# Model-Driven Development of AJAX Web Applications

*Master's Thesis*

S.V. Vahid Gharavi

# Model-Driven Development of AJAX Web Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

S.V. Vahid Gharavi
born in Tehran, Iran

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

West Consulting B.V.
Delftechpark 5
Delft, the Netherlands
www.west.nl

# Model-Driven Development of AJAX Web Applications

| | |
|---|---|
| Author: | S.V. Vahid Gharavi |
| Student id: | 1070878 |
| Email: | `s.v.vahidgharavi@ewi.tudelft.nl` |

**Abstract**

As a response to the limited degree of interactivity in Web interactions, a new breed of web application, dubbed AJAX, has rapidly gained popularity. At the same time, model-driven software development is gaining increasing acceptance for the Web, largely due to the growing complexity of Web applications. In this thesis, we explore a model-driven approach toward the development of AJAX web applications. First, we present a metamodel, which can be used to model AJAX user interfaces. We later expand this metamodel and document the steps taken to use it to create an AJAX plugin for the open-source MDA toolkit, ANDROMDA. Lastly, we report on our work in terms of the degree of AJAX functionality in the generated web application and the approach we have taken regarding the model-driven development of AJAX web applications.

Thesis Committee:

| | |
|---|---|
| Chair: | Prof. Dr. Arie van Deursen, Faculty EEMCS, TU Delft |
| University supervisor: | Ir. Ali Mesbah, Faculty EEMCS, TU Delft |
| Company supervisor: | Dr. Ir. Nico Plat, West Consulting B.V. |
| Committee Member: | Dr. Ir. Eelco Visser, Faculty EEMCS, TU Delft |
| Committee Member: | Prof. Dr. Geert-Jan Houben, Faculty EEMCS, TU Delft |

# Acknowledgements

First and foremost, I would like to express my gratitude towards my supervisors, Ali Mesbah and Nico Plat. Ali introduced me to the concepts which form the basis of this thesis and has provided me with invaluable insight and assistance throughout the whole project. His enthusiasm regarding the subject of this study has been a great source of inspiration. I would like to thank Nico for giving me useful feedbacks on my thesis and for his flexibility during the last few hectic weeks. He has cooperated with me closely for a timely finish of this project. Thank you both for giving me a great deal of your time.

At West, I would like to thank my colleagues for their support and kindness. In particular I would like to thank Gertjan van Oosten who carefully reviewed the research report of this project, and Rob Westermann for his flexibility regarding my working hours and the occasions I needed to take time off to work on this project.

I would like to thank Prof. Arie van Deursen for the helpful meetings we had discussing the progress of my work, and his valuable comments regarding the structure of this thesis.

A special thank you goes to Behnam Jalilzadeh for proofreading chapters of this thesis and for being a great friend and housemate.

Last, but certainly not least, my dear parents and family receive my heartfelt gratitude for their continuous support and never-ending love. Thank you Narges and Mohammad for being my great little sister and brother, and thank you mom and dad for providing me with such a great opportunity.

S.V. Vahid Gharavi
Delft, the Netherlands
September 3, 2008

# Contents

# List of Figures

# Chapter 1

# Introduction

In the years following the arrival of the Internet, new web-based software technologies and platforms have been emerging in an overwhelming pace. Recently, a broad collection of new trends have appeared under the *Web 2.0* umbrella, changing the classical web interaction significantly.

A prominent enabling technology in *Web 2.0* is AJAX (Asynchronous JavaScript and XML) [10, 22], in which a clever combination of JavaScript and Document Object Model (DOM) manipulation, along with asynchronous delta-communication is used to achieve a high level of interactivity on the Web [21]. Since the coining of the term in 2005, numerous AJAX frameworks and libraries have appeared and the technology has been evolving fast.

What enables AJAX to be more responsive and interactive is the ability to exchange messages with a server behind the scenes and update individual components on the user interface as required. The intriguing feature of AJAX and what makes it different from classic Web interactions is that this communication occurs without the need for the entire web page to reload. Through such a possibility, giving users the richness and responsiveness of desktop applications has become feasible and accordingly many major web applications have employed AJAX in parts or all of their user interfaces. Others have even ported versions of popular desktop applications, such as e-mail agents and document processing software to the web.

The high pace in which technologies have been appearing for the web leaves software developers with many possibilities and options to choose from. However, it also brings with itself certain challenges. Maintaining web applications and keeping them up to date with new technologies are often complex and expensive tasks. Furthermore, the integration of different technologies, from front-end to back-end, seems challenging, yet a necessity in building enterprise web applications. When adopting a new and evolving technology such as AJAX, it is very important to be able to cope with changing environments from an architectural point of view. A framework which is fit for the project today could easily be out featured by a new one tomorrow.

One way to tackle these challenges is by abstracting from the implementations through a Model-Driven Engineering [27] approach, in which an application is defined in a modelling language and the corresponding code is automatically generated. This approach enables us to define the application once and generate the same application to different frameworks.

There are different approaches to employing a model-driven style of software development. The Object Management Group (OMG) has devised a number of standards for model-driven software development under its Model-Driven Architecture (MDA) approach [25]. The following is a summary of the main advantages of using an MDA approach for software development [24]:

**Program comprehension** prior and during the development of a web system, information on it needs to be communicated and documented. Using an MDA approach means up to date and accurate models of the software system are available.

**Evolution** an application never needing change or not going through evolution through technological advances is a myth. New technologies constantly come and disappear. With MDA the current concept can be shipped to that new 'hot technology' with less effort.

**Migration** possibility to extract models from legacy web applications and generate AJAX web applications from them (referred to as *Ajaxification* in [20]). Possibly reuse the extracted models for the next new technology.

**Maintenance** modelling software systems prior to their development leads to software which is easier to develop, maintain and integrate. Having machine-readable designs at ones disposal greatly reduces maintenance costs.

**Automation** through MDA, many parts of the development processes are automated. Best practices can be put to use in an automated fashion. Developers need not be concerned with these and other implementation details and can therefore concentrate on the semantics of the application. The models can go through automatic validation, testing and be checked for inconsistencies.

**Integration** easier integration between different web components when models are available. For instance, the possibility for an application based on the combination of JSF-Spring-Hibernate, JSF-Hibernate or JSF-Spring-EJB to be created in an automated way.

In response to the inception of the MDA standard by the Object Management Group (OMG), a number of tools, both open-source and commercial, have been created to implement the guidelines of MDA and allow for the transformation of software models to code for a variety of different platforms. These tools typically take a precise software model as input and generate more specific models or code, as output. Sometimes the generated code is ready for use and does not need any manual implementation. This is usually the case for MDA toolkits that generate applications tailored towards a specific problem domain. In other cases, the toolkits generate the framework of an application and leave out specific business logic code to be implemented by the developers.

In this thesis, we explore an MDA approach toward AJAX web applications. The first step in an MDA approach to application development is modelling; We therefore look into how an AJAX web application can be modelled, while having the ultimate goal of code generation from the models in mind. To this end, we propose a UML scheme for modelling AJAX user interfaces and attempt to incorporate this modelling scheme in an AJAX plugin for an open-source MDA toolkit called ANDROMDA. We document the steps taken in the implementation phase and present the results of the implementation. We conclude with an evaluation of the generated AJAX web application, the approach we have taken in realising our goal, and plans for future work.

A summary of the work presented in this thesis was published [12] in the proceedings of the 7th International Workshop on Web-Oriented Software Technologies (IWWOST'08). Implementation details not available when the paper was being written can be found in Chapter 9 of this thesis.

The remainder of this thesis is organised as follows. We start in Chapter 2 by briefly exploring the related work on current model-driven approaches for the web. We continue in Chapter 3 by giving a more detailed introduction to the concepts mentioned in this chapter and other concepts used throughout the rest of this thesis. In Chapter 4, we present the problem statement we attempt to solve and in Chapter 5 we outline the approach we take to solving that problem. Subsequently we focus on the difference between classic and AJAX web applications in Chapter 6, using a sample web application used for managing persons. In Chapter 7 we discuss ANDROMDA, an open-source MDA toolkit and continue in Chapter 8 by presenting a metamodel which can be used for modelling AJAX user interfaces. Next, we document the implementation process of that metamodel for ANDROMDA in Chapter 9 and in Chapter 10 we present the application resulting from our work. An evaluation of our work and ANDROMDA is given in Chapter 11. We conclude this thesis and lay out ideas for future work in Chapter 12.

# Chapter 2

# Related Work

There have already been several developments in the area of MDA for the web. In this section we explore some of the most relevant approaches.

The Web Modelling Language (WebML) [5] is a visual notation for specifying the structure and navigation of legacy web applications. The notation greatly resembles UML class and Entity-Relation diagrams. Presentation in WebML is mainly focused on look and feel and lacks the degree of notation needed for AJAX web user interfaces [4, 28].

Conallen [6] proposes a UML profile (Figure 2.1) for modelling web applications. This approach is widely referenced as a web modelling scheme. Koch and Kraus [14, 16, 15] propose UWE, a UML profile and notations for modelling the navigation and conceptual aspects of a web application. Both approaches are aimed at classic multi-page web applications.

RUX-model [4, 28] is an MDA approach towards Rich Internet Applications (RIA) based on WebML. In this approach WebML is extended with notations which indicate whether certain data is stored or presented on the client or the server. In the latter stages, the RIA is modelled using the *RUX-model* notation (not UML) and subsequently Flash-based RIA instances (e.g., OpenLaszlo) are generated. According to the authors, also the same models can be used to generate AJAX applications. It appears that only the user interface part is generated by RUX-model since issues such as the back-end or the toolkits employed in the generation of the whole web application are not mentioned. RUX-model is currently unavailable for experimenting.

Another attempt at an MDA approach for RIAs is found in [17]. The approach is based on XML User Interface description languages and XSLT is used as the transformation language between the different levels of abstraction. Again, this approach is oriented towards the User Interface and lacks flexibility in an MDA approach for the whole web application. It also lacks a visual notation and cannot be modelled using existing CASE-tools. This is also true in the case of other XML-based UI languages such as XUL [1], XAML [2] and UIML [1] which do not offer the simplicity of a visual model.

Visser [29] proposes a domain-specific language called WebDSL for developing dy-

[1] `http://www.mozilla.org/projects/xul/`

[2] `http://www.xaml.net/`

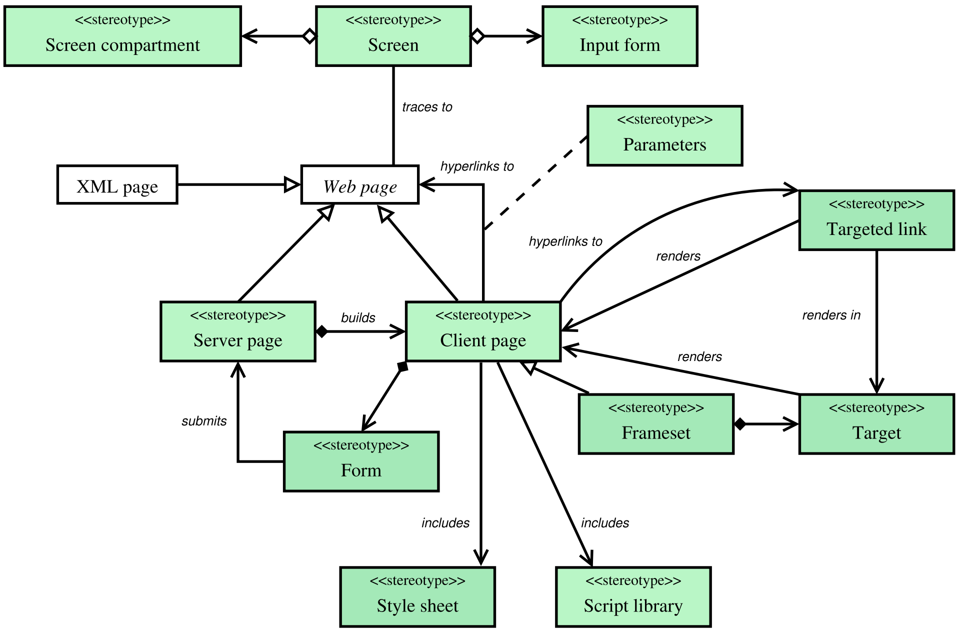

<<stereotype>> Screen compartment
<<stereotype>> Screen
<<stereotype>> Input form
traces to
<<stereotype>> Parameters
XML page
Web page
hyperlinks to
<<stereotype>> Targeted link
hyperlinks to
renders
<<stereotype>> Server page
builds
<<stereotype>> Client page
renders in
renders
submits
<<stereotype>> Frameset
<<stereotype>> Target
<<stereotype>> Form
includes
includes
<<stereotype>> Style sheet
<<stereotype>> Script library

Figure 2.1: Conallen's Web metamodel

namic web applications with a rich data model. WebDSL applications are translated to classical Java web applications, building on the JavaServer Faces (JSF), Hibernate, and Seam frameworks. An approach to generate an Echo2 web application from WebDSL can be found in [13]. WebDSL was in an experimental phase when this project was started and was therefore not considered as an alternative.

openArchitectureWare[3] (oAW) is currently one of the leading open-source MDA generator frameworks. It is very extensible and supports model-to-model and model-to-text transformations. However, oAW does not come with complete integrable transformations for different web platforms, nor does it define platform-independent elements which can be used across web applications (and possibly other paradigms). The lack of the aforementioned possibilities are certainly not necessities for a good MDA framework, yet we consider them very convenient when working with a set of varying technologies intended for web applications.

Comparing ANDROMDA and oAW, a point in favour of openArchitectureWare is the clean separation between the different models. Where in ANDROMDA it is not possible[4] to tweak the PSM before transforming to code or other PSMs, this remains attainable with openArchitectureWare. On the whole, oAW has a higher level of adherence to the MDA standard. As we mentioned above, the availability of a set of ready to use cartridges can be seen as a benefit of ANDROMDA.

Aside from RUX, none of the approaches mentioned above explore an MDA approach

[3] `http://www.openarchitectureware.org`

[4] The latest version of ANDROMDA is 4.0 and our work is based on version 3.3

for AJAX web applications.

# Chapter 3

# Concepts and Terminology

In this chapter we briefly describe the concepts and technologies which will be referred to throughout this thesis.

## 3.1 AJAX

AJAX (Asynchronous JavaScript and XML) [10, 22] is a technique used for creating interactive Web user interfaces. The interactivity aspect of an AJAX web application manifests itself in that an AJAX web application can consist of a *single page*, which does not reload as a result of round trips to the server. A famous example of AJAX can be found in Google Maps[1], where users can seamlessly explore maps as if all the data has already been loaded in the client, while in reality, data is constantly being pulled in from the servers to satisfy the users' dragging and zooming.

This form of communication is fundamentally different from classic web applications, where any communication with the server involves the whole page being fetched and rendered again in the browser. The possibility to communicate with the server asynchronously, provides a richness and responsiveness similar to desktop applications. To achieve this higher level of interactivity, asynchronous communications with the server are combined with dynamic alterations of the user interface. In this manner, a user interface can communicate with the server and only update relevant parts (components) of the page, leading to a fine-grain interaction. On the whole, AJAX incorporates the following techniques:

- The `XMLHttpRequest` [30] object for asynchronous communication with the server.
- Dynamic alteration of the user interface using DOM
- Presentation using XML, XSLT and CSS.
- JavaScript for integrating the previous techniques.

Today, many web sites employ AJAX in their user interfaces, though often to different degrees. We present three main categories as to the extent AJAX is used in a user interface:

[1] `http://maps.google.com/`

1. No AJAX at all. All communication with the server is synchronous.
2. AJAX in parts of the web application. For instance, the photo sharing web application Flickr[2] employs AJAX only for a specific number of features, such as editing the title and label of a photograph.
3. Fully single-page AJAX. For example, the new version of Yahoo! Mail[3] which is an e-mail application with the look and feel of a desktop e-mail agent, such as Mozilla Thunderbird or Microsoft Outlook.

In the context of this thesis, a complete single-page interface is our understanding of an AJAX web application and the application we wish to generate.

## 3.2 Model-Driven Architecture

Model-Driven Architecture (MDA) [24] is an approach to software development introduced by the OMG in 2001. As defined by the OMG, MDA provides an approach for, and enables tools to be provided for [24]:

1. Specifying a system independently of the platform that supports it;
2. Specifying platforms;
3. Choosing a particular platform for the system; and
4. Transforming the system specification into one for a particular platform.

In other words, MDA proposes to start a software project by first defining platform-independent models. Subsequently, transformations should be defined to transform these models, possibly in a number of iterations, to other models which are platform-specific. The ultimate step is to generate code from the lowest-level platform-specific models. MDA formalises a set of terms to denote the abstraction levels of the models. Important terms are:

**PIM** a Platform Independent Model (PIM) is a view of a system exhibiting a specified degree of platform independence so as to be transformable to different platforms of similar type.

