ulm university universität
uulm

Ulm University | 89069 Ulm | Germany

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

Workflows on Android: A Framework Supporting Business Process Execution and Rule-Based Analysis

Master Thesis at Ulm University

Submitter:
Wolfgang Wipp
wolfgang.wipp@uni-ulm.de

Reviewer:
Prof. Dr. Manfred Reichert
Dr. Rüdiger Pryss

Supervisor:
Johannes Schobel

2016

Version March 2, 2016

Satz: PDF-LaTeX 2$_\varepsilon$

Abstract

In companies, *Business Process Management* is often supported by *Process-aware information systems* (*PAIS*). However, such systems are mostly restricted to stationary desktop computers. To overcome this restriction, smart mobile devices may be used for mobile business process execution. However, with traditional PAIS having a client-server architecture, the computation is done on server side, whereas the client only visualizes business process tasks and interacts with the user. Therefore, smart mobile devices must rely on their mobile connection to provide PAIS features to its users.

A possible solution is the transfer of server side features to the smart mobile device itself, enabling the latter to instantiate business process models and analyze business process instances by itself, without the need of a direct connection to a workflow-server.

This thesis presents *Workflows on Android* (*WOtAN*), a modular and flexible framework for business process management running on Android smart mobile devices. However, the thesis focuses on flexible and robust business process execution as well as the analysis of business process instances. For this, predefined evaluation rules are applied on data that was collected during business process instance execution.

Different concepts and interesting implementation aspects are presented in this thesis. Further, an application scenario is shown, where WOtAN is used to properly support a mobile data collection application.

Acknowledgement

First of all, I thank *Johannes Schobel* for his important mentoring through all this thesis. Further, I'd like to thank my fellow students, especially *Michael*, *Bernd*, *Steini*, and *Markus* for their support. Furthermore, I thank *Kenny Loggins*, who's song *Danger Zone* escorted me through the long nights of implementation and writing.

A special thanks goes to *Andrea* and *Florian*, who saved me a huge amount of time by styling the graphical user interface for the use case project.

Finally, I thank my family for motivating me during the whole thesis, again ,and again, and again...

Contents

1 Introduction **1**

1.1 Purpose of this Thesis 2

1.2 Structure of this Thesis 3

2 Fundamentals **5**

2.1 Object-Relational Mapper 5

2.1.1 General Overview 5

2.1.2 Model Generation Approaches 6

2.2 Android 8

2.2.1 Application Structure 9

2.2.2 Application Communication 9

2.3 ADEPT2 10

2.3.1 Process Model 10

2.3.2 Process Execution Logic 12

3 Related Work **15**

3.1 jBPM 15

3.2 MARPLE 16

3.3 Questionnaire Process Execution Engine 17

3.4 Discussion 17

4 Requirements **19**

4.1 Functional Requirements 19

4.2 Non-Functional Requirements 20

4.3 Discussion 21

5 Concepts and Architecture **23**

5.1 General Overview 23

5.2 WOtANExecution 25

5.2.1 Concepts 26

5.2.2 Architecture 29
5.3 WOtANAnalysis 32
5.3.1 Concepts 32
5.3.2 Architecture 33

6 Implementation Aspects **35**
6.1 Interface Design 35
6.2 Executable Business Process Management 36
6.2.1 Structure 36
6.2.2 Loading at Runtime 37
6.2.3 Development Challenges 38
6.3 Object-Relational Mapping 39
6.3.1 Schema Generation 40
6.3.2 Database Communication 40
6.4 Rule Evaluation Process 42
6.4.1 General Usage 42
6.4.2 Rule Evaluation 43

7 Application Scenario: Mobile Data Collection Application **47**
7.1 Introduction 47
7.2 System Architecture 48
7.2.1 Questioneer 49
7.2.2 QuestionRule 49
7.2.3 Questionizer 49
7.2.4 Questionnaire 49
7.3 Using WOtAN in the Context of Mobile Data Collection Applications . . . 50
7.3.1 Overall Architecture 50
7.3.2 Executing Questionnaires using WOtANExecution 50
7.3.3 Evaluating Questionnaires using WOtANAnalysis 52
7.3.4 Page Executable Business Process 52

8 Conclusion **55**

8.1 Vision 56

A Sources **65**

A.1 Dynamic EBP Loading 65

A.2 Example Layout Inflater Factory 66

A.3 greenDAO Schema Generation Code 66

A.4 Database Access with GreenDAO 67

1

Introduction

To sustain on global market, companies must act in a flexible and agile way. For example, they must react on market changes rapidly and continuously improve their products and services. Currently, business process management offers promising perspectives to achieve these goals. As a result, optimized business processes improve companies in terms of costs, time to market, and product quality [1]. Despite the fact that BPM does not require an IT system support, most companies use *Process-aware information systems* (*PAISs*) [2] to support process management.

During the last years, PAISs have become very powerful systems, providing flexibility by separating application code from the business process logic [3]. As a result, dynamic changes of business processes became possible, without updating PAIS applications. PAISs are most likely designed in a Rich-Client/Thin-Server architecture, where the server provides all process management features, whereas the client is only responsible for visualization and interaction [4]. Unfortunately, as for all IT systems, PAISs are locally restricted to given IT-endpoints (desktop computer, laptops). Therefore, some domains may have processes, that cannot be executed in a PAIS's reach [5]. In clinical ward rounds, for example, the doctor may first perform the *real-world business process task*, then go back to an computer and perform the *IT supported business process task* executed on a PAIS. This workaround is error-prone, as collected data may be transcribed manually. Further, the duplicate enactment of business process tasks, as well as media breaks, are inefficient. Within larger processes there may exist only few business process tasks, that are not in reach of the supporting PAIS.

Since a business process' state of execution is stored in the PAIS, the duplicate enactment of tasks stalls the process for the time between real world execution and IT

supported execution. The system does not know, that the real world task is already been performed and what outcome it has. This delay of execution may become a severe problem in time critical processes [6]. As a result, *mobile IT support for business processes* (i.e., enable workers to perform IT supported business process tasks at any location and at any time), is a highly desired feature [7, 8, 9].

With the fast dissemination of *smart mobile devices* (smart phones, tablet computers) as well as their technological development over the last years, the power of common office computers shrunk to the size of a pocket device. Therefore, smart mobile devices may serve as possible hardware for mobile IT support for business processes. They are handy enough to be carried anywhere, at any time, providing enough computation power to work on sophisticated process tasks.

1.1 Purpose of this Thesis

Client and server of a PAIS must be connected all the time to function properly. This is no problem with locally set up workstations but can become a problem with smart mobile devices and their respective mobile connection. For example, a sales representative may need to start the companies `contract process` at a customer's location, with no available mobile connection. However, if no connection to the PAIS's server is available, the smart mobile device may not be able to perform process control features. A missing mobile connection may occur because of several reason, like standing in a dead spot, network service provider issues, or PAIS server issues.

To overcome such difficulties, this thesis introduces *Workflows on Android* (*WOtAN* for short), focusing on process execution (*WOtANExecution*) and process analysis using rules (*WOtANAnalysis*). WOtAN thereby is a framework for dedicated business process management on Android smart mobile devices, using current state-of-the-art PAIS technology. Furthermore, WOtAN enables smart mobile devices to manage and execute business processes without any external communication, hence eliminating the issue of a lost mobile connection.

1.2 Structure of this Thesis

The structure of this thesis is as follows: Chapter 2 describes relevant fundamentals in the context of this thesis. Subsequently, Chapter 3 introduces related work and elaborates common features as well as differences in state-of-the-art products. Next, Chapter 4 lists and discusses both functional and non-functional requirements for WOtANExecution and WOtANAnalysis. Chapter 5 introduces the overall concepts and presents the architecture of the developed lightweight mobile process engine. Afterwards, Chapter 6 provides deeper insight into the implementation, specifically focusing on the *Executable Business Process*, a software template for Android executed in business process tasks during runtime. Following, Chapter 7 illustrates the use of WOtAN within a mobile data collection application scenario. Finally, Chapter 8 summarizes the thesis and proposes future work.

2

Fundamentals

This chapter introduces the fundamentals of this thesis. First, Section 2.1 describes object-relational mapping. Subsequently, Section 2.2 shows Android basics needed in this thesis. At last, Section 2.3 illustrates the, for the thesis relevant, concepts of ADEPT2.

2.1 Object-Relational Mapper

Object-oriented programming languages (*OOP-Languages*) and *relational databases* are based on different paradigms. Therefore, the communication between OOP-Languages and relational databases is difficult [10]. As a result, *object-relational mapper* (*ORM*) are developed to close the gap between object paradigm and relational paradigm. This section introduces *object relational mapping*. Section 2.1.1 provides a general overview regarding object-relational mapping, whereas Section 2.1.2 presents different approaches for model generation.

2.1.1 General Overview

ORM simplifies and abstracts the communication between application and database. Instead of manually creating objects and propagating them with database values, ORMs automatically map application classes to database tables, encapsulating the value to object mapping process, as illustrated in Figure 2.1. Hence, an application programmer does not need database language experience, but can rely on application

language based database access i.e., using automatically generated functions to query the database, insert or remove entries.

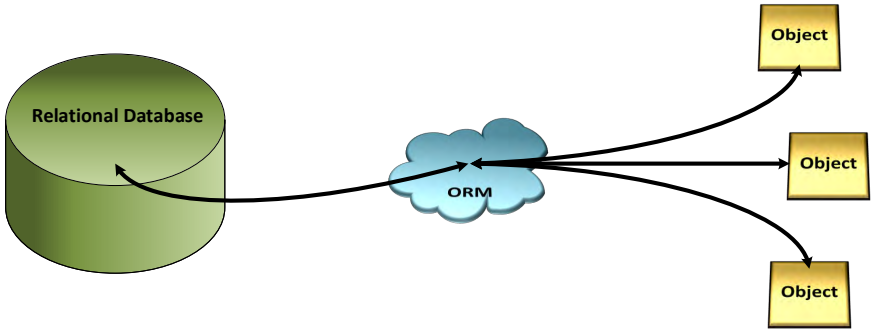

Figure 2.1: ORM Overview

Additionally, ORM reduces duplicate work by either generating the database schema or the application data classes. As a result, the application programmer needs to develop only one data model instead of two separate ones for database and application. Accordingly, maintainability of such produced data models are drastically improved, as changes affect both database and application.

However, despite the lose decoupling from the database, a programmer should have basic knowledge of databases in general and good knowledge of specific database system used.

2.1.2 Model Generation Approaches

This section describes the automatic generation of a data model and the respective counterpart on either application or database side. There are several approaches for data model generation:

1. *Model First*: Create application side classes first and generate database tables based on them.
2. *Database First*: Create database tables first and generate application side classes based on them.

3. *Hybrid Generator*: Both database tables and application classes are generated based on a meta model.

The following sections describe each approach and gives deeper insight.

1. Model First Approach

This approach generates database tables from previously written model classes. Mapping respective classes to database tables can be achieved with different techniques. First, annotations can be used to set database attributes i.e., table name for a class, primary key, include member as field. Listing 2.1 shows an example using ORMLite [11].

```
1  @DatabaseTable(tableName ="accounts")
2  public class Account {
3
4      @DatabaseField(id = true)
5      private String name;
6      @DatabaseField
7      private String password;
8
9      public Account() {
10         // ORMLite needs a no-arg constructor
11     }
12     public Account(String name, String password) {
13         this.name = name;
14         this.password = password;
15     }
16     // Getter and Setter for all Fields of this class
17     ...
18 }
```

Listing 2.1: ORMLite Class Example

The `@DataBaseTable` annotation declares this class to be mapped to a database table with the name `accounts`, with the member variable `name` as its primary key.

