universität wien

MASTERARBEIT

Titel der Masterarbeit

„Mobile Application Discovery through Conceptual Models“

verfasst von

Jasmin Brakmic, BSc

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2013

Studienkennzahl lt. Studienblatt:	A 066 926
Studienrichtung lt. Studienblatt:	Wirtschaftsinformatik
Betreuerin / Betreuer:	o. Univ.-Prof. Dr. Prof. h. c. Dimitris Karagiannis

Acknowledgments

First of all, I would like to thank God for the accomplishment of this master thesis. I would like to express deepest thanks to my parents, who gave me this opportunity and supported me with everything I need in order to complete my studies at the University of Vienna, and also thank my brother and his family and all my friends who encouraged and supported me in the past five years of my study.

My sincere appreciation I give to my supervisor o. Univ.-Prof. Dr. Prof. h. c. Dimitris Karagiannis who guided me with suggestions and advices during my research and writing of this master thesis. Special thanks to Dr. Robert Buchmann and Dipl.-Ing. Niksa Visic who invested their time and effort in reading of this master thesis and giving me research advices for writing. I would also like to thank all my colleagues at the department of Knowledge Engineering who gave his contribution to the completion of this master thesis.

Last but not the least, I expand my thanks to all people which I did not mention explicitly, but supported me in any way in this stressful and exhausting time of my life.

Zusammenfassung

Die Entwicklung von mobilen Anwendungen (Apps) verweist einen steigenden Verlauf im Bereich der Wirtschaftsinformatik auf. Im Vergleich zum Entwicklungsprozess von Desktop Anwendungen, ist jedoch der Entwicklungsprozess von mobilen Apps oftmals noch immer mit geringer Effizienz, Wiederverwendbarkeit und Schnelligkeit verbunden. Konzeptuelle Modelle welche heutzutage herkömmlich sind in der Entwicklung von Desktop Anwendungen, finden nur geringe Anwendung in der Entwicklung von mobilen Apps. Die verwendeten Konzeptuellen Modelle ermöglichen es dem Entwickler nur teilweise Code zu generieren, unter anderem die Generierung von generischen Frames, leeren Stubs von Klassen und Templates. Aus diesem Grund, ist es notwendig ein Repository für das Speichern und den Zugang zu einer großen Anzahl von entwickelten mobilen Apps zu ermöglichen. Im Laufe dieser Arbeit wird ein einzigartiger Suchmechanismus mit unterschiedlichen Suchalgorithmen vorgeschlagen, um passende mobile Apps basierend auf den zur Verfügung gestellten Eingabekriterien aus dem Repository zu finden. Unter bestimmten Umständen sollte es möglich sein dem Entwickler fertig-implementierten Code von mobilen Apps zur Verfügung zu stellen. Diesen Code könnte er entweder vollständig wiederverwenden oder falls notwendig gewisse Zeilen adaptieren und im weiteren Verlauf wiederverwenden.

Abstract

The development of mobile applications (apps) shows a permanent rising trend in the field of business informatics. In contrast to the software development process for desktop apps, the development process of mobile apps still lacks in efficiency, reusability and rapidness. Conceptual models, which are nowadays common in the development process of desktop applications, are rarely used in the development of mobile apps. Those conceptual models only provide the developer with partial code generation options, including the generation of generic frames, empty stubs of classes and templates. Therefore, a repository of mobile apps for storing and providing access to a large amount of developed mobile apps is a necessity. This work proposes a unique search mechanism with different algorithms in order to discover proper mobile apps from the repository, based on the provided search input criteria. Under certain circumstances, it should be possible to provide the developer with a complete implementation of the mobile app in which he would only adapt a few lines of code and reuse the implementation.

Keywords: mobile application, conceptual model, development process, repository, search algorithm

Table of Contents

List of Figures

List of Tables

List of Equations

1 Introduction

1.1 Motivation

The evolution of technologies used in mobile devices facilitated a growing trend in the development of mobile applications (apps)[1]. In the past, because of the missing technology in the area of mobile devices, apps did not play a significant role. Besides, the accessibility to apps has also been challenging for the most of the mobile device users. Nowadays, the improved technology of mobile devices and introduced app repositories (also called application stores) increases the accessibility to apps. Those repositories serve a high number of different apps. Stores like the iOS App Store[2] or the Google Play Store[3] each provide more than 600.000 and the Nokia Store[4] and Microsoft Store[5] each 120.000 different apps. Each of those apps is currently developed from scratch, without any consideration of code reuse from already developed apps. This issue makes the app development process time consuming and resource intensive.

It is also relevant to handle the large amount of different apps, which are stored in app repositories, making search algorithms and mechanisms a necessity. These search algorithms and mechanisms provide different capabilities for the discovery of apps, like keyword-based search, rating-based search or a search based on the number of downloads, etc. In addition to the rich search capabilities of those app repositories, they often lack in the identification of a proper app.

In this work a proper app is described as an app which fulfils the needs and requirements of a developer for the functionality of the app. Last but not least, all of the mentioned app repositories provide only executable apps, which are not suitable for developers who want to find implementation specific information about apps.

[1] Note that in this master thesis the word app is used only for the replacement of mobile applications.

[2] https://itunes.apple.com/en/app/apple-store/id375380948?mt=8

[3] https://play.google.com/about/features/index.html#

[4] http://store.ovi.com/

[5] http://www.windowsphone.com/en-US/store

A repository together with supporting web technologies can be used to enable the reusability of app design mock-ups and code, and also provide the developer with a unique, app content-based search mechanism. The web technology should be built on open standards and the provided search mechanism should support the developer with adequate implementation patterns and app suggestions.

For the purpose of searching apps in this work, the use of conceptual models is proposed, instead of pure text-based search mechanisms that already are present in many app repositories. The aim of conceptual models in this work will not only be restricted to the search mechanism, but also to the specification and documentation of apps and better communication between users of apps, requirements engineers and app developers. The conceptual models are assumed to be created with the app requirements-oriented fragment of the *ComVantage*[6] modelling method. *ComVantage* is a European research project, concerned with supporting processes with a mobile app front end. The *ComVantage* modelling method is aimed to capture among others requirements for such front ends. [1], (public deliverables 3.1.1 and 8.2.1).

1.2 Research Questions and Goals

The primary goal of this thesis is the research of conceptual models as representation form of apps, and the searching and retrieving of app implementation specific information from a repository of apps. This goal can be divided into several research objectives. Those objectives include the following aspects:

- Research about conceptual models which are capable of representing the requirements and interaction of users with apps. This topic also includes the investigation about a modelling language which is used for this purpose.
- Defining and establishing of an app repository which stores and provide implementation specific information for app developers.
- The specification and implementation of a search algorithm for extracting implementation specific information about apps stored in a repository. This search

[6] http://www.comvantage.eu/. Accessed: 29.06.2013.

algorithm takes as input information from the defined conceptual model describing the required app.

As can be derived from the objectives, the aim of this work is to support the development process of apps. In this thesis a scenario is been proposed for which following **preconditions** are relevant:

1. Available models and apps are stored in the repository,
2. The repository is encapsulated behind a published web service developed in this thesis,
3. The models used, have to be created using the *ComVantage* modelling method for apps and the extensions proposed in this thesis.

The **scenario** is assumed as follows:

1. The developer of an app will define the user requirements for an app through conceptual models.
2. The developer sends the conceptual model to the search mechanism.
3. The search mechanism extracts information about the model and searches for possible apps in the repository.
4. The repository sends the code of the available app stored in it back to the developer.
5. The developer tries to identify and select possible implementation information of apps which he can reuse.

Finally, it is important to mention that those conceptual models are not used for code generation, but for the discovery and selecting of implementation-specific information about apps.

1.3 Thesis Structure

The structure of the master thesis is defined as follows:

In chapter 2 the relevant related work for this master thesis is mentioned. This related work includes fundamental knowledge about different types of UI representations and how the UI can be built through conceptual modelling. This chapter also discusses how web services can be discovered in the web, which knowledge and concepts can be reused for app discovery and includes fundamentals about meta-modelling, modelling methods and models, which are relevant for the investigation of the used modelling method.

In chapter 3 the proposed framework, procedure and architecture is illustrated. The investigation is based on the considerations of the previous chapters and it will therefore discuss how the used modelling method fits to the standards and frameworks described in the previous chapter. Another important part of this chapter is the conceptual definition of the web service interface, web logic and app repository.

Chapter 4 gives a brief description about the implemented prototype for this work. In this chapter the implemented architecture and the design decisions for it are described. Each layer of the architecture is explained in detail and the prototype will be described using pseudo-code.

In chapter 5 the proposed procedure and operation of the prototype is explained. Chapter 6 gives an overview of the possible future research for this thesis, and chapter 7 will summarize the whole research and give some conclusions about the thesis.

2 Related Work

This chapter describes and discusses relevant research work and several topics. Because of the different research topics in this work, the upcoming subsections will be divided into:

1. User interface representation,
2. User interfaces modelling through conceptual models,
3. Web service publishing, and
4. Meta-modelling framework.

The topics about UI representation and modelling will be used as input for the analysis of the *ComVantage* modelling method in order to identify relevant concepts and attributes which can be used as input for the search algorithm. The topic web service publishing will be used as inspiration for the implementation of the prototype of this work. The last topic, dealing with the meta-modelling framework will be used as foundation for describing the *ComVantage* modelling method.

2.1 User Interface Representation

In this work the user interface (UI) of an app is also considered for the search mechanism. Therefore, several user interface representation techniques and formats are discussed. This chapter is divided into two parts. The first part aims to introduce several domain-specific languages (DSLs) which are currently used for describing and defining the UI of apps. Through this introduction of already existing DSLs for the UI of apps, I will try to identify how the UI of apps is structured and what important parts should be considered in the search algorithm for apps.

In the second part of this chapter, several abstract UI (AUI) definitions are presented. The aim of this part is to conduct on those abstract UIs in order to identify and define how the UI can be represented through a higher level language. This part is correlated with the on-going investigation and work on the *ComVantage* modelling method for apps, described in chapter 3.2. This modelling method uses similar concepts of abstract UIs and

therefore the mentioned standards in this chapter serve for better understanding of the modelling method.

2.1.1 Mobile Device Specific Formats

As already mentioned, a huge number of different user interface DSLs exist, depending on the mobile operation system (OS) used. In this chapter three most popular user interface DSLs will be presented in order to identify and show the commonalities and differences in those languages, which are the *Android UI*, the *iOS UI* and the *XAML*.

2.1.1.1 Android UI [7]

In [2] the *Android UI* framework is described as a modern and asynchronous framework which is similar to other desktop-based frameworks. It is considered to be a fourth-generation UI framework, like the *JavaFX*, *Microsoft Silverlight* and *Mozilla User Interface Language (XUL)*. The term fourth generation UI framework is related to a DSL which enables the definition of the UI in a declarative and independent way.

The *Android UI* is defined as a set of XML files in which each screen of the app is referred as an *Activity*. Each of those *Activities* can include multiple *Views*. In the Android specification *Views* are described as objects which can draw something interactive on the screen for the user. All *View* objects can be bundled together into a *ViewGroup*, which enables the user to specify a layout for his interface. The concepts of *View* and *ViewGroup* are defined in a hierarchy similar to a tree, as can be seen in Figure 1.

[7] http://developer.android.com/guide/topics/ui/index.html. Accessed: 01.12.2012

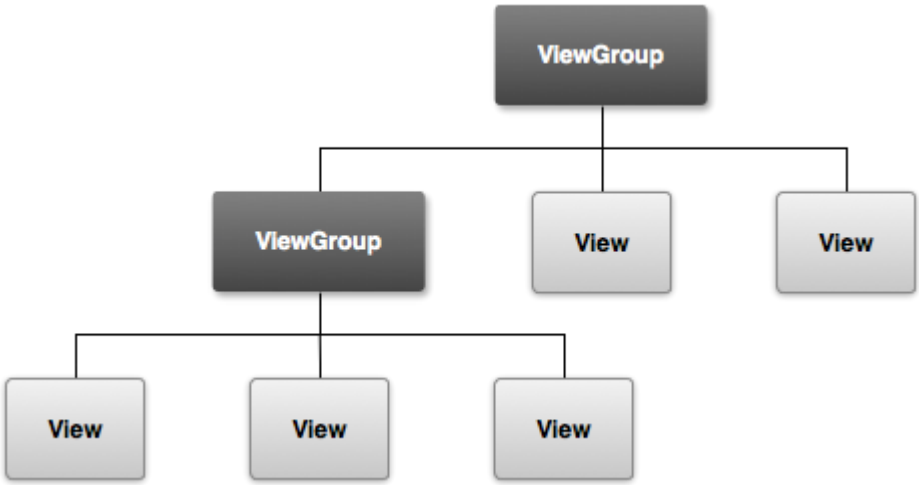


Figure 1 Tree hierarchy of *Views* and *ViewGroups*.[7]

Each *View* object defines different types of input controls (buttons, radio buttons etc.), output types (graphic, text etc.) or some widgets. An example of the *Android UI* with a vertical layout, a text view and a button is demonstrated in Figure 2.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="I am a Button" />
</LinearLayout>
```

Figure 2 Example of an Android UI.[7]

2.1.1.2 iOS UI [8]

App UIs which are developed in the *iOS* platform are usually described in an interface builder, which is implemented inside the *Xcode*[9] integrated development environment

[8] http://developer.apple.com/library/ios/#documentation/userexperience/conceptual/mobilehig/Introduction/Introduction.html. Accessed 02.12.2012.

(IDE). UIs developed in *Xcode* are saved into so called *Nib*[10] files (recently *".xib"*) which are part of the *Cocoa*[11] framework. An *Nib* file defines visual elements of the app´s interface, including windows, controls and others. The *Nib* file allows also the user to specify elements which are non-visual components, like objects for window or view management. *Nib* files are defined in a XML-based language containing a root called *<archive>* with several child nodes which describe the elements of the app´s UI. The basic components of an *iOS* UI are *windows* and *views*, whereas a *window* defines a geometric space on a screen and a *view* can be correlated to a canvas. All other elements of the UI, like buttons, text boxes and so on, are put anchored to a *view* which is a part of a *window.* Figure 3 demonstrates an example of a XIB file containing a button with some formatting properties. [3]

```
<?xml version="1.0" encoding="UTF-8"?>
<archive type="com.apple.InterfaceBuilder3.CocoaTouch.XIB" version="7.10">
    <data>
        <object class="NSMutableArray" key="IBDocument.RootObjects" id="1000">
            <object class="IBUIView" id="774585933">
                <reference key="NSNextResponder"/>
                <int key="NSvFlags">274</int>
                <object class="NSMutableArray" key="NSSubviews">
                    <bool key="EncodedWithXMLCoder">YES</bool>
                    <object class="IBUIButton" id="837768469">
                        <reference key="NSNextResponder" ref="774585933"/>
                        <int key="NSvFlags">292</int>
                        <string key="NSFrame">{{20, 20}, {280, 125}}</string>
                        <reference key="NSSuperview" ref="774585933"/>
                        <reference key="NSNextKeyView" ref="1000163707"/>
                        <bool key="IBUIOpaque">NO</bool>
                        <string key="targetRuntimeIdentifier">IBCocoaTouchFramework</string>
                        <int key="IBUIContentHorizontalAlignment">0</int>
                        <int key="IBUIContentVerticalAlignment">0</int>
                        <object class="NSFont" key="IBUIFont" id="138782212">
                            <string key="NSName">Helvetica-Bold</string>
                            <double key="NSSize">64</double>
                            <int key="NSfFlags">16</int>
                        </object>
                        <int key="IBUIButtonType">1</int>
                        <string key="IBUINormalTitle">START</string>
                        <object class="NSColor" key="IBUIHighlightedTitleColor" id="787094828">
                            <int key="NSColorSpace">3</int>
                            <bytes key="NSWhite">MQA</bytes>
```

Figure 3 Example of a XIB file.

[9] https://developer.apple.com/technologies/tools/. Accessed 02.12.2012.

[10] https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/LoadingResources/CocoaNibs/CocoaNibs.html. Accessed:02.12.2012

[11] https://developer.apple.com/technologies/mac/cocoa.html. Accessed: 02.12.2012

2.1.1.3 XAML [12]

Apps running on Windows Phone are defined with the *Extensible Application Markup Language (XAML)*. *XAML* is a declarative markup language used to define visible UI elements like text, controls, shapes and other UI elements. *XAML* also separates the UI of an application from its run-time logic, so that the *XAML* cannot be referred to one of the common languages which are executed by a common language runtime (CLR). Instead the *XAML* types are mapped to the CLR types in order to instantiate a run time representation of the UI. Besides this separation between UI and application logic, *XAML* provides the user with additional interaction mechanisms between the two.

The structure of an *XAML* file is based on a visual tree, which means that an element can describe other elements underneath it. This visual tree can be demonstrated with an example which is shown in Figure 4, describing a *Grid* which contains a *Stack Panel* which also contains a sequence of three *TextBlocks.*

