>
 *         type of the class that declares the attribute
 * @param <A>
 *         type of the attribute's value
 */
public abstract interface AttributeDescriptor<C, A>
{
    /**
     * Descriptor that contains the attribute.
     *
     * @return the descriptor that contains the attribute
     */
    ComponentDescriptor<C> getComponentDescriptor();

    /**
     * Type that declares the attribute.
     *
     * @return the type that declares the attribute
     */
    Class<C> getDeclaringType();

    /**
     * Member that this attribute is built from.
     *
     * @return the Java bean member that this attribute is built from
     */
    Member getDeclaringMember();
}

```

```

/**
 * Annotated element that this attribute is built from.
 *
 * @return the annotated Java bean element that this attribute is
 *         built from
 */
AnnotatedElement getDeclaringAnnotatedElement();

/**
 * Name of the attribute.
 *
 * @return the name of the attribute
 */
String getName();

/**
 * Type of the attribute which corresponds to type of the field or
 * accessor method that this attribute maps to. E.g.
 * <code>Collection<A></code> if this is a collection attribute.
 *
 * @return the type of the attribute
 */
Class<A> getValueType();

/**
 * Whether this attribute is a collection.
 *
 * @return <code>true</code> if this attribute is {@link Collection}
 *         valued
 */
boolean isCollection();

/**
 * Whether this attribute is read-only or can be written.
 *
 * @return <code>true</code> if this is a read-only attribute
 */
boolean isReadOnly();

/**
 * Factory method to create a {@link ValueDelegate} that is capable
 * of reading or writing the attribute value on the given type
 * <code>C</code>.
 *
 * @param instance
 *        the instance from/to which the attribute value is
 *        read/written
 * @return {@link ValueDelegate} that operates on the given instance
 *         or <code>null</code> if either instance is
 *         <code>null</code> or instance doesn't contain the
 *         property, referenced by this attribute.
 */
ValueDelegate<A> createValueDelegate(C instance);

/**
 * @param <C>
 *        type of the class that declares the attribute
 * @param <A>
 *        type of the attribute's value
 * @author Vivian Steller
 * @since 1.0
 */
interface SingularAttributeDescriptor<C, A> extends

```

```
    AttributeDescriptor<C, A>, Typed<A>
{
}

/**
 * @param <C>
 *      type of the class that declares the attribute
 * @param <A>
 *      type of the attribute's value
 * @param <E>
 *      element type of the collection
 * @author Vivian Steller
 * @since 1.0
 */
interface PluralAttributeDescriptor<C, A, E> extends
    AttributeDescriptor<C, A>, Typed<E>
{
}

/**
 * Builder interface for creating (immutable)
 * {@link AttributeDescriptor}s.
 *
 * @author Vivian Steller
 * @since 1.0
 */
public static interface Builder<C, A, T>
{
    /**
     * Specifies the component descriptor that owns the attribute
     * being built. The passed component descriptor might not be
     * fully initialized yet, that is, not all attributes, parts and
     * facets have been instantiated and added to the component
     * descriptor.
     *
     * @param componentDescriptor
     *      the (not necessarily fully initialized) component
     *      descriptor that owns the built attribute
     * @return the builder for fluent invocations
     */
    Builder<C, A, T> owningDescriptor(
        ComponentDescriptor<C> componentDescriptor);

    /**
     * Specifies the property that the built descriptor is bound to.<br/>
     * <br/>
     * The specification of the property is <em>required</em>.
     *
     * @param property
     *      the property to that the built instance should be
     *      bound
     * @return the builder for fluent invocations
     */
    Builder<C, A, T> property(Property<A> property);

    /**
     * Specifies the component type that declares the attribute.<br/>
     * <br/>
     * The specification of the component type is <em>optional</em>
     * and defaults to the property's
     * {@link Property#getDeclaringClass()} property.
     *
     * @param declaringType

```

```

        *           the component type that declares the attribute
        * @return the builder for fluent invocations
        */
Builder<C, A, T> declaringType(Class<C> declaringType);

/**
 * Specifies the name of the built attribute.<br/>
 * <br/>
 * The specification of the name is <em>optional</em> and
 * defaults to the property's {@link Property#getName()}
 * property.
 *
 * @param name
 *           the name of the built attribute
 * @return the builder for fluent invocations
 */
Builder<C, A, T> name(String name);

/**
 * Specifies the type of the built attribute's bound property.<br/>
 * <br/>
 * The specification of the member type is <em>optional</em> and
 * defaults to the property's {@link Property#getJavaClass()}
 * property.
 *
 * @param memberType
 *           the member type of the built attribute's bound
 *           property
 * @return the builder for fluent invocations
 */
Builder<C, A, T> memberType(Class<A> memberType);

/**
 * Specifies the type of the built attribute.<br/>
 * <br/>
 * The specification of the type is <em>optional</em> and is by
 * default determined from to the property's
 * {@link Property#getJavaClass()} property.
 *
 * @param baseType
 *           the base type of the built attribute
 * @return the builder for fluent invocations
 */
Builder<C, A, T> baseType(Class<T> baseType);

