FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Ein Ansatz für Prozess-zentrierte Software basierend auf Geschäftsprozess-Modellen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Ralph Hoch BSc
Matrikelnummer 0405156

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Hermann Kaindl
Mitwirkung: Univ.Ass. Dr. Roman Popp

Wien, 29. September 2015

Ralph Hoch	Hermann Kaindl

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

An Approach for Process-centric Software based on Business Process Models

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Ralph Hoch BSc
Registration Number 0405156

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dr. Hermann Kaindl
Assistance: Univ.Ass. Dr. Roman Popp

Vienna, 29th September, 2015

Ralph Hoch

Hermann Kaindl

Erklärung zur Verfassung der Arbeit

Ralph Hoch BSc
Neulerchenfelder Strasse 87/33, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. September 2015

Ralph Hoch

Acknowledgements

First, I would like to thank my parents who supported me my whole life, morally and financially, and always encouraged me to use my talents in every way possible. Also, moving to Vienna without their help, and the use of their car, would have been so much harder. Many trips with them to IKEA provided my first accommodation. Although I am stubborn and had countless arguments with them, they never lost their patience with me.

I also would like to thank my girlfriend, who helps me with words and deeds. I am pretty sure my parents are glad that somebody took pity and is willing to deal with my stubbornness on a daily basis. As she is also a fellow engineer I had many interesting discussions with her, that helped me during my studies. I am looking forward to many more great years together.

Last but not least, I would like to thank my advisor Prof. Hermann Kaindl and assistance advisor Dr. Roman Popp. During our conversations, and sometimes disputes about little details, I always learned something new or found a new perspective to look at a topic. They always were open minded for new ideas and supported me as I worked on my thesis.

Kurzfassung

Firmen verwenden moderne Software-Applikationen häufig, um Aufgaben des täglichen Geschäftsalltags zu organisieren und unterstützend abzuarbeiten. Mitarbeitern wird dadurch die Möglichkeit gegeben miteinander Computer-unterstützt an Aufgaben bzw. Aufgabenketten zu arbeiten. Weiters ermöglicht es die Kollaboration mit externen Partnern auf eine automatisierte Weise. Unternehmensarchitekturen bilden solche Organisationsstrukturen ab und definieren, wie eine Firma ihre geschäftlichen Tätigkeiten ausführt. In solch einer Unternehmensarchitektur sind Geschäftsprozessmodelle ein integraler Bestandteil, da sie spezifizieren, wie Aufgaben organisiert sind, wer verantwortlich ist und mit welchen externen Partner zusammengearbeitet wird. Sie beschreiben also den Ablauf von komplexen Aufgabenabfolgen.

Traditionelle Software-Lösungen spiegeln diese Unternehmensarchitektur oft nicht wieder, beziehungsweise implementieren dessen Logik oft direkt im Programmcode. Adaptierungen der Software, zum Beispiel auf Grund veränderter Anforderungen, sind daher nicht ohne größeren Aufwand möglich. Oftmals ist es notwendig Teile oder die ganze Software neu zu implementieren. Flexibilität und die korrekte Abbildung der Unternehmensarchitekturen in der Software sind wichtige Faktoren, da sie helfen die Wartungskosten gering zu halten und ein schnelles Reagieren auf veränderte Anforderungen erlaubt.

Eine getrennte Spezifikation von Aufgabenfolgen abseits des Programmcodes, in der Form von Geschäftsprozessmodellen, die von der Software Applikation verarbeitet werden können, ist ein Weg um solch eine Flexibilität zu erreichen. BPMN 2.0 erlaubt das Spezifizieren von Geschäftsprozessmodellen und ermöglicht es diese in die Software einzubinden. Mit diesem Ansatz verwendet die Software Geschäftsprozesse um die Logik der Applikation zu steuern.

Allerdings liegt der Fokus von BPMN 2.0 auf der Spezifikation einer Ablauffolge von Aufgaben. Nicht behandelt wird, welche zusätzlichen Informationen ein Benutzer benötigt um die Aufgaben zu erledigen. Es sind zur Ausführungszeit nur Informationen zu der aktuellen Aufgabe verfügbar und der Kontext, in welchem dieser Geschäftsprozess ausgeführt wird, steht nicht zur Verfügung. Dadurch ergibt sich eine taskgetriebene Software-Applikation, bei deren Anwendung ein Benutzer nur einen Teil des gesamten Informationsgehaltes zur Verfügung hat. Das spiegelt sich vor allem auch in der Benutzeroberfläche der Applikation wieder.

Wir schlagen daher eine Software-Architektur vor, die es erlaubt BPMN 2.0 Modelle auszuführen und Kontextinformationen mittels Business-Artefakten auszudrücken. Die

daraus resultierende Software verwendet diese Kontextinformation in Kombination mit den Informationen der Aufgabe um daraus eine einheitliche, alle notwendigen Informationen enthaltende, Benutzeroberfläche zu erzeugen. Zur Repräsentation des Kontext verwenden wir Modelle von Business-Artefakten, welche dann in der Benutzeroberfläche dargestellt werden. All diese Teile zusammen ermöglichen es eine prozess-zentrierte Applikation zu entwickeln.

Abstract

Modern software applications are often used in companies to organize tasks and support the business. These applications enable employees to collaborate with each other as well as with external partners in a computer assisted way. Companies employ Enterprise Architectures, defined for a given business, which provide information about how they operate. An essential part of Enterprise Architectures are Business Processes. They are a means to specify how specific tasks are organized, who is responsible and with which partners the company collaborates.

Traditional software solutions often do not reflect the Enterprise Architecture or, at most, directly implement it in the program code rather than using models. This makes it difficult to adapt the software as requirements change over time. Rebuilding parts or the entire software application may become necessary in such a case. Flexibility of software applications and their alignment with an Enterprise Architecture are an important factor to keep maintenance costs low and to allow fast adaptation to changed requirements.

Having a separate specification of workflows in the form of Business Process Models suitable for software applications, is one option to allow flexibility. BPMN 2.0 provides such a specification of Business Processes and enables embedding them into software applications. An application that uses this approach utilizes Business Processes to control the software and its *Business Logic*.

However, BPMN 2.0 does not account for additional information to be presented to the user. It only provides the means to execute a sequence of tasks and at the time of execution only information about the currently active task is available. This often results in task-driven software applications, where users only have limited information about the process and its environment. The context in which the Business Process is enacted, is not available and thus also not visible in the User Interface.

We propose a software architecture including a BPMN 2.0 engine and a model of Business Artefacts for aligning the business, its supporting software and the User Interface. The resulting framework shows how Business Processes defined in BPMN can be enriched with additional context information provided by Business Artefacts. During execution of such processes, a coherent Graphical User Interface is presented to the user where not only information about the currently active task is displayed, but also information about the context in which it is executed. We use models of Business Artefacts to present this information and visualize it. Utilizing these parts, we move towards a process-centric application.

Contents

CHAPTER 1

Introduction

Software is used in a wide field of applications and commonly supports users during their everyday tasks. Companies use it to allow collaboration of users in employed workflows. Furthermore, the software facilitates the exchange of information between users as well as other companies. Developing such software applications is a complex task, in particular because features or specifications may change over time. A common case is that users have new requirements to the software and that the employed workflows change. This often involves rebuilding parts of the application and updating it across all users. To ensure that the software operates satisfactory, these adaptations have to be tackled. Providing flexibility of software applications is an important factor to keep maintenance costs low and allow fast adaptation to changed requirements.

Having a separate specification of workflows, in form of Business Processes (BPs), that is suitable for software applications, is one option to allow flexibility. This enables the application to work with a dynamic specification of how certain tasks are connected with each other and by doing so helps to provide a customizable software application. Furthermore, it makes the software reusable in different application areas and by users with various roles and/or privileges. An application that follows this approach uses BPs as the driving force for the software and its *Business Logic*. This is the foundation of a process-centric approach, which is often used in *Enterprise Software Systems* [74].

As process-centric software should support users during their tasks, an adequate presentation of the necessary information is imperative. This means that the User Interface (UI) which is presented to the user should contain all information necessary to fulfill the current task of a BP. Such a UI makes use of information concerning the BP as well as Business Artefacts.

This diploma thesis proposes a software framework capable of handling BPs along with a configurable context view based on Business Artefacts. This provides a custom UI for each BP and helps to move towards a process-centric approach [35, 64].

1.1 Motivation

Modern software applications often support users in handling BPs that are employed by a company. The logic of these processes is often implemented directly in the source code of the application. This makes the software rather inflexible and also makes it difficult to extend the application to support new BPs.

The Business Process Model Notation (BPMN) 2.0 [62] standard allows the specification of BPs in a common way. Its purpose is to have an explicit specification of the process so that the orchestration of the software components of this process is decoupled from the business logic of the software application. An *execution engine* is used to automatically execute BPs and to control the applications behavior. Various BPMN execution engines exist to support this approach [32].

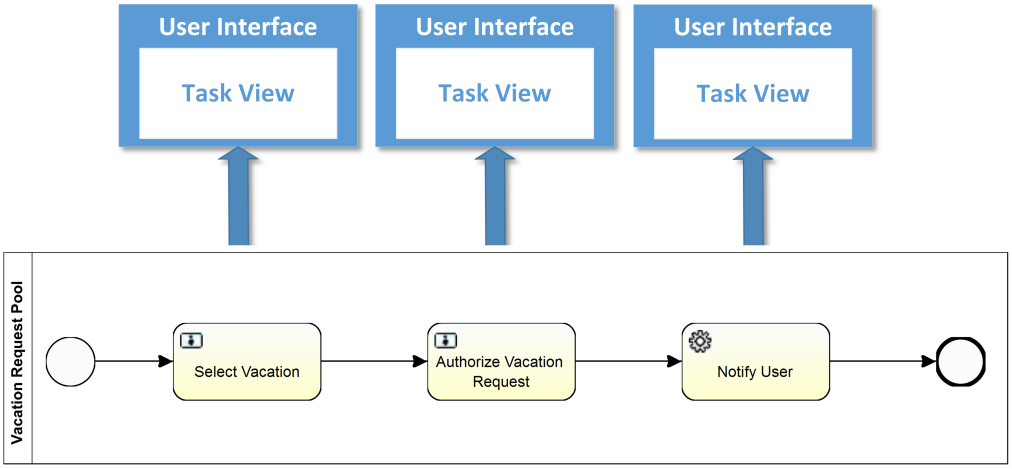

Figure 1.1: Behavior of BPMN framework user interface with isolated Task Views

These frameworks often place an emphasis on single tasks rather than the process as a whole. This task-centric approach is often not sufficient for users since there is no additional context information of the process available. Only the current task is presented to the user and all other information is hidden or not easily accessible. This approach leads to a UI with isolated tasks popping up, as illustrated in Figure 1.1. Furthermore, these frameworks often use an "inbox"-like system that is similar to modern email applications, where new tasks pop up and can be processed.

To illustrate this problem, the following example can be considered: Let us assume that there is a BP that manages vacation-related topics of a company. In this process, a task *approve vacation request* is passed to a manager to be authorized. The task itself only contains the information that is directly relevant for the task, namely the time frame of the vacation request and who issued the request. This information is usually insufficient for the manager as more context is needed to decide whether this request can be approved or not. For the manager it is of interest if other employees from the same department are on vacation at the same time, if the employee is involved in any projects

with an upcoming deadline, how much vacation days the employee has left, etc. In a task-centric approach, all this information has to be gathered manually.

Overall, BPMN-driven frameworks support dynamic processes through an explicitly defined specification, but lack the inclusion of context information, which is essential in Enterprise organizations. The user is limited to the information that a task provides (task-centric) and does not have any additional information of the overall process.

1.2 Problem Statement

This diploma thesis proposes a software framework capable of handling business processes specified in BPMN along with a configurable context view for them. This provides a custom UI, including all necessary information, for each BP and helps to move towards a process-centric approach.

The main questions that arise are:

1. What sort of features are required to provide a satisfactory process-centric software application?

2. How can a BP execution engine be integrated into a software application? What kind of prerequisites does this impose on the software architecture?

3. How is it possible to align the UI based on Business Artefacts with the BP?

1.3 Aim of Work & Methodological Approach

The envisioned advantage of this approach is that a user has all necessary information for the process available and visible at once and thus can handle processes similarly to common Enterprise applications. Figure 1.2 shows a simple mock-up of how such a framework could look like. The framework provides the means to use a specification for additional context information that is used in combination with the employed BP to provide a coherent UI.

In addition, the software framework helps to keep the user in a given BP as long as there are tasks available rather than switching with each new task between processes and thus different context information. To support a manual context switch between processes (for example if a process with higher priority is available) the user is always able to choose a different task.

To achieve the expected result, first a comprehensive analysis of the features for such a framework was performed. These features were then compared with the demand of a process-centric approach for software based on BPs. During this step, extensive research on both literature as well as other frameworks has been performed.

After the features had been specified, an architectural concept of the framework was conducted. In this architectural concept, a description of the interfaces for the components as well as their interaction pattern was given. Furthermore, a specification

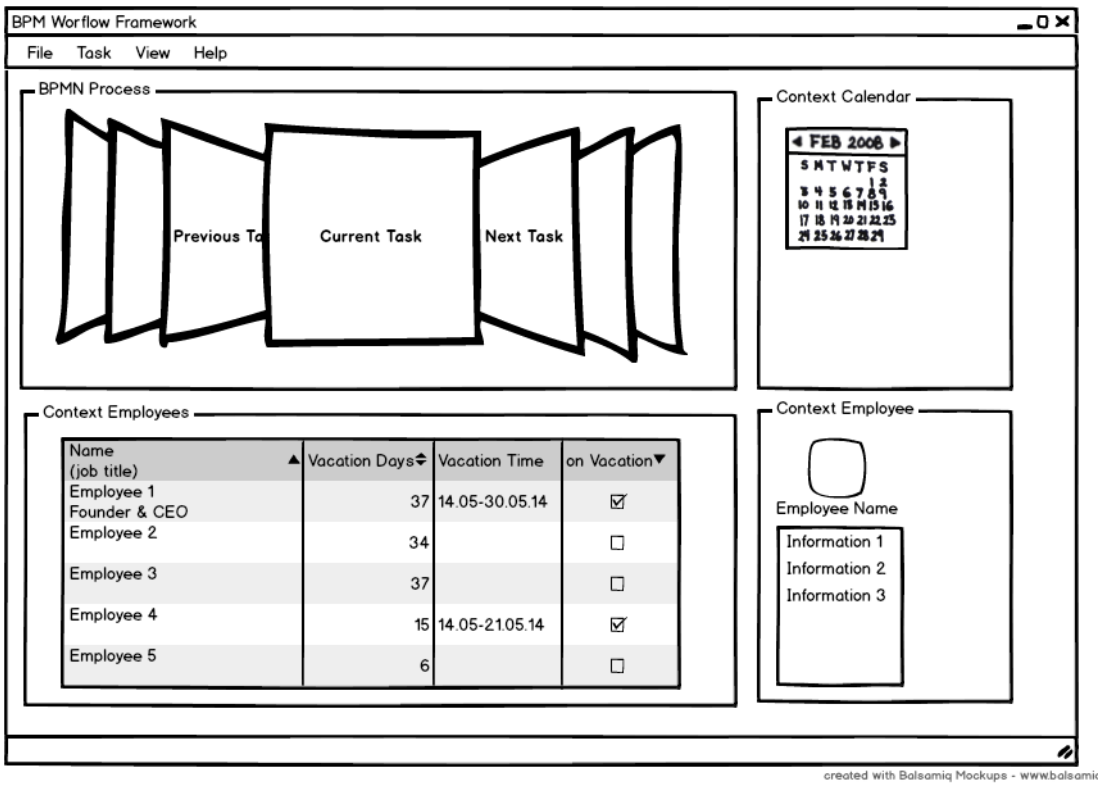

Figure 1.2: Mock-up of resulting process-centric Framework

for the configurable UI and how the context information is represented was developed. The context is presented through Business Artefacts.

According to the architectural concept, a prototype was implemented to show the feasibility of the proposed framework. Development and implementation of the prototype has been carried out in an iterative manner.

1.4 Structure of Work

Chapter 2 provides background information on the basics, such as BPMN, Business Artefacts or Design Patterns as well as technologies that are necessary to understand the remaining parts of the thesis. In addition, a running example, which is used throughout the thesis, is introduced.

Chapter 3 gives an overview of work related to this thesis as well as a state-of-the-art analysis of Business Process Management Systems (BPMSs). It relates the thesis to software design approaches and frameworks that utilize models of BPs.

Chapter 4 lays out the general architecture of the feasibility prototype along with an in-depth description of selected essential parts. It explains in detail how the UI is constructed and how the architecture relates to the Business Artefacts.

In Chapter 5 the handling of the context configuration is described in detail. The chapter shows how the additional context configuration is specified and how the UI is aligned to it.

Chapter 6 shows the result of the feasibility prototype and the technologies used. Furthermore, it discusses limitations of the prototype and provides pointers to possible future work.

The thesis ends with Chapter 7, where a summary is given.

CHAPTER 2

Background

This chapter gives an overview of the basics that are necessary to fully understand the remaining parts of the thesis. First, a running example is introduced, which is used throughout the thesis to showcase shortcomings of traditional approaches as well as improvements of the proposed solution. In addition, concepts and technologies concerning process-oriented software along with related software patterns are presented.

2.1 Running Example

Companies employ structured sequences of tasks through which they cope with recurring assignments. These structured sequences are specified by Business Process Models (BPMs) and provide a specification of how they are handled. For the running example, a BP is taken into account that deals with *vacation-related* topics of a company. The process basically handles a simple *vacation request* of an employee, which needs approval by a supervisor. This is somewhat simplified as there might be other roles involved as well, but for this case it is sufficient.

Figrue 2.1 shows how such a *vacation-request* process can be specified. Basically, there are several different roles involved and information on the current request is exchanged between them. The data-flow is not explicitly visualized in the figure, but it can be assumed that all assets, that are created by a person or task, are directly passed to the successor task.

The process is initialized by an employee and first a selection of a date for her vacation request has to be entered. This is handled by the *Vacation Request* task where the employee is able to enter her request. The upper part of the figure marked by the caption *Employee* visualizes this initialization. After the *vacation-request* has been specified, it is passed to another role, a supervisor, where the request is reviewed. The task *Authorize Vacation Request* handles this assignment and as a result the request is granted or denied. This is shown in the middle part, labeled *Supervisor*, of the figure.

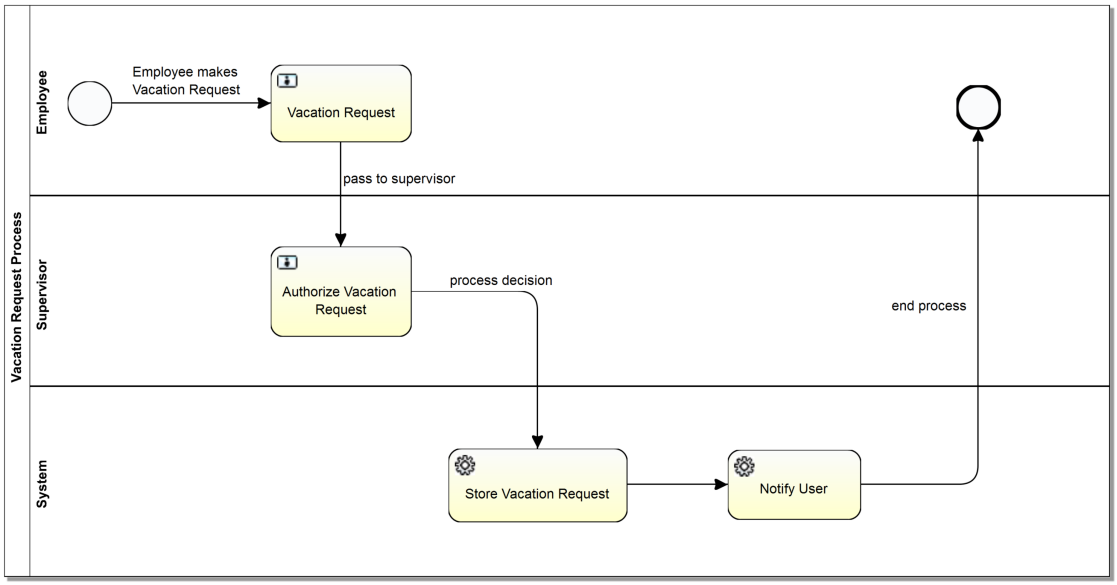

Figure 2.1: Running example: vacation request BP specified in BPMN

The result of the *Authorize Vacation Request* is then passed to yet another role, the *System*, where the *vacation-request* along with the result is stored within the software by an automated task *Store Vacation Request*. As a final step, the employee is informed, for example via email or within the software application itself, about the result of her *vacation-request*. After this notification, the BP is fully processed and terminates.

This simple BP already involves various roles and different kinds of tasks. There are tasks that require user interaction, such as the *Authorize Vacation Request*, as well as automated tasks, such as *Notify Employee*. Each role is separated by a so-called "lane" and all tasks placed within this "lane" are processed by the respective role. More information on the semantics of the figure is given in Section 2.2.3.

As mentioned above, the data exchanged between tasks is not explicitly visualized. However, there are several data objects involved in this simple BP. The employee, for example, is represented as a data object or, more precisely, an artefact in the business. The same applies to the vacation request, which is represented in the software through a data object. These Business Artefacts are exchanged in between the tasks and are used to provide the necessary information to process the tasks. The problem is that this information is often insufficient, as there is often more context necessary. Let us revisit the task *Authorize Vacation Request*, where the supervisor decides whether the request is granted or denied based on the *vacation-request*. This means, that only information on the *employee* and the time-frame of the request is available to the supervisor. Strictly speaking, this is sufficient to complete the task, however, it may not in the grand scheme of the company. For example, it may be mandatory to know if other employees are on vacation at the same time, if there are deadlines on projects that the requesting employee is involved with, or if the employee has enough vacation days left. This sort of

information is available in the overall business, and most of the time accessible via software applications, but not directly specified in the BP itself. So, meaning the information has to be gathered manually. Furthermore, different roles may have different privileges on what they are allowed to access or may simply need other information to process a task.

In essence, a BPM in BPMN only specifies how tasks are related with each other and what kind of information is exchanged, but not what kind of additional context information may be necessary to process these tasks.

2.2 Process-centric software based on Business Process Models

Software applications are subject to constant change as new technologies are introduced, more effective ways are discovered, or technologies are applied in new ways. Furthermore, users of software may introduce new requirements that have to be fulfilled. That is, the software evolves over time [51]. Introducing flexibility into software applications helps to reduce costs and makes it easier to adapt to changes [9, 33, 70].

Typically, software applications have been developed for a specific assignment that they have to fulfill. However, with this approach the software is rather inflexible and adaptations require changes of the software as the application logic was often hard-coded directly into the source code. A separate specification of this logic that drives the application would improve the flexibility. In companies, software is used to support and automate business cases, which are handled via BPs. Furthermore, it should help employees to work on their tasks. This means, that the application logic is preset by BPs that are in place in the company. Davenport [19] describes a BP as follows:

> "In definitional terms, a process is simply a structured, measured set of activities designed to produce a specified output for a particular customer or market. It implies a strong emphasis on how work is done within an organization, in contrast to a product focus's emphasis on what."

Utilizing models of these BPs as the application logic of software applications, helps to move towards a *process-centric* software approach. In a *process-centric* software, the focus is on the BP and on the flow of activities rather than individual elements such as Business Artefacts. To make use of BPs, it is necessary to have a processable specification of them. These specifications are called BPMs. There are various languages available to specify BPMs [56].

A separate specification of BPs also facilitates further possibilities not directly related to the resulting software. It allows, for example, an easier use of optimization or verification techniques. As this is a broad field of applications, we refer to further literature [34, 77, 86].