```
<Grid Background="Red" x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
   <StackPanel Margin="20" Background="Blue" MouseLeftButtonDown="commonMouseHandler">
      <TextBlock Name="firstTextBlock" FontSize="30" >First TextBlock</TextBlock>
      <TextBlock Name="secondTextBlock" FontSize="30" >Second TextBlock</TextBlock>
      <TextBlock Name="thirdTextBlock" FontSize="30" >Third TextBlock</TextBlock>
   </StackPanel>
</Grid>
```

Figure 4 Example of a *XAML* file.

2.1.2 Device Independent Formats

The aim of this chapter will be to identify important parts of already existing AUI languages and to reuse this information to understand and to work on the *ComVantage Mobile Mockup model*. For this purpose the *User Interface Markup Language (UIML)* and the *Abstract User Interface Markup Language (AUIML)* were researched. In the following chapter only the *UIML* will be described, because the *AUIML* were mainly developed for internal use in IBM and therefore most of the information is confidential.

[12] http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj207025%28v=vs.105%29.aspx. Accessed: 02.12.2012.

2.1.2.1 UIML[13]

The *UIML* language is a product of the Organization for the Advancement of Structured Information Standards (OASIS)[14]. *UIML* is defined as a declarative, XML-compliant meta-language for describing UIs. The main purpose of the UIML is to design UIs for a broad range of different devices and that the used language is suited for real-world apps. Altogether, the OASIS mentioned following motivations for the introduction of UIML:

- UIs should be implemented by anybody without knowledge about specific languages and application programming interfaces (APIs),
- Reduction of time necessary to develop a UI for a variety of different devices,
- Separating the UI from the programming logic,
- UI prototyping,
- Simplify internationalization and localization,
- Enable an efficient mechanism for downloading UIs over a network,
- Enable the support of new UI extensions.

This means that *UIML* provides a solution how to develop a UI with a single syntax and use defined UI in a variety of different vendor technologies, like *.NET*, *Java*, *HTML* etc. The benefit of *UIML* is not only in the multiplatform usage of it, because *UIML* is also useful for the development of multimodal, multilingual and dynamic UIs.

UIML, as already mentioned, is a XML-based language for describing UIs. Elements of the UI are divided into *parts*. For each part the developer is able to specify *content*, which is responsible for the communication with the user of the UI. Examples for *contents* would be text, image, video, sound, etc. *Parts* which describe *content* provide the user with information; meanwhile the *UIML* distinguishes also *parts* which take inputs from the user. Those *parts* use interface artefacts (e.g. clickable button, input field, etc.) to realize the communication to the user. An overview on how UIML is modelled is shown in Figure 5.

[13] https://www.oasis-open.org/committees/download.php/28457/uiml-4.0-cd01.pdf Accessed: 03.12.2012

[14] https://www.oasis-open.org/ Accessed: 23.06.2013

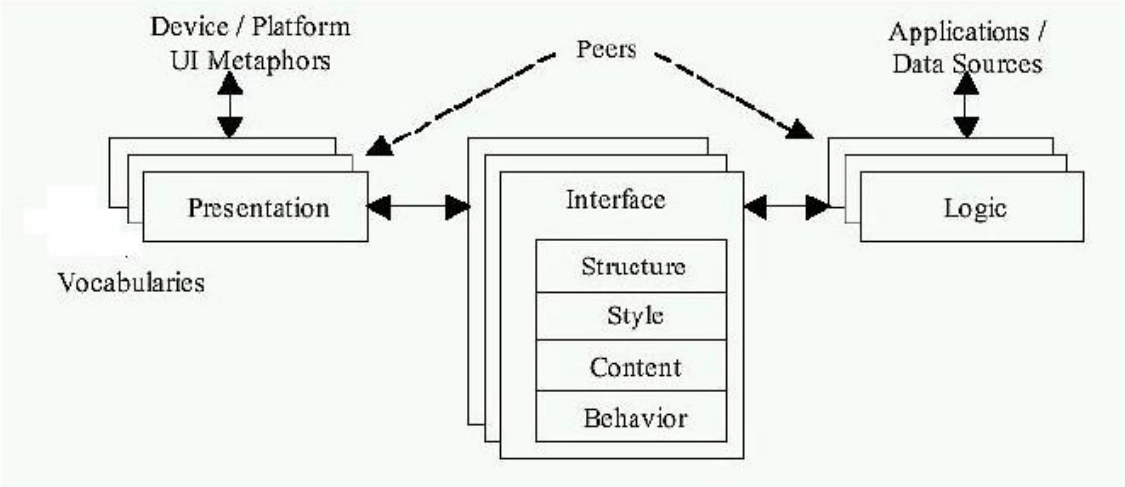


Figure 5 Separation of the UIML concepts. [13]

Figure 5 shows that the *UIML* is separated from the device specific presentation and the application logic. *UIML* specifies only the middle part of Figure 5, indicating the *Structure*, *Style*, *Content* and *Behaviour* of the UI. The *Structure* is built out of *parts*, which are defined in a device specific vocabulary. In the *Style* component, the developer specifies UI specific information, like e.g. size, colour etc. by assigning values to the *parts*. The *content* component as already mentioned, contains the information which should be visible in the UI. The *Behaviour* component specifies which actions should be executed under which conditions. Those conditions are described as platform-independent rules, when evaluating true, trigger some actions (e.g. application functions). The *Logic* describes the application logic which is behind the UI. Last but not least, the *Presentation* of the *UIML* defines how the *UIML parts* are mapped to device specific UI languages (e.g. *Java Swing*, *XAML*, etc.). [4]

2.2 UI Modelling through Conceptual Models

In this chapter several approaches are introduced that will show how to use models for representing and defining UIs. Through those approaches similar concepts which are used in the *ComVantage* modelling method can be understood and described. For this purpose following approaches were observed; the *USer Interface eXtensible Markup Language (UsiXML)* and the *eXtensible Interface Markup Language (XIML)*.

2.2.1 UsiXML[15]

The *UsiXML* is a product of the World Wide Web Consortium (W3C)[16], who describes the *UsiXML* as following:

"*UsiXML is a formal Domain-Specific Language (DSL) used in Human-Computer Interaction (HCI) and Software Engineering (SE) in order to describe any user interface of any interactive application independently of any implementation technology."*

Derived from this definition, the *UsiXML* represents a language which describes the UI of an application at different levels of abstraction. Similar to the *UIML*, the *UsiXML* description of the UI is independent of any devices or platforms on which the user is working on, interaction modalities (e.g., haptic, vocal), languages, user profiles and contexts defined by the user tasks. In [5], *UsiXML* is described through the following principles:

- *Expressiveness of UI*. The UI is defined through multiple models in dependency of the context. Those models can be edited, analysed and manipulated by specific software.
- *Models are centrally stored*. The mentioned models are stored in a central repository, expressed in the same user interface description language (UIDL).
- *Transformational approach*. Models which are stored in the central repository can be used for various transformations.
- *Multiple development paths*. The *UsiXML* supports the combining of different development steps into multiple development paths. Those development paths conform to different constraints, contexts of use and conventions of an organization.
- *Flexible development approaches*. Designers of applications are freely to choose a development path from a various set of development approaches like e.g. bottom-up, top-down, wide spreading and middle-out.

[15] UsiXML: http://www.usixml.org/en/home.html?IDC=221. Accessed 07.12.2012

[16] http://www.w3.org/. Accesses: 23.06.2013

The *UsiXML* structure is based on the Cameleon Reference Framework. The Cameleon Reference Framework expresses a UI by decomposing it into four abstraction layers. The abstraction layers are shown in Figure 6 and are as following:

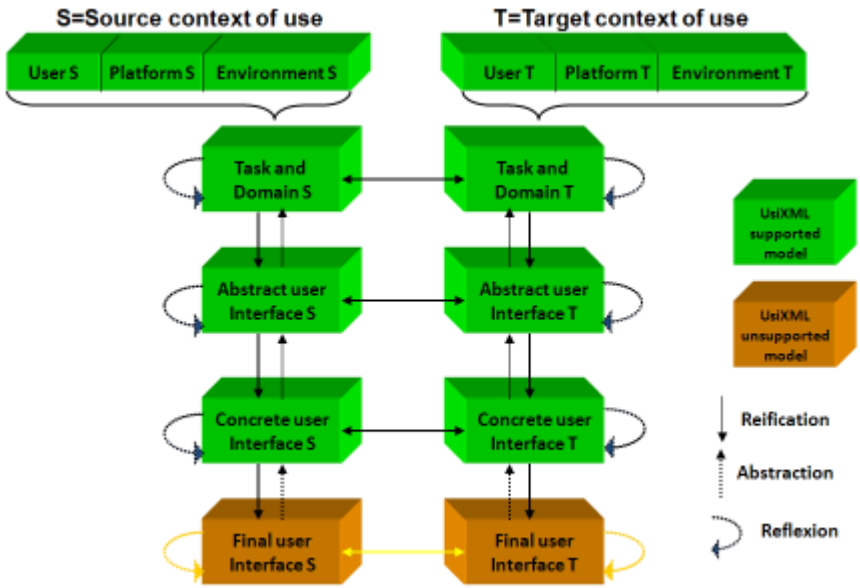


Figure 6 The Cameleon Reference Framework. [6]

1. *Tasks and Concepts (Domains).* In this abstraction level different tasks, which should be performed, are defined in order to satisfy the needs and goals of the users. This abstraction level also corresponds to the Computational-Independent Model (CIM) in Model-Driven Engineering (MDE). The *UsiXML* specifies three model types in this abstraction level [7] [8]:

    a. *Task Model.* Consists of the *Task* and *Task relationship* concept. A *Task* not only describes the activities which the user has to execute, but also information like the relative frequency of executing the *Task*, the relevance of the *Task* from the perspective of the user and the type of *Action* which is being performed. This additional information can be important when a UI has to be adapted to a new context and also to allow a better classification of *Tasks*. The *Task model* also allows the decomposition of *Tasks* through a *Decomposition* relationship and making temporal relationships between *Task* siblings through a *Temporal* relationship. Right now the *UsiXML* supports the modelling of the *Task model* with the *IdealXML* workflow

system. In Figure 7 an example of a *Task model* in *IdealXML* is demonstrated.

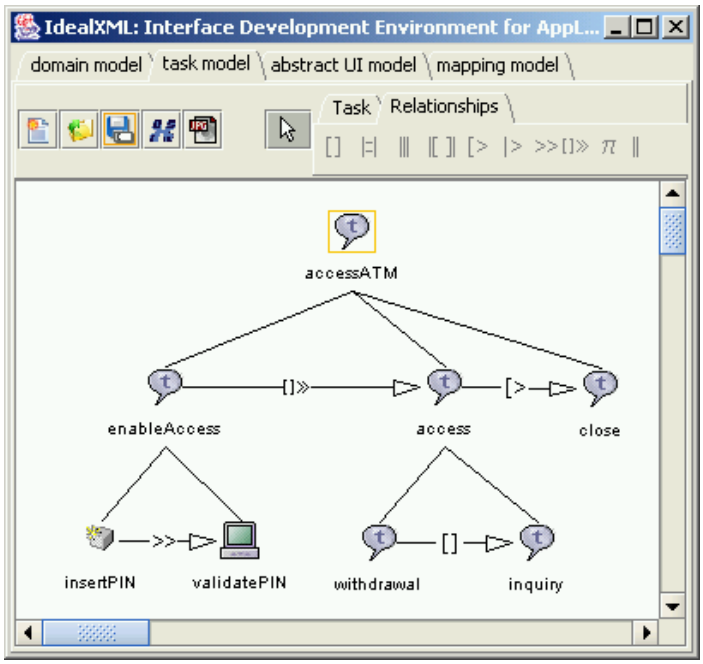


Figure 7 Example of a Task model in IdealXML. [8]

b. *Domain Model.* The *Domain model* describes information about real-world concepts which are independent of the visualization of the objects or the type of method invocation. The *Domain model* of the *UsiXML* is based on the UML *Class Diagram.* The concepts of the *Domain model* are manipulated at a certain degree by the users. By means of concepts the *UsiXML Domain model* distinguishes following concepts: *classes, methods, attributes* and *domain relationships.* The *domain relationships* include three types of relationships: *generalization, aggregation* and *ad-hoc.* The *Domain model* can also be modelled in the *IdealXML* workflow system as shown in Figure 8.

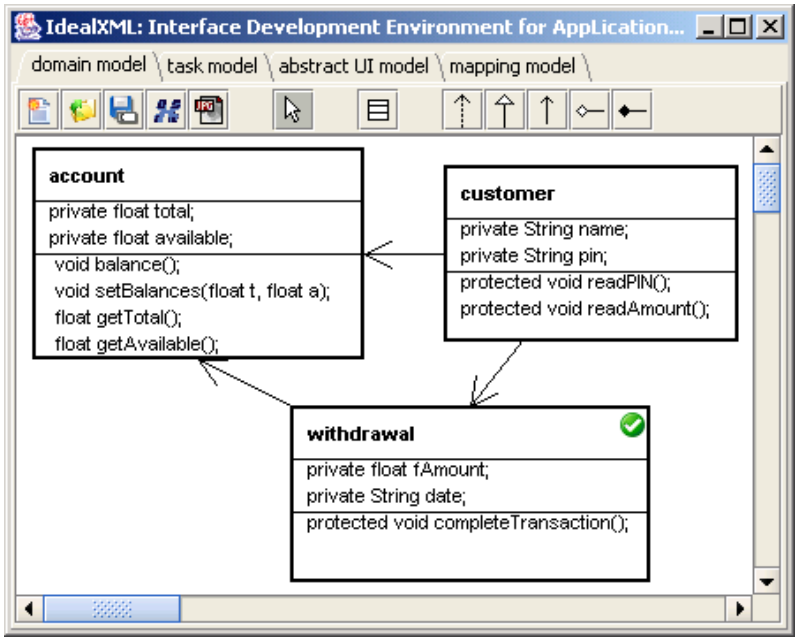


Figure 8 Example of a Domain model in IdealXML. [8]

2. *Abstract UI (AIU).* The *Abstract UI* abstracts the *Concrete UI (CIU)* so that it is independent of any interaction modality (e.g., graphical, haptical, etc.) or computing platform. The *Abstract UI* corresponds to the Platform-Independent Model (PIM) in MDE. It is expressed in terms of *Interaction spaces* (presentation units), a *Navigation scheme* which is defined between the *Interaction spaces, Abstract Interaction Objects (AIOs)* for each defined concept and *Abstract user interface relationships*. *Interaction spaces* provide the modeller with a possibility to group *AIOs* which are connected with a logically connected task. An *Interaction space* can therefore contain multiple *AIOs* or other *Interaction spaces.* In a *CUI* an *Interaction space* could refer to a window or dialog box, but it could also be mapped to an object without visible notation. The concept of *AIO* is platform and modality independent and structured in a hierarchy. *AIOs* can be composed into facets called *Multi-facets.* Following facets can be distinguished based on their function *Input, Output, Navigation* and *Control facet.* The *Input facet* defines an input type of an *AIO.* Through the *Output facet* it is possible to show data to the user. The *Navigation facet* allows the modeller to define a container transition which is triggered by an *AIO.* The *Control facet* defines a method of the functional core that can be enabled by a widget. An *AIO* has the possibility to define multiple facets at the same time. The *AUI* defines also relationships between the already

described concepts. The *AUI relationship* distinguishes between two main relationships, the *Dialog transition* and *Spatio-temporal* relationship. The *Dialog transition* allows the navigation between one *Interaction space* to one or several other *Interaction spaces.* The *Dialog transition* can result into four possibilities, *Suspend, Resume, Disables* and *Enables. Spatio-temporal* relationships describe physical constraints between *AIOs* in dependency of time and space. The *Spatio-temporal* relationship must be defined in a modality-independent way. The modelling of the *AUI* is also supported by the *IdealXML.* An example model of the *AUI* in *IdealXML* can be seen in Figure 9.

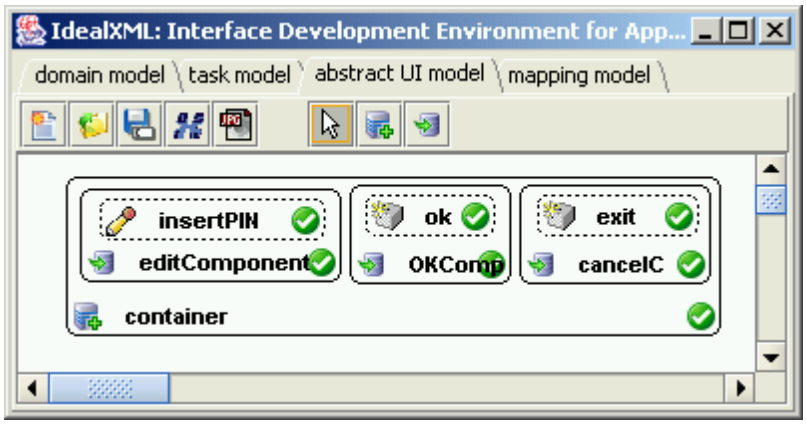


Figure 9 Example of an *AUI* modelled in *IdealXML*. [8]

3. *Concrete UI (CUI).* The CUI uses *Concrete Interaction Objects* to concretise an *AUI* for a certain context of use and interaction modality. The *CUI* would correspond to a Platform-Specific Model (PSM) in MDE. Through the *CUI* it is possible to define the layout of applications and interface navigation. However, the *CUI* still lies on an abstraction level beyond the *Final UI,* which means that it is independent of any computing platform and therefore only a mock-up of the *Final UI.* The modelling of the *CUI* is supported by many tools, like GrafiXML, ComposiXML, PlastiXML and VisiXML.

4. *Final UI (FUI).* The *FUI* represents the source code of the UI in MDE. This code can also be interpreted or either compiled. Examples of instances at this abstraction level would be the already mentioned representations in chapter 2.1.1.

As it can be derived the UsiXML is a model-driven approach (MDA), in designing and developing UIs. In order to understand the whole model-driven process in *UsiXML*, the

development process, each transformation and the necessary tools for *UsiXML* are presented in Figure 10.

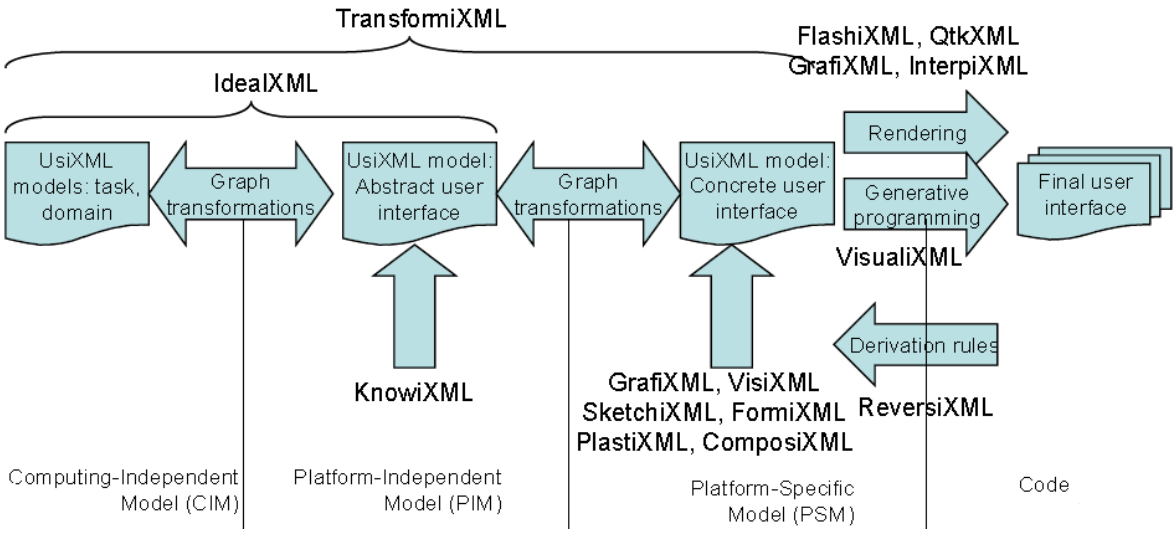


Figure 10 UsiXML development process.[17]

2.2.2 XIML[18]

XIML is a framework which allows the defining of interactive systems through models. *XIML* was developed by the research laboratory of RedWhale Software Corp and is now supported by the *XIML* forum. In the defining of UIs, *XIML* separates the UI into an abstract (contextual) which defines tasks, domains and users, and a concrete aspect (implementation-specific) which defines the presentation and dialogue. *XIML* also specifies a mapping from the abstract aspects to the concrete aspects.

The general structure of *XIML* is shown in Figure 11, describing the main representational units, which are interface *components* and *elements*, *relations* and *attributes*. In *XIML*, interface *elements* are hierarchically structured and separated into one or more interface *components*. The number and types of *components* are not limited in any way. In the first version of *XIML* (1.0), five basic interface *components* were introduced: task, domain, dialog and presentation.

[17] UsiXML: http://sfe-public.morfeo-project.org/wiki/index.php/UsiXML. Accessed: 09.12.2012

[18] XIML: http://www.ximl.org/. Accessed: 11.12.2012

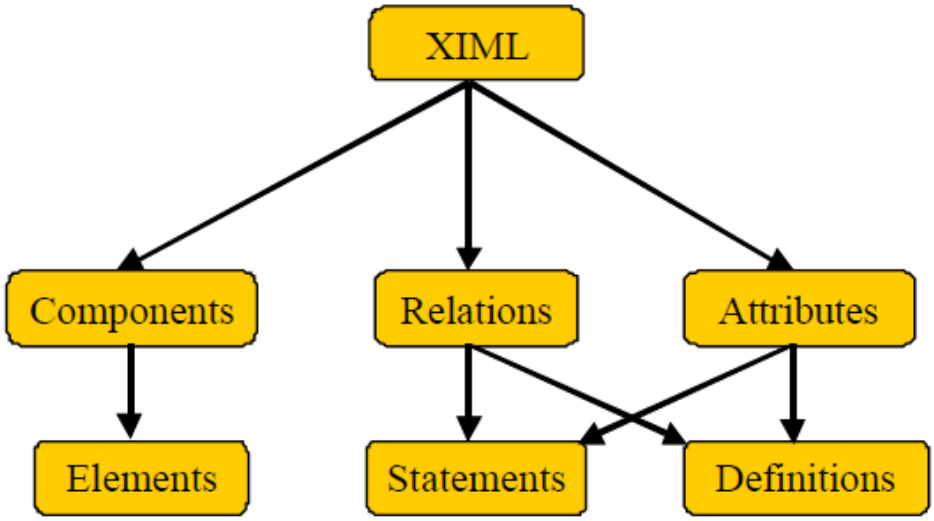


Figure 11The representational structure of XIML. [9]

1. *Task component*. The *Task component* is responsible for the business process and user tasks part of the supported interface. The *Task component* is hierarchically structured into tasks and subtasks. A flow relation can be defined between tasks, indicating the execution direction of the tasks. This component is only responsible for the user interaction part of a business process and not for the logic of the application. *XIML* does not specify the granularity of the tasks, so that the modeller is free to define its tasks (e.g. “Enter password”, “See table”, etc.).

2. *Domain component.* The *Domain component* can be seen as a basic ontology of objects, which defines a hierarchy of classes and objects. The definition of objects is provided through attribute-value pairings. Examples of objects would be e.g. “License”, “Map”, “Vehicle”, etc.

3. *User component.* The *User component* aims to represent the audience which will be using the UI. Therefore, *XIML* specifies in this component the concepts of users and user groups. Instance of those concepts would be for a user “Professor John Smith” and for a user group “Professor”. Instances can be further described through attribute-value pairs. Last but not least, *XIML* does not use the *User component* to focus on the cognitive states of a user, but for information and features that are relevant for design, operation and evaluation.

4. *Dialog component.* The *Dialog component* is responsible for the definition of a collection of interaction actions that are provided to the user of a UI. Besides, those definitions for interaction actions, the *Dialog component* also specifies the flow of those interaction actions that are possible in the UI. As it can be realized, the *Dialog component* represents a similar concept as the already described *Task component,* but additionally it works with the concepts of the concrete level rather than concepts of the abstract one like the *Task component.* Examples of instances in a *Dialog component* would be "Gesture", "Drag", etc.

5. *Presentation component.* The *Presentation component* provides the modeller with the possibility to specify a hierarchy of interaction elements which represent concrete objects that are used to capture the interaction of the user. The granularity of the interaction elements should be very high at this level, so that the application logic can be separated from the definition of the interaction elements. Examples of instances at this level would be buttons, text boxes, sliders, etc.

In *XIML*, *relation* representation units define statements that connect two or more *XIML elements* inside of one or several interface *components.* Through *relations*, *XIML* allows the creation of a knowledge body that enables the design, operation, and evaluation of UIs. Through a set of *relations* the *XIML* framework captures the design knowledge and through the runtime manipulation of those *relations* defines the operation of the UI. Besides, *XIML* does not enforce the semantic specification of those relations, so that the semantic is defined at the specific application which is developed. Last but not least, the *XIML* framework also provides a mechanism for the specification of *attributes,* which can be assigned to various values, defined with several data types and restricted with different ranges. For each *element* it is possible to define a set of attribute-value pairs. [9]

2.3 Web Service Discovery

The aim of this chapter will be to introduce techniques and mechanisms which are currently used and researched in the field of web service discovery. Besides, another focus will also be reasons for the evolvement of semantic descriptions and discovery of web services. Altogether, this related work will provide valuable information which can be used as inspiration for further thoughts and considerations about the discovery of apps.

In order to find, select and compose multiple web services it is necessary to consider three important issues. Those issues cover:

1. The description of web service interaction and integration of the provided functionalities.
2. The discovery of web services which provide the required operations specified in the plan.
3. The management of the interaction with those web services.

For the purpose of this work the first two issues are very important, which both together can be referred as the service discovery process. Service discovery was initially performed through the Universal Description, Discovery and Integration (UDDI) standard. UDDI enables the description and look up of web services through their functional aspects. In Figure 12 the publish-bind model with an UDDI registry is presented.

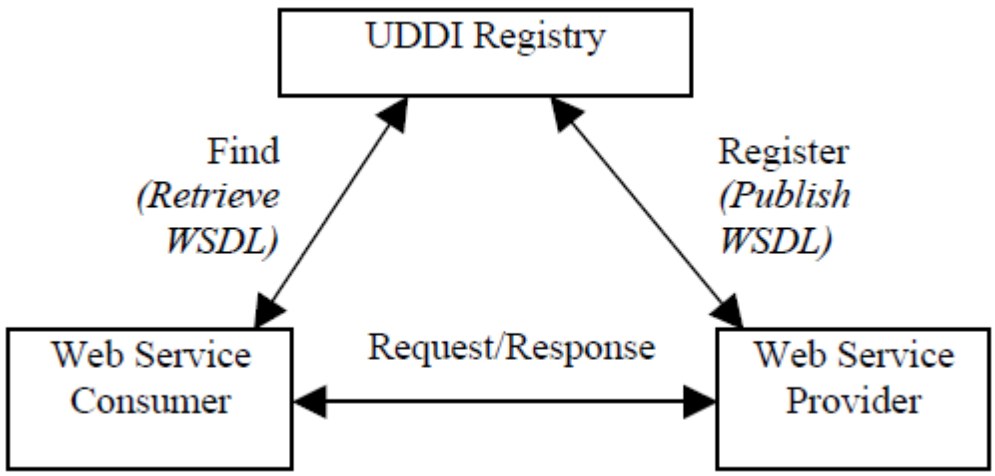


Figure 12 Publish-bind model. [10]

However, the focus of the UDDI standard lays in the syntactical description of the web service and therefore leads to two major limitations. The first limitation is the impossibility to represent and search for web services based on the operations they provide. Because of the functional description of the web service the consumer is not able to search about business relations or rules in the web service, neither to reason about the ordering or meaning of the exchanged messages. Those issues yield to the fact that through the UDDI standard it is not possible to automatically find or compose web services. The second

limitation of UDDI is that the used XML representation does not provide sufficient semantic description which is necessary for the discovery mechanism. [10] [11]

In order to deal with the shortcomings of the classification-based search mechanism of the UDDI standard, the UDDI standard was extended with a capability-based search mechanism. The capability-based search mechanism is based on the semantic description of the web service, which is achieved through ontologies like OWL-S or DAML-S. Through this search mechanism it is possible to search and identify a web service based on its required *inputs* (e.g. inputs for a book buying service would be the title and author), *preconditions* from the world which have to be true in order to execute the web service (e.g. a valid credit card for the book buying service), *outputs* which are produced by the web service (e.g. confirmation from the web service that the book was bought) and *effects* which represent actions in the world (e.g. the credit card is charged or the book changes property). [10] [12]

Last but not least, the Web Service Modelling Ontology (WSMO)[19] describes a web service through its own description model which is based on three core elements, including following:

1. *Capability*. Describes the functionality of the web service and also interfaces of the choreography and the orchestration.
2. *Choreography.* Defines how the web service describes its capabilities through interacting with the users.
3. *Orchestration.* Specifies how the web service achieves its capabilities by using the provided functionalities of other web services.

In Figure 13, all core elements of the WSMO web service description are presented, showing what each element describes. Besides, it is important to mention that the internal logic of the web service is not of relevance for the description and discovery of a web service. [13]

[19] http://www.w3.org/Submission/WSMO/. Accessed: 23.06.2013

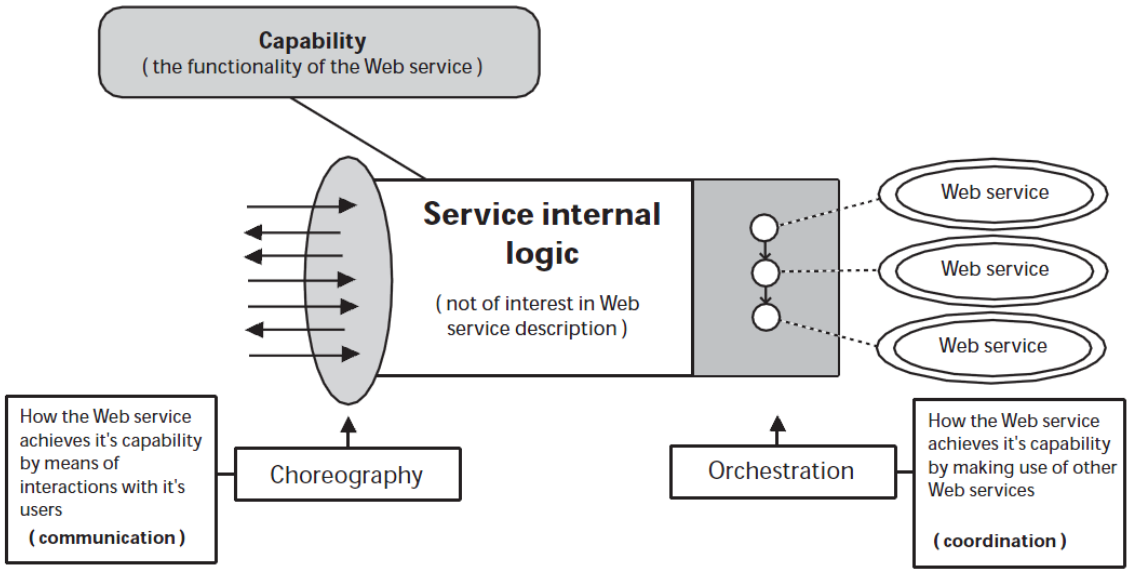


Figure 13 WSMO web service description. [13]

2.4 Meta-modelling Framework

One of the main goals of this work will be to research and investigate the *ComVantage* modelling method for apps [1], (public deliverable 8.2.1). This modelling method defines capabilities of apps through conceptual models, so that it should be possible to query and find an implementation for a specific requirement. Therefore, important foundations for this analysing of the *ComVantage* modelling method are the terms and concepts of modelling, modelling method and meta-modelling.

The fundamental term in modelling is the term of a model which is defined in [14] as an illustration, shortening and pragmatism of the original in which not all features of the original can be mapped to the model. The creation of a model is done by instantiating instances of a meta-model. The meta-model is created through a meta-modelling language and is also based on a meta-meta-model or the meta^2-model.

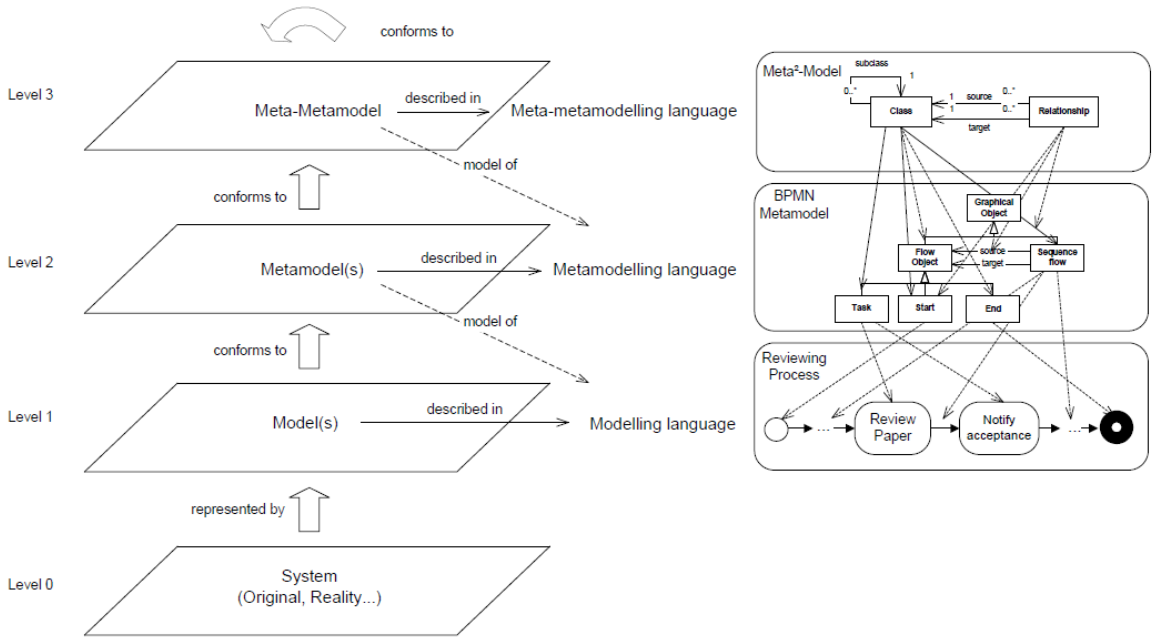


Figure 14 Modelling hierarchy. [15]

As it can be realized the model, meta-model and meta^2-model are linked in a hierarchy, called the modelling hierarchy which is depicted in Figure 14. The modelling languages are defined by a syntax, semantic and notation. The syntax of a modelling language is defined by a grammar and consists of elements and rules which are used for model creation. The semantic defines the meaning of a modelling language and the notation defines the visual appearance of the syntactical constructs. Besides, in order to properly apply a modelling language and instantiate instances, i.e. models, it is also important to define a modelling procedure.

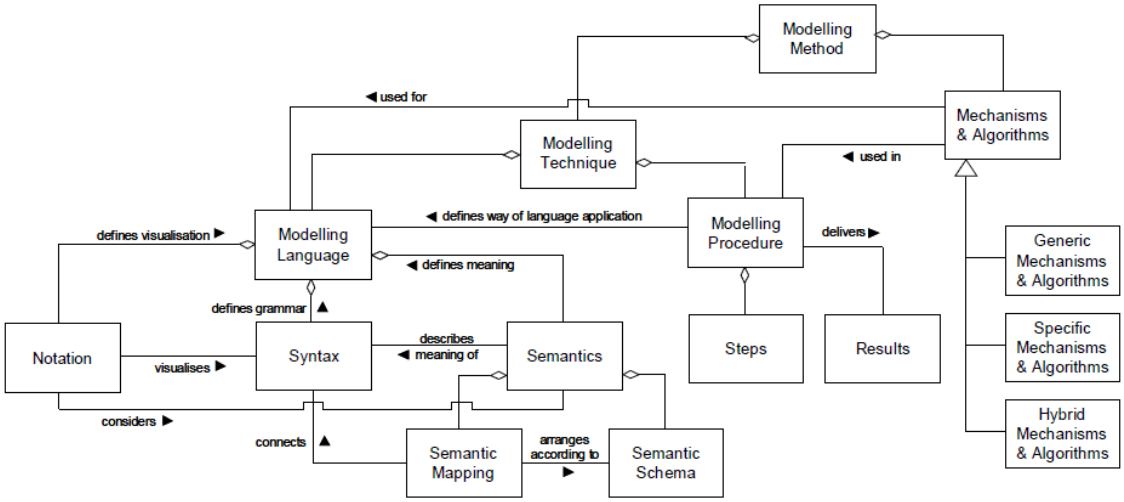


Figure 15 Components and relations of a modelling method. [15]

Both together the modelling language and modelling procedure define the modelling technique which together with mechanisms and algorithms define a modelling method. The components of a modelling method and their relations are visible in Figure 15. [15] [16] Based on those fundamentals, the *ComVantage* modelling method will be described in chapter 3.2 in order to identify relevant concepts which will be used for queries in the app search algorithm.

3 Framework, Procedure and Architecture

3.1 Overview

The aim of this chapter will be to introduce and formalize a solution for discovery of apps through conceptual models, which will fulfil the developer's requirements. Therefore, this chapter is divided into subchapters, describing each component which is necessary to establish the framework. Those subchapters include: (1) the description of a modelling method for apps, (2) a web application which will automate the process of retrieving apps, (3) a search algorithm for discovery of developer requirement-based apps and a central repository of all apps. Each subchapter will also contain a brief description of the fundamentals which are relevant for this subchapter.

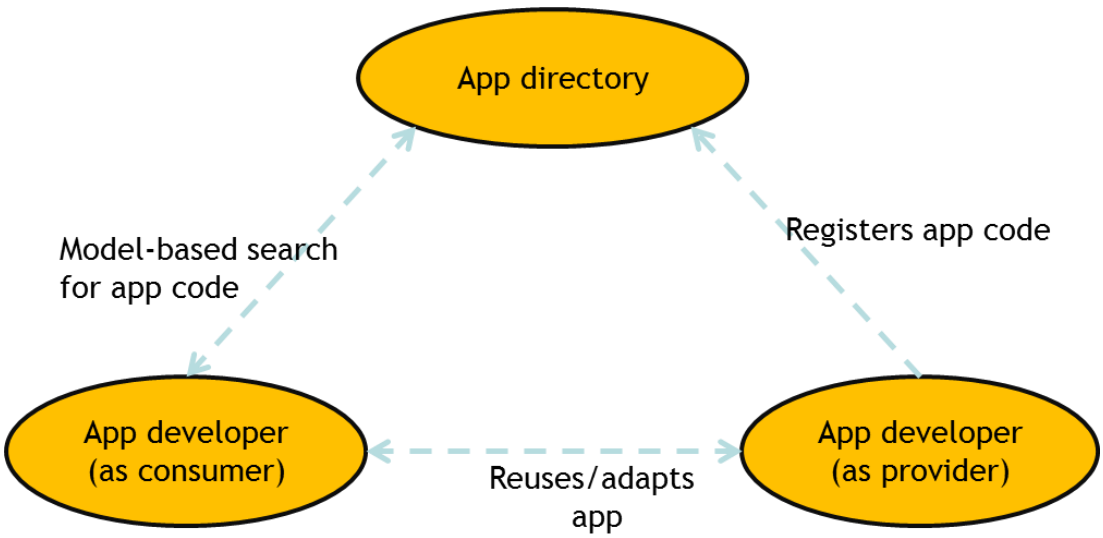


Figure 16 App discovery model.

The framework described in this work is inspired by the web service publish-bind model [8] described in Figure 12 and adapted for the purpose of app discovery. In Figure 16 the proposed framework is depicted, showing the process of defining, discovering and retrieving apps. In the first step an application developer, acting as a provider of apps, registers an app at an app repository. This app repository will be defined as a web service, which encapsulates extraction and search mechanisms for apps and accesses an app database. The aforementioned mechanisms will be used by another application developer, which acts as a consumer of apps, and based on a conceptual model-based search

algorithm returns as response an app as source code. The returned source code can be afterwards either directly reused by the developer or adapted to fit additional requirements.

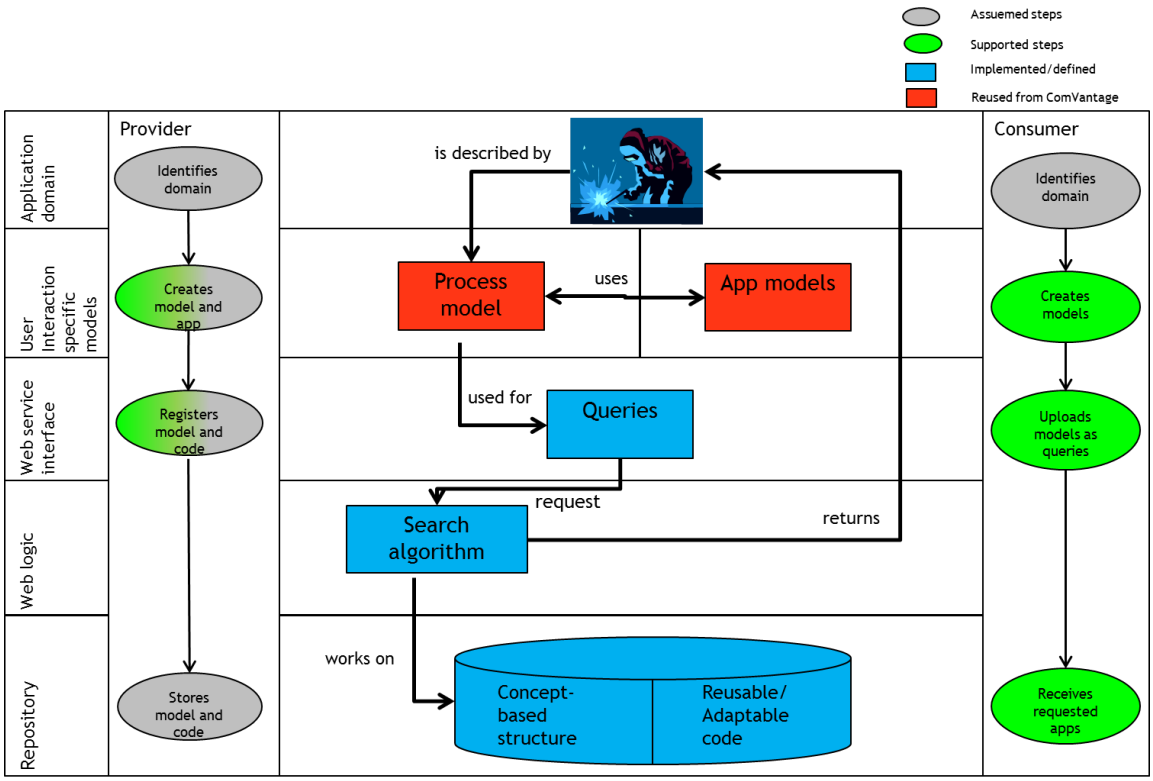


Figure 17 Proposed procedure and architecture.

In order to better illustrate and describe each of the mentioned steps, components and their relationships, in Figure 17 the proposed procedure and architecture is depicted. The architecture is built of five different layers:

1. *Application domain.* Defines the domain in which app support is required (e.g., maintenance, production, etc.). On this level each of the developers (provider and consumer) identifies the domain for which apps have to be developed.
2. *User Interaction-specific models.* This level of the architecture describes the relevant conceptual models which should be defined and discovered through apps. For the description of the domain in which the app is necessary, a *Process model* will be used and for the description of the required app additional *Mobile Mockup models* will be used. Those conceptual models will be further described in chapter 3.2. Each of the developers would on this level define the conceptual models, but with distinct purposes. The provider of an app would define each part of the developed app through conceptual models and additionally create the

implementation for the app. On the other hand, the consumer of the app would only create the conceptual models, based on his requirements for an app.

3. *Web service interface.* The web service interface defines how the developer has to interact with the web service. It is assumed that the provider of an app would send to this interface a message containing the conceptual models and implementation of the app in order to register the app in the repository. This step will only be partially supported in this master thesis by registering only the uploaded conceptual models. Meanwhile, the consumer will only send the defined conceptual models as a request to the interface, which will extract relevant information and create queries for the search algorithm. The consumer side of the web service interface will be further described in chapter 3.3.
4. *Web logic.* The web logic will encapsulate the search algorithm for app discovery. This search algorithm will use the before mentioned queries and try to discover possible apps in the repository. The search algorithm will only consider the concept-based structure of the registered apps and try to find possible matches for the defined queries. The web logic containing the search algorithm and comparison approaches for each conceptual model will be described in chapter 3.4.
5. *Repository.* The repository of apps is aimed to hold the registered apps of the providers. It will be separated into a concept-based structure part and an implementation part holding the reusable/adaptable code. This implementation part will be returned as response to the app consumer, if the search algorithm finds possible matches for the requested queries. The structure and description for the app repository can be found in chapter 3.5.

3.2 App Modelling Approach

Based on the fundamentals described in chapter 2.4, this chapter will introduce several conceptual models for describing apps. The conceptual models presented in this chapter were created considering the requirements from the researched literature about apps described in chapters 2.1.1, 2.1.2 and 2.2. Moreover, because of the similarity and huge experience in the field of web service discovery described in chapter 2.3, important considerations and knowledge were also gained from those concepts and ideas.

Inspired by the web service description model of the WSMO presented in Figure 13 following app areas and corresponding conceptual models were identified in the *ComVantage* modelling method and extended by additional ones:

1. *Mobile app choreography.* This area is used to describe how the app achieves its goals by interacting with the user. It is mainly covered by the *Interaction flow model* and partly by the *Mobile Mockup model* defined in the *ComVantage* modelling method.
2. *Mobile app orchestration.* The app orchestration defines how the app achieves its goals through internal interaction and interaction with other apps. For this purpose the *Mobile Mockup model* is used.
3. *Mobile app capabilities.* Through the app *Capabilities*, the core functionalities of the app are described. Besides, through the app *Capabilities* it also possible to specify how the app affects the real world through *Effects* and *Preconditions*. Those concepts are defined through a *Resource pool* which is not described in the *ComVantage* modelling method, but it is proposed as an extension of the modelling method.

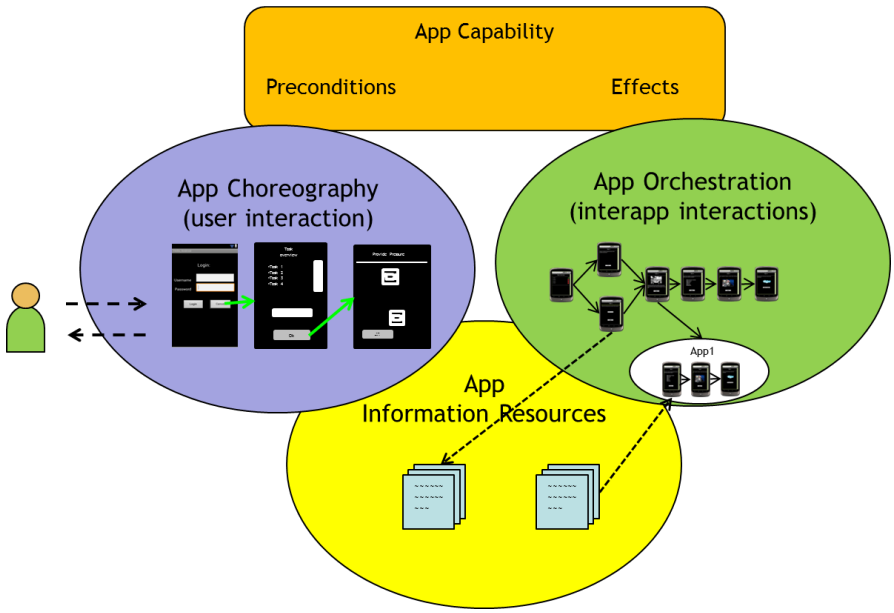


Figure 18 Areas covered by the proposed modelling method.

4. *Mobile app information resources.* The *Information space model* defines the input and output objects which are used or produced by the app. Those objects can range

from simple objects like a provided first name to complex objects like a created bill. The *Information space model* is not defined in the *ComVantage* modelling method and is proposed as an extension to the modelling method in this master thesis.

In Figure 18 the above mentioned areas and linking between them are displayed. In the following chapters each of the above mentioned models and used elements and relations between the elements are described.

3.2.1 ComVantage Model Types

3.2.1.1 Interaction Flow Model

The *Interaction flow model* will be used to capture the choreography of the mobile app with the user. Through the *Interaction flow model*, the expected user behaviour and requirements are described and will be used in order to define the *Mobile Mockup model*, which will be used for app discovery. The *Interaction flow model* was initially inspired by the *Task model* of the *UsiXML* described in chapter 2.2.1. Moreover, this model will also be extended and improved in comparison to the *UsiXML Task model*. The *Interaction flow model* will be modelled and defined as a *Business process model*, and will represent the interaction with the user in a more formal and executable way than it is done with the *UsiXML Task model*. Besides, the *UsiXML Task model* lacks also in the definition and representation of the user interaction flow, because the relations between tasks are not always straight forward to understand. The *Interaction flow model* will also enable the modelling of multiple paths, which the user can choose by interacting with the app. This model and its elements will not be used for the search algorithm and therefore will not be further explained. However, in Figure 19 an example and used concepts of the *Interaction flow model* are shown. In this figure it can also be seen that each *Activity* of the *Interaction flow model* is linked to a *Point of Interaction* of the *Mobile Mockup model* indicated by small icons of the *Point of Interaction* type in the *Activity* rectangle. Through this linking it is possible to indicate which *Point of Interaction* has to be executed next by the user. [1], (public deliverable 8.2.1).

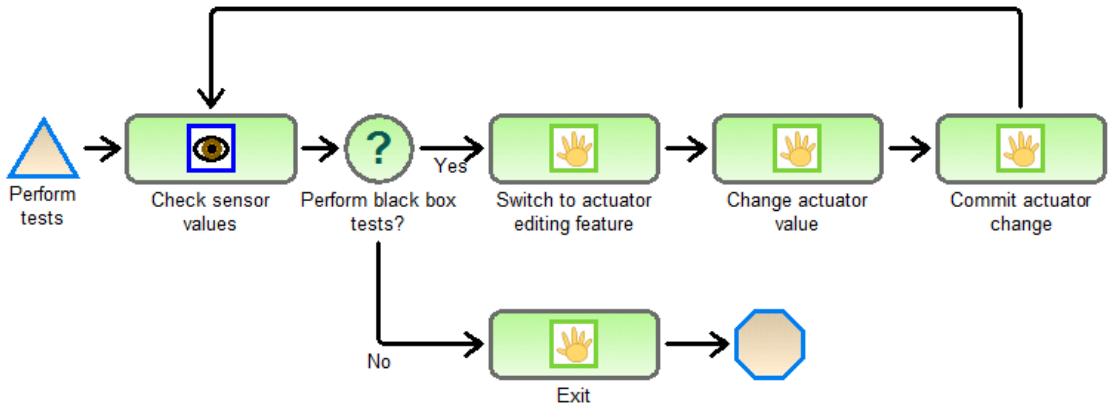


Figure 19 Example of an *Interaction flow model.*

3.2.1.2 Mobile Mockup model

The *Mobile Mockup model* is aimed to represent and describe the structure and orchestration of apps, showing how each screen of an app interacts with other screens or other apps. The term *abstract* in AUI is used in different contexts, but in this model it refers to a language which is independent of any implementation modality or platform and does not describe the presentation of the UI elements, instead describes the interaction styles of the user with the interface. However, the AUI elements can be mapped in the model through attributes to concrete UI elements in order to define the style of user interaction with the element. Besides, it provides the user with a possibility to identify how the information in the abstract elements is processed (e.g. local, server processed, etc.). The *Mobile Mockup model* is inspired by the concepts of *UsiXML* and *XIML AUI* described in chapter 2.2.

