ulm university universität
uulm

Ulm University | 89069 Ulm | Germany

Faculty of Engineering and Computer Science
Institute of Databases and Information Systems

Design and Development of a Run-time Object Design and Instantiation Framework for BPM Systems

Master's Thesis at Ulm University

Submitted by:
Kevin Andrews
kevin.andrews@uni-ulm.de

Reviewer:
Manfred Reichert
Peter Dadam

Supervisor:
Jens Kolb

2014

Version September 4, 2014

© 2014 Kevin Andrews

Abstract

Current Business Process Management (BPM) systems are not tailored to small or medium-sized enterprises (SME) lacking expertise in BPM. This is a disadvantage for SMEs wishing to document and/or automate their business processes. There are multiple barriers that can hinder SMEs from automating their processes, such as the lack of programming skills and general understanding of typical programming language data types. Also the multitude of gateway and event types in traditional BPM systems can have a deterring effect on potential process designers. Finally, most BPM systems require deployment to on-site servers, which might not be feasable for SMEs lacking dedicated systems administrators.

As a consequence this thesis and the work accompanying it define a concept for an integrated solution for simple cloud-based business process modeling and execution. This solution offers collaboration functions and rich process documentation via a modern web-interface, sketching support for rapid iterative process development, and an object-oriented (OO) data model. Created processes can be executed directly in a web-interface as part of the seamlessly integrated modeling and runtime environment. The main focus of the solution is simplicity in every aspect of the process creation and execution workflow.

This thesis contributes concepts and prototypical implementations for the engine modifications that need to be applied to a typical traditional BPM engine in order to support these features. The basic concepts and their implementations can be put into the following three categories, corresponding to the main chapters of this thesis:

- User-customizable data objects in process models, allowing for the creation of data models that are more readable and better structured
- A simple programming interface allowing the use of external code in service tasks, enabling non-professional programmers to integrate their code into processes
- Support for the manipulation of process flow to allow for advanced simplification concepts, such as error resolution at process run-time or manually selectable XOR-gateway paths

Contents

1

Introduction

1.1. Motivation

Data objects are an integral part of almost any electronically supported *business process*. They contain the information which instances of the business process models create, read, and update. The contents of a data object can vary, ranging from simply a name to the information of an entire person, e.g., birth date, address, and marital status. Information contained in a data object must be retained in a form where it is of use either to a human business process participant or a *business process management system (BPMS)*. In some cases the latter is not possible, for instance, when the data object in question is a scanned image of a document. As long as the image is just an image, with no attached meta-information, the BPMS does not have any understanding of its contents and can not make any decisions based on its contents.

This is where structured information in data objects, i.e., complex *business objects*, come into play. Instead of a data object containing only simple information or being a "black box" to the BPMS, information is structured into *business object types* that the BPMS has knowledge of. In the context of this thesis, a business object is defined as a structured data object. To be more precise, a business object contains not only the *data* that an equivalent data object would contain, but also a reference to a business object type. This business object type defines the business object's internal structure in a way that makes it usable in a BPMS. This allows us to define structures, such as *Person* using *attributes*, such as *Age*, *Name* or, *Marital Status,* which themselves are business objects.

There are multiple advantages of supporting such clearly structured business objects. For one, they offer the ability to group the multitudes of data objects usually present in a *process model* into fewer structured groups. Also, with the introduction of business object types, acting as a template of sorts for these business objects, it is possible to hide the internal complexity of the business objects from *process model designers*. This allows them to introduce complex business objects into their models by simply adding one new data object to the model. As this data object is a business object based on a business object type, it brings all its internal structure information and even default values for its attributes with it. This allows for a separation of concerns when building process models as one person can model the business object types and another can model the actual process model utilizing these complex structures in a simple, atomic style.

Furthermore, through the use of complex business objects it is possible to generate user forms at run-time automatically, based entirely on the internal structure of said business objects. This means that by building well structured complex business object types one can not only reduce the amount of data objects in a process model and hide information from non-technical users, but also reduce the need for designing custom user forms, as these can be generated. This is possible because the BPMS itself can read the structural information of any business object which is determined by its business object type.

Most currently available BPMSs support complex business object types in one way or another [3, 9, 22]. A common approach, seen for instance in Intalio|bpms [10], is to allow grouping of data object types, such as integers and strings, into XML *<complexType>* elements. Parts of these XML types can then be mapped to parameters of automatically executed tasks (service tasks) and form-based tasks for human interaction (user tasks). The disadvantage of this approach is actually its generic nature. XML schema, the description language for XML types, is so mighty and flexible that it is also inherently error prone. Also, users wishing to incorporate business processes into their workflow in small and medium-sized enterprises might not have the necessary know-how to create complex XML schema types, as these can consist of up to 42 different XML tags.

Another issue when dealing with complex business objects in BPMS is their use in external Java programs and algorithms which one might wish to use as part of a process model. Some BPMSs, such as the AristaFlow BPM Suite [2], allow using complex business objects using special Application Programming Interface (API) classes provided by the BPMS. This, however, forces the programmers of such *plug-ins* to update their programs when these APIs change. Usually this would not pose a problem, but in the cloud-based context that this thesis is based around, updating such an API offered by the BPMS can often break plug-ins already in use in various business processes. As the Clavii BPM Cloud is still a prototype and, therefore, work in progress, a method was developed to allow the use of complex business objects in external code without having to use any supplied APIs.

As previously mentioned, business object types can be structured in a way that allows the generation of user forms based on their structural information. This is also true when a business object is missing at run-time, for instance because of an error in a plug-in, or because the process model designer did not map it correctly. In this case the BPMS can analyze the business object type that the task in question is requesting and, based on its structure, generate a user form, allowing a user to input the entire business object at run-time.

Utilizing this ability to generate user forms in an ad-hoc fashion that can resolve data flow errors at run-time, this thesis also examines the possibilities of supporting iterative

process model development. Iterative process model development as proposed in this thesis allows process model designers to test process models without the data flow being complete, showing them generated error resolution forms at run-time. This is something many BPMSs do not support directly, Intalio|bpms, for instance, allows process model designers to start process instances in almost any state of completion, e.g., without data edges, end events or even without any control flow elements. This often results in crashes which are hard to debug. The other extreme, the AristaFlow BPM Suite, uses the so-called correctness by construction principle in which the process model design tool forces process model designers to build correct process models [34]. This even includes the correctness of the entire data flow. This does, however, limit the ability of process model designers to test process models that are not “perfect” yet, e.g., if there are some data edges missing in a branch of the model completely unrelated to the one they wish to test. As both these approaches have their advantages, a concept for a compromise is given in this thesis.

Finally, the main objective is to design the engine for a BPMS that has one primary goal: simplicity. This desired simplicity starts with a cloud and browser-based approach to both modeling and execution of process models, the simple design and usage of business objects and the generated forms for user input at run-time. It even extends to the way external code for use in Clavii BPM Cloud business processes is written. Offering a cloud-based solution eliminates the need for SMEs to set up a specialized process server infrastructure, including databases and web-servers, and employ administrators to keep said infrastructure running.

1.2. Contribution

The concepts described in this thesis add important modifications to the basic methods for handling complex business objects, such as the one seen in Intalio|bpms (cf. Section 1.1):

- Full flexibility for the definition business process types while the BPMS is running, i.e., infinite nesting and collections as parts of business object types

- Direct consumption and creation of complex business object instances through service tasks, without the implementation language having specific class information
- Simple re-usage of business object types throughout process models by providing global (or context specific) business object type definitions

These extensions to traditional BPMS are realized as proof of concept implementations on top of the Activiti BPM engine [1]. As all operations related to business objects are handled in our own code instead of relying on the Activiti BPM engine we can add additional functionality to our BPMS. As the complex business object instances are not just “black boxes” to our code, it was possible to implement so-called “run-time error resolution”. Run-time error resolution not only allows re-running of failed service tasks with different input values directly in the user interface, but also supports rapid process model prototyping. Rapid process model prototyping allows process model designers to test incomplete process models by asking them for missing input parameters when a service task located in an incomplete process model tries to execute, instead of just failing. Additionally, rapid process model prototyping is enhanced by the addition of “empty” tasks to the engine, i.e., tasks that can be completed with a simple double-click on the empty task at run-time. As empty tasks can be added to a process model with a single click and data flow does not have to be complete while testing, process models can be prototyped and tested very quickly.

Furthermore, when supporting complex business objects, the development of plug-ins, i.e., Java code that can be executed as part of a service task, has to be considered as well as there is a significant increase in complexity when utilizing complex business objects. Along with this support for advanced plug-in concepts this thesis also highlights the implementation concept of a trigger, a type of a plug-in that can be used to start the execution of a process instance. Plug-ins and triggers can both be coded without using any special Java class libraries. Also, plug-ins and triggers can be integrated into the BPMS while it is running. Furthermore, they can consume and create the complex business object instances that users can define. The approach commonly used in BPMS is to allow complex objects but force service tasks to only accept primitive or very simple complex types as valid input and output parameters. Plug-ins and triggers created as

described in this thesis can actually handle the complex business objects and collections of these objects natively, without using any special Java class library.

1.3. Structure of the Thesis

Figure 1.1 shows the topics covered in this thesis at a glance, including the sections in which the topics are covered in detail.

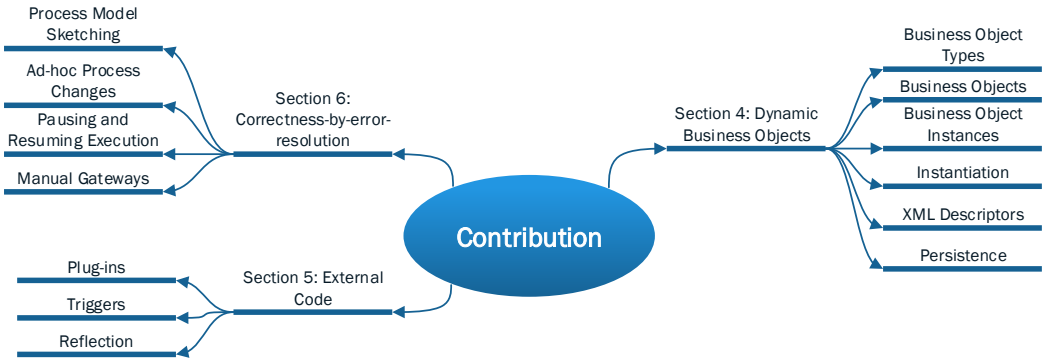

Figure 1.1.: Topics Covered in this Thesis

Section 2, defines the terminology used across the other sections of this thesis, including this introduction. Next, the requirements that were stated for the Clavii engine are listed and explained in Section 3. The three main implementation-related chapters are located directly after the requirements section. The first main section is Section 4 which explains the concept and implementation of complex business objects and the method the Clavii engine uses for allowing them to be instantiated. Then Section 5 explains how the Clavii engine supports the usage of external Java code and web services inside a Clavii BPM Cloud business process. Finally, Section 6 explains how ad-hoc pauses and rerouting of business process flow is implemented to allow for correcting data flow errors at run-time. After the main content sections of the thesis, Section 7 describes other, BPMSs that

have similar mechanics to Clavii in some regards. Section 7 also lists papers working on similar problems to this thesis. Finally, Section 8 contains the conclusion to this thesis.

2

Fundamentals

This section introduces terminology and concepts commonly referred to in this thesis. Section 2.1 introduces the notions of business processes, process models, and process instances. Different types of tasks supported in process models are introduced in Section 2.1.2. The Clavii BPM Cloud supports the integration of Java programs into the business process flow by allowing tasks to use “plug-ins”, which are explained in Section 2.2.1. Furthermore, Section 2.2.2 explains the three incarnations of business objects that exist in the Clavii BPM Cloud context.

2.1. Fundamentals of Business Process Management

A definition of a *business process* is given by Rummler & Brache: “A business process is a series of steps designed to produce a product or service” [35].

A *process model* is a graphical representation of a business process, i.e., a documentation of these steps and their interoperation, as well as the logic for executing the individual steps. BPMS enables process designers to create such a process model of a business process they wish to execute. This includes letting them define logic for executing the steps of the business process.

Furthermore, a *process instance* describes one business case, performing steps of a process model in a BPMS to produce the "product or service".

To illustrate relations between the notions of "*business process*", "*process model*" and "*process instance*", regard Example 1a:

Example 1a (Candy Shipping, process model versus process instance):
Alice wants to automate the online shopping for her weekly supply of candy on candydelivery.com. The service that the necessary business process provides is automating the ordering process on the web-site. Alice may use a BPMS to document the latter in a process model. To be more precise, all tasks required to order the candy are described in a BPMN process model. Later on she executes the process model in a BPMS. Executing the business process creates a process instance of said business process, ordering one weeks worth of candy. Each time the orders new candy a new process instance is created.

Figure 2.1.: Process Model for Example 1a

2.1.1. Business Process Model and Notation

The notation used in Figure 2.1 is the business process model and notation (BPMN) which is used throughout the thesis. The elements of BPMN that are used in this thesis are shown in a minimal example in Figure 2.2.

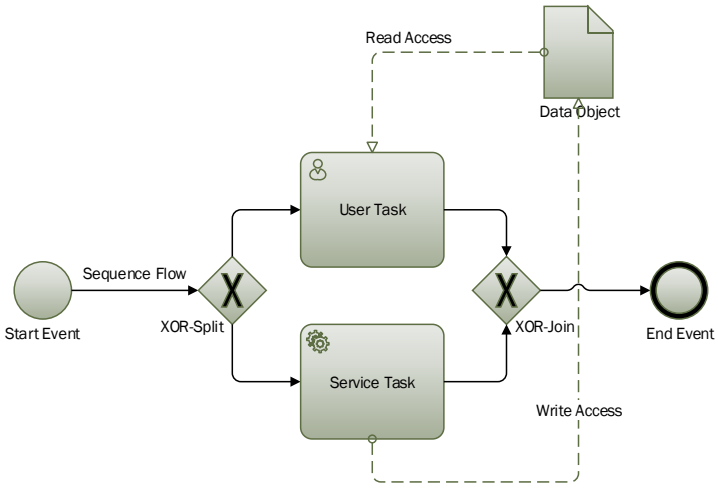

Figure 2.2.: BPMN Example

The *start event* denotes the starting point of the process model, this is where the *control flow* of the process, i.e., the execution, begins. Opposed to this is the *end event*, which symbolizes the end of the process model. The control flow of the process model follows the *sequence flow* arrows, which connect the *tasks* and *gateways*. The gateways used in the examples in this thesis are all of the *XOR* type, which can be seen used in Figure 2.2, denoted by the labels “XOR-Split” and “XOR-Join”. An XOR-gateway forces the control flow of the process to continue on exactly one of the sequence flows leaving the XOR-split gateway. The XOR-join gateway denotes the point in the control flow of the process model where the branches are rejoined.

2.1.2. Task Types

Apart from the sequence flows and gateways, a process model can contain various tasks. *Tasks* are the nodes of a process model on which actions can be performed by human process participants or the BPMS itself. A task may be further categorized as: empty task, user task, service task.

User tasks require a user to fill out a user form or provide information to a user. This is often done through the use of a web interface and is the primary way for a user to interact with a process instance.

Service tasks are tasks that are not meant for user interaction, i.e., they are executed by the BPMS. Examples of service tasks could be tasks that fetch information from a database or call a web service. The “logic” behind such service tasks is bundled into so-called *plug-ins*, which are reusable implementation artifacts (cf. Section 2.2.1) containing service task logic.

Empty tasks are tasks that are not yet set up as a user task or service task, i.e., they perform no specific action. Empty tasks do not perform any operations except allowing the execution of the process instance to continue after they are completed. They are mostly be used as placeholders for other task types in the early stages of process model creation (cf. Section 3.3.5). They can be used to symbolize work that can not be done at a computer, similar to a BPMN manual task.

Example 1b (Candy shipping, utilizing additional task types):
Adding tasks to the process from Example 1a, Alice specifies a user task *Input Order*, which provides a user form asking for type and quantity of candy to order. Additionally, Alice adds a service task to the process model which automatically sends an e-mail before the business process is completed, notifying Trudy that a new shipment has arrived.

Figure 2.3.: Process Model for Example 1b

2.1.3. Data Objects

Tasks in process models interact with *data objects* by reading or writing their values. Data objects can contain many kinds of values and act as containers for these. Concepts for extending these data objects to more sophisticated and complex business objects is one of the main focuses of this thesis.

Example 1c shows how data objects can be used in the context of the candy shipping process.

Example 1c (Candy shipping, utilizing data objects):
CandyOrder is a data object attached to the candy shipping process model. It can be mapped to the output parameter of the user task *Input Order* and to the input parameter of the service task *Order Candy*. The user task *Input Order* at the beginning of the process model shows Alice a form where she can write data to the *CandyOrder* data objects. Once she is done creating her order, the plug-in operation assigned to the *Order Candy* service task receives the data object *CandyOrder*, containing the ordering information, and places the order.

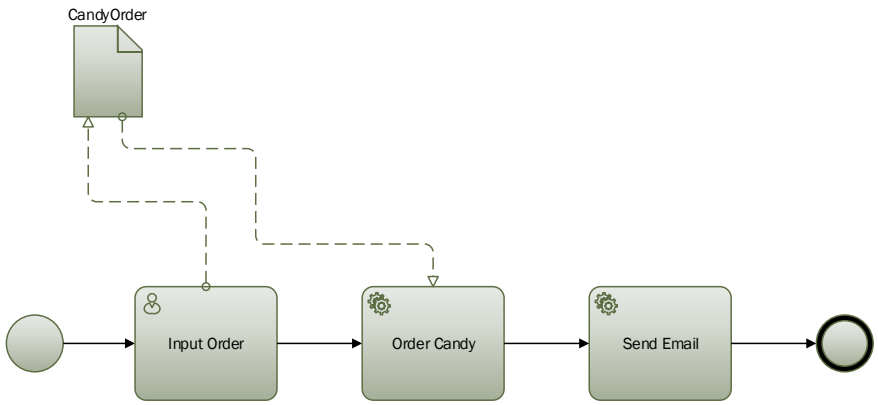

Figure 2.4.: Process Model for Example 1c

2.2. Fundamental Terminology

2.2.1. Plug-Ins

Plug-ins are executable components to be assigned to service tasks, i.e. they are generic implementations of the functionality a service task can offer. For example a plug-in could contain the logic required to connect to a file transfer protocol (FTP) server, allowing a service task to upload or download files using the FTP protocol. Plug-ins can contain multiple “operations”, each offering a different variant of the plug-in’s functionality. In the case of the FTP plugin-in possible operations could for instance be “upload” and “download”. An analogy in object-oriented programming languages for the relation of a plug-in to an operation would be to view the plug-in as a class and an operation as a method of said class.

A plug-in operation may have a fixed set of input and output parameters, which must be mapped to data objects using data edges (cf. Figure 2.5) in order for the operation to function.

In the context of Example 1b, the service task responsible for ordering items from candyshop.com has to have a plug-in and respective operation assigned to order candy. Business objects mapped to input parameters of this plug-in operation contain the amounts and sorts of candy to order. Furthermore, to send the e-mail notifying Trudy, the respective service task requires an e-mail plug-in offering an operation for sending e-mail messages.

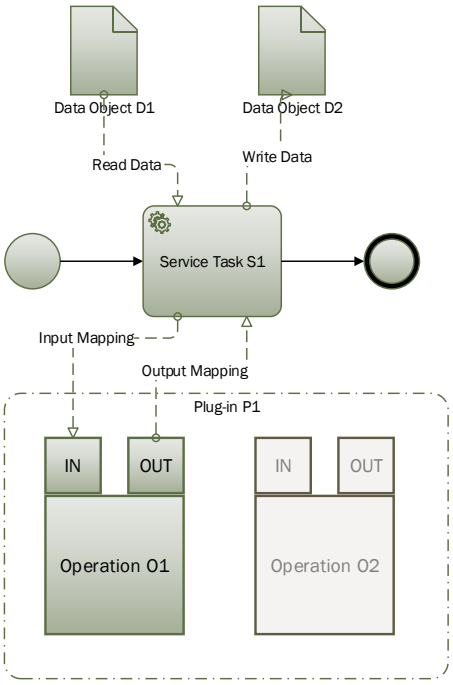

Figure 2.5.: Mapping of Data Objects to Parameters via a Service Task

2.2.2. Business Objects

In an object-oriented language such as Java or C#, an object is the instance of a type[1]. The type defines a structure of fields and the object, or “instance”, of said type holds concrete values for each field defined by the type. The creation of such an instance of a type is called instantiation. Business objects on the other hand are objects that represent an entity participating in a business process. Examples of business objects could be a person, an event, a document or any other structured information that a business process has to handle.

The “two-tier” approach used for normal objects (object type and object instance) is extended to a “three-tier” model for business objects in the Clavii BPM Cloud. The

[1] A broader term for “class”, including primitives, enums, structs, etc.

reasons for this are explained later in Section 3.1.2. The three “tiers” of business objects are as follows:

- Business object type
- Business object
- Business object instance

A *business object type* is a template for a business object defined globally, i.e., it is not process model specific. Such a template contains structural information of a business object, such as which forms of data it can hold, or, if the business object is more complex, which other business objects it contains. Business object types have a *name*, for example, “WordDocument”. A business object type can be used in a model, thereby creating a business object. This is necessary if a process model is supposed to use a business object of said type.

A *business object* is a process model-specific business object type. Its structure is defined by the corresponding business object type it was created from. Business objects are given a *name* which is unique in the context of a specific process model they are assigned to. A name for a business object of the *WordDocument* business object type might be “Recruitment Document”. Business objects may be mapped to input and output parameters of tasks in a business process by assigning respective data edges. Once the process model is executed, instances of all business objects are instantiated in the context of the new process instance. Business objects may be given default values that are used as the initial values for the business object instances. The data type of the default value, e.g., string, integer, is also defined in the business object type that corresponds to the business object that the default value belongs to. Any business object instances created from this business object can also only hold values of said data type.

A *business object instance* is an instance of a business object that is associated with exactly one process instance. Its structure and default values are defined by the business object it was instantiated from. Business object instances can hold values that can be manipulated by tasks of the process instance.