/**
 * Retrieves the attribute descriptor instance built by this
 * builder. If this method is called by framework, the component
 * descriptor passed to the builder via the
 * {@link #owningDescriptor(ComponentDescriptor)} method may not
 * be fully initialized, that is, not all attributes, parts and
 * facets have been instantiated yet.<br/>
 * <br/>
 * Refer to {@link #initialize()} which is called when the
 * component descriptor is fully initialized.
 *
 * @return the (not necessarily fully initialized) attribute
 *         descriptor
 */
AttributeDescriptor<C, A> build();

/**
 * Finally initializes the attribute descriptor built by this

```

```
        * builder. Invoked when the descriptor that this attribute is
        * part of, has been fully initialized, that is, all attributes,
        * parts and facets (whose initialize method may or may not have
        * been called though) are available.
        *
        * @return the fully initialized attribute descriptor
        */
        AttributeDescriptor<C, A> initialize();
    }
}
```

Interface PartDescriptor

```
package configurator.metadata;

import java.lang.reflect.AnnotatedElement;
import java.lang.reflect.Member;
import java.util.Collection;

import configurator.model.Typed;
import configurator.util.ValueDelegate;

/**
 * Holds meta data of a part of component C. The part's
 * value is of type P.
 *
 * @param <C>
 *         type of the class that declares the part
 * @param <P>
 *         type of the part's value
 */
public abstract interface PartDescriptor<C, P>
{
    /**
     * Descriptor that contains the part.
     *
     * @return the descriptor that contains the part
     */
    ComponentDescriptor<C> getComponentDescriptor();

    /**
     * Type that declares the part.
     *
     * @return the type that declares the part
     */
    Class<C> getDeclaringType();

    /**
     * Member that this part is built from.
     *
     * @return the Java bean member that this part is built from
     */
    Member getDeclaringMember();

    /**
     * Annotated element that this part is built from.
     *
     * @return the annotated Java bean element that this part is built
     *         from
     */
    AnnotatedElement getDeclaringAnnotatedElement();

    /**
```

```

    * Name of the part.
    *
    * @return the name of the part
    */
    String getName();

    /**
     * Base type of the part which corresponds to type of the field or
     * accessor method that this part maps to. E.g.
     * <code>Collection<P></code> if this is a collection part.
     *
     * @return the base type of the part's value
     */
    Class<P> getValueType();

    /**
     * Whether this part is a collection.
     *
     * @return <code>true</code> if this part is {@link Collection}
     *         valued
     */
    boolean isCollection();

    /**
     * Factory method to create a {@link ValueDelegate} that is capable
     * of reading or writing the part value on the given type
     * <code>C</code>.
     *
     * @param instance
     *         the instance from/to which the part value is
     *         read/written
     * @return {@link ValueDelegate} that operates on the given instance
     *         or <code>null</code> if either instance is
     *         <code>null</code> or instance doesn't contain the
     *         property, referenced by this part.
     */
    ValueDelegate<P> createValueDelegate(C instance);

    /**
     * @param <C>
     *         type of the class that declares the part
     * @param <A>
     *         type of the part's value
     * @author Vivian Steller
     * @since 1.0
     */
    interface SingularPartDescriptor<C, P> extends
        PartDescriptor<C, P>, Typed<P>
    {
    }

    /**
     * @param <C>
     *         type of the class that declares the part
     * @param <A>
     *         type of the part's value
     * @param <E>
     *         element type of the collection
     * @author Vivian Steller
     * @since 1.0
     */
    interface PluralPartDescriptor<C, P, E> extends
        PartDescriptor<C, P>, Typed<E>

```

```
{
}

/**
 * Builder interface for creating (immutable) {@link PartDescriptor}
 * S
 *
 * @author Vivian Steller
 * @since 1.0
 */
public interface Builder<C, P, E>
{
    /**
     * Specifies the component descriptor that owns the part being
     * built. The passed component descriptor might not be fully
     * initialized yet, that is, not all attributes, parts and
     * facets have been instantiated and added to the component
     * descriptor.
     *
     * @param componentDescriptor
     *        the (not necessarily fully initialized) component
     *        descriptor that owns the built attribute
     * @return the builder for fluent invocations
     */
    Builder<C, P, E> owningDescriptor(
        ComponentDescriptor<C> componentDescriptor);

    /**
     * Specifies the property that the built descriptor is bound to.
     * <br/>
     * The specification of the property is <em>required</em>.
     *
     * @param property
     *        the property to that the built instance should be
     *        bound
     * @return the builder for fluent invocations
     */
    Builder<C, P, E> property(Property<P> property);

    /**
     * Specifies the component type that declares the part.<br/>
     * <br/>
     * The specification of the component type is <em>optional</em>
     * and defaults to the property's
     * {@link Property#getDeclaringClass()} property.
     *
     * @param declaringType
     *        the component type that declares the part
     * @return the builder for fluent invocations
     */
    Builder<C, P, E> declaringType(Class<C> declaringType);

    /**
     * Specifies the name of the built part.<br/>
     * <br/>
     * The specification of the name is <em>optional</em> and
     * defaults to the property's {@link Property#getName()}
     * property.
     *
     * @param name
     *        the name of the built part
     * @return the builder for fluent invocations
     */
}
```

```

        Builder<C, P, E> name(String name);