Because of the importance of the *Mobile Mockup model* for the sake of this work it is necessary to describe the used concepts in this model type. The used concepts in this model type are described in the following tables.

Model Attributes

Attribute name	Type	Description
Device type	Enumeration	The *device type* attribute allows the modeller to specify the device on which the mobile app is able to run. The values of this attribute are specified as enumeration list with following options: Nokia, Sony, Ericsson, LG, Samsung,

		iPhone and Blackberry.
OS	Enumeration	Through the *OS* attribute it is possible to specify the operation system on which the mobile app is running. The values of this attribute are specified as enumeration list with following values: Android, Windows, iOS, Symbian, HP WebOS and BlackberryOS.
OS Version	String	The *OS version* attribute allows the user to specify additionally the version of the intended OS.

Table 1 Model attributes of the *Mobile Mockup model*. [1], (public deliverable 8.2.1)

Readable POI

Interactive POI

Class: **Point of interaction**

The *Point of interaction* (*POI*) allows the developer of a mobile app to specify his requirements and expectations for a certain part of the developed app. In the *Mobile Mockup Model* two different types of *POIs* are distinguished:

1. *Readable* – specifies a requirement for an output which is shown on the screen to the user. This *POI* type is a non-interactive component of the UI. E.g. non-interactive charts or pictures, static text, tables, videos, etc.
2. *Interactive* – represents a *POI* which has the ability to interact with the user of the mobile app. This *POI* type does not change the appearance of the mobile app (e.g. zoom, rotate, etc.), instead it changes the data or the object which are related to this *POI* (e.g. drop-down selections, buttons, text-boxes, etc.).

The name of the *POI* should be shown underneath of the representation.

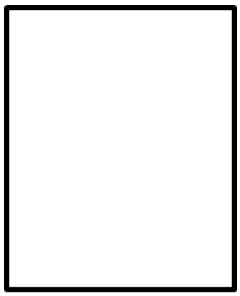
Attributes (of *Point of interaction*)

Attribute name	**Type**	**Description**
Name	String	Defines the name, which identifies this element.
Type	Enumeration	The *type* attribute of the *POI* specifies how the user is can interact with the object. Possible values are: readable and interactive. Only one selection of the values is possible for each object.

Description	String	The *description* attribute is used to specify an additional specification about the function or purpose of the object.
Abstract UI type	Enumeration	The *Abstract UI type* attribute describes further the interaction modality of this object depending on the previous selection of the *type* attribute. Based on the *type* attribute, this attribute allows the selection of following values: 1. *Readable* – single value (e.g. label), multi value (e.g. table), document (e.g. PDF) or media (e.g. picture). 2. *Interactive* – trigger (e.g. link, button), simple selection (e.g. drop down), multiple selection (e.g. checkbox), single value (e.g. text box) or grid input (e.g. table of text boxes).
Behaviour	Enumeration	The *Behaviour* attribute describes the interaction possibilities with other devices (e.g. server, other apps, etc.). Depending on the attributes *type* and *abstract UI type* one of the following values are possible: 1. *Readable* – from previous app, from server non-refreshable, from server refreshable pull, from server refreshable push, from local processing non-refreshable or from local processing refreshable. The "non-refreshable" behaviour describes a *POI* which is loaded once (e.g. the name of a product); while refreshable describes the possibility to refresh the value of the *POI*. Push or pull describes the updating type of values of *POI*, which can be that the server pushes the data to the client or the client pulls the data from the server. 2. *Interactive*: a. Simple or multiple selection – processed locally, passed to server or passed to the next app. b. Trigger – triggers remote procedure, triggers local procedure, triggers screen switching or triggers app switching. c. Single value, single or multiple

		row – sent to server, processed locally or passed to next app.

Table 2 Description and attributes of the *POI* class. [1], (public deliverable 8.2.1)

 <Name>	Class: **Screen** The *Screen* object defines a new screen of the app. Therefore, multiple *Screen* objects can be defined in one *Mobile Mockup model*. This object is defined as an aggregation, enabling to define multiple *POI* objects inside of it. The *Screen* object is also used to structure the visible surface and to arrange different *POIs* in different *Screen* fragments. In order to define the level of fragmentation the "Fragmentation level" attribute is used. The name of the *Screen* is displayed underneath the object.

Attributes (of *Screen*)

Attribute name	**Type**	**Description**
Name	String	Defines the name of the *Screen* used for identification of the object.
Description	String	The *description* attribute is used to specify an additional specification about the function or purpose of the object.
Height	Number	Defines the height of the *Screen* object.
Width	Number	Defines the width of the *Screen* object.
Fragmentation level	Enumeration	The *fragmentation level* attribute is used to define the level of detail of the *Screen* object. The level of detail includes also the device type. One of the following values is possible: a) phone, b) tablet, c) monitor or d) generic. The generic value can be used in case the device does not fit any other *fragmentation level* value.

Table 3 Description and attributes of the *Screen* class. [1], (public deliverable 8.2.1)

	Relation Class: **Triggers screen** The *Triggers screen* relation is used to specify the next *Screen*, which should be shown after the *interactive POI* is triggered. If the *Triggers screen* relation is defined between *Screens* of the same model then it should be indicated with a notation. Otherwise, if the *Triggers screen*

	is between *Screens* of different models then it may not be visible. **FROM**: *Point of interaction (type "Interactive")* **TO**: *Screen* **Cardinality:** Many to Many

Table 4 Description and attributes of *Triggers Screen* relation class. [1], (public deliverable 8.2.1)

In order to demonstrate the working of the *Mobile Mockup model* an example is shown in Figure 20. In this figure different *POI* types arranged in *Screen* objects are shown. The navigation possibilities are described through the *Triggers Screen* relation between *POIs* and *Screen* objects.

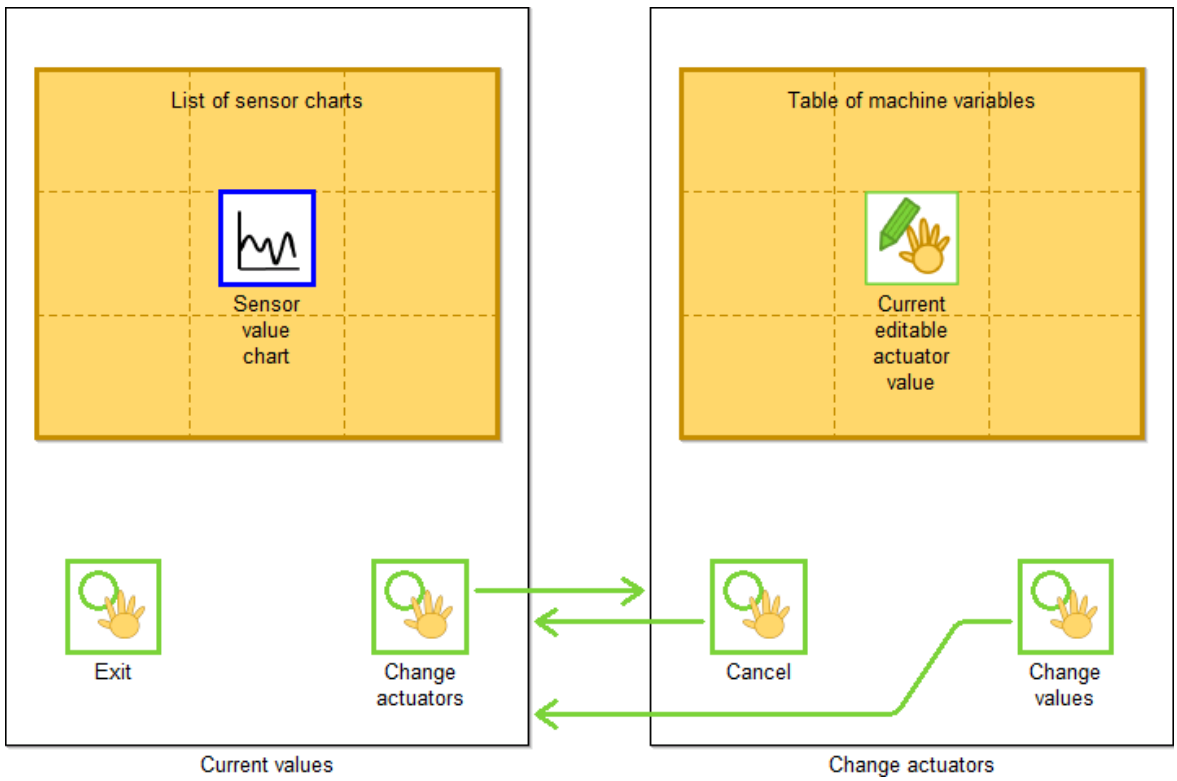


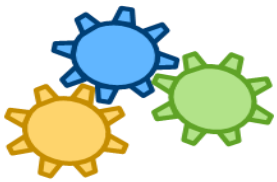
Figure 20 Example of a *Mobile Mockup model.*

3.2.2 ComVantage Model Type Extensions

3.2.2.1 Resource pool

The core functionality of the app will be also captured as a model, the *Resource pool*. Besides capturing the core functionality it is also relevant to capture how the app is

influencing the real world. This issue is also handled by the *Resource pool* through the elements of *Precondition* and *Effect*. Those elements are inspired by the web service search mechanism based on capabilities, described in chapter 2.3. Therefore, the meaning of the *Precondition* and *Effect* elements in the *Resource pool* is equal to those described in chapter 2.3. The mentioned classes will be explained in the following tables.

<table>
<tr><td> 
<name></td><td>Class: Capability
The Capability is used to express the desires about the capabilities of a certain Screen from the business perspective. Examples of business capability desires would be e.g. “The machine expert should be able to get a list of machine sensors” or “The machine expert should get a possibility to authenticate”.
The name of the Capability is displayed below the object.</td></tr>
<tr><td colspan="2">Attributes (of Capability)</td></tr>
</table>

Attribute name	**Type**	**Description**
Name	String	The name of the *Capability*.
Description	String	Contains an optional description about the *Capability*.

Table 5 Description and attributes of *Capability* class.

<table>
<tr><td> 
<name></td><td>Class: Precondition
The Precondition class describes which preconditions from the real world must be considered before a certain Capability can be realized (e.g. valid credit card for online buying, etc.).
The name of the Precondition is displayed below the object.</td></tr>
<tr><td colspan="2">Attributes (of Precondition)</td></tr>
</table>

Attribute name	**Type**	**Description**
Name	String	The name of the *Precondition.*
Description	String	Contains an optional description about the *Precondition*.

Table 6 Description and attributes of *Precondition* class.

 <name>	Class: **Effect** The *Effect* class defines the results in the real world of a certain *Capability* (e.g. money is booked from credit card, etc.). The name of the *Effect* is displayed below the object.

Attributes (of *Effect*)

Attribute name	**Type**	**Description**
Name	String	The name of the *Effect.*
Description	String	Contains an optional description about the *Effect*.

Table 7 Description and attributes of *Effect* class.

	Relation Class: **Belongs to** The *Belongs to* relation class specifies which *Preconditions* and *Effects* belong to a certain *Capability*. **FROM**: *Precondition; Effect* **TO**: *Capability* **Cardinality:** Many to Many

Table 8 Description and attributes of *Belongs to* relation class.

3.2.2.2 Information Space Model

The aim of the *Information space model* is to indicate which information is requested or produced by the app. Based on the granularity, two types of information can be distinguished, *Entities* which represent smaller information parts (e.g. first name, address, etc.) and *Information resources* which represent physical or digital information (e.g. PDF, bill, etc.). The mentioned elements will not for now be used for the execution of the search algorithm, and therefore will not be further explained. However, this model type can be used as a query extension of the search algorithm in further works.

3.3 Web Service Interface

The web service interface allows developers of apps to communicate with the web logic in order to receive required implementations for apps. This communication between web

service and developer is done by sending queries by the developer. The sent queries will be defined as conceptual models containing the search parameters for the search algorithm. As conceptual models, the aforementioned model types from *ComVantage*, described in chapter 3.2 will be used. In chapter 4.2 an example for such a query will be shown. For demonstration purposes of queries some textual examples of queries will be mentioned:

- *Capability-based* queries:
    - Find me an app which provides authentication and book buying capabilities.
    - Find me an app which checks as precondition if a bank account has sufficient money and as effect charges this account.
- *Information resource-based* queries:
    - Find me an app which produces a bill as output.
    - Find me an app which requires a username and password as input.
- *App structure-based* queries:
    - Provide me with an app which generates a chart from data provided by a server.
    - Provide me with an app which allows me to select an option from a list of given options and send it as request to a server.

3.4 Web Logic

In order to prove the concept of this work, a web application will be introduced and developed. This web application will provide some elementary functionality for automating the process of app discovery. In Figure 16 the app discovery model was presented. From this figure the position and relationship of the web application can be identified, whereas the web application is accessed by both developer; the provider and consumer of the app. Last but not least, the web application will carry and implement a search algorithm for app discovery. For all mentioned components different functions have to be implemented. Functions which need to be implemented are the following:

1. Data extraction from conceptual models.
2. Search algorithm for app discovery

3. Mobile application retrieval.

3.4.1 Data Extraction from Conceptual Models

As already mentioned, this work will define an app through conceptual models, which will be modelled in a modelling tool. In order to be able to search and query about implementations for apps it is necessary to extract data from the created conceptual models from the modelling tool. For this purpose the modelling tool will provide a possibility to export the aforementioned conceptual models in a format which will enable data extraction. For this purpose the (XML) data format will be used, which will provide a simple and fast way to extract data from the exported conceptual models.

3.4.2 Search Algorithm

One of the most important and interesting parts of this work will be the definition and implementation of the search algorithm. The search algorithm will consider and take as input the defined queries and try to find implementation-specific information from the app repository. As it can be realized from the previous chapter about the *ComVantage* modelling method, different conceptual models will be used. Because of this fact the search algorithm will require different comparison approaches in order to successfully identify the required implementations for apps based on the developer's queries. Those comparison approaches will be based on following:

1. *Capabilities* and *Information resources*,
2. App structure,
3. App orchestration.

It has to be mentioned that it is expected that the first two comparison approaches are less resource intensive and time consuming than the third one. Therefore, the search algorithm will first apply the first two comparison approaches in order to reduce the search space for the third comparison.

3.4.2.1 Comparison based on Capabilities and Information Resources

In chapter 3.2 the *ComVantage* modelling method was presented. In this modelling method it could be realized that the *Resource pool* and *Information space model* are defined as sets of objects referenced by other models. Because of this fact, a comparison approach for comparing sets can be used. Therefore, for this comparison approach the *Jaccard similarity coefficient* will be used [17]. The *Jaccard similarity coefficient* is a statistical measure used for comparing correspondence and diversity between two different sets. The *Jaccard similarity coefficient* is defined as the intersection of two different sets divided by the size of the union those two sets.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}, \mathrm{J(A,B)} \in [0,1]$$

Equation 1 *Jaccard similarity coefficient.*

The formula of the *Jaccard similarity coefficient* can be seen in Equation 1. The evaluated results of the *Jaccard similarity coefficient* range between zero and one, whereas a value of zero means that the two sets are completely different and a value of one that means that the sets are completely equal. An example will be shown in order to demonstrate how the *Jaccard similarity coefficient* will be applied to the search algorithm.

Example 1: Assumed is that in the repository exists two apps, with each three *Screens*. The constellation of assigned *Capabilities* to *Screens* is visible in Table 9.

	App A	App B
Screen 1	Show maintenance plan	Enable user authentication
Screen 2	Show history data about sensors	Show maintenance plan
Screen 3	Provide pressure data	Show cycle time of machine

Table 9 Example of assigned *Capabilities* to *Screens*.

Assuming that a consumer of apps is requesting an app R with *Capabilities* for "Show maintenance plan" and "Enable user authentication" the results of the search algorithm based on the aforementioned app examples would be following:

$$J(A,R) = \frac{|A \cap R|}{|A \cup R|} = \frac{1}{4} = \mathbf{0.25} \qquad J(B,R) = \frac{|B \cap R|}{|B \cup R|} = \frac{2}{3} = \mathbf{0.66}$$

Equation 2 *Jaccard coefficient* applied between apps A and R; B and R.

Equation 2 shows the results of the *Jaccard similarity coefficient* applied on the apps A and B from the example and the consumer request for a certain app R. The results of the evaluated *Jaccard similarity coefficient* shows that app B has a higher value (nearer to one) than app A and therefore it better satisfies the requirements of the consumer for an app R.

This comparison approach will not only be applied to *Capabilities*, but also to *Preconditions* and *Effects* of *Capabilities*. Last but not least, the *Jaccard similarity coefficient*, as already mentioned, will not only be used for the *Resource pool*, but also on the *Information space model*, precisely on the concepts of *Entity* and *Information resource*.

3.4.2.2 Comparison based on Mobile App Structure

This comparison approach is applied on the *Mobile Mockup model* described in chapter 3.2.1.2. The *Mobile Mockup model* is modelled by *Screen* objects containing a list of different *POIs*. Therefore, this comparison will compare each *Sreen* separately by applying on each pair the *Pearson's product-moment correlation coefficient (Pearson correlation)* [18] in order to measure the similarity between those two *Screens. Pearson's correlation* is today the most used statistical measure of relationship between two variables.

$$r_{ab} = \frac{n\sum a_i b_i - \sum a_i \sum b_i}{\sqrt{n\sum a_i{}^2 - (\sum a_i)^2}\sqrt{n\sum b_i{}^2 - (\sum b_i)^2}}, \; r_{ab} \in [-1, 1]$$

Equation 3 *Pearson's product-moment correlation coefficient.*

In Equation 3 the formula for the *Pearson's correlation* is shown whereas it is defined as the covariance of two variables divided by the product of the standard deviations of those two variables. It can also be noticed that the result of the *Pearson's correlation* takes values between minus one and one. *Pearson's correlation* value near minus one or one means that two variables have a high correlation while a *Pearson's correlation* value which equals to zero means that there is no correlation between two variables and no conclusions can be made about their similarity. The more the value of the *Pearson's*

correlation is nearer to minus one it means that two variables have less in common and a low similarity, otherwise the more the value of the *Pearson's correlation* is nearer to one it means that two variables have more in common and are more similar.

As already mentioned before, this comparison approach will be applied on each *Screen* of the *Mobile Mockup model*. In chapter 3.2.1.2 it could be seen that each of the *Screens* contains several *POIs* which have specified attributes. For the comparison, three attributes of the *POI* classes will be used: *Type*, *Abstract UI* and *Behaviour*. Through specifying those *POI* attributes different constellations of *POI* instances are possible. The search algorithm would consider the *POI* instances in each *Screen* and provide a similarity between the compared *Screens*. Following is an example which demonstrates how the *Pearson's correlation* will be applied on the comparison of two *Screens*.

Example 2: Assuming that in the app repository a *Screen* (S1) containing four different *POIs* exists, with the constellations of assigned attribute values as given in Table 10.

Name	**Type**	**Abstract UI type**	**Behaviour**
Description	Readable	Value	From server
Account number	Interactive	Single value	Sent to server
PIN	Interactive	Single value	Sent to server
Submit button	Interactive	Explicit trigger	Triggers remote procedure

Table 10 *POI* attribute constellations of *Screen* S1.

Besides, it is assumed that a consumer of apps who is requesting a *Screen* (R) with four *POI* attributes exists, which constellation of assigned values are described in Table 11.

Name	**Type**	**Abstract UI type**	**Behaviour**
Login button	Interactive	Explicit trigger	Triggers remote procedure
Username	Interactive	Single value	Sent to server
Password	Interactive	Single value	Sent to server
Cancel button	Interactive	Explicit trigger	Triggers local procedure

Table 11 *POI* attribute constellation of *Screen* R.

$$S1 = \begin{Bmatrix} 2 \\ 1 \\ 1 \\ 0 \end{Bmatrix} \qquad R = \begin{Bmatrix} 2 \\ 1 \\ 0 \\ 1 \end{Bmatrix}$$

Equation 4 *Screens* S1 and R represented as vectors.

The above mentioned *Screens* S1 and R can be represented as vectors which are shown in Equation 4, where it can be seen that both *Screens* are missing one *POI* constellation from the other *Screen*, have two instances of a certain *POI* constellation and also one instance of another *POI* constellation in common.

$$r = \frac{4 * \big((2 * 2) + (1 * 1) + (1 * 0) + (0 * 1)\big) - (2 + 1 + 1 + 0) * (2 + 1 + 0 + 1))}{\sqrt{4 * (2^2 + 1^2 + 1^2 + 0^2) - (2 + 1 + 1 + 0)^2} * \sqrt{4 * (2^2 + 1^2 + 0^2 + 1^2) - (2 + 1 + 0 + 1)^2}}$$
$$r = \frac{20 - 4 * 4}{\sqrt{24 - 16} * \sqrt{24 - 16}}$$
$$r = \frac{4}{4\sqrt{2}}$$
$$\boldsymbol{r = 0.70}$$

Equation 5 *Pearson's correlation coefficient* between S1 and R.

Based on those two vectors of *Screens* it is possible to calculate the *Pearson's correlation coefficient* as shown in Equation 5. According to the result value it can be concluded that between S1 and R a relative high correlation (r = 0.7) exists and therefore this *Screen* should possibly fit some of the requirements of the app consumer, who would extend or adapt the missing functionality.

3.4.2.3 Comparison based on Mobile App Orchestration

In the previous subchapter a comparison approach was presented which captures the comparison with only one requesting *Screen* as query. This approach is straight forward because the requested *Screen* is compared against all available *Screens* in the repository. But what if multiple *Screens* are requested by the app consumer? In this case the app consumer will be provided with alternatives in order to find the best fitting combination of *Screens*. Following alternatives will be provided:

1. *Less different mobile apps.* This alternative will try to identify as much as possible matching *Screens* in the same app. In order to measure when a requested *Screen* matches an available one, the app consumer will be provided with a threshold value. The algorithm will first find an app which has the highest number of

matched *Screens*. If not all requested *Screens* are matched in this app it would search in other apps finding the first match which fulfils the threshold value. If no match fulfilling the threshold value for a certain *Screen* can be found, then the highest match will be used. In order to reduce the computation time necessary for this algorithm, it will be defined as a greedy algorithm, which means that the best possible solution will be found quickly, but it does not mean that it is the best solution. Besides, this algorithm will provide the app consumer with the possibility to receive a solution which maybe not fulfils exactly the requirements for a single *Screen*, but it supports him with a solution in which maybe he would have less work in connecting each *Screen* with others or maybe he would reuse a whole app with all defined *Screens*.

2. *Less new connections between Screens.* This alternative can be seen as an extension to the previous one, which means that in this alternative an additional restriction will be introduced. This restriction defines that the algorithm will try to find a list of available *Screens* which are in the same app and simultaneously connected with each other. This will allow the app consumer that the effort of integrating the *Screens* together is less, because the connections between the *Screens* can be preserved. The search algorithm will first try to find the longest path of interconnected *Screens*. Afterwards, if not all requested *Screens* are contained in this path, the search algorithm will try to find the missing one first in the same app and afterwards in other available apps. Again, the threshold value will be used in order to indicate what the minimum similarity between requested and available *Screens* has to be.
3. *Most similar Screens.* By selecting this alternative, the app consumer would receive the best matches for each requested *Screen*. This alternative will be an exhausting algorithm, which will compare all available *Screens* against all requested *Screens*. After the algorithm is finished, the consumer would have to take care of the linkage between *Screens* (e.g. passing information between *Screens*, etc.). Besides, the app consumer could also provide a threshold value in this algorithm in order to reduce the necessary computation time.

The aforementioned different alternatives will help the app consumer to find a combination of *Screens* which will reduce the necessary effort in adapting single *Screens*

or connections between different *Screens*. Besides, it will also provide a way based on the requested *Screens* to possibly reuse a whole app.

3.4.3 Mobile Application Retrieval

The previous chapters described how the search algorithm would find available implementations for apps in the repository. The output of this search algorithm and what the app consumer can expect will be explained in this chapter. The proposed architecture in in Figure 17 showed that the second part of the repository represents an adaptable/reusable code. This part of the repository will be returned to the app consumer. Based on the specified conceptual models and selected search alternative the results can range from implementations for a single *Screen* to multiple apps.

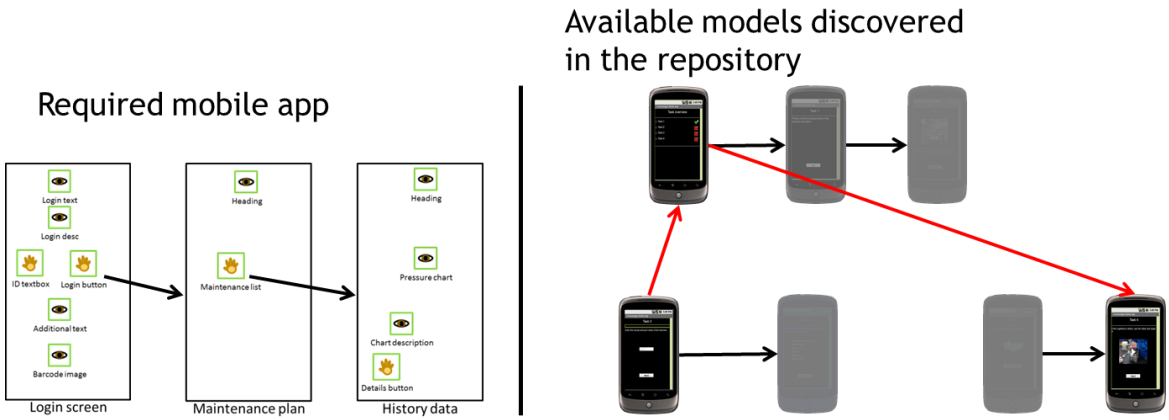


Figure 21 Returned implementations based on a certain app requirements.

An example of a returned result for a certain query is demonstrated in Figure 21. In Figure 21 shows that the query, defined as *Mobile Mockup model*, consists of three *Screens* containing different *POIs* connected in a certain order. The result of such a query could be spread on components of multiple mobile apps. In such a result the app consumer could reuse the single components, but would have to take care of integrating those components into a whole app (illustrated Figure 21 through connections coloured in red). If stored, the app consumer will be able to retrieve the platform-specific implementation of the logic or UI for each component.

3.5 Mobile Application Repository

In Figure 17 it was demonstrated that the app repository will be aimed to hold implementation and conceptual specific information about apps. The stored implementation specific information about apps will not be restricted to a certain platform or implementation type (e.g., Android OS, iOS, etc). The conceptual part of the repository will be constituted of information stored in the conceptual models from *ComVantage* described in chapter 3.2. In order to be able to store the conceptual models and corresponding implementations it is necessary to establish a well-defined structure for the repository.

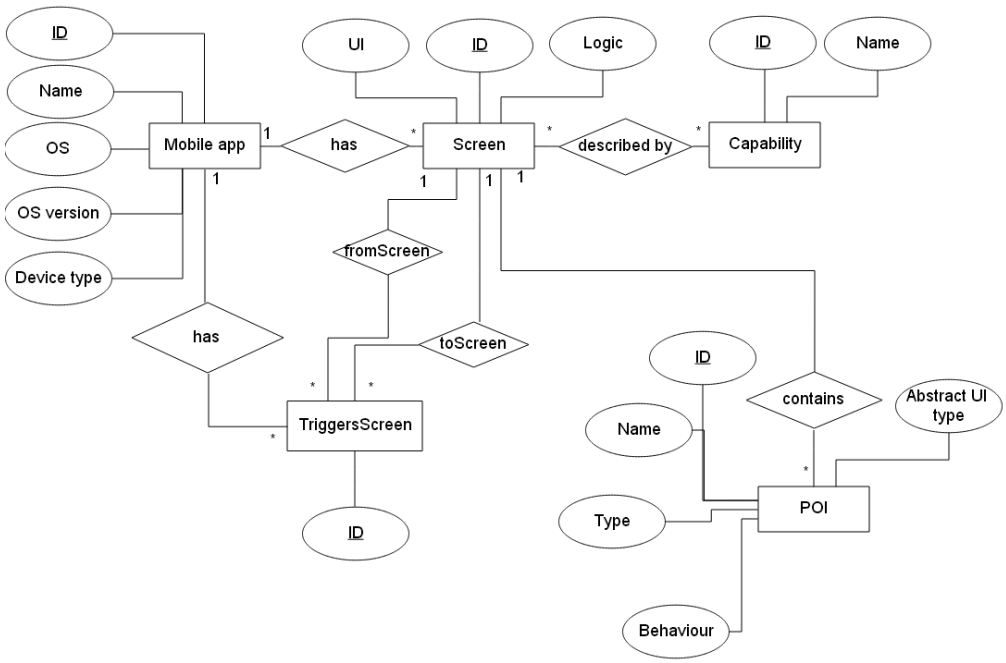


Figure 22 Entity-relationship diagram for the structure of the app repository.

For this purposes an Entity-relationship (ER) diagram was created based on the concepts and the attributes of the concepts from the conceptual models of the *ComVantage* project, which is depicted in Figure 22. Figure 22 also shows that one conceptual model from the *ComVantage* project and additionally one invented in this thesis are included, whereas the mapping between entities from the ER diagram to conceptual models is defined as follow:

1. *Mobile app*, *Screen*, *TriggersScreen* and *POI* → *Mobile Mockup model*
2. *Capability* → *Resource pool*

4 Implementation

This chapter describes how the aforementioned framework and architecture are implemented in a concrete prototype and which technologies and software architecture patterns are used in the implementation of the prototype. The architecture of this work was implemented by using the layers pattern, in which the application is structured by decomposing groups of subtasks and in which each group of subtasks is located at a single abstraction level [19]. This pattern allows that the changes in the source code should not affect the whole system; instead they only affect a certain component on which the changes are applied. The application was separated into three different abstraction levels, as seen in Figure 23. This figure shows that the used technologies in this architecture are open and free to use. In the following subchapters each of those layers and the used technologies will be further described.

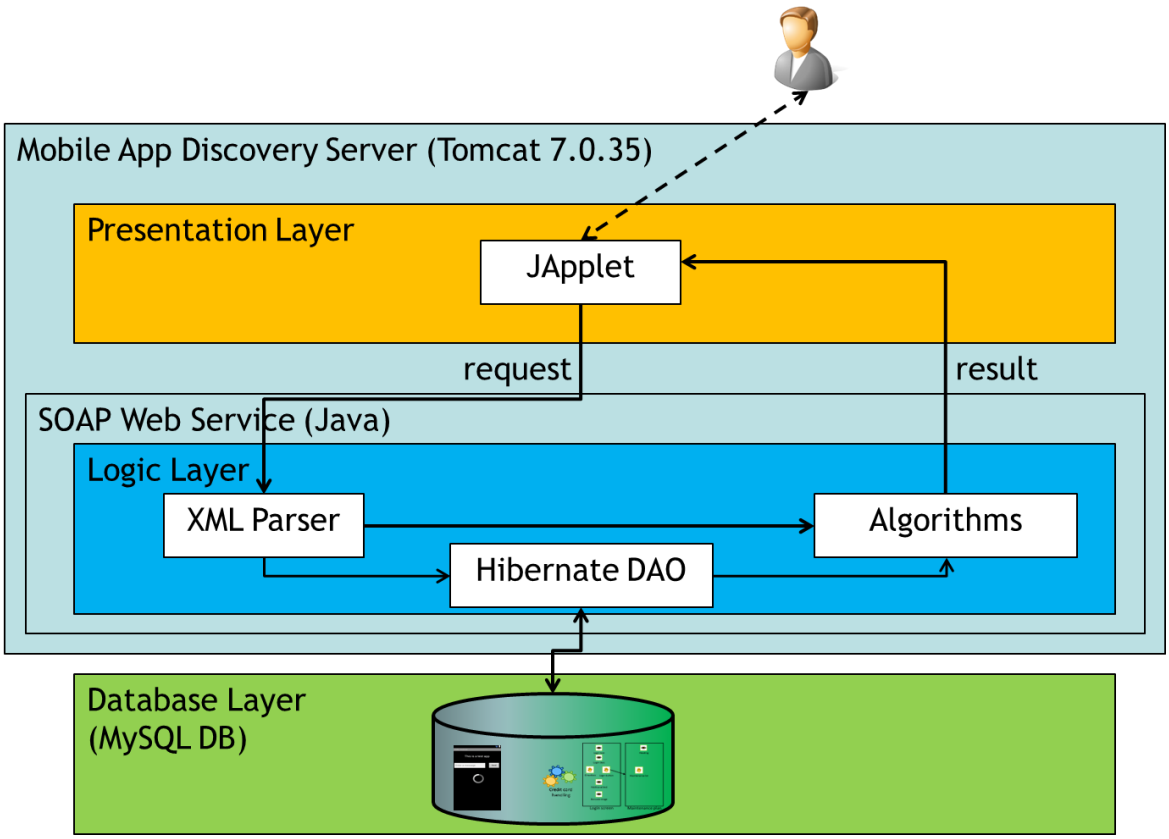


Figure 23 Architecture of the implemented prototype.

4.1 Presentation Layer

The presentation layer is aimed to capture the interaction of the user with the web application. As already mentioned one of the requirements for this work was that the application runs over the internet and because of this requirement the user interaction will be captured over the browser by using *Java Applets (JApplet)*. Technologies used to implement the user interface of the *JApplet* are the following:

1. *JGraphT*[20]. *JGraphT* is a free Java graph library which implements different representation types and algorithms for graphs. In this work *JGraphT* is used to represent the structure and sequence of *Screens* in the app. Besides, it is also used to indicate which *Screens* should be reused and which *Screens* of the app are irrelevant for the developer who is searching for an app. Through the "*Triggers Screen*" relations it is also possible to indicate which connections between *Screens* can be reused and which should be implemented by the developer.
2. *RSyntaxTextArea*[21]. *RSyntaxArea* is a Java library which is used in *Java Swing* applications. The main purpose of this library is to highlight text components in *Java Swing*. This library is used to implement the code retrieval for a certain *Screen* of an app as described in chapter 3.5. Through the *RSyntaxTextArea* library the issuer of the query will not only be able to select code moreover he will also retrieve the code which is already highlighted for a certain programming language (e.g. Java, XML, etc.).

Both of those libraries can be seen in Figure 24 where the *JGraphT* library is marked in the red dotted rectangle and the *RSyntaxTextArea* library in the blue dotted rectangle. Last but not least, it is important to mention that the UI does not implement any logic or algorithms which are needed for the discovery of apps, it only handles the selection of requests (specified as XML files) from the file system sending those request over the internet to the discovery web service described in chapter 4.2 and shows the results of the applied algorithm in the web service to the user. The UI provides the user with the option to select which algorithm he wants to apply on request. In this selection he can choose

[20] JGraphT. http://jgrapht.org/

[21] RSyntaxTextArea. http://fifesoft.com/rsyntaxtextarea/

between the different variations of the *Pearson's correlation* algorithms described in chapter 3.4.2.3 and by specifying *Capabilities* in the request he can also automatically execute the *Jaccard index algorithm*. The execution of the algorithms is done by passing the results of the *Jaccard index algorithm* to the *Pearson's correlation* algorithm.

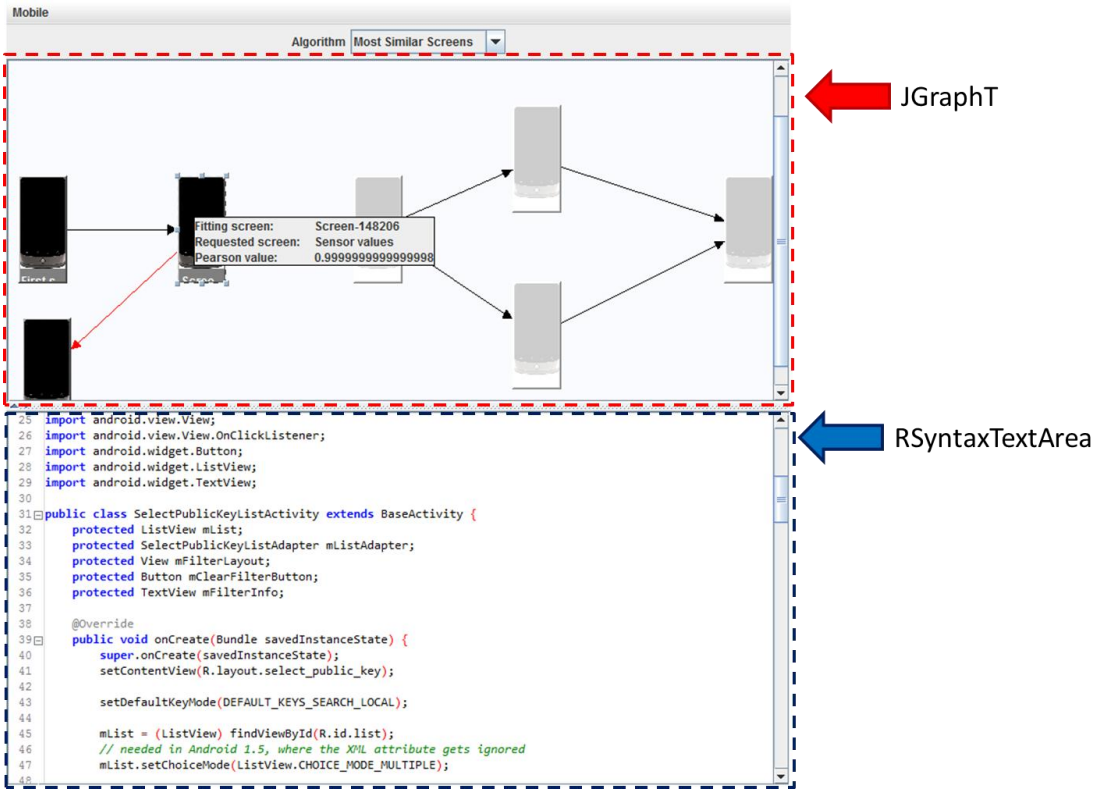


Figure 24 Screenshot of the prototype showing the used technologies in the UI.

4.2 Logic Layer

The logic layer is aimed to hold all computations which are necessary to take the inputs, requests (described as XML) and available models and code from the database, and produce the desired output as described in chapter 3.4.3. The logic layer is encapsulated into a SOAP (Simple Object Access Protocol) web service implemented in Java running on a Tomcat 7.0.35 server. The different functions of the web service are implemented into several components in order to increase the cohesion and reduce the coupling between those functions. Those components include the following: XML parser, Hibernate Data Access Object (DAO) and different types of algorithms. In the following subchapter each of those components will be further described.

4.2.1 XML Parser

The XML parser component is responsible for information extraction from the requests specified in XML. For this purpose the *JDom2* java library is used and each node of the request XML tree is accessed by using *XPath* expressions. The XML parser for now extracts only information about two model types, the *Resource pool* and the *Mobile Mockup model*. In both models the XML parser extracts information about the position of the single concepts in order to determine the positions of the *Screens* in the returned result. The XML parser extracts besides the position of concepts also different attributes in the concepts which are the following:

1. ***Mobile Mockup model.*** Attributes: Name, Device type, OS and OS version.
    a. ***POI.*** Attributes: Name, Class, Behaviour, Abstract UI type, Behaviour trigger and Trigger.
    b. ***Screen.*** Attributes: Name and Has capability.
    c. ***Triggers screen.*** Attributes: From POI and To screen.
    d. ***Is inside.*** Attributes: From screen and To POI.
2. ***Resource pool.*** Attributes: Name.
    a. ***Capability.*** Attributes: Name and Description.