**PSM** a Platform Specific Model (PSM) is a platform specific view of the system. Specific information about the platform may be used in the views for better understanding and implementation of the system.

MDA encourages declaring a PIM using a platform independent language such as UML. By using this approach, developers can ignore secondary details while focusing on the core

[2]`http://www.flickr.com/` observed on 25-08-2008
[3]`http://mail.yahoo.com` observed on 25-08-2008

functionality of the system. When the PIM is modelled sufficiently, transformation techniques are employed to transform it to a PSM. Subsequently, a PSM may be used for (semi-)automatic code generation. However, generating code from a PIM is also a possibility [24]. Figure 3.1 demonstrates an example of how a PIM can be used to generate code for two different platforms.

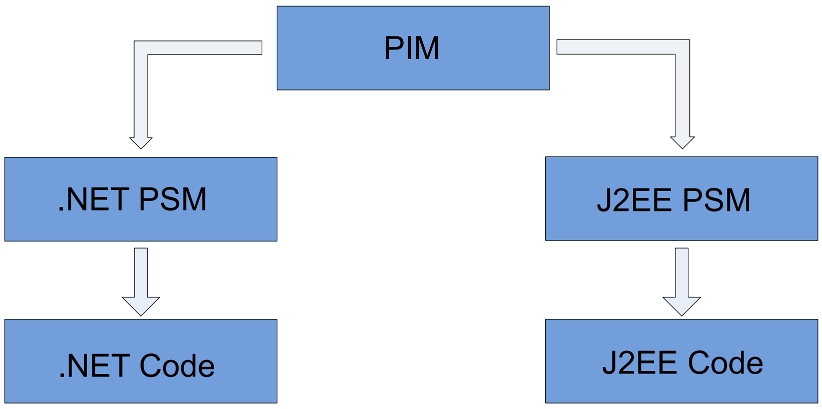


Figure 3.1: Example of PSMs and code being generated for different platforms from a single PIM.

## 3.3 OMG Standards Related to MDA

In order to accommodate the MDA process for software development, the OMG has released a set of standards that can be employed during the development process. In this section we briefly discuss these standards.

### 3.3.1 Unified Modelling Language and Extension Mechanisms

The Unified Modelling Language (UML) [3] is a general-purpose modelling language defined by the OMG. It includes a set of predefined notations for expressing models of a software system. While offering a range of diagrams for a variety of different aspects of a software system, UML may not be well suited to cover certain specific problems, such as defining a user interface of a software system, or modelling a system based on a very specific problem domain. For cases like these, UML offers an extensibility mechanism which permits developers to extend the language. These mechanisms consist of *stereotypes*, *tagged values* and *constraints*. Stereotypes allow the definition of new building blocks if these are needed for expressing a new type of concept. Tagged values can be used to extend the properties of a building block, making it possible to create new information in that element's specification. Constraints modify the semantics of building blocks by adding rules or modifying current ones. Together, the extensibility mechanisms of UML allow software architects to tailor UML for their specific needs. We explain these extensibility mechanisms in more detail in the following sections:

**Stereotypes** The standard UML package contains several types of building blocks. These consist of structural, behavioural, grouping and annotational blocks, which should be enough to be able to model most software-intensive systems. But sometimes one needs more specific and tailored building blocks in order to express concepts in a crisper fashion, in a new problem domain, and using primitive building blocks. Booch *et al.* point out that a stereotype should not be viewed as a parent class in a parent/child generalisation. Rather, a stereotype should be seen as a meta-type, a new building block next to existing ones, just as *class* is a meta-type. In web-based systems for example, a primitive building block representing a web page may be a candidate for a new stereotype. One defines stereotypes by enclosing them in guillemets (such as «webpage») and placing them above the name of the element.

**Tagged values** In UML, all building blocks have a set of properties. These properties define the character of a building block. For instance, classes have names, operations and attributes. Relationships have multiplicities, role names and directions. While using stereotypes we can add new building blocks, with tagged values it is possible to add new *properties* to them. Tagged values can be applied to existing UML elements as well as new stereotypes.

Once more, Booch *et al.* [3] point out that tagged values should not be viewed in the same way as class attributes. Tagged values do not apply to instances, but to elements themselves. For example, a tagged value *version* can be added to different stereotypes in a class diagram. Graphically, tagged values are represented by a string containing a name (the tag) and a value. This is placed under the name of the element they serve as a property for.

**Constraints** All building blocks in UML have their own semantics. A dependency relationship has for example as its semantics (rules) that the target is dependent on the source. With constraints, new rules can be added and semantics can be modified. For example, a *secure* rule can be added to an association implying the constraint that the association is a secure link. A constraint spanning multiple elements is also possible. An *or* constraint between two associations implies that only one of them is possible at any moment. The Object Constraint Language (OCL) [31] is an example of language used to define constraints on UML elements.

Besides UML and its extension mechanism, the following terms are referred to throughout this thesis:

**Meta Object Facility (MOF)** Whereas UML extensibility mechanisms provide a way to extend UML and create a *UML Profile*, MOF [26] can be used to create new meta-models such as UML. These meta-models will no longer be an extension to UML, but will be independent metamodels, parallel to it.

**XML Metadata Interchange (XMI)** For the exchange of models in a platform-independent format, the OMG proposes the XMI standard. It should be possible to express any model defined by a MOF-based meta-model using XMI. A platform-independent format to exchange models allows a wide range of CASE tools to be used for modelling, since most CASE tools support import/export from/to XMI.

**Object Constraint Language (OCL)** This standard can be used to apply constraints on models of any metamodel described using MOF. For instance, an OCL [31] rule can be applied to a UML class to indicate that the type of a certain parameter should be specified. OCL can be used in early model validation, or model understanding. OCL has been developed at IBM but is now an OMG standard.

## 3.4 Domain-Specific Languages

An alternative to UML for modelling web applications is a Domain Specific Language (DSL). A Domain Specific Language (DSL) is usually a high level language which is specifically designed for solving problems of a certain domain. DSLs often borrow parts of their syntax from the vocabulary of the problem domain. This is the converse case of general-purpose languages such as Java or UML, which are meant for solving problems from a wide range of domains. Van Deursen *et al.* [8] propose the following definition for DSLs:

> A *domain-specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

DSLs are usually small languages, hence they are often referred to as micro-languages or little languages [2]. In Section 3.6 we discuss the advantages/disadvantages of UML and DSLs for the purpose of this project.

## 3.5 MDA Toolkits

MDA toolkits provide a framework within which metamodels and transformations can be defined to be used in model-driven software development. Two major open-source MDA frameworks are openArchitectureWare[4] and ANDROMDA[5].

ANDROMDA is a well-known open-source MDA toolkit that has gained popularity for the availability of ready to use transformations for a number of different web technologies and platforms. These transformations, so-called *cartridges*, allow for the generation of code for several different web technologies from the same UML model definitions. For the front-end of web applications, cartridges exist for generating code for the JavaServer

[4] `http://www.openarchitectureware.org/`

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

Faces[6], Struts [7] and .NET[8] technologies. Until this moment, there is no available transformation for AJAX nor any other type of Rich Internet Application (RIA). On the forums of ANDROMDA it can be read that an AJAX transformation is desirable, especially so since the ever-growing popularity of AJAX web applications.

The choice for ANDROMDA has been made mainly because of the availability of cartridges for different web application platforms, something not available with openArchitectureWare. An AJAX cartridge could greatly benefit from the possibility to be hooked up with different back-end technologies, which are model-driven themselves. Another major asset for ANDROMDA is the user interface modelling mechanism, which is already transformed to JSF and Struts PSMs in their respective cartridges. As a consequence, we can focus on the AJAX PSM and its transformations. As a result of the lack of availability of ready to use transformations for openArchitectureWare, such a user interface metamodel is also unavailable.

## 3.6 Modelling Language

The foremost problem when discussing an MDA approach for AJAX web applications is how to go about modelling the user interface. We discussed this problem in the literature study [11] preceding this project. There we found two main approaches to modelling (AJAX) user interfaces. We describe these below, listing their advantages and disadvantages:

**UML** As a general-purpose modelling language, it should be possible to model user interfaces to some extent using UML. We see the following advantages:

- UML is universally known and extensively used as a software modelling technique. UML is arguably the most well established modelling language for software development.
- Wide tool support for UML. There are numerous tools available for making models with UML.
- Specifications such as Java Metadata Interface (JMI) already make UML accessible programmatically through XMI.
- Possibility to employ MDA frameworks which have already defined platform-independent metamodels for the user interface.

The main disadvantage to using UML would be that it is not sufficient enough to express enough detail for certain aspects of the user interface such as layout and positioning.

[6] `http://java.sun.com/javaee/javaserverfaces/`

[7] `http://struts.apache.org/`

[8] `http://www.microsoft.com/NET/`

**DSL** The main advantage of DSLs is that the language designers have the freedom to make the language as broad and customised as possible in order to capture the desired amount of detail. For user interfaces, the level of detail can increase rapidly due to the detail possible in a user interface. This however might cause maintenance-related problems when certain aspects need to be changed, such as redefining the transformations [7]. Furthermore, it might result in the DSL being not so easy to use. Another disadvantage is that we have not encountered a stable open-source MDA toolkit which defines a platform-independent metamodel for user interfaces using a DSL.

After the goal of this thesis is presented in Chapter 4, we discuss the final choice of modelling language in Chapter 5.

# Chapter 4

# Problem Statement

The advantages of MDA [24] and the technological advances in web technologies under the *Web 2.0* umbrella have encouraged us to explore the possibilities of creating a tool which can transform a PIM, used as a basis to create a variety of different applications, to a model which specifies an AJAX web application. From the AJAX-specific model, we intend to generate the code for an AJAX web application. In this context, we define the goal of this project as follows:

> Applying the MDA approach to the process of creating an AJAX web application. The starting point of the web application should be a platform-independent model from which an AJAX user interface is generated. It should be possible to generate web user interfaces for other platforms from the same platform-independent model, and to combine the AJAX front-end with other technologies, typically technologies which are used in the back-end of web applications.

While MDA tools exist for the generation of web applications for specific platforms such as J2EE[1] and .NET[2], we have not encountered any such transformation to AJAX web applications. With such a transformation, models used to generate legacy web applications can be used to generate single-page web applications and vice versa. Also the same models can be used to generate code for a variety of different AJAX frameworks.

We would like to point out that in the context of the web, MDA toolkits are very useful for larger, form-oriented web applications, or for web applications often expanded to include new functionality. This is a logical statement since employing MDA is only effective in cases where a common and not so trivial task has to be performed repeatedly. Examples of the aforementioned classes of applications can easily be found when starting a large web application, or when an existing web application needs to be adapted to use a new technology. Currently, there is a common desire for these types of web applications to use AJAX seeing that the interactivity and usability of such applications can be improved considerably.

[1]http://java.sun.com/javaee/

[2]http://www.microsoft.com/NET/

With reference to the goal discussed above, we define the following *research questions* which we reflect upon in Chapter 11:

**RQ 1** How can we model an AJAX web application?

**RQ 2** How suited is the chosen modelling language for modelling AJAX user interfaces.

**RQ 3** What is the degree of AJAX functionality in the generated web application? Also, what do we mean by an AJAX web application and when are we satisfied by the amount of AJAX functionality in such an application.

**RQ 4** Regarding the incorporation of the modelling scheme into an MDA framework: how do we provide a mapping between the generic platform-independent meta-models and those of the cartridge metamodel?

**RQ 5** If required, how can choices for different AJAX components be propagated through the PIM, while maintaining the PIM platform independence?

In the literature study of this project we examined several different approaches focused on classic web and desktop applications [11]. Also several different approaches to modelling AJAX web applications using UML were proposed. In this report, the choice for a modelling approach should be finalised and further expanded to cover a more complete set of possible AJAX components.

# Chapter 5

# Approach

Based on the discussed concepts of Chapter 3, mainly ANDROMDA and UML, we propose the following solution in order to reach the goal stated in Chapter 4:

> Creating an AJAX transformation for the open-source MDA toolkit ANDROMDA. The toolkit takes UML models as input and employs an AJAX framework for generating AJAX code.

Figure 5.1 depicts a bird's-eye view of the problem statement. The solution presented above fits in the segment depicted in blue.

To realise this solution, we have created an AJAX cartridge for the open-source MDA toolkit ANDROMDA. From an implementational point of view, such a cartridge can be created using many different approaches. In this chapter we discuss the approach we have taken to solve the problem.

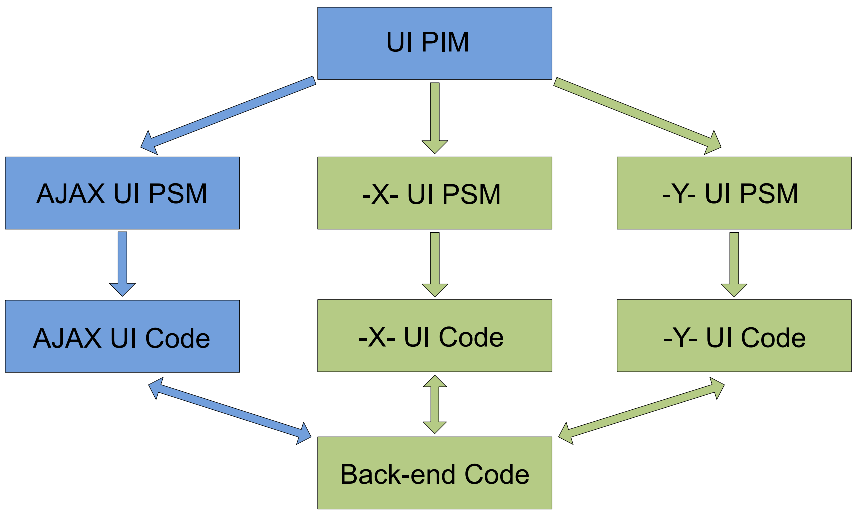


Figure 5.1: Bird's-eye view of the problem statement

A first step which has to be taken regardless of the way we decide to implement the cartridge, is creating an AJAX metamodel which is to be used as the basis for a generated PSM. Given that the implementation is based on ANDROMDA, it is evident that the PSM will be expressed using UML building blocks.

In regard to the implementation, many options are available to choose from. The initial option we pursued was creating a cartridge which creates code for a prominent AJAX-centric framework. We specifically looked at the Google Web Toolkit (GWT)[1] and Echo2[2] frameworks. After assessing these options we realised that they are not the best solutions in the scope of this project, since through such an approach a cartridge has to be built completely from scratch. Furthermore, issues such as compatibility with back-end code generated by other ANDROMDA cartridges would have to be taken into consideration, distracting from the main goal of generating an AJAX user interface. Since generation of an AJAX user interface was the primary goal of this project, using an AJAX framework based on JSF seemed as the most efficient option to model and generate a single-page AJAX web application. The reason for this being that a JSF cartridge already exists for ANDROMDA and we can base our implementation on this cartridge. In this way, code can be reused and we can focus on the AJAX user interface, while relying on the JSF cartridge for back-end compatibility.

Based on this assumption, we have examined a set of AJAX implementations of the JSF framework in order to choose a solution to use in the cartridge. Specific options we have considered have been JBoss RichFaces[3] and ICEFACES[4]. Many other options exist and an overview of existing AJAX JSF implementations and their feature-sets can be found in `http://www.jsfmatrix.net/`. Our final choice was in favour of ICEFACES, based on the set of AJAX features, components, available documentation and support. It is worth noting that the offered features in these frameworks are rapidly increasing as the technology advances and our choice regarding the choice of AJAX framework has to be evaluated considering the period it was taken in.

After choosing ICEFACES as the AJAX framework to use, we initially attempted to modify the existing JSF cartridge by incorporating ICEFACES functionality. There we faced the problem of having to remove an external library (Oracle's ADF) from the JSF cartridge first (mentioned in Chapter 9). After this was accomplished, we began incorporating ICEFACES components in the JSF cartridge. This task however did not go as smooth as we had anticipated, since AJAX required the structure of the front-end templates to be changed dramatically and testing the cartridge would not be possible until a fair amount of implementation had been done. Furthermore, comprehending the code of the JSF cartridge was not always trivial since source code and cartridge documentation is almost non-existent. We therefore attempted to start a bottom up approach, building the cartridge from scratch but copying main JSF parts that were easily understandable and common between the approaches. This was the ultimate approach pursued in this project, and we documented this in Chapter 9.