        /**
         * Specifies the type of the built part's bound property.<br/>
         * <br/>
         * The specification of the member type is <em>optional</em> and
         * defaults to the property's {@link Property#getJavaClass()}
         * property.
         *
         * @param memberType
         *         the member type of the built part's bound property
         * @return the builder for fluent invocations
         */
        Builder<C, P, E> memberType(Class<P> memberType);

        /**
         * Specifies the type of the built part.<br/>
         * <br/>
         * The specification of the type is <em>optional</em> and is by
         * default determined from to the property's
         * {@link Property#getJavaClass()} property.
         *
         * @param baseType
         *         the base type of the built part
         * @return the builder for fluent invocations
         */
        Builder<C, P, E> baseType(Class<E> baseType);

        /**
         * Retrieves the part descriptor instance built by this builder.
         * If this method is called by framework, the component
         * descriptor passed to the builder via the
         * {@link #owningDescriptor(ComponentDescriptor)} method may not
         * be fully initialized, that is, not all attributes, parts and
         * facets have been instantiated yet.<br/>
         * <br/>
         * Refer to {@link #initialize()} which is called when the
         * component descriptor is fully initialized.
         *
         * @return the (not necessarily fully initialized) part
         *         descriptor
         */
        PartDescriptor<C, P> build();

        /**
         * Finally initializes the attribute descriptor built by this
         * builder. Invoked when the descriptor that this part is part
         * of, has been fully initialized, that is, all attributes,
         * parts and facets (whose initialize method may or may not have
         * been called though) are available.
         *
         * @return the fully initialized attribute descriptor
         */
        PartDescriptor<C, P> initialize();
    }
}

```

Interface Product

```

package configurator.product;

import java.util.Collection;

```

```
import javax.enterprise.util.TypeLiteral;

import configurator.model.Attribute;
import configurator.model.Part;

public interface Product<T>
{
    /**
     * Name of the product. Usually the attribute annotated with
     * {@link info.openconfigurator.products.annotation.Product.Name}.
     *
     * @return the name of the product
     */
    String getName();

    /**
     * Description of the product. Usually the attribute annotated with
     * {@link info.openconfigurator.products.annotation.Product.Description}
     *
     * @return the description of the product
     */
    String getDescription();

    Asset getImage();

    /**
     * Attributes of the product. Usually attributes annotated with
     * {@link info.openconfigurator.products.annotation.Product.Attribute}
     *
     * @return the attributes of the product
     */
    Collection<Attribute<T, ?>> getAttributes();

    /**
     * Attributes of the product with the given level(s). Usually
     * attributes annotated with
     * {@link info.openconfigurator.products.annotation.Product.Attribute}
     * whereby the annotation's level attribute is specified.<br/>
     * <br/>
     * The level is interpreted as bit mask.
     *
     * @param level
     *         the level(s) to retrieve
     * @return the attributes of the product fulfilling the given levels
     */
    Collection<Attribute<T, ?>> getAttributes(int level);

    /**
     * Parts of the product.
     *
     * @return the parts of the product
     */
    Collection<Part<T, ?>> getParts();

    public static class Literal
    {
        public static <T> Class<Product<T>> forType(Class<T> type)
        {
            return new TypeLiteral<Product<T>>()
            {
                }.getRawType();
        }
    }
}
```

```

    }
}

```

Interface Configuration (Facet)

```

package configurator.configuration;

import java.util.Collection;

import configurator.model.Attribute;
import configurator.model.Part;

public interface Configuration<T>
{
    SpecificationMethod getSpecificationMethod();

    AttributeType getAttributeType(Attribute<T, ?> attribute);

    Collection<Attribute<T, ?>> getAttributes(AttributeType type);

    PartType getPartType(Part<T, ?> part);

    Collection<Part<T, ?>> getParts(PartType type);

    SpecificationMethod
        getSpecificationMethod(Part<T, ?> configuredPart);
}

```

Interface Data

```

package configurator.data;

import configurator.model.Attribute;
import configurator.model.Part;

public interface Data<T>
{
    // D O M A I N S

    DomainScope getDomainScope();

    /**
     * The initial component type domain. Domains contain values that
     * were subject to initial constraint application. Though, no user
     * specification has happened so far, which could have lead to
     * domain changes.
     */
    Domain<Class<? extends T>> getInitialComponentTypeDomain();

    Domain<Class<? extends T>> getComponentTypeDomain();

    /**
     * The initial component value domain. Domains contain values that
     * were subject to initial constraint application. Though, no user
     * specification has happened so far, which could have lead to
     * domain changes.
     */
    Domain<T> getInitialComponentValueDomain();

    Domain<T> getComponentValueDomain();

    /**
     * The initial attribute value domain. Domains contain values that

```

```
* were subject to initial constraint application. Though, no user
* specification has happened so far, which could have lead to
* domain changes.
*/
<A> Domain<A> getInitialAttributeValueDomain(
    Attribute<T, A> attribute);

<A> Domain<A> getAttributeValueDomain(Attribute<T, A> attribute);

// not yet implemented
Domain<Integer> getInitialAttributeQuantityDomain(
    Attribute<T, ?> attribute);

// not yet implemented
Domain<Integer>
    getAttributeQuantityDomain(Attribute<T, ?> attribute);