```
<INSTANCE id="obj.155406" class="Readable" name="Login text">
<ATTRIBUTE name="Position" type="STRING">NODE x:3cm y:.5cm w:1.5cm h:1.5cm index:2</ATTRIBUTE>
<ATTRIBUTE name="Selection" type="EXPRESSION">EXPR val:error:"[acoexhan-01]\nNo expression!"</ATTRIBUTE>
<ATTRIBUTE name="Behaviour" type="ENUMERATION">From local processing</ATTRIBUTE>
<ATTRIBUTE name="Abstract UI type" type="ENUMERATION">Value</ATTRIBUTE>
<ATTRIBUTE name="Description" type="STRING"></ATTRIBUTE>
</INSTANCE>
```

Figure 25 Example of a fragment of a query defined as XML.

In Figure 25 an example of a fragment of a query is demonstrated. In this figure it is shown how the instance of a *POI* is defined after it is exported from the modelling toolkit. This fragment also contains attributes which are consumed by the XML parser like e.g. position, behaviour, abstract UI type, description, etc.

After the information has been extracted, it is further passed to the Hibernate DAO and to the algorithms component, which apply further processing on the extracted information.

4.2.2 Hibernate Data Access Object (DAO)

The Hibernate DAO component is responsible for providing a single and shared access to the underlying database. Therefore, the Hibernate DAO component is designed by using the Shared Repository pattern [20]. The Shared Repository pattern is used to structure applications in which the collaboration is purely data-driven. The Shared Repository pattern is responsible for the shared data which is used by the components of the application. Those components are provided with mechanisms for data access and modification in the shared repository. The data access and modification is realized by using a query API or a language. Through this pattern it is possible to ensure data consistency and it also handles problems of resource contention (e.g. by locking accessed data). In Figure 23 it can be seen that the components of the logic layer (XML parser and Algorithms) uses the common Hibernate (Hibernate version 3) data access to communicate with the underlying database.

4.2.3 Algorithms

The main part of this work is the description and implementation of the invented algorithms which are used for the discovery of apps. In chapter 3.4.2 the implemented algorithms of this work were described. From this chapter four different algorithms are derived and successfully implemented in the prototype. The invented algorithms were implemented and structured by using the Strategy pattern [21]. The Strategy pattern defines a family of algorithms in which each of the implemented algorithms is encapsulated into an own class and the implementation of the algorithm is accessed via a common interface. Through this structuring each algorithm is interchangeable, what means that it is possible to change the implementation of the algorithms without affecting other components which use this algorithm. The Strategy pattern allows the user to select dynamically and at run-time a specific algorithm's behaviour. The dynamical selection possibility of the Strategy pattern enabled the possibility of selecting between the three *Pearson's correlation coefficient* algorithms. The fourth algorithm based on the *Jaccard index* comparison is selected only if the necessary *Resource pool* is specified in the XML file and loaded into the web application. This algorithm is used as an initial filtering of apps, which means that the results of this algorithm are passed to the input of the selected

Pearson's correlation coefficient algorithm. The implemented Strategy pattern and the four algorithms are depicted as a class diagram in Figure 26. Figure 26 shows that the algorithm interface, which holds all four algorithms, is dynamically instantiated by the *Search Mechanism* class. Depending on this instantiation it is possible to select one of the four algorithms implemented and apply it on the passed model query. Each of those algorithms will be described in the following subchapters. Last but not least, it is important to mention that the result of the algorithms is a specialized data structure which will be also explained in the following subchapter.

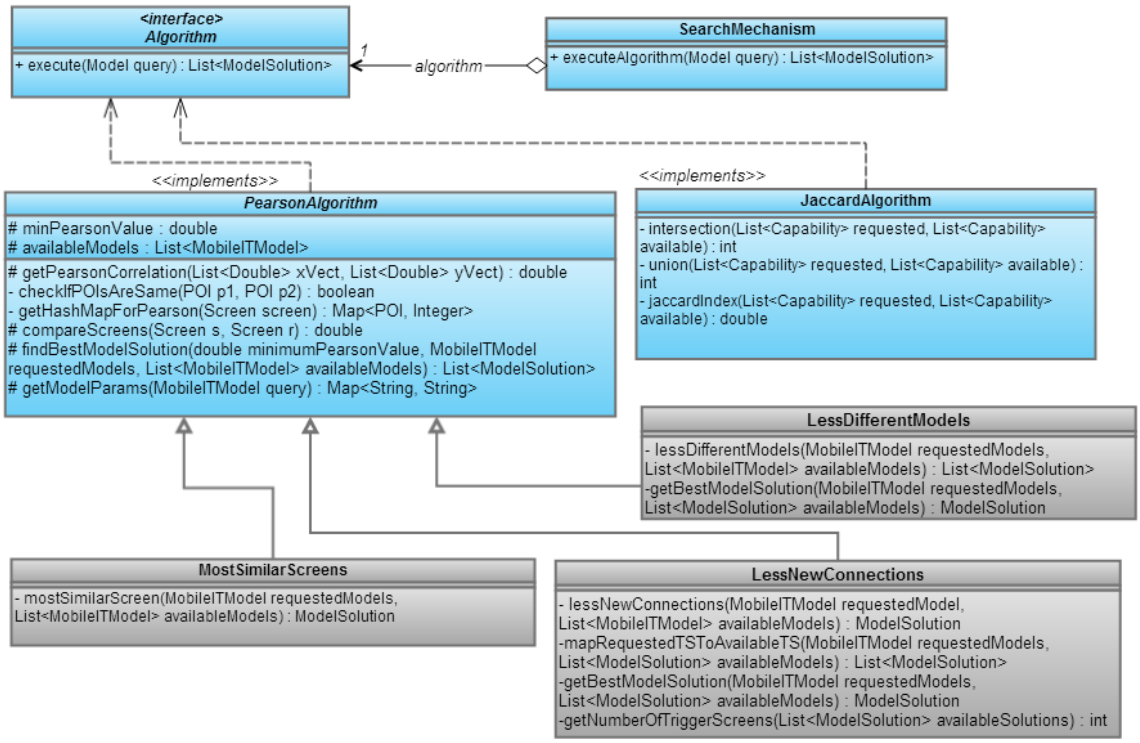


Figure 26 Class diagram of the implemented Strategy pattern.

4.2.3.1 Returned Model Solutions and Screen Solutions

In order to explain the operation of the different algorithms which were implemented, it is necessary to explain the structure of the returned results created by those algorithms. The returned result of the algorithms is defined by the *Model Solution* and (*Screen*) *Solution* data structures. The *Solution* data structure is responsible for the mapped *Screens*. Therefore, the available *Screen*, requested *Screen*, comparison value of those two *Screens* and the available model of the available *Screen* are captured in this data structure. The *Model Solution* data structure contains a list of *Solutions* objects, and two lists of *Triggers*

Screen objects. The list of *Solution* objects is used to define the mapped *Screens* for this *Model Solution*. The other two lists for *Triggers Screen* objects are used to capture *Triggers Screen* relations of the available and requested models. Both of those data structures are represented in Figure 27. Figure 27 also shows how the used classes of the *Model Solution* and *Solution* data structure are build.

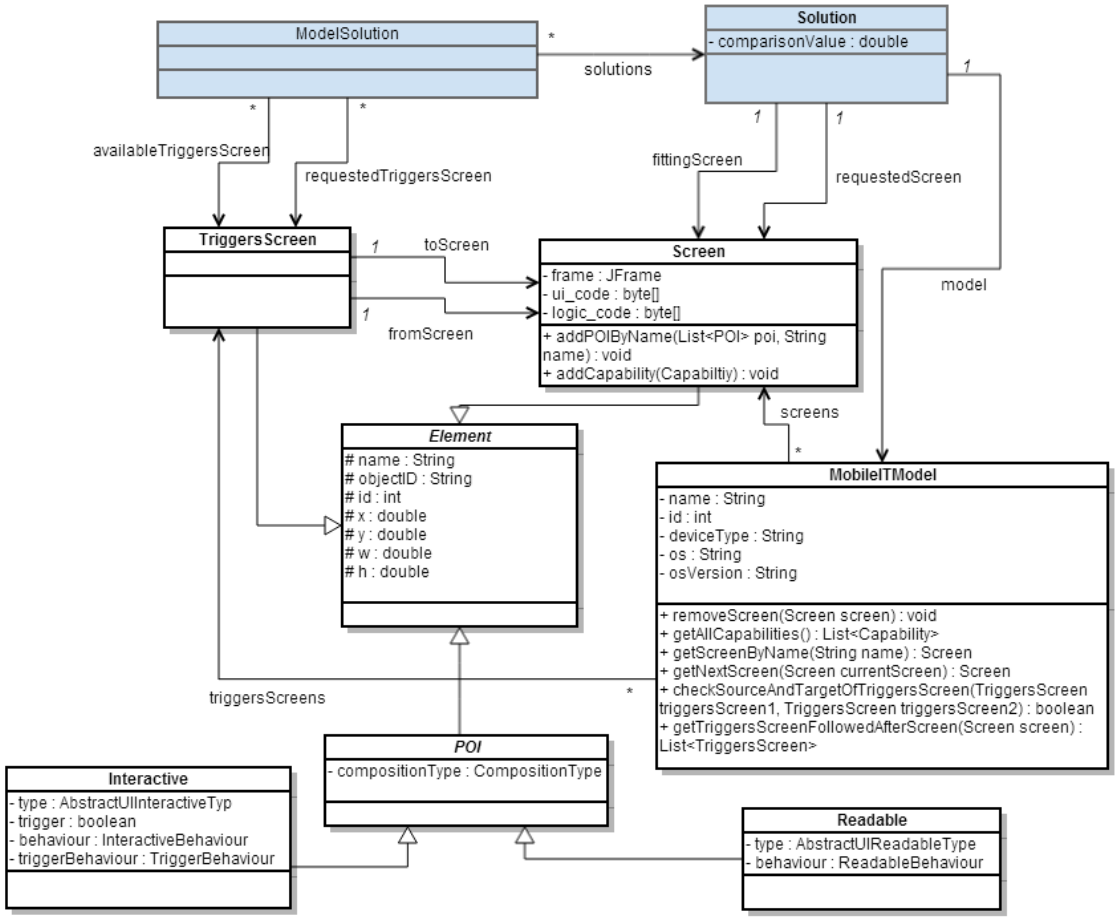


Figure 27 Class diagram for *Model Solution* and *Solution* data structure.

4.2.3.2 Pearson Algorithm

As seen in Figure 26, the *Pearson Algorithm* class is implemented as abstract class allowing only the selection of one of the three variants as applied algorithm. The *Pearson Algorithm* class is also used to implement the shared implementation of all three variants. This implementation includes the following methods:

1. *Get hash map for Pearson.* This method builds a vector of *POIs* for a certain *Screen*. This vector indicates the number of occurrences of each *POI* type in the

Screen. *POIs* are checked for similarity by calling the "*checkIfPOIsAreSame*" method. The complete pseudo-code of this method is shown in Figure 28.