Using another technique, classes can be directly mapped to database tables by extending ORM specific abstract classes, as shown in Listing 2.2, using Sugar ORM [12].

```
1  public class Book extends SugarRecord {
2    String name;
3    String ISBN;
4    String title;
5    String shortSummary;
6  }
```

Listing 2.2: Sugar ORM Class Example

The example shows a `Book` class that has a `name`, `ISBN`, `title`, and `shortSummary` attribute. The super class `SugarRecord` takes care of database mapping, including all member variables as respective columns within a table named as the class.

This approach is handy when an application data model already exists. Further, an application developer does not need database language experience to create a database schema.

2. Database First Approach

This approach takes a given database table schema and generates corresponding application classes. As mentioned in [13], *doctrine*, an ORM system for PHP, provides this feature. For this, doctrine generates an XML mapping file from which it generates the respective PHP class sources, as described in [14].

3. Hybrid Generator Approach

The hybrid generator approach neither take a data model nor a database schema to generate the other part, but uses a meta model to generate both. This meta model describes database and table features ,like primary keys, relations between them, and indexes. Furthermore, application features, like class hierarchies, or interface implementations may be defined. GreenDAO [15], which will be explained in more detail in Section 6.3, follows this approach.

2.2 Android

This section describes basic knowledge about the Android operating system, called *Android* for short. Therefore, the Android application structure, as well as inter-application communication is explained in more detail.

2.2.1 Application Structure

Any Android application consists of the following parts:

- `Sources`: Java class files, later translated into DalvikVM byte code format.
- `Resources`: Resource files, like XML layout files, String values, or images.
- `Assets`: Files, that are not part of resources.
- `Manifest`: Contract file between the application and the operating system.

According to [16], resource files are stored in a predefined folder structure that allow multiple values, dependent on device screen size and location. The `Manifest` is an XML file responsible for presenting application information to Android. Such information is e.g., the *package name* of the application, which serves as global identifier, and implemented *activities* with their processable *intents*. Additionally, permissions for device resources like the camera, storage and network service, are stored within the `Manifest` [17].

After compiling, all parts are stored in an *Android application package* (`apk`). This is the only usable format to install applications on Android smart mobile devices.

2.2.2 Application Communication

Communication between *activities* [18] and *fragments* [19] are performed using *intents* [20]. Thereby, an intent is a messaging object used to request an action from another application component [20], regardless of this component being part of the calling application or another one. As described in Section 2.2.1, the `Manifest` act as the contract between respective application and Android. Therefore, every activity used inside an application must be registered in the applications `Manifest` file. Additionally, to be able to be called from an external application, an *intent filter* [20] must be added.

Using this contract, Android is able to search for applications installed on the device, that can serve a given intent. Figure 2.2 illustrates such an activity call. *AppA*'s activity *Activity1* starts an intent, that can be served by *AppB*'s activity *Activity5*. At ①, AppA

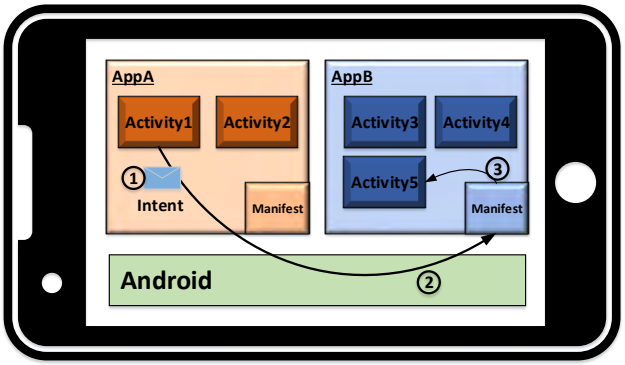

Figure 2.2: Calling Another Activity Using Intents

sends its intent to Android, which searches for matching, registered activities ②. For this, it uses the applications' `Manifest` file. If a matching activity is found (in this case Activity5) ③, Android automatically starts AppB with its activity Activity5. Optionally, Activity5 may return a result to its caller (Activity1) after execution. For sending this result, or more general any data between activities, intents have a common key-value storage inside, called *bundle* [21].

2.3 ADEPT2

The *Application Development based on Encapsulated Pre-modeled Process Templates* (*ADEPT2*) project [4] started in 1995 aiming at delivering a next generation business process management technology. This section introduces relevant topics for this master thesis. Therefore, Section 2.3.1 describes the process model of ADEPT2, whereas Section 2.3.2 illustrates ADEPT2's concept of process execution.

2.3.1 Process Model

ADEPT2 process models are based on acyclic graphs [22, 23]. It consists of *nodes*, *control edges*, *data elements*, and *data edges* [24]. One node is one step in the process

model. Control edges are unidirectional edges connecting two nodes and thereby determining the order of execution. Hence, if a control edge connects node **A** with **B**, it means that **A** must be executed before **B**.

Nodes may have different *node types* [23]. Node type `NT_STARTFLOW` marks the start of a process model, consequently `NT_ENDFLOW` marks the end of a process model. Note that a process model can only have one start and one end node. Further, `NT_NORMAL` is the node type used for declaring a node with one incoming and one outgoing control edge, thus describing a sequence of execution. Furthermore, `NT_AND_SPLIT` splits the control flow into multiple parallel control flow branches. In contrast, `NT_XOR_SPLIT` also splits the control flow, but allows only one of these branches to be selected and executed. At last, `NT_LOOP_SPLIT` declares a reverse jump to an earlier point of execution, enabling parts of a process model to be performed multiple times. Additionally, each `SPLIT` type has a corresponding `JOIN` type, spanning a *control block*. ADEPT2 only allows control blocks with one incoming and one outgoing control edge. Additionally, control blocks may not overlap, but can be nested [25].

Data elements are *process variables* storing data during execution [25]. Data elements' values are written and read by nodes. The access type is modeled with unidirectional data edges. On one hand, a data edge from a node to a data element means `write access`. On the other hand, a data edge from a data element to a node declares `read access`. Figure 2.3 shows an example process with all elements.

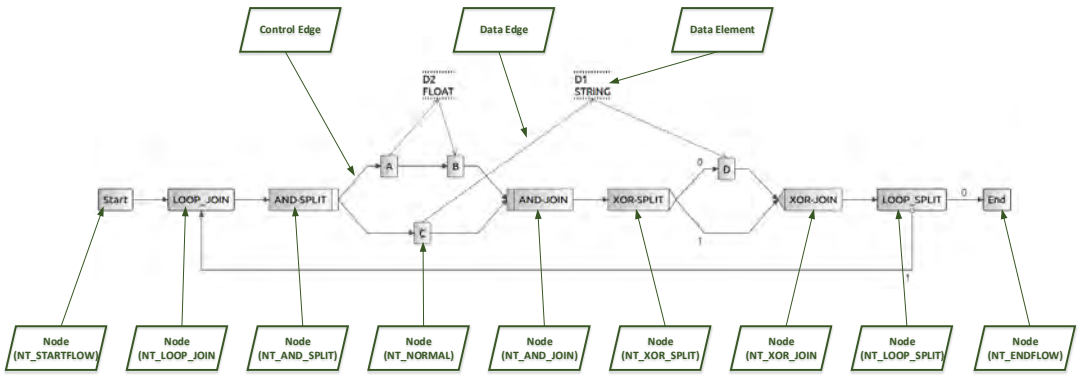

Figure 2.3: Example Process with Additional Annotations

To ensure structural correctness, ADEPT2 uses a principle called *correctness by construction* [26] i.e., offering the process modeler only modeling options that keep the process model syntactically correct.

2.3.2 Process Execution Logic

During the execution of the process model, nodes are transitioning through different *node states*, as Figure 2.4 illustrates.

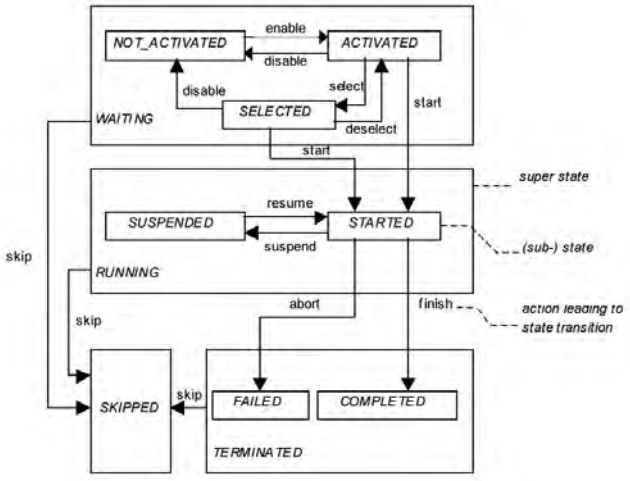

Figure 2.4: Node States Chart [25]

Initially, a node is in node state `NOT_ACTIVATED`. In this state, a node is not ready to be performed. If all preconditions are met, a node switches into node state `ACTIVATED`. Now, the node can be triggered to be performed, sending it in node state `STARTED`. While executing this node, it can be `SUSPENDED` to interrupt its current execution. After completion, a node changes into node state `COMPLETED` or, if an error occurred, `FAILED`.

Additionally to nodes, control edges are also marked during process execution. Control edges have 3 *edge states* [23]: `NOT_SIGNALED`, `TRUE_SIGNALED`, `FALSE_SIGNALED`. A control edge is in edge state `NOT_SIGNALED` only when their source node is not in node state `COMPLETED`, `SKIPPED`, or `FAILED`. Further, a source node with node state

SKIPPED or FAILED gets the control edge into FALSE_SIGNALED. For marking control edges with TRUE_SIGNALED marking rules dependent on the source node must be declared. Also, activation rules for nodes must be defined.

Both marking and activation rules are defined in [23]. Therefore, after completing a node of type NT_XOR_SPLIT, or NT_LOOP_SPLIT, only one of the outgoing control edges is marked TRUE_SIGNALED, all other are set to FALSE_SIGNALED. In contrast, completing nodes of any other type results in marking all outgoing control edges as TRUE_SIGNALED. For nodes, there are two rules for activation: either, all incoming control edges (*at least all*) or only one (*at least one*) must be marked TRUE_SIGNALED. Table 2.1 shows the node types activation rule.

Node Type	Activation Rule
NT_STARTFLOW	-
NT_ENDFLOW	at least all
NT_NORMAL	at least all
NT_XOR_SPLIT	at least all
NT_XOR_JOIN	at least one
NT_AND_SPLIT	at least all
NT_AND_JOIN	at least all
NT_LOOP_SPLIT	at least all
NT_LOOP_JOIN	at least one

Table 2.1: Activation Rules

Note that the start node does not have any incoming control edge and therefore must be activated *manually* at process execution start [23].

3

Related Work

Currently, several workflow engines are available that may be used on Android or are specifically designed for smart mobile device support. Sections 3.1 and 3.2 cover more general workflow engines, whereas Section 3.3 introduces the workflow engine previously used for the application scenario, described in Section 7. Finally, Section 3.4 discusses the provided features of each engine.

3.1 jBPM

jBPM [27] is a Business Process Management Suite, written entirely in Java. It offers a generic process execution infrastructure, allowing the integration of different process model notations (BPEL, EPC,..) but having a basic core engine, ready to execute BPMN 2.0 [28] process models. Further, jBPM offers features for modeling, monitoring, and managing process models and process instances. As this thesis deals with the issue of process execution and analysis, this section covers the execution concepts of jBPM.

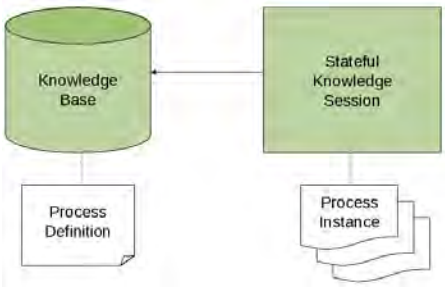

Figure 3.1: jBPM Overview [29]

As illustrated in Figure 3.1, a user interacting with the engine first must create a *stateful knowledge session* for process execution. It is used for engine communication and needs a reference on a *knowledge base*. The latter holds all relevant *process definitions* (i.e., process models). Whenever a process is started, a new *process instance* is created based on the respective process definition.