/**
 * The initial part type domain. Domains contain values that were
 * subject to initial constraint application. Though, no user
 * specification has happened so far, which could have lead to
 * domain changes.
 */
<P> Domain<Class<? extends P>> getInitialPartTypeDomain(
    Part<T, P> part);

<P> Domain<Class<? extends P>> getPartTypeDomain(Part<T, P> part);

/**
 * The initial part value domain. Domains contain values that were
 * subject to initial constraint application. Though, no user
 * specification has happened so far, which could have lead to
 * domain changes.
 */
<P> Domain<P> getInitialPartValueDomain(Part<T, P> part);

/**
 * May return <code>null</code>/unbound domain, if domain is not
 * explicitly defined using @Domain.Query on neither the part nor
 * the referenced component type. In this case, attribute level
 * domain definitions are assumed.
 */
<P> Domain<P> getPartValueDomain(Part<T, P> part);

// not yet implemented
Domain<Integer> getInitialPartQuantityDomain(Part<T, ?> part);

// not yet implemented
Domain<Integer> getPartQuantityDomain(Part<T, ?> part);

// D E F A U L T S

Class<? extends T> getDefaultComponentType();

T getDefaultComponentValue();

<A> A getDefaultAttributeValue(Attribute<T, A> attribute);

// not yet implemented
Integer getDefaultAttributeQuantity(Attribute<T, ?> attribute);

<P> Class<? extends P> getDefaultPartType(Part<T, P> part);
```

```

    <P> P getDefaultPartValue(Part<T, P> part);

    // not yet implemented
    Integer getDefaultPartQuantity(Part<T, ?> part);
}

```

Interface Domain

```

package configurator.data;

/**
 * Represents a domain of something.
 */
public interface Domain<T> extends Cloneable
{
    Class<T> getValueType();

    DomainType getDomainType();

    boolean isOrdered();

    boolean isNumeric();

    /**
     * Determines the number of values contained in the domain.
     *
     * @return the number of values.
     */
    int getSize();

    boolean isEmpty();

    boolean isFixed();

    T getValue();

    boolean isEnumerable();

    /**
     * Whether the domain's values are enumerable or not.
     *
     * @param threshold
     *         the maximum number of items, the domain is considered
     *         enumerable.
     * @return <code>true</code> if the domain's value are enumerable
     *         and the number of items is less than or equal to
     *         <code>threshold</code>.
     */
    boolean isEnumerable(int threshold);

    Iterable<T> getValues();

    boolean contains(T value);

    /**
     * Whether the domain's size is unlimited or not.
     *
     * @return <code>true</code> if the domain is limited,
     *         <code>false</code> otherwise.
     */
    boolean isBounded();

    boolean isBoundedAbove();
}

```

```
    boolean isBoundedBelow();

    T getUpperBound();

    T getLowerBound();
}
```

Interface Validation

```
package configurator.validation;

import java.util.Collection;

import configurator.model.Attribute;
import configurator.model.Part;

public interface Validation<T>
{
    Collection<Constraint<T>> getComponentConstraints();

    /**
     * Important: must also include those constraints, defined on type
     * level that reference attribute as target ?? this would
     * potentially require existence of a bean instance (which is
     * assumed for component level constraints anyway)?
     */
    <A> Collection<Constraint<A>> getAttributeConstraints(
        Attribute<T, A> attribute);

    <P> Collection<Constraint<P>> getPartConstraints(Part<T, P> part);

    /**
     * Explicitly triggers validation of all constraints defined for
     * this component, its attributes and parts.
     */
    void validate();

    /**
     * Whether the element itself has validation errors.
     */
    boolean hasViolations();

    /**
     * Returns the violations reported for that component. Includes
     * violations of all component, attribute and part constraints but
     * not those of nested components.
     */
    Collection<ConstraintViolation<?>> getViolations();
}
```

Interface Constraint

```
package configurator.validation;

import java.util.Collection;

import configurator.model.Element;

public interface Constraint<T>
{
    ConstraintDescriptor getMetadata();

    /**
```

```

        * The element that defines the constraint.
        */
        Element<T> getTarget();

        void validate();

        /**
         * Whether the constraint is satisfied (valid) or not.
         */
        boolean isSatisfied();

        Collection<ConstraintViolation<T>> getViolations();
    }

```

Interface ConstraintViolation

```

/**
 * (c) 2012 Copyright by Vivian Steller
 */
package configurator.validation;

import configurator.model.Element;

/**
 * @author Vivian Steller
 * @since 1.0
 */
public interface ConstraintViolation<T>
{
    Constraint<T> getConstraint();

    Element<T> getTarget();

    String getMessage();
}

```

Interface Agenda

```

package configurator.task;

import configurator.model.Component;

/**
 * Represents the main entry point of the task based API.
 */
public interface Agenda<T> extends Binding<Component<T>>
{
    <E> Iterable<Task<E>> getRootItems();

    <E> Iterable<Task<E>> selectItems(Filter filter);

    void accept(Task.Visitor visitor);
}

```

Interface Task

```

package configurator.task;

import java.util.Collection;

import configurator.data.Domain;
import configurator.validation.ConstraintViolation;

public interface Task<T>

```

```
{
    // structural
    Task<?> getParent();