Example 1d shows how business objects can be used in the context of the candy shipping process.

Example 1d (Candy shipping, utilizing business objects):
A *CandyOrder* is a business object type collection capable of containing *Candy* business object types. In order for Alice to use the *CandyOrder* and *Candy* business object types in her business process, she has to use the business object type *CandyOrder* in her process model by giving it a name, e.g., “WeeklyCandyOrder”. The resulting business object is a collection of *Candy* business objects, which is empty by default. The “WeeklyCandyOrder” business object can be mapped to the output parameter of the user task “Input Order” and to the input parameter of the service task “Order Candy”. The user task “Input Order” at the beginning of the process model shows Alice a form where she can create instances of *Candy* business objects and add them to the “WeeklyCandyOrder” business object instance. Once she is done creating her list of orders, the plug-in operation assigned to the “Order Candy” service task receives the “WeeklyCandyOrder” business object instance, containing all the *Candy* instances, and places the order.

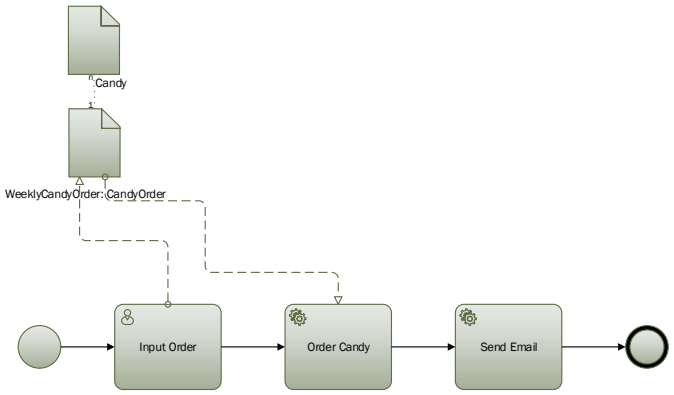

Figure 2.6.: Process Model for Example 1d

2.2.3. Additional Terminology

To better understand the different points in time referred to in this thesis the following definitions should be noted:

- Build-time
- Run-time

Build-time is the time span in which a process model designer creates a process model, this is only possible while the BPMS is running. Finally, run-time is the time span in which a process instance is executed, i.e., users are working with the finished process.

3

Requirements Analysis

The following sections contain requirements in order to support non-technical users in specifying complex data flow. All requirements concerning business objects are located in Section 3.1. Requirements concerning the interplay between complex data flow, service tasks and plug-ins can be found in Section 3.2. Finally, Section 3.3 describes the requirements concerning the mapping capabilities of complex data flow, process flow and resulting capabilities allowing for ad-hoc resolution of data flow errors.

3.1. Business Object Requirements

3.1.1. Complex Business Objects

In order to stand out from the multitude of existing BPMS, we offer a new approach to central aspects of business process modeling: the modeling of data flow and the

mapping of business objects to service and user tasks. The approach proposed in this thesis has multiple advantages, as explained in this section.

The engine of this new BPMS should support capsuling complex sets of attributes, like the server settings for an SMTP account, hiding typical attributes from the non-technical user, furthering the goal of simplicity [29]. An example is shown in the UML class diagram [32] in Figure 3.1.

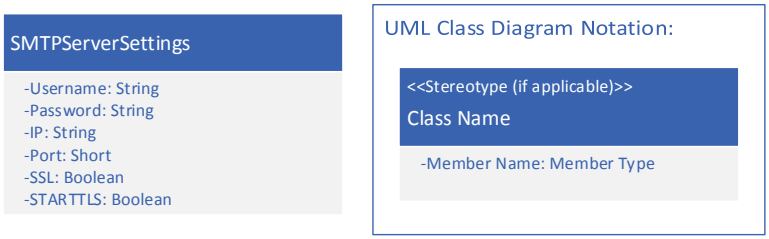

Figure 3.1.: Encapsulating SMTP Server Settings

The business object type *SMTPServerSettings* is defined by the IT department of the fictional “Contoso Ltd.” enterprise and given the default values of the enterprise’s SMTP server. Two things can be achieved by doing this, firstly the individual business objects that comprise the complex business object are capsuled and do not clutter the process model and thereby confuse users and secondly, users do not have to know the individual attributes and values necessary to communicate with the Contoso Ltd. SMTP Server. This is also in the interest of information hiding, a well established principle in programming.

These two advantages and the simplicity for users resulting from them lead us to formulate Requirement REQ-1.

Requirement REQ-1: (Complex Business Objects) A BPMS should support complex business objects.

3.1.2. Typed Business Objects

Consider the base *SMTPServerSettings* business object type, which consists of the properties listed in Table 3.1 (also cf. Figure 3.1).

Username	String
Password	String
IP	String
Port	Short
SSL	Boolean

Table 3.1.: *SMTPServerSettings* business object type

In a traditional BPMS implementation the input parameters for configuring a typical SMTP mailer operation would most likely consist of exactly those six properties. The problem here is that forcing process model designers to enter this data once for every usage of the SMTP mailer operation in the process model is error prone for two reasons: replication of the same data input at multiple places in the process model and, more importantly, the inability to type-check the data input. The BPMS might be intelligent enough to detect that a password entered into the IP address box is not valid and would not allow entering the port number in a field where a simple yes/no answer is expected. Most BPMS however, could not protect from mapping the username to the password input and vice versa, as they have no knowledge of the format that these two fields have to be in.

This is where the BPMS should offer help. Firstly, in this use case, the process model designer should only see the *SMTPServerSettings* business object type, created by a professional from the IT department, as a match for the input field of an emailing plug-in which expects a business object of exactly that business object type. Secondly, if the business object type framework proposed in this thesis is used in its strictest form, the internal structure of the *SMTPServerSettings* business object type shown in Table 3.2 would be plausible.

Username	Username
Password	Password
IP	IP Address
Port	Port Number
SSL	Decision

Table 3.2.: Strongly Typed *SMTPServerSettings* business object type

This means that the business object type *SMTPServerSettings* would not only consist of attributes represented by standard Java types, e.g., *String*, *Integer*, etc., but would in itself contain further custom or predefined business object types, like *IP Address* or *Port Number*. Again, this has multiple advantages: it allows the structure of business objects and the data model to be more readable for people who are not accustomed to the standard programming data types. Furthermore it also allows the designers of the business object types to define properties, like custom user interface display colors and even icons for each type. Additionally, custom regular expressions for applicable string-based simple business object types can be defined for form validation uses.

This increase of form input safety at build and run-time through the additional meta-information that can be added to simple business object types lead us to formulate Requirement REQ-2.

Requirement REQ-2: (Strongly Typed Business Objects) A BPMS should support strongly typed business objects.

3.1.3. Inheritance Capabilities for Business Object Types

Consider an extension to the example business object type *SMTPServerSettings* given in Section 3.1.2, based on the UML class diagram shown in Figure 3.1. The basic *SMTPServerSettings* business object type is extended to be the Contoso enterprise-specific *ContosoSMTPServerSettings* business object type. The “Username” field from the original business object type is changed from being of the normal *Username* business object type to a new *ContosoUsername* business object type, which itself extends the

Username business object type. This leaves the new *ContosoSMTPServerSettings* business object type fully type-compatible with the base *SMTPServerSettings* business object type, but allows changing of the regular expression assigned to the “Username” field (cf. Section 3.1.2). As the basic *Username* business object type has no regular expression assigned to it, since there is no general rule on username composition, a great amount of validation safety can be achieved by adding a simple regular expression to the derived business object type, thus limiting the valid values to those matching the Contoso internal username schema.

Apart from swapping a simple business object type with a simple business object that inherits from it to change the regular expression or other validation properties, inheritance, as proposed as a requirement here, has an additional use. As can be seen in Figure 3.2, the derived business object type *ContosoSMTPServerSettings* contains a field “InhouseServerLocation” which is not present in the parent business object type. This means that a derived business object type can contain additional fields holding information which is only relevant to the sub-type, in this case a textual description of the physical server location of the SMTP server in the Contoso enterprise headquarters.

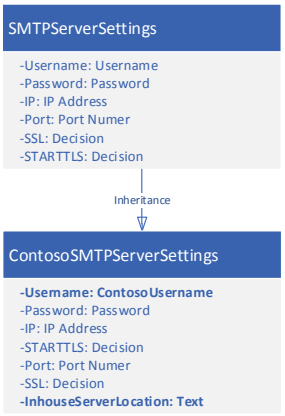

Figure 3.2.: Inheritance Example

Keep in mind that business objects instantiated from the *ContosoSMTPServerSettings* business object type can still be mapped to the input parameters of the any generic SMTP

mailer plug-in which expects a business object created from the base *SMTPServerSettings* business object type. Handling inheritance and mapping this way is a requirement do to the main goal of the BPMS extensions proposed in this thesis: simplicity. An alternative approach to the example of a simple SMTP mailer plug-in and mapping a derived type to it would be to have the mailer plug-in expose all its input parameters individually, instead of as the one large *SMTPServerSettings* business object type. One could still logically group the fields in a complex business object type in the process model, even including additional fields such as the “InhouseServerLocation”, but would have to map all of them individually to the input parameters of the plug-in. Therefore we formulate the following requirement.

Requirement REQ-3: (Inheritance Capabilities for Business Object Types)
A BPMS should support inheritance for complex business object types.

3.1.4. Business Object Collections

A BPMS supporting collections of business objects can handle process models in a more flexible way than other BPMS. To verify this, take, for instance, a business process that helps check if a shipment of parts is complete. It does this by reading part numbers that are expected to be in the shipment from a database and writing them to an integer business object list. Then it allows the worker who is checking the parts in the shipment to input every part number he comes across into a form, saving these to a second integer list. After the worker has input all the part numbers that were actually in the shipment, a small service task plug-in can iterate over the two lists and check if all the expected parts were present. Without support for collections, in this case a list, of business objects, this business process would have had to been implemented in some other, more complicated, way.

However, not only include collections of primitive business objects, such as the simple integer array necessary for the shipment example, but also business objects with almost any degree of complexity imaginable should be supported by the engine extensions proposed in this thesis. An example of such a business object is given in Figure 3.3.

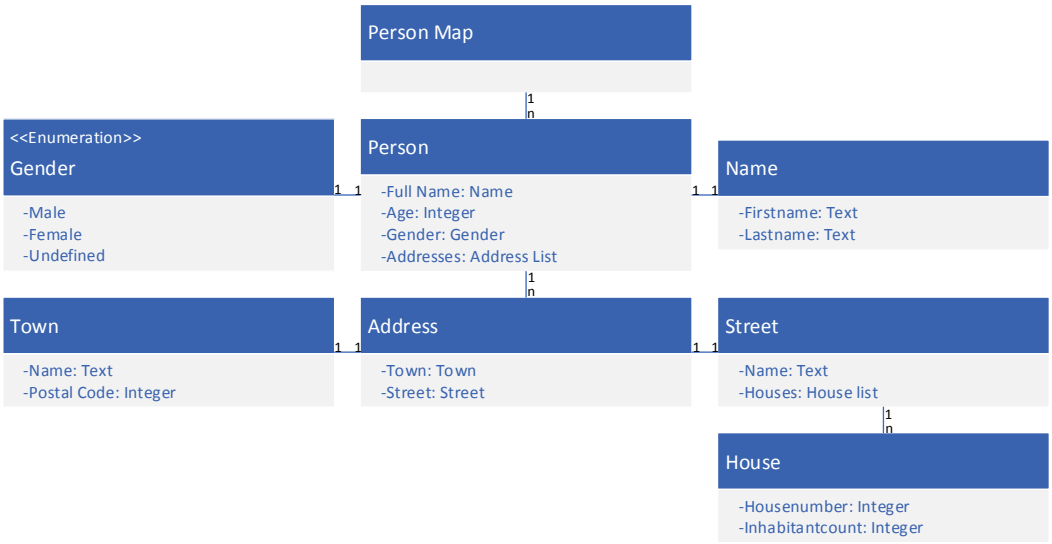

Figure 3.3.: Nested Complex Business Object Collection Example

In the example the top-level business object is a *Map* data structure with a string key, containing business objects of a *Person* business object type. The *Person* business object type itself consists of a large amount of fields backed by other business objects, such as *Name*, *Age*, *Gender*, and even a further complex business object collection called “Addresses”. “Addresses” contains objects of a new type *Address*, again consisting of further fields. The BPMS should allow for virtually endless flexibility, as long as no circular dependencies are created while building the business object types. A circular dependency would be for instance an *Address* containing a field of the *Person* business object type, whereby the *Person* business object type itself would contain the *Address* business object type anywhere in its type sub-tree. Effectively this means that all data structures defined in the BPMS must be completely acyclic, which is necessary for the correct generation of user forms and persisting of the defined business object types in a relational database management system (RDBMS). Also [22] states that data analysis algorithms can not work on cyclic data structures.

The reason why business object collections of any complexity should be allowed is actually quite simple: the complex business objects in the BPMS are supposed to be

seen as atomic by process model designers. A *Person* should not differ from an *Integer* in the way it is used while modeling a business process. Disallowing lists and maps of complex business object types would therefore just add confusion to a system otherwise trimmed for simplicity, while also hindering the ability for modeling business processes similar to the shipping example but using more complex business objects. This expected consistency and the general need for business object collections leads us to the following requirement.

Requirement REQ-4: (Business Object Collections)
A BPMS should support collections of complex business objects.

3.2. Service Task Requirements

3.2.1. Complex Business Objects in Service Task Plug-Ins

The BPMS proposed in this thesis aims at reducing the complexity for process model designers by bundling configuration parameters into complex business objects, thereby hiding most detailed configurations from the user actually modeling the business process, as required by Requirement REQ-1. These complex business objects must be readable and writable in service tasks for this to make any sense though. This means that a programmer developing a plug-in for service tasks must have a way to access the entire structure of a complex business object type in his code. The problem is that these complex business objects are not known to the Java environment that a plug-in developer would be working with.

These business object types would not necessarily even be defined in code, but most likely using a web interface offered by the BPMS. So developers cannot just import a JAR file and have the “class” files of the complex business objects they wish to use available on their class path as the available business object types can grow and change while the BPMS is up and running.

All in all this means that plug-in developers should be completely independent of any interfaces or strict API packages that most BPMS enforce, an example of which can be

seen in [9]. This approach makes sense in the current prototype environment, as any engine specific API is still subject to change at any time, making a “stable” programming API for plug-in developers impossible. Additionally this enables custom complex business object instances to be used by plug-ins without having to import their classes from the BPMS into the developer’s development environment (IDE).

The necessity for having complex business objects in available in plug-in code is directly derived from Requirement REQ-1. In order for a plug-in to allow mapping a complex business object atomically to one of its parameters, it must be able to understand the structure of the business object type internally. Additionally, regarding Requirement REQ-4, plug-ins must be able to read and update collections of business objects in order for the BPMS to support business processes like the shipping example from Section 3.1.4. These reasons lead us to the formulation of the following requirement:

Requirement REQ-5: (Complex Business Objects in Service Task Plug-Ins) A BPMS should support the use of complex business objects and collections in service tasks.

3.2.2. Variable Length Arguments for Plug-Ins

As an addition to the business object collections proposed in Section 3.1.4, a BPMS should support variable length arguments for plug-ins. A variable length argument, or varArgs [18] for short, is syntactic sugar in programming languages such as C# or Java. It allows a method that handles an array of an object type to also handle multiple individual variables of that object type.

This is useful in a BPMS as well, as shown in the following example. Consider a service task plug-in “Sum” that summarizes a business object list of the *Integer* type. Usually a process model would need to have a business object of the *Integer List* type to be able to map to this input parameter. If a process model does not have a “real” *Integer List*, the BPMS should allow mapping multiple individual *Integer* business objects to the input parameter expecting the list. The BPMS could then create a virtual list at run-time, comprised of all mapped business objects. The service task would receive a list, just as it would expect, so there would be no additional coding or configuration work

necessary on the plug-in's side to support varArgs parameters. This would also allow a single business object to be mapped to a parameter expecting a list of business objects, which is unproblematic as long as the service task does not expect any minimum length for the list, i.e., uses only *Iterator*s [11] or for-each [6] constructs to access list items. Clearly, varArgs can only be implemented for input parameters, as there is no way to infer which element of a list returned by a service task plug-in would be serialized into which business object. This behavior mirrors that of popular object-oriented languages and is dictated by common sense.

This allows use to state the following requirement:

Requirement REQ-6: (Variable Length Arguments for Plug-Ins) A BPMS should support the use of variable length arguments (varArgs) for service task plug-ins.

3.3. Mapping and Ad-hoc Process Flow Requirements

3.3.1. Mapping Individual Parts of Complex Business Objects

The inclusion of complex business objects (cf. Section 3.1.2) into a BPMS can increase complexity for the user [22]. For example, consider a service task plug-in that takes one parameter of a complex type, for example *BankAccount*. The type consists of two fields, "Owner" and "Balance", that are of the *Person* and the *Integer* type, respectively. The problem is that for this service task to function, a *BankAccount* business object instance is necessary. The advantage that the BPMS should offer here is that the engine should actually be aware of the composition of the business object *BankAccount*. It should know how a *Person* is structured internally and also how any contained types in *Person*, e.g., *Address* are structured. This would allow for the creation of ad-hoc *BankAccount* business object instances, constructed at run-time, using other business object instances that can be combined to form a *BankAccount* instance.

This means that in our example we could just map a *Person* and an *Integer* to the input parameter of the service task instead of a complete *BankAccount*, as seen in Figure 3.4.

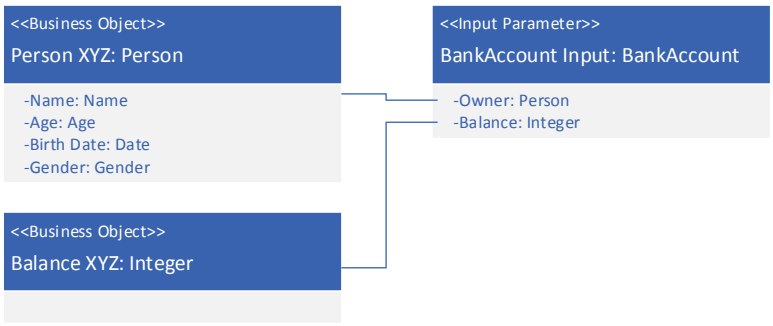

Figure 3.4.: Mapping Partial Business Objects

This flexibility should go both ways, making it possible to map parts of existing complex business objects to simpler parameters. Consider a service task that expects a parameter of the *Integer* type. Regarding our previously used *BankAccount* business object type, it should be possible to use only the “Balance” field for mapping to the input of the service task. The same goes for the output mapping, i.e., if the service task additionally outputs a business object instance of the *Integer* type, one should be able to map that output back into the *BankAccount* “Balance” field. The “Balance” field should then update that field of the complex business object instance with the value returned by the service task.

This mapping flexibility eliminates the need for users to create complex business objects just because a service task requires a parameter to be of a certain complex business object type. Also it eliminates the need for users to manually break complex business objects down into simple business objects in order to map them to a service task that can only handle the simple business object type. It is therefore necessary to formulate Requirement REQ-7 as follows in order to keep the complex business objects required by Requirement REQ-1 usable without increasing the complexity for process model designers:

Requirement REQ-7: (Mapping Individual Parts of Complex Business Objects) A BPMS should support bidirectional mapping of individual parts of complex business objects.

3.3.2. Generation of User Forms Based on Complex Business Objects

The regular expressions defined for types (cf. Section 3.1.2) should be used in many places across the web interface of the proposed BPMS, for instance in dynamically created forms and settings windows. Apart from the regular expressions, the actual structure of the business objects, i.e., their type, should be used for displaying form based user tasks. Figure 3.5 shows a typical form created dynamically using a complex business object's type information.

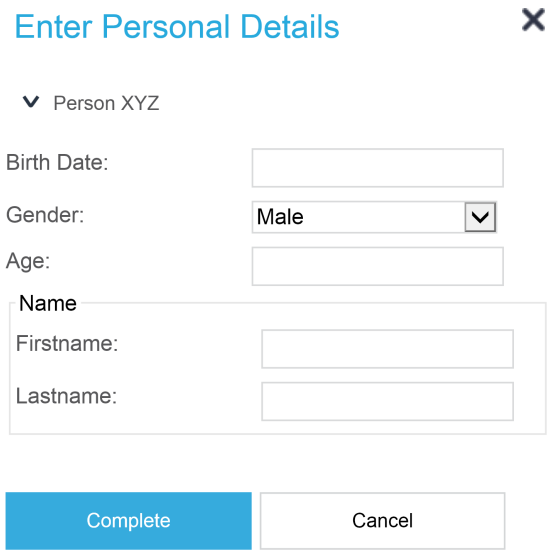

Figure 3.5.: User Form Composed of Complex Business Objects

Clearly, under the assumption that the business objects used in a process model are structured properly, the generated forms created using these rules eliminate most of the workload for process model designers, generally associated with designing basic forms for business processes [31]. Generated user forms in BPMS is a highly active field of research, both in commercial and academic fields[27, 26, 30, 36]. These generated forms should also be reusable as error correction forms, which are required to support Requirement REQ-10, described in Section 3.3.4. This is formulated in the following requirement.

Requirement REQ-8: (Generation of User Forms Based on Complex Business Objects) A BPMS should support manipulating complex business objects in dynamically generated forms.

3.3.3. Manual Gateway Execution

The BPMS should offer users a new way of executing their business processes by viewing and interacting with the process model for a process instance in a graphical web interface. This would offer a unique and simple way to reduce the amount of user tasks and therefore forms with simple one-line questions such as “Skip Task B?” or “Continue to Task C or D?”. This would allow the user to double-click an XOR gateway and just select the path he would like the process instance to continue on. This “empty” or “manual” gateway need not have any conditions and could just be inserted into the process model. Instead of displaying the standard dialog, which just allows selection of the next task name (cf. Figure 6.2), process model designers should be allowed to set a custom question with answers corresponding to the outgoing branches of the gateway. This very specific requirement stems from the way users interact with a process instance they are working on, i.e., directly on the process model graph. This is a unique feature following the notion of eliminating work-lists and having users execute process instances by interacting with the process model [19], justifying Requirement REQ-9 being the following.