2.2.1 Services and process-centric software

Since one important part of BPMSs is that BPMs are automatically executable via a software application, it is necessary to explain how this can be accomplished. BPs consist of several tasks, where some of them should be automatically executed without human interaction. These tasks are often referred to as *service tasks*. They can be realized with various technologies and, in essence, they are just software components that are invoked. However, with the rise of Web-based applications, an option is to use Web Services. In essence, Web Services are self-contained, self-describing components that can be invoked and provide some sort of functionality [28]. From a more technical point of view Web Services are described as entities that support interoperable machine-to-machine interaction over a network. They provide a machine-processable interface and communicate through well-defined messages [30]. The interface is usually provided via the Web Service Definition Language (WSDL) [17] and is commonly supported by a variety of tools.

Services and Web Services, in particular, have been used as the basic building blocks for a software architecture paradigm for quite some time. Service Oriented Architecture (SOA) describes this paradigm, where services are used in larger scale applications. The basic idea is that a user can find services in a repository through a discovery mechanism and then invoke the service. All this is possible during runtime of the application, which makes it very flexible, as services can be exchanged with ease [21]. Building software based on this principle allows creating software applications consisting of several services instead of a single monolithic software component. Furthermore, it enables software developers to create distributed systems that communicate over a network. This is especially useful for companies, as they can provide services, that allow business partners to easily collaborate with them. Nevertheless, there remain some unsolved problems with this approach. First of all, it has to be specified somewhere which services are supposed to be integrated. In addition, it has to be specified in which order they are invoked. This specification would basically create a simple BP through choreography and orchestration of services [66].

One option to solve this problem is to combine BPMs with SOA [39]. Tasks that are supposed to be automatically executed are implemented as services and the specification of the process via the BPM provides the connection to these services. Furthermore, it allows the execution engine of the BPM to connect to external resources such as services provided by business partners. Considering the running example from Section 2.1, the *service tasks*, which are executed by the *system* role, should be implemented as services and dynamically called during runtime. This enables developers to use the advantages of both technologies and combine them to build more dynamic applications [7, 22]. Utilizing SOA as a mechanism for workflow systems is not limited to a specific domain but can be applied in a wide field of application areas [78]. It is important to note that SOA and BPMS are not equivalent, but rather use each other to achieve a higher goal [38].

2.2.2 Using Business Artefacts in Business Processes

Giving a clear definition of what a Business Artefact involved in a BP is, may differ depending on the environment. In BPs, Business Artefacts are often considered to be the objects that are used within a specific Business Domain and provide some sort of information. Nigam et al. [58] describe Business Artefacts as:

> "Any business, no matter what physical goods or services it produces, relies on business records. It needs to record details of what it produces in terms of concrete information. Business artifacts are a mechanism to record this information in units that are concrete, identifiable, self-describing, and indivisible. We developed the concept of artifacts, or semantic objects, in the context of a technique for constructing formal yet intuitive operational descriptions of a business."

Business Artefacts provide the means to store information about objects that are used in the Business Domain. More generally speaking, Business Artefacts are the objects that are used in the Business Domain. The BPs use them as the basic data-objects on which the activities operate. Considering the services described in Section 2.2.1, the Business Artefacts are the inputs that they consume and also the result, which they produce. In a business, most of the activities will operate on Business Artefacts. However, this definition does not account for objects, that may be involved during a BP, but are not directly manipulated or created. If we consider the running example, the employee that makes the vacation-request would not be considered a Business Artefact. For the purpose of this thesis a Business Artefact is any object that is involved within a Business Domain.

How Business Artefacts are represented and where they are defined or specified, depends on the used technologies. There exist several approaches, each with its own advantages. For example, in a typical software application a database schema alongside with classes in object-oriented programming languages can be considered as Business Artefacts for the software application. However, a more declarative specification is preferable as it allows separating the specification from the implementation. There are model-driven approaches available that enable software developers to automatically generate parts of software applications from these models. Another approach is to use a language, that also has formal semantics such as Web Ontology Language (OWL) (see Section 2.3).

Considering BPM, the *data objects* involved are the representation of Business Artefacts in the process specification. They typically reference a structure, for example a class or a schema definition, and should be automatically processable. During process execution these references are resolved and the actual data is gathered from the corresponding storage facility where the Business Artefacts are managed.

Typically, BPs are designed with the overall workflow in mind. The sequence of activities performed is the main focal point and Business Artefacts are there to support them. However, there exist other approaches to designing BPMs. One option is to put the focal point on the Business Artefacts themselves rather than the activities that

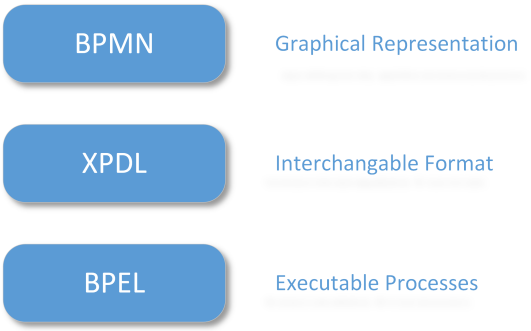

Figure 2.2: Technologies involved in a BPMS (pre BPMN 2.0): based on [43]

operate on them. In this case, the life-cycles of Business Artefacts are used to design the processes around them and the evolution of Business Artefacts controls the process. Basically, the BP shows how Business Artefacts are changed over time by activities. Challenges of artefact-centric BPMs have been discussed by Hull [37].

2.2.3 BPMN

The BPMN [62] standard is one possibility to model BPs. It provides a graphical notation that is easily accessible by business users who have long-term experience with BPs modeled with flow-chart techniques. Up to version 1.2 of BPMN, only a graphical representation of the BP was available and the XML Process Definition Language (XPDL) [84] standard was used to provide an interchangeable format. These earlier versions of BPMN lacked a possibility to directly execute the defined BP. Furthermore, BPMN up to version 1.2 did not provide enough semantics for developers to efficiently work with the designed BPs. Thus a mapping to the Business Process Execution Language (BPEL) [59] standard was specified, which allowed for execution of BPs. Figure 2.2 gives an overview of the technologies involved in BPMS. Please note that other technologies are available as well, as stated by Ko et al. [43].

This gap between the BP design done by business users and process implementation done by software developers, proved to be troublesome as continuous transformations from one technology to another are error-prone. In addition, the transformation from BPMN to BPEL is not trivial as they have fundamental differences in their approach. BPEL uses a block-structured approach to construct BPs as opposed to the directed-graph approach used by BPMN [63]. With the introduction of BPMN 2.0, many of these shortcomings have been dealt with and the gap between process design and process implementation has been targeted [16]. For this purpose, a technique that maps the visualization to the execution format has been specified. To accomplish this, a meta-model has been introduced into the standard and a more comprehensive description of the building blocks is given. Machine processable normative files are also defined via Extensible Markup Language (XML)-based technologies and the execution semantic is now defined. This further helps developers to provide an exact specification for automatically executable

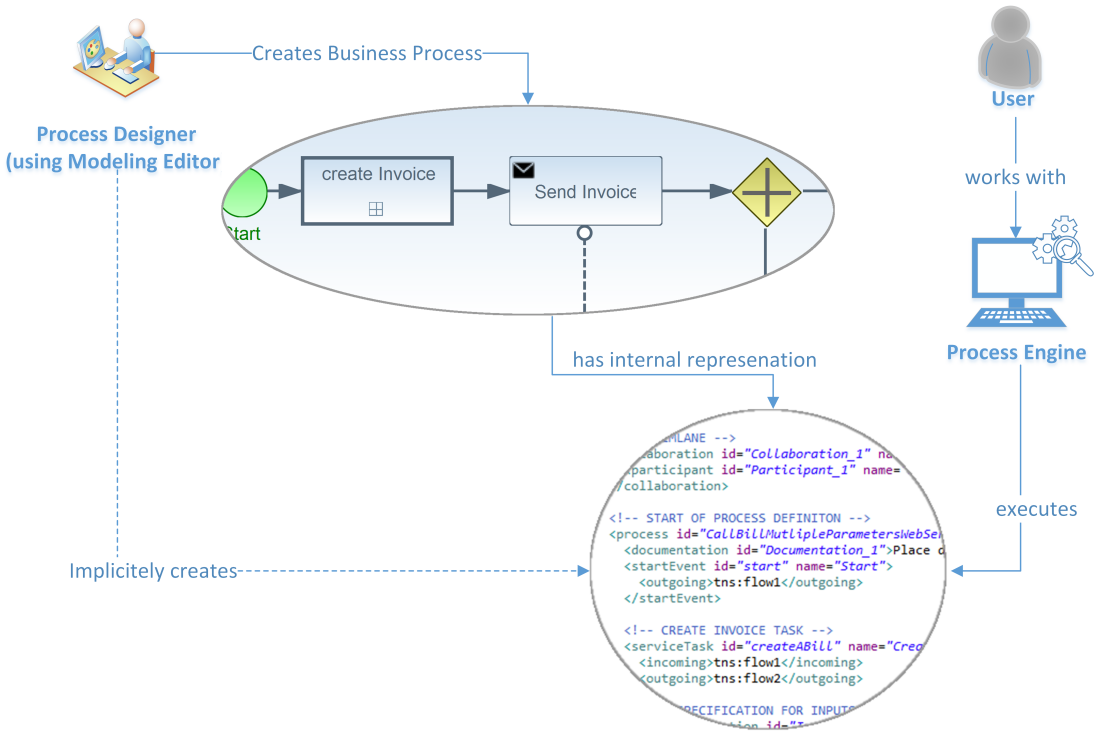

Figure 2.3: BP creation and execution with BPMN 2.0

BPs. A more comprehensive look at BPMN is provided by Allweyer [4].

The graphical representation is still based on flow-chart techniques and is easily accessible for business users. For each graphical representation, an XML-based specification exists with well-defined semantics. The idea is that a process designer does not have to worry about details hidden in the XML representation during the BP creation as there are tools which support her. There are various tools available that support BPMN, but no reference implementation is available. In the standard, there are four types of conformance specified that an application has to fulfill to be fully BPMN-compliant (Process Modeling Conformance, Process Execution Conformance, BPEL Process Execution Conformance, and Choreography Modeling Conformance). Applications often only support a subset of these and focus on Process Modeling Conformance and Process Execution Conformance.

Overall, BPMN 2.0 provides the means to graphically design BPs, exchange them between users and automatically execute them. BPMN 2.0 is used in this thesis, because it provides a single model for both, the design of BPs as well as their execution. Figure 2.3 illustrates how BPMN 2.0 is used. The *process designer* first creates a BP via the graphical notation of BPMN. This step is usually supported by a *modeling editor*, that allows working directly with the graphical elements so that the *process designer* does not have to work with the more complex XML representation. Internally, an XML specification of this graphical representation is used and, in most cases, created automatically by the *modeling*

editor. This XML specification is the standardized format, which allows interoperability as well as the automatic execution of the BP. To perform such an execution, an *execution engine* is necessary. It creates an instance of a specific BP and performs the defined steps, e.g., runs tasks in the defined order, evaluates decision gateways on the current data instances, etc..

This re-enactment of BPs can be accessed by users to perform tasks, which are assigned to them. This is a somewhat simplified description of using BPMN as there are often more challenges, such as managing, deploying or monitoring of BPs, involved.

2.2.3.1 Basic Elements

The designed BPMN aimed at a simple and understandable mechanism for constructing BPMs. To accomplish this, the graphical aspects of the notation have been separated into categories, where each of the categories provides a small set of notations. This makes it easy to identify the basic elements and, consequently, the diagrams. In this section, a short introduction is given to some of the basic and most used elements. Basically, the elements of BPMN are organized in five categories and each of these categories holds several elements. These elements themselves have specialized sub-elements with additional properties. In general, a BPMN diagram defines a process consisting of activities that are connected with each other or with data elements.

1. Flow Objects
    a) Events
    influence the flow of BPMs. They are triggered and have an impact. Typical elements are the *start* and *end* event of a process.
    b) Activities
    perform some sort of work in a process. They can either be a sub-process or a *task*. They are visualized through rounded rectangles.
    c) Gateways
    control how the sequence flow in BPs is executed (branching, forking, etc.).
2. Data
    a) Data Objects
    are objects that provide information about entities.
    b) Data Inputs
    are data objects that are specifically used as an input for activities.
    c) Data Outputs
    are data objects that are specifically used as an output for activities.
    d) Data Stores
    are storage facilities for data objects.
3. Connecting Objects

a) Sequence Flows
define the order in which activities are executed.

b) Message Flows
in contrast to the sequence flow, the *message flow* shows how two partners collaborate. They are commonly used to show how two processes in a diagram communicate.

c) Associations
allow connecting additional information such as annotations with the process (artifacts)

d) Data Associations
bind data elements to activities and show what kind of input or output a specific activity provides.

4. Swimlanes

a) Pools
visualize single processes or, more generally, a participant in the diagram. Pools can have several lanes to organize processes.

b) Lanes
help to organize and categorize activities within processes.

5. Artifacts

a) Group
allows to group elements and, by doing so, to categorize them.

b) Text Annotation
provides additional information for the reader of a diagram.

As some of the elements are often used throughout the thesis, a more detailed description of them is necessary. This applies especially to *Activities*, *Gateways*, *Data Objects* and *Lanes*. *Activities* are an essential part of BPs, as they represent units (or *Tasks*) that perform some kind of work. As there are many options on how such an *Activity* can be performed, a differentiation is mandatory. For example, *Activities* may require human interaction (*UserTask*) to be completed or are completely autonomic (*ServiceTask*). To differentiate between these kinds of *Activities*, sub-elements with specific properties have been introduced. These properties allow a designer to automatically assign services to tasks or to provide additional information on what kind of interaction is necessary. Closely related to the type of *Tasks* are the *Data Objects*. These basically are the objects that are unconditionally inevitable for *Tasks* to perform their work. Similar to the *Activities* described above, *Data Objects* can have specialized properties based on their usage. *Data Objects* can be used as collections or as single objects, and often they are specifically used as an input or output of a *Task*.

As sequence flows between *Activities* can be quite complex, additional elements are necessary that control them. This is where *Gateways* come into play and provide a

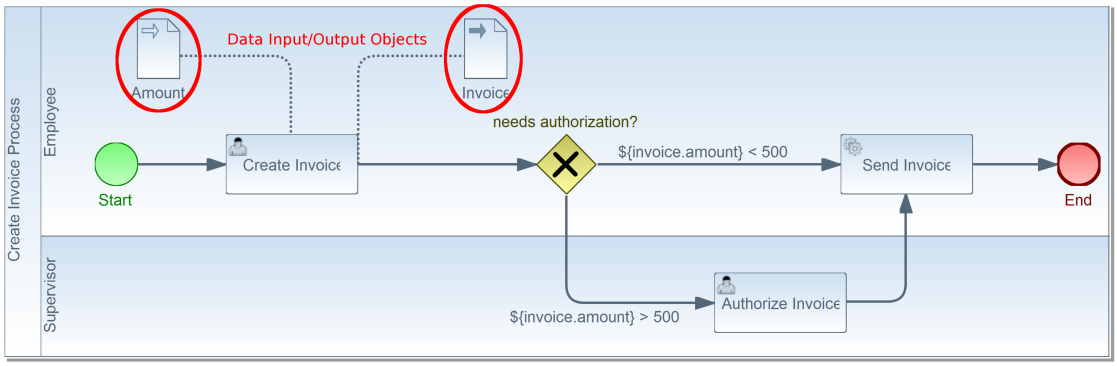

Figure 2.4: Example process with basic elements of BPMN

mechanism to control the flow. They provide several control structures such as parallelism or branching via decision nodes.

The definition of a *Lane* is not directly specified in the standard and is open to interpretation. The modeler can decide on her own how she wants to use *Lanes* as a mean to organize and categorize *Activities*. *Lanes* are always used within a *Process Pool* and often illustrate that different *Roles* are involved in a process.

Figure 2.4 shows the basic elements and how they can be used to define a process. In this particular process called *Create Invoice Process* are two roles involved, which are visualized through the lanes *Employee* and *Supervisor*. The *start-event* is triggered by the execution engine and the *sequence flow* indicates that the *UserTask* called *Create Invoice* is executed first. The *UserTask* is identified by the small user icon in the rounded rectangular box. The *Create Invoice* task is performed first and can only be completed with additional user interaction (*UserTask*). It takes one input *Value* and creates one output *Invoice*. Afterwards, the flow reaches an exclusive *Gateway*, where a decision is made. If the *Value* of the *Invoice* is larger than 500, a separate task *Authorize Invoice* has to be performed. This task is executed by another role *Supervisor*. As a final step, the *Invoice* is processed in a *ServiceTask* called *Send Invoice*. This task is automatically performed without human interaction as indicated by the gear icon in the upper left corner of the task-box.

This basic example indicates how BPMN diagrams are read and what kind of semantics the elements involved have. The *Data Objects* are often not visualized in the diagram to keep it simple and manageable. However, they are of high importance for the tasks and a specification in the XML process specification is necessary.

As it is possible to have several process definitions in a single diagram, there needs to be an option to specify their interaction. This is accomplished through messages that are exchanged between processes or participants. This is especially useful if communication with external services or processes hosted by another company is required. Furthermore, BPMN provides the means to deal with transactions, error handling and other functions. These functions are omitted here as they are not essential for the running example of the thesis. For further information we refer the reader to Shapiro et al. [75] or the BPMN standard [62].

2.2.3.2 Extension mechanism

BPMN provides a mechanism to import external elements that are not defined in the standard. With this technique in place, it is, for example, possible to import data structures of objects used in the BP. Considering Business Artefacts of a specific business domain, this enables process designers to reference them directly during the process creation. In principle, BPMN allows imports of all kind of types and only a couple of types are pre-defined in the standard. XML-based structures are preferred as they are easy to process and integrate nicely with the, also XML-based, BPMN specification. Each import has its own namespace and can be referenced by it.

Listing 2.1 shows an example of such an import statement, where an XML-based data structure is imported into the BP. None XML-based types are allowed as well, for example to establish relationships between elements, but this depends on the used BPMN vendor. In general, non-standard *importTypes* are subject to specific vendors and may only be supported by them.

Listing 2.1: Import statement of BPMN

```
<import importType="http://www.w3.org/2001/XMLSchema"
    location="InvoiceDataDefinition.xsd"
    namespace="http://ict.tuwien.ac.at/InvoiceData"/>
```

In addition, BPMN provides means to extend elements, such as *UserTasks*, by custom properties or attributes. These *ExtensionDefinitions* are bound to the BPMN model definition and can then be used throughout the model. This technique can, for example, be used to provide additional properties, such as rendering information on attributes, for *UserTasks*. Often these extensions are used by specific tools to provide additional information for their execution engine. Thus they are commonly applied to deal with technical limitations.

2.2.3.3 Web Services in BPMN

As mentioned in Section 2.2.1, Web Services are commonly used as a technology to establish communication, exchange information, or simply to provide some sort of functionality. As BPs are often used not only to specify the internal aspects of a process, but also how and with whom a company collaborates, the integration of Web Services is relevant. Utilizing Web Services also helps to make the BP more flexible as they can be automatically invoked and changed during runtime. BPMN has an option built in that enables the use of Web Services as an implementation technology for tasks such as *Service Tasks*.

With the import facility of BPMN, WSDL specifications can be introduced into BPs. The WSDL *importType* is already pre-defined in the standard. Listing 2.2 shows how such a specification is imported. With this import, all elements of the corresponding WSDL file are available in BPMN. As each of these imports has its own namespace, the namespace has to be included when referencing elements.

Listing 2.2: Import statement for WSDL defined Web Service

```
<import importType="http://schemas.xmlsoap.org/wsdl/"
    location="http://localhost/services/InvoiceService?wsdl"
    namespace="http://ict.tuwien.ac.at/InvoiceService"/>
```

Listing 2.3 shows what kinds of elements are necessary to correlate BPMN with WSDL Web Service. *InvoiceServiceNS* is the namespace of the imported WSDL specification and all elements that reference the Web Service have to include this namespace. First, a couple of *itemDefinitions* are given, which introduce local elements, that reference a structure from the Web Service, into BPMN. These *itemDefinitions* are then used to create *messages*, so that communication with the Web Service is possible. Finally, the *interface* is defined with its *operations* to relate specific messages with specific operations of the WSDL specification.

Listing 2.3: Correlate BPMN elements with WSDL elements

```
<!-- WEB SERVICE CONNECTION DEFINITION -->
<itemDefinition id="createInvoiceAmountInputSoap"
    structureRef="InvoiceServiceNS:createInvoiceOperation"/>
<itemDefinition id="createInvoiceInvoiceOutputSoap"
    structureRef="InvoiceServiceNS:createInvoiceOperationResponse"/>
<itemDefinition id="sendInvoiceInvoiceInputSoap"
    structureRef="InvoiceServiceNS:sendInvoiceOperation"/>
<itemDefinition id="sendInvoiceInvoiceOutputSoap"
    structureRef="InvoiceServiceNS:sendInvoiceOperationResponse"/>

<message id="createInvoiceRequestMessage"
    itemRef="tns:createInvoiceAmountInputSoap"/>
<message id="createInvoiceResponseMessage"
    itemRef="tns:createInvoiceInvoiceOutputSoap"/>
<message id="sendInvoiceRequestMessage"
    itemRef="tns:sendInvoiceAmountInputSoap"/>
<message id="sendInvoiceResponseMessage"
    itemRef="tns:sendInvoiceInvoiceOutputSoap"/>

<interface id="Interface_1"
    implementationRef="InvoiceServiceNS:InvoiceService" name="Invoice
    Service">
  <operation id="createInvoice"
      implementationRef="InvoiceServiceNS:createInvoiceOperation"
      name="Create a Invoice">
    <inMessageRef>tns:createInvoiceRequestMessage</inMessageRef>
    <outMessageRef>tns:createInvoiceResponseMessage</outMessageRef>
  </operation>
  <operation id="sendInvoice">...</operation>
</interface>
```

This *interface*, or more precisely the *operation* definition can be used as an implementation reference in *Tasks*. If the implementation of a *Task* is a Web Service, this is indicated by the attribute *implementation* and has the value *##WebService*. Listing 2.4 shows how this is accomplished. The *ServiceTask Create Invoice* has a Web Service

implementation based on the previously defined operation *tns:createInvoice* (compare Listing 2.3). Its *dataInput* and *dataOutput* are bound to the messages that have been defined.

Listing 2.4: Usage of Web Service operations in BPMN tasks

```
<!-- CREATE INVOICE TASK -->
<serviceTask id="createInvoice" name="Create Invoice"
    implementation="##WebService" operationRef="tns:createInvoice">
  <incoming>tns:flow1</incoming>
  <outgoing>tns:flow2</outgoing>
  <!-- SPECIFICATION FOR INPUTS AND OUTPUTS -->
  <ioSpecification id="InputOutputSpecification_1">
    <dataInput id="dataInputOfCreateInvoiceServiceTask"
        itemSubjectRef="tns:createInvoiceAmountInputSoap" name="Amount"/>
    <dataOutput id="dataOutputOfCreateInvoiceServiceTask"
        itemSubjectRef="tns:createInvoiceInvoiceOutputSoap" name="Invoice"/>
    <inputSet id="InputSet_1">
      <dataInputRefs>dataInputOfCreateInvoiceServiceTask</dataInputRefs>
    </inputSet>
    <outputSet id="OutputSet_1">
      <dataOutputRefs>dataOutputOfCreateInvoiceServiceTask</dataOutputRefs>
    </outputSet>
</ioSpecification>
```

Figure 2.5 shows in a more abstract way how WSDL elements are related to the elements in the graphical notation in BPMN. In essence, the messages in WSDL, which define what kind of information is passed to the Web Service, are closely related to the *DataObjects* in BPMN. As shown in Listing 2.3 the WSDL structures can not be used directly. They have to be mapped to BPMN *ItemDefinitions* so that they can be used to define the messages that are necessary to accomplish communication. The *operations* of WSDL are related to the *Tasks* in BPMN. Similar to the *DataObjects*, there is also a mapping necessary. This is done by the *interface* and *operation* specification in Listing 2.3. With this in place the Web Service can be called from a *ServiceTask* as shown in Listing 2.4.