    // task
    String getLabel();

    String getDescription();

    Domain<T> getDomain();

    T getValue();

    void setValue(T value);

    // task properties
    boolean isObsolete();

    boolean isOptional();

    // task status
    boolean isSpecified();

    boolean isValid();

    boolean isComplete();

    // validation errors
    Collection<ConstraintViolation<T>> getErrors();

    // visitor
    void accept(Visitor visitor);

    public static interface Visitor
    {
        boolean visit(Task<?> task);
    }
}
```

Bibliography

- [Abels2004] ABELS, H. *Kapitel Organisationsformen der Produktion der Vorlesung Produktionsplanung und -steuerung I+II. (WS 2004/2005).* Fachhochschule Köln, Fakultät für Fahrzeugsysteme und Produktion, Institut für Produktion, 2004.
- [Aldanando2002] ALDANANDO, Michel, et al. *Mass customization and configuration - Requirement analysis and constraint based modeling propositions.* , 2002.
- [Anderson1996] ANDERSON, David M. and PINE, B. Joseph II. *Agile Product Devevelopment for Mass Customization: How to Develop and Deliver Products for Mass Customization, Niche Markets, JIT, Build-To-Order and Flexible Manufacturing.* Irwin Professional Pub, 1996.
- [Andreasen1997] ANDREASEN, Mogens Myrup, HANSEN, Claus Thorp, and MORTENSEN, Niels Henrik. On the Identification of Product Structure Laws. *Proceedings of the 3RD workshop on product structuring.* 1997, p. 1-26.
- [Asikainen2004] ASIKAINEN, T., MÄNNISTÖ, T., and SOININEN, T. Using a configurator for modelling and configuring software product lines based on feature models. *Workshop on Software Variability Management for Product Derivation, Software Product Line Conference (SPLC3).* 2004.
- [Axel_Hahn2003] AXEL HAHN, Universität Oldenburg. Integriertes Produktkatalog- und Konfigurationsmanagement. *Industrie Management.* 2003, 1, p. 29 -32.
- [Barták2005] BARTÁK, Roman. *Constraint Propagation and Backtracking-based search.* , 2005.
- [Bieniek2001] BIENIEK, Christian. *Prozeßorientierte Produktkonfiguration zur integrierten Auftragsabwicklung bei Variantenfertigmern.* Shaker, 2001.
- [Blecker2004] BLECKER, T., et al. Product configuration systems: state of the art, conceptualization and extensions. . 2004.
- [Blecker2005] BLECKER, Thorsten, et al. *Information and management systems for product customization.* Springer Science+Business Media, 2005.
- [Bourke2000] BOURKE, R. Product configurators: key enabler for mass customization - an overview. *Midrange Enterprise.* 2000, vol. 8.
- [Brown2008] BROWN, David C. *Some Thoughts on Configuration Processes.* , 2008.
- [Businger1993] BUSINGER, A. *Expertensysteme für die Konfiguration - Architektur und Implementierung.* Philosophische Fakultät II, Universität Zürich, 1993.
- [Child1991] CHILD, P. and DIEDERICH, R. The management complexity. *McKinsey Quarterly.* 1991, vol. 4, p. 52-68.

- [Dadam2009] DADAM, Peter, et al. From ADEPT to AristaFlow BPM Suite: A Research Vision has become Reality. *Proceedings Business Process Management (BPM'09) Workshops, 1st Int'l. Workshop on Empirical Research in Business Process Management (ER-BPM '09)*. 2009.
- [Davis1987] DAVIS, Stanley M. *Future perfect*. Addison-Wesley, 1987.
- [Domschke2005] DOMSCHKE, Wolfgang and SCHOLL, Armin. *Grundlagen Der Betriebswirtschaftslehre*. Gabler Wissenschaftsverlage, 2005.
- [Eizaguirre2008] EIZAGUIRRE, F., et al. A CSP based distributed product configuration system. *Workshop on Configuration Systems*. 2008, p. 19.
- [Felfernig2000] FELFERNIG, A., FRIEDRICH, G. E., and JANNACH, D. UML as domain specific language for the construction of knowledge-based configuration systems. *International Journal of Software Engineering and Knowledge Engineering*. 2000, vol. 10, 4, p. 449–470.
- [Felfernig2001] FELFERNIG, Alexander, et al. *Distributed configuration as distributed dynamic constraint satisfaction*. , 2001.
- [Fielding2000] FIELDING, Roy T. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000.
- [Forza2002] FORZA, Cipriano and SALVADOR, Fabrizio. Managing for variety in the order acquisition and fulfilment process: The contribution of product configuration systems. *International Journal of Production Economics*. 2002, vol. 76, 1, p. 87-98.
- [Georg_Elsner2003] GEORG ELSNER, ORISA Software GmbH. Kundenbindung durch Produktkonfiguratoren und Baukästen. *Industrie Management*. 2003, 1, p. 33 -36.