[1] `http://code.google.com/webtoolkit/`

[2] `http://echo.nextapp.com/site/echo2`

[3] `http://www.jboss.org/jbossrichfaces/`

[4] `http://www.icefaces.org/`

# Chapter 6

# Example Application

In order to demonstrate the AJAX features which the cartridge should provide we will use a simple web application which is used to manage persons. This sample application is called PERSONMANAGER and contains the *create*, *read* and *delete* functions. For each function we describe the views that are involved:

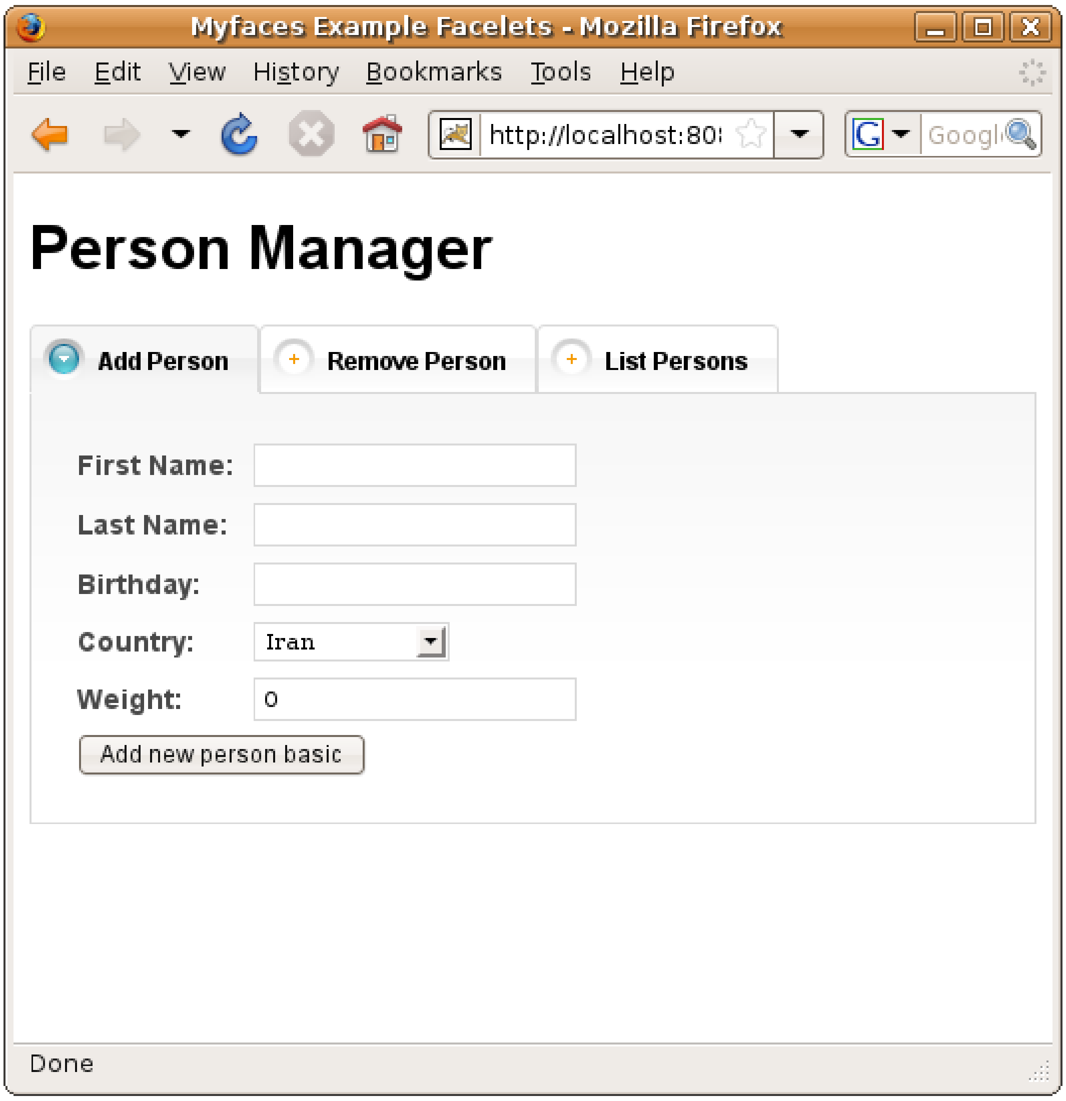
**Add** Consists of two views used to pick up sufficient information about a person and add it to the database. In the first view, the basic information (e.g. name) of a person is entered in a form. In the second phase, the detailed information (e.g. password) is filled. Clicking on a submit button in the first view renders the second view, and doing the same on the second view, should redirect to the list view in order to view the result.

**List** Consists of a view which displays all the persons in a database using a table. The table has a column for every field of information a user has to enter in the add views.

**Remove** A view used to remove a person from the database. The *id* of the person to remove is entered in a single text field and by submitting, the person is removed from the database. Just as in the second *add* view, submitting leads to the *list* page being displayed.

We have created PERSONMANAGER in both AJAX (using ICEFACES) and classic JSF in order to demonstrate the differences between the two user interfaces. Also, by creating an AJAX version we can define the web application we aim to generate. We describe the two types of user interfaces for PERSONMANAGER below:

**Classic JSF** In classic user interfaces, clicking on each of the three menu items causes the entire page to reload in order for the desired page to be displayed. The same principle also holds when views change in a use case, such as in the *add* use cases and when one use case flows into another, for instance when the list of persons needs to be displayed after a new person is added to the database. Figure 6.3 depicts the *list persons* screen of PERSONMANAGER generated using the ANDROMDA JSF cartridge, connected to a Spring-Hibernate back-end.


Myfaces Example Facelets - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:80
Google
Done

Person Manager

Add Person | Remove Person | List Persons

First Name:

Last Name:

Birthday:

Country: Iran

Weight: 0

Add new person basic

Figure 6.1: PERSONMANAGER *add person basic* screen using AJAX

**AJAX Using ICEFACES** With AJAX, only the subset of the page containing the relevant information can be reloaded. Figures 6.1 and 6.2 display the AJAX version of the web application when viewing the *add person basic information* and *add person detailed information* pages respectively. Clicking on the *Add new person basic* button of Figure 6.1 only replaces the input fields and the button with the content shown in Figure 6.2. Clicking on the tabs only loads the corresponding tab contents, and never does the whole page get reloaded. Using the ANDROMDA JSF cartridge, which generates a classic web application, each click causes the whole page to be reloaded.
We will use PERSONMANAGER as an example in the remainder of this thesis to clarify ANDROMDA concepts, our AJAX metamodel and the implementation of the cartridge.

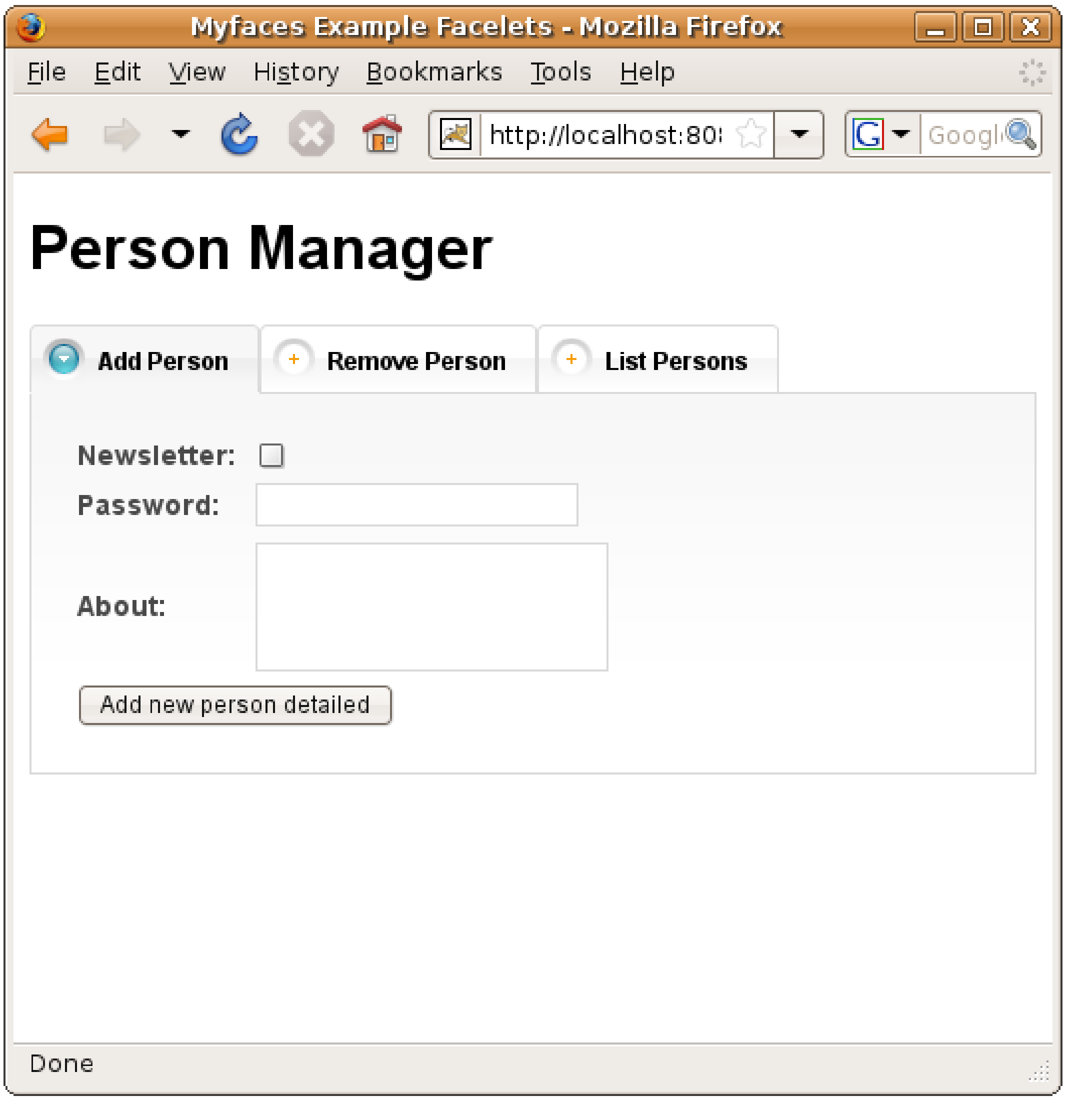
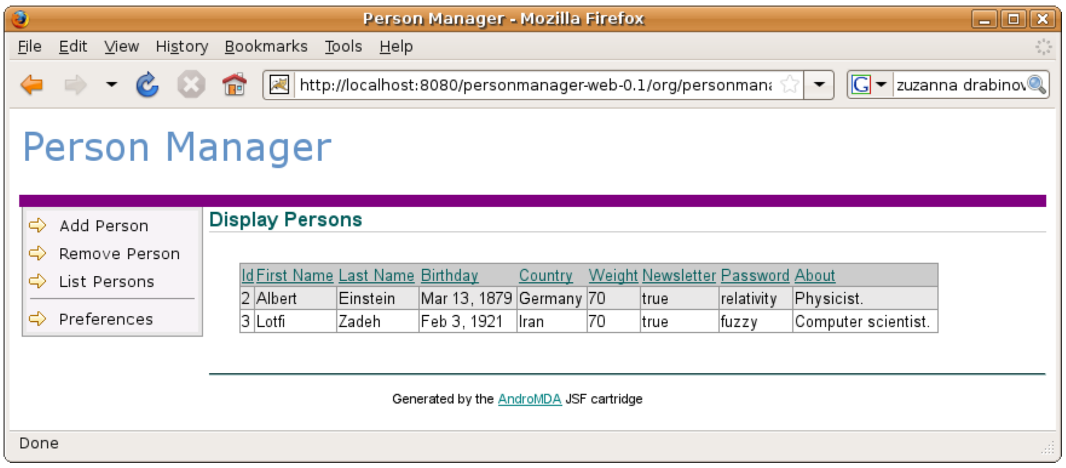


Figure 6.2: PERSONMANAGER *add person detail* screen using AJAX


Person Manager - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/personmanager-web-0.1/org/personmana
zuzanna drabinov
Person Manager
Add Person
Remove Person
List Persons
Preferences
Generated by the AndroMDA JSF cartridge
Done

Display Persons

| Id | First Name | Last Name | Birthday | Country | Weight | Newsletter | Password | About |
|---|---|---|---|---|---|---|---|---|
| 2 | Albert | Einstein | Mar 13, 1879 | Germany | 70 | true | relativity | Physicist. |
| 3 | Lotfi | Zadeh | Feb 3, 1921 | Iran | 70 | true | fuzzy | Computer scientist. |

Figure 6.3: PERSONMANAGER *list persons* screen using the JSF cartridge.

# Chapter 7

# MDA with ANDROMDA

ANDROMDA is an extensible MDA toolkit that transforms UML models to code for a variety of platforms for which a transformation (cartridge) is available[1].

While ANDROMDA can be extended to produce code for any software platform, it already contains an array of ready to use cartridges for common web related technologies such as JSF, Struts, Spring, .NET and Hibernate. The focus of ANDROMDA, as can be seen through the available cartridges, is an MDA approach to *web development* rather than anything else. As a result, developers can leverage the availability of the cartridges to generate code for the most common aspects of a web application. Figure 7.1 demonstrates the general transformation process from model to code for a three-tier web application, adhering to the Model-View-Controller (MVC) architectural pattern.

While some MDA toolkits claim to generate all the code for a given model, some generate the common coding tasks leaving only the core business logic (typically found in the *controller* of the MVC pattern) to the developers. ANDROMDA fits in the latter category of MDA tools.

In the rest of this chapter we present an introduction to ANDROMDA and its workflow. Moreover, we provide an example of an input model that can be used to generate a three-tier web application using the cartridges ANDROMDA currently offers.

## 7.1 Structure of ANDROMDA Web Applications

Current available ANDROMDA cartridges deal with the following web application layers:

**Presentation layer** interacts with the user of the web application. Example of generated files are web pages, backing beans and controller classes.

**Business layer** provides the presentation layer with a lean interface for carrying out transactions. Behind this interface is often a *service* component for accessing the data layer.

[1] It is worth pointing out that our work has been based on the latest stable version of ANDROMDA which is version 3.3. A version 4 is in the makings though it seems that progress has been temporarily halted:
`http://galaxy.andromda.org/forum/viewtopic.php?t=5643`

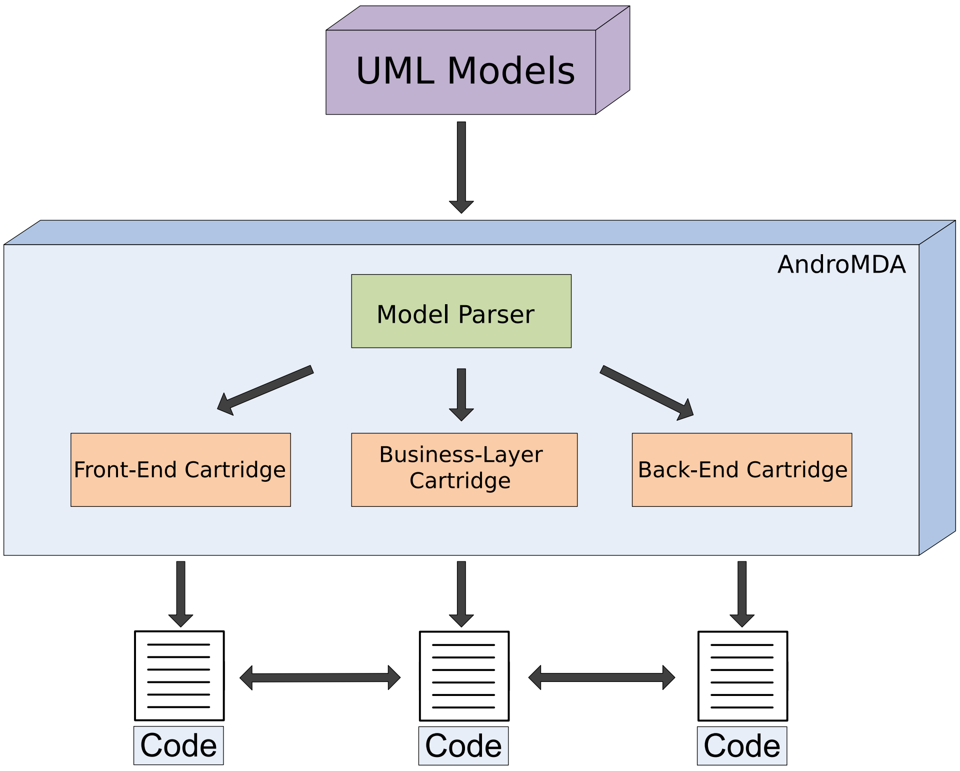


Figure 7.1: ANDROMDA transformations

**Data access layer** defines a clear Application Programming Interface (API) through which the data store can be manipulated. The data access layer hides the semantics of the data store, making the underlying technology irrelevant for the business layer. It provides out of the box Create-Read-Update-Delete (CRUD) functionality.