```
1   Declare Hash Map for each POI type and its occurrence
2   For each POI in screen
3       Declare a boolean variable called added
4       Set added to false
5       If size of Hash Map bigger than 0
6            For each entry in Hash Map
7               Get next entry from Hash Map
8               If checkIfPOIsAreSame with parameters POI
9               and key of Hash Map entry is true
10                   Increment value of map entry by 1
11                   Set added to true
12                   Break from loop
13      If added is false
14           Put POI into Hash Map with value 1
15  Return Hash Map
```

Figure 28 Pseudo-code for *getHashMapForPearson* method.

2. *Check if POIs are same.* This method takes as input two *POIs* and compares the attributes of those two *POIs* in order to identify whether they are the same or not based on their attributes.
3. *Get Pearson correlation.* The "*getPearsonCorrelation*" method implements the *Pearson correlation coefficient* between two vectors. Those vectors are the results of the aforementioned methods and should be structured as described in Equation 4. The pseudo-code of this method is presented in Figure 30.

```
1   Declare two double variables called meanX and meanY
2   Set value of meanX and meanY to 0
3   For i = 0 to size of xVect
4       Get value of xVect at position i and add to meanX
5       Get value of yVect at position i and add to meanY
6   Set value of meanX to division of meanX by xVect size
7   Set value of meanY to division of meanY by yVect size
8   Declare three double variables called sumXY, sumX2, sumY2
9   Set value of sumXY, sumX2 and sumY2 to 0
10  For i = 0 to size of xVect
11      Add to sumXY (xVect at i  meanX) * (yVect at i  meanY)
12      Add to sumX2 (xVect at i  meanX)^2
13      Add to sumY2 (yVect at i  meanY)^2
14  Return sumXY / (sqrt(xVect at i) * sqrt(yVect at i))
```

Figure 29 Pseudo-code of *getPearsonCorrelation* method.

4. *Compare Screens.* The result of the "*compareScreens*" method is calculated by calling the aforementioned methods. The result of the "*compareScreens*" method is

the *Pearson's correlation coefficient* between "*Screen* r" and "*Screen* s". The corresponded pseudo-code is depicted in Figure 30.