- [Hallerbach2010] HALLERBACH, Alena, BAUER, Thomas, and REICHERT, Manfred. Capturing Variability in Business Process Models: The Provop Approach. *Journal of Software Maintenance and Evolution: Research and Practice*. 2010, vol. 22, 6-7, p. 519-546.
- [Hedin1998] HEDIN, G., OHLSSON, L., and MCKENNA, J. Product configuration using object oriented grammars. *System Configuration Management*. 1998, p. 107–126.
- [Hildebrand1997] HILDEBRAND, V. G. *Individualisierung als strategische Option der Marktbearbeitung: Determinanten und Erfolgswirkungen kundenindividueller Marketingkonzepte*. Deutscher Universitäts-Verlag, 1997.
- [Holthöfer2001] HOLTHÖFER, Norbert and SZILÁGYI, Sándor. *Marktstudie: Softwaresysteme zur Produktkonfiguration*. ALB-HNI-Verlagsschriftenreihe, 2001.
- [Homburg1996] HOMBURG, C. and WEBER, J. Individualisierte Produktion. *Handwörterbuch der Produktionswirtschaft*. 1996, p. 653–664.
- [Hvam2008] HVAM, Lars, MORTENSEN, Niels Henrik, and RIIS, Jesper. *Product customization*. Springer, 2008.
- [Ihl2006] IHL, C., et al. Kundenzufriedenheit bei Mass Customization: Eine empirische Untersuchung zur Bedeutung des Co-Design-Prozesses aus Kundensicht. *Die Unternehmung*. 2006, vol. 60, 3, p. 165–184.
- [Ilog2001] , . *ILOG Configurator - White Paper*. , 2001.
- [JSR2992009] KING, Gavin. *JSR-299: Contexts and Dependency Injection for the Java EE platform*. , 2009.

-
- [JSR3032009] BERNARD, E. and PETERSON, S. JSR 303: Bean validation, Version 1.0, Final Release. *Bean Validation Expert Group*. 2009.
- [JSR3172009] DEMICHIEL, Linda (Spec. Lead). JSR 317: *Java Persistence API, Version 2.0, Final Release*. Java Persistence API Expert Group, Sun Microsystems, 2009.
- [John1999] JOHN, U. and GESKE, U. *Reconfiguration of Technical Products Using ConBaCon.* , 1999.
- [John2002] JOHN, Ulrich. *Konfiguration und Rekonfiguration mittels Constraint-basierter Modellierung.pdf.* , 2002.
- [Josef_Wüpping2003] JOSEF WÜPPING, Dr. Wüpping Consulting GmbH. *Praxiserfahrungen Variantenmanagement und Produktkonfiguration*. *Industrie Management*. 2003, 1, p. 49 -52.
- [Jørgensen2003] JØRGENSEN, K. A. Information Models Representing Product Families. *Proceedings of 6th Workshop on Product Structuring, 23rd and 24th January 2003, Technical University of Denmark, Dept. of Mechanical Engineering*. 2003.
- [Knaplesch2010] KNUPLESCH, David, et al. On Enabling Data-Aware Compliance Checking of Business Process Models. *29th International Conference on Conceptual Modeling*. 2010.
- [Kolb2012] KOLB, Jens, HÜBNER, Paul, and REICHERT, Manfred. Automatically Generating and Updating User Interface Components in Process-Aware Information Systems. *20th International Conference on Cooperative Information Systems*. 2012.
- [Kotha1996] KOTHA, Suresh. From mass production to mass customization: The case of the National Industrial Bicycle Company of Japan. *European Management Journal*. 1996, vol. 14, 5, p. 442-450.
- [Kratochvíl2005] KRATOCHVÍL, Milan and CARSON, Charles. *Growing Modular: Mass Customization of Complex Products, Services And Software*. Springer, 2005.
- [Krug2010] KRUG, A. *Entwurf eines integrativen Grundmodells für Produktkonfiguratoren*. Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, 2010.
- [Kumar1992] KUMAR, Vipin. *Algorithms for Constraint-Satisfaction Problems - A Survey.* , 1992.
- [Künzle2011] KÜNZLE, Vera and REICHERT, Manfred. PHILharmonicFlows: towards a framework for object-aware process management. *Journal of Software Maintenance and Evolution: Research and Practice*. 2011, vol. 23, 4, p. 205–244.
- [Lanz2010] LANZ, Andreas, WEBER, Barbara, and REICHERT, Manfred. Workflow Time Patterns for Process-aware Information Systems. *Proceedings Enterprise, Business-Process, and Information Systems Modelling: 11th International Workshop BPMDS and 15th International Conference EMMSAD at CAiSE 2010*. 2010.
- [Leckner2004] LECKNER, T., STEGMANN, R., and SCHLICHTER, J. Reducing complexity for customers by means of a model-based configurator tool and personalized product recommendations. *Proceedings of the International Conference on Economic, Technical and Organizational Aspects of Product Configuration Systems*. 2004.
- [Leckner2006] LECKNER, Thomas. *Kundenkooperation beim Web-basierten Konfigurieren von Produkten*. Eul, 2006.