**Data store layer** represents the actual data store, could be a database or a file system.

ANDROMDA takes the following general steps to achieve the model-to-code transformation.

1. Loading the models in XMI format into memory.

2. Providing access to elements in the models through the ANDROMDA base *metafacades* (which will be discussed shortly).

3. Allowing cartridge templates access to the metafacades in order to write the generated output. Templates could also access platform-specific metafacades which are defined in the cartridge and specify the base metafacades for a given platform.

The first two steps of the above list are carried out in the ANDROMDA core engine and do not differ per cartridge. The last stage is cartridge-specific and is performed differently for each cartridge. Below we attempt to make these steps clearer by describing the stages in more detail.

## 7.2 Loading UML Models into Memory

In order for ANDROMDA to gain access to the models which serve as its input, the recent versions rely on the JMI [23] specification for reading the models and making them programmatically accessible through the Java programming language. JMI can therefore be referred to as an *XMI to Java mapping* specification capable of reading any model described by any modelling language, which is based on the Meta Object Facility (MOF). This makes a wide range of CASE tools and domain-specific metamodels compatible with ANDROMDA. The NetBeans MDR [19] implementation of JMI by the NetBeans Group[2] is the JMI implementation used by ANDROMDA.

## 7.3 ANDROMDA Metafacades

To shield the complexity of the Java representation of the input model, and to add some useful behaviour to it, ANDROMDA uses a mechanism called *metamodel facades*, or *metafacades* for short. Based on the Facade design pattern [9], metafacades provide an interface to the models which is tailored for use by the cartridge templates.

Figure 7.2 may somewhat clarify the use of metafacades. It demonstrates a simplified model-to-code process for a UML `Attribute` element found in an ANDROMDA input model. In the figure, a different colour corresponds with each ANDROMDA step introduced in the beginning of this chapter. Elements have been given a different `Attribute` could for example be an attribute of a UML class or association. The first phase of the transformation process consists of acquiring a JMI representation of the `Attribute` element from the XMI version of the model. From this stage onwards, an instance of `AttributeFacade` becomes the only contact point between the cartridges and any `Attribute` instance found in the models. In Figure 7.2, `AttributeFacade` provides an interface for `Attribute` while adding convenience methods which can be benefited by the cartridges. Some of these methods pick up the UML stereotypes and tagged values applied to an `Attribute` element in the model. Others can be used to pick up specific properties of an `Attribute` instance.

In Figure 7.2 methods have been included which pick up the stereotypes and tagged values applied to an `Attribute` instance. Other methods can be used to pick up specific properties of an `Attribute` instance. Just below the `AttributeFacade` class name of Figure 7.2 an OCL constraint is visible which defines validation rules for the `Attribute` element. In the case of the validation rule in the figure, an ANDROMDA user will receive a validation error in case an attribute is specified in the model without having a corresponding type.

Another advantage of the model encapsulation by metafacades is that the templates are unaffected by changes in the metamodel technology. For example, ANDROMDA cartridges can process models based on UML 1.4 or UML 2.0 by using the common base metafacades.

[2] `http://mdr.netbeans.org/`

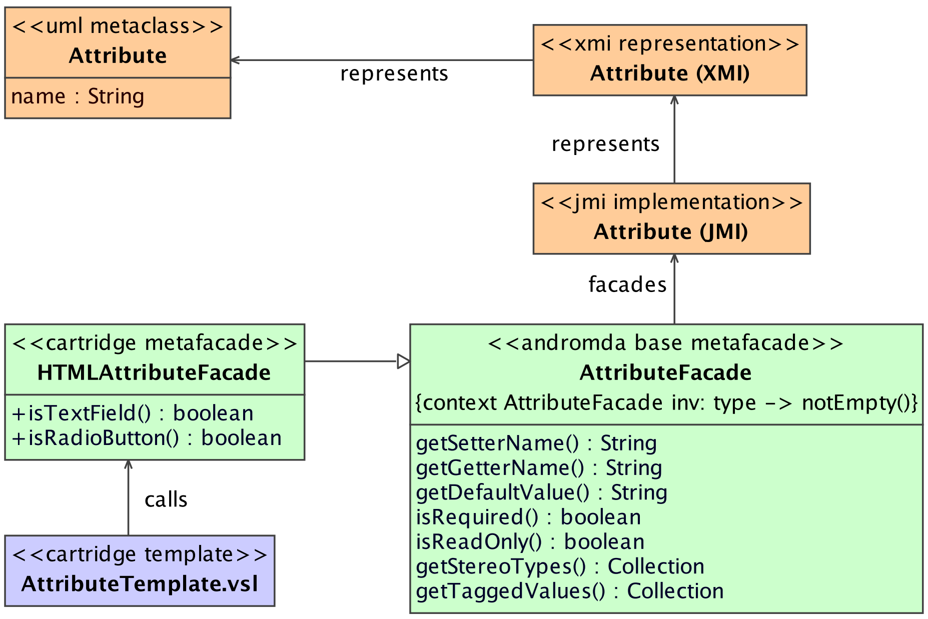


Figure 7.2: ANDROMDA metafacades

While the base metafacades are simple interfaces to the model elements, the cartridge metafacades can extend these to specify platform-specific logic and therefore in effect carry out the transformation from PIM to PSM. In Figure 7.2 `AttributeFacade` has been extended by a cartridge metafacade `HTMLAttributeFacade`. Consider this metafacade as part of a very simple HTML cartridge which given a UML class and some variables, generates an HTML form with some fields. Once the templates of this cartridge gain access to the `Attribute` instances of the model through `HTMLAttributeFacade`, they can call its methods to determine what output to generate. An example of such a method is `isTextField` which might rely on tagged values applied to an attribute to come up with its answer.

## 7.4 ANDROMDA Templates

Two Java template engines, FreeMarker[3] and Apache Velocity[4], are supported by ANDROMDA and can be used in the template files. Apache Velocity seems the most popular of the two, since it is used almost exclusively in the available cartridges. As hinted in the previous section, through these Java template engines it is possible to call metafacades' methods and write output accordingly. Figure 7.3 demonstrates an Apache Velocity template snippet which could write the output for the example HTML cartridge mentioned in the previous section. The template snippet uses methods both from the base metafacade

[3] `http://freemarker.sourceforge.net/`
[4] `http://velocity.apache.org/`

```
#if ($htmlAttribute.textField)
  <input type="text" readonly="$htmlAttribute.readOnly"
  value="$htmlAttribute.defaultValue"/>
#elseif ($htmlAttribute.radioButton)
  <input type="radio" readonly="$htmlAttribute.readOnly"
  value="$htmlAttribute.defaultValue"/>
#end
```

Figure 7.3: Example template for an HTML input

(`AttributeFacade`) and the cartridge metafacade extending it (`HTMLAttributeFacade`) to determine what HTML *input* type to write and what values to give to its attributes, respectively.

## 7.5 Adjusting the Output

In all probability, what ANDROMDA generates for the front-end needs to be adjusted by the developers. This might be because the layout needs to be changed, or the structure of the page needs to be slightly tweaked with. It should be noted that a Cascading Style Sheets (CSS) file is generated for ANDROMDA front-ends which could be used to adjust the layout to fit ones needs. CSS might however not be powerful enough for achieving the desired goals. There are two solutions to this. The less recommended way is to have ANDROMDA generate the output first and then adjust that output to match the desired layout. One can do this safely by copying the adjusted sources from the *target* directory of the generated application to the *src* directory, in order to avoid the adjusted output to be overwritten in subsequent runs. The downside of this approach is that it is not scalable, in that, it has to be repeated for new generated code. A better way to achieve this is to modify the template files so that they generate the desired output. This might initially require more effort than adjusting the output each time for new content, but is the most scalable solution in the long run.

## 7.6 Modelling for ANDROMDA

In order to get an impression of the ANDROMDA input, in this section we discuss some modelling aspects and present parts of the UML model for the PERSONMANAGER application. This model can be transformed to a three-tier JSF-Spring-Hibernate web application. Other cartridges could also be used, for instance the Struts cartridge as an alternative to the JSF cartridge. The basic structure of the model is as follows: a UML use case diagram is used to divide the application in several use cases. Each use case is specified further using a UML activity diagram. In the activity diagrams, controller methods are called which delegate to the backend methods. The controllers and backend classes are modelled in a UML class diagram. Below we explain these concepts in more detail.

### 7.6.1 Modelling the Page Flow

The page flow modelling in ANDROMDA is done using UML use case diagrams and UML activity diagrams, with the latter specifying the individual use cases of the former.

#### Use Cases

An ANDROMDA front-end is mandatorily split up in one or more use cases. Each use case defines its own specific logic. Typical use cases in an e-commerce web application could be *login*, *purchase* or *add new item*. To define the page-flow within each use case, an activity diagram is used which is linked to the use case.

Figure 7.4 depicts how the use cases of the PERSONMANAGER example are modelled in ANDROMDA. Two UML stereotypes are visible. `FrontEndUseCase` is applied on use cases which have to be rendered in the front-end, and `FrontEndApplication` is applied on the use case which has to be the starting point of the web application. Hence in the PERSONMANAGER front-end, three use cases are presented to the user and the web application starts with displaying the *List Persons* use case.

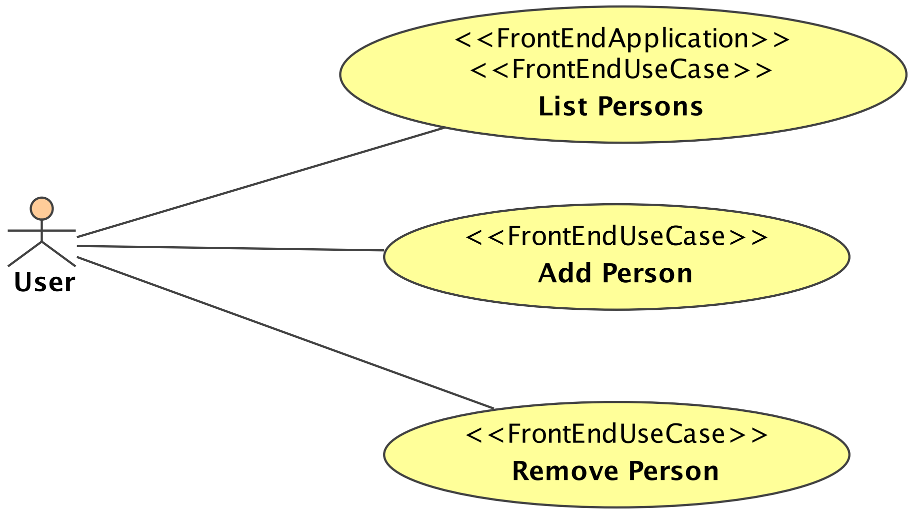


Figure 7.4: ANDROMDA use case diagram

#### Activity Diagram

The page flow inside each of the use cases of Figure 7.4 is defined in a UML activity diagram. In order to link a use case to an activity diagram, the activity diagram should be defined as a child element of the use case. In case a CASE tool does not support this feature, a tagged value can be defined and applied on a use case which specifies the name of the activity diagram corresponding with it. In order to specify the flow in a UML activity diagram, ANDROMDA uses the following elements:

**Initial state:** Starts the flow in the activity diagram. There can be only one initial state and one outgoing transition from it.

**Server-side states:** Server-side states perform some logic on the server. They are often used to initialise content going in, or to process information coming out of a web page.

**Client-side states:** Represent states rendered on the client-side, or in simpler terms, web pages. Combined with transitions, they are configured to pick up desired data using a form.

**Transitions:** Define the navigation between the states. Attributes on transitions define the data flow between the states as explained in more detail below:

- Attributes on a transition exiting a client-side state mean input fields should be rendered for them on that client-state.
- Attributes on a transition entering a client-side state mean they are rendered on that client-side state.
- Attributes on a transition entering a server-side state can be used in the logic on that state.
- Attributes on a transition exiting a server-side state represent the output of the logic on that state.

UML tagged values are used to specify the type of input field that has to be rendered. For instance, a `@andromda.presentation.web.view.field.type=radio` tagged value applied on a given attribute on a transition exiting a client-side state renders it as a HTML radio button on that state.

**Final states:** Mark the end of the flow in an activity diagram. There can be multiple final states in an activity diagram. Branching could occur, based on the result of a call on a server-side state. A final state should contain the name of the next use case to which the user should be forwarded to. For instance, the name of a final state in a *login* use case could be *show homepage*, which refers to the *Show Homepage* use case in the model.

In Figure 7.5, the activity diagram corresponding with the *Add Person* use case is depicted (the activity diagrams corresponding with the *List Person* and *Remove Person* use cases can be found in Appendix A). There are two client-side states responsible for picking up the *basic* and *detailed* information of a new person who is to be added in the PERSONMANAGER web application (see Chapter 6 for more information on PERSONMANAGER). Some of the attributes, such as `firstName` and `lastName` are rendered as HTML text fields. Text field is the default value of the the `@andromda.presentation.web.view.field.type` tagged value and therefore does not need to be specified. Other attributes however should be rendered using a different input type. For these tagged values refer to Table 7.1.

| | |
|---|---|
| country: | `@andromda.presentation.web.view.field.type=list` |
| newsletter: | `@andromda.presentation.web.view.field.type=checkbox` |
| password: | `@andromda.presentation.web.view.field.type=password` |

Table 7.1: *Add Person* attributes and their corresponding tagged values

In case any of the attributes need to be initialised, this can be done in a server-side state preceding the client-side the attribute is to be rendered on. This is the case for the *country* attribute which has to be rendered into a list of countries from which a user can select an item. The initialiser of the list is to be found in the *Initialise Fields* server-side state, which contains a call to a controller method.

Both client-side states in Figure 7.5 are succeeded by server-side states, which process the retrieved data. The parameters coming out of the client-side states have to match the parameters of the functions on the server-side states, which are also methods of the controller classes.

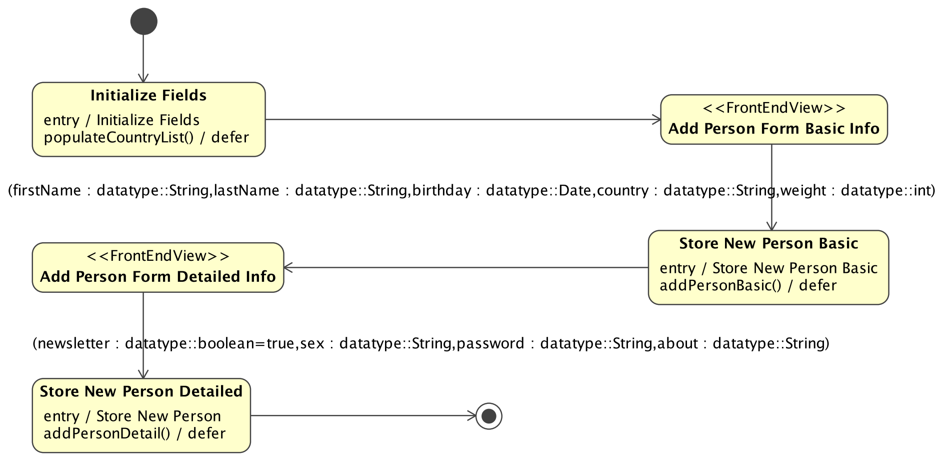

Initialize Fields
entry / Initialize Fields
populateCountryList() / defer
<<FrontEndView>>
Add Person Form Basic Info
(firstName : datatype::String,lastName : datatype::String,birthday : datatype::Date,country : datatype::String,weight : datatype::int)
Store New Person Basic
entry / Store New Person Basic
addPersonBasic() / defer
<<FrontEndView>>
Add Person Form Detailed Info
(newsletter : datatype::boolean=true,sex : datatype::String,password : datatype::String,about : datatype::String)
Store New Person Detailed
entry / Store New Person
addPersonDetail() / defer

Figure 7.5: ANDROMDA activity diagram

### 7.6.2 Modelling the Controllers, Services and Entities

All server side calls from the activity diagrams are defined in the controllers, which are depicted in Figure 7.6. The methods in the controllers are stubs generated by the front-end cartridge and have to be implemented by the developer. Typically, they delegate the task at hand to the corresponding method of the *Service* class. The `Service` stereotype is used by the *Spring*[5] cartridge to create a layer between the front-end and backend, which also contains the business logic. The methods of a *Service* class also have to be implemented by hand.