These are basically all steps that are necessary to invoke a WSDL-defined Web Service. BPMN-compliant tools have to support these steps and must be able to interpret and invoke Web Services. Currently, BPMN 2.0 supports WSDL 2.0 and thus, also *RESTful* [53] Web Services can be invoked. In addition, the BPMN 2.0 standard also imposes an additional constraint on *ServiceTasks*, limiting the amount of input parameters to one [32]. This means, that an extra wrapping of parameters to one single, in case of Web Services, message is necessary.

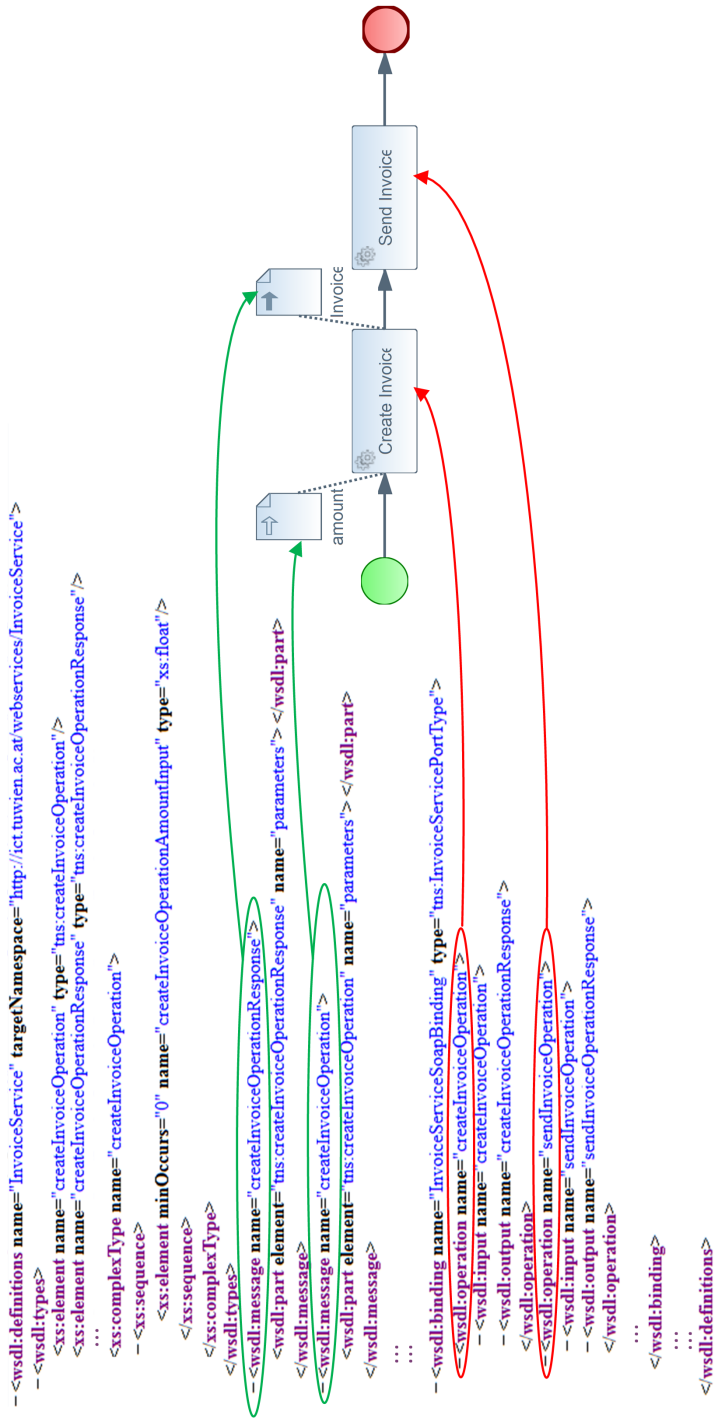

Figure 2.5: Relation of WSDL elements with BPMN elements

2.2.3.4 Tool Support

At the time of this thesis there are a number of vendors available that provide BPMN-compliant tools or frameworks. Some of them focus only on the modeling aspects, while others provide a full BPMS, including the automatic execution of processes, based on the standard.[1] Modeling tools focus on the design of BPMN-compliant process models and provide an editor that allows the process designer to create process models. A comparison of modeling tools has been done by Geiger et al. [26] who evaluated how these tools serialize graphical BPMN models into their corresponding XML format and what kind of inconsistencies arise. They present a set of serialization constraints, which the modeling tools are evaluated against. They show that, although tools claim to be BPMN-compliant, not all constraints can be satisfied and that interoperability is still an issue [27].

As this thesis focuses on process-centric software that uses BPs as the driving force for the application, direct execution of these BPs is mandatory. There are also several vendors, open-source as well as proprietary, available, which provide frameworks that fulfill this requirement.[2] Although there are tools available from larger companies such as IBM[3] or Oracle[4], an open-source solution was preferred for this thesis, as many of the proprietary frameworks only provide a full suite and do not provide an easy access point to the standalone *execution engine*. A comparison of the execution engines Activiti[5], jBPM[6] and Camunda[7], is performed by Geiger et al. [25], who show that process models are often vendor specific and not easily exchangeable between them.

Many of these vendors have their focus on the creation, management and execution of BPMN-compliant BPs as opposed to the UI of the resulting application. During the execution, tasks are assigned to users and they can fulfill their assignments with simple UIs based on forms. These forms are used to gather the input of the user for specific *UserTasks*.

However, they neglect the need for additional context information that is often required to efficiently process a task. Considering the running example of Section 2.1, the *UserTask Authorize Vacation Request* would only provide a form with an option to grant or deny the request. The only other information available would be the requester and the date of the request. However, as stated above, this is insufficient as there are other conditions, such as project deadlines where the requester is involved or vacation dates of other employees, that have to be considered.

In this thesis, the Activiti framework is used. The decision was made based on an extension that Activiti provides, which allows specifying additional *formProperties*

[1] Comparison of Business Process Modeling Notation tools: `https://en.wikipedia.org/wiki/Comparison_of_Business_Process_Modeling_Notation_tools`

[2] List of BPMN 2.0 engines: `https://en.wikipedia.org/wiki/List_of_BPMN_2.0_engines`

[3] `http://www-03.ibm.com/software/products/en/business-process-manager-family`

[4] `http://www.oracle.com/us/technologies/bpm/suite/overview/index.html`

[5] `http://www.activiti.org/`

[6] `http://www.jbpm.org/`

[7] `http://www.camunda.org`

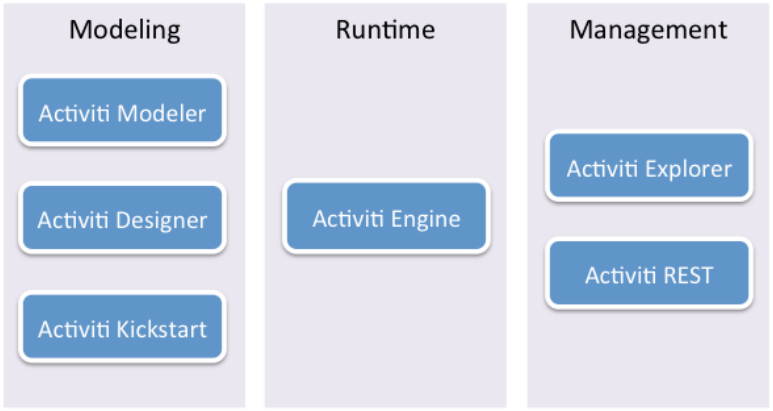

Figure 2.6: Activiti Components Overview (image taken from official Web-site `http://www.activiti.org/components.html`)

for *UserTasks*. Using these *formProperties* helps to create UIs for *UserTasks* as they enable users to define some information on how certain attributes are supposed to be visualized. This means that not only simple forms can be generated for tasks, but rather more complex custom views for *UserTasks* are possible. As an adequate presentation of necessary information is one of the key aspects of this thesis, this is a clear contribution. The framework itself is written in the Java[8] programming language and uses Groovy[9] as a scripting language. A more comprehensive view at Activiti is given by Rademakers [69].

Figure 2.6 gives an overview of the components of the Activiti framework. The *modeling* layer has several components that support the user during the process creation phase. The single components have a common basis and are front-ends, that can be used in different applications. The *Activiti Modeler*, for example, provides a Web interface, which can be used via the *Activiti Explorer* application, and the *Activiti Designer* provides an Eclipse-Plugin for designing BPs. These modelers already have built-in support for Activiti-specific extensions such as the *formProperties*. The BPs are executed by the *Activiti Engine*. This execution (or process) engine supports all extensions that Activiti introduced and is able to process them. It is possible to directly use the *Activiti Engine* in a software application either as a dependency or via a REST interface provided by *Activiti REST*. An example of a BPMS is provided in form of the *Activiti Explorer*, which allows for task/process management, instantiation of processes, management features and a storage facility for reports and history information.

As mentioned above, Activiti introduces a couple of extensions to the BPMN 2.0 Standard. *FormProperties*, for example, can be used to provide an additional specification on what kind of information should be visualized for *UserTasks*. They have several attributes such as *type*, *variable*, *expression* or *writable*. The attribute *type* encodes the type of a property and thus how it should be presented to the user. To accomplish this,

[8] `http://www.oracle.com/technetwork/java/javase/overview/index.html`
[9] `http://www.groovy-lang.org/`

Activiti provides an interface to access this specification and to register custom renderes for each type. The *variable* attribute is the name of the variable (or *dataObject*) that is internally used in the BPMN diagram. This variable also holds the value of the property that should be visualized. The variable, or any variable managed by Activiti for that matter, can be used in the expression. The expression attribute defines the value of the *FormProperty*. Activiti provides the means to access Java Beans, and their methods, via the *$* character. Basically, the *$* character denotes an expression which is evaluated during runtime. Finally, the *writable* attribute signals whether the value of this property can be changed or not. Listing 2.5 shows an example for *FormProperties* on *UserTasks*. More information on how this is realized technically can be found in Section 4.2.1.1.

Listing 2.5: FormProperties usage in *UserTask*

```
<userTask id="authorizeVacationUserTask" name="Authorize Vacation Request">
  <extensionElements>
    <activiti:formProperty id="vacationDate" name="Vacation Date"
        type="dateRange" expression="${vacationDateVar}"
        variable="vacationDateVar" writable="false"></activiti:formProperty>
    <activiti:formProperty id="vacationAuthorized" name="Authorize Vacation"
        type="boolean" expression="${vacationAuthorizedVar}"
        variable="vacationAuthorizedVar"
        required="true"></activiti:formProperty>
  </extensionElements>
</userTask>
```

Additionally, Activiti has an option to not only invoke Web Services from *ServiceTasks*, but also directly execute implementations located in Java classes. The attribute *activiti:class* holds the reference to the Java class and all these classes have to implement a specific interface specified by Activiti. The *activiti:field* extension element is used to define the name of the parameters that can be accessed in the implementing class, and the *activiti:expression* extension element passes the value to this parameter. Listing 2.6 shows how these extensions are specified. A more detailed look into this technique is given in Section 4.2.2.1.

Listing 2.6: *ServiceTask* in Activiti

```
<serviceTask id="notifyEmployeeServiceTask" name="Notify Employee"
    activiti:class="at.ac.tuwien.ict.services.NotifyUserService">
  <extensionElements>
    <activiti:field name="vacationDate">
      <activiti:expression>${vacationDateVar}</activiti:expression>
    </activiti:field>
    <activiti:field name="vacationAuthorized">
      <activiti:expression>${vacationAuthorizedVar}</activiti:expression>
    </activiti:field>
  </extensionElements>
</serviceTask>
```

On top of the *Activiti Modeler* and *Activiti Engine*, the BPMS application *Activiti Explorer* is built. This Web application allows users to create processes, execute them,

and work on tasks assigned to them. Many features of the *Activiti Explorer* are commonly shared by other vendors in one way or another. Figure 2.7 shows the UI of the *Activiti Explorer* during process execution. There are several areas of interest. First, an "inbox"-like system, marked by 1 and 2, is implemented, where available or already processed tasks are shown. The middle part of the figure presents the current process and task. In the upper half, marked by 3, information on the current process is provided. This includes the user who has started the process, a process description, and a priority. The currently active task, this means the task that the user is currently working on, is shown in the lower half. Marked by 4 is the *related content* that is available for this task. This can be anything from documents to references to external Web sites. Finally, the UI for the current task is shown in the section marked by 5. In this case, only a simple form is presented, where the user can choose to approve or decline the vacation request.

This UI does not provide enough information to efficiently process tasks. The only option to provide additional context information is via links or documents that are attached to the current task. However, this requires users to constantly switch between applications to gather the information which they need. Furthermore, it only allows references to either static documents or links to separate applications. If a user needs information about the projects that the requester is involved with, then the only option is to provide a link to a separate application or Web site where this information can be found. This makes it hard for the user to focus on the task, as she constantly has to switch between applications or tabs in the browser.

Such additional context information should be available for each task and should be visualized directly in the UI. Providing this additional context information in a coherent UI is one of the main focal points of this thesis.

2.3 Business Domain Model

As discussed in previous sections (2.2.1 and 2.2.2), BPs and Business Artefacts are important parts of a process-centric software application. But speaking more generally, they are also important components of a Business Domain. Basically, the Business Domain describes in what domain a company conducts its business. This is a somewhat abstract concept and has been a research topic for quite some time. A model of such a Business Domain describes the parts that are involved and is described by Oldfield [60] as:

> "A domain model is a model of the domain within which an Enterprise conducts its business. The Domain Model for one Enterprise should be the same as that for any other Enterprise conducting business in the same domain. When we get down to more detailed levels, different people have different ideas about what constitutes a Domain Model."

So, the Domain Model should be interchangeable between different Enterprises and should facilitate an easy exchange of information between them as everybody uses the

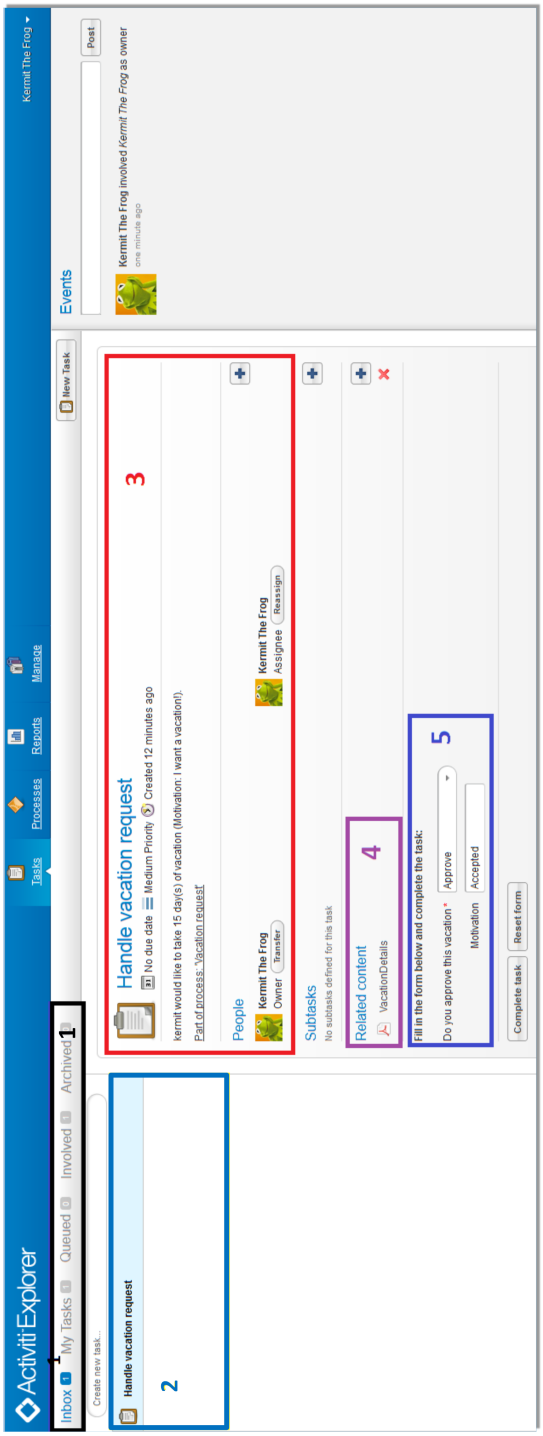

Figure 2.7: Activiti Explorer UI during process execution

same vocabulary to describe parts of the business. In essence, the Domain Model is a representation of concepts from the real-world that are supposed to be used via software applications. The conceptual modeling is not limited to the Business Domain, but can rather be extended to include other parts of software applications, such as architecture, as well. This leads to a model-driven approach for information systems as described by Olivé [61].

Furthermore, a specification of the Business Domain Model in an Enterprise Architecture provides the possibility to generate artefacts that can be directly used by software applications. This helps to align Business Artefacts, on which a software operates, and their specification in the Enterprise Architecture to further bridge the gap between the model of entities and their software counterparts. Figure 2.8 illustrates this approach.

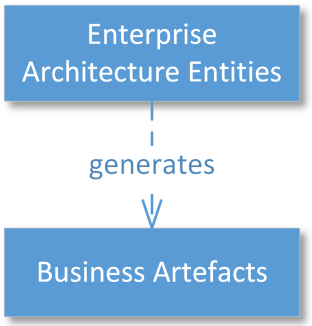

Figure 2.8: Aligning Enterprise Architecture Entities and Business Artefacts in the Software

2.3.1 Business Domain definition with OWL

The necessity of having a formal representation of the aforementioned components, that an Enterprise Architecture is built around, is discussed by Allemang et al. [3]. They show how semantic Web [10] technologies, in particular Resource Description Framework (RDF) [54, 13] and OWL [82], can be used as a reference model for parts of an Enterprise Architecture.

Ontologies are one of the most important concepts for such a formal specification. Through conceptualization an abstract and simplified view of the real world is created including involved entities and relationships among them [31]. OWL is a knowledge representation language for such ontologies and provides formal semantics. It allows to define a common vocabulary or taxonomy and can be used to create knowledge structures for varying domains. Such a domain is often referred to as a knowledge base. OWL is built upon the RDF standard which is a meta-data data model. RDF describes resources in form of triples (subject, predicate and object) and thus allows to connect resources with each other.

In principle, there are three main concepts involved in OWL: OWL *class* specifies what kind of concepts are available in a Business Domain. OWL *individuals* are closely

related to *classes* as they are instances of a specific *class*. And OWL *properties*, which relate *classes* with literals or with each other. Classification technologies and knowledge reasoners can be used to infer new knowledge from an existing knowledge base.

The ability of OWL to be used in many different domains makes it very versatile. As Enterprise Architectures may incorporate many different domains from different businesses or environments, a formal representation of the domains involved is essential. Antunes et al. [5] show in their work how OWL can be used to accomplish such a task. Building these Domain Ontologies is not an easy task and many approaches have been formulated [87]. In essence, theses ontologies describe what kind of concepts are available in a specific Business Domain. However, this does not yet account for instances of these concepts, which are also part of the *domain model*. A definition of what a *domain model* exactly is, is given by Musen as:

> "... when we combine a domain ontology with an enumeration of the instances intended by that ontology for a particular application, the resulting set of classes and instances is what we call a *domain model*." [57]

So basically, the *domain model* holds the concept definitions of the *domain ontology* as well as the *instances* of theses concepts that are used in applications. To accomplish this it is of importance to have these models in a machine processable format. Although OWL is supported by a variety of libraries, a presentation in a programming language is often preferable. Using the formal representation in form of ontologies to automatically generate concepts in programming languages, especially in object-oriented languages, has been discussed for some time [46]. Many modern frameworks use this approach as part of the software development process. The Protégé [10] framework, as one of many, supports generation of object-oriented class structures for Domain Models.

Software applications use a Business Domain Model in some way or another, and often relational database models are used to store instances of concepts. Since this approach has been widely used, a variety of database models along with libraries for various languages exists. One option is to use Hibernate[11] as an object-relational mapping framework to access instances stored in databases. Using an explicit Business Domain Model in combination with typical database concepts, such as Hibernate, is possible as the necessary parts can be generated from an OWL Business Domain Model. Another option would be to directly use a triple store for the Business Domain Model such as Sesame[12], which allows processing and handling of RDF data. A query language, SPARQL Protocol and RDF Query Language (SPARQL) [2], can be used to access information in such a triple store.

It has to be noted that OWL can not only be used to describe Business Domains, or more precisely, the model of Business Artefacts and their relationships, but also to describe workflows based on them. One way to accomplish this is discussed by Wood et al. [85]. Aslam et al. [6] on the other hand describe how semantically enriched specifications of

[10] `http://www.protege.com`
[11] `http://hibernate.org/`
[12] `http://rdf4j.org/`

Web Services, in their case Semantic Markup for Web Services (OWL-S) [1], can be used as part of BPs. Lee et al. [49] show in their work how OWL can be used to describe BPs. They introduce a business ontology, Business-OWL (BOWL), which is modeled as a Hierarchical Task Network (HTN) for the dynamic formation of BPs. However, there is no standard for defining workflows based on OWL available yet and since there is wide spread support for BPMN this technology has been used in this thesis.

2.3.2 Business Domain Example in OWL

In this thesis, OWL, in combination with the Protégé framework, has been used to specify a Business Domain Model. If we consider the running example, where a *vacation request* is processed, the Business Domain Model only involves a couple of structures that are necessary during the BP execution. A structure for an *Employee* is needed, one for the *request* and one for the *projects* that an employee works on.

Figure 2.9 shows how this Business Domain Model is modeled in OWL. The dotted lines basically are relationships between the different concepts. For example, the line from *Employee* to *Project* specifies the relationship *worksOnProject*. This is of course a simplified structure for such a Business Domain Model and they can be rather complex in general. An advantage of using such a OWL defined Business Domain Model is that it is interchangeable between Enterprises and that with a reference model the taxonomy is clearly defined. Knublauch et al. [42] have formulated *A Semantic Web Primer for Object-Oriented Software Developers* to provide an entry point for software developers.

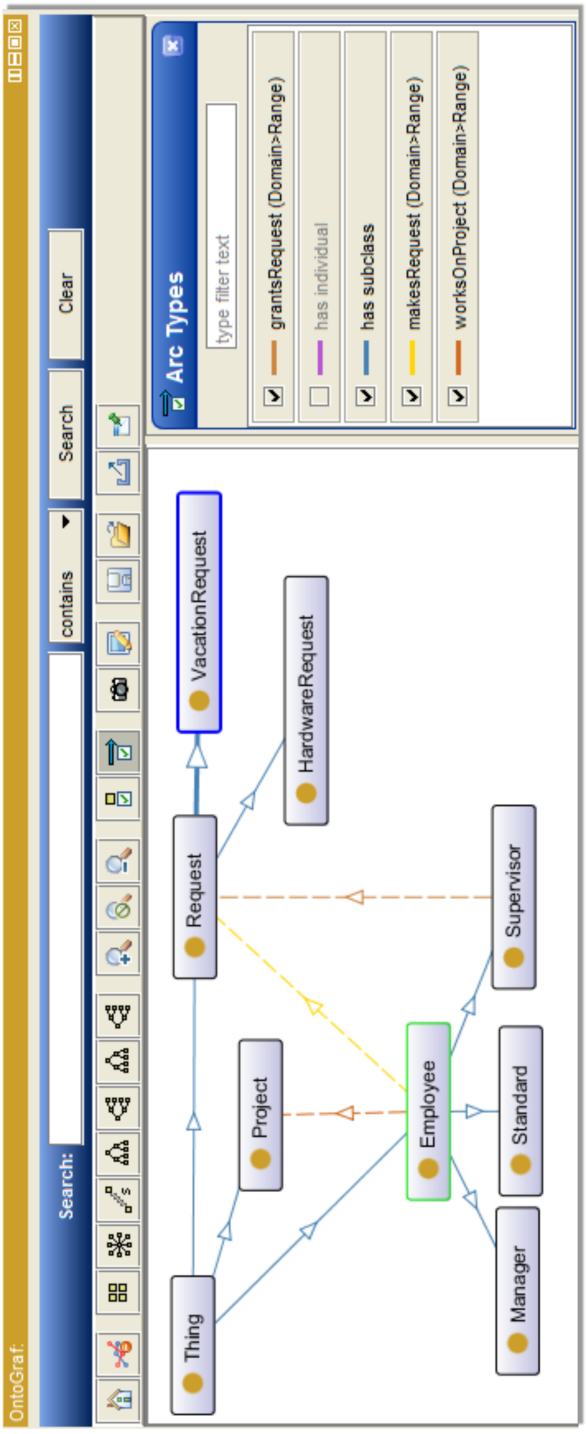

Figure 2.9: Simplified Business Domain Model for the running example in OWL

CHAPTER 3

Related Work

This chapter gives an overview of work related to this thesis. First, approaches to BP modeling, especially with an emphasis on artefact-centric approaches, are presented. In succession, an overview of BPMSs is provided and they are related to the approach used in this thesis for the UI presentation. Finally, approaches for handling context in frameworks with BPMs are discussed.