For domain specific tasks, jBPM uses so called *custom work items* [30]. A custom work item, however, is a BPMN `<task>` containing a `WorkItem` Java class to be used by a `WorkItemHandler`. A `WorkItemHandler` executes (or aborts) a given work item. The `WorkItemHandler` gets registered at the `WorkItemManager` of the stateful knowledge session used for process execution.

3.2 MARPLE

MAnaging Robust mobile Processes in a compLEx world (*MARPLE*) [31] is a light-weight process engine for smart mobile devices developed at Ulm University, enabling execution of central stored process models across different smart mobile devices by a client-server architecture. It consists of two main components [31], the *MARPLE Mediation Center* and the *MARPLE Mobile Engine*(of Figure 3.2).

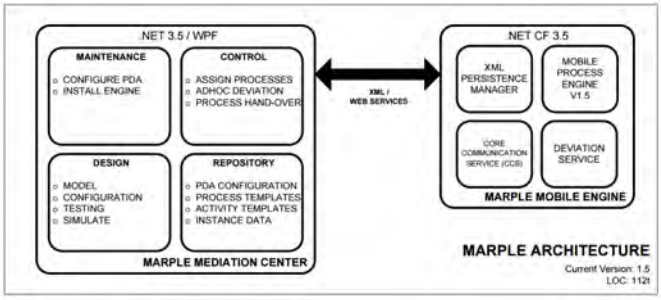

Figure 3.2: General MAPRLE Architecture [31]

The MARPLE Mediation Center is responsible for the administration of process models and smart mobile devices. This includes, for example, deploying the MARPLE mobile

engine on the smart mobile devices as well as modeling, transferring, and executing process models.

The MARPLE mobile engine takes care of process execution with pre-set *Activity Templates*, pre-manufactured application components, executed at runtime as process tasks [7]. Further, it tackles the issue of process derivation, as well as ad-hoc changes during the execution og a specific process instance. Therefore, it uses ADEPT 2.3 concepts. Finally, MARPLE Mobile Engine is responsible for communication to the MARPLE Mediation Center.

3.3 Questionnaire Process Execution Engine

In the context of a diploma thesis [32], a questionnaire process engine was implemented. Used concepts are derived from ADEPT, resulting in a fully operational process execution engine. However, implementations for process tasks are stored inside the engine at build time, if a new process task implementation is needed or an already existing one needs to be modified, this results in updating the engine itself as well as the application using the engine. Further, communication between engine and application is based on XML documents. Hence, serialization of the latter results in high computation effort for communication.

3.4 Discussion

This chapter presented three different workflow engines for executing business processes. First, *jBPM* is introduced. It is a very powerful light-weight engine which can be used on devices capable of running Java. However, jBPM is not specifically designed for smart mobile devices. In contrast, *MARPLE*, subsequently introduced, is specifically designed for the execution of business processes on smart mobile devices. This engine is also very powerful, but relies on a client-server architecture. Finally, the questionnaire process execution engine is presented. However, it lacks plug & play features for process task implementations. A change in process task implementations results in redeploying all

respective applications. Table 3.1 shows key-features for the discussed mobile process engines. The legend is as follow: '(✓)' stands for partially available, '✓' for fully available and 'x' for not available

Feature	jBPM	MARPLE	Questionnaire Engine
Smart Mobile Device Support	(✓)	✓	✓
Flexible Process Execution	✓	✓	✓
Executable Components	✓	✓	✓
Ad-hoc Changes	x	✓	x
Monitoring	✓	✓	x
Analysis	✓	x	x

Table 3.1: Key-Features for Mobile Process Engines

4

Requirements

This chapter covers basic requirements for WOtAN, derived from the workflow engines introduced in Chapter 3. First, Section 4.1 introduces functional requirements, followed by Section 4.2 describing non functional requirements. Finally, Section 4.3 emphasises and discusses key requirements in the context of this thesis.

4.1 Functional Requirements

This section covers functional requirements for WOtAN, i.e., requirements which specify the systems behaviour:

F_1 - Import Process Model: WOtAN shall be able to import process models.

F_2 - Process Notation: WOtAN shall be able to import different process model notations.

F_3a - Execute Process: WOtAN shall be able to traverse through an imported process and execute each traversed process task.

F_3b - Execute Process: WOtAN shall be able to work without any external connection.

F_4a - Use Executable Business Process: WOtAN shall be able to use executable business processes for process task execution.

F_4b - Executable Business Process: An executable business process is a reusable software template.

F_4c - Load EBPs: WOtAN shall be able to load externally stored executable business processes at runtime.

F_5 - Log Process Execution: WOtAN shall be able to log the execution of an executed process model.

F_6a - Export Process Execution Data: WOtAN shall be able to export process execution data.

F_6b - Process Execution Data: Process execution data consists of all relevant data created at execution time.

F_7 - Delete Process Model: WOtAN shall be able to delete imported process models.

F_8 - Import Rule: WOtAN shall be able to import evaluation rules.

F_9 - Evaluate Rule: WOtAN shall be able to evaluate imported evaluation rules.

F_10a - Use Rule Functions: WOtAN shall be able to use rule functions.

F_10b - Load Rule Functions: WOtAN shall be able to load rule functions at evaluation time on demand.

F_11a - Report Evaluation Report: WOtAN shall be able to return an evaluation report.

F_11b - Evaluation Report: An evaluation report consists of the result of evaluation, the evaluated evaluation rule, and other meta data.

4.2 Non-Functional Requirements

This section covers non-functional requirements which define quality goals of the process engine.

NF_1 - Programming Language: WOtAN shall be written in Java, to ensure full compatibility with Android.

NF_2 - Modularity: WOtAN's architecture shall be modular for easy implementation and maintainability.

NF_3 - Standalone Modules: WOtAN modules shall be usable without any other module.

NF_4 - Common Communication Language: WOtAN modules shall be loosely coupled by a common communication interface.

NF_5 - High-Level API: The offered API shall be high-level enough to cover internal structures, e.g., process model structure, engine structure.

NF_6 - Process Data Flow: Process data values shall be managed internally, not visible to the application programmer.

NF_7 - Object-Oriented Communication: Communication is achieved using objects, not serialized data structures.

NF_8 - Decoupled Process Task Implementation: Implementation of process tasks, e.g., user forms or scripts, must be decoupled from the workflow engine itself.

NF_9 - Extensibility at Runtime: At runtime, import and export plug-ins, rule functions, as well as executable business processes shall be added without main application recompilation.

4.3 Discussion

As one can see in Section 4.1 there are requirements for storing and deleting of both process models and rules, but none for editing them. This results from the general approach for WOtAN, described in Chapter 5, by separating modeling and executing of process models into different modules. **F_2** defines the requirement for different process model notations being executable using WOtAN. The main idea, hereby, is to transform other notations into the one WOtAN uses internally.

One problem of the questionnaire engine, introduced in Section 3.3 is the unnecessary serialization and de-serialization of XML documents for internal communication. Therefore, **NF_7** restricts communication to object-based manner. This is no contradiction to the functional requirements for import and export, since these plug-ins are not meant for direct communication, but for communication, where objects are not possible to use, e.g., device to device communication or web services.

The high-level API claimed by **NF_5** and internally managed process data flow of **NF_6** are meant to reduce complexity for applications programmers. This way, they can focus on the application itself and not on the workflow engine.

5

Concepts and Architecture

This chapter describes the core architecture of *Workflows on Android (WOtAN)* developed in this thesis. First, a brief overview regarding the main architecture and core concepts of WOtAN are introduced. Second, more details for the process model execution engine are discussed. Finally, more information with respect to the rule-based analysis engine are given. Note that **(F_x)** or **(NF_x)** means, that respective requirement is served at this point of the thesis.

5.1 General Overview

This Section gives an overview over WOtANs' general architecture. It starts at the top-level concept towards deeper ones.

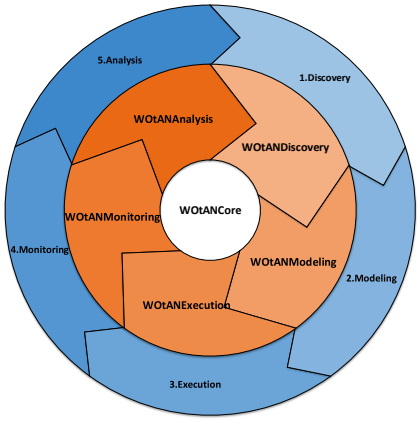

Figure 5.1: WOtAN Modules Integrated in BPM Lifecycle

A process engine is a very complex technology [33]. To handle this complexity, WOtAN consists of five *modules* connected by one common core, shown in Figure 5.1, called *WOtANCore*. Each module represents one step in the *process model lifecycle* [34]. The five modules are **(NF_2)**:

1. *WOtANDiscovery*: Responsible for process discovery i.e., finding existing process modeles in companies.
2. *WOtANModeling*: Responsible for modeling of new process modeles or introducing ad-hoc manipulations of existing process modeles.
3. *WOtANExecution*: Responsible for process execution.
4. *WOtANMonitoring*: Responsible for live monitoring of process instances.
5. *WOtANAnalysis*: Responsible for analyzing executed process modeles.

This thesis focuses on WOtANExecution and WOtANAnalysis. All modules can be used without any other module, but rely on the WOtANCore **(NF_3), NF_4**. However, some modules may be useless alone. For example, WOtANMonitoring monitors running process modeles. Therefore, WOtANExecution is needed to execute process modeles that can be monitored.

The WOtANCore can be seen as the *common language* of all modules. The latter consists of high level data- and control-layer interfaces, enabling object-based intercommunication between all modules **(NF_5, NF_7)**.

The general architecture of each WOtAN module is 3-Tier, consisting of a data- and a control-layer, shown in Figure 5.2. The view-layer is not needed in the engine itself, but the WOtAN using application. Each layer has a single entry point, called *manager*, for its respective upper layer, defined by a WOtANCore interface. WOtANCore provides for each module a corresponding *data* and *control manager*, e.g., for WOtANExecution this is the *execution manager*. Every action called on a layer should be done over its manager interface.

A data manager is responsible for storing, deleting and fetching the respective data. Because the data manager itself as well as the objects exchanged are based on interfaces,

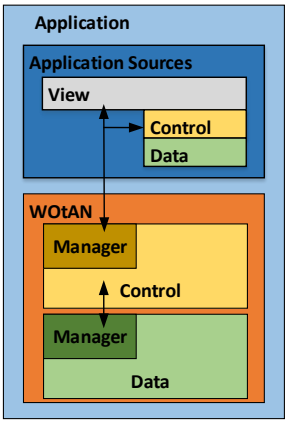

Figure 5.2: WOtAN 3-Tier Architecture

the underlying data structure can be exchanged without any upper layer changes. The same principle of using interfaces is used in the control-layer as well.

This architecture enables the almost unlimited mixing of all WOtAN modules in one application. For example, WOtANExecution and WOtANAnalysis can be combined to enable a smart mobile application to execute and later analyze business process instances. Another useful combination could be to integrate WOtANModeling and WOtANExecution, whereas WOtANExecution works as *test client* for newly modeled process modeles.

5.2 WOtANExecution

As mentioned in Section 5.1, WOtANExecution enables applications to execute process modeles in a flexible and robust way. The following section describes the architecture of WOtANExecution. Subsection 5.2.1 introduces the concept of WOtANExecution. Afterwards, Subsection 5.2.2 shows the WOtANExecution specific architecture.