[5] `http://galaxy.andromda.org/docs-3.2/andromda-spring-cartridge/howto3.html`

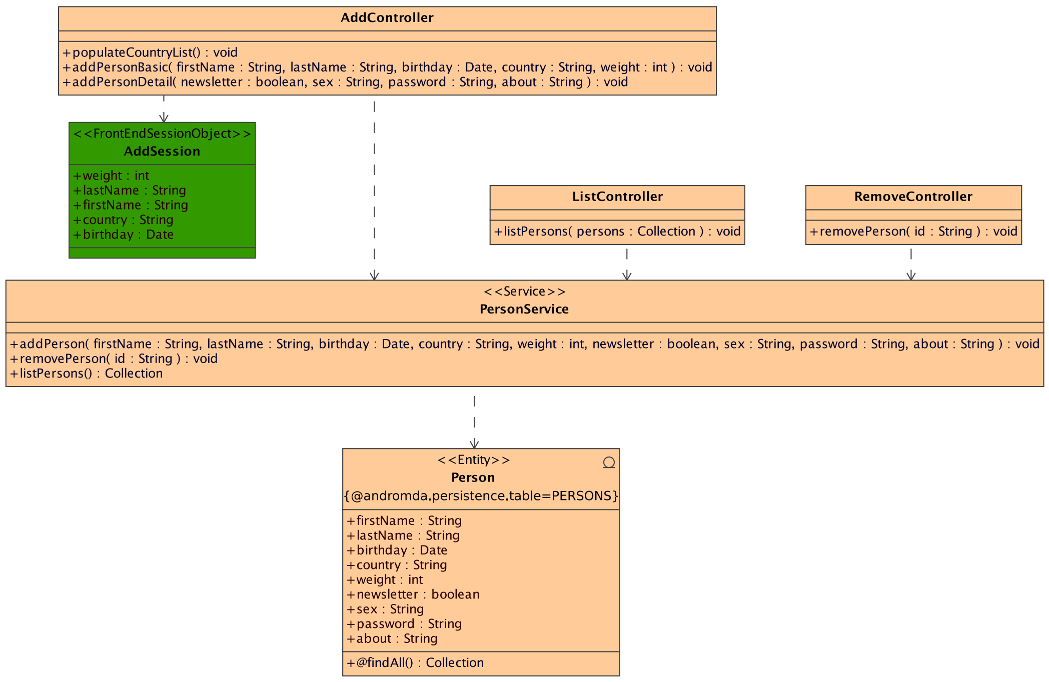

AddController
+populateCountryList() : void
+addPersonBasic( firstName : String, lastName : String, birthday : Date, country : String, weight : int ) : void
+addPersonDetail( newsletter : boolean, sex : String, password : String, about : String ) : void
<<FrontEndSessionObject>>
AddSession
+weight : int
+lastName : String
+firstName : String
+country : String
+birthday : Date
ListController
+listPersons( persons : Collection ) : void
RemoveController
+removePerson( id : String ) : void
<<Service>>
PersonService
+addPerson( firstName : String, lastName : String, birthday : Date, country : String, weight : int, newsletter : boolean, sex : String, password : String, about : String ) : void
+removePerson( id : String ) : void
+listPersons() : Collection
<<Entity>>
Person
{@andromda.persistence.table=PERSONS}
+firstName : String
+lastName : String
+birthday : Date
+country : String
+weight : int
+newsletter : boolean
+sex : String
+password : String
+about : String
+@findAll() : Collection

Figure 7.6: ANDROMDA class diagram

In Figure 7.6 we make use of a session bean to store the intermediary (basic) person information retrieved in the first view. After retrieving the rest of the needed information (detail), the transaction of adding a person is carried out by calling the `addPerson` method of the service class. This method has been implemented in `handleAddPerson(...)` in Figure 7.7. The controller method employing the session bean is presented in Figure 7.8.

The data store of the web application is represented in the `Person` class marked with an `Entity` stereotype. This stereotype is affiliated with the ANDROMDA Hibernate cartridge which generates a Hibernate `Persons` entity. No manual coding is needed for the Hibernate cartridge. In Figure 7.7 this entity is retrieved using the `getPersonDao` method. Setting up a database is a question of providing the correct database address and Java Database Connectivity (JDBC) driver, which is done in the Apache Maven[6] Project Object Model (POM) of the project. ANDROMDA generates Structured Query Language (SQL) statements for creating the tables corresponding with the Hibernate entities.

### 7.6.3 The Resulting Web Application

After implementing the generated *controller* and *service* classes according to the code fragments provided, we should have a working application. The only thing which still has to be done is configuring access to a database and running the generated SQL scripts to create the

[6] `http://maven.apache.org/`

```
public class PersonServiceImpl extends
    org.personmanager.service.PersonServiceBase {
  protected void handleRemovePerson(java.lang.String id)
      throws java.lang.Exception {
    this.getPersonDao().remove(Long.valueOf(id));
  }

  protected java.util.Collection handleListPersons() throws java.lang.Exception {
    return this.getPersonDao().findAll();
  }

  protected void handleAddPerson(String firstName, String lastName,
      Date birthday, String country, int weight, boolean newsletter,
      String password, String about) throws Exception {
    this.getPersonDao().create(firstName, lastName, birthday, country, weight,
        newsletter, password, about);

  }
}
```

Figure 7.7: The service class methods after implementation

```
public class AddControllerImpl extends AddController {
  public void populateCountryList() {
    AddPersonFormImpl form = getAddPersonForm();
    form.setCountryValueList(new Object[] { "Iran", "Kazakhstan",
        "Netherlands", "South␣Africa" });
    form.setCountryLabelList(form.getCountryValueList());
  }

  public void addPersonBasic(AddPersonBasicForm form) {
    final AddSession addPersonSession = getAddSession();
    addPersonSession.setFirstName(form.getFirstName());
    addPersonSession.setLastName(form.getLastName());
    addPersonSession.setBirthday(form.getBirthday());
    addPersonSession.setCountry(form.getCountry());
    addPersonSession.setWeight(form.getWeight());
  }

  public void addPersonDetail(AddPersonDetailForm form) {
    final AddSession addPersonSession = getAddSession();
    try {
      this.getPersonService().addPerson(addPersonSession.getFirstName(),
          addPersonSession.getLastName(), addPersonSession.getBirthday(),
          addPersonSession.getCountry(), addPersonSession.getWeight(),
          form.isNewsletter(), form.getPassword(), form.getAbout());
      removeAddSession();
    }
    catch (Exception e) {
      e.printStackTrace();
      throw new RuntimeException(e);
    }
  }
}
```

Figure 7.8: The methods of the Add Person controller class after implementation

needed tables. The application generated by using the JSF, Spring and Hibernate cartridges has been illustrated in Chapter 6.

## 7.7 Lack of PSM Modifiability

As a last remark in this chapter on ANDROMDA, we would like to point what we perceive as its major limitation. Using ANDROMDA, the PSM model instance which is generated from the PIM only resides in memory as bytecode during the code generation process. As a result of this limitation, the PSM cannot be modified before code is generated. This somewhat distances ANDROMDA from a pure MDA approach, wherein it should be possible to generate a PSM from the PIM, improve it with platform-specific information, and ultimately generate code. Since the PSM layer cannot be modified using ANDROMDA, to assist in code generation, annotations are used in the PIM to give hints as to how certain elements should be transformed. The MDA specification calls these annotations *marks*.

# Chapter 8

# Modelling AJAX with UML

Most user interface modelling for web applications so far has revolved around the concept of pages and navigations [6, 15]. With AJAX, the vital role pages and navigations play in classic web applications is replaced by that of individual components (on a single page) and the interactions that occur between them. We could therefore say that components are the most prominent artifacts in the user interface of AJAX web applications.

Another point to consider when creating an AJAX metamodel is that from a *functionality* point of view, AJAX user interfaces closely resemble the user interfaces of desktop applications. In both cases, the user interface consists of a tree of containers and components which contain, interact or influence one another. The style of user interaction based on events and listeners is also very much alike in both types. The main difference that comes to mind is that when using AJAX, user interface components could be chosen to be served on the client side or the server side, which is not an issue with desktop applications. This difference is however very much dependent on what the AJAX framework we intend to use offers in terms of components and whether they will be served from the server or the client. We shall therefore not take this into account in our metamodel.

Awareness of the aforementioned similarities will make it conceptually easier to devise a metamodel which captures the essence of AJAX user interfaces since we are already familiar with this pattern in desktop user interface toolkits such as Java Swing. The success of Java-based AJAX toolkits such as Echo2[1] and GWT[2] can be attributed to this similarity.

Since different AJAX frameworks may possibly offer different components, we have decided to include the components which are found in the most common frameworks in our metamodel. We strive to keep the efforts of adding or removing framework-specific components (often referred to as *widgets*) to the metamodel at a minimum. The following is a list of components we will be dealing with:

**Layout components** serve as containers and are holders of a set of user interface components.

[1] `http://echo.nextapp.com/site/echo2/`
[2] `http://code.google.com/webtoolkit/`

- Tabset: a set of selectable tabs which upon selection render their content viewable.
- Collapsible: the page is vertically divided between segments which contain components. Each segment can collapse and expand to hide and show it's content respectively.
- Menu: a menu containing a set of menu items which display their content when chosen. This way of accessing content is very common in desktop applications.
- Panel layout: The screen is split up between regions which hold the components.

**Custom components** are the fine-grained components in the user interface which form the content of the containers mentioned above.

- Input components through which user input is received. These are: text field, text area, radio buttons, check boxes, drop down menus and lists.
- Partial submit components can send (submit) data entered by the user to the server directly as data is entered, without having to wait until the user submits a form. This feature is an enhancement to the ordinary input components.
- Suggest function gives suggestions originating from the server to user input at the same time the user enters data in a manner similar to Google Suggests[3].
- Data tables which display data in a table and at the same time make it editable upon user clicks.

## 8.1 An Ideal AJAX Metamodel

Firstly we present a general metamodel for AJAX user interfaces and subsequently in Chapter 9 enhance the metamodel so that it can be used in combination with ANDROMDA. Figure 8.1 shows a general metamodel we have devised for modelling the user interface of AJAX web applications. This model can be expanded by adding more user interface components when needed.

In order to clarify the process of designing the metamodel, we have made several assumptions regarding the type of AJAX application it should capture. We list these assumptions as follows:

- A web applications can be divided in several main parts with each part (use case) responsible for a certain task.
- Each of the above parts can be composed of one or more *view*s. A *view* is a holder of one or more user interface components. The views are not displayed simultaneously but together fulfil the task in the parent container sequentially.

[3] `http://www.google.com/webhp?complete=1`

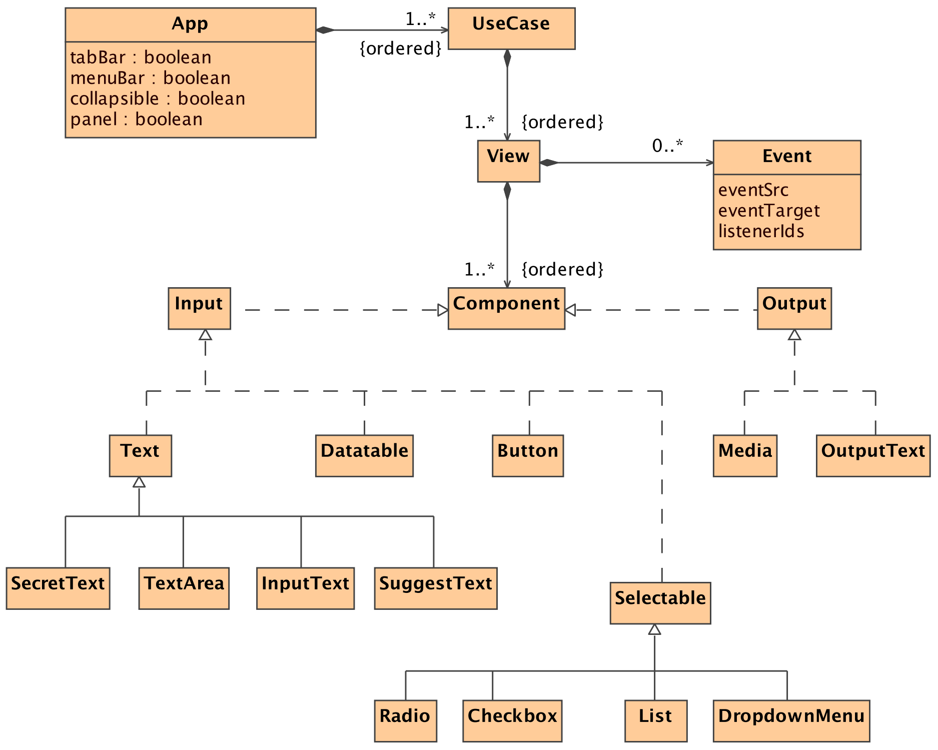


Figure 8.1: A general AJAX metamodel

- The views themselves are composed of one or more components which represent the contents of the views.
- The fine-grained components should have the option to asynchronously send or receive data to the server where this is applicable (not visible in Figure 8.1).

The above assumptions draw clear lines as to what we expect of an instance of our metamodel to do. This may be restricting but makes the initial task of code generation more manageable. The assumptions could be expanded to cover more cases in a later stage.

Below we discuss the different aspects of the metamodel.

## 8.1.1 Containers and Navigation

The navigation aspect of the web applications is defined in the `App` metatype of the metamodel. As can be seen in the `App` metatype in Figure 8.1, four styles of navigation are available. The navigation type is simply based on the layout components that ICEFACES offers and defines in which way instances of `UseCase` are to be accessed. In the context of the PSM, instances of `UseCase` define a section of the user interface which contains the elements responsible for performing a certain use case and should be navigable through a single-page mechanism. Ideally, information such as which navigation mechanism is to be used should be specified by the developers in the PSM, once it has been generated from the PIM. As mentioned earlier, this is not possible in the version of ANDROMDA we have been working on and therefore *marks* should be defined in the PIM to support a correct and complete generation of the PSM. Marks are pieces of information in the PIM which can be used by the transformation to determine how certain PIM elements should be transformed (see Chapter 3). Nested containers are conceptually possible by allowing an instance of `UseCase` an association to one or more `App` instances but we omit this feature at this time to maintain the simplicity of the implementation and leave this to future work.

### 8.1.2 Views and Components

The core of the user interface which is presented to the user is contained in instances of `View`. As mentioned above, the views are not to be displayed simultaneously and can be used as an ordered sequence. Multiple views are convenient in cases where all the components should be replaced by new components, for example in a large form spanning several parts. Each view can contain a number of ordered user interface components. While the containers are reflected in the `App` instance, the components belong in `View` and either handle input from the user, or display information such as text or images on the page. The order of the components in the model counts for the order they will be presented in the screen.

### 8.1.3 Events and Listeners

The way events are initiated and handled is one of the fundamental differences between AJAX and classic web applications. With AJAX a whole refresh of the page is not required as a consequence of an event. Just as in desktop applications, components which should be involved in an event can be registered as listeners and be notified when the event has occurred. The `Event` metatype is designed to capture this information for components contained in a view. Accordingly each view contains zero or more events. For each event it should be specified which element has initiated it (`srcId`), which element should be displayed as a result (`targetId`), and a list of elements which have been registered as listeners for this event (`listenerIds`).

## 8.2 The PERSONMANAGER Model

We have modelled the PERSONMANAGER web application discussed in Chapter 6 using the proposed AJAX metamodel. The result can be seen in Figure 8.2.