Requirement REQ-9: (Manual Gateway Execution) A BPMS should support manual gateway execution.

3.3.4. Correctness by Run-time Error Resolution

The implementation of the manual gateway execution feature proposed in Requirement REQ-9 is actually a lot more complicated than one might assume, as it involves pausing and resuming individual execution branches of the process instance. Luckily, the features developed to support Requirement REQ-9 are reusable for another proposed feature. The feature in question is the correctness by run-time error resolution system which we introduce.

Correctness by run-time error resolution should allow for process models to be deployed and tested in a much earlier state than in other BPMS, which either force the correctness by construction principle [21], or just fail the process instance execution when there are errors. Correctness by run-time error resolution focuses on data flow errors and should therefore be combined with a simplified correctness by construction system for control flow correctness verification. It should allow for service task input parameters that are not correctly mapped to be filled with data at run-time. Also service tasks that are correctly mapped but fail due to errors in the mapped data (e.g., wrong username or password used for an SMTP service task, resulting in an exception) should be retryable with different data during process instance execution.

In essence, the BPMS should allow running a process model in any state: the web interface should ensure correctness by construction for the process model structure, and the correctness by run-time error resolution system should ensure that service tasks can be executed with little to no prior configuration or business object mapping. This should be facilitated by pausing the process instance when a service task cannot execute and showing the user a generated form, as seen in Figure 6.6. The form should collect all missing data needed to execute the service task and retry it once the form is completed by the user.

This adds the ability to test individual service tasks without having to set them up “properly”, which is helpful when developing a process model iteratively. Also after the process model is in a “final” stage and deployed in productive use errors can still occur during process instance execution. The run-time error resolution offers quick instance-specific fixes for errors such as changed passwords for accounts used by service tasks etc. For some BPMS, changing the password of for instance an SMTP account would result in process instances not being able to complete normally as updating the service task in the process model with the new password would not affect the running instances. An engine supporting run-time error resolution, however, would allow resolving the error for the currently running process instances. Therefore, this this formulated as Requirement REQ-9:

Requirement REQ-10: (Correctness by Run-time Error Resolution) A BPMS should support the operations necessary for run-time error resolution.

3.3.5. Rapid Process Model Prototyping

Supporting the idea of having a process model which can be executed and thereby tested at a very early stage in process design, as was mentioned in the introduction and in Requirement REQ-10, we propose a simple way of setting up empty tasks. These can be used as placeholders for service tasks or user tasks and should be executable by simply double-clicking them. They should also be useable similar to the BPMN manual task, i.e., a task that is executed “offline” which must simply be marked as completed for a process instance to progress.

By default all new tasks in a process model should be empty tasks, which then get replaced with service tasks or user tasks on the fly during modeling. This means that a first draft of a process model could be created and executed by adding empty tasks and manual gateways (cf. Section 3.3.3) to the process model and executing the process model immediately. This way the “flow” of the business process could be tested early on, and, additionally, allow for seamless extension of the early draft with functioning service tasks, user forms and even gateway logic.

In the interest of simplicity, the build-time web interface should allow making an empty task into a dynamic user form by simply connecting business objects to the input/output parameters of the placeholder. Dragging and dropping a service task plug-in on to a placeholder should automatically convert the placeholder into a service task, reducing the amount of clicks necessary for such actions. Lastly the web interface and engine should also support interchanging of AND and OR gateways on the fly by simply double-clicking them. More detailed descriptions of these features can be found in [19].

These simplifications to structural modeling speed up the development of simple process models and therefore the engine should support them, as formulated in the following requirement:

Requirement REQ-11: (Rapid Process Model Prototyping) A BPMS should support the operations necessary for rapid process model prototyping.

3.4. Summary

The requirements shown in this section and summarized in Table 3.3 are are supported by the Clavii engine, the main contribution of this thesis, the concept and implementation of which is detailed in the following sections. These requirements are needed to support advanced data flow concepts in BPMS.

REQ #	**Requirement Name**	**Thesis Section**
REQ-1	Complex Business Objects	4.1
REQ-2	Strongly Typed Business Objects	5.3
REQ-3	Inheritance Capabilities for Business Object Types	5.3
REQ-4	Business Object Collections	4.1
REQ-5	Complex Business Objects in Service Task Plug-Ins	5.3
REQ-6	Variable Length Arguments for Plug-Ins	5.4
REQ-7	Mapping Individual Parts of Complex Business Objects	6.4
REQ-8	Generation of User Forms	6.5
REQ-9	Manual Gateway Execution	6.1
REQ-10	Correctness by Run-time Error Resolution	6.6
REQ-11	Rapid Process Model Prototyping	6.6

Table 3.3.: Overview of Requirements

4

Dynamically Structured Complex Business Objects

A central requirement in the context of advanced data flow concepts is Requirement REQ-1, the ability to handle complex business objects that are defined by the user. As stated in Section 3.1.2, this aims at reducing the complexity of data flow in business process models and increases the readability of business object mappings. Furthermore, these complex business objects should be accessible in the user interface for mapping to service task parameters and displaying in user forms (cf. Requirements REQ-5 and REQ-8).

In addition, the serialization of new business object types and their instances requires a proper concept (cf. Section 4.1). Particularly, serialization is necessary to transfer business object types and business objects to the remote user interfaces. Furthermore,

a serialization concept is also required to store business object types persistently in, for example, a relational database.

A more technical view on the relation of business object types to business objects to business object instances (cf. Section), including their representation towards plug-ins is given in the following:

1. *Business object type representation* for BPMS engine and user interface:
   A *business object type* is the dynamically structured type defining what attributes a business object consists of. A type can be defined using the user interface. Basically, a business object type consists of a name and a list of attribute fields. A field is a named child business object type of the respective business object type, for example, a *Person* business object type may contain a field with the name *Age* and the business object type *Integer*. Particularly, the representation has to be persistable in a database or XML document to ensure persistence and portability for business object types created by users.

2. *Process model-specific object representation*:
   When process model designers use a type in a process model, they effectively create a process model-specific instance of that business object type, a *business object*. A process model-specific business object can be named, given default values for the business objects contained within it (if it is complex), and mapped to input/output parameters of plug-in operations. Creating an instance of a business object type is thus necessary for actually using it in a process model. For process model designers this equates to selecting a business object type they wish to use, and giving it a name. The resulting business object then shows up in the business object list for the current process model, making it available for mapping uses. This representation has to be persistable as well to allow saving mappings and default values persistently on a per-process model basis.

3. *Instance-specific object representation*:
   Once a business object has been set up as part of the data flow of a process model it is available to a process instance in the form of a *business object instance*. This means that it can hold values specific to a process instance, and not just

default values used by all process instances, as it is the case with the business object. This representation has to be persistable in the Activiti BPM engine itself, as all instances of business objects and their values are managed by the Activiti BPM engine. As the Activiti BPM engine is not able to deal with the business object types and their respective business objects, the resulting business object instances have to be packaged into *Maps* and *Lists*, similar to what is necessary for representing the business object types for plug-ins (Section 3.2.1).

4. *Plug-in business object type representation*:
   As described in Section 3.2.1, plug-in implementations need an extra representation of complex business object types, to be independent of a BPMS API. This can be achieved by packaging business object types into data structures like *Maps* and *Lists*. This enables plug-in developers to use the complex business object instances in their code without Java needing information on the corresponding business object types and structure. Particularly, a plug-in developer may access elements of a business object utilizing a dot-based notation (cf. Section 3.2.1).

The next Section 4.1 describes our concept to serialize definable complex business objects. Section 4.2 details the instantiation of these business objects. Section 4.4 describes the serialization of default values that business objects may have. Furthermore, Section 4.5 introduces the actual implementation of our serialization concept, i.e., persisting of the business objects and business object types to a database. Finally, Section 4.6 describes the definition of business object types using an XML descriptor language.

4.1. Persisting User-Definable Business Objects

The concept that the Clavii engine uses to represent business object types and business objects in Java is determined by the requirements that both the business object type and a business object of said type have to be persistable, for example, in a database, and that business object type information must still be applicable after deserializing it again. Java theoretically supports creating new classes and instances of those classes

while executing Java code, using byte-code manipulation libraries such as CGLIB [4] or Javassist [13]. As Java is a statically typed language it is, generally, not desired to introduce new classes at run-time, thus it is not supported without the use of these additional libraries. Further, a Java class not present at compilation would necessitate an extensive use of reflection [17] to analyze it after deserialization from a database at run-time. Additionally, the flexibility offered by using byte-code manipulation to create "real" Java classes at run-time is not required in a BPMS.

4.1.1. Business Object Types

The Clavii engine uses a different concept to create business object types and business object instances at run-time: the use of a few Java classes mimicking the behavior of the core Java class creation and object instantiation features. For business object types, this is illustrated in Figure 4.1.

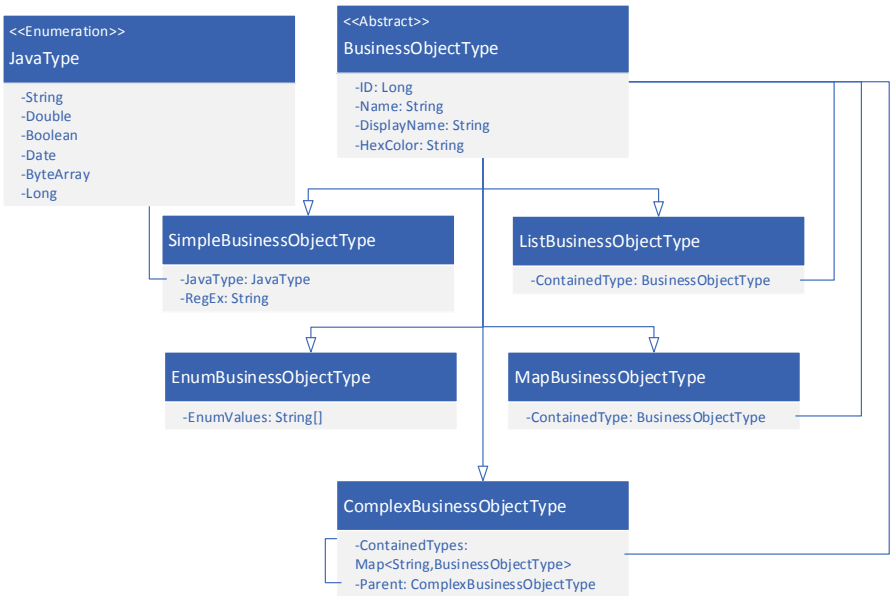

Figure 4.1.: *BusinessObjectType* Serialization Concept

The following summarizes the roles of all participating classes in the serialization concept for business object types:

BusinessObjectType class: The *BusinessObjectType* class is the abstract base class for all manifestations of business object types and forces sub-classes to be serializable. It comprises the *Name* and database *ID* which represent the business object type. The class also acts as the root element for JAXB serialization, needed for importing new business object types from XML files (cf. Section 4.6). To support advanced user interface concepts, this class also holds the fields *HexColor* and *DisplayName* which are detailed in [19].

SimpleBusinessObjectType class: The *SimpleBusinessObjectType* class represents a primitive business object type consisting of the fields *JavaType* and *RegEx*. The *JavaType* field holds an enumeration value determining the underlying Java data type. The Java data type is used for serializing values in the simple business objects created from this simple business object type. Also, the Java data type is necessary for type checking and for determining the type of input elements that user forms will display to users at run-time. The Clavii engine uses a subset of common Java data types as base types: *String*, *Long*, *Double*, *Boolean*, *Date* and *ByteArray*. The latter offers support for serializing any form of document or file. Next, the *RegEx* field supports the user form component of the Clavii BPM Cloud further by allowing a per-type definition of regular expression rules which can be used for field validation. Simple business object types are found on the lowest tier of any complex business object type construct as they can not contain further business object types.

EnumBusinessObjectType class: Similar to the *SimpleBusinessObjectType* class, this class represents a business object type consisting of a limited amount of strings. The enum business object that can be created from the enum business object may have exactly one of these strings as its value. Field *EnumValues* holds the aforementioned list of strings.

ListBusinessObjectType class: To represent a collection of business objects, this class contains a field *ContainedType*, holding a reference to the business object type of which business object instances will be allowed to exist in the collection at run-time.

MapBusinessObjectType class: Analogous to the *ListBusinessObjectType* the *MapBusinessObjectType* class also has exactly one field *ContainedType* which holds a reference a business object type.

ComplexBusinessObjectType class: The *ComplexBusinessObjectType* class represents what is referred to as a “class” in an object-oriented language such as C++, C# or Java, i.e., it holds named fields consisting of references to other business object types, thereby defining the structure of a complex business object type. A simple *Map<String, BusinessObjectType>* structure *ContainedTypes*, the only field in the *ComplexBusinessObjectType* class holds these references. The *Person* business object type (cf. Section 3) is an example of such a complex business object type. The *ComplexBusinessObjectType* class can hold references to any class inheriting from the *BusinessObjectType* class. By allowing the referencing of other instances of the *ComplexBusinessObjectType* class, the concept allows process model designers to create almost any complex business object type. The one limitation (cf. Section 3.1.4), is that there must never be cyclic dependencies, i.e., a complex business object type may never contain itself at any depth of nesting. If this were allowed, user form generation would be impossible based on the infinite loop structure of the complex business object’s structural graph. The *ComplexBusinessObjectType* class also holds a reference to a parent complex business object type, if there is one, to allow for inheritance. Inheritance is only possible for complex business object types as other business object types are not extensible with new fields by definition, as they have none.

As these six classes are serializable, using them to describe a business object type allows for serialization of said business object type, described in Section 4.6. This can be used to send types to client machines for displaying in the user interface of a BPMS. Additionally, having serializable classes describing the structure of a business object type allows it to be persisted in a database (c.f. Section 4.5).

Deserializing a persisted business object type for usage is as easy as deserializing instances of the composing classes (e.g., *BusinessObjectType* etc.) from the database and then traversing the structure via the references and collections offered by them. The

structure of a variant of the complex business object type *Person,* shown in Figure 4.2, can be analyzed using the method *getContainedTypes()*.

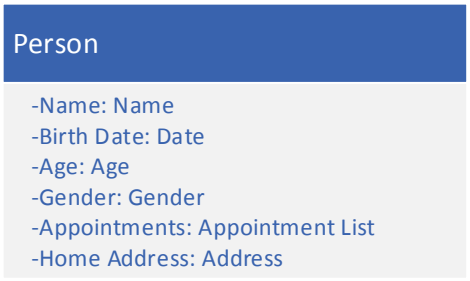

Figure 4.2.: *Person* Business Object Type

The *getContainedTypes()* method returns field names and business object types for all fields contained in the *Person* business object type. For example, for the *Person* business object type, it would return*:*

- *"Age":(SimpleBusinessObjectType Age)*
- *"Name":(ComplexBusinessObjectType Name)*
- *"Birth Date":(SimpleBusinessObjectType Date)*
- *"Gender":(EnumBusinessObjectType Gender)*
- *"Appointments":(ListBusinessObjectType Appointment List)*
- *"Home Address":(ComplexBusinessObjectType Address)*

To access nested business object types, such as the simple business object types *Firstname* and *Lastname*, belonging to the complex business object type *Name*, recursive calls are necessary. Traversing the type structure this way is preferable from a performance standpoint to using reflection, mentioned as an alternate approach in the introduction to this section, as Java reflection is generally slow do to missing virtual machine optimizations [17]. Also the API and structure of the *ComplexBusinessObjectType* is much cleaner and simpler than with a completely generic class that has to be analyzed using reflection.

4.1.2. Business Objects

Business objects, as explained in the introduction to Section 4, are named process model-specific business object types. They are created when a process model designers wishes to use a certain business object type in a process model. To be more precise, they can be mapped to service and user tasks in a process model and may hold default values in fields. The class hierarchy for the *BusinessObject* class is similar to that of the *BusinessObjectType* class (cf. Section 4.1.1), as the requirements are similar, for instance business objects must also be serializable. Also the structural graph of a business object has to be traversable in order for the user interface to parse it and display the dynamic user interface elements for mapping and entering default values correctly.

The following summarizes the roles of all participating classes in the serialization concept for business objects.

BusinessObject class: Abstract base class for all other *BusinessObject* classes. Holds a reference to the business object type that defines the structure of the business object, it also holds a *Name* field containing the name the business object has in the process model. The name is only used for top-level (i.e., root) business objects that are displayed directly in the user interface. Nested business objects must not necessarily be named, i.e., a *Person* business object *Person A* that is used directly in the user interface has to be named; the business objects contained therein, on the other hand, holding the values for the *Age* or *Gender* attributes of *Person A*, do not.

AttachableBusinessObject class: Direct sub-class of the *BusinessObject* class. This abstract class is the base class for all business objects that can be mapped to a parameter directly, i.e. are viewed as atomic in the context of mapping. This includes all further *BusinessObject* sub-classes, except for the the *ComplexBusinessObject* class, i.e., *SimpleBusinessObject, EnumBusinessObject, ListBusinessObject,* and *MapBusinessObject*. The corresponding four types of business objects cannot be broken down further for mapping. They are always mapped in entirety to one single parameter of a service task. A complex business object, on the other hand, is actually never mapped to a parameter directly, as only the contained instances are mapped in order to support partial complex business object mapping (cf. Section 3.3.1). Class *AttachableBusines-*

sObject holds two *Map<String, PathDescription>* collections, one for mapping input and one for mapping output. The keys of this map's entries each hold a reference to one individual service or user task situated in the process model that the business object belongs to. The value associated with one of these keys, a so-called *PathDescription*, holds a reference to one of the input parameters of the service task plug-in that the *BusinessObject* is mapped to. If the *AttachableBusinessObject* is mapped to a user task instead of a service task, the *PathDescription* is *null*.

SimpleBusinessObject class: Represents a *simple business object*, holding a value of a the Java type specified the *JavaType* attribute of the simple business object type that the simple business object was created from. This "default" value dictates what the process instance specific business object instance will be initialized with when an actual process instance is started. The value is editable in the user interface at build-time.

EnumBusinessObject class: Represents an *enum business object*, holding one of the values defined in the respective enum business object type's *EnumValues* field. The enum business object can only have *String* values.

ListBusinessObject class: Represents a *list business object*, has a *List<BusinessObject>* field holding references to all business objects that should be initially contained in a business object instance instantiated from this list business object.

MapBusinessObject class: Represents a *map business object*, contains a *Map<String, BusinessObject>* field holding references to all business objects and their map keys that should be initially contained in a business object instance instantiated from this map business object. A use case for this might be a map business object that holds default mapping values for filling out a PDF form. The map business object in question could use its map keys as field names and *SimpleBusinessObjects,* using *String* as their *JavaType*, as the values to be written to the PDF form fields.

ComplexBusinessObject class: Represents a *complex business object*, contains a single *Map<String, BusinessObject>* field, but in contrast to the *MapBusinessObject* class, the *Map* keys are field names of the contained business objects. The field names are the exact field names defined in the corresponding complex business object type and are written into the *Map* during creation of the complex business object.

4.2. Creating Business Objects from Business Object Types

Creating a business object is performed by calling the abstract *getInstance(String name, Long parentModelId)* method located in the *BusinessObjectType* class. The method takes parameters name and parentModelId, i.e., a reference to one specific process model that the instance should belong to, and returns a respective business object. As the method is abstract, each sub-class of class *BusinessObjectType* must offer a respective implementation. An example implementation, found in the *ComplexBusinessObjectType* class, is given in Listing 4.1.

```
1  public class ComplexBusinessObjectType extends BusinessObjectType {
2      //getters omitted
3      private ComplexBusinessObjectType superType;
4      private Map<String, BusinessObjectType> containedTypes = new HashMap<>();
5
6      @Override
7      public ComplexBusinessObject getInstance(String name, Long parentModelId) {
8          //call constructor and pass "this"
9          ComplexBusinessObject toReturn = new ComplexBusinessObject(name, this, parentModelId);
10         //iterate over contained types, call instantiation method recursively
11         //add any instantiated objects to containedInstances Map
12         Map<String, BusinessObject> containedInstances = new HashMap<>();
13         for (Map.Entry<String, BusinessObjectType> entry : containedTypes.entrySet()) {
14             containedInstances.put(entry.getKey(), entry.getValue().getInstance(entry.getKey(), null));
15         }
16         toReturn.setContainedInstances(containedInstances);
17         return toReturn;
18     }
19 }
20
21 public class ComplexBusinessObject extends BusinessObject {
22     //getter omitted
23         private Map<String, BusinessObject> containedInstances = new HashMap<>();
24
25     public ComplexBusinessObject(String name, SimpleBusinessObjectType instanceOf, Long parentModelId) {
26         super(name, instanceOf, parentModelId);
27     }
28 }
```

Listing 4.1: Instantiation Code for a Complex Business Object

Line 14 in Listing 4.1 shows how the “ContainedObjects” *Map* held by the *ComplexBusinessObject* class is filled with further new instances recursively during instantiation (cf. line 7). A business object created using the *getInstance()* method is immediately ready to receive default values for its fields, which is discussed in Section 4.4. Since the entire class structure below *BusinessObject* is serializable, new business objects can

be persisted in the database (cf. Section 4.5), or sent to the user interface via GWT-RPC (Remote Procedure Call) [37].

4.3. Instantiation of Business Objects

As introduced in Section 4, business objects created based on business object types are process model-specific and may only hold default values for individual process instance-specific business object instances. Business object instances are instantiated for use in the context of a specific process instance in the Activiti BPM engine. Particularly, the latter has no knowledge of the concept of business object types and business objects, and, therefore, business object instances must be composed using standard Java constructs, specifically the *Map* and *List* classes. This "flattening" of *BusinessObjects* into a dot-based *Map* compatible form, thereby instantiating them to business object instances, is done by the *BusinessObjectFlattenVisitor* class (cf. Appendix B.1). The visitor pattern applied by the *BusinessObjectFlattenVisitor* class is described in Section 4.3.1.