- [Li2005] LI, Jingxin. *A Novel Approach to Computer-Aided Configuration Design Based on Constraint Satisfaction Paradigm.* , 2005.

- [Lindemann2003] LINDEMANN, Udo (Hrsg.). *Marktnahe Produktion individualisierter Produkte. Arbeits- und Ergebnisbericht 2001-2004. SFB 582*. Technische Universität München. München, 2003.
- [Lindemann2006a] LINDEMANN, Udo, REICHWALD, Ralf, and ZÄH, Michael F. (ed.). *Individualisierte Produkte — Komplexität Beherrschen in Entwicklung Und Produktion*. Springer-Verlag, 2006.
- [Lindemann2006b] LINDEMANN, U. and BAUMBERGER, G. Individualisierte Produkte. In LINDEMANN, U., REICHWALD, R., and ZÄH, M. F. *Individualisierte Produkte — Komplexität Beherrschen in Entwicklung Und Produktion*. Springer-Verlag, 2006.
- [Lindemann2006c] LINDEMANN, U. and BAUMBERGER, G. Adaptionprozesse für individualisierte Produkte. In LINDEMANN, U., REICHWALD, R., and ZÄH, M. F. *Individualisierte Produkte — Komplexität Beherrschen in Entwicklung Und Produktion*. Springer-Verlag, 2006.
- [Lohrmann2012] LOHRMANN, Matthias and REICHERT, Manfred. Efficacy-aware Business Process Modeling. *20th International Conference on Cooperative Information Systems*. 2012.
- [Maher1990] MAHER, M. L. Process models for design synthesis. *AI magazine*. 1990, vol. 11, 4, p. 49.
- [Michael_Hüllenkremer2003] MICHAEL HÜLLENKREMER, camos Software und Beratung GmbH. Erfolgreiche Unternehmen arbeiten mit Produktkonfiguratoren. *Industrie Management*. 2003, 1, p. 37 -40.
- [Mundbrod2012] MUNDBROD, Nicolas, KOLB, Jens, and REICHERT, Manfred. Towards a System Support of Collaborative Knowledge Work. *1st Int'l Workshop on Adaptive Case Management (ACM'12), BPM'12 Workshops*. 2012.
- [Nareyek1999] NAREYEK, Alexander. *Structural Constraint Satisfaction*. , 1999.
- [Nilles2002] NILLES, Volker. *Effiziente Gestaltung von Produktordnungssystemen - Eine theoretische und empirische Untersuchung*. Ph.D. Dissertation, Technische Universität München, München, 2002.
- [OSGi_Alliance2004] , . *Listeners Considered Harmful: The Whiteboard Pattern*. OSGi Alliance, 2004.
- [Picot1982] PICOT, A. Transaktionskostenansatz in der Organisationstheorie. *Die Betriebswirtschaft*. 1982, vol. 42, 3, p. 267-284.
- [Piller1998] PILLER, Frank Thomas. *Kundenindividuelle Massenproduktion: Die Wettbewerbsstrategie der Zukunft*. Hanser, 1998.
- [Piller2001] PILLER, Frank Thomas. *Mass Customization. Ein wettbewerbsstrategisches Konzept im Informationszeitalter*. Deutscher Universitäts-Verlag, 2001.
- [Piller2003a] PILLER, Frank Thomas. *Mass Customization: Ein wettbewerbsstrategisches Konzept im Informationszeitalter*. Deutscher Universitäts-Verlag, 2003.
- [Piller2003b] PILLER, Frank and STOTKO, Christof. *Mass Customization und Kundenintegration. Neue Wege zum innovativen Produkt*. Symposion Publishing GmbH, 2003.
- [Piller2006] PILLER, Frank. *Mass Customization: Ein wettbewerbsstrategisches Konzept im Informationszeitalter*. Deutscher Universitätsverlag, 2006.
- [Pine1992] PINE, B. Joseph. *Mass Customization: The New Frontier in Business Competition*. Harvard Business Review Press, 1992.

-
- [Pine1999] PINE, B. Joseph II and GILMORE, James H. *The Experience Economy: Work Is Theater & Every Business a Stage*. Harvard Business School Press, 1999.
- [Polak2008] POLAK, Benjamin. *Kundenorientierte Gestaltung von Produktkonfiguratoren*. Universität St. Gallen, Hochschule für Wirtschafts-, Rechts- und Sozialwissenschaften (HSG), 2008.
- [Porter1980] PORTER, Michael E. *Competitive strategy: techniques for analyzing industries and competitors*. Free Press, 1980.
- [Porter1998] PORTER, Michael E. *Competitive Advantage: Creating and Sustaining Superior Performance*. Free Press, 1998.
- [Pryss2010] PRYSS, Rüdiger, et al. Towards Flexible Process Support on Mobile Devices. *Proc. CAiSE'10 Forum - Information Systems Evolution*. 2010.
- [Pulm2004] PULM, U. *Eine systemtheoretische Betrachtung der Produktentwicklung*. Technische Universität München, Universitätsbibliothek, 2004.
- [Reichert1998] REICHERT, Manfred and DADAM, Peter. ADEPTflex-Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems*. 1998, vol. 10, 2, p. 93-129.
- [Reichert2010] REICHERT, Manfred, et al. The Proviado Access Control Model for Business Process Monitoring Components. *Enterprise Modelling and Information Systems Architectures - An International Journal*. 2010, vol. 5, 3, p. 64-88.