In this PSM, the application use cases are accessible through a tab bar mechanism since the `tabBar` boolean attribute has a value of `true` while the other attributes are set to `false`. We can also see the three use cases of PERSONMANAGER symbolised in the model using the three instances of the `UseCase` metatype. In the case of the `RemovePerson` use case, it is consisted of a view composed of several components which shape the form which the user can fill in, in order to remove a person from the PERSONMANAGER application. In the diagram of Figure 8.2 we can also see an instance of `Event` being associated with the `ReviewPersonForm` view, which captures the event performed on the `SubmitRemoved`

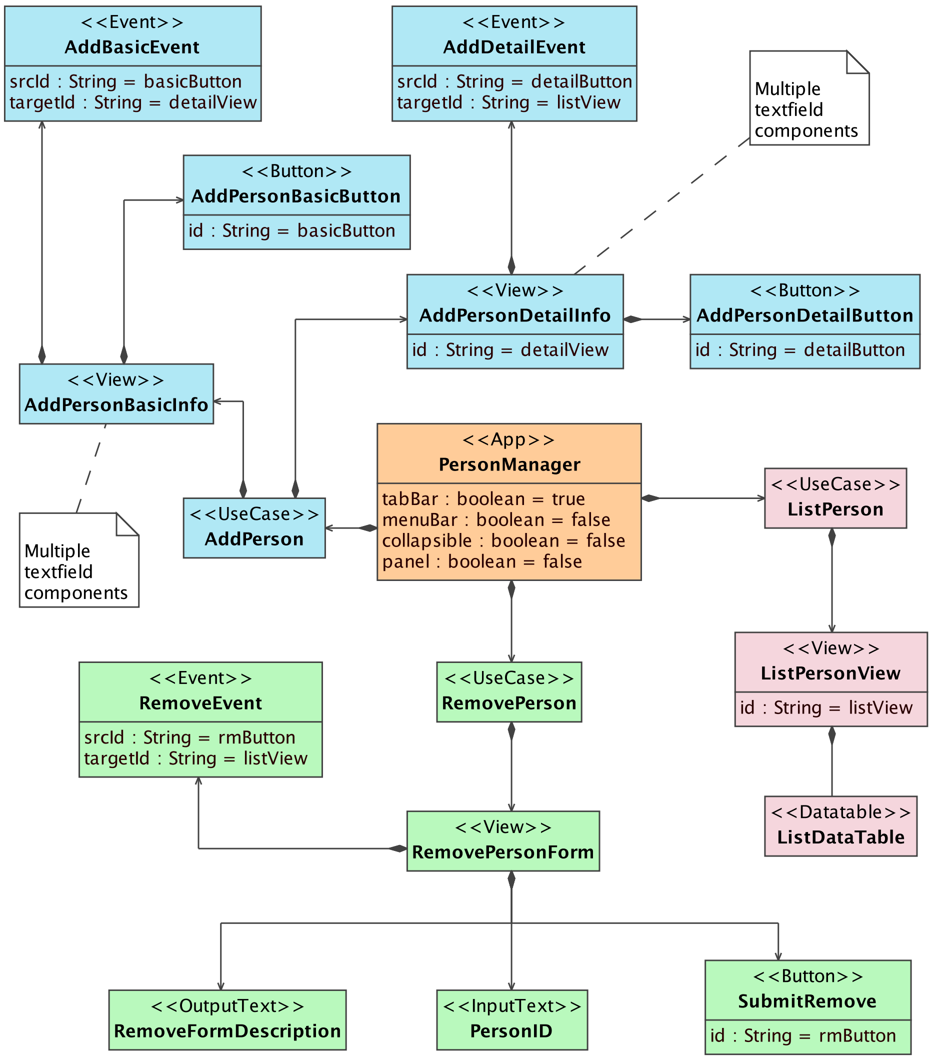


Figure 8.2: PERSONMANAGER modelled using the AJAX metamodel

button. This is reflected in the fact that in `RemoveEvent`, the `srcId` is equal to the ID of `RemoveButton` and `targetId`, which is the ID of the component that should be displayed as a result of this event, is set to `listView`. This means that upon the removal of a person from the web application the user is directed to the `ListPersonPage` instance, which is part of the `ListPerson` use case.

In the `RemovePerson` use case, one view was sufficient to complete the task of removing a person. In the `AddPerson` use case however, more than one view is required to fulfil it's task since the forms for picking up information on a new person span two views. In such a case, each view has it's own set of events and components. The event instance in each page determines the changes on the screen such as which components should be updated or displayed as a result. In the case of the `AddPersonBasicInfo` view, adding the basic information on a person should result in `AddPersonDetailInfo` to be displayed. In the second view, clicking the `AddPersonDetailButton` will render the `listView` visible.

The `ListPersonPage` view's task is to display a list of all the persons in the database, and it does so using a datatable component. A datatable component takes a collection as input and displays all the elements in a table. Datatable is a standard JSF component and ICEFACES has extended it with features such as asynchronous in-line editing, sorting and pagination.

In this chapter we merely discussed a conceptual metamodel for modelling the user interface. In order to use such a metamodel with toolkits such as ANDROMDA however, modifications have to be made which we will discuss as part of the implementation of our proposed AJAX cartridge in Chapter 9.

# Chapter 9

# Implementing the Cartridge

In this chapter, we first introduce a set of non-functional requirements for the web application which will be generated as a result of using the AJAX cartridge. Thereafter, we present the steps that form the workflow of the implementation followed by a detailed discussion of each step.

Since ICEFACES is an implementation of the JSF specification, it is possible to copy many parts of the implementation from the current JSF cartridge of ANDROMDA. This shall be indicated for the relevant segments throughout this chapter.

The implementation process has been carried out using ICEFACES version 1.7.1 and Apache Tomcat[1] version 6.0.16, as web server. Other versions of Apache Tomcat or the web server from JBoss[2] could also be used, however it is important to use correct versions of the library files since incompatibility issues could easily arise.

## 9.1 Reusing the JSF Cartridge

The main reason for choosing ICEFACES for developing the front-end has been to reuse code from the current JSF cartridge of ANDROMDA. After all, ICEFACES is an implementation of JSF and many parts of the existing JSF cartridge implementation can be reused. We have also used the JSF cartridge to learn about the semantics of an ANDROMDA cartridge and the workings of ANDROMDA.

The current JSF cartridge of ANDROMDA heavily utilises an old version of the Oracle ADF Faces[3] component set for many parts of the user interface. In order to be able to reuse parts of the JSF cartridge, we have refactored it by removing the ADF libraries from the cartridge. This refactoring was done since it is not clear what the motivation is for using ADF, and secondly removing the ADF libraries reduces the risk of complications and conflicts arising as a result of using ICEFACES.

Removing Oracle ADF Faces from the cartridge is done in three steps:

- Removing all ADF tags from the front-end templates.

[1] `http://tomcat.apache.org/`

[2] `http://www.jboss.org/jbossweb/`

[3] `http://www.oracle.com/technology/products/adf/adffaces/index.html`

- Removing the ADF properties from the cartridge configuration files (mainly the cartridge and namespace descriptors).
- Removing the use of `AdfFacesContext` throughout the controller classes and using the JSF `FacesContext` instead.

The first step requires going through the front-end template files and replacing any ADF tag with its equivalent in the standard JSF *core* and *html* component tag libraries. An example is replacing `af:table` and its attributes with their equivalent in the *html* tag library, which is `h:dataTable`.

The second step consists of removing any ADF references, declarations and properties from the cartridge and the namespace descriptors since ADF will no longer be used.

The third step is the most complicated, since `AdfFacesContext` is used throughout the cartridge instead of `FacesContext`, possibly[4] because of the ADF *processScope*[5] feature. This feature is unique to ADF and is used to pass form values between pages using a *map* in a new scope in between the *session* and *request* scopes. As a consequence of removing ADF from the cartridge, this feature is also removed and *session maps* are used instead.

## 9.2 Requirements

In Section 8.1, we discussed an ideal metamodel for AJAX user interfaces. The metamodel presented in that section can be used to model a simple AJAX user interface with a specific set of requirements. Below we formalise the requirements for the AJAX cartridge, which were indirectly discussed in the aforementioned section.

**Single-page interface** The web application generated by the cartridge should contain a *single-page interface* [22]. Only components relevant to an event should reload and never the whole page.

**Views** A use case in the web application may consist of one or more views to fulfil its task. The views are sequential, hence no two views in a use case can be displayed simultaneously. During a view change, only the segment of the page related to the view has to be replaced.

**Use-case containers** It should be possible to access use cases in the web application using three different container mechanisms. Developers employing the cartridge can choose which mechanism should be used. These are *tab bar*, *collapsible* and *menu bar*, which have been explained at the beginning of Chapter 8.

**Partial submits** It should be possible for input fields to submit data *partially* [18], that is, asynchronously send data to the server as the user interacts with the component,

[4]We have asked the reason for using ADF in the cartridge in the ANDROMDA forums but unfortunately received no answer. The *processScope* feature may be the reason for the usage.

[5] `http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/doc/devguide/communicatingBetweenPages.html`

without the need to submit a form. This allows for finer granularity interaction and the user interface to promptly react to user input.

**Suggest** For input text fields, it should be possible to give server-side suggestions to the user as the user types in the text field.

We have attempted to retain the following functionality which is offered by current ANDROMDA front-end cartridges :

- Ability to utilise the back-end cartridges such as Spring and Hibernate.
- Completion of one use case should lead to the presentation of the succeeding use case as specified in the activity diagrams.

## 9.3 Workflow

The implementation process will follow the steps below:

1. Since modifying the generated PSM is not possible with ANDROMDA (see Section 7.7), a set of *marks* should be created for the PIM to reflect the AJAX-specific choices.
2. Creating the metafacades:
   - Modelling a complete metamodel with metafacades. This metamodel should contain all the details needed for the implementation.
   - Running ANDROMDA's *meta* cartridge on the metamodel of the previous step. This will generate Java stubs of the modelled metafacades and metatypes.
   - Implementing the generated stubs of the previous step. The methods of these classes (metafacades) represent the transformations.
3. Creating the templates which use the metafacades to generate the source code of the web application.

In the rest of this chapter, a section is dedicated to each of the above steps.

## 9.4 PIM Modification

For AJAX functionality, the following tagged values have been defined for the PIM:

**@andromda.ajax.container.type** Is applied on an *Actor* element in a UML use case diagram. Defines the mechanism through which the different use cases of the web application can be accessed. Current possible values of this tagged value are: *tab*, *collapsible* and *menu*.

**@andromda.ajax.suggest.function** Is applied on a text field attribute exiting a client-side state in an activity diagram. The value of this tagged value should match the name of a controller class method. This method should return a collection of string objects which contain the suggestions that should appear under a text field as a user is typing[6].

**@andromda.ajax.partial.function** Is applied on an attribute exiting a client side state in an activity diagram. The value of the tagged value should correspond with a method in the controller class that serves as the partial submit method for the input field, corresponding with this attribute.

Using these tagged values, enough information is available during the code generation phase to be able to satisfy the last three requirements of Section 9.2. For the first two requirements, variations are not possible hence no further information is required.

## 9.5 AJAX Metamodel for ANDROMDA

In this section we enhance the proposed metamodel of Section 8.1 to work within the ANDROMDA framework. This enhancement is necessary since the previously mentioned conceptual metamodel lacks details that are needed for a complete generation of code. Furthermore, in the previous metamodel we exclusively concentrated on the AJAX user interface and excluded the classes which serve as a glue between the front-end and the back-end of the web application. These will also have to be included in the cartridge metamodel.

For the sake of clarity, we have split the two different aspects of the metamodels mentioned above in two different figures, presented in Figure 9.1 and Figure 9.2.

Figure 9.1 depicts the new AJAX metamodel diagram for ANDROMDA, from a user interface point of view, after having added more properties and methods to be used within the templates. Since the ANDROMDA base metafacades are gateways to the elements of the input model, a cartridge metafacade (in green in Figure 9.1) should extend an ANDROMDA base metafacade (in white) for it to be associated with an element in the input model. In its base metafacades, ANDROMDA incorporates facades to all UML elements which could appear in its UML input models. Below we observe the metafacades of the AJAX cartridge and the base ANDROMDA metafacades[7] they extend.

**AjaxApp** contains the type of use case accessing mechanism for the web application. It extends `ActorFacade`, a base metafacade for an *Actor* element in a use case diagram. In order to determine the type of use case container specified on the *Actor* element, the value of the *@andromda.ajax.container.type* tagged value is retrieved from `ActorFacade` and made available in the form of three boolean attributes. See Figure 9.1

**AjaxUseCase** represents a use case in the PIM, for instance, *Add Person* in Figure 7.4. Extends `FrontEndUseCase` which is the base metafacade to a UML use case. Most

[6] This is a common AJAX feature popularised by Google Suggest: `http://www.google.com/webhp?complete=1`

[7] For the Javadoc API of the base metafacades see: `http://team.andromda.org/docs-3.3/andromda-metafacades/andromda-uml-metafacades/andromda-metafacades-uml/apidocs/index.html`

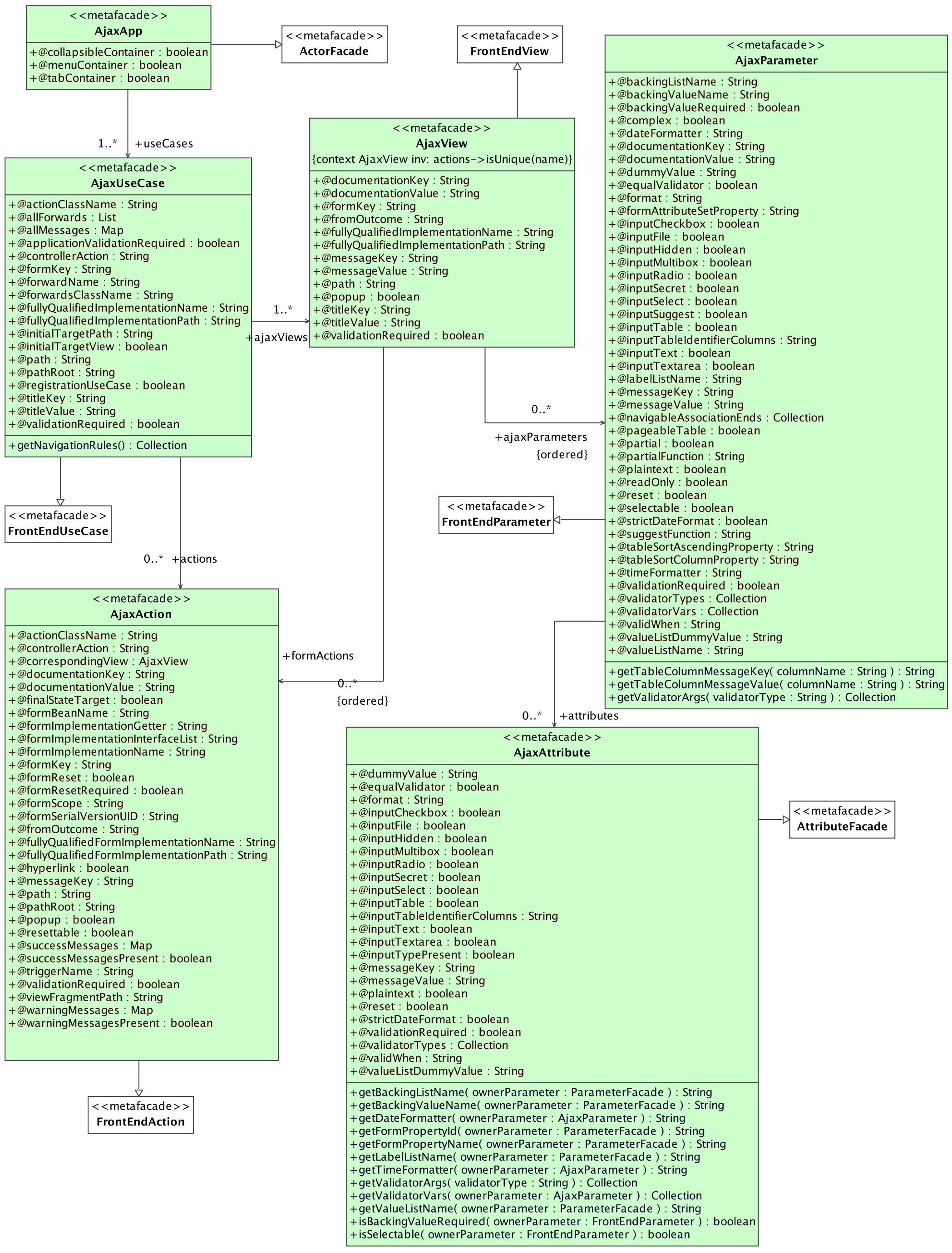
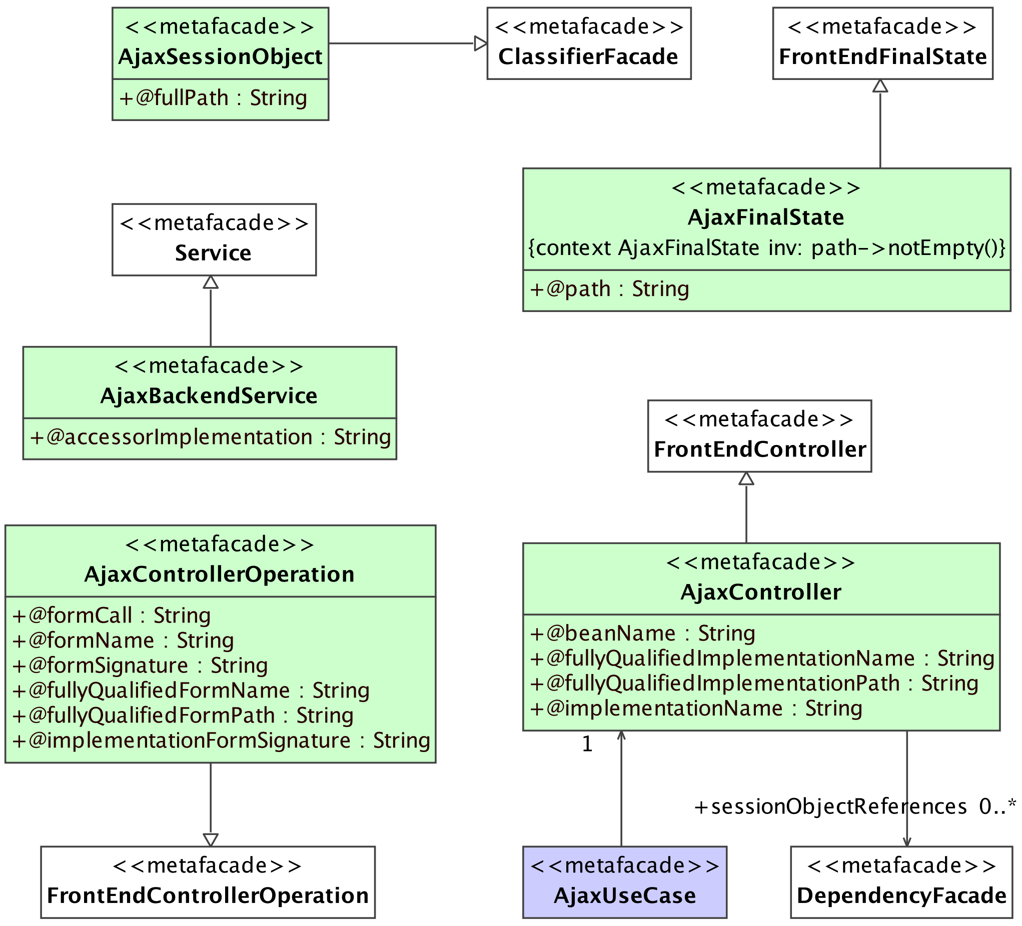