Basically the Activiti BPM engine has one *Map* per process instance that contains the values of all all data objects for the process instance. The *Map* is comparable to a standard Java *Map* implementation in that it uses *Strings* as keys for all contained entries. The challenge is to keep the structure of the business objects intact inside the *Map*. This is achieved by using the aforementioned dot-based notation, thereby flattening the structure of the business objects into the *Map*. An example of what the flattened version of a complex business object *PersonYXZ* of a complex business object type *Person* looks like is shown in Figure 4.3.

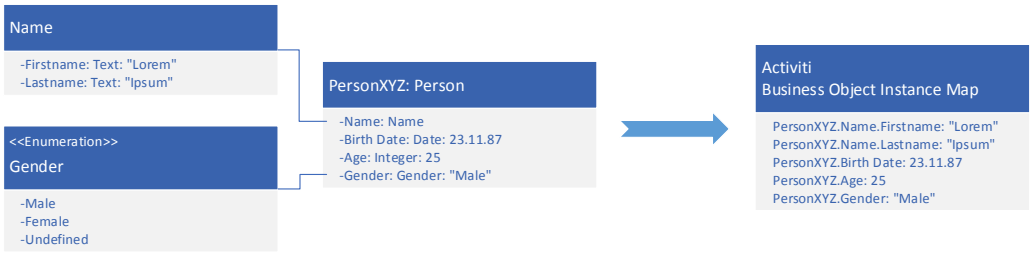

Figure 4.3.: Flattening of a Complex Business Object

More complex data structures, for instance a complex business object containing a *List* further complex business objects, can be flattened by placing, e.g., a *List<Map<String, Object>>* at one of the keys in the Activiti BPM engine's data object map. This enables the business objects the Clavii engine supports to be of greater complexity by allowing nesting of complex business objects into each other. The main workload that this relatively complex flattening system has to manage is done using recursion and the visitor pattern to correctly handle business object nesting. As the instantiation of the business objects, and therefore also the flattening, only takes place once during the instantiation of the process there is no overhead associated with complex business object instances during the actual process instance execution.To minimize the overhead during creation of process instances caused by instantiating and flattening the business objects associated with the process model can also be minimized through the use of caching. This is possible as the flattening process is deterministic as long as the business objects and their respective business object types remain unchanged.

Arranging business object instances at run-time according to the structure that the respective business object types dictate does allow for some of the special features of the Clavii engine to function, namely the detection of missing parameters and their ad-hoc creation. To be more precise, the arrangement of the business object instances in this structured, dot-based form, mirroring the structure of the business object types they were created from, allows the Clavii engine to inspect and insert values at run-time

in exactly the place where a user form or service task would expect them (cf. Section 6.2).

To make this clearer, consider a process instance in which the *PersonXYZ* business object (cf. Figure 4.3) has a missing attribute *Birth Date*, i.e. the value for the key "PersonXYZ.Birth Date" is null or the key entirely non-existent. As the *Person* business object type dictates that a *Person* business object has a value for *Birth Date*, a missing value for this attribute can be detected as an error before the invocation of a service task plug-in, which could potentially throw an exception as a consequence of a missing value (cf. Section 6.2). In this case the correctness by error resolution system proposed in Requirement REQ-10 would show the user an error resolution form allowing him to input the missing value. Through the aforementioned special, flattened, arrangement of the business object instances in the Activiti BPM engine's data object *Map*, the key at which the "new" value has to be inserted is clear, namely "PersonXYZ.Birth Date". This not only allows the service task that would have failed to execute properly, but also the remaining tasks in the process instance to use the newly entered *Birth Date* associated with *PersonXYZ*.

4.3.1. Applying of the Visitor Pattern

The *BusinessObjectFlattenVisitor* (cf. Appendix B.1) uses the visitor pattern [23]. This pattern provides an object-oriented way of recursively calling methods in the right subclasses necessary for instantiating different types of business objects without using the *instanceof* operator, often viewed as code-smell in object-oriented languages [38]. The language limitation that makes the visitor pattern necessary is shown in Listing 4.2.

```
1  class A{}
2  class B extends A{}
3  class C extends A{}
4  class Test{
5          public static void main(String args){
6                  A b = new B();
7                  doWork(b); //throws error as there is no overload doWork(A)
8          }
9          static void doWork(B b){
10         }
11         static void doWork(C c){
12         }
13 }
```

Listing 4.2: Java Language Limitation Example

As Listing 4.2 shows, Java does not attempt to infer the concrete sub-class of “A” and call the correct overload. As the *BusinessObjectFlattenVisitor* class provides the same “visit” method that does the actual flattening to all *BusinessObject* sub-classes, in a traditional setup the class would run into exactly this problem. The visitor pattern circumvents this by using double-dispatch via a “visitor”, a specially prepared interface (cf. Listing 4.3).

```
1  public interface BusinessObjectVisitor {
2      void visit(SimpleBusinessObject simpleBusinessObject);
3      void visit(EnumBusinessObject enumBusinessObject);
4      void visit(ComplexBusinessObject complexBusinessObject);
5      void visit(MapBusinessObject mapBusinessObject);
6      void visit(ListBusinessObject listBusinessObject);
7  }
```

Listing 4.3: *BusinessObjectVisitor* Interface

To complete the visitor pattern, all sub-classes that use implementations of this visitor interface must have a method *accept*, analogous to the following Listing 4.4:

```
1  @Override
2  public void accept(BusinessObjectVisitor businessObjectVisitor) {
3      businessObjectVisitor.visit(this);
4  }
```

Listing 4.4: *Accept* Implementation

The method *accept* is forced on these sub-classes by *BusinessObject* super-classes, enabling calls such as the one in Listing 4.5:

```
1 //instantiation of businessObjectXYZ and BusinessObjectFlattenVisitor omitted
2 //businessObjectXYZ is an instance of SimpleBusinessObject
3 businessObjectXYZ.accept(BusinessObjectFlattenVisitor);
```

Listing 4.5: Usage of the *Accept* Method

By utilizing the visitor pattern and its double dispatching capabilities, Java is able to take an instance of a sub-class of the *BusinessObject* class*,* e.g., *SimpleBusinessObject,* and without knowing which concrete sub-class the object in question is actually of, can indirectly call the correct *visit(...)* overload(cf. Listing 6.4).

4.4. Serializing Default Values for Simple Business Objects

Once a simple business object is created from a simple business object type (cf. Section 4.2), it can be assigned default values in the user interface. The data type of such a default value is dictated by the respective simple business object type. The data type can be one of the following supported Java data types: *String, Long, Double, Boolean, Date, ByteArray*.

Therefore, the *SimpleBusinessObject* class must have a field that can hold a value of any of these types. Please note, that an instance of the *SimpleBusinessObject* class must be serializable not only for insertion into a database, but also for delivery to the user interface. As the web interface of Clavii was written using GWT, which uses JavaScript internally, it cannot handle Java serialization/deserialization, meaning that it is not possible to simply serialize all possible values using common Java techniques, such as the *Serializable* interface and the *ObjectOutputStream* class.

The to circumvent this limitation we use the *String* data type as the basis for transporting and serializing all possible default values for simple business objects. Most Java types have a *toString()* method, which returns *String* values for respective data types. Furthermore, most standard Java data types provide a static *parse(String string)* method, that delivers an instance of said type based on a *String* value. Methods *toString* and *parse* are supported in GWT, making *String*-based transport to the user interface viable. The

following listing gives a short example of how a typical conversion works, on both server- and client-side.

```
1 Boolean bool = true;
2 //convert boolean to string
3 String boolAsString = bool.toString(); //boolAsString == "true"
4 //convert string back to boolean
5 Boolean stringAsBool = Boolean.ParseBoolean(boolAsString); //stringAsBool == true;
```

Listing 4.6: Conversion of a *Boolean* to and from a *String* value

This principle is used for all supported classes except *ByteArray* and *Date*. The methods for these data types are explained in the following sections.

4.4.1. Handling of Byte Arrays

Byte arrays are used in Java to hold the content bytes of various files, such as documents. As the base data type for all data types in the Clavii engine is *String*, byte arrays have to be convertible to *String* values as well. Byte arrays are converted to *String* values using a Base64 encoder on the server-side. Base64 is an encoding scheme that allows encoding of any binary information to ASCII [24]. As a byte array is effectively binary information, a Base64 encoder can be used to encode it to a string. The encoded representation can then be persisted in a database using the same VARCHAR field necessary for serializing the other possible value types (*Long, Date, ...)*. GWT does not support Base64 encoding/decoding on the client side, which is actually a non-issue, as downloading and uploading files (i.e., byte arrays) does not work using GWT-RPC anyway. Files are uploaded using servlets, as in most other web frameworks.

To be more precise, a *String* value representing a byte array does not have to be delivered to and decoded by the user interface, as a byte array that can be decoded from the *String* is only used to hold the contents of files. Files can be, for instance, documentation belonging to a business process, which users can upload to the respective process model. This means that they are effectively black boxes and have no other function than being downloadable. As previously stated, downloading files in most web-frameworks is done by offering the file in a file servlet and placing a link to the file in the actual user interface, not actually displaying the contents of the file in the user interface.

To facilitate this, the encoded *String* representing the byte array, i.e. the file, is replaced by a random *Long* representing a virtual address. This is done before each delivery of the simple business object that contains a byte array to the user interface. The *String* containing the *ByteArray* is served up by a servlet on the Clavii engine server using exactly this “address” as the key. The user interface recognizes the address and provides a link to the servlet serving the file for the user to click on.

The advantage, apart from there not being any other way to realize file downloads in GWT, is that such an encoded string containing the byte array, which can be very large, does not have to be transferred to the user interface along with the simple business object. Instead, only an address, which is typically smaller, is delivered, and only in case the user wants to open the respective file is the file transferred via servlet request to the client. This concept, the replacing of the byte array with an address, is visualized in Figure 4.4.

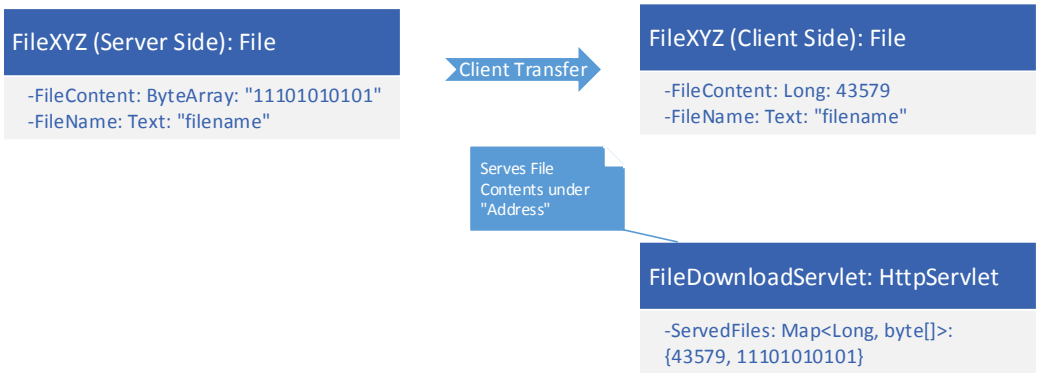

Figure 4.4.: Replacement of *Byte* Arrays with Addresses for Client Transfer

4.4.2. Handling of Date Values

The Java *Date* class provides methods *toString* and *parseDate(String)*, which would suffice in most environments for converting a *Date* to and from a *String* value. The GWT user interface actually processes all business logic in the browser. Particularly, GWT uses the locale of the client for methods *Date.toString* and *Date.parseDate(String)*. For

example, the server may encode the date 10th of December 2013 to the *String* value “10.12.2013”. Subsequently, the client might parse 10/12/2013, the 12th of October 2013. Additionally, these methods are mar deprecated as of Java 1.1 for reasons including the one just described. The replacement methods for converting *Date* to and from *String* are methods *DateFormat.parse(String)* and *DateFormat.format(Date)*. However, the latter are not supported by GWT. The GWT framework contains a similar replacement class [7] called *DateTimeFormat*, but strings produced by class *DateFormat* on the server-side are not parseable by the *DateTimeFormat* class on the client-side.

To circumvent this, a different approach is chosen for handling the conversion of *Date* values to *String* for serialization: the conversion of the *Date* values to the common *milliseconds since epoch* representation . The *milliseconds since epoch* representation is a 64 bit count of milliseconds that have passed since Thursday, January 1, 1970 12:00:00 AM GMT (i.e., the UNIX epoch) until the date represented by the *Date* object. The *Date* class has a constructor that takes a *Long* to create an instance based on the *millisecond since epoch* representation. The *Date* class also supplies a method *getTime* that delivers the *milliseconds since epoch* representation as a *Long*.

Therefore, the issue is reduced to converting data type *Long* to *String* and back, which is done in the same way as for the other data types, using methods *toString* and *parseLong(String)*.

4.5. Persisting of Defined Business Object Types and Business Objects

This section describes how business object types and business objects can be serialized and persisted. For business object types, there is almost no additional work necessary. Particularly, the *BusinessObjectType* class as well as all its sub-classes (cf. Section 4.1.1) that are necessary to describe the structure of a business object type may only hold references to instances of the class *BusinessObjectType*. In a database this means that, applying a table-per-class strategy, each sub-class of *BusinessObjectType* is assigned to its individual database table. When doing this the ID column has to be forced to

contain unique values across all tables for the sub-classes of the *BusinessObjectType* class. This may be automated by Hibernate, using a sequencing table. The advantage of this approach is simple table structure and easy referencing between the sub-class tables. As an example the Data Definition Language (DDL) for creating a database table for the *ListBusinessObjectType* class is listed in Listing 4.7.

```
1  CREATE TABLE LISTBUSINESSOBJECTTYPE
2  (
3        ID BIGINT PRIMARY KEY NOT NULL,
4       #Required for optimistic locking
5        VERSION BIGINT,
6       #UI-Metainfo
7        DISPLAYNAME VARCHAR(255),
8        HEXCOLOR VARCHAR(255),
9        ICONSTRING CLOB,
10      #Unique Name
11       NAME VARCHAR(255) NOT NULL UNIQUE,
12      #Owner can be an Organization/Orgunit/Agent
13       OWNER BIGINT,
14      #Can refer to the ID column of any of the following tables:
15      #LISTBUSINESSOBJECTTYPE,MAPBUSINESSOBJECTTYPE,
16      #COMPLEXBUSINESSOBJECTTYPE,ENUMBUSINESSOBJECTTYPE,SIMPLEBUSINESSOBJECTTYPE
17       CONTAINEDTYPE_ID BIGINT NOT NULL
18 );
```

Listing 4.7: DDL for *ListBusinessObjectType* Table

Database tables for the other sub-classes of *BusinessObjectType*, except for *ComplexBusinessObjectType*, are analogous. Class *ComplexBusinessObjectType* references class *BusinessObjectType* in a *Map<String, BusinessObjectType>* structure, making an extra table called COMPLEXBUSINESSOBJECTTYPE_BUSINESSOBJECTTYPE containing the *String* keys paired with the references necessary.

The *BusinessObject* sub-classes (cf. Section 4.1.2), are treated in almost the same way. They all (except for classes *EnumBusinessObject* and *SimpleBusinessObject*) hold references to the *BusinessObject* class, which can be mapped to tables in exactly the same way as the *BusinessObjectType* sub-classes. However, the *SimpleBusinessObject* class has one *String* field, which can hold any form of data supported by the Clavii engine using the approach explained in Section 4.4. The DDL code necessary for creating the *SimpleBusinessObject* table is shown in in Listing 4.8.

```
1  CREATE TABLE SIMPLEBUSINESSOBJECT
2  (
3       ID BIGINT PRIMARY KEY NOT NULL,
4      #Required for optimistic locking
5       VERSION BIGINT,
6      #Unique to each process model
7       NAME VARCHAR(255),
8      #ID of the process model the instance is attached to
9       PARENTMODELID BIGINT,
10     #ID of the BUSINESSOBJECTTYPE that was used to instantiate this object
11      BUSINESSOBJECTTYPE_ID BIGINT NOT NULL,
12     #Serialized mapping information
13      CONNECTEDINPUTTASKS BLOB NOT NULL,
14      CONNECTEDOUTPUTTASKS BLOB NOT NULL,
15     #Column for a string with 2 gigabyte maximum
16      INTERNALVALUE CLOB
17 );
```

Listing 4.8: DDL for *SimpleBusinessObject* Table

The introduced relational database model makes debugging values and understanding and extending the business object serialization framework easier than the alternative of serializing the entire objects to Binary Large Object (BLOB) values. Also serializing business objects to BLOBs eliminates the possibility of creating database queries such as “select all business objects attached to process model X”. The information, which process model the business object is attached to would be serialized in the BLOB, requiring a full deserialization of the BLOB to determine the required value.

4.6. Definition of Business Objects Types Using XML Descriptors

The user interface supports defining of business object types using a special form which is shown in Figure 4.5. Note that *Name, Address List* and *Password Map* are complex business object types and collections, demonstrating the complexity that can be achieved with this user interface.

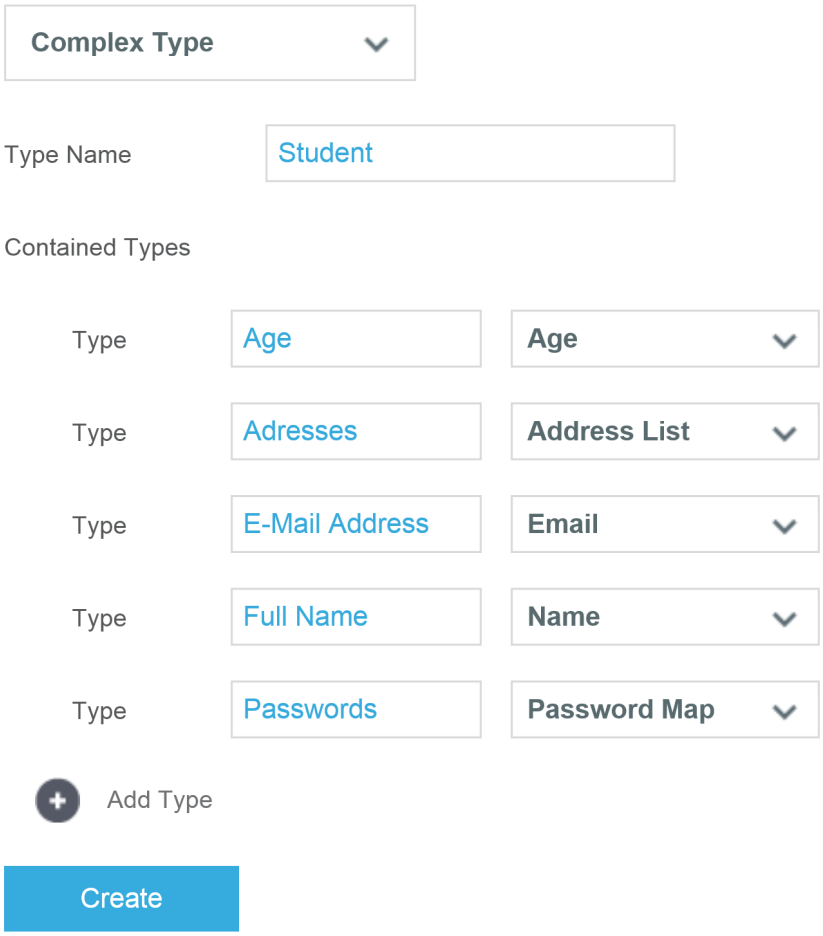

Figure 4.5.: Business Object Type Definition View

Such a web interface is meant for end-users who profit from type safety and support when defining required business object types. Additionally, business object types may be defined using an XML document adhering to an XML schema. Specifying a strict XML schema allows users that with to design business object types via XML descriptors to profit from code completion and syntax checking. It also allows the Clavii engine to validate the XML code contained in Listing 4.9, and use it to inject a working business object type into the running BPMS.

```
<complexType>
    <name>Person</name>
    <hexColor>#42368C</hexColor>
    <properties>
        <property>
            <type>Name</type>
        </property>
        <property>
            <!--this is the name of the type that this field should have-->
            <type>Date</type>
            <!--this is the name of the field,
                the tag is optional and used only when the field name
                differs from the name of the type that the field has-->
            <name>Birth Date</name>
        </property>
        <property>
            <type>Age</type>
        </property>
        <property>
            <type>Gender</type>
        </property>
    </properties>
</complexType>
```

Listing 4.9: XML Definition of *Person* Complex Type

Thereby, any complex object structure definable using the business object framework described in this thesis can also be defined using XML files. Note that in Listing 4.9 the property of type “Name” may be a complex business object type as well, defined in a different XML file. This demonstrates how nesting of complex types is achievable.

Such an XML descriptor can be uploaded using, for example, the user interface, which results in the actual creation of the business object types described in the files. After parsing and type creation are finished, business object types can be used alongside existing ones (cf. Section 4.2). Internally, the parsing is done using JAXB [15], a framework that allows for the serialization of entire classes and their instances to and from XML, with the help of annotations on the classes and their members

4.6.1. Using Java Annotations to Prepare a Class for XML Serialization / Deserialization

The combination of Hibernate and JAXB annotations allows parsing an XML definition like the one shown in Listing 4.9. Afterwards, an instance of the *ComplexBusinessOb-*

jectType class may be created and serialized to a database, thereby creating a new complex business object for use in the user interface. An example of the annotations required for a typical class is given in Listing 4.10.

```
1  //Instruct JAXB on the ordering the XSD should inforce
2  //Also marks this class as a class that should be serializable to XML
3  @XmlType(propOrder = {"name", "displayName", "hexColor"})
4  //Instruct JAXB which other classes are related to this class
5  @XmlSeeAlso({SimpleBusinessObjectType.class, ComplexBusinessObjectType.class, EnumBusinessObjectType.class,
       ListBusinessObjectType.class, MapBusinessObjectType.class})
6  //Instruct Hibernate that this class is a database entity
7  @Entity
8  public abstract class BusinessObjectType implements Serializable {
9          //Instructs JAXB that this field is required for valid parsing
10         @XmlElement(required = true)
11         //Instructs Hibernate that this field is required for serialization
12     @NotNull
13     protected String name;
14         //Instructs JAXB that this field is not required for valid parsing
15         @XmlElement(required = false)
16     protected String displayName;
17         //Instructs JAXB that this field is not allowed in the XML representation
18         @XmlTransient
19     protected Long owner;
20         //Instructs JAXB that this field is not required for valid parsing
21         @XmlElement(required = false)
22     protected String hexColor;
23 ...
```

Listing 4.10: Annotations Required for JAXB and Hibernate

This is a minimal set of annotations required to make a class Hibernate and JAXB compatible, relations and other specifics require additional configuration, which are not in the scope of this thesis.