5.2.1 Concepts

Process Model

A *process model* is a graph-based description of a business process. It consists of different elements shown in Figure 5.3:

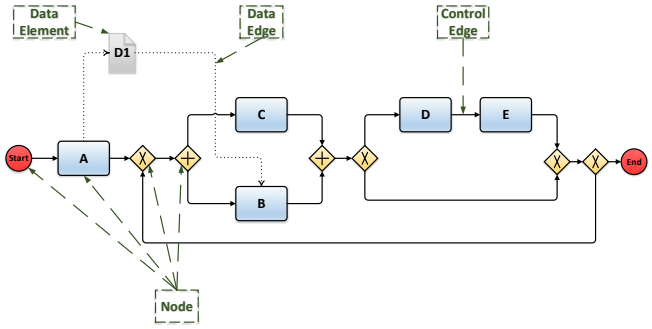

Figure 5.3: Process Model Example

- *Node*: A node is a step within the process model. It can have different *node types*, describing the semantics (e.g., gateway, normal, start, end).
- *Control Edge*: A control edge is an unidirectional edge, describing the order of execution between two nodes, e.g., D→E meaning *D must be executed **before** E can start.*
- *Data Element*: A data element is a variable storing data for a respective instance of the process model. It may be read or written by nodes.
- *Data Edge*: A data edge is an unidirectional edge, describing the behaviour between a data element and a node e.g., A→D1 meaning *A **writes** data in D1* or D1→B meaning *B **reads** data from D1*.

A node's type specifies its behaviour at run time as well as the permitted number of incoming and outgoing control edges. Table 5.1 explains the possible node types.

Node Type	Description
STARTFLOW	Start point for the process model.
ENDFLOW	End point for the process model.
NORMAL	A normal process model task, doing process model tasks.
XOR	Splits the control flow into multiple branches. Only one branch will be activated and executed at run time.
AND	Splits the control flow into multiple branches. All branches will be activated and executed.
LOOP	Allows to jump back at an earlier point of the control flow to execute parts of the process model model again.

Table 5.1: Node Types [23]

The node types XOR, AND and LOOP are divided into a JOIN and a SPLIT subtype. The split and join node automatically generates a block, with the splitting node as entry point, and the joining node as exit point [35].

Process Model Execution

process model execution means traversing the process model graph, performing all nodes on the way, from start to end. To assure a proper and valid process model execution (according to the modeled control flow), the nodes are assigned to different *node states*. All node states and their possible transitions are shown in Figure 5.4. First, a node is in state NOT_ACTIVATED.

When either the node's preconditions are met, the node is set to state ACTIVATED. During this node state, the engine can start a node changing its node state to STARTED executing this step. Afterwards, the node gets terminated as either successful executed (COMPLETED), or executed with errors (FAILED). Subsequently, all following nodes are tested for activation.

When starting a new process model instance, the node with node type `STARTFLOW` is set `ACTIVATED`, thus can be started from the engine. After reaching the node with node type `ENDFLOW`, the execution is considered (successfully) finished.

The node states `SELECTED` and `SKIPPED` are not used for execution behaviour, but as internal helper node states. Therefore, they are not explained at this point.

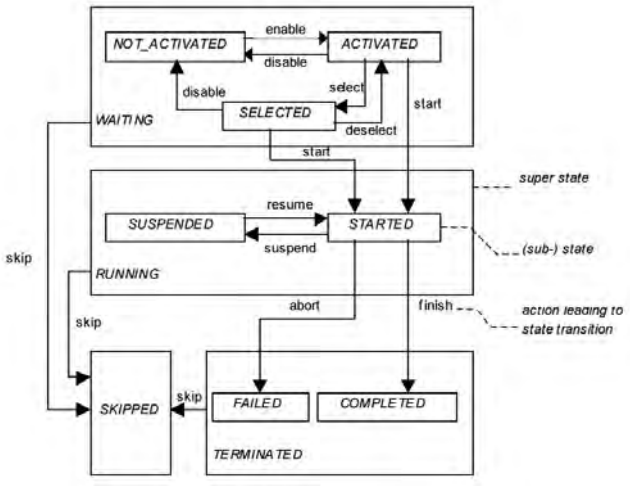

Figure 5.4: Node States Overview [25]

Executable Business Process

An *executable business process (EBP)* is a software template that can be stored in nodes. Its function can vary from a graphical user interface (e.g., a user form) to a complex automatic computation. An EBP uses its own parameters, which must be mapped at run time to the corresponding data elements of the stored-in node. Its execution begins when respective node is started.

5.2.2 Architecture

WOtANExecution has three managers to control process instances: an *execution manager*, *instance manager*, and a *runtime manager*. The manager dependencies are shown in Figure 5.5. The following sections describe each manager and its tasks.

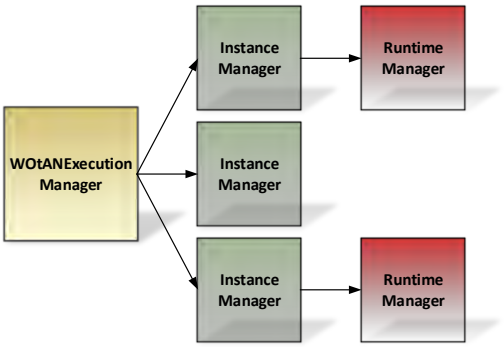

Figure 5.5: WOtANExecution Manager Hierarchy Overview

Execution Manager

The execution manager is the top manager of the control layer. It is responsible for the communication between the application and the engine i.e., all commands to the engine are passed to the execution manager. This should be the only manager ever used by an application developer. The main task of this manager is to distribute given commands to the correct instance manager and return a possible result back to the caller. Further, administration tasks like process import **(F1)**, process deletion **(F7)**, and process execution data export **(F6a)** are distributed to respective components.

Instance Manager

An instance manager is responsible for exactly one currently running business process instance. It is dynamically created by a execution manager when a new process model instance is started and destroyed after the process model has finished. The instance

manager takes care of all related workflow control i.e.,changing node states and instance states correctly **(F_3a, F_3b)**. The instance manager has one runtime manager for controlling the executable business process used in respective process model instance.

Runtime Manager

The runtime manager is responsible for bidirectional communication between the engine and the EBP, shown in Figure 5.6 **(F_4a)**. Therefore, it uses an communication channel. The runtime manager sends EBP control message such as `start`, `suspend`, `restart`, `finish` to the EBP. Additionally, the manager receives messages from the EBP, e.g., `saving data`, `EBP's GUI`.

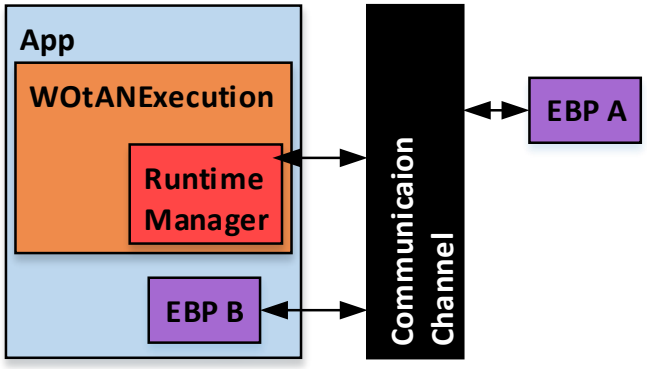

Figure 5.6: Runtime Manager to EBP Communication

As one can see, an EBP can be stored within a smart mobile application or be stored externally **(F_4c, NF_8)**, as illustrated in Figure 5.7. This is possible because WOtANExecution uses weakly referenced EBPs. Internally stored EBPs are only visible to the smart mobile application it inherits. In contrast, externally stored EBPs are visible and can be used by all smart mobile applications on the device that use WOtANExecution. The drawback of internally stored EBPs is that a change in the EBP results in updating the whole smart mobile application the EBP is stored in. A change of externally stored

EBPs does not require the update of any smart mobile application **(NF_9)**. However, on one hand, loading externally stored EBPs uses more computation power than the use of internal EBPs. On the other hand, by storing EBPs internally, a EBP may be stored twice in two separate smart mobile applications, thus increasing storage consumption.

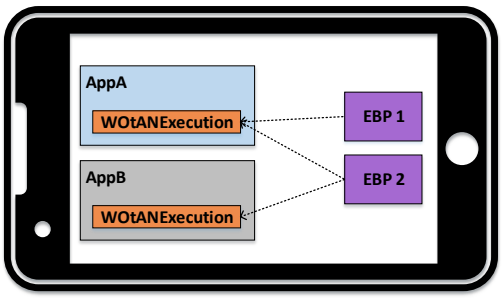

Figure 5.7: Coupling the Engine with Standalone EBPs

Manager Behaviour Example

An example behaviour of managers at runtime is shown in Figure 5.8. This example illustrates the start of a node, that inherits an EBP. The application which uses WOtAN propagates a `start node` command to the execution manager ①. The execution manager passes this command to the responsible instance manager ②. The instance manager, however, generates a new runtime manager and commands it to start the nodes EBP ③. Then, the runtime manager creates a new EBP object, and gets binded to it ④. Then, the runtime manager passes environment data to the EBP for complete initialization ⑤. Finally, the runtime manager starts the EBP with its `execute` command ⑥. Now, that the EBP is running, the application can ask for the EBP's graphical user interface via execution managers `getGUI` command ⑦, traversing the manager stack downwards to the EBP. Note that all commands are always passed down the manager stack to the responsible manager.

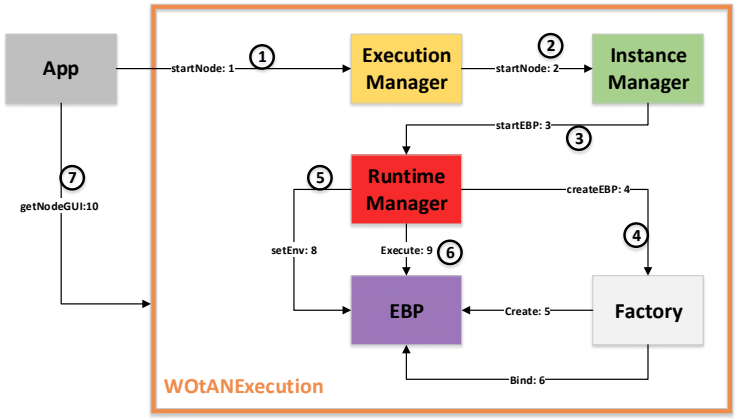

Figure 5.8: Start Node Example

5.3 WOtANAnalysis

The WOtANAnalysis module enables applications to evaluate process modeles executed using WOtANExecution based on predefined rules. The first section describes concepts used, whereas the second section describes WOtANAnalysis' architecture in more detail.

5.3.1 Concepts

Rule

The concept of rules are adopted from [36]. Thereby, a rule consists of `comparisons` connected with `Boolean AND` or `OR`. A comparison consists of two `operands` and a `Boolean operator`. `Operands` can be pre-set values, variables, or functions. A function describes a specific behavior at evaluation time. Figure 5.9 illustrates a simple rule.

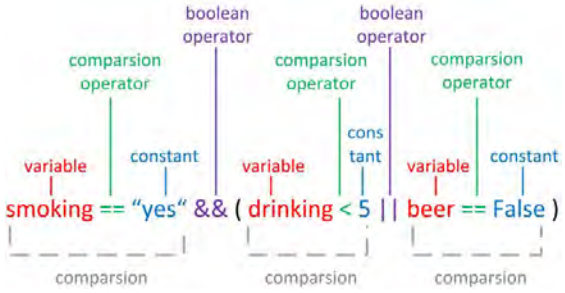

Figure 5.9: Rule Example [36]

5.3.2 Architecture

WOtANAnalysis consists of a *rule manager*, a *rule formatter*, a *rule importer*, and a *rule evaluator* component. The following sections describe their tasks and behaviour, illustrated in Figure 5.10.