<<metafacade>>
AjaxApp
+@collapsibleContainer : boolean
+@menuContainer : boolean
+@tabContainer : boolean
<<metafacade>>
ActorFacade
<<metafacade>>
FrontEndView
<<metafacade>>
AjaxParameter
+@backingListName : String
+@backingValueName : String
+@backingValueRequired : boolean
+@complex : boolean
+@dateFormatter : String
+@documentationKey : String
+@documentationValue : String
+@dummyValue : String
+@equalValidator : boolean
+@format : String
+@formAttributeSetProperty : String
+@inputCheckbox : boolean
+@inputFile : boolean
+@inputHidden : boolean
+@inputMultibox : boolean
+@inputRadio : boolean
+@inputSecret : boolean
+@inputSelect : boolean
+@inputSuggest : boolean
+@inputTable : boolean
+@inputTableIdentifierColumns : String
+@inputText : boolean
+@inputTextarea : boolean
+@labelListName : String
+@messageKey : String
+@messageValue : String
+@navigableAssociationEnds : Collection
+@pageableTable : boolean
+@partial : boolean
+@partialFunction : String
+@plaintext : boolean
+@readOnly : boolean
+@reset : boolean
+@selectable : boolean
+@strictDateFormat : boolean
+@suggestFunction : String
+@tableSortAscendingProperty : String
+@tableSortColumnProperty : String
+@timeFormatter : String
+@validationRequired : boolean
+@validatorTypes : Collection
+@validatorVars : Collection
+@validWhen : String
+@valueListDummyValue : String
+@valueListName : String
+getTableColumnMessageKey( columnName : String ) : String
+getTableColumnMessageValue( columnName : String ) : String
+getValidatorArgs( validatorType : String ) : Collection
1..*
+useCases
<<metafacade>>
AjaxView
{context AjaxView inv: actions->isUnique(name)}
+@documentationKey : String
+@documentationValue : String
+@formKey : String
+@fromOutcome : String
+@fullyQualifiedImplementationName : String
+@fullyQualifiedImplementationPath : String
+@messageKey : String
+@messageValue : String
+@path : String
+@popup : boolean
+@titleKey : String
+@titleValue : String
+@validationRequired : boolean
<<metafacade>>
AjaxUseCase
+@actionClassName : String
+@allForwards : List
+@allMessages : Map
+@applicationValidationRequired : boolean
+@controllerAction : String
+@formKey : String
+@forwardName : String
+@forwardsClassName : String
+@fullyQualifiedImplementationName : String
+@fullyQualifiedImplementationPath : String
+@initialTargetPath : String
+@initialTargetView : boolean
+@path : String
+@pathRoot : String
+@registrationUseCase : boolean
+@titleKey : String
+@titleValue : String
+@validationRequired : boolean
+getNavigationRules() : Collection
1..*
+ajaxViews
0..*
+ajaxParameters
{ordered}
<<metafacade>>
FrontEndUseCase
<<metafacade>>
FrontEndParameter
0..*
+actions
<<metafacade>>
AjaxAction
+@actionClassName : String
+@controllerAction : String
+@correspondingView : AjaxView
+@documentationKey : String
+@documentationValue : String
+@finalStateTarget : boolean
+@formBeanName : String
+@formImplementationGetter : String
+@formImplementationInterfaceList : String
+@formImplementationName : String
+@formKey : String
+@formReset : boolean
+@formResetRequired : boolean
+@formScope : String
+@formSerialVersionUID : String
+@fromOutcome : String
+@fullyQualifiedFormImplementationName : String
+@fullyQualifiedFormImplementationPath : String
+@hyperlink : boolean
+@messageKey : String
+@path : String
+@pathRoot : String
+@popup : boolean
+@resettable : boolean
+@successMessages : Map
+@successMessagesPresent : boolean
+@triggerName : String
+@validationRequired : boolean
+@viewFragmentPath : String
+@warningMessages : Map
+@warningMessagesPresent : boolean
+formActions
0..*
{ordered}
0..*
+attributes
<<metafacade>>
AjaxAttribute
+@dummyValue : String
+@equalValidator : boolean
+@format : String
+@inputCheckbox : boolean
+@inputFile : boolean
+@inputHidden : boolean
+@inputMultibox : boolean
+@inputRadio : boolean
+@inputSecret : boolean
+@inputSelect : boolean
+@inputTable : boolean
+@inputTableIdentifierColumns : String
+@inputText : boolean
+@inputTextarea : boolean
+@inputTypePresent : boolean
+@messageKey : String
+@messageValue : String
+@plaintext : boolean
+@reset : boolean
+@strictDateFormat : boolean
+@validationRequired : boolean
+@validatorTypes : Collection
+@validWhen : String
+@valueListDummyValue : String
+getBackingListName( ownerParameter : ParameterFacade ) : String
+getBackingValueName( ownerParameter : ParameterFacade ) : String
+getDateFormatter( ownerParameter : AjaxParameter ) : String
+getFormPropertyId( ownerParameter : ParameterFacade ) : String
+getFormPropertyName( ownerParameter : ParameterFacade ) : String
+getLabelListName( ownerParameter : ParameterFacade ) : String
+getTimeFormatter( ownerParameter : AjaxParameter ) : String
+getValidatorArgs( validatorType : String ) : Collection
+getValidatorVars( ownerParameter : AjaxParameter ) : Collection
+getValueListName( ownerParameter : ParameterFacade ) : String
+isBackingValueRequired( ownerParameter : FrontEndParameter ) : boolean
+isSelectable( ownerParameter : FrontEndParameter ) : boolean
<<metafacade>>
AttributeFacade
<<metafacade>>
FrontEndAction

Figure 9.1: AJAX metafacades for ANDROMDA



<<metafacade>> AjaxSessionObject
+@fullPath : String
<<metafacade>> ClassifierFacade
<<metafacade>> FrontEndFinalState
<<metafacade>> AjaxFinalState
{context AjaxFinalState inv: path->notEmpty()}
+@path : String
<<metafacade>> Service
<<metafacade>> AjaxBackendService
+@accessorImplementation : String
<<metafacade>> FrontEndController
<<metafacade>> AjaxControllerOperation
+@formCall : String
+@formName : String
+@formSignature : String
+@fullyQualifiedFormName : String
+@fullyQualifiedFormPath : String
+@implementationFormSignature : String
<<metafacade>> AjaxController
+@beanName : String
+@fullyQualifiedImplementationName : String
+@fullyQualifiedImplementationPath : String
+@implementationName : String
1
+sessionObjectReferences 0..*
<<metafacade>> FrontEndControllerOperation
<<metafacade>> AjaxUseCase
<<metafacade>> DependencyFacade

Figure 9.2: AJAX metafacades for connecting with the back-end

of the implementation has been taken from the JSF cartridge. Some key methods inherited from `FrontEndUseCase` include:

- `getController` returns the corresponding controller for this use case.
- `getActions` returns the actions in this use case, such as form submits and server-side state methods.

`AjaxUseCase` adds the following methods and attributes:

- `getNavigationRules` returns the set of transitions in the activity diagram of this use case.
- Attributes which can be used to represent this use case in the template files such as the use case's *title* and *implementation path*.
- `getAjaxViews` returns the client-side states (views) in this use case.

**AjaxView** represents a client-side state in an activity diagram, for instance, *Add Person Form Basic Info* in Figure 7.5. A large part of the metafacade has been reused from the JSF cartridge. Extends `FrontEndView` and inherits the following methods:

- `getActions` similar to the method with the same name in `FrontEndUseCase`, but only returns the actions for this client-side state.
- `getVariables` returns a set of variables entering this client-side state, which means they have to be rendered on the page.

`AjaxView` adds the following constraint and attributes:

- An OCL constraint which causes an error during the code generation process, if the there are multiple actions for a client-side state which share the same name, preventing conflicts in the configuration file of the target web application.
- A set of properties for this view used in the template files.
- `getAjaxParameters` returns the collection of parameters that are rendered on this view.

**AjaxParameter** represents a parameter modelled on a transition in an activity diagram. This parameter could be either on a transition entering a client-side state or exiting it. In the former case an input field should be rendered on the client-side state representing the parameter and in the latter case it should be rendered as output on the client-side state. This metafacade extends `FrontEndParameter` therefore inheriting the following methods:

- `isTable` indicates whether this parameter has to be rendered on its corresponding client-side state as a table.
- `getTableColumnNames` in case the parameter is a table, this method returns the name of the columns.
- `getAction` returns the action to which this parameter corresponds.
- `getView` returns the view this parameter should either be picked up from, or rendered on.

`AjaxParameter` adds the following properties and methods:

- A boolean attribute for every possible input type on the front-end used to determine what type of input field should be rendered. In the metamodel of Figure 8.1, we modelled this differently by assigning a class to each possible *component*. Since the properties are mostly similar, we use the `AjaxParameter` metafacade to represent all component types. This integration into one class greatly reduces implementation complexity and maintenance.
- A string `partialFunction` containing the name of the controller method used to submit partial data for this parameter. If this parameter is not to submit data partially, this string is left empty.

- A string `suggestFunction` containing the name of the controller method used to pick up suggestions for a text field.

**AjaxAction** represents an action such as a form submit and server-side state methods. Extends `FrontEndAction` and has been reused from the JSF cartridge except for a new method used to pick up the `AjaxView` which corresponds to this action.

The metafacades included in Figure 9.2 are related to the back-end compatibility and have been taken from the JSF cartridge without any modification. Here we briefly explain their functions:

**AjaxBackendService** is used to retrieve the accessor of the *service* class instance, used in the *controller* classes to communicate with the middle-tier cartridges. Accessor mechanisms are available for the Enterprise JavaBeans (EJB) and JSF cartridges.

**AjaxController and AjaxControllerOperation** Are used in the generation of the *controller* classes. The former represents a controller class and the latter represents the operations of that class. Through the association to `DependencyFacade`, modelled *session objects* as in `AddSession` of Figure 7.6.

**AjaxFinalState** Is used for determining which use case has to start as a result of one ending.

**AjaxSessionObject** Represents a *session object* as in `AddSession` of Figure 7.6. Can be used to store intermediate information before submitting to the database. It extends `ClassifierFacade`, the most high-level base metafacade for classifier elements.

## 9.6 Templates

The templates are responsible for generating the final code for the web application. They consist of a set of Apache Velocity template files which are mapped to the metafacades representing elements of interest in the input UML model. For instance, the template file responsible for generating the code for the input fields of a web page could be mapped to `AjaxParameter`, since input fields are characterised by *parameter*s in the input models. In this example, `AjaxParameter` extends `FrontEndParameter` which is the facade to *parameter* elements found in the models.

The generated front-end source codes use FACELETS[8], which is a view technology for JSF. FACELETS offers a great templating system which makes the development of the front-end pages more manageable, but its main advantage and the essential reasons for its usage in the cartridge is the dynamic file inclusion mechanism it offers. Using this mechanism of FACELETS, it has been possible to update the *view* segment of the use cases asynchronously, a feature which ICEFACES does not offer.

Below we present a list of the template files of the AJAX cartridge, explaining their function and their corresponding metafacade.

[8] `https://facelets.dev.java.net/`

ICEFACES.

**layout.xhtml.vsl** This template generates a FACELETS layout template which defines the layout of the main (single) page of the web application. The generated file is purely of interest to FACELETS and is not needed for any AJAX functionality.

**web.xml.vsl** Generates the deployment descriptor of the JSF web application. The contents of the generated file are static and contain configurations for ICEFACES and FACELETS.

**faces-config.xml.vsl** Taken from the JSF cartridge. Generates *faces-config.xml*, a JSF configuration file. Instantiates the backing bean objects for the front-end forms using the `AjaxAction` metafacade. Furthermore, the *controller* classes (one for each use case) are instantiated.

**Form.java.vsl and FormImpl.java.vsl** generate the backing beans for the front-end forms. *From.java.vsl* generates a Java interface and *FormImpl.java.vsl* generates the implementation of that interface. This interface-implementation pair is generated for each client-side state in the activity diagrams and have as attributes the parameters on the transition exiting that state. If a parameter consists of a selectable collection of values, holders for these values are also generated in the form of array properties. For the *Add Person Form Basic Info* view of Figure 7.5, a class is generated with variables for each of the parameters which have to be rendered on the view as input fields. For the `country` parameter, an additional array variable is generated which holds the values of the items in the list rendered for this parameter. We have reused the *Form.java.vsl* and *FormImpl.java.vsl* templates from the JSF cartridge without any modification.

**index.jsp.vsl** generates the default JavaServer Pages (JSP) page *index.jsp*, which is the starting point of the web application. This page redirects immediately to the ICEFACES/FACELETS root. No metafacade is used in combination with this template since its generated contents is static.

**index-ajax.xhtml.vsl** generates the ICEFACES/FACELETS root mentioned above. The generated file represents the page in the *single-page* application and should give access to the available use cases. An instance of `AjaxApp` is mapped to this template, which represents the *Actor* element of the input use case diagram. Through `AjaxApp`, the value of the *andromda.ajax.container.type* tagged value, which is applied on the *Actor* element of a use case diagram is determined and code is written for the corresponding use case. If no tagged value has been applied on the *Actor* element, the default value of *tab* is used and a tab container is generated for the use cases. The other metafacade used within this template is `AjaxUseCase`, which is needed to access information on the use cases in order to write the use case containers.

Each use case container includes the current view using a FACELETS *ui:include* tag, which is able to include another source file dynamically. Changing the name of the included file causes the contents of the new file to appear in the user interface. Which *view* should be shown in the individual use cases is determined through the *controller* classes corresponding with the use cases, which are also accessed through the `AjaxUseCase` metafacades. These are discussed below. Figure 9.3 contains the Apache Velocity template segment used for writing the container mechanism.

**Controller.java.vsl and ControllerImpl.java.vsl** Generate the controller classes which are a pair of an abstract class (*Controller.java.vsl*) and an extension (*ControllerImpl.java.vsl*). The extending class is generated once and placed in the *src* directory, hence it will not be overwritten in subsequent runs. It contains stubs for the methods of the input model controller classes, which have to be implemented by hand. In Figure 7.8 we presented an example of the generated file after having implemented the empty methods for *AddController*. The `AjaxController` metafacade is used in these templates.

The main functions of the generated abstract class are twofold:

- Contains a method that returns the name of the current *view* which has to occupy the view portion of the parent use case. The initial view is the first client-side state of the activity diagram. After subsequent actions, the variable holding the page name is updated with the name of the next view which has to be displayed.
- Contains action methods for the forms, that is, methods which are executed as a result of the user submitting a form. These methods pass on an instance of the form to the methods of the extending class (*ControllerImpl.java.vsl*). The name of the next view to be rendered is set in these action methods, using an instance of `AjaxAction`, since the target view of the the action that represents the form submit is the view that has to be displayed.