- [Reichert2012] REICHERT, Manfred and WEBER, Barbara. *Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies*. Springer, 2012.
- [Reichwald2002] REICHWALD, R. and PILLER, F. T. Der Kunde als Wertschöpfungspartner: Formen und Prinzipien. *Wertschöpfungsmanagement als Kernkompetenz*, Wiesbaden, S. 2002, p. 27-51.
- [Reichwald2006a] REICHWALD, R., et al. Der Interaktions- und Kaufprozess für individualisierte Produkte. In LINDEMANN, U., REICHWALD, R., and ZÄH, M. F. *Individualisierte Produkte — Komplexität Beherrschen in Entwicklung Und Produktion*. Springer-Verlag, 2006.
- [Reichwald2006b] REICHWALD, R., et al. Marketing- und Vertriebswerkzeuge für individualisierte Produkte. In LINDEMANN, U., REICHWALD, R., and ZÄH, M. F. *Individualisierte Produkte — Komplexität Beherrschen in Entwicklung Und Produktion*. Springer-Verlag, 2006.
- [Reichwald2009] REICHWALD, Ralf and PILLER, Frank. *Interaktive Wertschöpfung: Open Innovation, Individualisierung und neue Formen der Arbeitsteilung*. Gabler Verlag, 2009.
- [Renneberg2010] RENNEBERG, Volker. *Adaptives, baukastenbasiertes Recommendersystem*. Eul, 2010.
- [Rogoll2003] ROGOLL, T. and PILLER, F. T. *Konfigurationssysteme für Mass Customization und Variantenproduktion: Marktstudie 2003; Strategie, Erfolgsfaktoren und Technologie von Systemen zur Kundenintegration*. ThinkConsult, 2003.
- [Rogoll2004] ROGOLL, T. and PILLER, F. Product configuration from the customer's perspective: A comparison of configuration systems in the apparel industry. *International Conference on Economic, Technical and Organisational aspects of Product Configuration Systems, Denmark*. 2004.
- [Runte2006] RUNTE, Wolfgang. *YACS - Ein hybrides Framework für Constraint-Solver zur Unterstützung wissensbasierter Konfigurierung*. , 2006.

- [Sabin1998] SABIN, Daniel and WEIGEL, Rainer. *Product Configuration Frameworks - A survey.* , 1998.
- [Scheer2006] SCHEER, C. *Kundenorientierter Produktkonfigurator: Erweiterung des Produktkonfiguratorkonzeptes zur Vermeidung kundeninitiiertter Prozessabbrüche bei Präferenzlosigkeit und Sonderwünschen in der Produktspezifikation.* Logos-Verl., 2006.
- [Schneeweiss2011] SCHNEEWEISS, Denny and HOFSTEDT, Petra. *FdConfig - A Constraint-Based Interactive Product Configurator.* , 2011.
- [Schuh2001] SCHUH, Günther and SCHWENK, Urs. *Produktkomplexität managen: Strategien - Methoden - Tools.* Hanser Fachbuch, 2001.
- [Schwarze1996] SCHWARZE, Stephan and SCHÖNSLEBEN, Paul. *Configuration of Multiple-Variant Products.* vdf Hochschulverlag AG, 1996.
- [Schönsleben2000] SCHÖNSLEBEN, P. *Integrales Logistikmanagement: Operations und Supply Chain Management innerhalb des Unternehmens und unternehmensübergreifend.* Springer, 2000.
- [Stumptner1997] STUMPTNER, M. An overview of knowledge-based configuration. *Ai Communications.* 1997, vol. 10, 2, p. 111–125.
- [Svensson2001] SVENSSON, C. and JENSEN, T. The customer at the final frontier of mass customisation. *Proceedings of the World Congress on Mass Customization and Personalization.* 2001, p. 1–8.
- [Tack2009] TACK, Guido. *Constraint Propagation.* , 2009.
- [Teresko1994] TERESKO, J. Mass customization or mass confusion. *Industry Week.* 1994, vol. 243, 12, p. 45–48.
- [Thorsten_Blecker2003] THORSTEN BLECKER, Herwig Dullnig und Franz Malle. Kundenkohärente und kunden-inhärente Produktkonfiguration in der Mass Customization. *Industrie Management.* 2003, 1, p. 21 -24.
- [Tiihonen] TIIHONEN, J., et al. Workshop on Configuration Systems. . .
- [Tiihonen1997] TIIHONEN, J. and SOININEN, T. *Product Configurators: Information System Support for Configurable Products.* Helsinki University of Technology, 1997.
- [Toffler1984] TOFFLER, Alvin. *The Third Wave.* Bantam, 1984.
- [Ulrich1991] ULRICH, K. T, TUNG, K., and MANAGEMENT, Sloan School of. Fundamentals of product modularity. *Issues in Design/Manufacture Integration 1991. American Society of Mechanical Engineers. Design Engineering Division (Publication) DE.* 1991, 39, p. 73-79.
- [Veron1999] VERON, Mathieu, FARGIER, Hélène, and ALDANONDO, Michel. *From CSP to configuration problems.* , 1999.
- [Zwicky1966] ZWICKY, Fritz. *Entdecken Erfinden , Forschen im Morphologischen Weltbild.* Knaur, 1966.

Ich erkläre, dass ich die Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den 13. August 2012

Vivian Steller
Matrikelnummer: 539132