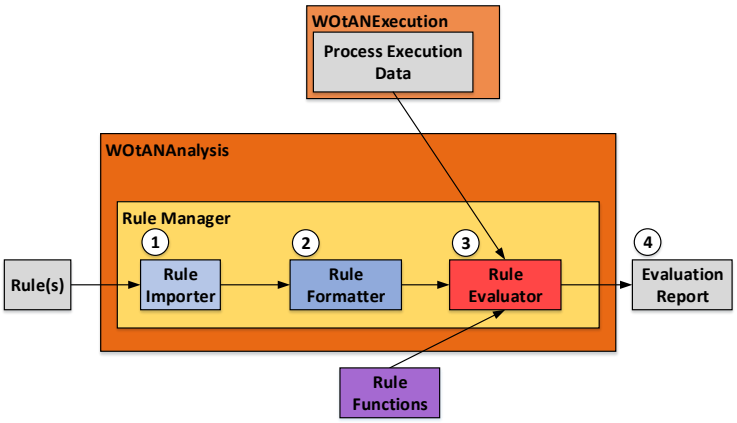

Figure 5.10: Rule Evaluation Overview

Rule Manager

The rule manager is WOtANAnalysis' control manager. Therefore, it is the entry point for every application developer to use WOtANAnalysis. Its task is to orchestrate the rule

evaluation process by distributing all evaluation tasks in correct order to the responsible components.

Rule Importer

The rule importer is the first step in the rule evaluation process. The importer imports a given rule ① **(F_8)**. The source of a rule can be, for example, a file or web-service. Note, that a rule importer must be specifically built for a given external rule format. Therefore, the importer must first create an object structure of the imported rule format, then call the respective rule formatter to transform the rule into the persistence format used ②.

Rule Formatter

The rule formatter is, as the name suggests, a formatting component. It is called after the import of a rule by rule importer, but before persistence of respective rule. Like the rule importer, a rule formatter must be implemented for each external rule format that shall be importable.

Rule Evaluator

The rule evaluator is responsible for evaluation ③ i.e., testing rules on given process execution data **(F_9)**, generated by WOtANExecution, and create an evaluation report as answer ④ **(F_11a)**. The evaluation report includes the result, the tested rule, and other meta information like `id` and `name` of the process model instance that generated the process execution data **(F_11b)**.

As described in Section 5.3.1, a rule can have functions in it **(F10a)**. The function's implementation is stored externally and will be dynamically loaded by the rule evaluator at runtime if required **(F_10b)**.

6

Implementation Aspects

This chapter describes some technical aspects as well as design choices regarding WOtAN's implementation. Section 6.1 explains the design philosophy behind the interfaces. Subsequently, Section 6.2 shows EBP implementation aspects. Following, Section 6.3 introduces the library used for object-relational mapping. Finally, Section 6.4 presents the rule evaluation implementation.

6.1 Interface Design

To ensure a modular and flexible architecture, all functions of any module are offered through interfaces. Hence, a developer works on interfaces of the respective modules. This allows to switch implementations without any changes in the application's code. Therefore, interfaces are an important aspect of WOtAN.

Taking the limited data types passable to an Android `Bundle` in account , all interfaces shown to application developers uses primitive data types and Java's UUID class as parameters only. The following code shows method examples of the execution manager interface:

```
1 public interface ExecutionManager{
2         public void startNode(UUID modelId, Long nodeId);
3         public ProcessModel getProcessModel(UUID modelId);
4         ...}
```

Listing 6.1: Excerpt from Interface Methods

Instead of passing an object, only their identifiers are passed. As a result, a `Bundle` can be filled with fast serializing types, reducing serialization computing for objects that are

most likely cached in the data manager and , therefore, quickly reloaded. Additionally, object loading can be skipped if no further information is required.

As one can see at method `getProcessModel`, return types can be objects. Alike methods are called after changing to another Android `Activity` or inside a new Android `Fragment`. Further, one can see the rather high level API approach at method `startNode()`, which reduces the starting of a specified `Node` to just one method call.

6.2 Executable Business Process Management

This Section gives an overview over EBP specific aspects of implementation. First, Section 6.2.1 describes the general structure for every EBP. Then, Section 6.2.2, introduces dynamic EBP loading at runtime. Finally, Section 6.2.3 describes challenges which need to be considered by EBP developers.

6.2.1 Structure

An EBP is developed as an Android application, weakly coupled to WOtAN by referencing WOtANCore. However, this application does not have any Android `Activity` in its `Manifest`. Despite the fact that the EBP is installed on the Android smart mobile device as a normal application, it cannot be launched by the Android Launcher and is *invisible* in the application overview. The reason for installing EBPs is to use its resource files (e.g., predefined String values, images, layouts). Furthermore, installation routines makes storing EBPs easier, because there is no need to manage an EBP repository.

As in Figure 6.1, EBPs should extend from `AbstractExecutableBusinessProcess`, which is an Android `Fragment` and implements `ExecutableBusinessProcess` interface. This way, the EBP developer can focus on semantic implementation, instead of thinking about infrastructural belongings.

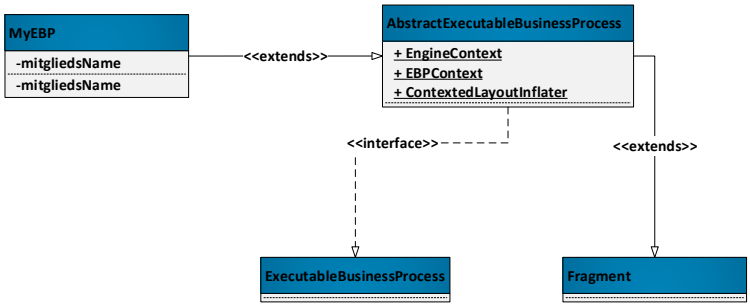

Figure 6.1: EBP Overview

6.2.2 Loading at Runtime

Because all EBPs are stored outside the main application's sources, they need to be loaded at runtime. To do so, Android offers a `DexClassLoader` to load `jar` or `apk` (android application) files. The code in Listing 6.2 describes, how an EBP can be dynamically loaded at runtime:

```
1 // Create DexClassLoader
2 DexClassLoader dl = new DexClassLoader(dexPath, optOutPath,
3 null, this.getClass().getClassLoader());
4
5 // Load EBP from given class path and cast to interface
6 ExecutableBusinessProcess newEBP =
7 (ExecutableBusinessProcess)dl.loadClass(classPath)
8                                 .newInstance() ;
```

Listing 6.2: Loading EBPs dynamically at Runtime

Line 2 instantiate a new DexClassLoader for the respective `jar` or `apk` file, that inherits the sources of the requested EBP with following parameters [37]:

- `dexPath`: One or multiple file paths to `jar` or `apk` files with demanded classes.
- `optimizedDirectory`: File path to a folder, where the `DexClassLoader` can store optimized `dex` files (optOutPath).
- `libraryPath`: List of native library directories (`null` in the example)
- `parent`: The parent `ClassLoader` (`this.getClass().getClassLoader()`)

After creation, a `DexClassLoader` is used like any other Java class loader. Line 7 and 8 show the loading of a class and instantiation of an object.

For an EBP to be usable, several requirements have to be satisfied:

1. The android application `package name` of the EBP must be equal to the package path of the EBP's main class.
2. The node using the EBP must specify the correct `implementation class path`.
3. The EBP application must be installed on the Android smart mobile device.
4. The EBP main class must either extend `AbstractExecutableBusinessProcess` or implement `ExecutableBusinessProcess`.

A small example: Let `com.example.TestEBP` be the full qualified name for the EBP main class. Hence, the node in the process model must have stored `com.example.TestEBP` in its `implementationClass` attribute. However, the EBP application's package name is only `com.example`.

A fully commented version of EBP loading source code in A.1 exemplarily shows the whole process of loading an EBP and illustrates the need for given conventions.

6.2.3 Development Challenges

Since EBPs are normal Android applications, they have their own `Context`. Therefore, an EBP developer has to deal with two contexts: the `EBP context` and the `engine context`. The EBP context must be used to access resources of the EBP, like layouts, strings, or images. Trying to use the engine context for accessing EBP resources may end in retrieving the wrong resource or an exception. Therefore, it is essential to use a `LayoutInflater` with the respective `EBP context`, so references on resources in XML layout files are correctly resolved. In contrast, the `engine context` must be used for runtime information (e.g., `Locale`, and `Theme`). Further, the `engine context` is used to add `Fragments` to the current running `Activity`.

The `AbstractExecutableBusinessProcess` class provides static access to both Android `Context` as well as a `LyoutInflater` with the correct `Context`. However, if custom XML tags are used in layout files (e.g., from android support libraries), a `LayoutInflaterFactory` must be implemented and added to the predefined `LayoutInflater`. Otherwise, the `LayoutInflater` cannot load the classes mapped to these tags. An example factory is shown in Source A.2. Figure 6.2 illustrates the problem. The GUI classes of an EBP for its custom made XML tags in layout files are loaded with the respective `DexClassloader`. However, the *LayoutInflater* was loaded by another `Classloader`. As a result, the *LayoutInflater* cannot load the required classes, because its `Classloader` can not access them. In contrast, a `LayoutInflaterFactory` loaded by the respective `DexClassloader` has access to the required classes.

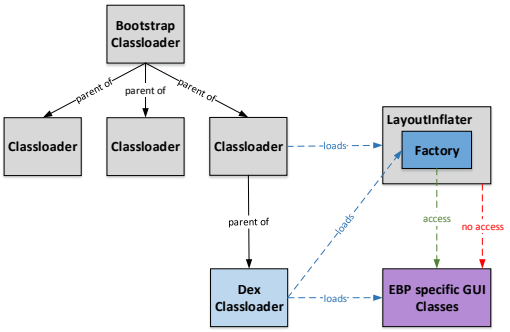

Figure 6.2: Problem of Classloaders

6.3 Object-Relational Mapping

GreenDAO is an open source object relational mapper project, initialized by greenrobot [15]. It uses Android's built-in SQLite database, mapping its tables to common Java objects.

Figure 6.3: GreenDAO Overview [15]

6.3.1 Schema Generation

GreenDao lets the developer *program* their database tables, i.e., the application developer has to write a Java program, that describes the database tables of A.3 [38]. The meta model used for generation is based on database table structure of Figure 6.4.

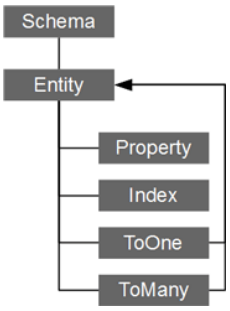

Figure 6.4: GreenDAO Meta Model [38]

The generator automatically creates POJO and DAO classes, as well as a database table manager class. Further, the greenDAO Core Lib must be imported into the Android application. All actors are shown in Figure 6.5.

6.3.2 Database Communication

The database can be accessed in 3 different ways [40]:

1. Direct DAO access: Calling a `load()` method from a responsible DAO object.
2. QueryBuilder queries: Create a query using the provided `QueryBuilder`.
3. Raw queries: Write a highly customized SQLite query as `java.lang.String`.

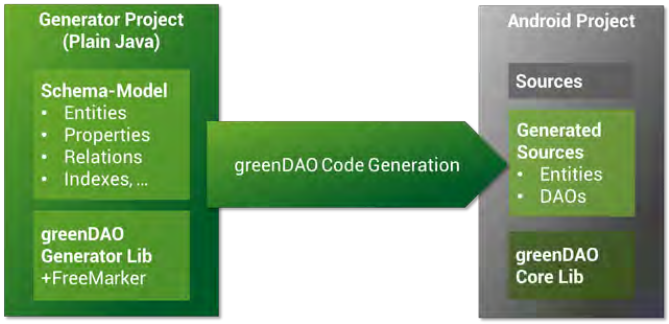

Figure 6.5: GreenDAO Generation Actors [39]

Source A.4 shows all 3 access types. Note that a DAO is only available through the generated `DaoSession` class.