3.1 Approaches for Software and BPM design

3.1.1 Data-centric approaches

Yongchareon et al. [88] state that views for users with adequate process information are critical in BPMSs. They propose a process view framework for artefact-centric BPs in which they include BPMs, view models and a technique to create views out of BPMs. Furthermore, they introduce consistency rules to keep constructed views and the underlying BPM consistent. To accomplish this, business rules with pre- and post-conditions are used. Yongchareon et al. [89] show how this approach can be used to generate Web-based Business Process Driven UIs. In contrast to the approach of this thesis, another paradigm for designing BPs is used. The main focal point is on the artefacts and how views for artefact-centric BPMs can be constructed. In addition, the approach is not based on a well-defined standard such as BPMN and additional information in the resulting UI is also not discussed.

Bhattacharya et al. [11] present a formal model for artefact-centric BPs and use it to apply static analysis of occurring problems.

Meyer et al. [55] show that it is possible to transform process-centric BPMs into artefact-centric BPMs. They present algorithms on how this can be accomplished with synchronized object life-cycles.

3.1.1.1 PHILharmonicFlows Framework

Ming Chiao et al. [15] describe the framework *PHILharmonicFlows* [80], which allows modeling, execution and monitoring of object-aware processes, as well as the explicit processing of business data and business objects, respectively. To accomplish this, a build-time as well as a run-time environment are provided. The behavior of objects is described through micro processes and the interactions of objects are captured through macro processes. One of the fundamental requirements for their approach is that object types along with their relations are captured in a data model. This is similar to the domain model described in Section 2.3. A more detailed description of the framework can be found in [45]. In contrast to the approach used in this thesis, the proposed framework is based on a data-driven rather than an activity-centric approach to BPs. In addition, we focus on consistent integration and visualization of data objects or Business Artefacts in activity-centric BPs based on the BPMN 2.0 standard.

3.1.2 Process-centric approaches

Typically, BPMs have the overall workflow of activities as a focal point. They focus on the overall process rather than how Business Artefacts are manipulated. Software applications that are designed based on such BPs are process-centric. BPEL is one language that supports this approach as it is entirely focused on the workflow of activities or Web-services.

The overall UI of such software systems should reflect this and provide views to manage current tasks as well as allow navigation between tasks. Seshan [74] gives a comprehensive analysis of process-centric software architectures for Enterprise software systems. He explains how Web-services and J2EE (Java 2 Platform, Enterprise Edition) components can be used to build BPMSs. In addition, BP modeling languages, including BPEL, are discussed and how they relate to SOA. The work is more focused on how all these technologies can be combined to build a BPMS as opposed to the resulting UI of the software. This is in contrast to our approach, as we want to provide a coherent UI for users to handle BPs.

Freund et al. [24] describe how BPMN can be used to automate processes in a company. They provide an introduction on the notation of BPMN 2.0 and show how processes can be modeled and improved. Their focus is on illustrating how BPMN can be utilized for real-world problems and how the business and IT alignment can benefit from BPMN. Camunda is used as their execution engine of choice. Although they state that graphical views are important for business users, they do not focus on providing a consistent UI enriched with additional Business Artefact information, but rather on how BPMN can be applied in companies and what kind of benefits it has on the business and IT alignment.

3.2 Software Architectures and Patterns

Software architecture is, in general a broad topic with many facets and various approaches have been discussed. Many recurring patterns have been identified that can be applied while constructing a software application [48]. These patterns provide guidance for software developers as they implement the application. They range from architectural patterns to common software design patterns. As software applications can have diverse objectives or are running in different environments, there is no single pattern to solve all problems [23].

One common architecture pattern is the layered architecture paradigm. This pattern describes how a software can be built in layers, which expose an interface to the surrounding layers. Often communication is only allowed between adjacent layers. This makes development and also testing of software easier and also facilitates the separation of concerns, which is a design principle that states that each component should only address a separate concern. Bass et al. [8] provide a comprehensive look at software architectures.

The separation of concerns principle is also commonly used when developing Graphical User Interfaces (GUIs) [29]. Since GUIs usually involve UI designers as well as developers the separation is even more important. Patterns used during GUI development range from architectural patterns to usability patterns. In this thesis, the architectural patterns are of interest [40].

One of the most well-known architectural patterns is the Model-View-Controller (MVC) [71] pattern which has been introduced by Reenskaug. It describes how the UI can be separated from the logic of an application and its data. Basically, there are three components involved: Controller, Model and View. The controller is used to manipulate data of the application and decides which view has to be loaded. The view itself receives this information from the controller and displays the data to the user. Finally, the model stores the data that is supposed to be visualized and is manipulated by the controller. Krasner et al. [44] explain how MVC can be utilized.

An Enterprise Architecture is used to describe and control an organization in various ways. It typically includes available BPs and the structures or systems. Enterprise Architectures are specified through modeling languages [47]. We try to align Enterprise Architectures with software architectures by utilizing parts of Enterprise Architectures such as models of BPs and Business Artefacts within the software.

3.3 Frameworks utilizing BPM

3.3.1 Red Hat JBoss BPM Suite

Red Hat JBoss BPM Suite[1] is a BPMS. It is a BP management platform that allows to simulate, manage, monitor and execute BPs. Furthermore, it provides the possibility to use policies, or Business Rules, during the process execution. The platform aims at

[1] `http://www.jboss.org/products/bpmsuite/overview/`

making collaboration between business and IT users/developers more efficient. Flexibility and an easy adaptation of business applications is another target of the platform.

Flexibility into applications is introduced to the platform as BPs are used to provide the Business Logic. As a run-time or execution engine jBPM[2] is used. jBPM uses BPMN-specified BPs and supports BPMN up to version 2.0. Similar to the Activiti project described in Section 2.2.3.4 it provides a mechanism to create, deploy and execute BPs. The creation of BPs is possible either through a Web-interface or via an Eclipse-plugin. The notation of BPMN is used to help business users to create and design processes. In addition, an Application Programming Interface (API) is provided that allows the IT user/developer to access the run-time engine. A custom data model can also be implemented and subsequently used in the BPMs. In a similar fashion to Activiti, it provides a *FormModeler*, which allows the creation of forms for specific tasks.

In contrast to the approach presented in this thesis, jBPM does not allow the specification of additional context information. It only provides the option to create or generate forms for individual tasks, but not to specify additional context information that might be relevant.

3.3.2 CUBA Platform

The CUBA Platform[3] is a full stack Java framework for Enterprise applications development. It makes use of various technologies[4] to support the developer during the development of a product. This framework provides a high level of abstraction, but also allows direct access to the low level API.

For this thesis, the workflow module is of high interest. It allows the execution of BPs and is based on the jBPM 4.0 framework. The CUBA Platform supports the developer during the implementation of BPs and provides a Web-interface for the creation of BPs. To create BPs, first a data model has to be created, or to be reused from previous processes. This data model is used throughout the BP and specifies the objects that the BP works on. This data model is similar, from a conceptual point of view, to the Business Domain model we described in Section 2.3.

Based on the data model, a view (or screen) can be generated as well as adapted by selecting attributes of the object. These views provide information about the data objects, or more precisely their instances, involved in a BP. Such views also provide the possibility to search for instances of a particular entity or to use filter mechanisms on them. The views can be created either programmatically, as shown in Listing 3.1, or declaratively.

Listing 3.1: Creating a view programmatically

```
View view = new View(Order.class)
  .addProperty("date").addProperty("amount")
  .addProperty("customer", new View(Customer.class).addProperty("name"));
```

[2] http://www.jbpm.org
[3] https://www.cuba-platform.com/
[4] https://www.cuba-platform.com/technologies

The declarative approach uses an XML descriptor and deploys it to a repository. These views can then be retrieved by the application at run-time. An example for such a declarative view specification is given in Listing 3.2. The content of these listings is based on an example given in the official documentation.

Listing 3.2: Creating a view declarative

```
<view class="sales.entity.Order" name="orderWithCustomer" extends="_local">
  <property name="customer" view="_minimal"/>
  <property name="items" view="itemsInOrder"/>
</view>
```

These views allow using relations of objects as well as attributes. Through relationships it is possible to navigate to other views or embed them in the current view. This approach is similar to what we try to accomplish with our approach. In contrast to our approach, they use their own specification language for views and only allow the creation of Web-interfaces. Our approach provides a separate specification of the context, and thus the views, which can then be used by an arbitrary implementation. For more information on the CUBA Platform, please refer to the official Web-site and documentation.[5]

3.4 Context in BPM Frameworks

BPMs are used in a certain context, but often this context information cannot be specified adequately. Furthermore, there is a gap between the definition of the business model with its artefacts and the definition of the process itself. Saidana et al. [73] present an approach for formalizing the contextual knowledge to be able to adapt BPs according to their context. They propose a meta-model for the context representation called CM4BPM (Context Meta-Model for Business Process Management). The idea is to identify and formalize factors that directly influence the execution of BPs. To represent this knowledge an upper ontology built in OWL is used. In contrast to our approach, they focus on the adaptation of BPs rather than enriching the UI with additional information.

Similar work has been carried out by de la Vara et al. [20], who propose an approach for BP contextualization via context analysis. Context reasoning as well as discovery of relevant properties are discussed, and a specification for such context variants of BPs is given. By doing so, context-aware BPs are created and can be executed. The approach has its focus on the identification and discovery of context information. A generic BP is created first and context variations that are not present in the BPM are identified subsequently. These variations influence the execution of the BP and thus the contextualized BP is adapted. We aim at a flexible user interface for presenting additional artefact information rather than adapting BPs.

Contextualization of BPs has also been studied by Roseman et al. [72], who show the necessity for flexibility of processes to adapt to new environments. They use a goal-oriented process modeling approach to identify relevant context elements and they

[5] https://www.cuba-platform.com/documentation

propose a framework with a meta-model to classify relevant context information. Also this work has its focus on the adaptation of BPs.

Traditionally, BP modeling has its focus on the workflow of activities, while Business Artefact are often only considered as an afterthought. However, they are directly related to activities in workflow systems and thus should be considered up-front. Cohn et al. [18] describe how Business Artefacts can be used as the foundation of BP modeling and why they are an integral part of business applications. We make use of Business Artefacts to provide additional information that is not directly specified in the BP itself.

Traetteberg et al. [79] discuss inflexibility of user interfaces of BPM tools and combine process and UI modeling to enhance the usability of BPM tools. Parts of their concerns have been addressed with the introduction of BPMN 2.0, but the problem of inflexibility remains. They propose a model-based user interface design to be combined with BPMN. In contrast to our approach, the focus is rather on how a user interface should be designed as opposed to what kind of information is to be presented.

Integration of UI Services into BPM applications gained popularity with the introduction of BPMN 2.0. User Tasks are now a standardized form of task in BPMN. Hohwiller et al. [36] describe how UI Services can be integrated into SOA-based BPM applications. They also describe how reusable specifications of UI Services can be used to embed UI components into BPM applications. They use JSR-286 Portlets and WSRP 2.0 for embedding system-independent and reusable UI Services. The focus is on providing interchangeable specifications for user tasks as opposed to additional artefact information for BPs that we propose.

Braun et al. [12] focus on the development of domain-specific extensions for BPMs. New tasks are introduced with specialized attributes to support the execution of such a process. In contrast, we aim at a more generic solution that can be used across several domains.

Liptchinsky et al. [52] discuss context-aware processes with an emphasis on highly dynamic non-routine processes that often occur during social collaboration tasks. They propose a custom modeling framework that includes context dependency rules to provide the necessary information. In contrast to our approach, the focus is not on an automatic execution of BPs but rather on how different partners collaborate with each other and what kind of context is necessary for them.

Sousa et al. [76] discuss how a model-driven approach can be used to derive UI from BPs. Their approach consists of four steps and each step refines the mapping of the source to target model. They use a *TaskModel* and *DomainModel* to create an *Abstract UI* which is refined to a *Concrete UI* and in a final step to a *Final UI*. They state that there is a lack of correlation between BP and UI design and that information is often spread in different artefacts. However, in contrast to our approach, they focus more on how a UI for BPs can be created and how models can be refined, rather than what should be displayed to the user. We try to enrich the BPM with additional context information, which is provided via Business Artefacts.

CHAPTER 4

Architecture for a Process-centric Software Application

The proposed software framework aims at integrating an existing execution engine for BPMs into a software architecture. The main focal point of the framework is to provide a coherent UI that supports users throughout their daily routine. In this chapter, based on the requirements for such a framework, the architecture of the software and their essential components are described in detail.

4.1 Requirements

The resulting framework should primarily help users to process their computer-supported tasks through a coherent UI. However, it should also allow business users to easily change or add BPMs. Furthermore, software developers should be able to provide specifications and implementations of UI parts and to add them without the need of adaptation of the architecture. In essence, the framework should fulfill the following requirements:

1. **BPM-driven software**
   The framework should use a separate specification of the business logic given through BPMs. That is, the business logic shouldn't be hard-coded into the source code as it is subject to change. Thus, these BPMs control how the resulting application behaves and which tasks are accessible. The specification of the BPMs is given accordingly to the BPMN 2.0 standard and the software shall be able to handle and execute these BPs.

2. **Browse available BPs and tasks**
   The framework shall provide a list of available processes and tasks. Users shall be able to select and start BPs. Furthermore, they can decide whether they want to work on specific tasks or not.

3. **Dynamic UI generation for tasks**
   Each task that requires user interaction has its own custom UI. This UI shall be generated dynamically based on the properties of the task.

4. **Coherent UI during BP execution**
   During execution of BPs, the framework shall provide a coherent UI to the users. The UI shall provide all necessary information for users so that they can fulfill their tasks without the need to gather information from other applications.

5. **Business Repository**
   Business applications often use and share services across several applications in a business company. The framework shall provide the means to access services and Business Artefacts stored in a repository.

6. **Additional Context configuration**
   Since BPs do not provide all context information to create an adequate UI, a separate specification of such information is necessary. The framework shall be able to load and process such a configuration.

7. **Presenting additional context through Business Artefacts**
   Presenting the necessary information to the users is mandatory. The additional context configuration should be automatically translated and a visual representation should be created.

8. **Single Machine Environment**
   The feasibility prototype is implemented in a single machine environment. This means that there are no concurrent, distributed actions on BPs allowed and that at most one user works on a task at any given point in time.

4.2 Architecture

The architecture of the framework is shown in Figure 4.1. It follows the layered architecture paradigm.[1] It extends the standard 3-layer architecture with an additional layer, the *Business Repository*, and thus results in a 4-layer architecture. The idea of such an architecture is that only higher layers can directly access lower level layers. That is, only higher layers have a dependency on the lower level layers, but not the other way around. Lower level layers trigger higher level layers only through notifications, but not via direct access. The actual implementation of these patterns might deviate somewhat. More information on architectural and software patterns are given in [8, 23].

In each layer, there are several modules with more complex inner structure, which is omitted here. The UI is handled through the *User Interface* layer, where views for Business Artefacts are created. The BPM and its execution engine are part of the

[1] Software Architecture Patterns: `http://www.oreilly.com/programming/free/files/software-architecture-patterns.pdf`

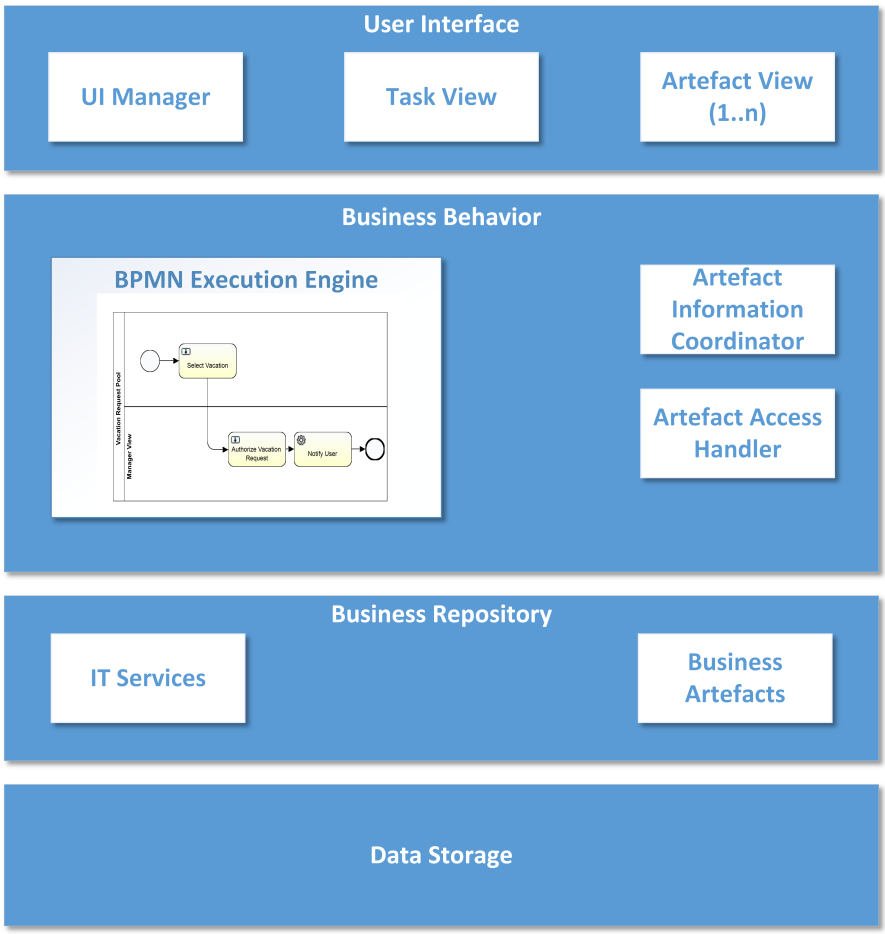

Figure 4.1: 4-layer software architecture for BPMN execution

Business Behavior layer. The Web Services are part of the *Business Repository* layer. They implement the IT Services that the given application is intended to provide for the business through the given BPM specified in BPMN.

4.2.1 User Interface Layer

An essential property of the software architecture is its flexibility with regard to a custom UI for each process or even task. This is handled in the *User Interface layer*. Basically, a *UI Manager* is used to coordinate and place parts of the UI in the overall UI. This *UI Manager* is notified from the lower layers and then internally creates the views accordingly. All changes to the UI, e.g., a new view for a task, are first dealt with by the *UI Manager*.

The UI is based on JavaFX[2], which allows us to use a descriptive specification of UI parts that can be easily integrated into other views. For this purpose, an XML-based

[2] `http://docs.oracle.com/javase/8/javafx/get-started-tutorial/`

language, FXML[3], is used, which allows the declarative specification of UI parts. This modular approach allows us to have separate specifications and implementations of views based on Business Artefacts, which can then be placed in the overall UI on demand. In addition, JavaFX supports the usage of Cascading Style Sheets (CSS) [14], a common style technique used in Web applications, to manipulate the look and formatting of the UI. Furthermore, JavaFX provides a built-in technique that allows data binding between views and a data model. With this data binding, it is possible to dynamically propagate changes from the view to the model and the other way around. This is especially useful as "boilerplate" code[4], that is code that is seemingly repetitive and used in many places to achieve the same result in a specific language, becomes unnecessary.

JavaFX basically allows the separation of the UI into two parts: *View* and corresponding *Controller*. Each FXML-View can reference a controller which handles, for example, events from the view. This technique is closely related to the MVC pattern and allows a separate specification and implementation of the view and the logic. It has to be noted that JavaFX enables the usage of the MVC pattern but doesn't enforce it and other patterns are possible as well. In fact, in this thesis another pattern has been used to implement the UI (more information is given in Section 4.2.1.2).

With JavaFX, it is possible to develop desktop applications as well as Rich Internet Applications (RIAs), which run on various devices. It is intended to be the replacement of the Swing toolkit and has been introduced into the Java Development Kit (JDK) of Java 1.8. For more information on JavaFX please refer to [83].

4.2.1.1 Task View

A *Task View* is a visual representation of a *User Task* in the BP. It is generated during runtime based on the properties of the corresponding task. Figure 4.2 illustrates schematically how such a *Task View* is generated in our architecture. The *BPM Engine*, which consists of the BP model and an execution engine, notifies the UI that a new task is available, by triggering a *Task View*. This view is created dynamically based on the properties of the task and using *Business Artefacts*. Internally, first the *UI Manager* is triggered, which then creates the *Task View*. This step is omitted in the figure.

Let us consider our running example again. In the vacation request process, more specifically its *create request* task, a property for this task may store a value representing within the BPMN process the time frame of the vacation request. For entering this time frame, a corresponding form has to be generated. In this case a simple form with two input fields for the start date and end date of the vacation request is generated. The selection is either possible via a *DatePicker* widget or via a simple text input. Figure 4.3 shows how such a simple form of a UserTask looks like.

[3] `http://docs.oracle.com/javafx/2/fxml_get_started/jfxpub-fxml_get_started.htm`

[4] `https://en.wikipedia.org/wiki/Boilerplate_code`

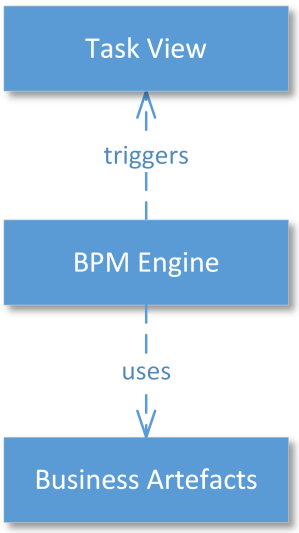

Figure 4.2: Generating *Task Views*

To accomplish this technically, a custom renderer for so-called *FormProperties*, properties that belong to a task, has to be created. The properties are specified at the task level directly in the BPMN specification of the BP. This allows the execution engine, in our case Activiti, to access these *FormProperties* of tasks. Listing 4.1 shows the *FormProperty* of the *Select Vacation UserTask*. In this case it is only one property with datatype *dateRange* for which a UI has to be generated. This technique is similar to the one implemented by the Activiti-Explorer Web application. For a more detailed description please refer to Section 2.2.3.4.