4.7. Summary

This section has shown the concept and implementation of the core requirements of the Clavii BPM Cloud, i.e., Requirements REQ-1, REQ-2, REQ-3, and REQ-4. These requirements address the support of complex business object types, business objects, and business object instances, as well as collections of business objects and their inheritance model. Examples given in this section are mostly specific to BPMS and business objects, but in theory one could apply the concept demonstrated here to other use cases as well.

The inheritance concept, allowing only the addition of fields to a complex business object type and the replacement of the business object type assigned to a field with a direct sub-type is sufficient for the requirements of the Clavii engine, specifically Requirement REQ-3, but might need work when generalizing the concept for other uses.

5

Service Task Plug-Ins and Process Triggers

In order to support the execution of Java code and programs for process models running in the Clavii engine, the Clavii BPM Cloud allows developers to write plug-ins. Plug-ins can be applied to the tasks of a process model, thereby converting the task in question into a service task. Input and output parameters of the plug-in may then be mapped to complex business objects in the process model. One can even map to parts of complex business objects, as mentioned in Section 3.3.1. Generally, a plug-in may be composed of multiple “operations”, which contain variations of the plug-in’s functionality (cf. 2.2.1).

Plug-ins can be integrated into the Clavii engine in different ways. These integration options are called plug-in “types”, each serving a different purpose. The different variants of plug-ins are explained in Section 5.1. As the different types should be usable transparently by the process model designer, an object-oriented solution for

calling methods in classes that are unknown at compile-time is necessary. Section 5.2 describes how this can be done using the Reflection API and dynamic dispatching.

One of the main requirements that the Clavii engine should fulfill is providing the ability to use complex business object instances in plug-ins (cf. Requirement REQ-5). Therefore, 5.3 describes the integration of business object instances in plug-ins. Section 5.4 introduces the algorithm necessary to fulfill Requirement REQ-6, i.e., allowing the Clavii engine to handle variable length arguments for service task input parameters.

Finally, Section 5.5 and section 5.6 contain information on triggers, effectively plug-ins which can start a process instance, and the XML descriptors for plug-ins and triggers.

5.1. Plug-In Types

In general, three types of plug-ins are supported by the Clavii BPM Cloud, which are explained in detail in the following sections. The idea is to allow plug-in developers flexibility, giving them multiple options on how to integrate their components. Furthermore, the different plug-in types' usage is completely transparent to the process model designer. The process model designer does not even notice what kind of a plug-in he is using in his process model.

5.1.1. Integrated Plug-Ins

An *integrated plug-in* is written in Java and is bundled with the Clavii BPM Cloud. To be more precise, the source files for integrated plug-ins are actually located inside the Clavii project files and are compiled at the same time the Clavii engine is. Integrated plug-ins do, however, do not use any API unavailable to external plug-ins. Having the exactly same programming model for external plug-ins and integrated plug-ins allows for plug-ins that were previously not included directly in the Clavii project files to be repackaged and integrated into it without having to change any code.

Integrated plug-ins aim at offering fundamental functionality like sending e-mail messages at certain points in the business process, executing SQL queries, managing file uploads

to services like DropBox or OneDrive, and even managing CalDAV compatible Calenders. All integrated plug-ins have very general uses and, potentially, offer functionality to a multitude of users.

5.1.2. OSGi Plug-Ins

Users or enterprises wishing to extend the Clavii BPM Cloud's functionality with custom, external plug-ins can write plug-in classes and bundle them as OSGi bundles. OSGi allows Java classes to be loaded, including dependencies, into the class path of a running Java application. This means that an enterprise can upload an OSGi bundle in the user interface containing the implementation of a plug-in accompanied by an XML descriptor (cf. Section 5.6), which enables the Clavii engine to provide a new plug-in for use in process models. *OSGi plug-ins* have exactly the same feature-set and programming model as integrated plug-ins do. After initial loading, which occurs when the plug-in is uploaded and once on every Clavii server restart the plug-in code is also executed at the same speed as the integrated plug-in code. This holds true because OSGi does not use proxies but normal java class loading features, i.e., the overhead is only during the initial loading of the bundle, not during component code execution.

5.1.3. Web Service Plug-Ins

Web service plug-ins allow for calling existing web services that are not aware of Clavii through a generic web service calling interface. This is made possible by leveraging the JAX-WS [14] framework. Additionally, web services created for the Clavii BPM Cloud, using the complex type framework consisting of *Map* and *List* nesting (cf. Section 5.3), can be called. These web services must be written in Java, as they rely on the web service host language understanding and delivering Java *HashMaps* as input/return values. Web service plug-ins not requiring the use of complex business object types, but only simple data types known to standard XML can be written in any language and consumed by the Clavii engine. To use a web service, a user or enterprise must upload a plug-in descriptor detailing input and output parameters of the web service.

Web services have the advantage of not putting additional workload on the BPMS, but instead distributing it to other servers. The drawback is that the SOAP-based communication between the BPMS and the server hosting the web service. To be more precise, the communication may introduce slight delays in the process instance execution speed [33]. Furthermore, the availability of the web services, and therefore the stability of the business processes hosted on the BPMS, are not guaranteed to the level that they are when hosting all plug-ins locally.

5.2. Calling Plug-Ins Using the Java Reflection API

Integrated and OSGi plug-ins reside on the Clavii engine class path while the BPMS is running. Particularly, instances of them can be created using the Java Reflection API [17]. The Reflection API allows dynamically calling methods in classes at run-time, using the class and method name. As a consequence, class and method names are not required when compiling the BPMS. This allows for inclusion of integrated plug-ins by just saving the plug-in classes and XML descriptors into a folder accessible to the Clavii engine at start-up. In the context of OSGi plug-ins, reflection is necessary, as class and method names of a plug-in are not known when compiling the BPMS.

Parameters and return values of plug-ins are fully supported, and even exceptions thrown by the plug-ins can be caught in the Clavii engine which is useful in the context of correctness by error resolution (cf. Section 6.5). As there are three types of plug-ins (cf. Section 5.1), three different classes are necessary to pass parameters to plug-ins and initiate their execution. Listing 5.1 shows an example of calling a method of a class that is not known at compile-time.

```
1 //instructs the default classloader to load the "someclass" class
2 Class<?> someclass = Class.forName("someclass");
3 //Get a reference to the "somemethod" in the "someclass" class
4 Method  method = c.getDeclaredMethod ("somemethod");
5 //clazz.getConstructor().getInstance() delivers a new instance of the class
6 //params is an array of parameters to pass to the method
7 method.invoke (clazz.getConstructor().newInstance(), params);
```

Listing 5.1: Calling Methods Using Reflection

Next we examine how the reflection API is used in the Clavii engine to facilitate transparently calling the different plug-in types

5.2.1. Use of Dynamic Dispatching for Calling Plug-Ins

Calling operations of plug-ins must be transparent towards the process model, to allow exchanging a plug-in implementation from, for instance, an integrated plug-in to a web service plug-in. For this reason the Clavii engine uses dynamic dispatching to ensure that each plug-in calls the correct code necessary for creating an instance of its class using reflection. Therefore, for every type of a plug-in one sub-class of the *Plugin* class exists, i.e., *IntegratedPlugin*, *OSGIPlugin*, and *WebServicePlugin.*

The *Plugin* class has an abstract method with the signature shown in Listing 5.2.

```
1 public abstract Object callDispatcher(PluginCallDispatcher dispatcher, String methodName, Map<String, Object>
      parameters)
```

Listing 5.2: Abstract Method *callDispatcher*

The abstract nature of the method forces the three sub-classes of *Plugin* to implement the *callDispatcher* method, allowing each sub-class to call the code necessary for executing the respective plug-in type. The *PluginCallDispatcher* is an interface defining the methods necessary to execute any one of the plug-in types (cf. Listing 5.3).

```
1 public interface PluginCallDispatcher {
2     Object call(IntegratedPlugin integratedPlugin, String methodName, Map<String, Object> parameters);
3     Object call(OSGIPlugin osgiPlugin, String methodName, Map<String, Object> parameters);
4     Object call(WebServicePlugin webservicePlugin, String operationName, Map<String, Object> parameters);
5 }
```

Listing 5.3: Interface *PluginCallDispatcher*

Through the use of polymorphism, the abstract method *call* with separate implementations for each plug-in type allows for dynamic dispatching, i.e., all sub-classes of class *Plugin* have an implementation of the method *callDispatcher* identical to the one in Listing 5.4.

```
1 @Override
2 public Object callDispatcher(PluginCallDispatcher dispatcher, String methodName, Map<String, Object> parameters){
3     return dispatcher.call(this, methodName, parameters);
4 }
```

Listing 5.4: Implementation of Method *callDispatcher*

Even though all sub-classes implement the abstract method with exactly the same code, there is a difference for the compiler. If one would not use dynamic dispatching as shown in Listing 5.3, but instead have one *callDispatcher* method located directly in the super-class *Plugin*, Java could not infer which of the overloads of method *call* should be called. Therefore, such code would not even compile, as there is no overload *call(Plugin plug-in, String methodName, Map<String, Object> parameters)*. This is in essence the same limitation that was shown in Listing 4.2.

Class *PluginCallDispatcherImpl*, contains the implementation of the interface shown in Listing 5.3 (cf. Appendix B.2). Doing so is recommended, as the actual calls to the plug-in operations are an essential part of the Clavii engine.

5.3. Using Complex Business Object Instances in Plug-Ins

The plug-in API that the Clavii BPM Cloud offers has one fundamental requirement (cf. Requirement REQ-5): The plug-in code must not require any Clavii engine specific classes. As a consequence, the business object types that are definable using the Clavii engine and user interface must also be usable in plug-in implementations without requiring specific classes. This requirement results in a Java *Map* representation of defined business object types (cf. Section 4). Java *Maps* and *Lists* can be used to nest any complex business object type supported. Developers can look up the structure of any business object type their plug-in should be compatible with and use it accordingly. An example for a plug-in using an instance of business object type *Person* nested into a *HashMap* is given in Listing 5.5.

```
1  public Map<String, Object> mixPeople(Map<String, Object> parameters) {
2      Number person1Age = (Number) parameters.get("Person 1.Age");
3      Number person2Age = (Number) parameters.get("Person 2.Age");
4      String person1Firstname = (String) parameters.get("Person 1.Name.Firstname");
5      String person2Firstname = (String) parameters.get("Person 2.Name.Firstname");
6      String person1Lastname = (String) parameters.get("Person 1.Name.Lastname");
7      String person2Lastname = (String) parameters.get("Person 2.Name.Lastname");
8
9      Map<String, Object> toReturn = new HashMap<>();
10
11     toReturn.put("Combined Person.Age", (personAAge.intValue() + personBAge.intValue()) / 2);
12     toReturn.put("Combined Person.Name.Firstname", personAFirstname + '-' + personBFirstname);
13     toReturn.put("Combined Person.Name.Lastname", personALastname + '-' + personBLastname);
14     toReturn.put("Combined Person.Birth Date", new Date());
15     toReturn.put("Combined Person.Gender", "Female");
16
17     return toReturn;
18 }
```

Listing 5.5: Plug-In Example Using Complex Business Object Types

The plug-in implementation shown in Listing 5.5 concatenates the *Firstname* of one *Person* business object instance with the *Lastname* of another. A plug-in always expects a *Map<String, Object>* object as the sole input parameter for each of its operations. The return value can either be *void*, i.e. no return value, a single parameter, or a *HashMap* object containing multiple business object instances. Output parameters consisting of complex business objects are created by putting all individual business object instances required to compose the complex business object instance into a *HashMap* object, adhering to the structure of the respective complex business object type.

In case the input or output parameters are collections of business object types, or a complex business object type contains a collection of other business object types, the Clavii engine passes the business object instances nested into a *List<HashMap<String, Object>>*. This allows for greater flexibility concerning the structure of business object types.

This approach also supports the inheritance of complex business object types and usage of instances of derived business object types in plug-ins. Consider a complex business object type *Customer* that extends business object type *Person*. Business object type *Customer* has an additional field *Regular Customer* of data type *Boolean*. Our framework detects that business object type *Customer* extends business object type *Person*. Hence it allows the mapping of a business object of business object type

Customer to an input parameter *Person 1* of the *mixPeople* plug-in, (cf. Listing 5.5). Particularly, *mixPeople* has no knowledge of the *Customer* business object type, but can still execute normally. Extended complex types can only add fields to existing complex types, i.e., they can never rename or remove them.

One challenge when designing the inheritance model was ensuring that plug-ins would not delete the sub-type information upon outputting the business object instance from the plug-in. For instance, if input parameter *Person 1* and output parameter *Combined Person* are connected to the same *Customer* business object, the fields that are not output by the plug-in, i.e., *Regular Customer*, could be lost. We prevent this by using a single *Map* for saving all business object instances, thereby "automatically" preventing fields missing from plug-in output parameters from erasing existing information. This has the additional advantage that plug-in developers do not have to output an entire complex business object instance but only the fields they actually modify. Listing 5.6 shows the code that the Clavii engine runs after receiving a *HashMap* return value from a plug-in.

```
1  //is the return value a map?
2  if (returnValue instanceof Map){
3   //iterate over the mapping,
4   //i.e., the names of the business objects in the process mapped to
5   //the names of the business objects in the plug-in
6   for (ProcessToPluginVariableKeyMapping concreteOutputMapping : outputMappingForThisTask){
7    //saves one individual simpletype to one key in the activiti internal business object map
8    execution.setVariable(concreteOutputMapping.getProcessVariableKey(),((Map<String, Object>) returnValue).get(
         concreteOutputMapping.getPluginVariableKey()));
9   }
10 }
```

Listing 5.6: Handling of Plug-In Return Values

Listing 5.6 shows how the Clavii engine does not overwrite the entire complex business object instance, but instead only those parts returned by the plug-in.

5.4. Variable Length Arguments

To fulfill Requirement REQ-6, variable length arguments for service tasks, a mechanism supporting this was introduced into the *Caller* class. The *Caller* class is the class that contains the code for calculating the required parameters of a plug-in operation (cf.

Section 6.4.2), and making sure that they are all accounted for before starting the service task. Omitted from the code that was shown in Listing 6.3 in Section 6.4.2 was the algorithm for detecting variable length arguments.

The following is an example of a scenario where this method would be helpful: A service task in a process model uses a plug-in operation *Summarize* which requires a parameter of the business object type *Integer List*, i.e., a list of integers. The operation calculates the sum of the integers in the list and returns it. This *Integer List* parameter is called *SummarizeInputList*. The process model does not have a business object of the *Integer List* business object type though, but it does have two business objects of business object type *Integer*, *IntX* and *IntY*.

The Clavii engine allows the latter to be mapped to the same input parameter *SummarizeInputList*. As a result a virtual *List* of business object instances containing the parameters is created. The respective algorithm is presented in Listing 5.7. Particularly, it is executed once for every parameter present in the input mapping for the service task.

The algorithm, after the required and optional parameters were calculated (cf. Listing 6.2), removes all correctly mapped parameters from the list of input parameters. If the business object instance in question is neither required, nor optional, the algorithm classifies it as a *varArgs* parameter. Afterwards, it calculates which *List* input parameter the business object is mapped to by analyzing the mapping information of the business object.

Returning to the previous example, this means that when the algorithm examines the business object instance belonging to the business object *IntA*, it recognizes that *IntA* is mapped to the input parameter *SummarizeInputList*. By checking the name of the business object the algorithm determines that *IntA* is not a complex business object instance. Finally, it checks if there is already a *List* inserted into the parameters map by a previous iteration of the algorithm. As there is no *List* yet, the algorithm creates a new *List<Object>* and inserts the business object instance into it. Afterwards, the new *List* is written to the input parameters for that service task.

Further iterations, for instance for the second business object *IntB*, also present in the process model, are identical, except for the fact that they recognize and utilize the *List*

created by the iterations before, instead of creating a new one. This way the ad-hoc *List* is filled with all parameters that are mapped to a specific list input parameter of the service task.

```
1  String lastProcessVariableName=""; //defined outside of the for-loop iterating all input parameters
2  Map<String, Object> lastVirtualComplexType = null; //these variables are persistent across all iterations
3  ...
4
5  //remove parameter from required/optional lists
6  if (requiredParameters.remove(pluginVariableKey) == null && optionalParameters.remove(pluginVariableKey) == null) {
7     //parameter was not in optional or required parameters, this is a single value mapped to a list (varargs)
8     //find the name of the top level business object instance this parameter belongs to (PersonA.Age => PersonA)
9     String correctProcessVariableName = processVariableKey.substring(0, processVariableKey.indexOf('.'));
10    //find the name of the list parameter that this varargs parameter is trying to map to
11    String correctPluginVariableName = pluginVariableKey.substring(0, pluginVariableKey.indexOf('.'));
12
13    if (processVariableKey.contains(".")) {
14       //parameter is of a complex business object type
15       //perform check if there is already a virtual list from a previous iteration, create a new one if not
16       List<Map<String, Object>> adHocList = parameters.containsKey(correctPluginVariableName) ?
17       (List<Map<String, Object>>) parameters.get(correctPluginVariableName) : new LinkedList<Map<String, Object>>();
18
19       //variables arrive alphabetically, this allows us to rebuild complex business object instances and
20       //insert them into maps, just as the plug-in operation expects
21       if (!correctProcessVariableName.equals(lastProcessVariableName)) {
22          lastProcessVariableName = correctProcessVariableName;
23          lastVirtualComplexBusinessObject = new HashMap<>();
24          adHocList.add(lastVirtualComplexBusinessObject);
25       }
26       lastVirtualComplexBusinessObject.put(processVariableKey.substring(processVariableKey.indexOf('.') + 1,
               processVariableKey.length()), inputVariable);
27
28       //the virtual list fills the required parameter, remove it if it wasnt already
29       requiredParameters.remove(correctPluginVariableName);
30       //put the new or changed virtual list into the input parameter map of the plug-in operation
31       parameters.put(correctPluginVariableName, adHocList);
32    } else {
33       //parameter is of a simple business object type
34       //perform check if there is already a virtual list from a previous iteration, create a new one if not
35       List<Object> adHocList = parameters.containsKey(correctPluginVariableName) ?
36       (List<Object>) parameters.get(correctPluginVariableName) : new LinkedList<>();
37
38       adHocList.add(inputVariable);
39
40       //the virtual list fills the required parameter, remove it if it wasnt already
41       requiredParameters.remove(correctPluginVariableName);
42           //put the new or changed virtual list into the input parameter map of the plug-in operation
43       parameters.put(correctPluginVariableName, adHocList);
44    }
45 } else {
46    parameters.put(pluginVariableKey, inputVariable);
47 }
```

Listing 5.7: Detection of VarArgs and Creation of Virtual Lists

5.5. Triggering Process Instances

In contrast to plug-ins, *triggers* are not executed as part of the process flow, i.e., during the execution of a service task, but can actually "trigger" a process instance. Triggering a process instance not only means instructing the Clavii engine to instantiate a certain process model, thereby creating the process instance, but also handing business object instances directly to the new process instance. These could be, for instance, headers or attachments of a received e-mail. Basically, a trigger can be run at fixed intervals and told to perform a certain task, like checking an IMAP account for new e-mail. When the specified trigger condition is met, it can start a new process instance.

The aforementioned IMAP account trigger might be used to monitor an e-mail account and automatically start a process instance of a business process that archives email messages for every new mail received. Multiple triggers may be assigned to a process model. As a trigger for a certain process model is defined on the start event, the start event has its set of output parameters fixed after the first trigger is defined. A use case for multiple triggers could be to define an IMAP trigger to archive every e-mail as it arrives and, additionally, having a *Timer* trigger with an interval of 24 hours set to run the archiving business process every day at 03:00.

In summary, triggers are assigned to a process model and configured to start process instances when a set of conditions is met. This configuration is done by mapping business objects that have the configuration parameters set as their default values mapped to the trigger's input parameters. The output parameters of a trigger can be used to start a process instance with default values for the mapped business objects based on the event that triggered the process instance, for instance the content of a received e-mail.

Next we examine the implementation of triggers and the code supporting their execution in the Clavii engine.

5.5.1. Leveraging the Reflection and Executor Frameworks for Triggering Process Instances

The triggers rely on the same principle as plug-ins, of not requiring developers of triggers to import any Clavii engine specific classes into their projects. Developers of triggers do, however, have to adhere to a few strict rules, listed in the following excerpt from the Clavii knowledge base:

1. A trigger class must implement *Serializable* and *Runnable*, this ensures that the state of a trigger can be persisted in a database and reloaded after server restarts. State information must be saved in fields in order for serialization to work.
2. A trigger has to keep a *Map<Long, Object>* field, mapping IDs of the process models it should trigger to whatever information it needs for triggering (it can also use multiple maps). For instance, the following code maps timer intervals in seconds as integers to process model IDs:

```
1 private final Map<Long, Integer> idMapping = new HashMap<>();
```

3. The method *run*, which the class is forced to have implementing interface *Runnable*, gets called only once after installing a new trigger via OSGi and once on every server restart. The actual monitoring that the trigger does should be set up in the *run* method, leveraging the *SingleThreadScheduledExecutor* class, part of the Java Executor Framework [5]. The *run* method must restart all monitoring activity, using the information saved in the persistent fields of the class.
4. Two methods to register and deregister process models to the trigger must be added.
5. The *register* method must have a *Long* as its first parameter, this is the ID of the process model that will be triggered. The second parameter must be a *Map<String, Object>* containing all business objects required for the trigger to set up its monitoring. The *Map* object is formatted in the same dot-based notation for complex business object instances as the input map of a regular plug-in (cf. Section 5.3).
6. One field must be defined with the following signature:
"*transient java.lang.reflect.Method*"

The name of the field does not matter (e.g., *private transient Method receiver;*). This field is injected using reflection once the trigger gets loaded. It represents a reference from the trigger implementation to the Clavii engine.