6.4 Rule Evaluation Process

The rule evaluation process is performed by *Apache Commons JEXL*. The latter stands for *Java Expression Language*, and is a scripting language, inspired by Apache Velocity and the Expression Language defined in the Java Server Pages Standard Tag Library version 1.1 (JSTL) and Java Server Pages version 2.0 (JSP) [41]. This Section describes the use of the JEXL library and describes how it is embedded into the process of evaluating rules during runtime. Section 6.4.1 illustrates the general use of JEXL, whereas Section 6.4.2 applies JEXL specifically to the Rule Evaluation Process.

6.4.1 General Usage

Generally, JEXL has three actors for scripting actions, shown in Figure 6.6 [42]:

- *JexlEngine*: The main protagonist, processing the passed Script or Expression.
- *JexlContext*: A context for the JexlEngine, most likely a key-value store.
- *Expression/Script*: The script to be processed by JexlEngine. Essentially, a Script consists of multiple Expressions. Expressions are written in a JEXL specific Syntax, described in [43].

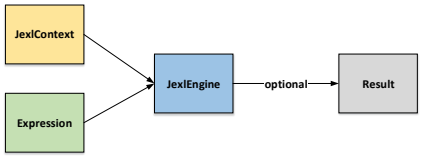

Figure 6.6: JEXL Actors

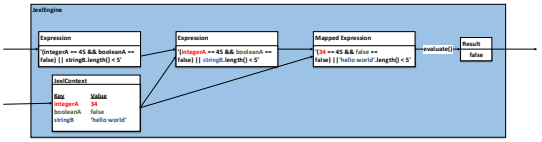

Figure 6.7: JEXL Processing Example

The `JexlContext` and an `Expression` are passed to the `JexlEngine`, which then processes the `Expression`, returning a possible result for the given input. The following example shows the whole process of JEXL processing in Figure 6.7. Let the `Expression` be:

```
1 (integerA == 45 \&\& booleanA == false) || stringB.length() < 5
```

Further, following key-value pairs are stored within the `JexlContext`:

```
1 (integerA , 34)
2 (booleanA , false)
3 (stringB , 'hello world')
```

First, JEXL recognizes variables in the given `Expression` and substitutes them with the respective value in given `JexlContext`. In the given example, substituting `integerA` with `34`, `booleanA` with `false` and `stringB` with `'hello world'`. Afterwards, it processes the given `Expression` which, in this case, is a Boolean expression. Therefore, JexlEngine evaluates it automatically and returns `false` as the result. Note that JexlEngine calls the `length()` method of `java.lang.String`. However, calling class methods is not restricted to strings. As a matter of fact, JEXL can call any available method from any class, as long as a object reference is stored in the `JexlContext`, granted that key and `Expression` variable match [42].

6.4.2 Rule Evaluation

This Section applies the introduced library from Section 6.4.1 to the Rule Evaluation Process. Therefore, the overall process of rule evaluation is described. Subsequently, the usage of rule functions is outlined.

Evaluation Process

For the rule evaluation process, illustrated in Figure 6.8, several data is required:

- *Execution Logs*: Log of the executed process instance, describing the order of execution, along with other data (e.g., timestamps).

- *Data Values*: All values for data elements ever written for a given process instance.
- *Data Elements*: The data elements involved in this process instance.
- *Rule Condition*: The rule condition (e.g., expression) to be tested.
- *Rule Functions*: Functions, that may be used when testing the condition.

A rule condition is a JEXL string as described in Section 6.4.1. However, two requirements must be met: First, variables must be named after their data element. Second, if a function is used, the variable name before the function name must be the full qualified name of the class implementing this function.

Because of the multiple occurrences of data values for one data element, only the latest values are stored in the JexlContext.

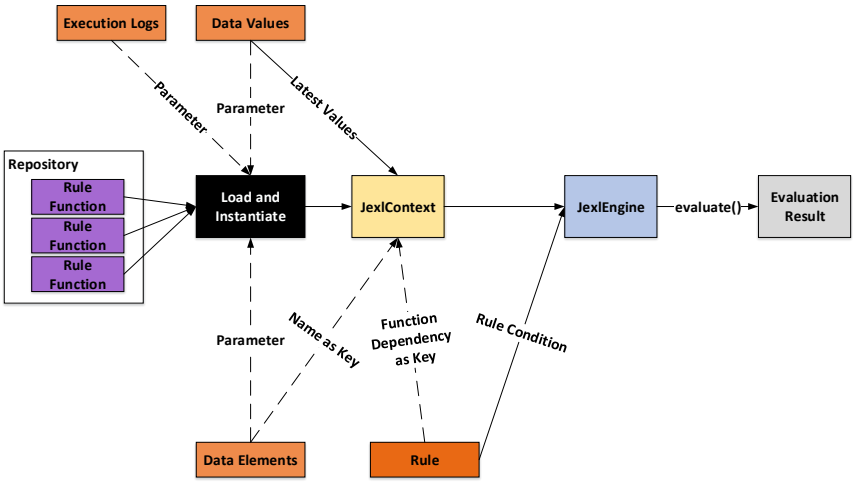

Figure 6.8: Rule Evaluation Process Overview

When evaluating a rule, the rule condition and JexlContext are given to the JexlEngine. The result of of the evaluation is a `Boolean`. This `Boolean`, combined with the evaluated rule and other process instance information is wrapped in an `EvaluationReport`, which is returned to the caller.

Rule Functions

WOtANAnalysis uses rule functions for analysis. These functions are implemented in Java, translating every function in a Java method. WOtAN provides an abstract class for rule function development, named `AbstractRuleOperation`. It provides a common constructor for all classes that may receive the execution logs, data values, and data elements passed. This data is needed for dynamically loading these type of classes at runtime with a DexClassLoader, as described in 6.2.2.

However, these classes are stored within an external repository, as one can see in Figure 6.8 and are not installed as standalone Android applications. Therefore, to find the correct class at runtime, the name of the jar file must be the same as the full qualified name of the implementing class. To find the relevant function classes for each rule, every rule has a `dependency` attribute, declaring a list of full qualified class names. The loaded classes are instantiated as an object, and set in the JexlContext. Note that the full qualified class name is used as key. The functions are now visible to JEXL and ready to use.

7

Application Scenario: Mobile Data Collection Application

This chapter introduces an application scenario for WOtAN. Therefore, Section 7.1 introduces the *QuestionSys* project. Following, Section 7.2 briefly discusses the system's overall architecture. At last, Section 7.3 describes WOtAN as process engine within this system.

7.1 Introduction

Started in September 2013 QuestionSys [44] aims at providing a generic approach to support the lifecycle of mobile data collection applications, as shown in Figure 7.1. **P1** is the first step in the lifecycle where a questionnaire is created. Subsequently, **P2** is responsible for the deployment of created questionnaires. Afterwards, **P3** supports the *flexible data collection* of deployed questionnaires. The next step, **P4**, focuses on evaluation and analysis of collected data. In the last step, **P5**, the archiving of enacted and evaluated questionnaires is done.

QuestionSys thereby relies on a process-driven approach [45], using a process meta model as questionnaire. The latter, is executed on a flexible process engine running on a smart mobile device.

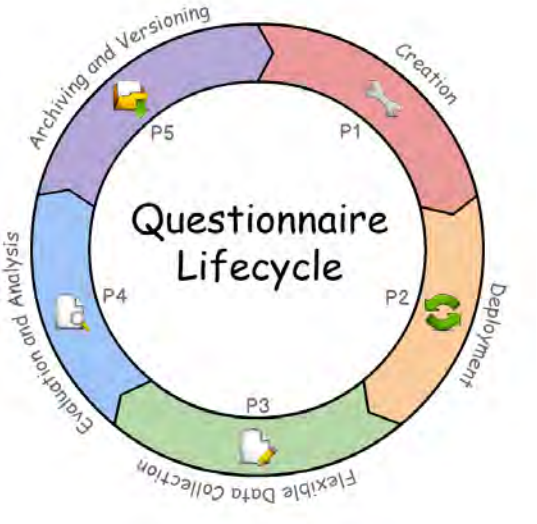

Figure 7.1: Mobile Data Collection Application Lifecycle [44]

7.2 System Architecture

QuestionSys is a system consisting of four main components [36]: *Questioneer*, *QuestionRule*, *Questionizer*, and *Questionnaire*. Following sections briefly describe their tasks. A completed architecture overview is shown in Figure 7.2.

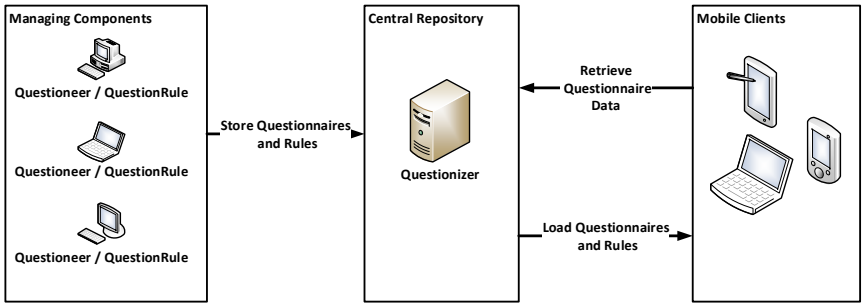

Figure 7.2: QuestionSys Overall System Architecture

7.2.1 Questioneer

This is the questionnaire configuration component, enabling domain experts to flexibly create and design electronic questionnaires. It allows for creating modular parts of a questionnaire, e.g., questions, pages, and combining them to a complete questionnaire using techniques from end-user programming. Further, the configurator allows to handle multilingualism as well as to deal with multiple versions of questionnaires. Modeled questionnaires are transferred to the Questionizer server-component.

7.2.2 QuestionRule

QuestionRule [36] is a rule configurator, enabling domain experts to create rules for the subsequent analysis of a given questionnaire. QuestionRule allows graphical modeling of rules, by using a tree-based drag & drop approach. Created rules are also transferred to the Questionizer server-component.

7.2.3 Questionizer

Questionizer is the management back-end component, responsible for centralized storage and distribution of both questionnaires and their rules. Further, results of completed questionnaires are returned to this component for a rule-based analysis.

7.2.4 Questionnaire

The Questionnaire component is a client that runs on smart mobile devices or web browsers. It is able to load and execute questionnaires. However, due to the fact that questionnaires are mapped to process models, each client uses a lightweight process engine to perform given questionnaires on smart mobile devices. When performing questionnaires in a web browser, a server side process engine may be used.

7.3 Using WOtAN in the Context of Mobile Data Collection Applications

As described in Section 7.2.4, the Questionnaire component is a client for enacting questionnaires on smart mobile devices. Recall that the QuestionSys framework uses process technology to model a questionnaire. Therefore, WOtANExecution is used as the flexible, lightweight process engine for Questionnaire on Android smart mobile devices. Further, WOtAN does not only provide the functionality to enact questionnaires with the WOtANExecution module, but also allows to evaluate collected data. This is achieved using the WOtANAnalysis module.

This section describes the execution of questionnaires as well as its analysis. First, Section 7.3.1 illustrates the overall project architecture of Questionnaire for the Android platform. Then, Section 7.3.2 describes the execution component of Questionnaire. Subsequently, Section 7.3.3 demonstrates the rule-based analysis component. At last, Section 7.3.4 illustrates the Executable Business Processes used to automatically generate and render the user interface to interact with the questionnaire.

7.3.1 Overall Architecture

Since both WOtANExecution and WOtANAnalysis are used within Questionnaire, it is clear that both modules plus the WOtANCore are required. Therefore, a combined database for WOtANExecution and WOtANAnalysis would be best practice. However, to prove the standalone capabilities of WOtAN modules, separated databases for each module are created. Additionally, both modules are completely separated stacks which are combined at application layer, as illustrated in Figure 7.3.