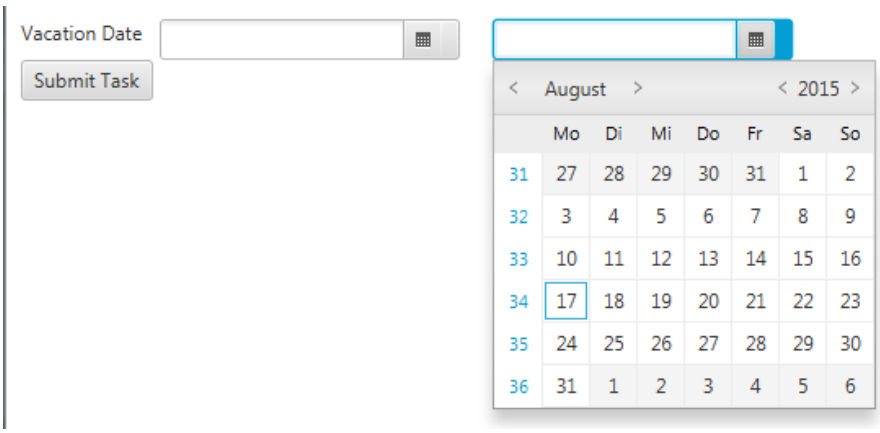

Figure 4.3: Generated form for *Create Vacation Request* task

Listing 4.1: *Select Vacation User Task* in running example

```
<userTask id="selectVacationUserTask" name="Vacation Request"
    activiti:assignee="${initiator}">
  <extensionElements>
    <activiti:formProperty id="vacationDate" name="Vacation Date"
        type="dateRange" variable="vacationDateVar"
        required="true"></activiti:formProperty>
  </extensionElements>
</userTask>
```

Specifying *FormProperties* provides information about what should be visualized. However, it does not provide information on how it should be visualized. Thus, a renderer is necessary that can process the information of the property and generate a UI accordingly. Activiti allows registering such renderers and accessing them during runtime. This enables us to generate a custom UI for each *UserTask*. A renderer for the *dateRange* datatype of the *FormProperty* from Listing 4.1 is given in Listing 4.2. Basically, a label is created with the name of the corresponding property and two input fields for the date values. In this case a simple *DatePicker* widget is used. The elements are placed in a horizontal container and styled via CSS. This results in the UI shown in Figure 4.3.

Listing 4.2: JavaFX renderer for *FormProperty*

```
/**
 * Generates a Node with elements for the current property.
 * @return      the node containing the elements
 */
@Override
public Node render() {
  HBox hbox = new HBox();
  Label dateRangeLabel = new Label(this.property.getName());
  dateRangeLabel.getStyleClass().add(ActivitiFormRenderer.CSS_CLASS);
  Node value = null;
  if (this.property.isWritable() == true) {
    HBox tempHBox = new HBox();
    tempHBox.getStyleClass().add("activitiInputFieldRow");
    this.startDate = new DatePicker();
    this.startDate.getStyleClass().add(ActivitiFormRenderer.CSS_CLASS);
    this.endDate = new DatePicker();
    this.endDate.getStyleClass().add(ActivitiFormRenderer.CSS_CLASS);
    tempHBox.getChildren().add(this.startDate);
    tempHBox.getChildren().add(this.endDate);
    value = tempHBox;
  } else {
    Label dateRangeValue = new Label(this.property.getValue());
    value = dateRangeValue;
  }
  hbox.getChildren().add(dateRangeLabel);
  hbox.getChildren().add(value);
  return hbox;
}
```

Since there are many variations of properties and datatypes, an implementation of a renderer has to be provided for each of them. The Activiti engine provides the possibility to access these FormProperties of tasks, so that only a custom JavaFX renderer for them is necessary. We provide the option to use custom datatypes for properties, for which a new renderer can be added easily. Properties can involve entire Business Artefacts such as employees. In this case, a more complex form is generated, that allows entering all necessary values (e.g., firstname, lastname, etc.). With this mechanism in place, it is possible to generate UIs in JavaFX for each task based on its properties [35].

4.2.1.2 Artefact Views

With the *Task Views* in place, we can focus on the context information that we want to present to the user. In the previous section, we described how we generate a *Task View* for the *Select Vacation Task* of our running example. The next task of our running example would be the *Approve Vacation Request Task*. The supervisor who works on this task now needs more information to complete it. For example, she might need information about projects that the requester is involved with or about the employees, that are involved with a project that the requester works on. This information, based on its Business Artefact, has to be presented to the supervisor. Figure 4.4 shows an example for such an *Artefact View*.

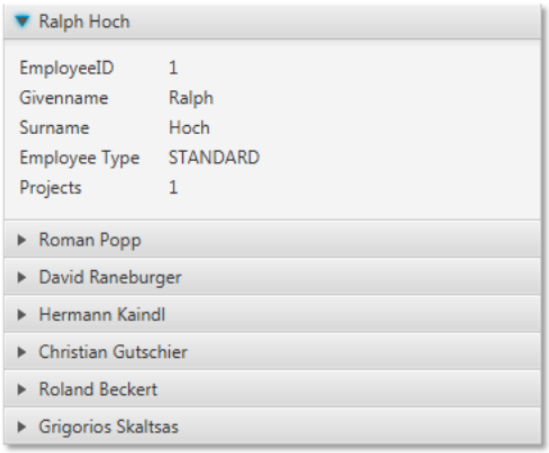

Figure 4.4: *Artefact View* of employees

This *Artefact View* holds a list of employees who work on a specific project. The view is kept simple and the employees are presented in an accordion widget with simple text information. These views can be as complex as necessary, but for the purpose of this thesis, a simple design is sufficient.

Still, we have to explain how such *additional artefact information* is fetched and presented in the UI technically. Figure 4.5 illustrates schematically how this is accomplished with dynamic *Artefact Views*. Basically, the *UI Manager* loads a view based on given configuration information assigned to the task. Such configurations specify for

each process or task what *additional artefact information* should be displayed. This is explained in more detail in Chapter 5. Each *Artefact View* either corresponds to one Business Artefact each, or in more complex views to several ones (for example in relationships among artefacts).

The *Artefact View*, being a module within the top layer of our 4-layer architecture, has an architecture itself, i.e., we actually propose a combination of architectures. The architecture of the *Artefact View* follows the Model-View-Presenter (MVP) [68] pattern. The MVP pattern is a derivation of the widely used MVC pattern, where the Presenter is used as a communication bridge between the so-called Model and the (FXML-)View. As MVP is based on the MVC pattern, it shares some of its concepts. Again, it uses three main components (View, Presenter and Model) and describes how they interact with each other [5]. In contrast to the Controller in MVC, the Presenter is always associated with exactly one view. The Presenter is decoupled from the view and only interacts with the view through an interface. This makes unit testing of UI parts much easier. In addition, MVP is often used if there are different UI technologies involved.

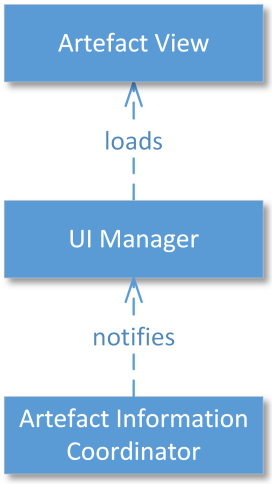

Figure 4.5: Loading Artefact Views

In essence, the view only handles what is directly presented to the user in the UI and the presenter holds the logic of the UI component. That is each time the view initiates an action, it is delegated to the presenter and handled there. An example of such an action would be saving a document. The presenter handles this invocation and upates the model accordingly. After the action has been completed, the presenter notifies the view again, so that the view is updated.[6,7]

[5] Martin Fowler - GUI Architectures: http://martinfowler.com/eaaDev/uiArchs.html

[6] http://stackoverflow.com/questions/2056/what-are-mvp-and-mvc-and-what-is-the-difference

[7] http://www.infoworld.com/article/2926003/microsoft-net/exploring-the-mvc-mvp-and-mvvm-design-patterns.html

There are two variants of the MVP pattern available: Passive View and Supervising Controller. The Passive View pattern is the one described above, where the view contains no logic and everything is handled by the presenter. Furthermore, the view does not know the model and does not interact with it directly (only through the presenter). In contrast, the MVP Supervising Controller pattern allows the view to bind to the model directly. In this case, the presenter passes the model to the view so that it can access the model. The presenter still handles the logic of the view, for example, if a button is pressed, but also performs the data binding between view and model.[8] [9]

Another variant of the MVC pattern that, for the sake of completeness, should be mentioned here, is Model-View-ViewModel (MVVM), which has been introduced by Gossman [10]. It introduces a *ViewModel* which basically acts as a model for a specific view. Data binding is a key concept of MVVM, as the view is bound to the *ViewModel*. The idea is that this *ViewModel* is another abstraction layer between the view and the model, since often properties of the underlying model can not be directly mapped to a view. This is were the *ViewModel* comes into play and maps parts of the model to a representation that can be bound to a view.[11]

There are also other GUI architecture patterns available [40], but the ones sketched here are some of the most commonly used ones. All these patterns have common properties and sometimes only deviate in details. It has to be noted that often implementations of these patterns vary in details and sometimes use concepts of others as well and combine them.

In this thesis, we use the MVP (SC) pattern. We use a presenter as the mediator between the view and the model, but also allow, if possible, direct data bindings between the model and the view. Still, all the logic of the view is handled in the presenter. The general architecture of an *Artefact View* is shown in Figure 4.6.

The *Presenter* has a reference to its *Model*, through which it accesses the Business Artefacts. More precisely, the *Model* is a representation of a Business Artefact. This *Model* holds the values of a Business Artefact that should be displayed in the *View*. The *View*, however, does not have direct access to these values and the *Presenter* has to mediate or map the *Model* to the *View*. Since we use the MVP (SC) pattern, the *Presenter* might also bind certain fields of the *View* directly to the *Model*. In this case, the *View* has direct access to the values. In addition, the *Presenter* also holds a reference to the *Artefact Access Handler*, as it might become necessary to access the *Business Repository*. Although the *Model* contains the business logic to update its values, there might be instances where it is necessary to pass the *Model* to another Business Artefact. This is accomplished through the *Artefact Access Handler*.

[8] http://www.codeproject.com/Articles/288928/Differences-between-MVC-and-MVP-for-Beginners

[9] http://www.codeproject.com/Articles/228214/Understanding-Basics-of-UI-Design-Pattern-MVC-MVP

[10] John Gossman - Introduction to Model/View/ViewModel pattern for building WPF apps: http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx

[11] https://dzone.com/refcardz/mvvm-design-pattern-formula

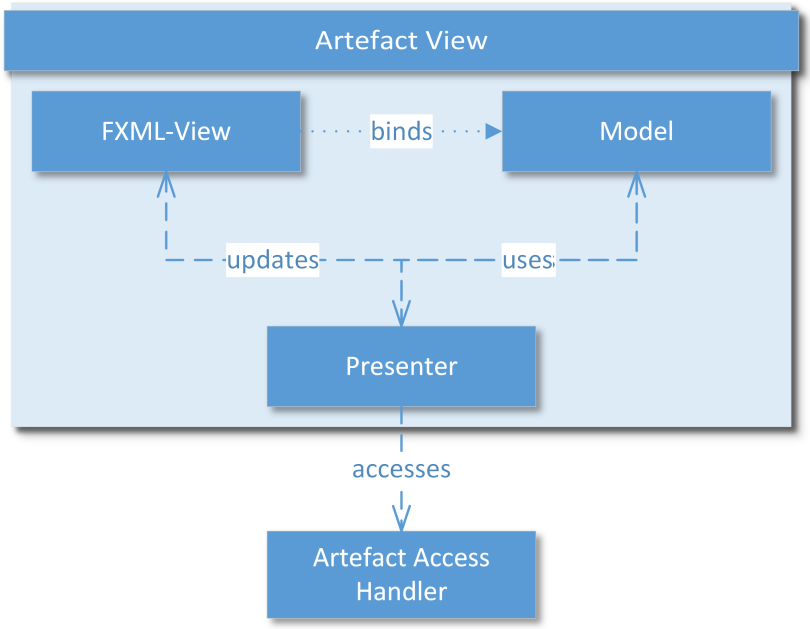

Figure 4.6: Architecture of embedded Artefact Views

In this thesis, we tried to use the data model directly in the view. However, there are certain restrictions that apply and not all properties of the models of our Business Artefacts could be mapped to the view directly. Thus we also used, in some instances, a *ViewModel*, which helps us to further abstract from the representation of a Business Artefact. In this case the *ViewModel* contains the elements that are necessary for a view. The elements are loaded from the corresponding model and then stored in the *ViewModel*. Thus the *ViewModel* is a state of the data model. In such a case, the *Presenter* is still implemented and just binds the view to the *ViewModel*.

With the details of the pattern explained, we can describe how this is technically accomplished. Considering the running example and the *Artefact View* of Figure 4.4, we describe how such a view of a list of employees, who work on a project, is created. Basically, there are two views with presenters involved. The first view contains the accordion widget. This view gets populated dynamically with views of the detailed information of users. This is our second view, which is embedded in the first view. The accordion widget is very simple and only contains the definition of the container. The FXML definition of the accordion widget is shown in Listing 4.3.

Listing 4.3: Accordion widget definition for Users

```
<Accordion fx:id="projectsEmployeeAccordion"
    xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="at.ac.tuwien.ict.fxml.FXMLAllProjectEmployeesPresenter">
</Accordion>
```

The view that contains the inner part of the accordion, the user details, is shown in Listing 4.4. This view contains several fields, such as *employeeGivennameLabel*, which hold the values of the data model. The view has a fairly simple structure and only a couple of labels are shown to the user. It has a field called *fx:controller* which holds a reference to its presenter. This is optional and could be realized in other ways as well.

Listing 4.4: User information view (FXML)

```
<VBox id="UserInformationView" xmlns="http://javafx.com/javafx/8"
    xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="at.ac.tuwien.ict.fxml.FXMLUserInformationPresenter">
   <children>
      <HBox minHeight="20.0">
         <children>
            <Label prefWidth="100.0" text="EmployeeID" />
            <Label fx:id="userInfEmployeeIDLabel"/>
         </children>
      </HBox>
      <HBox minHeight="20.0">
         <children>
            <Label prefWidth="100.0" text="Givenname" />
            <Label fx:id="employeeGivennameLabel"/>
         </children>
      </HBox>
      <HBox minHeight="20.0">
         <children>
            <Label prefWidth="100.0" text="Surname" />
            <Label fx:id="employeeSurNameLabel"/>
         </children>
      </HBox>
      <HBox minHeight="20.0">
         <children>
            <Label prefWidth="100.0" text="Employee Type" />
            <Label fx:id="employeeTypeLabel"/>
         </children>
      </HBox>
      <HBox minHeight="20.0">
         <children>
            <Label prefWidth="100.0" text="Projects" />
            <Label fx:id="employeeProjectsLabel"/>
         </children>
      </HBox>
   </children>
</VBox>
```

The view for the Business Artefact, including the look and feel, is completely defined via these two specifications. What is still missing is the business logic for actions that are performed and the connection to the data model. The corresponding presenter for the *User Information View* is shown in Listing 4.5. There are several attributes with the annotation *@FXML*, such as employeeGivennameLabel, defined. These attributes are automatically mapped to the corresponding field in the FXML-View. The presenter can set these

attributes, and, by doing so, also sets the value in the view, or binds them to an element from the data model. Here a bi-directional binding between the attributes and the data model is established. *Bindings.bindBidirectional(employeeGivennameLabel.textProperty(), this.dataModel.givenNameProperty());* for example, binds the *givenName* property of the data model to the label of the view.

Listing 4.5: User information view (Presenter)

```
@Component
public class FXMLUserInformationController
                        extends AbstractController<EmployeeDataModel> {
  @FXML
  Label userInfEmployeeIDLabel;
  @FXML
  Label employeeGivennameLabel;
  @FXML
  Label employeeSurNameLabel;
  @FXML
  Label employeeTypeLabel;
  @FXML
  Label employeeProjectsLabel;

  public void initialize() {
    this.userInfEmployeeIDLabel.setText(this.dataModel.getEmployeeID());
    ...
    // Bindings.bindBidirectional(userInfEmployeeIDLabel.textProperty(), this
        .dataModel.employeeIDProperty());
    Bindings.bindBidirectional(employeeGivennameLabel.textProperty(),
                                  this.dataModel.givenNameProperty());
    Bindings.bindBidirectional(employeeSurNameLabel.textProperty(),
                                  this.dataModel.surNameProperty());
    Bindings.bindBidirectional(employeeTypeLabel.textProperty(),
                                  this.dataModel.typeProperty());
    ...
    this.employeeProjectsLabel.addEventHandler(MouseEvent.MOUSE_CLICKED,
                                          new EventHandler<MouseEvent>() {
      @Override
      public void handle(MouseEvent event) {
        event.consume();
        FXMLUserInformationPresenter.this.
          notifyAllListenersForViewPart("employeeProjectsLabel", event);
      }
    });
  }
}
```

The Presenter of the *User Information View* uses a data model *EmployeeDataModel*, which represents an employee. In this case, this Domain Model is actually a *ViewModel*, which holds references to the actual model from the *Business Repository*. We introduced properties into our model and, by doing so, we are able to utilize the binding mechanism for properties. However, this has one disadvantage as Java classes with properties can not

be serialized anymore since the properties are not implementing the serializable interface. Thus, we can not store the model to the database and decided to use a *ViewModel* instead. Listing 4.6 shows the *EmployeeDataModel*.

Listing 4.6: EmployeeDataModel

```
public class EmployeeDataModel extends DataModel {
  private LongProperty emplID = new SimpleLongProperty(this, "employeeID");
  private StringProperty gName = new SimpleStringProperty(this, "givenName");
  private StringProperty sName = new SimpleStringProperty(this, "surName");
  private StringProperty type = new SimpleStringProperty(this, "type");
  private StringProperty vacStart =
                      new SimpleStringProperty(this, "vacationStart");
  private StringProperty vacEnd =
                      new SimpleStringProperty(this, "vacationEnd");
  private StringProperty fullEmployeeDescr =
                      new SimpleStringProperty(this, "fullEmployeeDescr");

  public EmployeeDataModel(Employee employee) {
    this.emplID.set(employee.getID());
    this.givenName.set(employee.getGivenName());
    this.surName.set(employee.getSurName());
    this.type.set(employee.getType().toString());
    this.vacationStart.set(employee.getVacationStart().toString());
    this.givenName.set(employee.getVacationEnd().toString());
    fullEmployeeDescr.bind(Bindings.concat(
        givenName.get() + " " + surName.get() + "(" + employee) () + ")"));
  }
  ...
}
```

Technically, each *Presenter* shares a common interface. This interface contains methods to access the *Data Model* and the *View*. In addition, there are methods defined to pass arguments to the *Presenter* or to add listeners for specific properties. The listener for properties is necessary to react to changes in the selection of certain properties. The property listener is explained in more detail in Chapter 5. Listing 4.7 shows the interface for *Presenters*.

Listing 4.7: Interface of Presenters

```
public interface IPresenter<M extends DataModel> {
  public void setView(Node view);
  public Node getView();
  public void setDataModel(M dataModel);
  public M getDataModel();
  public void addChangeListenerProperty(String viewPart, IController<?>
      relController, String property);
  public abstract void setArguments(Object... arguments);
  public boolean isPropertyInView(String property);
  public void loadDataModel();
}
```

In summary, this section showed how a view for Business Artefacts is built and how it relates to its Model and Presenter. Furthermore, it shows that views can be embedded into other views and that views can be exchanged and reused at any place. The MVP pattern is applied to build the *Artefact Views* and the implementation is shown. There are also other frameworks available that handle JavaFX with the MVP pattern. One well-known approach is the *afterburner.fx*[12] project by Adam Bien. What is still missing is how the view is loaded and how it is placed at the appropriate position on the screen. Since this relies heavily on the configuration of the context, it is described in detail in Chapter 5.

4.2.2 Business Behaviour

In principle, the applications main purpose is to execute BPMs and to support the users via an UI during their tasks. Since these BPMs dictate what kind of activities are performed, and in what order, the application does not have a typical business logic layer. The main part of the layer that handles the application logic is the embedded *BPM Engine*. But there are also other parts necessary for the application to function. The *Business Behaviour* layer holds these modules, and a description of them is provided in this section.

4.2.2.1 BPM Engine

The business logic of the proposed framework is handled through BPMs and thus not directly implemented in the application itself. The BPs have a specification and are executed during run-time. Thus an execution engine is necessary that handles them and performs the tasks assigned. In this thesis, the Activiti engine is used to process BPs specified in BPMN 2.0. The engine is embedded into the application and a single module, *BPM Engine*, handles the communication. Each request to the engine, for example, if a process should be started or a task should be submitted, is routed through this module.

From a technical perspective, the module exposes an interface that the application can use to invoke requests to the *BPM Engine*. Listing 4.8 shows an excerpt of this interface. The methods provided by the interface concern the deployment and starting of BPMs as well as handling single tasks. The method *getTaskFormProperty* is of particular interest, since it allows access to the *FormProperties* necessary to generate the *Task Views*.

Listing 4.8: Interface of BPM engine module

```
public interface IActivitiEngine {
  public void deployProcess(String processPath);
  public void runProcess(String processID);
  public List<FormProperty> getTaskFormProperty(String taskID, String pID);
  public void submitTask(String taskID, Map<String, String> properties);
}
```

[12] https://github.com/AdamBien/afterburner.fx

The sequence of activities is determined by the BPM and the engine executes one activity after another. As long as there are open tasks in the currently processed BP, the *BPM Engine* always stays within the active process. However, the user always has the option to switch the process and choose another process to work on. The interface of the *BPM Engine* provides methods to support this behaviour.

Since the main focal point of this thesis is to provide a useful UI, we will give a description about the *FormProperties* and how they are processed by the engine. Let us use the *Select Vacation Task* of Section 4.2.1.1 to illustrate how Activiti handles them. Listing 4.1 showed how we define *FormProperties* for *User Tasks*. The definition of the *FormProperty* is provided here again in Listing 4.9 for convenience.

Listing 4.9: Select Vacation User Task Form Property

```
<activiti:formProperty id="vacationDate" name="Vacation Date"
    type="dateRange" variable="vacationDateVar"
    required="true"></activiti:formProperty>
```

This *FormProperty* has a couple of attributes with the most important being its *id* and its *type*. The *type* determines how such a property is to be handled and, consequently, also how it is to be presented to the user. Activiti supports the standardized types, String, Integer, etc., but often custom types, such as a date range, are necessary. There is already a technique embedded in Activiti that allows to add custom *form types*. Listing 4.10 shows the definition of the date range form type.

Listing 4.10: Custom Form Type *DateRange*

```
public class DateRangeFormType extends AbstractFormType {

  private static final long serialVersionUID = -955513901410364673L;
  public static final String TYPE_NAME = dateRange;

  @Override
  public String getName() {
    return DateRangeFormType.TYPE_NAME;
  }

  @Override
  public Object convertFormValueToModelValue(String propertyValue) {
    DateRange range = DateRange.parse(propertyValue);
    return range;
  }

  @Override
  public String convertModelValueToFormValue(Object modelValue) {
    if (modelValue == null) {
      return null;
    }
    return modelValue.toString();
  }
}
```

Basically, Activiti already provides an *AbstractFormType* class, which can be extended. The methods *getName*, *convertFormValueToModelValue* and *convertModelValueToFormValue* have to be overwritten and provide the information on the type. In essence, this method describes how such a property can be converted to and from the model.

The implementation of the *FormType* is quite simple and the only step left is to make it known to the engine. Activiti provides the possibility to either programmatically configure its engine or use an XML specification. The configuration may involve multiple configuration parameters such as database type or custom form types. Listing 4.11 shows an example configuration, which adds our custom form type *DateRange*. The only line to make the *FormProperty* known is *<bean class="at.ac.tuwien.ict.activiti.form.DateRangeFormType"/>* where the class of the implementation is specified. This technique is built in into Activiti and is described in detail in its User Guide[13]. An example implemention is also provided in the Activiti-Explorer Web application[14].