```
1   Declare a Hash Map variable called mapR
2   Call getHashMapForPearson with parameter r and pass
3   result to mapR
4   Declare a Hash Map variable called mapS
5   Call getHashMapForPearson with parameter s and pass
6   result to mapS
7   Declare a List of Double variables called vectorR
8   Declare a List of Double variables called vectorS
9   For each entryS in mapS
10      Add value of entryS to vectorS
11      Declare a Boolean variable called foundInR and set
12      to false
13      For each entryR in mapR
14          If checkIfPOIsAreSame with parameters key of entrR
15          and key of entryS is true
16              Add value of entryR to vectorR
17              Remove entryR from mapR
18              Set foundInR to true
19              Break from loop
20      If foundInR is false
21          Add 0 to vectorR
22  For each entryR in mapR
23      Add value of entryR to vectorR
24      Add 0 to vectorS
25  Return value of getPearsonCorrelation method with
26  parameters vectorR and vectorS
```

Figure 30 Pseudo-code for *compareScreens* method.

5. *Find best model solutions.* The "*findBestModelSolutions*" method is used to find the best possible available model (app) solution based on the *Pearson's correlation coefficient* value for the given request. The best *Model Solution* is defined as the available model with the highest number of matched *Screens* which satisfy the specified minimum *Pearson's correlation coefficient* similarity. The pseudo-code for the "*findBestModelSolutions*" method is described in Figure 31.