7. When the trigger detects something that it should trigger for, call the code in Listing 5.8 from the *SingleThreadScheduledExecutor*.

```
1 Map<String, Object> output= new HashMap<>();
2 output.put("variablename", variablevalue);
3 receiver.invoke(null, [PROCESSMODELIDTOTRIGGER], output);
```

Listing 5.8: Triggering Code

[PROCESSMODELIDTOTRIGGER] represents the ID of the process model that is cached in the “idMapping” *Map*. The *Map<String, Object>* in Line 1 contains all the information that should be used as parameters for the new process instance. Line 3 statically calls the method behind the *Method* field that was defined in the previous step.

The entire implementation of a *Timer* trigger, which starts process instances after a given interval of seconds and passes the starting time to the process instance, is shown in Appendix B.3.

The Java Executor Framework allows for the use of thread pools and interval-based scheduling of tasks. By utilizing thread pools instead of threads for each monitoring activity started by a trigger, the efficiency of the Clavii engine can be increased, as *Thread* objects are expensive in their creation [12].

The use of reflection to inject the *Method* field into the trigger at run-time is essential. Otherwise the trigger, which for the most part runs completely autonomously, would have no way of communicating with the Clavii engine and instructing it to start a process instance. The code for injecting the *Method* field at run-time, and running the trigger’s *run* method afterwards, is given in Listing 5.9.

```
1  //get the last state the trigger was in
2  Serializable serializedState = trigger.getSerializedState();
3  //iterate all fields and find the "Method" field
4  for (Field field : serializedState.getClass().getDeclaredFields()) {
5          if (field.getType().equals(Method.class)) {
6                  //enable injecting private variables
7                  field.setAccessible(true);
8                  try {
9                          //inject the first method (the only one) from TriggerReceiver.class
10                         //this method is set up to receive the output of a trigger
11                         field.set(serializedState, TriggerReceiver.getClass().getMethods()[0]);
12                 } catch (IllegalAccessException e) {
13                         e.printStackTrace();
14                 }
15         }
16 }
17 //start the trigger (execute run method in new thread)
18 new Thread((Runnable) serializedState).start();
```

Listing 5.9: Injecting a Method into a Trigger

The non-transient fields are used in combination with the rule that the class has to implement *Serializable* to enable the Clavii engine to freeze and serialize the trigger for persisting to a database table on shutdown. This way the trigger can be “restarted” after starting the BPMS. Listing 5.9, shows the injection of the *Method* field and running of the trigger in a new thread, which is actually part of this restarting procedure.

5.6. XML Descriptors for Plug-Ins and Triggers

Plug-ins and triggers can be configured via XML descriptors, similar to how types can be defined using XML, (cf. Section 4.6). The XSD schema for the XML descriptors for a plug-in is very straightforward, as can be seen in Listing 5.10.

```
1      <xs:complexType name="plug-in" abstract="true">
2          <xs:complexContent>
3                  <xs:sequence>
4                      <xs:element name="location" type="xs:string"/>
5                      <xs:element name="name" type="xs:string"/>
6
7                      <xs:element name="description" type="xs:string" minOccurs="0"/>
8                      <xs:element name="author" type="xs:string"/>
9                      <xs:element name="operationDescriptions">
10                         <xs:complexType>
11                             <xs:sequence>
12                                 <xs:element name="operationDescription" type="operationDescription" maxOccurs="
                                       unbounded"/>
13                             </xs:sequence>
```

```
14                          </xs:complexType>
15                      </xs:element>
16                  </xs:sequence>
17          </xs:complexContent>
18      </xs:complexType>
19
20      <xs:complexType name="operationDescription">
21          <xs:complexContent>
22                  <xs:sequence>
23                      <xs:element name="methodName" type="xs:string"/>
24                      <xs:element name="name" type="xs:string"/>
25                      <xs:element name="description" type="xs:string" minOccurs="0"/>
26                      <xs:element name="inputParameters" minOccurs="0">
27                          <xs:complexType>
28                              <xs:sequence>
29                                  <xs:element name="inputParameter" type="operationParameter" minOccurs="0" maxOccurs=
                                        "unbounded"/>
30                              </xs:sequence>
31                          </xs:complexType>
32                      </xs:element>
33                      <xs:element name="outputParameters" minOccurs="0">
34                          <xs:complexType>
35                              <xs:sequence>
36                                  <xs:element name="outputParameter" type="operationParameter" minOccurs="0" maxOccurs
                                        ="unbounded"/>
37                              </xs:sequence>
38                          </xs:complexType>
39                      </xs:element>
40                  </xs:sequence>
41          </xs:complexContent>
42      </xs:complexType>
43
44      <xs:complexType name="operationParameter">
45          <xs:complexContent>
46                  <xs:sequence>
47                      <xs:element name="type" type="xs:string"/>
48                      <xs:element name="name" type="xs:string"/>
49                      <xs:element name="description" type="xs:string" minOccurs="0"/>
50                      <xs:element name="optional" type="xs:boolean"/>
51                  </xs:sequence>
52          </xs:complexContent>
53      </xs:complexType>
```

Listing 5.10: XSD Plugin Schema

In essence, the XSD dictates that a *plugin* must have a *location* element, i.e., the class name for integrated or OSGi plug-ins or URI of the Web Services Description Language (WSDL) file for web services. Furthermore, descriptive properties such as *author* and *name* as well as a list of *operationDescription*s are required. Moreover, an *operationDescription* element consists of a *methodName* element, i.e., the method or operation name in the class or WSDL, some descriptive properties such as *name* and

description and lists of input and output parameters. Each parameter is assigned one business object type that exists in the Clavii engine.

Omitted from Listing 5.10 are sub-classes of the abstract class *Plugin* (i.e., *IntegratedPlugin*, *OSGIPlugin*, and *WebServicePlugin*), as they are only markers. This means that instead of having <plugin> as the opening tag of a valid plug-in XML file, one has to write, e.g., <osgiplugin> to force the file to be interpreted as the definition of an OSGi plug-in. The tags are identical to those of <plugin>, as dictated by the XSD.

Triggers can also be defined using an XML document. The XSD is virtually identical and therefore omitted. The only difference is that the names of the trigger's register and deregister methods have to be added as tags. XML definitions of a sample plug-in and sample trigger can be reviewed in Appendix B.4 and Appendix B.5 respectively.

5.7. Summary

In summary, this section has explained how the Clavii engine supports Requirements REQ-5 and REQ-6, i.e., the use of complex business objects in plug-in implementation and allowing service tasks expecting lists of business objects to be mapped as variable length arguments. It has also given examples of the different types of plug-ins the Clavii BPM Cloud supports, and what techniques, such as dynamic dispatching and the Reflection API, were used to achieve this support. Also, the topic of triggers, their role, as well as their implementation, was discussed. Finally, the use of XML descriptor files to configure plug-ins for integration into the running BPMS was explained.

6

Ad-hoc Process Model Execution Control

In the following we explain the concept and implementation of one of ad-hoc control of process flow. To support Requirements REQ-9 and REQ-10, a BPMS must allow for ad-hoc workflow. This also allows for process models to execute which have, for instance, service tasks that have unmapped - but required - input parameters. The advantage, compared to a correctness by construction principle [21], is the ability to start testing the process model at an early stage of development. For instance, while process model designers are still modeling the data flow or setting up individual service tasks, the quality assurance team may start testing the process model and the individual service tasks by actually executing the process model.

Ad-hoc process execution control is used to support this use case by rerouting the execution to dynamically created user tasks. These user tasks show a form allowing users to input values for the missing parameters. This prevents the service task from

executing without having all the necessary parameters present. This means that that the user can input the missing parameters and run the service task normally afterwards.

Also, if any service task has all required parameters present, but fails later on, internally, due to an exception, this allows the engine to transparently generate a user form asking the user if he wants to retry the task. The generated user form even offers the user the ability to re-enter all input parameter data that the service task used in the current process instance. The BPMN engine then reroutes the process execution to display a respective user form to the user. If the user then chooses to retry a service task, with different parameters, the service task gets executed with the new parameters, and, if successful, the process flow continues normally at the expected point in the process model. This can not, however, ensure atomic execution of the actual service task logic, i.e., if the service task fails after having completed partially a retry will lead to multiple executions of the same code. One could eliminate this problem partially if the plug-in code could serialize state information for every process instance calling it, but the current prototype does not support this yet.

The following sections describe the concepts required to provide ad-hoc process model execution control. First, Section 6.1 describes the concept for rerouting the process instance execution in case of an missing parameters or run-time errors. The actual implementation of the first part of the concept, pausing the process instance, is described in Section 6.3. Then Section 6.4 continues with the description of the different situations where pausing the process instance is necessary. After the process instance is paused, an error correction form is displayed to the user, the construction of which is described in Section 3.5. After the form is completed the process instance can be resumed with the corrected parameters, this final step is described in Section 6.6.

6.1. Concept for Handling Missing Parameters and Run-time Errors

The implementation concept for the two very similar use cases, requesting input parameter values for incorrectly mapped input parameters and the possibility of retrying a service task which threw an exception with different parameters, is shown in Figure 6.1.

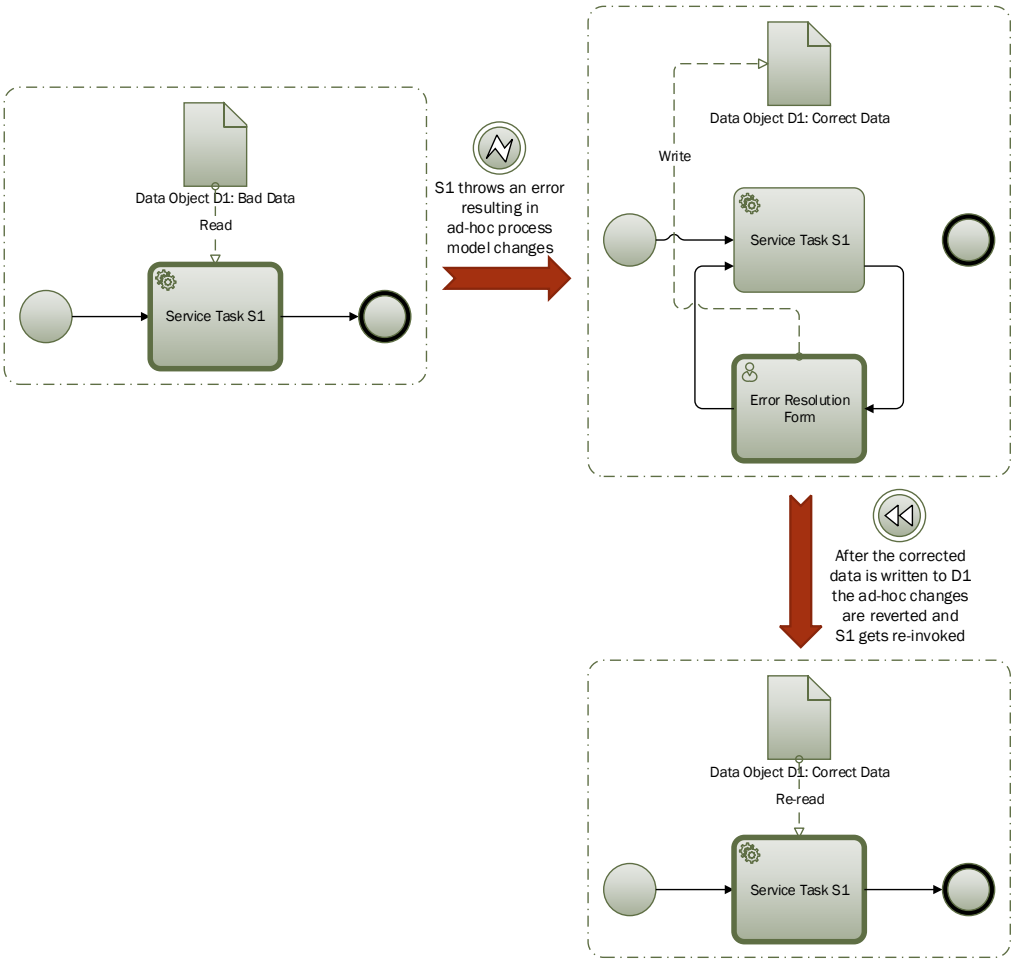

Figure 6.1.: Ad-hoc Process Execution Control Concept

Basically, when missing parameters are detected before the execution of a service task, or a service task throws an exception at run-time, a user form is generated which helps him correct the error in question. It does this by allowing the user to add values for the missing parameters or change business objects that are in use by the service task. The user task containing the user form is then inserted into the process flow directly after the service task in question, replacing the sequence flow that the process instance would normally take with one leading to the user task. Additionally, a sequence flow connecting the user task back to the service task is inserted into the process flow as well, effectively creating a loop control flow structure between the faulty service task and the user form. Once the user form is completed and execution arrives back at the service task, the ad-hoc changes applied to the process instance are reverted.

6.2. Implementation of the Concept in the Clavii Engine

The concept shown in Section 6.1 had to be adapted for use in the Clavii BPM Cloud prototype, mostly because the forms that the user interface uses are not prefab user forms, but generated on the fly using GWT. Also, as Clavii allows users to execute user tasks in their process models by clicking on them at run-time, the basic concept would have shown them the ad-hoc changes and had them click on the error correction forms in the run-time view. As to not confuse users by showing them the ad-hoc process model changes in the user interface the implementation concept hides the actual changes to the process model.

This allows us to use a so-called *receive task*[1] instead of a user task in the implementation of the concept. A BPMN receive task forces the process instance to wait until receiving a signal from an external source, in this case the user interface. The rest of the concept stays the same, the only real difference is that the process instance pauses because of a receive task and not because of a user task and that the missing parameters are not collected by a user task but by a form dynamically generated in the user interface. The error message or list of missing parameters to facilitate this is

[1] http://www.activiti.org/userguide/#bpmnReceiveTask

generated and sent to the user interface after the execution of the process instance is halted in the receive task. The user can then double-click the failed service task, at which point the user interface generates a dynamic form that, after requesting all necessary data from the user, sends the input back to the Clavii engine. The Clavii engine can then write the new or changed business object instances to the process instance. Once the data is persisted in the process instance, the *signal* event for the receive task is triggered, which causes the execution token to jump from the receive task back in front of the original service task. The two temporary sequence flows and the receive task are then erased from the model, effectively reverting the ad-hoc changes, as described in the original concept, thereby allowing execution to continue normally.

6.2.1. Use in Manual Gateway Execution

The implementation of this concept can be used in a simplified form to facilitate manual gateway execution, as discussed in Section 3.3.3. Manual gateway execution pauses the process instance using the same receive task method as soon as the process instance reaches a gateway marked as manual. The user interface reacts to a paused gateway by offering the user a pop-up where he can choose the execution path he would like the process instance to continue on (cf. Figure 6.2).

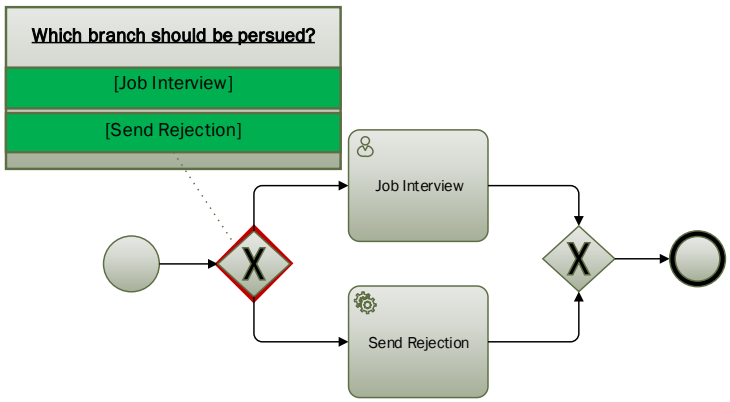

Figure 6.2.: Manual Execution Path Selection at Run-time

As the run-time user interface for process instances shows the respective process model itself, as opposed to the work lists that many BPMS offer[1, 2, 3, 8, 10], this is an extremely fast way for users to navigate an XOR gateway with any amount of outgoing branches and simple decision logic, which would otherwise necessitate an extra user task.

6.3. Implementation of Ad-hoc Pauses

The concepts discussed in Section 6.1 rely on pausing the execution of a process instance. Generally, a BPMS does not support pausing of process instances at arbitrary points in time, e.g., while executing a service task or traversing a sequence flow. The aforementioned pausing concept uses the BPMN[2] receive task coupled with ad-hoc sequence flows and *signal* events to circumvent this limitation. The advantage of this approach is that, if there is parallel execution due to control structures such as an AND gateway, the engine only pauses the branch of the execution which should be paused. Listing 6.1 shows the ease with which the implementation of this concept is possible using the Activiti BPM engine API. Note that in the context of the Activiti BPM engine API a task is referred to as an *activity*. Also, a sequence flow is referred to as a *transition*.

```
1 ScopeImpl process = thisTask.getParent();
2 ActivityImpl pause = process.createActivity("PauseFor" + activityId);
3 pause.setActivityBehavior(new ReceiveTaskActivityBehavior());
4 TransitionImpl toPause = thisActivity.createOutgoingTransition();
5 toPause.setDestination(pause);
6 TransitionImpl loopback = pause.createOutgoingTransition();
7 loopback.setDestination(thisActivity);
8 activityExecution.take(toPause);
```

Listing 6.1: Pausing a Process Instance Using the Concept from Section 6.1

The code in Listing 6.1 is executed at process instance execution, after a service task fails, a manual gateway or a missing parameter is detected. The code is very close to the concept:

In Line 1 the parent of the current task, i.e., the process instance, creates a new task. During creation, in Line 2, the task is assigned a unique ID, composed of a keyword

[2] http://www.omg.org/spec/BPMN/2.0/

identifying it as a pause state and the globally unique ID of the task. The ID is needed for clean-up purposes later on, as the activity is deleted after the process instance execution is resumed. In Line 3 the task is then assigned the behavior of a receive task. A behavior assigns functionality to the task in question, which it lacks directly after initialization. Afterwards, in Lines 4 to 7, two transitions are created, connecting to the new task and the current task, forming a loop. Finally, in Line 8, the *activityExecution*, a reference to the logical execution thread of the current parallel branch, is instructed to “take” the transition leading to the new pause state.

As soon as the execution is paused, the user interface shows the user a user form or gateway decision input, based on the reason for the pause. The gateway input and error correction form is shown in Figure 6.2 and Figure 6.6 respectively. After completion, the input is sent to the Clavii engine along with a special *signal* event targeted at the receive task. The *signal* event allows the process instance to continue execution on the branch that had been paused.

As the second ad-hoc transition connects the receive task back to the service task or gateway that was paused, continued execution leads to invoking that task again. Subsequently, the temporary receive task is deleted, along with the two transitions. This ensures that the process instance continues to flow normally, if no new errors are detected. If, however, errors are detected again, the entire procedure begins anew, re-executing the code from Listing 6.1.

6.4. Detection of Errors Leading to Dynamic Ad-hoc Pauses

The methods for detecting errors that lead to pausing of a process instance and requesting user input varies between the following three use cases:

A manual gateway triggering a pause is flagged as such and only has to send a push notification to the user interface telling it to display the XOR choice pop-up for the current gateway to the user (cf. Figure 3.3.3).

Detecting an error in an internally failed service task is also trivial as the service tasks are called using the Java Reflection API, which allows catching of an *InvocationTarge-*

tException, an umbrella exception for any exception that occurred in the service task code. In this case a list of all input parameters of the failed task has to be assembled and sent to the user interface for use in an error correction form (cf. Figure 6.6).

The most critical use case that causes execution pauses, is the failing of a service task due to a missing but required input parameter. In this case the execution of such a service task should be prevented to ensure that it does not get partially executed and then fails do to an internal exception. Detecting if a parameter is correctly mapped to a business object is challenging because the Clavii engine supports complex mapping structures, an example of which can be seen in Figure 6.4. Partial complex mapping introduces cases in which one can not simply check if an input parameter is null to determine if it is present.

Two cases of partial complex mapping can occur in a process model. Therefore, it is required to analyze for errors. The first case of complex parameter mapping is given in a process model that might have a *Person* business object and uses a service task only requesting a parameter of the business object type *Age* (cf. Figure 6.3).

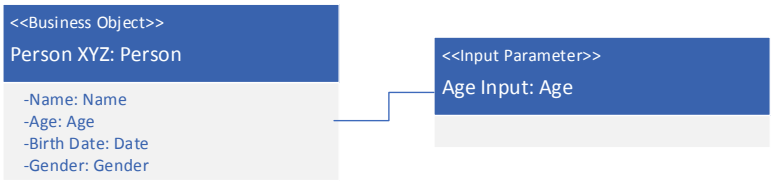

Figure 6.3.: Partial Complex Parameter Mapping Case 1

Consequently, the second case of complex parameter mapping is shown in Figure 6.4. In Figure 6.4 there is a complex parameter *Owner* of business object type *Person,* belonging to a service task input parameter of the *BankAccount* complex business object type. A *Person* consists of business object types *Name, Birth Date, Gender* and *Age*. However, the process model using this service task does not have to supply the entire business object “Owner”, but can rather map parts individually.