7.3.2 Executing Questionnaires using WOtANExecution

Since a questionnaire is nothing else but a process model [8], WOtANExecution simply stores a given questionnaire as process model. In order to enact the latter, it uses the concepts and implementation aspects described in Section 5.2 and Section 6. However,

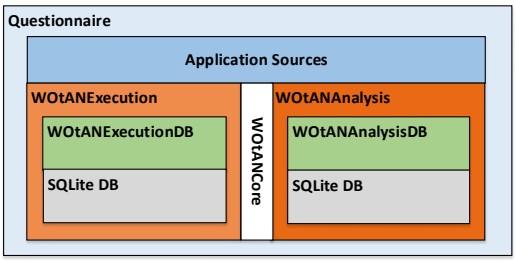

Figure 7.3: Overall Questionnaire Architecture

QuestionSys specific aspects need to be considered. First, every XOR branch in the process model has an empty node i.e., containing no executable business process. This workaround results from the ADEPT process model structure, which does not allow multiple empty branches. However, for easier XOR branch conditions, such structure is a desired feature, mitigate the use of combined, over the top complex Boolean terms. Further, a data element is a process variable mapped to one specific question. Despite its type, a `java.lang.String` type, it is a JSON string with a complex underlying data structure, not further explained in this thesis.

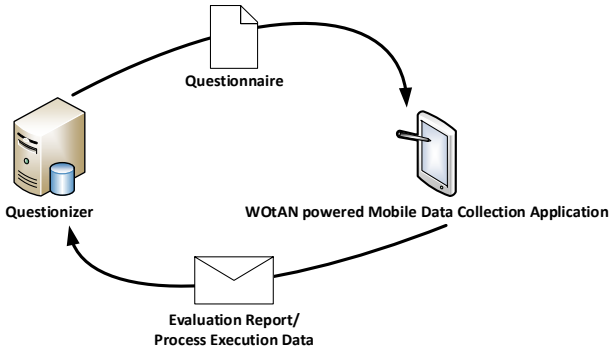

Figure 7.4: Questionnaire Overview

Generally, as illustrated in Figure 7.4, questionnaires are transferred to the smart mobile device. These questionnaires are executed using the developed lightweight process

engine, generating process execution data. This data can be evaluated using WOtAN-Analysis and sent back to the server.

7.3.3 Evaluating Questionnaires using WOtANAnalysis

Evaluating the collected data of executed questionnaires is achieved using WOtANAnalysis, which relies on techniques described in Sections 5.3 and 6.4. Therefore, rules defined by QuestionRule are stored in WOtANAnalysis' database. Applying rules on process execution data results in an Evaluation Report. This report may be displayed within the smart mobile application or transferred to the server. Rule functions, as described in [36] are stored in a local repository and loaded at runtime if needed.

7.3.4 Page Executable Business Process

The *Page Executable Business Process*(*PageEBP*) is the main component of page execution. It is responsible for visualization and input verification. Basically, all classes from the previous Questionnaire project are encapsulated into the WOtANExecution EBP structure. Further, the new `LayoutInflater` and `Android Context` are propagated through all classes to ensure resource and layout use. Further, as a practical part of their bachelor thesis', [46] and [47] styled the page component and Questionnaire GUI in Androids material design, as shown in Figure 7.5. The result is a reusable modular EBP that automatically creates the pages within a questionnaire which is controlled by WOtANExecution. On application level, this EBP is simply used as a `Fragment`, which can be added to any layout.

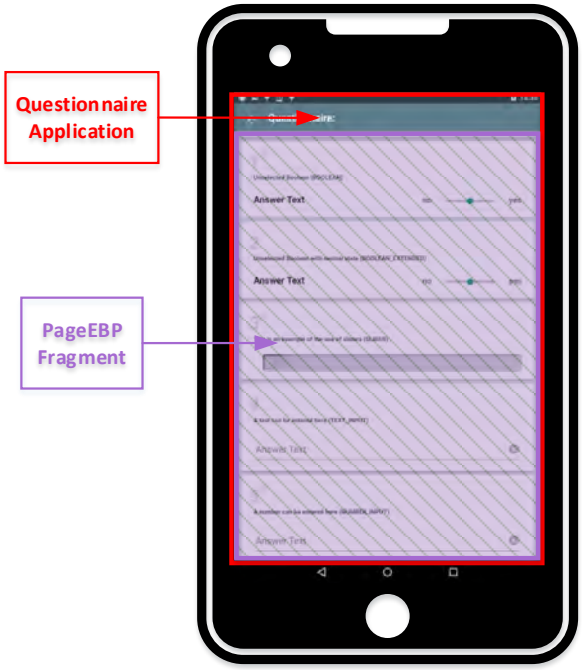

Figure 7.5: Questionnaire PageEBP Fragment

8

Conclusion

This thesis introduces aspects of business process execution and analysis using evaluation rules as modules of *Workflows on Android* (*WOtAN*). The latter is a modular framework that aims to supports the *BPM lifecycle* (of Figure 8.1) on Android devices without the need for external communication.

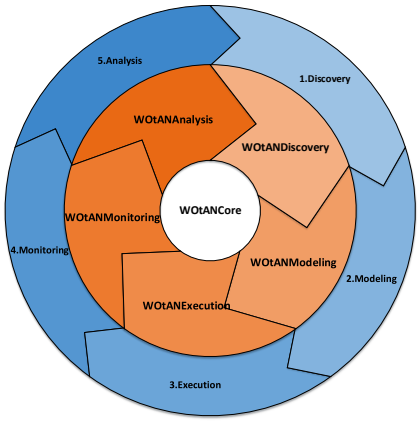

Figure 8.1: The BPM Lifecycle with Respective WOtAN Modules

First, *WOtANExecution*, the module responsible for business process execution, was presented. This thesis emphasizes the flexibility to dynamically extend the functionality of the engine. The latter, thereby, uses software templates, so called *executable business processes*, which are flexibly loaded at runtime on demand.

Next, *WOtANAnalysis*, the module responsible for business process analysis, was discussed. The latter uses the *process execution data* provided by WOtANExecution as well as previously defined *evaluation rules* to automatically create *evaluation reports*.

Like in the other module, *evaluation rule functions* are fetched and loaded automatically at runtime from a rule repository.

The dynamic extension aspect in WOtANExecution and WOtANAnalysis allows to change or add implementations for both executable business processes and evaluation rule functions, without recompiling and redeploying the application. Further, a high-level API is provided to application developers, ensuring an easy-to-use framework.

8.1 Vision

As mentioned throughout this thesis, WOtAN introduced business process execution and analysis to a framework supporting the whole BPM lifecycle. Hence, the missing steps of the BPM lifecycle must be designed and implemented. This results in three additional modules for WOtAN: WOtANModeling, WOtANDiscovery, and WOtANMonitoring:

WOtANDiscovery shall help to identify business processes in companies. Therefore, it shall support different strategies, such as *user stories* [48], or *process mining* [49]. However, the practicability of computing intensive process mining algorithms on smart mobile devices must be evaluated first.

WOtANModeling is responsible for business process modeling. Usability aspects have already been shown in [50]. However, the concept lacks aspects of working offline, as it directly communicates with a server.

WOtANMonitoring shall provide overviews regarding all currently running business process instances. Therefore, WOtANMonitoring shall inherit WOtANExecution's ability to log process executions.

In addition to the above described work, further refinements of WOtANExecution and WOtANAnalysis are needed:

WOtANExecution currently lacks the feature of dynamic data value types. It is currently restricted on hard coded value types like `Integer` or `Date`. Furthermore, *WOtANAnalysis* is currently restricted to rule-based analysis only. A more generic core infrastructure with extension points for other analysis methods should be designed. Another analysis

method, for example, is the use of *key performance indicators* (*KPIs*) to analyze process execution performance [51].

At last, the problem of limited battery and memory usage [31], as well as computation performance, should be tackled for the already realized modules. This is necessary for WOtAN to evolve into a real world utilizable approach, instead of staying a proof-of-concept.

Bibliography

[1] Smith, Howard and Fingar, Peter: Business Process Management: The Third Wave. (2003)

[2] Manfred Reichert and Barbara Weber: Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies. Springer, Berlin-Heidelberg (2012)

[3] Jens Kolb and Manfred Reichert: Data Flow Abstractions and Adaptations through Updatable Process Views. In: 28th Symposium on Applied Computing (SAC'13), 10th Enterprise Engineering Track (EE'13), ACM Press (2013) 1447–1453

[4] Peter Dadam and Manfred Reichert: The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support. Technical Report UIB-2009-01, University of Ulm, Ulm (2009)

[5] Rüdiger Pryss: Robuste und kontextbezogene Ausführung mobiler Aktivitäten in Prozessumgebungen. PhD thesis, University of Ulm (2015)

[6] Andreas Lanz and Barbara Weber and Manfred Reichert: Time patterns for process-aware information systems. Requirements Engineering **19** (2014) 113–141

[7] Rüdiger Pryss and Julian Tiedeken and Manfred Reichert: Managing Processes on Mobile Devices: The MARPLE Approach. In: CAiSE'10 Demos. (2010)

[8] Johannes Schobel and Marc Schickler and Rüdiger Pryss and Manfred Reichert: Process-Driven Data Collection with Smart Mobile Devices. In: 10th International Conference on Web Information Systems and Technologies (Revised Selected Papers). Number 226 in LNBIP. Springer (2015) 347–362

[9] Valiente, Pablo and van Heijden, H: A method to identify opportunities for mobile business processes. Stockholm School of Economics, SSE/EFI Working Paper Series in Business Administration **10** (2002)

[10] Ambler, Scott W: Mapping objects to relational databases: What you need to know and why. Ronin International (2000)

[11] Gray Watson: ORMLite - Getting Started. (http://ormlite.com/javadoc/ormlite-core/doc-files/ormlite_1.html#Starting-Class) Accessed: 2016-02-05.

[12] Satya Narayan: Sugar ORM - Design Your Entities. (http://satyan.github.io/sugar/creation.html) Accessed: 2016-02-05.

[13] Doctrine Team: doctrine - Getting Started: Database First. (http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/tutorials/getting-started-database.html) Accessed: 2016-02-05.

[14] SensioLabs: The Cookbook Doctrine - How to Generate Entities from an Existing Database. (http://symfony.com/doc/current/cookbook/doctrine/reverse_engineering.html) Accessed: 2016-02-05.

[15] greenrobot: greenDAO. (http://greenrobot.org/greendao) Accessed: 2016-02-05.

[16] Google: Android Developers, Providing Resources. (http://developer.android.com/guide/topics/resources/providing-resources.html) Accessed: 2016-02-05.

[17] Google: Android Developers, App Manifest. (http://developer.android.com/guide/topics/manifest/manifest-intro.html) Accessed: 2016-02-05.

[18] Google: Android Developers, Activities. (http://developer.android.com/guide/components/activities.html) Accessed: 2016-02-05.

[19] Google: Android Developers, Fragments. (http://developer.android.com/guide/components/fragments.html) Accessed: 2016-02-05.

[20] Google: Android Developers, Intents and Intent Filters. (http://developer.android.com/guide/components/intents-filters.html) Accessed: 2016-02-05.

[21] Google: Android API, Bundle. (http://developer.android.com/reference/android/os/Bundle.html) Accessed: 2016-02-05.

[22] Reichert, Manfred and Dadam, Peter: ADEPTflex-Supporting dynamic changes of workflows without losing control. Journal of Intelligent Information Systems **10** (1998) 93–129

[23] Manfred Reichert: Dynamische Ablaufänderungen in Workflow-Management-Systemen. PhD thesis, University of Ulm (2000)

[24] Ulrich Kreher: Konzepte, Architektur und Implementierung adaptiver Prozessmanagementsysteme. PhD thesis, University of Ulm (2014)

[25] Manfred Reichert and Peter Dadam: Enabling Adaptive Process-aware Information Systems with ADEPT2. In Cardoso, J., van der Aalst, W., eds.: Handbook of Research on Business Process Modeling. Information Science Reference, Hershey, New York (2009) 173–203

[26] Peter Dadam and Manfred Reichert and Stefanie Rinderle-Ma and Kevin Goeser and Ulrich Kreher and Martin Jurisch: Von ADEPT zur AristaFlow BPM Suite - Eine Vision wird Realität: "Correctness by Construction" und flexible, robuste Ausführung von Unternehmensprozessen. Technical Report UIB-2009-02, University of Ulm, Ulm (2009)

[27] Red Hat: jBPM. (`http://www.jbpm.org/`) Accessed: 2016-02-05.