```
1   Declare a List of ModelSolution variable called modelSolutions
2   For availableModel in availableModels
3       Declare a ModelSolution variable called modelSolution
4       For requestedScreen in screens of requestedModel
5           Declare a Solution variable called bestScreenSolution
6           Set comparison value of bestScreenSolution to -1
7           Declare a Boolean variable called foundSolution
8           Set foundSolution to false
9           For availableScreen in screens of availableModel
10              Declare a Solution called s
11              Set fittingScreen of s to availableScreen
12              Set requestedScreen of s to requestedScreen
13              Set comparisonValue of s to value of compareScreen
14              with parameters requestedScreen and availableScreen
15              If minimumPearsonValue and pearsonValue of
16              bestScreenSolution <= pearsonValue of s
17                  Set bestScreenSolution to s
18                  Set foundSolution to false
19          If foundSolution is true
20              Add bestScreenSolution to modelSolutions
21  Return modelSolutions
```

Figure 31 Pseudo-code for *findBestModelSolutions* method.

6. *Get model params.* The *"getModelParams"* method returns a hash map based on the *OS* and *device type* attributes of the model. This hash map is then used for database querying in order to find only available models which are of the specific *OS* or *device type*.

4.2.3.2.1 Pearson Algorithm Variant: Less Different Models

One of the algorithms which use the *Pearson's correlation coefficient* is the less different models algorithm. The main purpose of the less different models variant is to find a combination of requested *Screens*, specified in the request of the app consumer, as *Model Solution* which fulfils the minimum required *Pearson's correlation coefficient* for each *Screen* (specified by the app consumer), and requires as less as possible different apps. The operation of this algorithm can be explained through the following steps:

1. In the first step the app consumer would select the request and specify the minimum necessary *Pearson's correlation coefficient* which must fulfil each mapped *Screen*.
2. Afterwards, the algorithm would try to map each *Screen* of the request to any *Screen* in the app repository. This induces that one requested *Screen* can be

mapped to multiple available *Screens* from different models, but the found mapping is the best mapping for a certain requested *Screen* inside of this available model. The result of this step is a list of *Model Solutions* containing none or more matched *Screens* (*Screen Solutions*).

3. In the next step the list of *Model Solutions* will be sorted based on the number of possible *Screen Solutions*.
4. From the sorted list of *Model Solutions*, the one with the highest number of *Screen Solutions* will be taken as the part of the returned solution.
5. *Screen Solutions* from the rest of the list of *Model Solutions* which map requested *Screens* from the *Model Solution* with the highest number of *Screen Solutions* will be removed from each *Model Solution* which is not part of the returned solution.
6. In the last step the algorithm checks if all requested *Screens* are mapped to a *Screen* in the returned solution. In case that not all requested *Screens* could be mapped, the algorithm will return to step 2 until all requested *Screens* are mapped.

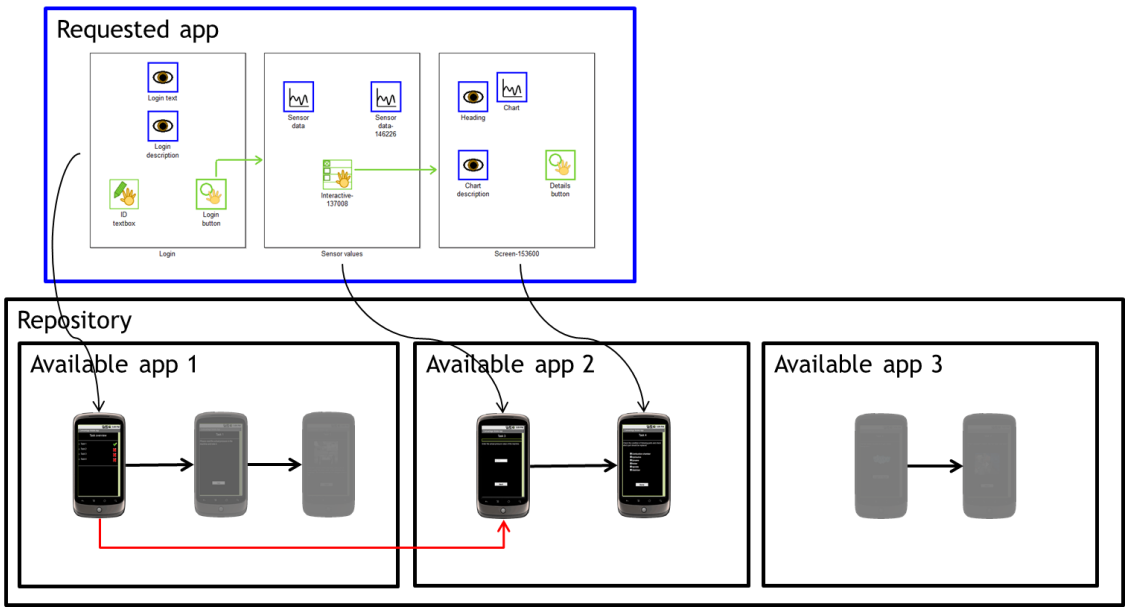


Figure 32 Example of the less different models variant.

Figure 32 presents an example of how the result and operation of the less different model variant could look. This example shows that the request contains three different *Screens*. However, the repository of apps contains three models with a different number of *Screens*. From this example it is visible that after applying the less different models variant, the first mapped app would be "Available app 2", because of the highest number of mapped

Screens. This app would be taken as the first part of the returned solution. After removing already mapped requested *Screens* from the *Screen Solutions* of the available apps, the next part of the returned solution would be "Available app 1". The *Model Solution* "Available app 3" would not be used at all as part of the returned solution. In this returned solution the app consumer would have to manually implement the connection between *Screen* one and two of the requested model.

In order to show each step in greater detail, the less different variant algorithm will be presented as pseudo-code. The two functions used for this implementation are shown in Figure 33 for "*getBestModelSolution*" and Figure 34 for "*lessDifferentModels*" function.

```
1   Declare a ModelSolution variable called bestModelSolution
2   For each availableModel in availableModels
3       Declare a ModelSolution variable called modelSolution
4       Set modelSolution to availableModel
5       If size of Solutions in bestModelSolution
6       < size of Solutions in modelSolution
7           Set bestModelSolution to modelSolution
8           Remove modelSolution from availableModels
9   For each solution in Solutions of bestModelSolution
10      Remove requestedScreen of solution from all requestedModels
11      For availableModel in availableModels
12          For each availableSolution in Solutions of availableModel
13              If requestedScreen of solution equals
14              requestedScreen of availableSolution
15                  Remove availableSolution from Solutions
16                  of availableModel
17  Return bestModelSolution
```

Figure 33 Pseudo-code for *getBestModelSolution* method.

```
1   Declare a   List of ModelSolution variable called   modelSolutions
2   Set value of modelSolutions to result of findBestModelSolutions
3   with parameters minPearsonValue, requestedModel
4   and availableModels
5   Declare a List of ModelSolution variable called bestModelSolution
6   Loop until size of Screens in requesteModel > 0
7       Add to modelSolutions result of getBestModelSolution with
8       parameters requestedModel and modelSolutions
9   Return bestModelSolution
```

Figure 34 Pseudo-code for *lessDifferentModels* method.

4.2.3.2.2 Pearson Algorithm Variant: Less New Connections

The second algorithm which is based on the *Pearson's correlation coefficient* is the less new connections algorithm. As mentioned in chapter 3.4.2, the less new connections algorithm can be seen as an extension of the less different models algorithm. However, the less new connections algorithm involves two constraints which are the following: a) each *Screen* which is a possible solution must satisfy the minimum *Pearson's correlation coefficient* value specified by the app consumer and b) the returned result should find a combination of *Screens* in which as much as possible *Triggers Screen* relations can be reused. This implies that the result will also contain as less as possible different models, which is also implemented by the less different models algorithm. Due to the complexity of this algorithm, it will be explained in following steps:

1. The first step of this algorithm is identical to the first step of the less different models algorithm described in chapter 4.2.3.2.1.
2. After the request is received by the web service, it reuses the "*findBestModelSolutions*" function and tries to map each *Screen* of the request to any *Screen* specified in the available models in the app repository. Again, it is possible that a single *Screen* of the requested model can be mapped to multiple *Screens* of different available models in the app repository.
3. In the next step the algorithm uses the *Triggers Screen* relations from the request and tries to calculate the number of mapped *Triggers Screen* relations in each available model of the app repository.
4. Afterwards the available models will be sorted ascending based on the number of matched *Triggers Screen* relations.
5. The available model with the highest number of matched *Triggers Screen* relations will be selected as the first part of the solution.
6. Mapped *Screens* and *Triggers Screen* relations will be removed from all other available models in the repository and from the requested model.
7. The algorithm returns to step 4 if not all *Triggers Screens* relations of the requested model are found and if the sum of all mapped *Triggers Screen* relations in all available models is higher than zero.

8. If no more mapped *Triggers Screen* relations exist in the available models and the requested model still contains *Screens* which are not mapped; then the algorithm finds any *Screen* from the available models which satisfy the minimum *Pearson's correlation coefficient* value, until all *Screens* are mapped.

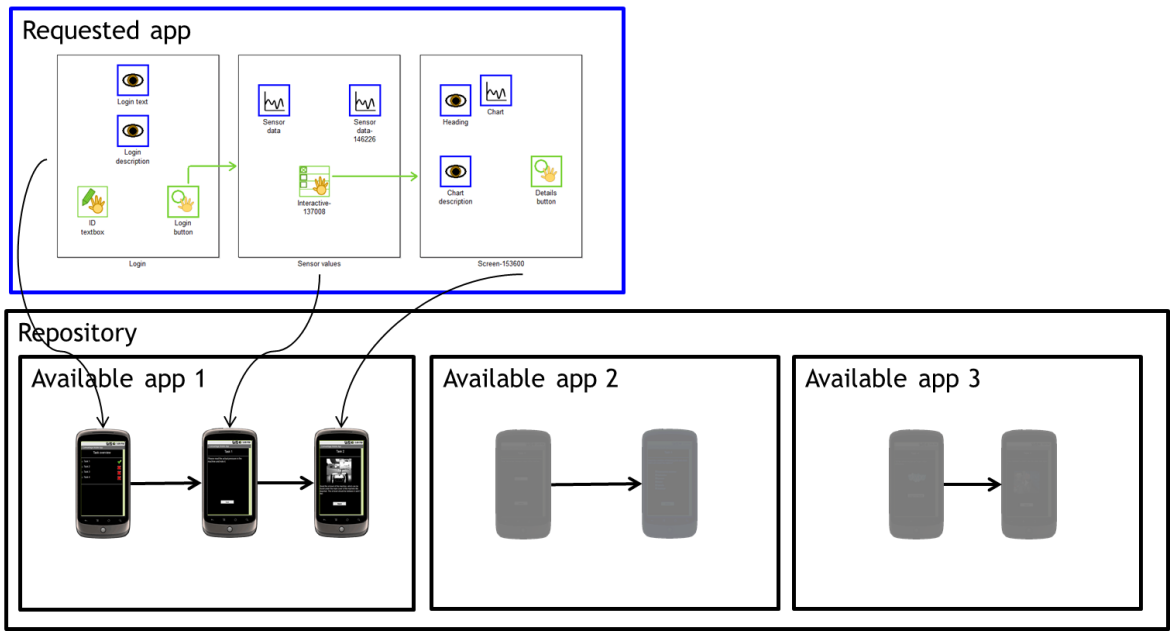


Figure 35 Example of the less new connections variant.

An example of the possible best case returned result of the less new connections algorithm is demonstrated in Figure 35. This example shows that the app consumer specified three different *Screens* in his requested model. Meanwhile, the app repository contains three apps in which one app contains three *Screens* and the other two each two *Screens*. After the less new connections algorithm is applied the possible best case solution would contain three *Screens* in the same model each connected in the right sequence as specified in the requested model. Therefore, the only possible available model satisfying this criterion would be "Available app 1". By returning this app to the app consumer he would reuse the whole app without additionally implementing any code or maybe just changing the modest fragments of the returned app.

```
1   Declare a List of ModelSolution variable called
2   mappedModelSolutions
3   For each modelSolution in modelSolutions
4       Declare a ModelSolution variable called possibleBestSolution
5       Declare a MobileITModel variable called availableModel
6       Set availableModel to model of modelSolution
7       For each firstScreenSolution in solutions of modelSolution
8           For each nextScreenSolution in solutions of modelSolution
9               For each requestedTriggerScreen in TriggersScreen of
10              requestedModel
11                  For each availableTriggersScreen in TriggersScreen
12                  of availableModel
13                      If requestedScreen of firstScreenSolution
14                      equals fromScreen of requestedTriggersScreen
15                      and requestedScreen of nextScreenSolution
16                      equals toScreen of requestedTriggersScreen
17                      and fittingScreen of firstScreenSolution
18                      equals fromScreen of availableTriggersScreen
19                      and fittingScreen of nextScreenSolution
20                      equals toscreen of availableTriggersScreen
21                          Set solutions of possibleBestSolution
22                          to solutions of modelSolution
23                          Set availableTriggersScreens of
24                          possibleBestSolution to
25                          availableTriggersScreen
26                          Set requestedTriggersScreens of
27                          possibleBestSolution to
28                          requestedTriggersScreen
29      If number of solutions in possibleBestSolution > 0
30          Add possibleBestSolution to mappedModelSolutions
31  Return mappedModelSolutions
```

Figure 36 Pseudo-code for *mapRequestedTSToAvailableTS* method.

```
1   Declare a List of ModelSolution variable called possibleSolutions
2   Declare a List of ModelSolution variable called allModelSolutions
3   Set value of allModelSolutions to result of findBestModelSolutions
4   with parameters minPearsonValue, requestedModel
5   and availableModels
6   Set value of possibleSolutions to result of
7   mapRequestedTSToAvailableTS
8   with parameters requestedModel and allModelSolutions
9   Loop until number of screens in requestedModel > 0 and number
10  of TriggersScreen in possibleSolution > 0
11      Declare a ModelSolution variable called bestModelSolution
12      Set value of bestModelSolution to result of
13      getBestModelSolution with parameters requestedModel
14      and possibleSolutions
15      Add bestModelSolution to bestModelSolutions
16      If number of screens in requestedModel > 0
17          Declare a ModelSolution variable called bestModelSolution
18          For each requestedScreen in requestedModel
19              Declare a Solution variable called bestSolution
20              Set comparison value of bestSolution to -1
21              Declare a Boolean variable called foundSolution
22              Set value of foundSolution to false
23              For each availableModel in availableModels
24                  For each availableScreen in screens of
25                  availableModel
26                      Declare a Solution variable called s
27                      Declare a double variable called pearsonValue
28                      Set pearsonValue to result of compareScreens
29                      with parameters requestedScreen
30                      and availableScreen
31                      Set comparisonValue of s to pearsonValue
32                      Set fittingScreen of s to availableScreen
33                      Set requestedScreen of s to requestedScreen
34                      Set model of s to availableModel
35                      If pearsonValue > minPearsonValue
36                      and pearsonValue > comparisonValue
37                      of bestSolution
38                          Set bestSolution to s
39                          Set foundSolution to false
40              If foundSolution equals true
41                  Add bestSolution to solutions of bestModelSolution
42  Add bestModelSolution to bestModelSolutions
43  Return bestModelSolutions
```

Figure 37 Pseudo-code for *lessNewConnections* method.

```
1   Declare a ModelSolution variable called bestModelSolution
2   For each possibleModelSolution in availableModels
3       If number of requesteTriggersScreen in
4       bestModelSolution < number of requestedTriggersScreen
5       in possibleModelSolution
6           Set bestModelSolution to possibleModelSolution
7   Remove bestModelSolution from availableModels
8   Declare a List of Screen variable called
9   removedRequestedScreens
10  For each solution in solutions of bestModelSolution
11      Remove requestedScreen of solution from requestedModels
12      Add requestedScreen of solution to removedRequestedScreens
13  For each screen in removedRequestedScreens
14      For each modelSolution in availableModels
15          For each solution in solutions of modelSolution
16              If screen equals requestedScreen of solution
17                  Remove solution from solutions of modelSolution
18  Declare an integer variable called size
19  For each requestedTriggersScreen in bestModelSolution
20      For each modelSolution in availableModels
21              If i < size and requestedTriggersScreen of
22              modelSolution contains requestedTriggersScreen
23                  Remove requestedTriggersScreen of
24                  bestModelSolution from requestedTriggersScreens
25                  of modelSolution
26  Return bestModelSolution
```

Figure 38 Pseudo-code for *getBestModelSolution* method.

```
1   Declare an integer variable called sumOfTriggersScreens
2   Set value of sumOfTriggersScreens to 0
3   For each modelSolution in availableSolutions
4       Add number of requestedTriggersScreen of
5       modelSoluton to sumOfTriggersScreen
6   Return sumOfTriggersScreen
```

Figure 39 Pseudo-code for *getNumberOfTriggersScreens* method.

The used functions and implementation of this algorithm will be demonstrated as pseudo-code in order to explain the operation of this algorithm detailed enough and from a technical view. The pseudo-code for the four used functions is demonstrated in Figure 36 for "*mapRequestedTSToAvailableTS*", Figure 37 for "*lessNewConnections*", Figure 38 for "*getBestModelSolution*" and Figure 39 for "*getNumberOfTriggersScreens*".

4.2.3.2.3 Pearson Algorithm Variant: Most Similar Screens

The last algorithm which is based on the *Pearson's correlation coefficient* is the most similar screens algorithm. This algorithm does not implement any restrictions on the returned results, and therefore it searches in all available models of the repository for

Screens which have the highest *Pearson's correlation coefficient* value for the requested *Screens*. In the best case this algorithm finds the best match for each *Screen* in the same model in which all *Screens* are already connected with each other, meanwhile in the worst case it finds again the best match for each requested *Screen* but the found *Screens* can be separated into different models and each *Screen* has to be manually connected in the right order with each other. In Figure 40 an example of the most similar screens variant of the *Pearson's correlation coefficient* is presented. In this example the worst case scenario of the algorithm is presented in which the algorithm finds the best match for each three *Screens* of the requested app, but it produces a huge effort to the app consumer of implementing the missing connections (indicated by the red connectors) between the available *Screens.*

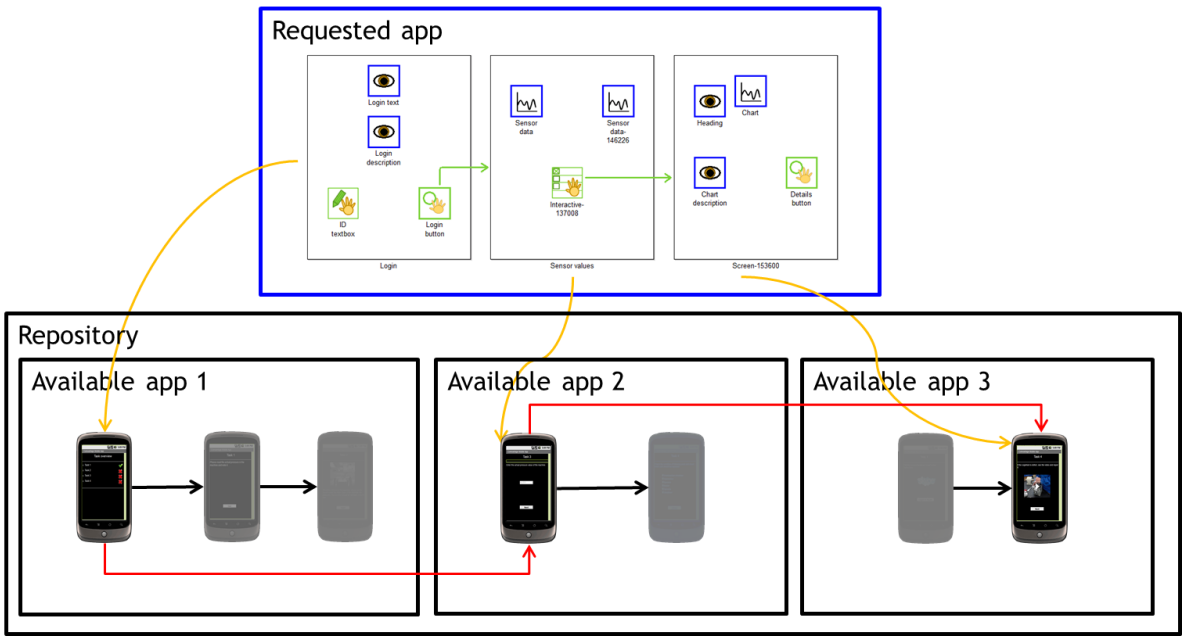


Figure 40 Example of the most similar screens variant.

The function required for the implementation of this algorithm is presented as pseudo-code in Figure 41. Altogether, it is visible that the most similar screens variant is the simplest variant of the three algorithms shown, requiring only one function for implementation.

```
1   Declare a ModelSolution variable called solution
2   For each screenR in screens of requestedModel
3       Declare a Solution variable called s
4       Declare a Solution variable called best
5       Declare a MobileITModel variable called bestModel
6       Declare a double variable called bestPearsonValue
7       Set bestPearsonValue to -1
8       For each model in availableModels
9           For each screen in screens of model
10              Declare a double variable called pearsonValue
11              Set value of pearsonValue to result of
12              compareScreens called with parameters screen
13              and screenR
14              If pearsonValue > bestPearsonValue
15                  bestPearsonValue = pearsonValue
16                  best = screen
17                  bestModel = model
18      Set comparisonValue of s to bestPearsonValue
19      Set fittingScreen of s to best
20      Set requestedScreen of s to screenR
21      Set model of s to bestModel
22      Add s to solutions of solution
23  Return solution
```

Figure 41 Pseudo-code for *mostSimilarScreen* method.

4.2.3.3 Jaccard Algorithm

As specified in chapter 3.4.2.1 the Jaccard algorithm uses the *Jaccard index* and applies it on the *Capabilities* which are assigned to the *Screens* of the model. The algorithm compares each *Capability* of the requested model with each *Capability* of the available model. This comparison results in the *Jaccard index*, which shows the similarity of the two compared models. The Jaccard algorithm is applied before the variants of the *Pearson's correlation coefficient* and can be seen as an initial filtering of the available models in the repository. The implementation of the Jaccard algorithm is separated into four functions, which can be seen in Figure 42 for "*intersection*" function and Figure 43 for "*union*" function.