Listing 4.11: Activiti Engine Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="processEngineConfiguration" class="org.activiti.engine.impl.
      cfg.StandaloneInMemProcessEngineConfiguration">
    <property name="databaseSchemaUpdate" value="true"/>

    <property name="customFormTypes">
      <list>
        <bean class="at.ac.tuwien.ict.activiti.form.DateRangeFormType"/>
      </list>
    </property>
  </bean>
</beans>
```

[13] http://www.activiti.org/userguide/#forms

[14] https://github.com/Activiti/Activiti/tree/master/modules/activiti-explorer

With the implementation of the *FormType* and the configuration in place, we can use *FormProperties* in our BPMN specification. Listing 4.12 shows an excerpt of the implementation of the BPM Engine module.

Listing 4.12: Activiti BPM Engine Module

```
@Service
public class ActivitiEngine implements IActivitiEngine {
  ...
  @PostConstruct
  public void init() {
    // create Activiti process engine
    this.processEngine = ProcessEngines.getDefaultProcessEngine();
    this.reposService = this.processEngine.getRepositoryService();
    this.identityService = this.processEngine.getIdentityService();
    this.runtimeService = this.processEngine.getRuntimeService();
    this.taskservice = this.processEngine.getTaskService();
    this.formService = this.processEngine.getFormService();
  }
  ...
  /**
    * Retrieves FormProperties for a given task in a process
    * @return list of formProperties for the task
    */
  public List<FormProperty> getTaskFormProperty(String id, String
      activeProcessID) {
    Task task = this.taskService.createTaskQuery().processInstanceId(
        activeProcessID).taskId(id).singleResult();
    TaskFormData taskFormData=this.formService.getTaskFormData(task.getId());
    return taskFormData.getFormProperties();
  }

  /**
    * submits task with the given properties
    */
  @Override
  public void submitTask(String taskID, Map<String, Object> properties) {
    this.formService.submitTaskFormData(taskID, properties);
  }
  ...
}
```

The *initialize* method shows what kind of steps are necessary to get the execution engine running, based on the previously specified configuration. The handling of *FormProperties* is implemented in the *getTaskFormProperty* method. *FormProperties* are simply accessed by selecting the task via the process and task id and then getting the *TaskFormData* from the *FormService*. A list of *FormProperties* is returned, which then can be further evaluated. In the case of this thesis a renderer uses them to build a JavaFX front-end (see Section 4.2.1.1).

4.2.2.2 Artefact Information Coordinator

The framework enables the use of defined Business Artefact information in combination with a BPM engine. In particular, defined artefact information can be provided in addition to the basic input and output of a given task. The *Artefact Information Coordinator* loads and manages this *additional artefact information*.

An OWL structure is used to specify the configuration. This configuration has to be processed first, so that it can be used by the Java-based framework. To accomplish this, a module is implemented, which maps the OWL specification to a Java data structure. The attributes of this data structure are shown in Listing 4.13. The inline javaDocs provide information about the attributes. Basically, the data structure holds a reference to the process or task and to the Business Artefact that should be displayed. In addition, information about other Business Artefact that are influenced by this configuration is stored.

Listing 4.13: List of the attributes of the Context Configuration

```
/**
 * The configConfiguration holds the process or task ID
 * for which this configuration is setup
 */
private String configConfiguration;
/**
 * The contextValue holds the ID of the artefact
 */
private String contextValue;
/**
 * The position (row and column) where this
 * context is placed
 */
private Position pos;
/**
 * The input holds the object which is used in this
 * context block
 */
private Object input;
/**
 * The onProperty stores the property that a
 * listener can register
 */
private String onProperty;
/**
 * The onViewPart holds the value of the view property
 * related to the context
 */
private String onViewPart;
/**
 * The list of relationships store which other context is influenced by this
 * context information
 */
private List<ContextConfiguration> relationships;
```

The configuration is provided in a two-step technique. First, a configuration of the process or task is given, which specifies what kind of *additional context information* should be displayed. It also specifies what other context is influenced by this configuration. The second step involves the configuration of the UI. This involves the front-end implementation, in our case the specific *Artefact View*, of the context as well as the placement on the screen. More information on how the context is provided and processed is given in Chapter 5.

4.2.2.3 Artefact Access Handler

The *Artefact Access Handler* provides an interface for the UI layer to access artefacts from the *Business Repository* layer. It encapsulates the calls to the Business Artefact and provides a single point of entry. All models used in the whole application, including the UI parts like the *Artefact View*, are loaded and retrieved via the *Artefact Access Handler*.

4.2.3 Business Repository

The *Business Repository* holds the elements of the Domain Model as well as corresponding services that are used in the Business Domain. The elements of the Domain Model are the *Business Artefacts* that are used in the business domain and thus in the application. These are the elements that the application operates on. The application uses *IT-Services* to perform computations on Business Artefacts. In some cases the Business Artefacts may contain some logic for computations as well.

As we use OWL as the specification language for our Domain Model and Java as the programming language for our application, we have to generate data structures in Java that correspond to the concepts in the Business Domain. These concepts are translated to Plain Old Java Objects (POJOs), which can be accessed and used by the application. This generation approach also allows us to use common storage facilities and libraries. In this thesis, we decided to use the object-relational mapping framework Hibernate[15] as a link between the data storage facility and the POJOs. Although Hibernate provides the means for a direct mapping of POJOs, either via annotation or XML-configuration, we decided to use Data Access Objects (DAOs) to further abstract the POJOs from the actual database. For the generation of the Java data structures we used Protégé.

Since we specify our Business Domain in OWL, the question arise why we do not directly operate in OWL. After all, there is the possibility to create instance data as well and also the option to persist it. One framework that allows to directly use a triple store is Sesame[16]. It allows processing and handling of RDF data as well as OWL structures. An advantage of this approach would be to work directly with the concept specifications in the Business Domain Model and that changes to the model are directly reflected in the application without the need of a generation step. Furthermore, it would allow us to use the SPARQL query language on the Business Domain Model.

[15] `http://hibernate.org/`
[16] `http://rdf4j.org/`

However, we decided against using a triple store as the data storage facility. This has several reasons: First of all, we would have had to create a data structure, in this case Java classes, anyway, as well as DAOs for them as we want to use them in our FXML-based front-end. Second, the BPM engine uses a common relational database to store its values. Another reason is that we can easily change the actual database implementation via a configuration.

The *IT-Services* should not be confused with services that operate only on one object and manipulate it (like services that are often used with DAOs). In this case, *IT-Services* are services that a BPM uses to perform some kind of activity. Such a service does not necessarily need to operate on any Business Artefact, but could also only trigger an action. An example of an *IT-Service* would be sending a notification to an employee. In our running example, after the supervisor has authorized the vacation request, we first store the request and then notify the employee about the decision.

In BPMN, we specify this task as a *ServiceTask* and have to provide an implementation reference for the service. Listing 4.14 shows how this *ServiceTask* is specified. The *activiti:class* attribute allows for providing a reference to the implementation of the service. The *activiti:field* elements provide the input for this task. The actual value of these inputs is evaluated during runtime. The *activiti:expression* elements are processed and the value of the variables, in this case vacationDateVar, vacationAuthorizedVar and initiator, are evaluated. Many elements of the listing use the *activiti* namespace. This indicates that this method of handling services or inputs is specific to the Activiti execution engine.

Listing 4.14: Notify Employee Service Task in BPMN

```
<serviceTask id="notifyEmployeeServiceTask" name="Notify Employee"
    activiti:class="at.ac.tuwien.ict.services.NotifyEmployeeService">
  <extensionElements>
    <activiti:field name="vacationDate">
      <activiti:expression>${vacationDateVar}</activiti:expression>
    </activiti:field>
    <activiti:field name="vacationAuthorized">
      <activiti:expression>${vacationAuthorizedVar}</activiti:expression>
    </activiti:field>
    <activiti:field name="initiator">
      <activiti:expression>${initiator}</activiti:expression>
    </activiti:field>
  </extensionElements>
</serviceTask>
```

The reference to the service implementation is in this case a Java class that has to implement a specific interface. Listing 4.15 shows the implementation. Basically, Activiti provides a variable *execution* of type *DelegateExecution*, which is passed to the *execute* method of the service. This *execution* variable holds the values that have been passed to the service. The *Expression initiator* attribute of the service corresponds to the *<activiti:field name="initiator">* element of Listing 4.14 and extracts the actual value from the *execution* variable. The same applies for the *vacationDate* and *vacatio-*

nAuthorized attributes, which correspond to the *activiti:field* elements *vacationDate* and *vacationAuthorized*.

Listing 4.15: Notify Employee Service Task in Java

```
public class NotifyUserService implements JavaDelegate {
  private Expression initiator;
  private Expression vacationDate;
  private Expression vacationAuthorized;

  @Override
  public void execute(DelegateExecution execution) throws Exception {
    String date = this.vacationDate.getValue(execution).toString();
    String authorized = (String)this.vacationAuthorized.getValue(execution);
    String initiatorString = this.initiator.getValue(execution).toString();
    // Notifaction logic
    ...
    System.out.println("Vacation Request of " + initiatorString + " at: " +
        date + " has been authorized: " + authorized);
  }
}
```

There are several options how services can be invoked from BPMN and many depend on the execution engine used. In fact, the BPMN standard only describes how WSDL Web-services can be referenced. Activiti supports Web-services as the implementation reference as well, and we have used both techniques in our prototype. In Listing 4.14 the parameters of the service are specified via *activiti:field* elements which can only be processed by Activiti. Specifying the input parameters as BPMN input types is possible as well, but since we only use the Activiti engine, we decided on using the shortcuts provided by Activiti.

4.2.4 Data Storage

The *Data Storage* facility stores the instances of the Business Artefacts. We decided to use the common relational database system H2[17] as our storage facility. The DAOs operate on the table structure of the database and provide an abstraction layer for the database. This enables us to keep the database implementation separated from the application and change it via configuration values in Hibernate.

[17] http://www.h2database.com/html/main.html

CHAPTER 5

Providing Context Information

On top of the architecture for integrating BPMs our approach also aims at providing a flexible and configurable UI for the users. The UI is based on information that the user needs to efficiently work on her tasks and may involve elements that are not directly visible or specified in the BPM. This chapter describes in detail, how this *additional context information* is specified and how a UI is created based on it.

5.1 Providing Additional Context Information for BPM

As indicated above we aim at providing a coherent and flexible UI for users with all information necessary to fulfill their tasks. In Chapter 4 we showed how our software architecture can facilitate such a flexible UI. Still, we have to elaborate on our approach for implementing this in the software architecture.

First, we have to explain how to provide *additional artefact information* for tasks or BPs. *Additional artefact information* can be anything ranging from extra documents to information about other people involved. All this information is supposed to be available in the Business Domain, which our software architecture uses, anyway. Thus, we strive for making it directly available through the UI together with the *Task View* of the related task, rather than requiring the user to switch to another program, open documents externally, or find information manually. So, a specification is necessary that lists all the related *artefact information* for processes or tasks. Based on this specification, the software needs to be able to dynamically load and place views of such information on artefacts and their relations on the UI [35].

It is important to note that the specification of the *additional artefact information* only states the kind of information to be displayed, but not how this information should be presented. Basically the configuration consists of two steps: The specification of the *additional artefact information* and the corresponding visualization through a view. This allows us to separate the specification of the *additional artefact information* from

the actual implementation of the UI. As indicated in the previous chapters, we use JavaFX as the front-end implementation technology for the artefact information. This provides flexibility of the configuration, since it is separated from the actual presentation. Furthermore, the BP designer handles the configuration of *additional artefact information* without having to worry about implementation details at the same time.

Still, the *additional artefact information* and its position on the screen, has to be specified. More precisely, a list of all related pieces of information, which are necessary for a useful UI, have to be defined. In conclusion, the additional context configuration basically consists of the parts shown in Figure 5.1.

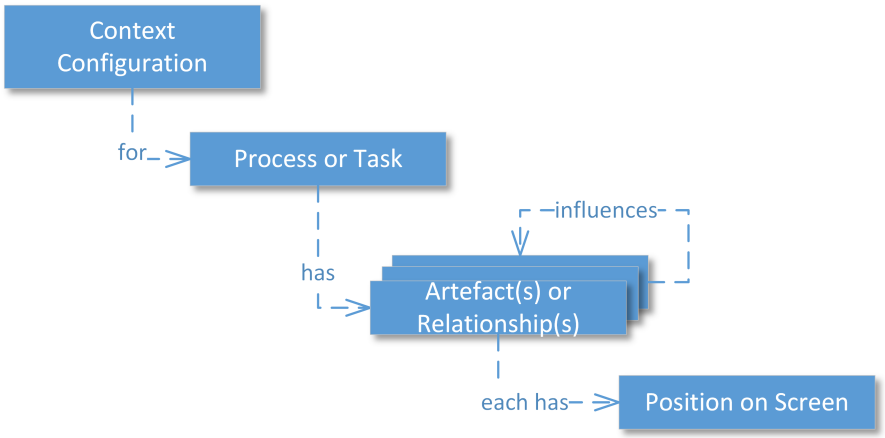

Figure 5.1: Abstract context configuration for additional artefact information

In essence, it specifies for a specific *task or process* the *artefact(s) and relationship(s)* which are to be displayed and defines their *position on the screen*. More precisely, it may specify for a process or task no or possibly several artefacts and/or relationships. For each piece of *additional artefact information*, a position on the screen is defined. For the positioning, we use a simple grid layout, since it provides a flexible and easy definition of where to put parts based on the column and row. Other layout definitions are also possible and only require minor adaptation at the UI layer.

For our running example, let us consider the task *Authorize Vacation Request*, where a supervisor has to authorize a vacation request or not, preferably based on the information directly available on the screen. A simplified configuration could involve a detailed view of the employee along with a list of projects that the employee is assigned to. This amounts to one artefact and one relationship of this artefact. In the user interface, this results in two separate views, one that deals with the employee information and one with a list of the assigned projects.

Figure 5.2 shows schematically how such a configuration is specified. For the *Authorize Vacation Request User Task*, two additional information blocks are assigned: *Employee Artefact* and *Employee->Projects Relationship*. The defined position on the screen for *Employee Artefact* is in the second column of the first row and the one for *(Employee->Projects Relationship)* is in the second column of the second row.

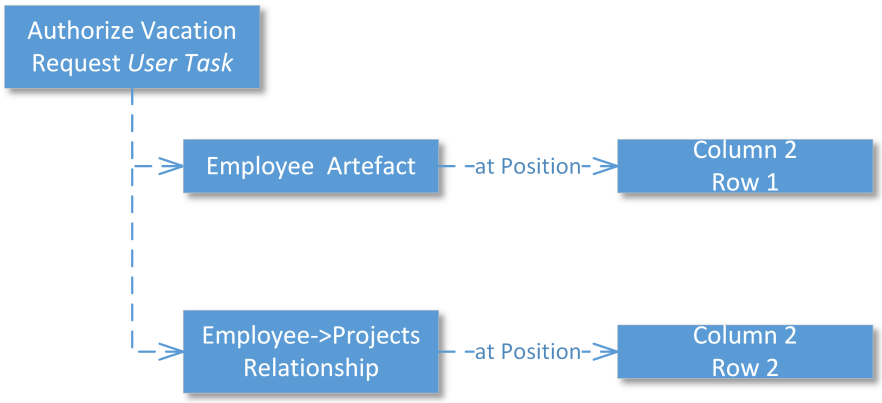

Figure 5.2: Example context configuration for Authorize Vacation Task

In the example above the additional information blocks use elements of the BPM as inputs. This means that the view, that is based on this additional artefact information, visualizes artefacts or relationships of elements that are handled in the BPM. The *Employee Artefact*, for example, provides detailed information on the employee who initiated the request. In this case, the view has to use an element of the BPM as an input and visualize it. Thereby, a connection between the element in the BPM and the context configuration has to be established.

It has to be mentioned that this is not always the case as there are other use-cases as well. For example, a supervisor may want to have a list of all active, not only the ones where the employee is involved with, projects, which she then can search and filter for deadlines overlapping the vacation request. In such a case the view does not need any information from the BPM, but rather uses the Business Domain Model directly. In our architecture and context configuration approach, both use-cases are possible and it only depends on the implementation of the *Artefact View* (compare Section 4.2.1.2).

In addition to the previously described examples, there is another use-case possible. Let us assume that there is a list of employees, with detailed information about them, visualized in a view. If we want to display projects that a specific employee is involved with, we can either display it directly in the detailed view within the list or we display it as a separate view. The first case is simple and the implementation of the view can handle this. The second case is of more interest as we want to display this relationship of an artefact as a separate view. This method is in essence the same that we use for our context configuration of relationships. The only difference is that the input of the view is not an element of the BPM, but rather the selection of an element in a view. In such a case one view influences, by providing the input, another view. With this in mind we can specify the configuration of this context as well through our configuration as illustrated by the *influences* relationship in Figure 5.1.

5.1.1 Configuration Model

There are several options available to specify such a configuration. Considering that we want to enrich BPMs with *additional artefact information*, we decided on using OWL as the language for the context configuration. This enables us to directly reference concepts from the Business Domain, which can also be part of an Enterprise Architecture, as it is also specified in OWL. More precisely, we reference the artefact or relationship concepts that are available from the Business Domain in the configuration. Figure 5.3 shows the ontology for the configuration.

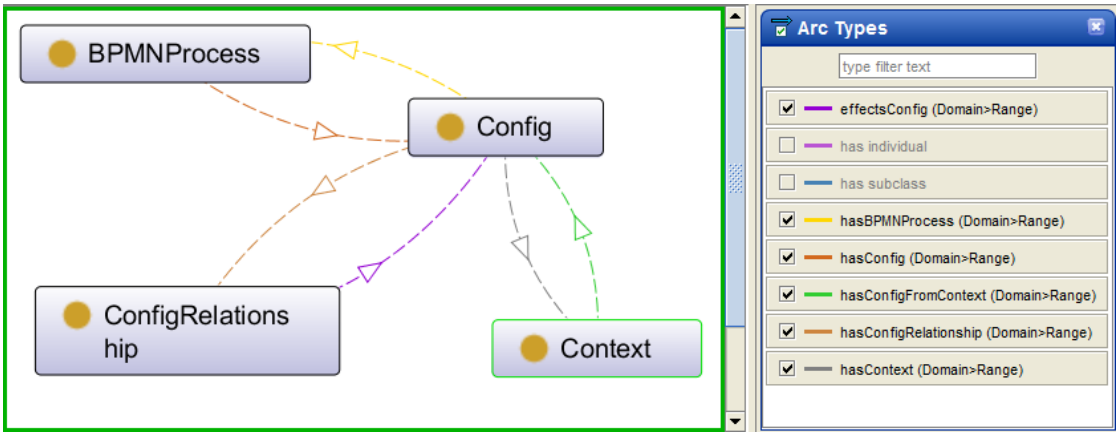

Figure 5.3: Configuration Model in OWL

This ontology introduces several new concepts, namely *BPMNProcess*, *Config*, *Context* and *ConfigRelationship*, which have relationships among each other as well as properties assigned to them. Basically, these concepts provide a model for the abstract context configuration given in Figure 5.1. Each *BPMNProcess* can have multiple relationships with *Config* values and each of them has one *Context* assigned to it. Additionally, the *Config* concept also has a relationship *hasConfigRelationship* which specifies if a specific *Config* concept has an influence, meaning provides an input, on another *Config* concept. There are also properties involved in the ontology that are assigned to their corresponding concepts. The context for example has an annotation *ContextEntityAnnotation*, which allows the specification of the artefacts or relationships through references to the Business Domain. In addition two simple integer data properties *column* and *row* are used to specify where the context should be placed in the UI. The data property *BPMNEntity* provides the reference to the BPMN process and thus specifies which element is used as an input. These are the necessary values that have to be set for a valid context configuration. However, there are additional properties defined that allow to define more precise constraints on views. For example a data property *viewPart* allows to specify if a specific *Config* only influences parts of another *Config* concept. The data properties are not visualized in Figure 5.3.

With these concepts in place the configuration for a BP or task can be created. For this purpose individuals of these concepts are created, which relate to a specific BP or

task given in BPMN. These individuals hold the concrete values of elements in the BPMN process specification. Considering our running example, the context configuration for the *Employee Artefact* creates individuals for the *Context* concept and sets the value for the *ContextEntityAnnotation* annotation to the *Employee* concept of the corresponding Business Domain ontology. In addition, the values for column and row are set to the integer values one and two. The last necessary property to be set is the *BPMNEntity* data property, which is set to *vacationRequestor*. The *vacationRequestor* value corresponds to the element in BPMN that holds the employee that makes the vacation request.

One drawback of this method is, that there can only be references from the OWL configuration to the elements of BPMN as string values. That is, there is no direct reference possible, but rather only references by simple text based values. However, we chose to utilize this method because it allows us to directly reference the Business Domain. In addition, the BP designer who creates the BPMN process is already familiar with the values used and thus can easily create the context configuration.

5.1.2 Integrating Context Configuration into Architecture

The configuration model along with its values, has to be integrated into the architecture of the framework. The *Artefact Information Coordinator* is implemented for this purpose. It loads the configuration and passes it to the UI layer. Figure 5.4 gives an overview of how such a configuration is processed. The *Artefact Information Coordinator* is triggered by the *BPM Engine* as soon as a new task is available and then loads the corresponding artefact configuration. Afterwards, it notifies the *UI Manager*, which displays the artefacts according to their defined position on the screen.

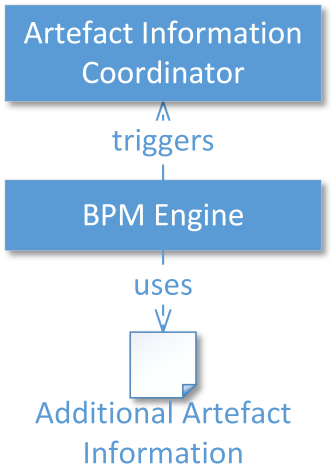

Figure 5.4: Context configuration handling

The *Artefact Information Coordinator* component can also be used as a standalone application to gather information from the configuration model. It uses the information of the configuration model and thus it is also responsible to load and access the context

information. Since the configuration model is provided as an OWL ontology the SPARQL language can be used to query the model. SPARQL queries can be rather complex and thus all necessary information can be gathered via a single query. This makes it very efficient. Listing 5.1 shows the query that loads all configuration values for a specific task or process.

Listing 5.1: SPARQL query for configuration model

```
PREFIX ...
SELECT  ?config ?context ?contextValue ?row ?column
WHERE
  { ?object proreuse:processID %ProcessOrTaskID .
    ?object proreuse:hasConfig ?config .
    ?config proreuse:row ?row .
    ?config proreuse:column ?column .
    ?config proreuse:hasContext ?context .
    ?context proreuse:implementationRef ?contextValue
  }
```

In the query several values (the *?context* which holds the Business Artefact, *?row* and *?column* etc.) are selected. This is accomplished by querying the configuration model for a specific task or process via the variable *%ProcessOrTaskID*. In successive steps the relationships of the configuration model are queried and the necessary values, such as *row* and *column*, are extracted. With these values the configuration for one specific process or task is complete.

However, we are still missing the relationships that exist in between different context blocks. To access these relationships another SPARQL query is implemented, which checks for each context block if there is an influence on another. The variable *%ConfigBlock* is the value of the current context block to be queried. Listing 5.2 shows the SPARQL query for these relationships.

Listing 5.2: SPARQL query for configuration relationships