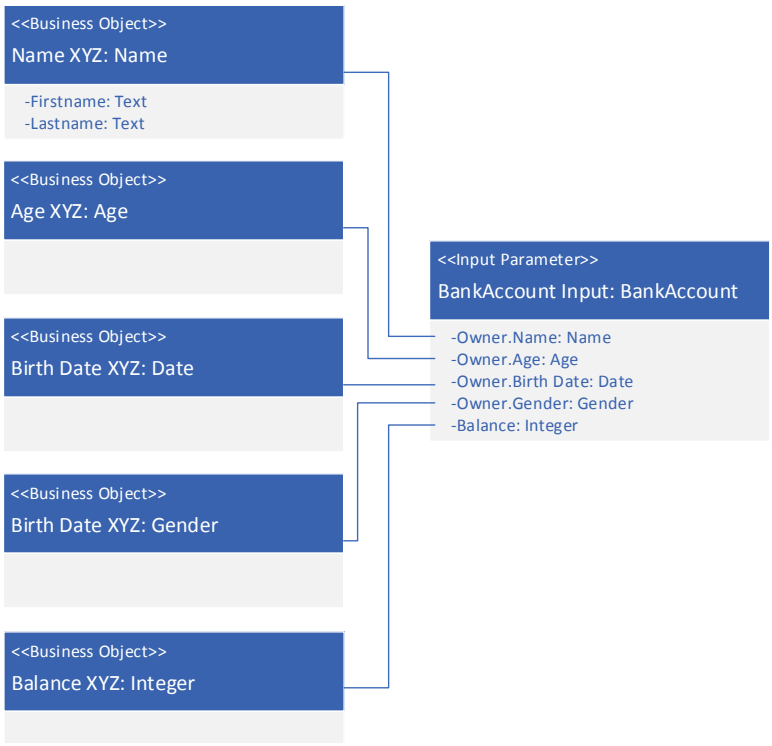

Figure 6.4.: Partial Complex Parameter Mapping Case 2

The implementation of the Clavii engine handles both these cases by using complex business object type structural information, which the underlying Activiti BPM engine has no knowledge of, to create a list of the primitive business objects that are required for execution of the service task. The Activiti BPM engine uses a data structure conceptually similar to a *Map*[3] internally for holding business object instances. Therefore, complex business objects that the Activiti BPM engine is unaware of must be unwrapped to unique positions in the Activiti BPM engine's internal business object map. Section 4.3 details the way an instance of the aforementioned *Person* business object type is unwrapped into a simple dot-based notation.

[3] *http://docs.oracle.com/javase/7/docs/api/java/util/Map.html*

6.4.1. Determining Required Input Parameters

The Clavii engine must create a list of keys for the business object instances for a process instance to check if each key exists and is not null. A keys is a representation of primitive business object instances in the Activiti BPM engine. This is done by the two methods shown in Listing 6.2. Determining input parameters that a service task requires is done based on the list of required parameters belonging the plug-in operation assigned to the service task.

```
1  public Map<String, BusinessObjectType> getRequiredParameters() {
2      Map<String, BusinessObjectType> returnVal = new HashMap<>();
3      for (OperationInputParameter operationInputParameter : inputParameters) {
4          if (!operationInputParameter.isOptional()) {
5              if (operationInputParameter.getType() instanceof ListBusinessObjectType || operationInputParameter.
                   getType() instanceof MapBusinessObjectType) {
6                  //if the type of the parameter is a list or map, there is no need to do recursion
7                  returnVal.put(operationInputParameter.getName(),
8                          operationInputParameter.getType());
9              } else {
10                 //call recursive method to find futher required parameters of complex types
11                 returnVal.putAll(recursiveAddParameters(operationInputParameter.getType(), operationInputParameter.
                       getName()));
12             }
13         }
14     }
15     return returnVal;
16 }
17
18 private Map<String, BusinessObjectType> recursiveAddParameters(BusinessObjectType businessObjectType, String prefix)
       {
19     Map<String, BusinessObjectType> returnVal = new HashMap<>();
20     if (businessObjectType instanceof ComplexBusinessObjectType) {
21         ComplexBusinessObjectType complexBusinessObjectType = (ComplexBusinessObjectType) businessObjectType;
22         for (Map.Entry<String, BusinessObjectType> entry : complexBusinessObjectType.getContainedTypes().entrySet())
               {
23             //this is a complex parameter that might contain further parameters, recursive call
24             returnVal.putAll(recursiveAddParameters(entry.getValue(), prefix + '.' + entry.getKey()));
25         }
26     } else {
27         returnVal.put(prefix, businessObjectType);
28     }
29     return returnVal;
30 }
```

Listing 6.2: Generating a List of Required Parameters for a Service Task

Method *getRequiredParameters()* in Listing 6.2 returns a map containing all keys of parameters required for service task execution, mapped to their business object types. This ensures that the list of missing parameters also contains structural information needed to create and display the user form for correcting the error. Once all required

parameters are determined for a service task, the Clavii engine continues by comparing the actually existing parameters with those on the list, which is described in Section 6.4.2.

6.4.2. Comparing Required Service Parameters to Existing Ones

After determining required parameters that compose the input parameters of the current service task, they must be compared with the existing business object instances belonging to the current process instance. This straightforward comparison is shown in Listing 6.3. This full example additionally differentiates between missing values and missing parameters. To be more precise, a parameter existing but not having a value or a parameter missing altogether, as they are treated differently later on.

```
1  boolean missingSomething = false;
2  Map<String, BusinessObjectType> requiredParameters = operationDescription.getRequiredParameters(); //see previous
       listing
3  for (ProcessToPluginVariableKeyMapping mapping : inputMappingForThisTask) {
4      //iterate over all mapped parameters using the mapping
5      String processVariableKey = mapping.getProcessVariableKey();
6      String pluginVariableKey = mapping.getPluginVariableKey();
7      Object inputVariable = execution.getVariable(processVariableKey);
8      //check if variable has value
9      if (!mapping.isOptional() && inputVariable == null) {
10         missingValues.put(taskId, processVariableKey);
11         missingSomething = true;
12     }
13     //remove parameter from required list
14     requiredParameters.remove(pluginVariableKey) == null;
15 }
16 for (Map.Entry<String, BusinessObjectType> entry : requiredParameters.entrySet()) {
17     //add all still required parameters to missingParameters list
18     missingParameters.put(taskId, new MissingBusinessObjectTypePathPair(entry.getKey(), entry.getValue()));
19             missingSomething = true;
20 }
21 if (missingSomething) {
22     //missing either mapped input or a mapped variable is null, pause execution
23     failedTasks.put(taskId, "Missing Parameters");
24     execution.setVariable(StringConstants.FAILED_TASKS, failedTasks);
25     execution.setVariable(StringConstants.MISSING_VALUES, missingValues);
26     execution.setVariable(StringConstants.MISSING_PARAMETERS, missingParameters);
27     return pauseExecution(taskId, thisServiceTask); //see previous listing
28 } else {
29     //normal execution of service task
30
31 //ommitted actual service task invocation
```

Listing 6.3: Detecting Missing Parameters and Values

This code uses the methods for pausing run-time execution (cf. Listing 6.1) and creation of the required parameter list (cf. 6.2) and is called before every service task execution. Thereby it ensures that all required input parameters exist and respective values are assigned. Otherwise, certain flags are set as instance variables, instructing the user interface on how to handle the resolution of the error.

Note the usage of the *ProcessToPluginVariableKeyMapping* class, which is essentially an implementation of a simple key-value pair. It handles the mapping of the name that a business object instance has in the Activiti BPM engine's internal business object instance *Map* (e.g., "VariableNameXYZ"), to the parameter name that the plug-in operation uses (e.g., "InputVariableXYZ"). This is necessary to allow plug-in developers to choose business object instance names for internal use that are detached from names in a process model It is also used in partial complex type mapping, as depicted in Figure 6.4.

If, based on the list of required parameters, a missing or invalid parameter is found, the user interface showing the process instance is notified of the error, allowing the user to display of an error correction user form.

6.5. Displaying an Error Correction User Form

Requirement REQ-8 states that a BPMS must be able to handle complex business objects. To support this, structural information of any missing parameter, i.e., the corresponding business object type, is sent to the user interface in case an error correction user form has to be displayed. With the help of the structural information contained in the business object types of the business object instances that are to displayed, the form can be generated. The user interface notifies users currently executing a process instance by marking the failed task in red and appending an error message, as shown in Figure 6.5.

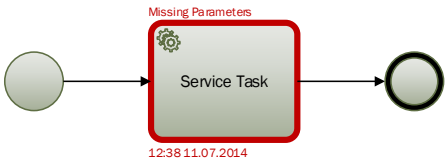

Figure 6.5.: Notification of Users in Case of an Error

A missing parameter notification sent to the user interface for a missing parameter of the *Person* type could contain the information <TaskId: ”123”, ParameterName: ”InputPersonX”, ParameterType: ”Person”>. After a user double-clicks the marked task, shown in Figure 6.5, the user interface requests the structural information of the missing parameters. The user interface can then use the structural information on the complex business object type *Person* to create the form fields for the *Name, Birth Date, Gender,* and *Age* (cf. Figure 6.6). Note that the *Name* business object type itself is complex, resulting in a nested form containing the input fields for the business object types *Firstname* and *Lastname*.

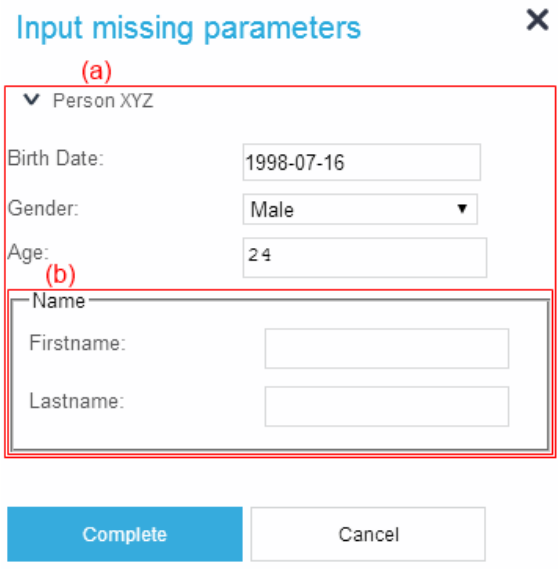

Figure 6.6.: Missing Parameters Form

The similarity to a normal *Person* user form (cf. Figure 3.5) stems from the complete reuse of the form component for both the normal user forms and the error correction forms. The current prototypes of the user form and error correction form components use a combination of the HTML *DisclosurePanel*, *CaptionPanel*, and *Label* elements along a few of standard input fields for different primitive data types, e.g., *TextBox*, *CheckBox*. Additionally, a table is automatically shown for objects of the *ResultSet* type, allowing for the use of service task plug-ins that return result sets from SQL queries. The main rules for determining the form structure are very simple:

- If a business object has more than three direct children, enclose them in a *DisclosurePanel* (cf. Figure 6.6, Marking (a)).
- If a business object has three or less direct children, enclose them in a *CaptionPanel* (cf. Figure 6.6, Marking (b)).
- Any primitive business object is paired with a *Label* holding its name and given an input field corresponding to its base Java type (e.g., *CheckBox* for boolean).
- Business object collections (cf. Section 3.1.4) are enclosed in *DisclosurePanels* and, in case they are editable, given an “Add” button so new elements can be added to the collection.

6.6. Re-invoking Service Tasks with Corrected Parameters

Once required business objects are supplied with correct data values, the Clavii engine must map the input from the form back into the process instance’s business objects. The method that the user interface calls using the new business object instances produced by the form component is shown in Listing 6.4.

```
1 public void writeMissingValues(String processInstanceId, Collection<BusinessObject> businessObjects) {
2     //create container for flattened variables
3     Map<String, Object> variables = new HashMap<>();
4     //retrieve parameter mapping from the process
5     Multimap<String, ProcessToPluginVariableKeyMapping> inputMapping = runtimeService.getVariable(processInstanceId,
          StringConstants.INPUTMAPPING);
6     //use visitor pattern to flatten the different business objects to business object instances
7     for (BusinessObject businessObject : businessObjects) {
8         businessObject.accept(new BusinessObjectFlattenVisitor(variables, businessObject.getName(), inputMapping,
              null));
```

```
9       }
10      //update the inputmapping to reflect new variables
11      runtimeService.setVariable(processInstanceId, StringConstants.INPUTMAPPING, inputMapping);
12      //write flattened variables to process
13      runtimeService.setVariables(processInstanceId, variables);
14  }
```

Listing 6.4: Persisting new Values for Missing Parameters

A listing for *BusinessObjectFlattenVisitor* is provided in Appendix B.1. The *Multimap* [16] object, which is used in Line 5 of Listing 6.4 represents the entire input mapping between business objects and the parameters of service tasks. The *Multimap* data structure supports multiple mappings for each key entry. A key entry is the ID of a service task. In Line 8, the *BusinessObjectFlattenVisitor* class unwraps complex business objects into primitive ones that are understood by the Activiti BPM engine, effectively instantiating them (cf. Section 4.3). Furthermore, it must also be ensured that the input mapping is updated with mappings for the newly created business object instances. Figure A.1 in Appendix A details the execution of the *BusinessObjectFlattenVisitor* class for a service task with an unmapped, but required input parameter of the *Name* business object type. This is performed directly after a user completes the error correction form (cf. Figure 6.6).

Afterwards, the *BusinessObjectFlattenVisitor* class returns the new primitive business objects paired with their respective key entries, along with the mapping to support the usage of the new business object instances by the service task for which they were created. Finally, the mapping and business object instances are persisted in the Activiti BPM engine (cf. Listing 6.4 Lines 11-13). Subsequently a process instance is in a state, in which it can re-run the previously failed service task. This is completely transparent to the Clavii engine, as the difference in comparison to normal service task execution is that the previously created transitions and the receive task are deleted from the process instance's model (cf. Section 6.3).

6.7. Summary

In summary, this section has shown the methods used by the Clavii engine to fulfill Requirements REQ-7, REQ-8, REQ-9, REQ-10 and REQ-11, as contributed by this thesis. These are the requirements addressing the advanced error resolution and rapid process model prototyping features for an advanced BPMS. Most of the work that has to be done to support these features, concerning the implementation and also required processing power at run-time, does not directly stem from the requirements for the features themselves, but actually directly from Requirement REQ-1.

Requirement REQ-1 states that a BPMS should support complex business objects, which increases the complexity and workload for trivial tasks like checking for unmapped or missing required input parameters at run-time. These complex business objects, and their dynamic definition, are discussed in detail in Section 4.

7

State-of-the-Art & Related Work

This section discusses selected BPMS, regarding their data flow concepts. Furthermore, scientific approaches relating to business object persistence, definition or manipulation are discussed.

7.1. State-of-the-Art BPMS

7.1.1. IBM Process Designer

The *IBM Process Designer*, previously known as IBM WebSphere Lombardi Edition, is a BPMS for the Microsoft Windows operating system. It uses a modified Eclipse environment for modeling business processes. The IBM Process Designer allows defining two types of complex business objects: *Structures* and *Complex Structure*

Types [8]. *Structures* are groups of primitive types, whereas *Complex Structure Types* support nesting of previously defined *Complex Structure Types* into new ones. The *Complex Structure Types* are defined in the user interface and are usable in *Java Integration Services*, the IBM Process Designer equivalent of Clavii service task plug-ins.

The approach to dynamic business objects chosen by IBM is different though, as they offer the *com.ibm.websphere.bo* package containing multiple classes extending the Java Service Data Objects (SDO) framework. Business objects designed in IBM Process Designer are compatible with the SDO specification, allowing developers to utilize SDO to access the data contained in the business objects [9]. IBM Process Designer also allows usage of lists and maps of business objects.

The IBM Process Designer does not enforce the correctness by construction principle and allows deploying of clearly dysfunctional process models, which may crash in case of data flow errors. It does, however, assist users in finding errors in process models by showing an error list in a special pane at build-time.

7.1.2. Intalio|bpms

Intalio|bpms is the world's most widely deployed BPMS, offering a free community edition as well as premium enterprise versions. The community edition is split into two components: the *Intalio|bpms designer* and *Intalio|bpms server*. *Intalio|bpms designer* is a plug-in view for the Eclipse IDE, offering a BPMN modeling tool and AJAX web form designer.

Intalio|bpms uses the built in Eclipse XSD design view to allow defining of complex types, as long as these adhere to XML specifications. Furthermore, it uses the Apache ODE BPEL (Business Process Execution Language) engine internally. As BPEL, or more specifically, BPEL4WS (BPEL for Web Services) is a web service orchestration language [33], Intalio|bpms can only support the use of web services for executing Java service tasks. The Intalio|bpms designer can analyze WSDL files and automatically

create service task templates to consume the web services they describe. If the WSDL defines any complex types, these can be used in the process models as well.

Intalio|bpms designer does not enforce correctness by construction and, at least in the community version, does not support process model designers in any way regarding data flow or model correctness. Incorrectly modeled business processes can be deployed to the Intalio|bpms server component. During the deployment the model is checked for the most basic structural problems, such as a missing start event or unreachable tasks.

7.1.3. Bonita BPM

BonitaSoft's *Bonita BPM* is an open source BPMS which is based on Eclipse RCP (Rich Client Platform). Bonita BPM has added support for complex business object types in its most recent version 6.3 [3]. The support comes in form of the Bonita Business Data Model (BDM), which allows defining business objects in POJO code and importing them into Bonita Studio, the modeling component of Bonita BPM. The POJO classes containing the business objects are analyzed using the Java Reflection API, allowing their use in process models. Bonita Studio also allows defining complex business object types utilizing a graphical design tool, and exporting them as XSD or JAR files for use in web services and Java service tasks [3]. This means that by importing such a JAR file into a plug-in implementation project, the complex business object is usable natively.

Bonita BPM has a different approach to persistence than Clavii does, as it serializes each business object type to its own database table, created specifically for the business object type in question. Therefore, instances of types are simply rows in said database table.

7.1.4. AristaFlow BPM Suite

The AristaFlow BPM Suite, a BPMS in development at the AristaFlow GmbH, is based on the principles developed as part of the ADEPT2 project[21]. These include correctness by construction and plug & play plug-in activities as well as ad-hoc flexibility.

AristaFlow offers support for complex types by defining User Defined Types (UDTs). UDTs are, however, fundamentally different from our approach to complex business object types, as their structure is effectively transparent to the Process Template Editor, the AristaFlow process modeling tool [22]. Therefore, mapping or access to parts of the UDTs is not possible without User Defined Functions (UDFs), which can convert a UDT or parts of a UDT to a format understandable by an activity implementation [22].

7.2. Related Work

In [22] the viability of advanced data flow and structure concepts in the context of the AristaFlow BPM Suite is investigated (cf. Section 7.1.4). Particularly, it states that the introduction of inheritance to a business object type framework is not necessary, because the same effect can be achieved by copying the fields of an existing data type to a new one [22]. Even though this might be true while only viewing the resulting data types, inheritance is important in the context of executing service tasks expecting certain data types as parameters (cf. Section 5.3). The thesis also proposes a data flow analysis algorithm for complex data types and collections thereof, which could be introduced into the Clavii engine as an extension to the correctness by run-time error resolution.

In [39] a set of best practices for persisting business objects are introduced, without using technologies like JPA. It demonstrates patterns for object transactions, change managers, and other lower-level problems, most of which the Clavii engine solves using a combination of Spring transaction management servlet filters and the Hibernate ORM.

The patent in [20], claims an object definition framework that works “based on the assumption that the definition of anything and everything will evolve over time” [20]. The basic idea behind the invention is to offer inter-enterprise business objects along with a definition language to define and evolve them. Among the 237 claims the patent contains are a few interesting concepts such as storing predefined form parts and rules for business objects in the business objects themselves, allowing for better-looking and more dynamic generic forms. Also among the claims is a system to categorize attributes of business objects into groups concerning a certain area of knowledge.

[30] introduces the concept of object-aware business processes, i.e., business processes where the state of the involved business objects and constraints placed on those objects dictate the actions that business process participants can execute at any given time. For this the PHILharmonicFlows framework, the proposed prototype for an object-aware BPMS, needs to persist the structures of highly complex business objects with relations, attributes and constraints, such as cardinalities and permissions. Additionally, the instances of these business objects must contain state information and markings at run-time as the state of the individual business object instances determines the possible user interactions with the process instance.

In [34] rules for structuring data flow are proposed to ensure that there are no inconsistencies such as lost updates at run-time. These rules, which are meant to support process model designers when designing the data flow of a process model, also ensure that there are no errors such as unmapped input parameters, which could cause exceptions in service tasks. The Ph.D. thesis also offers algorithms for efficiently analyzing the correctness of data flows. An important concept described in the thesis is that of *sync edges*. *Sync edges* ensure that data access from tasks on parallel execution branches in a process model can be synchronized, thereby ensuring correct reading and writing order and minimizing lost updates.

8

Conclusion

Currently available BPMS support complex business object types to a certain degree (cf. Section 7.1), though none achieve the simplicity to non-technical end-users that our approach offers. Also, none of the examined BPMS allow programming of plug-ins without the use of API packages except for in the context of web services. Additionally, the concepts of inheritance and variable length arguments with simple mapping set the Clavii engine apart from most others.

The correctness by process run-time error resolution features (cf. Section 6.1) of the Clavii BPM Cloud are a unique approach to iterative business process development not available in state-of-the-art BPMS, the value of which has still to be proven through testing of the implementation prototype.

The approach chosen to allow serialization and deserialization of complex business objects (cf. Section 4.1) is limiting compared to more general serialization techniques

based on the Java Reflection API, but is flexible enough to support any typical business object relevant to a process model. A huge advantage of “limiting” flexibility, compared to other approaches (e.g., UDT in the AristaFlow BPM Suite) is the usability of definable types in the user interface. This allows for mapping and manipulation of individual parts of complex business objects.

The Clavii engine is accompanied by the corresponding Google Web Toolkit (GWT) user interface and custom updateable process view[28] manipulation library, detailed in [19] and [25] respectively. These three components form the cloud-based Clavii BPM Cloud. All in all the Clavii BPM Cloud is an easy to use BPMS, compared to other, more technically-oriented BPMS. This ease of use is enabled by the extensions to the Activiti BPM engine, proposed in this thesis and bundled into the Clavii engine, as well as a cloud-based web interface concept, developed with non-technical users in mind.