[28] Object Management Group: BMPN 2.0 Specification. (`http://www.omg.org/spec/BPMN/2.0/`) Accessed: 2016-02-05.

[29] Red Hat: jBPM, User Guide-Core Engine API. (`http://docs.jboss.org/jbpm/v6.3/userguide/ch05.html`) Accessed: 2016-02-05.

[30] Red Hat: jBPM, User Guide-Domain Specific Processes. (`http://docs.jboss.org/jbpm/v6.3/userguide/ch21.html`) Accessed: 2016-02-05.

[31] Rüdiger Pryss and Julian Tiedeken and Ulrich Kreher and Manfred Reichert: Towards Flexible Process Support on Mobile Devices. In: Proc. CAiSE'10 Forum - Information Systems Evolution. Number 72 in LNBIP, Springer (2010) 150–165

[32] Reisser, A.: Technische Konzeption und Realisierung einer dynamisch generierten Anwendung für prozess-orientierte Fragebögen am Beispiel der mobilen Android Plattform. Diploma's thesis, University of Ulm (2014)

[33] Manfred Reichert and Peter Dadam and Stefanie Rinderle-Ma and Martin Jurisch and Ulrich Kreher and Kevin Goeser: Architecural Principles and Components of Adaptive Process Management Technology. In Heinzl, A., Dadam, P., Lockemann, P., Kirn, S., eds.: PRIMIUM - Process Innovation for Enterprise Software. Number P-151 in Lecture Notes in Informatics (LNI). Koellen-Verlag (2009) 81–97

[34] Jeston, John and Nelis, Johan: Business process management. Routledge (2014)

[35] Peter Dadam and Manfred Reichert and Stefanie Rinderle-Ma and Kevin Goeser and Ulrich Kreher and Martin Jurisch: Von ADEPT zur AristaFlow BPM Suite - Eine Vision wird Realität: "Correctness by Construction" und flexible, robuste Ausführung von Unternehmensprozessen. EMISA Forum **29** (2009) 9–28

[36] Bernd Mertesz: Concept and Implementation of a Rule Component Enabling Automatic Analysis of Process-Aware Questionnaires. Master's thesis, University of Ulm (2014)

[37] Google: Android API - DexClassLoader. (`http://developer.android.com/reference/dalvik/system/DexClassLoader.html`) Accessed: 2016-02-05.

[38] greenrobot: greenDAO - Modelling Entities. (`http://greenrobot.org/greendao/documentation/modelling-entities`) Accessed: 2016-02-05.

[39] greenrobot: greenDAO - Introduction. (`http://greenrobot.org/greendao/documentation/introduction`) Accessed: 2016-02-05.

[40] greenrobot: greenDAO - Query Data Base. (`http://greenrobot.org/greendao/documentation/queries`) Accessed: 2016-02-05.

[41] The Apache Software Foundation: Apache Commons JEXL. (`http://commons.apache.org/proper/commons-jexl`) Accessed: 2016-02-05.

[42] The Apache Software Foundation: Apache Commons JEXL - Evaluating Expressions. (http://commons.apache.org/proper/commons-jexl/reference/examples.html#Evaluating_Expressions) Accessed: 2016-02-05.

[43] The Apache Software Foundation: Apache Commons JEXL - JEXL Syntax. (http://commons.apache.org/proper/commons-jexl/reference/syntax.html) Accessed: 2016-02-05.

[44] Institute of Databases and Information Systems, University of Ulm: QuestionSys - A Generic and Flexible Questionnaire System Enabling Process-Driven Mobile Data Collection. (http://www.uni-ulm.de/in/iui-dbis/forschung/projekte/questionsys.html) Accessed: 2016-02-05.

[45] Johannes Schobel and Marc Schickler and Rüdiger Pryss and Fabian Maier and Manfred Reichert: Towards Process-Driven Mobile Data Collection Applications: Requirements, Challenges, Lessons Learned. In: 10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps. (2014) 371–382

[46] Andrea Reidel: Entwicklung eines Designkonzepts für unterschiedliche Anwendungsszenarien eines generischen Fragebogensystems. Bachelor's thesis, University of Ulm (2015)

[47] Florian Hofherr: Adaptierung und Neugestaltung einer Android Anwendung unter Berücksichtigung der Material Design Richtlinien. Bachelor's thesis, University of Ulm (2016)

[48] Pettersen, Thomas Bech and Carlsen, Steinar and Coll, Gunnar John and Sem, Helle Frisak: Bottom-up Process Discovery using Knowledge Engineering Techniques. Empowering Knowledge Workers: New Ways to Leverage Case Management (2013) 43

[49] Van Der Aalst, Wil: Process mining: discovery, conformance and enhancement of business processes. Springer Science & Business Media (2011)

[50] Matthias Dapper: Implementation of a Multi-Touch, Gesture-based Process Modeling Component for Apple iPad. (2012)

[51] Weber, Al and Thomas, Ivara Ron: Key performance indicators. Measuring and Managing the Maintenance Function, Ivara (2005)

A

Sources

This chapter shows some important source codes.

A.1 Dynamic EBP Loading

```
1  /* This method loads and externally stored EBP via
2   * DexClassLoaders. Therefore it uses Android's PackageManager
3   * to find the installed EBP app over its package name.
4   */
5  public ExecutableBusinessProcess getEBP(Node holdsEbp){
6    ExecutableBusinessProcess newEBP = null;
7
8    //EBP full qualified name ("com.example.TestEBP")
9    String classPath = holdsEBP.getImplementingClasspath();
10
11   //EBP app package name ("com.example")
12   String packagePath = classPath.substring(0,classPath.lastIndexOf("."));
13
14   //Android PackageManager class to find EBP app
15   PackageManager pm = context.getPackageManager();
16
17   //Get info for app with given package name ("com.example")
18   ApplicationInfo appInfo = pm.getApplicationInfo(packagePath,PackageManager.GET_META_DATA);
19
20   //Store path to apk file in String
21   String dexPath = appInfo.sourceDir;
22
23   // Create folder for optimized dex files
24   String optOutPath = ...;
25
26   //Create DexClassLoader for needed EBP
27   DexClassLoader dl = new DexClassLoader(dexPath,
28     optOutPath, null, this.getClass().getClassLoader());
29
30   //Load and instantiate EBP class
31   newEBP = (ExecutableBusinessProcess)dl.loadClass(classPath).newInstance() ;
32
33   return newEBP;
34 }
```

Listing A.1: Dynamic EBP Loading

A.2 Example Layout Inflater Factory

```
1  public class CustomLayouterFactory implements
2  LayoutInflater.Factory {
3
4    private final String TAG_CARDVIEW =
5      "android.support.v7.widget.CardView";
6
7    private final String TAG_DOCVIEW =
8      "com.bluejamesbond.text.DocumentView";
9
10   private final String TAG_TEXTINPUT =
11     "android.support.design.widget.TextInputLayout";
12
13   @Override
14   public View onCreateView(String name,
15     Context context, AttributeSet attrs) {
16   View v = null;
17   if(name.equals(TAG_CARDVIEW)){
18     v = new CardView(context, attrs);
19   }else if(name.equals(TAG_DOCVIEW)){
20     v = new DocumentView(context, attrs);
21   }else if(name.equals(TAG_TEXTINPUT)){
22     v = new TextInputLayout(context,attrs);
23   }
24   return v;
25 }
```

Listing A.2: Example Layout Inflater Factory

A.3 greenDAO Schema Generation Code

```
1  /* This main method generates a DB schema
2   * with one table in it, named "Node".
3   * This is a simplified version, where some lines
4   * are semantically rubbish.
5   */
6
7  public static void main(String[] args){
8    // Create DB schema
9    Schema schema = new Schema(1, package_name_for_gen_classes);
10
11   // Add new db table
12   Entity nodes = schema.addEntity("Node");
13   //Add attributes to table
14   //First: pk attribute with auto increment
15   Property p1 = nodes.addLongProperty("internalid")
16     .primaryKey().autoincrement().getProperty();
17
18   /* Process model id is Java UUID but must be split into longs
19    * for db storage. (No UUID storage type)
20    */
21   Property p2 = nodes.addLongProperty("processmodelidhigh");
22   Property p3 = nodes.addLongProperty("processmodelidhigh");
```

```
23
24    //Some other attributes...
25    nodes.addStringProperty("name");
26    nodes.addLongProperty("topologicalid");
27    nodes.addBooleanProperty("a boolean");
28
29    /* Create q 'unique combined pk' by using a unique index.
30     * This work around is needed for greenDAOs missing abillity
31     * to create combined pks.
32     */
33    Index i = new Index();
34    i.addProperty(p1);
35    i.addProperty(p2);
36    i.addProperty(p3);
37    i.makeUnique();
38    nodes.addIndex(i);
39
40    // Generate
41    new DaoGenerator().generateAll(schema, storage_path);
42  }
```

Listing A.3: greenDAO Schema Generation Code

A.4 Database Access with GreenDAO

```
1  DevOpenHelper helper = new DaoMaster.DevOpenHelper(context,
2      "engine-db", null);
3  db = helper.getWritableDatabase();
4  daoMaster = new DaoMaster(db);
5  daoSession = daoMaster.newSession();
6  // Get DAO object
7  HelloDao testDao = daoSession.getHelloDao();
8
9  //Direct DAO access
10 //Load for pk
11 Hello singleResult = testDao.load(24l);
12 //Load all values
13 List<Hello> multiResult = testDao.loadAll();
14
15 //QueryBuilder search for objects with attribute name == hi
16 List<Hello> builderResult =
17 testDao.queryBuilder().where(Property.name.eq("hi")).list();
18
19 //Raw query
20 Query query = testDao.queryRawCreate(
21   ", GROUP G WHERE G.NAME=? AND T.GROUP_ID=G._ID", "admin");
```

Listing A.4: Database Access with GreenDAO

List of Figures

2.1 ORM Overview 6
2.2 Calling Another Activity Using Intents 10
2.3 Example Process with Additional Annotations 11
2.4 Node States Chart [25] 12

3.1 jBPM Overview [29] 15
3.2 General MAPRLE Architecture [31] 16

5.1 WOtAN Modules Integrated in BPM Lifecycle 23
5.2 WOtAN 3-Tier Architecture 25
5.3 Process Model Example 26
5.4 Node States Overview [25] 28
5.5 WOtANExecution Manager Hierarchy Overview 29
5.6 Runtime Manager to EBP Communication 30
5.7 Coupling the Engine with Standalone EBPs 31
5.8 Start Node Example 32
5.9 Rule Example [36] 33
5.10 Rule Evaluation Overview 33

6.1 EBP Overview 37
6.2 Problem of Classloaders 39
6.3 GreenDAO Overview [15] 40
6.4 GreenDAO Meta Model [38] 40
6.5 GreenDAO Generation Actors [39] 41
6.6 JEXL Actors 42
6.7 JEXL Processing Example 42
6.8 Rule Evaluation Process Overview 44

7.1 Mobile Data Collection Application Lifecycle [44] 48
7.2 QuestionSys Overall System Architecture 48

7.3 Overall Questionnaire Architecture 51
7.4 Questionnaire Overview 51
7.5 Questionnaire PageEBP Fragment 53

8.1 The BPM Lifecycle with Respective WOtAN Modules 55

List of Tables

2.1 Activation Rules 13

3.1 Key-Features for Mobile Process Engines 18

5.1 Node Types [23] 27

Name: Wolfgang Wipp Matriculation Number: 698982

Declaration

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

Ulm, the .....

Wolfgang Wipp