```
PREFIX ...
SELECT  ?config ?contextValue ?row ?column ?property ?viewPart
WHERE
  { ?configStart rdf:type proreuse:Config
    FILTER regex(str(?configStart), %ConfigBlock)
    ?configStart proreuse:hasConfigRelationship ?configRelationship
    OPTIONAL
      { ?configRelationship proreuse:onProperty ?property}
    OPTIONAL
      { ?configRelationship proreuse:onViewPart ?viewPart}
    ?configRelationship proreuse:effectsConfig ?config .
    ?config proreuse:row ?row .
    ?config proreuse:column ?column .
    ?config proreuse:hasContext ?context .
    ?context proreuse:implementationRef ?contextValue.
  }
```

With these queries in place the configuration for a BPMN process or a corresponding task can be loaded. The description for integrating the configuration above, is a somehow simplified description as there are other parts involved as well. Figure 5.5 provides a more comprehensive look on how the configuration is processed and used in the framework. It shows how elements of BPMN are related to the configuration model in OWL and how they are visualized in the UI.

Let us consider the running example and focus on the *Authorize Vacation Request User Task*. In this task a supervisor has to grant or deny the vacation request. We have already explained what kind of additional artefact information could be of interest and here we want to focus on one: a detailed view of the employee. The configuration for this simplified example states that the task *Authorize Vacation Request User Task* has one additional artefact information *Employee Artefact*. This *Employee Artefact* should visualize the vacation requester who has initialized the vacation request. For this purpose the *Employee Artefact* takes an input with the value *vacationRequester*. This value is the element of the BPMN process which holds the employee that made the request. Thus it references the corresponding *itemDefintion* in BPMN. As described above, this is only a reference by a string value and the values of both of the elements must be identical. This relates the configuration to the actual BPMN process.

The BPM execution engine executes the process and when the *Authorize Vacation Request User Task* is reached, it *triggers* the *Artefact Information Coordinator* which loads the configuration for this task. The *Artefact Information Coordinator* first executes the SPARQL queries and then takes the value of the context input, in this case *vacationRequester*, and retrieves the actual value from the BPMN process via the BPM execution engine. Afterwards, the context configuration, including the value of the input *vacationRequester*, is passed to the *UI Manager*. The *UI Manager* loads the corresponding *Artefact View*, in this case the *Employee Artefact View* , and places it on the screen. Each *Artefact View* has a model which relates to Business Artefacts from the *Business Repository*. This is also the type of value that the *Artefact View* expects to visualize. In this case the *Employee Artefact View* expects an *Employee Artefact*. As Figure 5.5 illustrates the type of the context input has to be identical with the type that the *Artefact View* expects. It would, for example, not be possible to load an *Employee Artefact View* as context and provide a *Project Artefact* as input.

5.2 Context Switches in Business Process Management Systems

Even with the context configuration described above, there is still the problem of context switches for each isolated task execution. Each task that a user is supposed to work on belongs to a specific BP, and each BP has a different state and context. Many applications have an “inbox”-like system in place, where all available tasks are stored, but without taking into account to which BP they belong to. So, the applications may constantly switch between different BPs. Even in case there are still tasks available from the currently active BP, the next task to be executed may belong to a different BP.

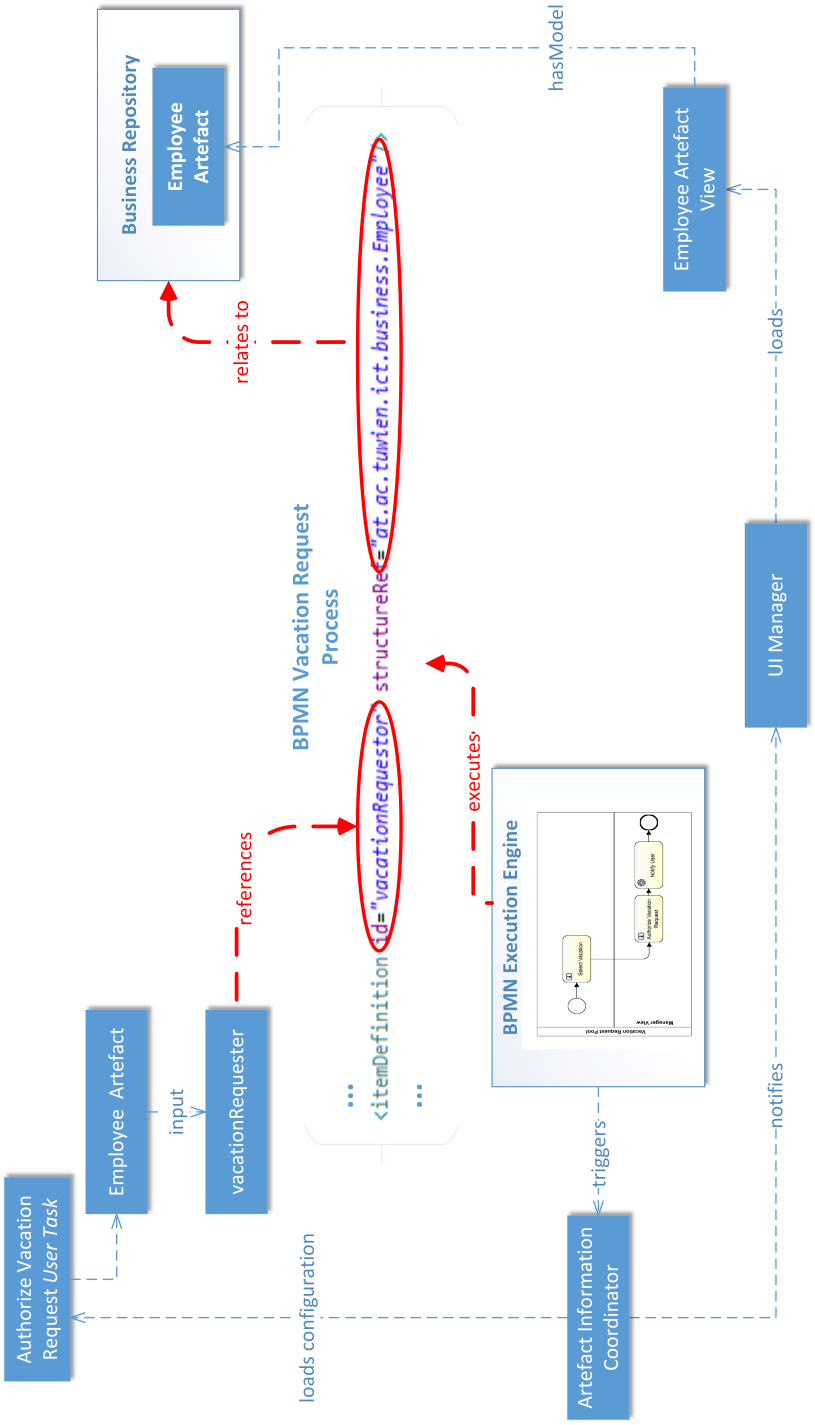

Figure 5.5: Overview of context configuration and relationship to the framework

Also if there are multiple instances of the same BP, each process enactment still has its own state based on the artefacts that are used and their inner state. If a user has to constantly switch between these process enactments, it is difficult to distinguish between them, even with additional artefact information displayed (based on [35]).

So, we decided to have the application stay within a BP enactment, as long as there are tasks available, who a user can work on. Another reason would be that many tasks are supposed to be executed consecutively anyway. This is, however, also not a perfect solution as some tasks with higher priority could become available. A notification system on incoming tasks can be implemented in future work.

5.3 Aligning the User Interface

The context configuration provides the means for specifying additional artefact information for tasks or processes. However, as stated in Chapter 4, the context configuration is implemented in a two-step technique. First, as described in the previous section, the type of context is specified. At this point the specification provides no information about the actual UI implementation. In the second step the implementation of the UI for a specific context is defined. With this method the specification and implementation is separated. Building on top of the context specification given in Section 5.1 we have to align the UI to it.

The *Artefact Information Coordinator* loads the context and passes it to the *UI Manager*. This information, however, can not directly be visualized as it is only a value of a concept from the Business Domain. For this purpose a second configuration is put in place, which holds for each Business Artefact a corresponding UI visualization. As we used JavaFX in this framework for the UI implementation, we can simply provide a FXML file for each Business Artefact or relationship. Figure 5.6 illustrates this.

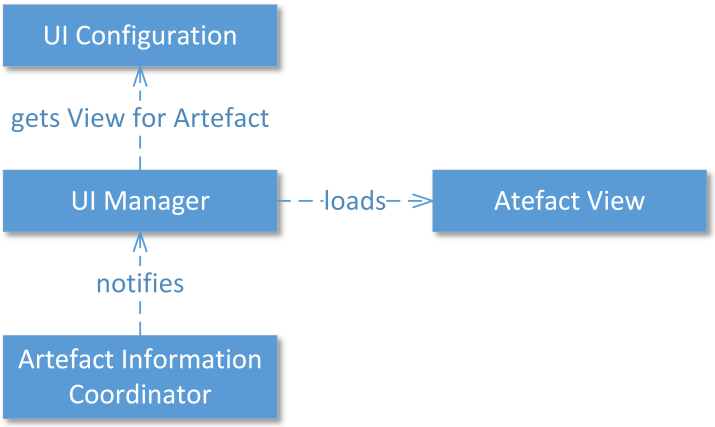

Figure 5.6: Loading context via the *UI Manager*

Let us consider our running example again. The configuration of the *Authorize Vacation Request User Task* states that an *Employee Artefact* should be visualized in the

UI. The *UI Manager* receives this information from the *Artefact Information Coordinator* and looks up what kind of implementation is available for this artefact. It finds a reference to an FXML file, in this case *EmployeeDetailView.fxml*, and then loads the corresponding view. An excerpt of the source code from the *UI Manager* is shown in Listing 5.3.

Listing 5.3: Loading context via the *UI Manager*

```
public void loadContext() {
  List<ContextConfiguration> contexts = this.coordinator.
      queryContextForProcessID(processID);
  ...
  for (ContextConfiguration context : contexts) {
    IController<?> newController = this.loadView(context.getContextValue());
    ...
    this.processview.placeView((Parent)controller.getView(), context.
        getColumn() - 1, context.getRow() - 1);
  }
  this.addRelationships(contexts);
}
...
public IController<?> loadView(String contextArtefact, Object... arguments) {
  ...
  String viewName = UIConfig.getViewForContext(contextArtefact);
  return this.fxmlLoader.load(viewName + ".fxml", model, arguments);
}
```

The relationships between *Artefact Views* are established after all *Artefact Views* have been placed in the UI. The *UI Manager* processes one context after another and checks if a relationship to another *Artefact View* is defined. If so, it retrieves the view and adds a *propertyListener* to it that reacts to changes. This listener notifies the view of any changes, via selection for example, and the *Artefact View* then updates its view. Listin 5.4 shows an excerpt of the relevant source code.

Listing 5.4: Loading context relationships via the *UI Manager*