```
1   Declare a List of Capability variable called intersection
2   For each r in list of requested capabilities
3       For each a in list of available capabilities
4           If name of r equals name of a
5               Add r to intersection
6               Break from loop
7   Return number of capabilities in intersection
```

Figure 42 Pseudo-code for *intersection* method.

Both functions, the "*union*" and "*intersection*", are then used for the calculation of the *Jaccard index* by dividing the value of "*intersection*" function with the value of the "*union*" function. This calculation is done in the "*jaccardIndex*" function.

```
1   Declare a List of Capability variable called union
2   Add all capabilities of available model to union
3   For each r in requested capabilities
4       Declare a boolean variable called inside
5       Set value of inside to false
6       For each a in available capabilities
7           If name of r equals name of a
8               Set value of inside to true
9               Break from loop
10      If inside is false
11          Add r to union
12  Return number of capablities in union
```

Figure 43 Pseudo-code for *union* method.

4.3 Database Layer

In this chapter the last layer, the database layer, of the implementation will be explained. The database layer is realized by using a MySQL 5.1.11 database. The implementation of the MySQL database will be explained through a database schema which implements the requirements of the described ER-diagram from chapter 3.5. In Figure 44 the MySQL database schema is presented. Key aspects of the implemented database schema which are relevant according to the ER-diagram are following:

1. The "*MobileITModel*" table stores the relevant information about the app (e.g. device type, OS, etc.).
2. The "*Screen*" table contains a foreign key to the "*MobileITModel*" table indicating to which app each *Screen* belongs. Besides, the "*Screen*" table also contains the most important information for the consumer of the app, which is the logic and UI code of the app. This information is stored into two separated columns called "*ui_code*" and "*logic_code*". Both columns are implemented as a Binary Large Object (BLOB) data type. Through this data type it is possible to store huge numbers of characters which are limited with an approximate size of 4 gigabyte.
3. The "*POI*" table contains a foreign key to the corresponded "*Screen*" table. The "*POI*" table stores the two different types of *POIs* into a single table, resulting that

the values of the single columns will not always be filled out because both *POI* types have different attributes.

4. The "*TirggersScreen*" table contains a foreign key to the "*MobileITModel*" table and two foreign keys to the *"Screen"* table, indicating the direction of the *Triggers Screen* relation.
5. The relation "described by" from the ER-diagram is resolved into a separate table called "*screen_capabilities*", because of the cardinalities of this relation. Therefore, this table contains a foreign key to the "*Capability*" table and another one to "*Screen*" table.

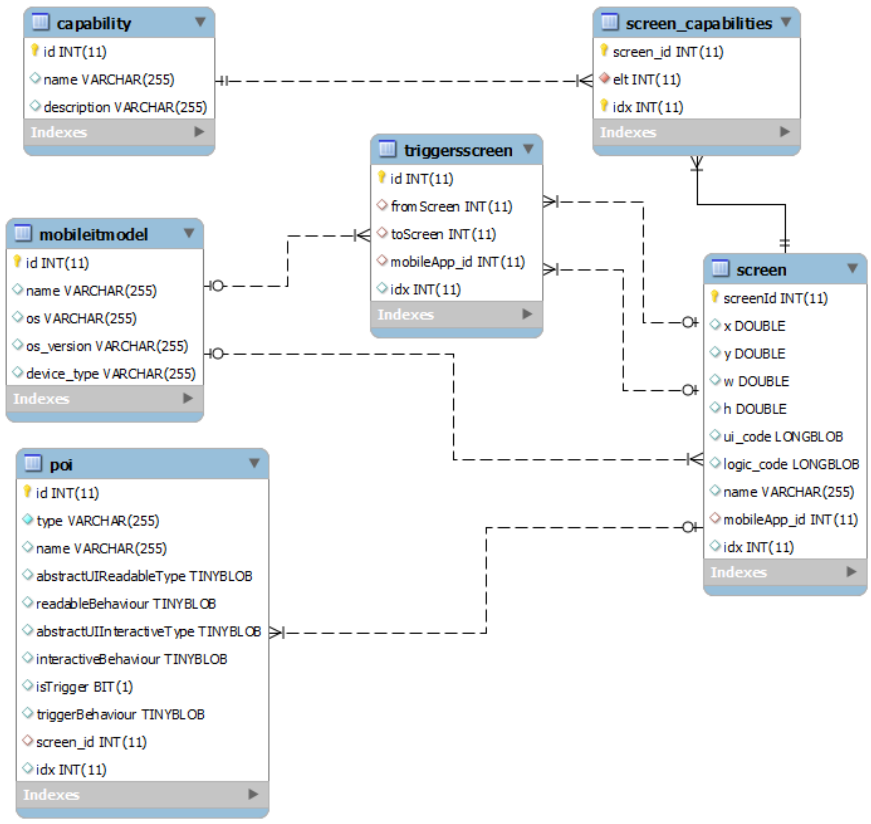


Figure 44 Database schema in MySQL.

5 User Guidelines

In this chapter the proposed procedure will be explained based on the steps defined in Figure 17. The figure shows that two different roles are involved, the app provider and consumer. Besides, it is assumed that the used tools and services which are used by the mentioned roles are provided and available. The following subchapters will explain the proposed procedure for each role individually with additional screenshots of the used tools.

5.1 Mobile App Provider

The app provider is in the first case responsible for the development of the app. In order to be able to develop a certain app it is necessary to *identify a domain* in which the app will be used. This is at the same time the first step which the app provider would execute. This step is usually done by identifying real world problems and finding a possibility for using apps for problem solving. In the next step the provider would *model the apps* which are modelled by using the *ComVantage* modelling method. Those models could then be used by the developer, based on requirements specified by a business expert, as a starting point for the app creation. The next part of this step would include the *development of the app* by using one of the technologies mentioned in chapter 2. In order to register the developed app at the provided web service it would be only necessary to define two models, which are the *Resource pool* model and the *Mobile Mockup* model. The creation of those models is done in the *ADOxx*[22] modelling toolkit displayed in Figure 45. In the next step the app provider would export his models from the *ADOxx* modelling toolkit as XML. After the models are exported as XML the app provider would *register his models* by loading the XML into the MySQL database. At this point the linking between the developed code and the registered models in the database is not covered, but as proof of concept the linking was done manually by assigning the developed code directly to the *Screens* of the apps in the database. After this step is executed the registered models would be ready for querying by the app consumer, which is described in the next chapter.

[22] http://www.adoxx.org/live/

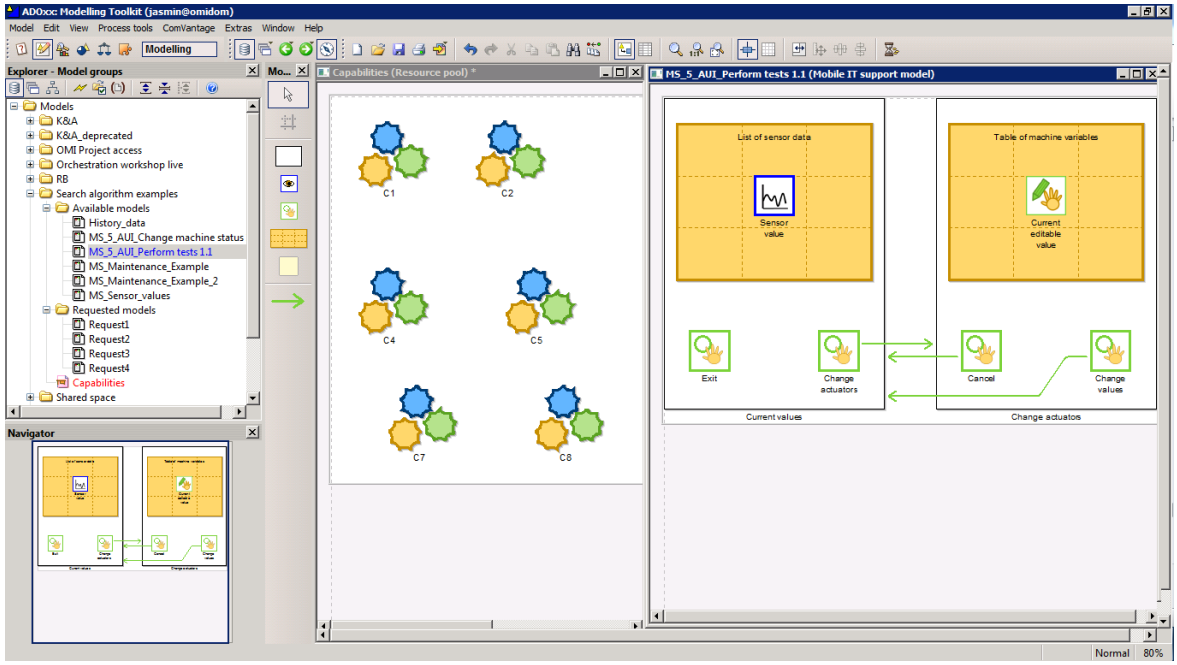


Figure 45 Screenshot of the *ADOxx* modelling toolkit with opened *Resource pool* and *Mobile Mockup* model.

5.2 Mobile App Consumer

The app consumer would execute similar steps as the app developer. The first step of the app consumer is identical to the first step of the app provider. In the following step the app consumer would only create models in the *ADOxx* modelling toolkit for the specified requirements of his app. Those models would be again exported as XML from the modelling toolkit and uploaded as query to the app discovery web service. The uploading of the models is shown in Figure 46. After the models are uploaded the web service would consume the query and apply search algorithms on the registered models of the app producers. Afterwards the search algorithm would return the results for the uploaded query back to the app consumer, who could reuse or adapt the returned code of the result. The returned results are displayed in Figure 46.

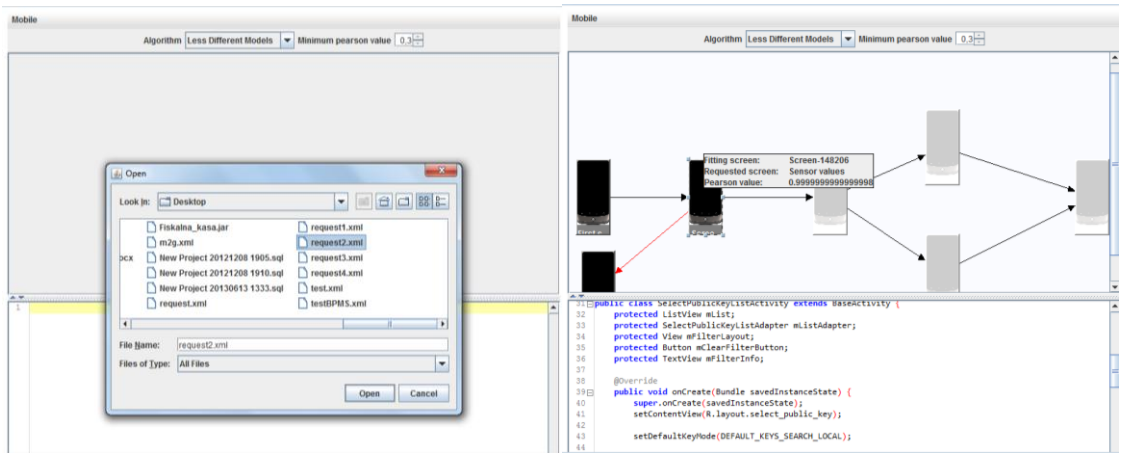


Figure 46 Uploading XML as query to web service and displaying results to the app consumer.

6 Future work

This master thesis showed a solution and prototype for app discovery. In order to provide a fully functional and complete web service for app discovery following improvements could be considered for further research and studies:

Interface for app provider. In chapter 5.1 it was mentioned that the current version of the provided web service does not implement a fully automated interface for the app provider. Such an interface should provide a possibility for the app provider to select parts of the implemented app code and assign it to the corresponding *Screen* of the *Mobile Mockup* model. This assignment could be realized by two approaches. The first approach would be completely model based, which means that the assignment of app code to *Screen* objects would be done in the modelling toolkit through specifying a specialized attribute in the *Screen* class. This attribute could be defined either as a long string containing the content of the implementation or as file path to the used implementation. In the second case it would be necessary to additionally implement a file reader in the web service, taking the input of the file and saving it into the database. The second approach would rely on the GUI of the web service. In this approach the app provider would use the current XML file and upload it to the web service, which would provide a web-based possibility for assigning code parts to *Screen* objects of the *Mobile Mockup* model.

Search algorithm. In chapters 3.4.2 and 4.2.3 the operation of the search algorithm was explained. In this explanation it was mentioned that the search algorithm will be a greedy algorithm, which means that it does not have to always deliver the best possible solution. Therefore, one possible improvement could be the extension of the search algorithm in order to provide the issuer of the query with the best possible solution. The search algorithm could also be extended by adding additional query parameters. Those query parameters would include information from the *Resource pool*, *Information space* and *Process* model types. Last but not least the search algorithm could also implement additional options for searching, like the possibility to prohibit same *Screen* solutions for different requested *Screens.*

ComVantage modelling method. The current version of the implemented *ComVantage* modelling method only provides two model types, the *Mobile Mockup* model, the

Resource pool model and the *Process* model. In order to extend the search algorithm it would be necessary to also extend the implementation of the *ComVantage* modelling method by adding the classes like *Precondition* and *Effect* to the *Resource pool* and creating an *Information space* model type with the mentioned classes from chapter 3.2.2.2.

GUI related improvements. The current version of the web service provides the issuer of the query with a *JApplet* based GUI, which means that the issuing client should have an installed version of the *Java Runtime Environment* (*JRE*) in order to be able to query the database. This additional pre-requirement for the app consumer could be avoided by providing an additional *Java-Script* based GUI, because the used technologies (*JGraphT* and *RSyntaxTextArea*) can be found also as *Java-Script* libraries.

Further testing and data collecting. Further tests should are also necessary in order to analyse and verify different parameters, which are relevant for the repository of apps. Those parameters would be related to the necessary computation time for the search algorithm and the reusability and exploitability of the returned results for the app consumer. However, in order to be able to analyse those parameters it would first be necessary to fill out the repository with a huge number of meaningful and real app implementations. Afterwards it could be possible to statistically examine the results of those parameters.

7 Conclusion

In the past, apps implemented modest and offline-based algorithms and computations, which used unpretentious hardware components. Therefore, the development process of those applications was less difficult and time-consuming. Nowadays, apps implement high complex algorithms and use different internet technologies and other protocols for communication. Those apps are currently developed in different programming languages and by utilizing different technologies in dependence of the used device and operation system. This means that for each new app it is necessary to know exactly each command of the programming language and start to write the code from scratch. However, snippets of code can be found on the internet, but the internet does not offer a structured way for querying and finding the adequate code for the requested app. In this thesis the structure of apps was analysed and out of that the requirements for an app repository including search mechanisms were identified. This search mechanism offers a model-based search for apps in which the requirements engineer could specify the requirements for a certain app without having any knowledge about programming. The created model could be used by the developer of the app to query for possible skeletons or snippets of code which could be reused or adapted by the developer. Through this app repository the necessary time and effort for developing could be reduced and the reusability of knowledge about app implementations successfully increased.

In this work an app repository was established. The establishment of this repository was done by using the web service technology and providing the access to the repository through the browser. The access to the web service was realized by implementing a unique GUI which allowed the user to analyse the results and selects the requested code. This GUI was presented in chapter 4.1. In order to define queries for the app repository it was necessary to analyse how apps are developed and built, and how apps can be defined through conceptual models. This part was presented in chapter 2. Besides of the identified query parameters it was also relevant to define and implement feasible search mechanisms which would return accurate results to the issuer of the queries. Therefore, in chapter 3.4.2 and 4.2.3 three different search algorithms were presented and explained through pseudo-code. In order to retrieve results from the mentioned search algorithms it was necessary to

store the apps and corresponding models in a structured way. The structure and realization of the implemented database for apps was presented in chapters 3.5 and 4.3.

8 Bibliography

[1] ComVantage, "ComVantage Public Deliverables," 29 10 2012. [Online]. Available: http://www.comvantage.eu/results-publications/public-deriverables/. [Accessed 13 06 2013].

[2] S. Hashimi, S. Komatineni and D. MacLean, Pro Android 2, Apress, 2010.

[3] J. Zdziarski, iPhone SDK Application Development: Building Applications for the AppStore, O'Reilly Media, 2009.

[4] S. Trewin, G. Zimmermann and G. Vanderheiden, "Abstract user interface representations: how well do they support universal access?," *SIGCAPH Comput. Phys. Handicap.*, vol. , pp. 77-84, 2002.

[5] J. Vanderdonckt, Q. Limbourg, B. Michotte, L. Bouillon, D. Trevisan and M. Florins, "USIXML: a User Interface Description Language for Specifying Multimodal User Interfaces," *Proceedings of W3C Workshop on Multimodal Interaction WMI,* vol. 2004, 2004.

[6] J. Vanderdonckt, F. Beuvens, J. Melchior and R. Tesoriero, "USer Interface eXtensible Markup Language (UsiXML), W3C Working Group Submission," 1 February 2012. [Online]. Available: http://www.usixml.org. [Accessed 23 June 2013].

[7] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, M. Florins and D. Trevisan, "Usixml: A user interface description language for context-sensitive user interfaces.," *Developing User Interfaces with XML: Advances on User Interface Description Languages,* pp. 55-62, 2004.

[8] F. Montero, M. Lozano and P. González, "IDEALXML: an Experience-Based Environment for User Interface Design and pattern manipulation.," 2005.

[9] A. Puerta and J. Eisenstein, *XIML: A Universal Language for User Interfaces,* 2001.

[10] K. Sycara, M. Paolucci, A. Ankolekar and N. Srinivasan, "Automated discovery, interaction and composition of Semantic Web services, Web Semantics: Science, Services and Agents on the World Wide Web," vol. 1, no. 1, pp. 27-46, 2003.

[11] R. Shuping, "A model for web services discovery with QoS.," *ACM Sigecom exchanges 4.1,* pp. 1-10, 2003.

[12] M. Paolucci, T. Kawamura, T. Payne and K. Sycara, "Semantic matching of web services capabilities.," *The Semantic Web—ISWC 2002,* pp. 333-347, 2002.

[13] R. Dumitru, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler and D. Fensel, "Web service modeling ontology.," *Applied Ontology 1*, vol. 1, pp. 77-106, 2005.

[14] H. Stachowiak, Allgemeine Modeltheorie, 1973.

[15] D. Karagiannis and H. Kühn, "Metamodelling Platforms," in *EC-WEB '02: Proceedings of the Third International Conference on E-Commerce and,* London, UK, 2002.

[16] H.-G. Fill and D. Karagiannis, "On the Conceptualisation of Modelling Methods Using the ADOxx Meta Modelling Platform," *Enterprise Modelling and Information Systems Architectures - An International Journal,* vol. 8, no. 1, pp. 4-25, 2013.

[17] P. Jaccard, "Nouvelles recherches sur la distribution florale.," 1908, p. 223–270.

[18] J. L. Rodgers and A. W. Nicewander, "Thirteen Ways to Look at the Correlation Coefficient," *The American Statistician,* vol. 42, pp. 59-66, 1988.

[19] F. Buschmann, Pattern oriented software architecture: a system of patters, John Wiley&Sons, 1999.

[20] R. N. Taylor, N. Medvidovic and E. M. Dashofy, Software Architecture: Foundations, Theory, and Practice, Wiley Publishing, 2009.

[21] G. Erich, H. Richard, J. Ralph and V. John, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, Inc., 1995.

Curriculum Vitae

Personal information:

Name:	Jasmin Brakmic
Degree:	BSc.
Gender:	Male
Date of birth:	27.04.1990.
Place of birth:	Zenica, Bosnia and Herzegowina
Marital status:	single
Citizenship:	Bosnia and Herzegowina

Contact information:

Address:	Zaunscherbgasse 4/404, 1210, Wien
Tel.:	+43-1-4277-789 41
	+43-6-6080-0437
Email:	jbrakmic@gmail.com

School and professional education:

1996 – 1999	Primary school in Gehlenbeck (Germany)
1999 – 2004	Primary school in Zenica
2004 – 2008	High school of Economics in Zenica
2008 – 2011	University of Vienna
	Field of study: Business informatics
2011 – 2013	University of Vienna
	Field of study: Business informatics

Language skills:

Bosnian/Serbian/Croatian (mother tongue)

German (excellent)

English (very good)

Schriftliche Versicherung

Ich versichere:

- dass ich die Diplomarbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.
- dass ich dieses Diplomarbeitsthema bisher weder im In- noch im Ausland (einer Beurteilerin/ einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- dass diese Arbeit mit der vom Begutachter beurteilten Arbeit übereinstimmt.

Datum Unterschrift