A

Figures

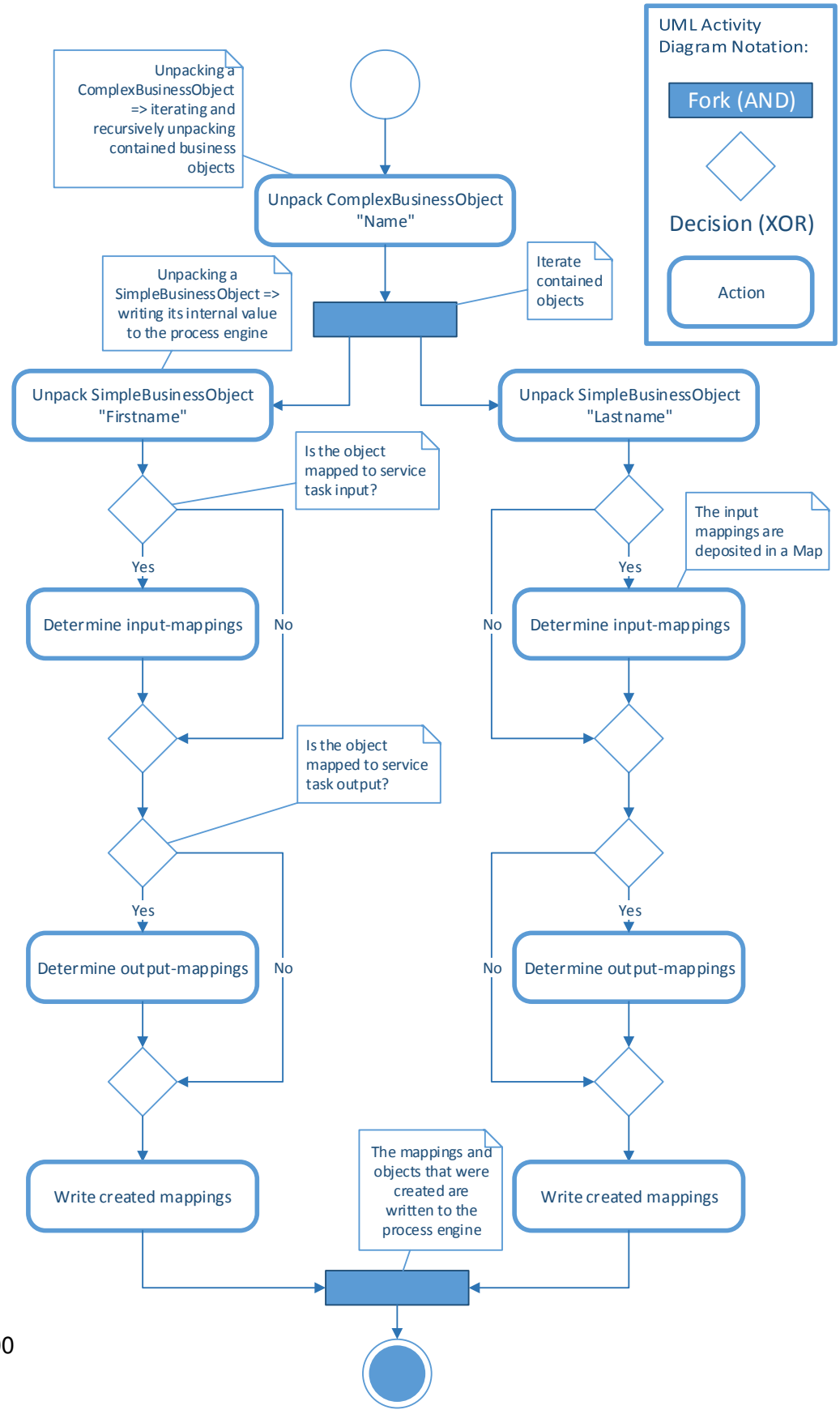

Figure A.1.: Instantiation and Mapping of Newly Created Business Objects

B

Sources

```
1  public class BusinessObjectFlattenVisitor implements BusinessObjectVisitor {
2      private final Map<String, Object> variables;
3      private final String key;
4
5      private final Multimap<String, ProcessToPluginVariableKeyMapping> inputMapping;
6      private final Multimap<String, ProcessToPluginVariableKeyMapping> outputMapping;
7
8      public BusinessObjectFlattenVisitor(Map<String, Object> variables, String key, Multimap<String,
           ProcessToPluginVariableKeyMapping> inputMapping, Multimap<String, ProcessToPluginVariableKeyMapping>
           outputMapping) {
9          this.variables = variables;
10         this.inputMapping = inputMapping;
11         this.outputMapping = outputMapping;
12         this.key = key;
13     }
14
15     public BusinessObjectFlattenVisitor(String key, Map<String, Object> variables) {
16         this(variables, key, null, null);
17     }
18
19     @Override
20     public void visit(SimpleBusinessObject simpleBusinessObject) {
21         writeMappings(simpleBusinessObject);
22
23         variables.put(key, simpleBusinessObject.get());
```

```
24      }
25
26      @Override
27      public void visit(EnumBusinessObject enumBusinessObject) {
28          writeMappings(enumBusinessObject);
29
30          variables.put(key, enumBusinessObject.get());
31      }
32
33      @Override
34      public void visit(ComplexBusinessObject complexBusinessObject) {
35          for (Map.Entry<String, BusinessObject> entry : complexBusinessObject.getContainedInstances().entrySet()) {
36              String nextVarKey = key.isEmpty() ? entry.getKey() : key + '.' + entry.getKey();
37              entry.getValue().accept(new BusinessObjectFlattenVisitor(variables, nextVarKey, inputMapping,
                        outputMapping));
38          }
39      }
40
41      @Override
42      public void visit(MapBusinessObject mapBusinessObject) {
43          writeMappings(mapBusinessObject);
44
45          Map<String, Object> containedInstances = new HashMap<>();
46
47          for (Map.Entry<String, BusinessObject> entry : mapBusinessObject.getContainedInstances().entrySet()) {
48              if (entry.getValue() instanceof ComplexBusinessObject) {
49                  HashMap<String, Object> complexContainer = new HashMap<>();
50                  entry.getValue().accept(new BusinessObjectFlattenVisitor("", complexContainer));
51                  containedInstances.put(entry.getKey(), complexContainer);
52              } else {
53                  entry.getValue().accept(new BusinessObjectFlattenVisitor(entry.getKey(), containedInstances));
54              }
55
56          }
57
58          variables.put(mapBusinessObject.getName(), containedInstances);
59      }
60
61      @Override
62      public void visit(ListBusinessObject listBusinessObject) {
63          writeMappings(listBusinessObject);
64
65          Map<String, Object> tempContainedInstancesMap = new HashMap<>();
66
67          List<BusinessObject> containedInstances = listBusinessObject.getContainedInstances();
68          for (int i = 0; i < containedInstances.size(); i++) {
69              BusinessObject current = containedInstances.get(i);
70              if (current instanceof ComplexBusinessObject) {
71                  HashMap<String, Object> complexContainer = new HashMap<>();
72                  current.accept(new BusinessObjectFlattenVisitor("", complexContainer));
73                  tempContainedInstancesMap.put(String.valueOf(i), complexContainer);
74              } else {
75                  current.accept(new BusinessObjectFlattenVisitor(String.valueOf(i), tempContainedInstancesMap));
76              }
77
78          }
79
80          variables.put(listBusinessObject.getName(), new LinkedList<>(tempContainedInstancesMap.values()));
81      }
82
```

```
83      //corrects the mappings that were used to construct this visitor instance
84      //this ensures, that the Caller class knows which process variable keys
85      // fit with which plugin variable keys
86      private void writeMappings(AttachableBusinessObject attachableBusinessObject) {
87          if (inputMapping != null) {
88              for (Map.Entry<String, PathDescription> inputEntry : attachableBusinessObject.getConnectedInputNodes().
                    entrySet()) {
89                  inputMapping.put(inputEntry.getKey(), new ProcessToPluginVariableKeyMapping(key, inputEntry.getValue
                        ().getPath(), inputEntry.getValue().isOptional()));
90              }
91          }
92          if (outputMapping != null) {
93              for (Map.Entry<String, PathDescription> outputEntry : attachableBusinessObject.getConnectedOutputNodes()
                    .entrySet()) {
94                  outputMapping.put(outputEntry.getKey(), new ProcessToPluginVariableKeyMapping(key, outputEntry.
                        getValue().getPath(), outputEntry.getValue().isOptional()));
95              }
96          }
97      }
98  }
```

Listing B.1: BusinessObjectFlattenVisitor.java

```
1   public class PluginCallDispatcherImpl implements PluginCallDispatcher {
2       //BEGIN INTEGRATED CALLER
3       @Override
4       public Object call(IntegratedPlugin integratedPlugin, String methodName, Map<String, Object> parameters)
5               throws ClassNotFoundException, NoSuchMethodException, IllegalAccessException, InvocationTargetException,
                        InstantiationException {
6           String className = integratedPlugin.getLocation();
7           Class<?> clazz = Class.forName(className);
8           if (clazz == null) {
9               return null;
10          }
11
12          return MethodUtils.invokeMethod(clazz.getConstructor().newInstance(), methodName, parameters);
13      }
14
15      @Override
16      public Object call(OSGIPlugin osgiPlugin, String methodName, Map<String, Object> parameters)
17              throws NoSuchMethodException, IllegalAccessException, InvocationTargetException, InstantiationException,
                        BundleException {
18          Class<?> clazz = loadClass(osgiPlugin);
19          if (clazz == null) {
20              return null;
21          }
22
23          return MethodUtils.invokeMethod(clazz.getConstructor().newInstance(), methodName, parameters);
24      }
25      //BEGIN OSGI CALLER
26      private Felix framework;
27
28      private Bundle getFramework() throws BundleException {
29          if (framework == null) {
30              //init OSGI framework if it is non existent
31              Map<String, String> configMap = new HashMap<>();
32              configMap.put(Constants.FRAMEWORK_STORAGE_CLEAN, "onFirstInit");
33              framework = new Felix(configMap);
34              framework.start();
```

```
35          }
36          return framework;
37      }
38
39      private Class<?> loadClass(OSGIPlugin plugin) throws BundleException {
40          //get the jar file containing the code for the plugin
41          byte[] bundleJar = plugin.getBundleJar();
42          //the classname in the bundle that contains the code
43          String bundleExecutionClass = plugin.getLocation();
44          BundleContext context = getFramework().getBundleContext();
45          //check if bundle is already installed
46          Bundle provider = context.getBundle(String.valueOf(plugin.getId()));
47          //if not, load the bundle from the byte[]
48          if (provider == null) {
49              provider = context.installBundle(String.valueOf(plugin.getId()),
50                                               new ByteArrayInputStream(bundleJar));
51          }
52          //start bundle
53          provider.start();
54          //get reference to execution class
55          ServiceReference reference = context.getServiceReference(bundleExecutionClass);
56          //get instance of class
57          Object service = context.getService(reference);
58          return service.getClass();
59      }
60      //BEGIN WEBSERVICE CALLER
61      private JaxWsDynamicClientFactory jaxWsDynamicClientFactory;
62      private final Map<String, Client> clientCache = new ConcurrentHashMap<>();
63
64      @Override
65      public Object call(WebServicePlugin plugin, String operationName, Map<String, Object> parameters) {
66          //get url to wsdl from plugin
67          String wsdlLocation = plugin.getLocation();
68          if (jaxWsDynamicClientFactory == null) {
69              jaxWsDynamicClientFactory = JaxWsDynamicClientFactory.newInstance();
70          }
71          //check if client was already created
72          //(clients are cached as wsdl introspection is costly)
73          Client client = clientCache.get(wsdlLocation);
74          if (client == null) {
75              //creates a dynamic "client" for a webservice using reflection on the wsdl
76              client = jaxWsDynamicClientFactory.createClient(wsdlLocation);
77              //cache the client
78              clientCache.put(wsdlLocation, client);
79          }
80          try {
81              //call the webservice with the Map as sole parameter and return the first return value
82              return client.invoke(operationName, parameters)[0];
83          } catch (Exception e) {
84          throw new RuntimeException(e.getMessage());
85          }
86      }
87  }
```

Listing B.2: PluginCallDispatcherImpl.java

```
1   public class TimerTrigger implements Serializable, Runnable {
2       private final Map<Long, Long> idMapping = new HashMap<>();
3       private transient Map<Long, ScheduledFuture> runningTimers;
```

```
4       private transient Method receiver;
5
6       @Override
7       public void run() {
8           for (Map.Entry<Long, Long> entry : idMapping.entrySet()) {
9               Map<String, Object> parameters = new HashMap<>();
10              parameters.put("Interval", entry.getValue());
11              registerIntervalTimer(entry.getKey(), parameters);
12          }
13      }
14
15      public void registerIntervalTimer(final long id, Map<String, Object> parameters) {
16          Long interval = (Long) parameters.get("Interval");
17          idMapping.put(id, interval);
18          if (runningTimers == null) {
19              runningTimers = new HashMap<>();
20          }
21          runningTimers.put(id, Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(new Runnable() {
22              @Override
23              public void run() {
24                  try {
25                      Map<String, Object> output = new HashMap<>();
26                      output.put("Trigger Time", new Date());
27                      receiver.invoke(null, id, output);
28                  } catch (IllegalAccessException | InvocationTargetException e) {
29                      e.printStackTrace();
30                  }
31              }
32          }, interval, interval, TimeUnit.SECONDS));
33      }
34
35      public void deregisterIntervalTimer(long id) {
36          if (idMapping.remove(id) != null) {
37              runningTimers.get(id).cancel(true);
38          }
39      }
```

Listing B.3: TimerTrigger.java

```
1   <integratedPlugin>
2       <location>de.clavii.plugins.PersonManipulator</location>
3       <name>Person Manipulator</name>
4       <description>Manipulates Person Objects</description>
5       <author>Kevin Andrews</author>
6       <activityDescriptions>
7           <activityDescription>
8               <methodName>mixPeople</methodName>
9               <name>Mix People</name>
10              <description>Mixes two people</description>
11              <inputParameters>
12                  <inputParameter>
13                      <type>Person</type>
14                      <name>Person 1</name>
15                      <optional>false</optional>
16                  </inputParameter>
17                  <inputParameter>
18                      <type>Person</type>
19                      <name>Person 2</name>
20                      <optional>false</optional>
```

```
                </inputParameter>
            </inputParameters>
            <outputParameters>
                <outputParameter>
                    <type>Person</type>
                    <name>Combined Person</name>
                    <optional>false</optional>
                </outputParameter>
            </outputParameters>
        </activityDescription>
        <activityDescription>
            <methodName>makeOlder</methodName>
            <name>Make older</name>
            <description>Makes a Person older</description>
            <inputParameters>
                <inputParameter>
                    <type>Person</type>
                    <name>Person X</name>
                    <optional>false</optional>
                </inputParameter>
            </inputParameters>
            <outputParameters>
                <outputParameter>
                    <type>Person</type>
                    <name>Aged Person</name>
                    <optional>false</optional>
                </outputParameter>
            </outputParameters>
        </activityDescription>
        <activityDescription>
            <methodName>sumAges</methodName>
            <name>Sum Ages</name>
            <description>Sum up ages of People</description>
            <inputParameters>
                <inputParameter>
                    <type>Person List</type>
                    <name>People</name>
                    <optional>false</optional>
                </inputParameter>
            </inputParameters>
            <outputParameters>
                <outputParameter>
                    <type>Age</type>
                    <name>Age Sum</name>
                    <optional>false</optional>
                </outputParameter>
            </outputParameters>
        </activityDescription>
    </activityDescriptions>
</integratedPlugin>
```

Listing B.4: PersonManipulator.xml

```
<trigger>
    <location>de.clavii.triggers.Timer</location>
    <name>Timer</name>
    <description>Timer Trigger</description>
    <author>Kevin Andrews</author>
    <registerMethodName>registerIntervalTimer</registerMethodName>
```

```
7       <deregisterMethodName>deregisterIntervalTimer</deregisterMethodName>
8       <inputParameters>
9           <inputParameter>
10              <type>Timespan</type>
11              <name>Interval</name>
12              <optional>false</optional>
13          </inputParameter>
14      </inputParameters>
15      <outputParameters>
16          <outputParameter>
17              <type>Date</type>
18              <name>Trigger Time</name>
19              <optional>false</optional>
20          </outputParameter>
21      </outputParameters>
22  </trigger>
```

Listing B.5: TimerTrigger.xml

Bibliography

[1] Activiti Overview, http://activiti.org/components.html, last visited on 2014/07/14

[2] AristaFlow Overview, http://www.aristaflow.com/AristaFlow_BPM-Suite.html, last visited on 2014/07/14

[3] Bonita BPM 6.3 Manual, Chapter: Data Handling, http://documentation.bonitasoft.com/product-bos-sp/data-handling, last visited on 2014/07/13

[4] CGLIB Knowledge Base, https://github.com/cglib/cglib/wiki, last visited on 2014/07/14

[5] Executor Framework Tutorial, http://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html, last visited on 2014/07/14

[6] ForEach Guide, http://docs.oracle.com/javase/1.5.0/docs/guide/language/foreach.html, last visited on 2014/07/14

[7] GWT JRE Emulation Reference, http://www.gwtproject.org/doc/latest/RefJreEmulation.html, last visited on 2014/07/14

[8] IBM Business Process Manager V8.5 Information Center, http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.main.doc/ic-homepage-bpm.html, last visited on 2014/07/13

[9] IBM Business Process Manager V8.5 Information Center: Business Objects Programming, http://pic.dhe.ibm.com/infocenter/dmndhelp/

v8r5m0/topic/com.ibm.wbpm.main.doc/topics/cbo_intro.html, last visited on 2014/07/13

[10] Intalio Overview, http://www.intalio.com/products/bpms/overview/, last visited on 2014/07/14

[11] Iterator JavaDoc, http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html, last visited on 2014/07/14

[12] Java Manual: Thread Pools, http://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html, last visited on 2014/07/14

[13] Javassist Web Site, http://www.javassist.org, last visited on 2014/07/14

[14] JAX-WS Web Site, https://jax-ws.java.net/, last visited on 2014/07/14

[15] JAXB Web Site, https://jaxb.java.net/, last visited on 2014/07/14

[16] MultiMap in Google Collections JavaDoc, http://google-collections.googlecode.com/svn/trunk/javadoc/com/google/common/collect/Multimap.html, last visited on 2014/07/14

[17] The Reflection API, http://docs.oracle.com/javase/tutorial/reflect/index.html, last visited on 2014/07/14

[18] VarArgs Guide, http://docs.oracle.com/javase/1.5.0/docs/guide/language/varArgs.html, last visited on 2014/07/14

[19] Bueringer, S.: Development of a Cloud Platform for Business Process Administration, Modeling and Execution. Master's Thesis, Ulm University (2014)

[20] Burke, M., Solar, R.: Building Business Objects and Business Software Applications Using Dynamic Object Definitions of Ingrediential Objects (2004), http://www.google.com/patents/US6789252

[21] Dadam, P., Reichert, M., Rinderle-Ma, S., Goeser, K., Kreher, U., Jurisch, M.: Von ADEPT zur AristaFlow BPM Suite - Eine Vision wird Realität: "Correctness by Construction" und Flexible, Robuste Ausführung von Unternehmensprozessen. EMISA Forum 29 (2009)

[22] Forschner, A.: Fortschrittliche Datenflusskonzepte für Flexible Prozessmodelle. Master's Thesis, Ulm University (2009)

[23] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education (1994)

[24] Josefsson, S.: The Base16, Base32, and Base64 Data Encodings (2006)

[25] Kammerer, K.: Enabling Personalized Business Process Modeling: The Clavii BPM Platform. Master's Thesis, Ulm University (2014)

[26] Kolb, J., Hübner, P., Reichert, M.: Automatically Generating and Updating User Interface Components in Process-Aware Information Systems. In: Proc 10th Int'l Conf. on Cooperative Information Systems (CoopIS'12). pp. 444–454 (2012)

[27] Kolb, J., Hübner, P., Reichert, M.: Model-Driven User Interface Generation and Adaptation in Process-Aware Information Systems. UIB 2012-04, Technical Report, Ulm University (2012)

[28] Kolb, J., Kammerer, K., Reichert, M.: Updatable Process Views for User-centered Adaption of Large Process Models. In: Proc 10th Int'l Conference on Service Oriented Computing (ICSOC'12). pp. 484–498. No. 7636 in LNCS, Springer, Shanghai, China (2012)

[29] Kolb, J., Reichert, M.: Data Flow Abstractions and Adaptations through Updatable Process Views. In: Proc 28th Symposium on Applied Computing (SAC'13), 10th Enterprise Engineering Track (EE'13). pp. 1447–1453. ACM Press, Coimbra, Portugal (2013)

[30] Künzle, V.: Object-Aware Process Management. Ph.D. Thesis, Ulm University (2013)

[31] Micheler, F.: Konzeption, Implementierung und Integration einer Komponente für die Erstellung Intelligenter Formulare. Master's thesis, Ulm University (June 2009)

[32] Object Management Group (OMG): Unified Modeling Language Specification (OMG UML) Version 2.4.1. Tech. rep.

[33] Peltz, C.: Web Services Orchestration and Choreography. Computer 36(10) (2003)

[34] Reichert, M.: Dynamische Ablaufänderungen in Workflow-Management-Systemen. Ph.D. thesis, University Ulm (2000)

[35] Rummler, G.A., Brache, A.P.: Improving Performance: How to Manage the White Space on the Organization Chart. John Wiley & Sons (2012)

[36] Sousa, K., Mendonça, H., Vanderdonckt, J., Rogier, E., Vandermeulen, J.: User Interface Derivation from Business Processes: A Model-Driven Approach for Organizational Engineering. In: Proc. ACM Symposium on Applied Computing (SAC'08). pp. 553–560 (2008)

[37] Tacy, A., Hanson, R., Essington, J., Tökke, A.: GWT in Action. Manning Publications (2013)

[38] Van Emden, E., Moonen, L.: Java Quality Assurance by Detecting Code Smells (2002)

[39] Yoder, J.W., Johnson, R.E., Wilson, Q.D.: Connecting Business Objects to Relational Databases. Urbana 51, 61801 (2005)

Name: Kevin Andrews Matrikelnummer: 671626

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Kevin Andrews