```
private void addRelationships(List<ContextConfiguration> contexts) {
  for (ContextConfiguration context : contexts) {
    if (context.hasRelationships()) {
      IController<?> controller = this.viewPositionToController.get(context.
          getColumn() + "_" + context.getRow());
      ...
      for (ContextConfiguration relContext : context.getRelationships()) {
        IController<?> relController = this.viewPositionToController.get(
            relContext.getColumn() + "_" + relContext.getRow());
        ...
        controller.addChangeListenerProperty(relContext.getOnViewPart(),
            relController, relContext.getOnProperty());
      }
    }
  }
}
```

In fact, the *UI Manager* coordinates both, the *Task Views* as triggered by the *BPM Engine* and the corresponding *Artefact Views* from the configuration which are included dynamically. In this way, using the configuration in combination with our software architecture enables the application to provide a custom UI for each task, which provides additional artefact information in the context of the enacted process. By doing so, we provide a basis for usable UIs, since tasks can be presented with the context information required, and within the context of their process rather than in isolation. This actually leads to a process view for the user, rather than a pure task view.

5.4 Alternative Approaches for Context Specification

In our current approach, we provide separate configuration of additional artefacts to be displayed, which is assigned to the related task. As an alternative, this context information could be directly included in the specification of the corresponding task. BPMN 2.0 has an extension mechanism, that allows adding custom elements or properties to the specification of a task. It could be used to include the additional artefact information directly as a property of the task specification. While this approach would avoid our separate configuration information, it makes it more difficult to use the same process in different environments where the tasks may need different context information. Creating several copies of the same process with different context information would entail maintenance problems [35].

CHAPTER 6

A BPMN Framework with Additional Context Information

In this chapter the final feasibility prototype is presented. In addition, an overview of the technologies involved and how they interact with each other is given. Since the framework is a feasibility prototype, it has some limitations, which are also discussed here. Finally, a discussion on possible future work is provided and how the framework could be extended.

6.1 Feasibility Prototype UI

The feasibility prototype aims at providing a software application driven by BPMs, and at providing a useful UI. More precisely, the feasibility prototype fulfills all requirements which have been identified (compare Section 4.1). In the previous chapters we described the single parts, e.g., *Artefact Views*, *Task Views* etc., in detail and we focus in this section on the overall GUI of the framework.

Figure 6.1 shows the initial screen of the feasibility prototype. The overall GUI of the framework provides a menu bar and a status bar. The menu bar provides standard operations such as closing the application or switching between screens as well as operations that are allowed on the BPMs. Initially, the GUI provides an overview of the available processes as well as the currently running processes. The selected process is visualized in the middle and the list on the right hand side allows the user to start working on available tasks. This screen is kept simple as the main focus is not on managing processes, but rather on providing an adequate GUI to work on them. There is also more detailed information shown in case the user hovers the mouse cursor over elements.

A new BPM can be added dynamically via an import feature at runtime. Furthermore, the framework also provides the option to view a process history. Features concerning the BPM execution engine can be activated or deactivated based on its configuration. Activiti already provides the means for this.

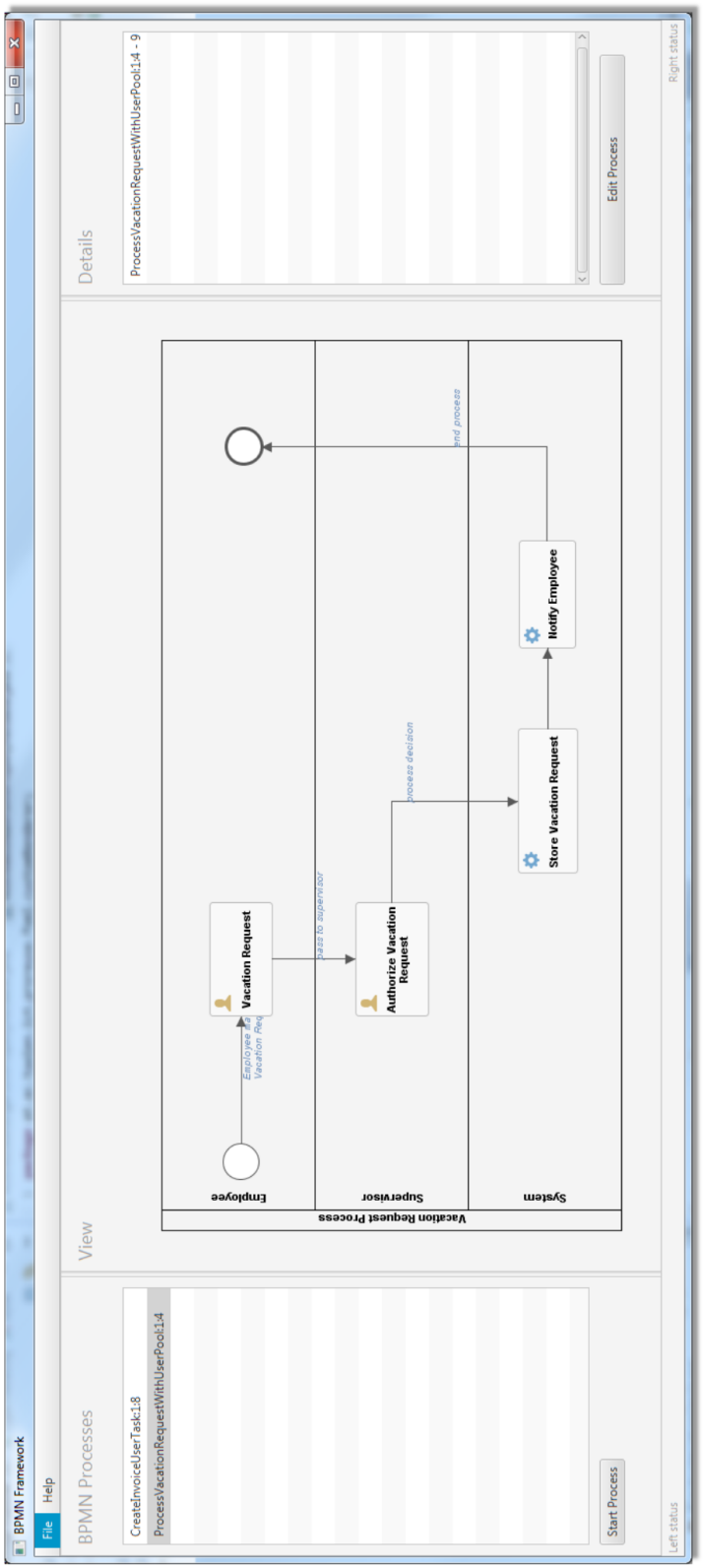

Figure 6.1: Initial screen of the feasibility prototype GUI

When the user selects a process and starts working on it, the initial screen is replaced with a *process view* with detailed information about the currently active task. Figure 6.2 shows an example of such a screen. This *process view* contains the view for the currently active task (*Task View*), which is automatically generated based on its properties. In addition, the context consisting of *Artefact Views* is visualized as well. These *Artefact Views* are dynamically loaded and positioned on the screen based on the context configuration.

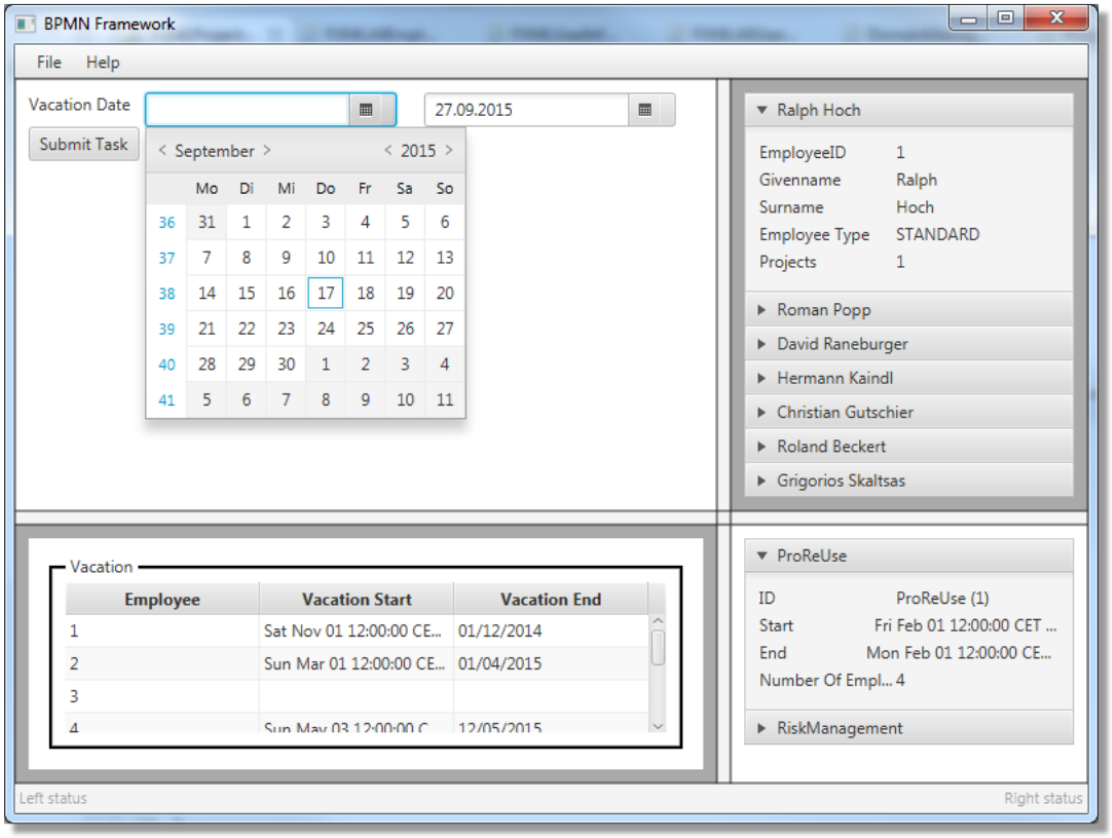

Figure 6.2: Process view with *Task Views* and additional *Artefact Views*

The figure shows a screen of the *Vacation Request* process. More precisely, it shows the first *User Task* of this process where an employee makes her request. The *Task View* in this case is simple and only contains two fields with a *DatePicker* widget and is always placed in the upper left corner. This view is automatically generated from the properties assigned to the *User Task*.

Additionally, there are *Artefact Views* placed around the *Task View*. They are placed on a grid layout based on the context configuration. In this example, an *Artefact View* with a list of employees is shown, alongside with a list of projects that this employee works on. The list of employees is placed in the upper right corner and, upon selecting an employee, provides the input for the *EmployeeProjects Artefact View* in the lower right

corner. Each selection of an employee changes the content of the view that displays the projects. This is all dynamically created and configured through the context configuration.

These two figures give a basic overview of the framework's capabilities. There are, of course, other facets as well, but the main focal point is enriching the BPM with additional artefact information and providing a useful and dynamic GUI.

6.2 Implementation Technologies

We organized the source code of the implementation in several modules, which are organized in Maven[1]. Maven also serves us as a dependency management tool. The definition of the modules is shown in Figure 6.3.

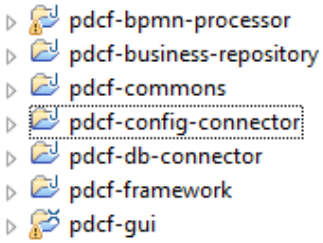

Figure 6.3: Maven modules of the framework

For our implementation, we chose Java as the implementation language and Activiti as the BPMN execution engine. The GUI is based on JavaFX, which allows us to use a descriptive specification of UI parts that can be easily integrated into other views. For this purpose we used FXML. Views of single Business Artefacts are supported as well as complex views for providing information about several Business Artefacts together. As a data storage facility we chose the H2 database and provided an extra abstraction layer via the object-relational mapping (ORM) tool Hibernate. The actual Business Domain Model is given in OWL, and we use a generation mechanism to create representation classes in Java, which are then used with Hibernate and the H2 database. The context configuration references the Business Domain Model and is also represented in OWL. Figure 6.4 gives an overview of these parts and how they interact with each other.

6.3 Discussion and Future Work

This discussion of future work is based on the one in [35].

As mentioned above, we currently rely on a single machine environment. That is, we run all parts of our framework, including the database, on a single machine. Typically, business applications are implemented in a distributed environment and several users work on it simultaneously. Since we already apply a 4-layer architecture, it would be possible to extend it to a 4-tier architecture. Commonly used in Enterprise applications

[1] `https://maven.apache.org/`

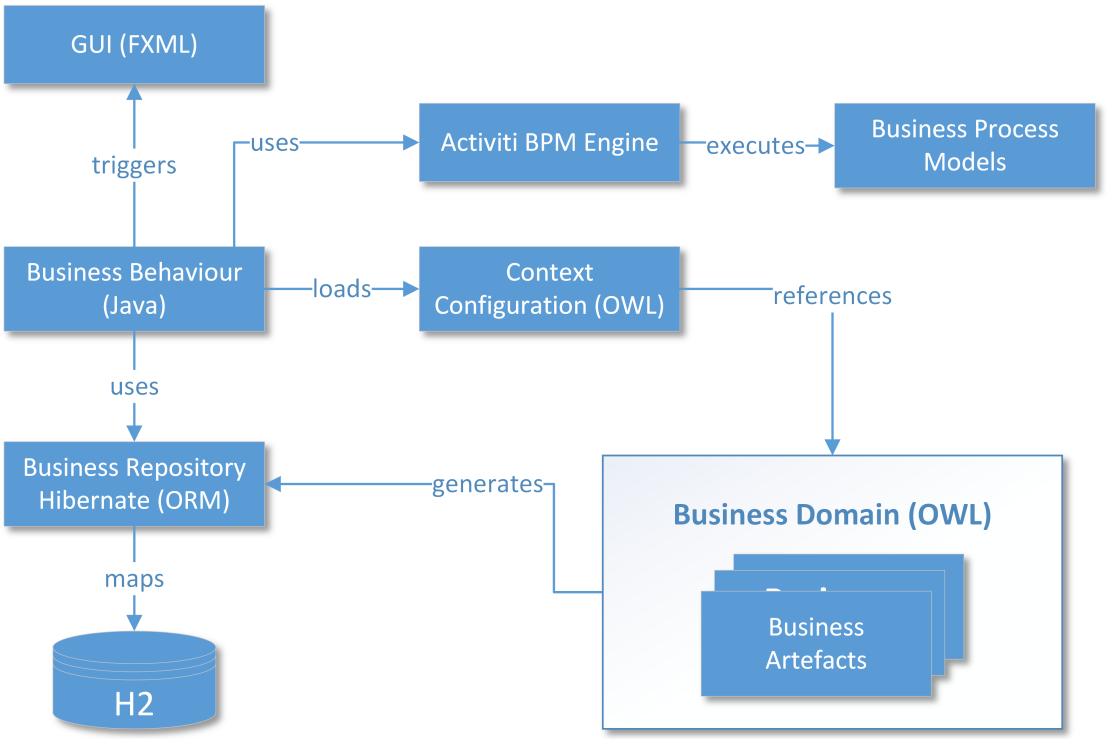

Figure 6.4: Overview of the Implementation

are various middleware systems. In future work, we could extend our application and include, for example, the JBoss Fuse[2] Enterprise Service Bus (ESB) or the Apache Camel[3] project to accomplish distribution. Lee et al. [50] discuss how BPMs can be used in a distributed and collaborative environment.

Currently, we rely on pre-defined views of artefacts, which are specified in the FXML language. An automatic generation of *Artefact Views* based on the artefacts included could be accomplished based on techniques from UI generation, making use of the attribute definitions of these artefacts [41, 65, 67, 81].

As mentioned in the previous chapters, automatic generation of Business Artefacts from Enterprise Architecture entities is desirable. However, this requires that after any change to the Enterprise Architecture, a new generation cycle has to be executed. Otherwise, this will lead to inconsistencies between the Enterprise Architecture and the Business Artefacts used in the software. In effect, it would deteriorate their alignment.

While our approach improves on the pure task-based views, it is still triggered by tasks, which is inherent to the BPMN engine involved. This task-to-task approach should become less prominent for software to be used in practice. So, in future work, our approach should be further extended by embedding information on currently available

[2] `https://www.jboss.org/products/fuse/overview/`
[3] `http://camel.apache.org/`

tasks and actions in the UI.

During the execution of BPMN BPs, the software needs to access and manipulate business artefacts. Since BPMN does not provide the means to have a comprehensive definition of Business Artefacts, it is necessary that the execution engine knows how to work with them. In most cases, the execution engine uses simple data structures (database objects, Java classes, etc.). This is a common problem of process modeling languages, including BPMN, as they focus on the process and flow of activities rather than the definition of artefacts. Often they only provide the means to reference data structures that define the artefacts. That is, we cannot directly use the entities as specified in the Enterprise Architecture, but need a representation that the execution engine can interpret. In our approach, we generated a Java class representation of the Business Artefacts.

Generally, there is also a possibility to extend an execution engine and to implement support for entities as defined in an Enterprise Architecture. This would allow us to directly access the entities as given in the Enterprise Architecture without prior generation of another representation. However, porting such models to another execution engine without such an extension would become very difficult.

Currently we use Protégé to create the configuration of the context. As Protégé is a tool that deals with ontologies in a general manner, it does not provide a specialized UI for our context creation. A tool assisted context configuration via an application or an Eclipse[4] plugin would be preferable. Furthermore, this tool could also provide an option to scan existing BPMN specifications and provide a selection mechanism for element identifiers.

[4] `https://eclipse.org/`

CHAPTER 7

Conclusion

The goal of this thesis was to provide an architecture for a software application which uses models of BPs as the driving force of the business logic and provides a dynamic and useful UI to the user. The main focal point is enriching the BPMs with additional context information provided by Business Artefacts and present it to the user in a meaningful way. The key questions addressed are:

1. What sort of features are required to provide a satisfactory process-centric software application?

2. How can a BP execution engine be integrated into a software application? What kind of prerequisites does this impose on the software architecture?

3. How is it possible to align the UI based on Business Artefacts with the BP?

We used a 4-layer architecture including a *Business Repository* layer. BPMN 2.0 is used as a modeling language for BPs and an execution engine is integrated in our architecture. This execution engine uses BPMs to drive the application and thus the business logic is provided by the models of the BPs themselves.

A process-centric software has its focus on the process rather than the single tasks or Business Artefacts. Using BPMs provides the foundation for such a software application. However, typical applications using BPMN do not account for the context that a BPM is enacted in. Considering the GUI and the user who works with it, it is especially of interest to have additional information on the context available. Presenting this information to the user is essential for a useful GUI.

In our approach, we chose to present additional context information in the form of the Business Artefacts involved in a business. These Business Artefacts are part of the Business Domain and specified through models in the Enterprise Architecture. We used them to provide so-called *Artefact Views* to the user, which can be placed as desired

in the GUI. An additional context configuration is provided and dynamically processed during runtime.

Based on the feasibility prototype, there are still possibilities for future work. The 4-layer architecture could be extended to a 4-tier architecture to support distribution as it is common in enterprise applications. In our current approach, we use pre-defined *Artefact Views*, but a more dynamic one with UI generation could be developed. Using models from an Enterprise Architecture requires an additional generation step to create resources in the programming language that the application can use. A possibility would be to directly reference the model without this generation step.

In summary, we enriched BPMs with additional artefact information provided through Business Artefacts, and created the means for a useful GUI. The software architecture uses a BPMN execution engine to control the business logic and uses the configuration of the additional artefact information to dynamically create an UI.

List of Figures

Listings

Bibliography

[1] OWL-based Web Service Ontology Version 1.2. http://www.daml.org/services/owl-s/1.2/, March 2004.

[2] SPARQL Query Language for RDF. Technical report, World Wide Web Consortium, Jan. 2008.

[3] D. Allemang, I. Polikoff, and R. Hodgson. Enterprise Architecture Reference Modeling in OWL/RDF. In Y. Gil, E. Motta, V. Benjamins, and M. Musen, editors, *The Semantic Web — ISWC 2005*, volume 3729 of *Lecture Notes in Computer Science*, pages 844–857. Springer Berlin Heidelberg, 2005.

[4] T. Allweyer. *BPMN 2.0: Introduction to the Standard for Business Process Modeling*. Books on Demand, 2010.

[5] G. Antunes, M. Bakhshandeh, R. Mayer, J. Borbinha, and A. Caetano. Using ontologies for enterprise architecture integration and analysis. *Complex Systems Informatics and Modeling Quarterly*, (1):1–23, 2014.

[6] M. Aslam, S. Auer, J. Shen, and M. Herrmann. Expressing Business Process Models as OWL-S Ontologies. In J. Eder and S. Dustdar, editors, *Business Process Management Workshops*, volume 4103 of *Lecture Notes in Computer Science*, pages 400–415. Springer Berlin Heidelberg, 2006.

[7] I. Bajwa, A. Samad, S. Mumtaz, R. Kazmi, and A. Choudhary. BPM Meeting with SOA: A Customized Solution for Small Business Enterprises. In *Information Management and Engineering, 2009. ICIME '09. International Conference on*, pages 677–682, April 2009.

[8] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, third edition, 2012.

[9] K. H. Bennett and V. T. Rajlich. Software Maintenance and Evolution: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM.

[10] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[11] K. Bhattacharya, C. Gerede, R. Hull, R. Liu, and J. Su. Towards Formal Analysis of Artifact-Centric Business Process Models. In G. Alonso, P. Dadam, and M. Rosemann, editors, *Business Process Management*, volume 4714 of *Lecture Notes in Computer Science*, pages 288–304. Springer Berlin Heidelberg, 2007.

[12] R. Braun, H. Schlieter, M. Burwitz, and W. Esswein. BPMN4CP: Design and implementation of a BPMN extension for clinical pathways. In *Bioinformatics and Biomedicine (BIBM), 2014 IEEE International Conference on*, pages 9–16, Nov 2014.

[13] K. S. Candan, H. Liu, and R. Suvarna. Resource Description Framework: Metadata and Its Applications. *SIGKDD Explor. Newsl.*, 3(1):6–19, July 2001.

[14] T. Çelik, I. Hickson, H. W. Lie, and B. Bos. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. W3C Recommendation, W3C, June 2011. http://www.w3.org/TR/2011/REC-CSS2-20110607.

[15] C. M. Chiao, V. Künzle, K. Andrews, and M. Reichert. A Tool for Supporting Object-Aware Processes. In *IEEE 18th Int'l Distributed Object Computing Conference - Workshops and Demonstrations (EDOCW 2014)*, pages 410–413. IEEE Computer Society Press, September 2014.

[16] M. Chinosi and A. Trombetta. BPMN: An Introduction to the Standard. *Comput. Stand. Interfaces*, 34(1):124–134, Jan. 2012.

[17] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Service Definition Language (WSDL). Technical report, Mar. 2001.

[18] D. Cohn and R. Hull. Business Artifacts: A Data-centric Approach to Modeling Business Operations and Processes. *IEEE Data Eng. Bull.*, 32(3):3–9, 2009.

[19] T. H. Davenport. *Process Innovation: Reengineering Work Through Information Technology.* Harvard Business School Press, Boston, MA, USA, 1993.

[20] J. de la Vara, R. Ali, F. Dalpiaz, J. Sánchez, and P. Giorgini. Business processes contextualisation via context analysis. In J. Parsons, M. Saeki, P. Shoval, C. Woo, and Y. Wand, editors, *Conceptual Modeling âĂŞ ER 2010*, volume 6412 of *Lecture Notes in Computer Science*, pages 471–476. Springer Berlin Heidelberg, 2010.

[21] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[22] M. Fiammante. *Dynamic SOA and BPM: Best Practices for Business Process Management and SOA Agility.* IBM Press, 1st edition, 2009.

[23] M. Fowler. *Patterns of Enterprise Application Architecture.* A Martin Fowler signature book. Addison-Wesley, 2003.

[24] J. Freund and B. Rücker. *Real-life BPMN: Using BPMN 2.0 to Analyze, Improve, and Automate Processes in Your Company.* Camunda, 2012.

[25] M. Geiger, S. Harrer, J. Lenhard, M. Casar, A. Vorndran, and G. Wirtz. BPMN Conformance in Open Source Engines. In *2015 IEEE Symposium on Service-Oriented System Engineering, SOSE 2015, San Francisco Bay, CA, USA, March 30 - April 3, 2015*, pages 21–30. IEEE, 2015.

[26] M. Geiger and G. Wirtz. BPMN 2.0 Serialization - Standard Compliance Issues and Evaluation of Modeling Tools. In R. Jung and M. Reichert, editors, *Enterprise Modelling and Information Systems Architectures: Proceedings of the 5th International Workshop on Enterprise Modelling and Information Systems Architectures, EMISA 2013, St. Gallen, Switzerland, September 5-6, 2013*, volume 222 of *LNI*, pages 177–190. GI, 2013.

[27] M. Geiger and G. Wirtz. Detecting Interoperability and Correctness Issues in BPMN 2.0 Process Models. In O. Kopp and N. Lohmann, editors, *Proceedings of the 5th Central-European Workshop on Services and their Composition, Rostock, Germany, February 21-22, 2013*, volume 1029 of *CEUR Workshop Proceedings*, pages 41–44. CEUR-WS.org, 2013.

[28] D. Georgakopoulos and M. P. Papazoglou. *Service-Oriented Computing.* The MIT Press, 2008.

[29] S. Goderis. *On the separation of user interface concerns: A Programmer's Perspective on the Modularisation of User Interface Code:.* Asp / Vubpress / Upa, 2008.

[30] W. W. Group. Web Services Glossary, Feb. 2004.

[31] T. R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowl. Acquis.*, 5(2):199–220, June 1993.

[32] C. Gutschier. How to execute parts of BPMN. Master's thesis, Vienna University of Technology, 2014.

[33] I. Herraiz, D. Rodriguez, G. Robles, and J. M. Gonzalez-Barahona. The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review. *ACM Comput. Surv.*, 46(2):28:1–28:28, Dec. 2013.

[34] R. Hoch, H. Kaindl, R. Popp, D. Ertl, and H. Horacek. Semantic Service Specification for V&V of Service Composition and Business Processes. In *System Sciences (HICSS), 2015 48th Hawaii International Conference on*, pages 1370–1379. IEEE, 2015.

[35] R. Hoch, H. Kaindl, R. Popp, and C. Zeidler. Aligning Architectures of Business and Software: Software Driven by Business Process Models and its User Interface. In *Proceedings of the 2016 49th Hawaii International Conference on System Sciences*, HICSS '16. IEEE Computer Society, 2016. (to appear).

[36] J. Hohwiller and D. Schlegel. Integration of UI Services into SOA Based BPM Applications. In W. Abramowicz, L. Maciaszek, and K. Wecel, editors, *Business Information Systems Workshops*, volume 97 of *Lecture Notes in Business Information Processing*, pages 53–64. Springer Berlin Heidelberg, 2011.

[37] R. Hull. Artifact-Centric Business Process Models: Brief Survey of Research Results and Challenges. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems: OTM 2008*, volume 5332 of *Lecture Notes in Computer Science*, pages 1152–1163. Springer Berlin Heidelberg, 2008.

[38] F. Kamoun. A Roadmap Towards the Convergence of Business Process Management and Service Oriented Architecture. *Ubiquity*, 2007(April), Apr. 2007.

[39] F. Kamoun. The Convergence of Business Process Management and Service Oriented Architecture. *Ubiquity*, 2007(June), June 2007.

[40] A. Karagkasidis. Developing GUI Applications: Architectural Patterns Revisited. In *EuroPLoP*, 2008.

[41] S. Kavaldjian, J. Falb, and H. Kaindl. Fully automatic content presentation specific to intentions. In *Proceedings of the IUI'09 Workshop on Model Driven Development of Advanced User Interfaces*, 2009.

[42] H. Knublauch, D. Oberle, P. Tetlow, and E. Wallace. A semantic web primer for object-oriented software developers. *W3C Working Group Note*, 9, 2006.

[43] R. K. Ko, S. S. Lee, and E. W. Lee. Business Process Management (BPM) standards: A survey. *Business Process Management journal*, 15(5), 2009.

[44] G. E. Krasner and S. T. Pope. A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, Aug. 1988.

[45] V. Künzle and M. Reichert. PHILharmonicFlows: towards a framework for object-aware process management. *Journal of Software Maintenance*, 23(4):205–244, 2011.

[46] L. Kuzniarz and M. Staron. Generating Domain Models from Ontologies. In Z. BellahsÃĺne, D. Patel, and C. Rolland, editors, *Object-Oriented Information Systems*, volume 2425 of *Lecture Notes in Computer Science*, pages 160–166. Springer Berlin Heidelberg, 2002.

[47] M. Lankhorst. *Enterprise Architecture at Work – Modelling, Communication and Analysis*. Springer, Berlin, Heidelberg, 2009.

[48] C. Larman. *Applying UML and Patterns*. Prentice Hall, third edition, 2005.

[49] E. W. Lee, R. K. L. Ko, and S. G. Lee. Business-OWL (BOWL) - A Hierarchical Task Network Ontology for Dynamic Business Process Decomposition and Formulation. *IEEE Transactions on Services Computing*, 5(2):246–259, 2012.

[50] J. Y. Lee, S. Lee, K. Kim, H. Kim, and C.-H. Kim. Process-centric Engineering Web Services in a Distributed and Collaborative Environment. *Comput. Ind. Eng.*, 51(2):297–308, Oct. 2006.

[51] M. M. Lehman. Laws of Software Evolution Revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, EWSPT '96, pages 108–124, London, UK, UK, 1996. Springer-Verlag.

[52] V. Liptchinsky, R. Khazankin, H.-L. Truong, and S. Dustdar. A Novel Approach to Modeling Context-Aware and Social Collaboration Processes. In J. Ralyté, X. Franch, S. Brinkkemper, and S. Wrycza, editors, *Advanced Information Systems Engineering*, volume 7328 of *Lecture Notes in Computer Science*, pages 565–580. Springer Berlin Heidelberg, 2012.

[53] J. Mangler, E. Schikuta, and C. Witzany. Quo vadis interface definition languages? Towards a interface definition language for RESTful services. In *Service-Oriented Computing and Applications (SOCA), 2009 IEEE International Conference on*, pages 1–4, Jan 2009.

[54] F. Manola and E. Miller, editors. *RDF Primer*. W3C Recommendation. World Wide Web Consortium, Feb. 2004.

[55] A. Meyer and M. Weske. Activity-Centric and Artifact-Centric Process Model Roundtrip. In N. Lohmann, M. Song, and P. Wohed, editors, *Business Process Management Workshops*, volume 171 of *Lecture Notes in Business Information Processing*, pages 167–181. Springer International Publishing, 2014.

[56] H. Mili, G. Tremblay, G. B. Jaoude, E. Lefebvre, L. Elabed, and G. E. Boussaidi. Business Process Modeling Languages: Sorting Through the Alphabet Soup. *ACM Comput. Surv.*, 43(1):4:1–4:56, Dec. 2010.

[57] M. A. Musen. Domain Ontologies in Software Engineering: Use of Protégé with the EON Architecture, 1998.

[58] A. Nigam and N. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.

[59] OASIS. *Business Process Execution Language 2.0 (WS-BPEL 2.0)*, 2007.

[60] P. Oldfield. Domain modelling. 2002.

[61] A. Olivé. *Conceptual Modeling of Information Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[62] T. O. M. G. (OMG). Business Process Model and Notation (BPMN). http://www.omg.org/spec/BPMN/2.0, Jan. 2011. [Online; accessed 03-July-2015].

[63] C. Ouvans, M. Dumas, A. ter Hofstede, and W. van der Aalst. From BPMN Process Models to BPEL Web Services. In *Web Services, 2006. ICWS '06. International Conference on*, pages 285–292, Sept 2006.

[64] C. Ouyang, M. Dumas, W. M. P. V. D. Aalst, A. H. M. T. Hofstede, and J. Mendling. From Business Process Models to Process-oriented Software Systems. *ACM Trans. Softw. Eng. Methodol.*, 19(1):2:1–2:37, Aug. 2009.

[65] F. Paternò, C. Santoro, and L. D. Spano. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. Comput.-Hum. Interact.*, 16:19:1–19:30, November 2009.

[66] C. Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, Oct. 2003.

[67] R. Popp, D. Raneburger, and H. Kaindl. Tool Support for Automated Multi-device GUI Generation from Discourse-based Communication Models. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*, New York, NY, USA, 2013. ACM.

[68] M. Potel. MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java. Technical report, Taligent Inc., 1996.

[69] T. Rademakers. *Activiti in Action: Executable business processes in BPMN 2.0.* Manning Publications Co., 2012.

[70] V. Rajlich. Software Evolution and Maintenance. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 133–144, New York, NY, USA, 2014. ACM.

[71] T. Reenskaug. MVC XEROX PARC 1978-79. http://heim.ifi.uio.no/ trygver/themes/mvc/mvc-index.html.

[72] M. Rosemann, J. C. Recker, and C. Flender. Contextualisation of business processes. *International Journal of Business Process Integration and Management*, 3(1):47–60, July 2008.

[73] O. Saidani, C. Rolland, and S. Nurcan. Towards a Generic Context Model for BPM. In *48th Hawaii International Conference on System Sciences, HICSS 2015, Kauai, Hawaii, USA, January 5-8, 2015*, pages 4120–4129, Jan 2015.

[74] P. Seshan. *Process-Centric Architecture for Enterprise Software Systems.* Infosys Press. CRC Press, 2011.

[75] R. Shapiro, S. A. White, C. Bock, N. Palmer, M. z. Muehlen, M. Brambilla, and D. G. e. al. *BPMN 2.0 Handbook Second Edition: Methods, Concepts, Case Studies and Standards in Business Process Modeling Notation.* Future Strategies, Incorporated, Dec. 2011.

[76] K. Sousa, H. Mendonça, J. Vanderdonckt, E. Rogier, and J. Vandermeulen. User Interface Derivation from Business Processes: A Model-driven Approach for Organizational Engineering. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 553–560, New York, NY, USA, 2008. ACM.

[77] A. Speck, S. Witt, S. Feja, A. Lotyzc, and E. Pulvermüller. Framework for Business Process Verification. In W. Abramowicz, editor, *Business Information Systems*, volume 87 of *Lecture Notes in Business Information Processing*, pages 50–61. Springer Berlin Heidelberg, 2011.

[78] W. Tan and M. Zhou. *Business and Scientific Workflows: A Web Service-Oriented Approach*. IEEE Press Series on Systems Science and Engineering. Wiley, 2013.

[79] H. Trætteberg and J. Krogstie. Enhancing the Usability of BPM-Solutions by Combining Process and User-Interface Modelling. In J. Stirna and A. Persson, editors, *The Practice of Enterprise Modeling*, volume 15 of *Lecture Notes in Business Information Processing*, pages 86–97. Springer Berlin Heidelberg, 2008.

[80] U. University. PHILharmonic Flows - Process, Humans and Information Linkage for harmonic Business Flows. http://www.uni-ulm.de/en/in/dbis/research/projects/philharmonic-flows.html, June 2015. [Online; accessed 01-June-2015].

[81] J. Vanderdonckt. A MDA-Compliant Environment for Developing User Interfaces of Information Systems. In *CAiSE*, pages 16–31, 2005.

[82] W3C. OWL 2 Web Ontology Language Second Edition, December 2012.

[83] J. Weaver, W. Gao, S. Chin, D. Iverson, and J. Vos. *Pro JavaFX 8: A Definitive Guide to Building Desktop, Mobile, and Embedded Java Clients*. The expert's voice in Java. Apress, 2014.

[84] WFMC. XML Process Definition Language Version 2.2, August 2012.

[85] I. Wood, B. Vandervalk, L. McCarthy, and M. Wilkinson. OWL-DL Domain-Models as Abstract Workflows. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, volume 7610 of *Lecture Notes in Computer Science*, pages 56–66. Springer Berlin Heidelberg, 2012.

[86] M. Wynn, H. Verbeek, W. van der Aalst, A. ter Hofstede, and D. Edmond. Business process verification - finally a reality! *Business Process Management Journal*, 15(1):74–92, 2009.

[87] Y. Yan, J. Zhang, and M. Yan. Ontology Modeling for Contract: Using OWL to Express Semantic Relations. In *Enterprise Distributed Object Computing Conference, 2006. EDOC '06. 10th IEEE International*, pages 409–412, Oct 2006.

[88] S. Yongchareon and C. Liu. A Process View Framework for Artifact-Centric Business Processes. In R. Meersman, T. Dillon, and P. Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2010*, volume 6426 of *Lecture Notes in Computer Science*, pages 26–43. Springer Berlin Heidelberg, 2010.

[89] S. Yongchareon, C. Liu, X. Zhao, and J. Xu. An Artifact-Centric Approach to Generating Web-Based Business Process Driven User Interfaces. In L. Chen, P. Triantafillou, and T. Suel, editors, *Web Information Systems Engineering — WISE 2010*, volume 6488 of *Lecture Notes in Computer Science*, pages 419–427. Springer Berlin Heidelberg, 2010.

Acronyms

API Application Programming Interface.

BP Business Process.

BPEL Business Process Execution Language.

BPM Business Process Model.

BPMN Business Process Model Notation.

BPMS Business Process Management System.

CSS Cascading Style Sheets.

DAO Data Access Object.

ESB Enterprise Service Bus.

GUI Graphical User Interface.

J2EE J2EE (Java 2 Platform, Enterprise Edition).

JDK Java Development Kit.

MVC Model-View-Controller.

MVP Model-View-Presenter.

MVP (SC) MVP Supervising Controller.

MVVM Model-View-ViewModel.

ORM Object-relational mapping.

OWL Web Ontology Language.

OWL-S Semantic Markup for Web Services.

POJO Plain Old Java Object.

RDF Resource Description Framework.

RIA Rich Internet Application.

SOA Service Oriented Architecture.

SPARQL SPARQL Protocol and RDF Query Language.

UI User Interface.

WSDL Web Service Definition Language.

XML Extensible Markup Language.

XPDL XML Process Definition Language.