**view.xhtml.vsl** Uses the `AjaxView`, `AjaxAction` and `AjaxParameter` metafacades to generate code for the views. The parameters of a view are iterated and depending on the input type, the corresponding ICEFACES tags are written. We have refactored this template from the JSF cartridge so that it uses ICEFACES instead of Oracle ADF tags which are used throughout the JSF cartridge.

**view-table.xhtml.vsl** For parameters entering a client-side state, renders a table displaying their contents. Uses instance of `AjaxParameter` to achieve this. Similar to *view.xhtml.vsl*, this template has been refactored from the JSF cartridge.

**UseCaseForwards.java.vsl** Uses the `AjaxUseCase` metafacade to create name-path mappings for the views. Used by *Controller.java.vsl* to retrieve the path of the current page of a use case. Has been reused from the JSF cartridge with minor adjustments.

**messages.properties.vsl** Uses instances of `AjaxUseCase` to iterate through all the elements of the model and create key-value properties for the messages which have to be rendered in the front-end. For instance, for the *firstName* parameter the `first.name=First Name` key-value pair is generated which is picked up everywhere a label has to be rendered for the aforementioned parameter. Another empty messages properties file is created in the *src* directory which can be used to override the key-value pairs of the generated properties file. This template has also been reused from the JSF cartridge.

```
#if($ajaxApp.tabContainer)
  <ice:panelTabSet id="icepnltabset" styleClass="componentPanelTabSetLayout">
  #foreach($useCase in $useCases)
    #set ($currentPage = "${useCase.controller.beanName}.page")
    <ice:panelTab label="$useCase.name">
      <ui:include src="#{$currentPage}" />
    </ice:panelTab>
  #end
  </ice:panelTabSet>
#elseif($ajaxApp.collapsibleContainer)
  <ice:panelGroup>
  #foreach($useCase in $useCases)
    #set ($currentPage = "${useCase.controller.beanName}.page")
    <ice:panelCollapsible styleClass="navPnlClpsbl" expanded="true" >
      <f:facet name="header">
        <ice:panelGroup>
          <ice:outputText value="$useCase.name"/>
        </ice:panelGroup>
      </f:facet>
      <ice:panelGroup>
        <ui:include src="#{$currentPage}" />
      </ice:panelGroup>
    </ice:panelCollapsible>
  #end
  </ice:panelGroup>
#else
```

Figure 9.3: Apache Velocity template for the use case container mechanism.

Chapter 10

# Generated AJAX Application

The ability to use a platform-independent metamodel to model user interfaces has been an important goal of this project. This has been a factor in choosing UML and ANDROMDA: to be able to use the same UML models to generate code for different platforms. As a result of this, the *same* input UML models for an ANDROMDA web application can be used to generate code for the JSF, Struts and the AJAX cartridge. For AJAX we have introduced three marks in the PIM, which are used to determine the type of the container to be used, the name of a *partial* function for input components and the name of a *suggest* function for text field components, respectively. The platform-specific tagged values are safely ignored when a model is used by other cartridges. In this chapter, we report on the AJAX web application generated using the AJAX cartridge.

## 10.1 Use Case Containers

Figure 10.1 depicts an ANDROMDA use case diagram in which the tagged value for the container type is assigned to *Collapsible*.

Generating the application with the *Collabsible* tagged value results in the screenshot

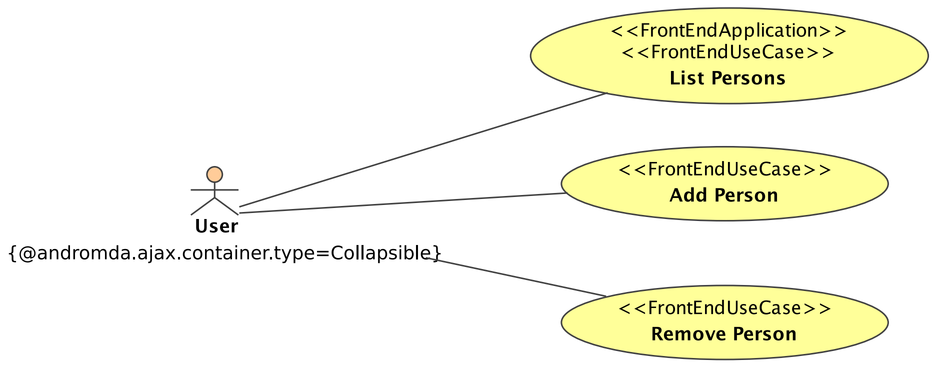
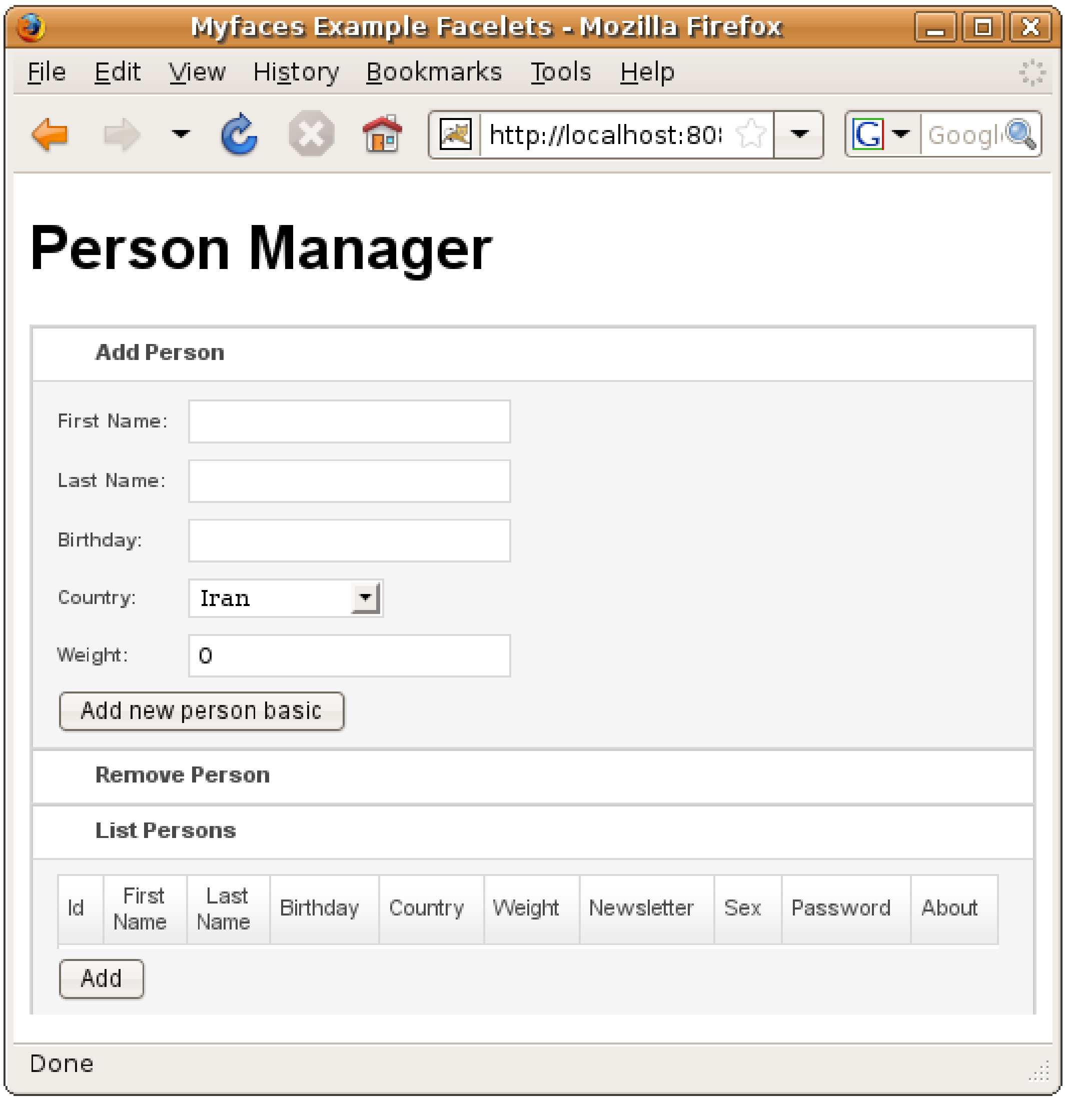


Figure 10.1: ANDROMDA use case diagram with *collapsible* tagged value


Myfaces Example Facelets - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:80
Google

Person Manager

Add Person

|  |  |
|---|---|
| First Name: |  |
| Last Name: |  |
| Birthday: |  |
| Country: | Iran |
| Weight: | 0 |

Add new person basic

Remove Person

List Persons

| Id | First Name | Last Name | Birthday | Country | Weight | Newsletter | Sex | Password | About |
|---|---|---|---|---|---|---|---|---|---|

Add

Done

Figure 10.2: PERSONMANAGER displayed using a collapsible use case container

shown in Figure 10.2. In this figure, the *Remove Person* use case header has been clicked to make its contents invisible. Changing the tagged value to *Tab* and regenerating the web application results in the screen depicted previously in Figure 6.1. Clicking on the header of a use case in these figures renders its contents, should they not already be selected. Besides *Tab* and *Collapsible*, a *Menu* value is also possible. At this time, the menu bar is rendered but the mechanism to display the corresponding contents on the screen has not yet been completed.

## 10.2 Views

Views form the contents of the use cases. In Figures 6.1 and 6.2 two views of the *add person* use cases are presented when implemented using ICEFACES. A requirement of the cartridge has been to switch the views in the context of the use cases, that is, the corresponding segment inside of use case should be the sole segment which is replaced. This is achieved with the AJAX cartridge and in the case of the *add person* use case, the *basic* page is replaced with the *detail* page as specified.

## 10.3 Components

The *partial* and *suggest* tagged values are applicable to the fine-grain input components. The *suggest* tagged value can only be applied on an input text component. At this point, the corresponding functions are generated in the *controller* classes but specifying them in the front-end and the mapping between the two has not been implemented yet. We believe that the *suggest* function should be realisable without complications since linking a front-end input suggest element to a controller function which returns a list of values should display the values under the text field. The *partial* feature however can be more complicated since the possible uses of such a component are twofold. One often used case for a partial submit component is to have it modify another component on a form. For instance, a drop-down menu containing a list of countries can communicate the chosen country to the server using a *partial* function, and have the contents of another drop-down menu with a list of provinces be adjusted to match the chosen country. Such a case should be realisable with the model provided by the AJAX cartridge, since an object representing the *form* on the user interfaces is available to the functions of the controller classes. However other cases such as communicating a value received through a partial submit to the databases can become more complicated and we have not taken such cases into account in the cartridge.

## 10.4 Back-end Compatibility

The beck-end compatibility which runs through the controller classes is functionality-wise the same as that of the JSF cartridge. An instance of the *service* instance is available to the controllers, which provides a facade to the persistence layer. We have used a Spring-Hibernate configuration for the backend, but only Hibernate or EJB-Hibernate should also be possible, though this has not been tested.

## 10.5 Generated Source Code Samples

We include two generated source code segments from the PERSONMANAGER example. Figure 10.3 contains a generated *Collapsible* use case including mechanism for the PERSONMANAGER example. The template of this generated file was given in Figure 9.3. The second generated file we include is the *view* containing the components for the *add person basic info* phase of PERSONMANAGER. This segment is presented in Figure 10.4.

```
<ui:define name="body">
  <ice:form partialSubmit="true" id="iceform">
    <ice:panelGroup>
      <ice:panelCollapsible styleClass="navPnlClpsbl" expanded="true" >
        <f:facet name="header">
          <ice:panelGroup>
            <ice:outputText value="Add␣Person"/>
          </ice:panelGroup>
        </f:facet>
        <ice:panelGroup>
          <ui:include src="#{addController.page}" />
        </ice:panelGroup>
      </ice:panelCollapsible>
      <ice:panelCollapsible styleClass="navPnlClpsbl" expanded="true" >
        <f:facet name="header">
          <ice:panelGroup>
            <ice:outputText value="Remove␣Person"/>
          </ice:panelGroup>
        </f:facet>
        <ice:panelGroup>
          <ui:include src="#{removeController.page}" />
        </ice:panelGroup>
      </ice:panelCollapsible>
      <ice:panelCollapsible styleClass="navPnlClpsbl" expanded="true" >
        <f:facet name="header">
          <ice:panelGroup>
            <ice:outputText value="List␣Persons"/>
          </ice:panelGroup>
        </f:facet>
        <ice:panelGroup>
          <ui:include src="#{listController.page}" />
        </ice:panelGroup>
      </ice:panelCollapsible>
    </ice:panelGroup>
  </ice:form>
</ui:define>
```

Figure 10.3: Generate collapsible index file containing for PERSONMANAGER

```
<ice:panelGrid columns="2">
  <ice:outputLabel value="#{messages['first.name']}:"
                   for="firstName"/>

  <ice:inputText id="firstName"
                 value="#{addPersonBasicForm.firstName}"
                 required="false"
                 readOnly="false"/>

  <ice:outputLabel value="#{messages['last.name']}:" for="lastName"/>

  <ice:inputText id="lastName"
                 value="#{addPersonBasicForm.lastName}"
                 required="false"
                 readOnly="false"/>

  <ice:outputLabel value="#{messages['birthday']}:"
                   for="birthday"/>

  <ice:inputText id="birthday"
                 value="#{addPersonBasicForm.birthday}"
                 required="false"
                 readOnly="false"/>

  <ice:outputLabel value="#{messages['country']}:" for="country"/>

  <ice:selectOneMenu id="country"
                     value="#{addPersonBasicForm.country}"
                     required="false"
                     readOnly="false"
                     partialSubmit="true"
                     valueChangeListener="$parameter.partialSubmit">
    <c:if test="${!empty␣addPersonBasicForm.countryBackingList}">
      <f:selectItems
        value="#{addPersonBasicForm.countryBackingList}" />
    </c:if>
  </ice:selectOneMenu>

  <ice:outputLabel value="#{messages['weight']}:" for="weight"/>

  <ice:inputText id="weight"
                 value="#{addPersonBasicForm.weight}"
                 required="false"
                 readOnly="false"/>

  <f:facet name="footer">
    <ice:commandButton value="#{messages['add.new.person.basic']}"
                 action="#{addController.addPersonFormBasicInfoAddNewPersonBasic}"
                 type="submit"/>
  </f:facet>
</ice:panelGrid>
```

Figure 10.4: Generate *add person basic info* view of PERSONMANAGER

# Chapter 11

# Evaluation and Discussion

In this chapter we first reflect on the research questions that were defined in Chapter 4. Thereafter, we give an evaluation and give an assessment of our developed AJAX cartridge, ANDROMDA as an MDA toolkit and ICEFACES as an AJAX framework.

## 11.1 Research Questions

In Chapter 4 we defined a set of research questions. We discuss these below:

**RQ 1: How to model an AJAX web application** We have presented a UML metamodel in Chapter 8 using which AJAX user interfaces can be defined. The metamodel is inspired by the close similarity we have perceived between AJAX and desktop user interfaces. Figure 8.1 depicts this metamodel which captures a user interface consisting of a tree of containers and components. Each component in the tree can have an association with an event, which can affect a set of listeners.

In more general terms, we believe the AJAX metamodel has the potential to cover the complete user interface of an AJAX web application, as long as the analogy between desktop and AJAX user interfaces holds. The main question here is: what is the amount of user interface related detail we want to capture in an AJAX model. A comprehensive amount of user interface detail in a model may easily derive the model of its simplicity, whilst model simplicity is an important factor to choose MDA in the first place. In the context of this project we consider enriching a *generated* PSM based on our proposed metamodel as feasible and reasonable in a real setting. However, expanding the PIM drastically, counteracts the advantages of a model-driven approach and is practically not a feasible solution.

**RQ 2: How suited is the modelling language for modelling AJAX user interfaces** We have used UML as the modelling language for the metamodels, partly due to ANDROMDA using UML, and in part because of advantages regarding the desired platform-independence of the models (see Chapter 5). While the platform-independent metamodel of ANDROMDA for user-interfaces is well suited for defining navigations and forms, fine-grain detail such as positioning and layout are more difficult to capture. UML extensions such as tagged values and stereotypes are used to gather sufficient information to generate a complete user interface and only leave the business logic to be programmed by hand. For layout and positioning, either the generated code should be customised, or the templates which are responsible for code-generation. With respect to the AJAX metamodel, we believe it can be expanded to cover many real-world cases of AJAX web applications.

**RQ 3: Regarding the degree of AJAX in the generated web application, Are we satisfied?** In the developed cartridge, our main AJAX goal is generating a *single-page* interface, implying that the page is never reloaded as a result of an event. As regards to the contents of that single-page, the cartridge is not capable of producing what the potential of an AJAX user interface is. This can be attributed to the metamodel and the level of detail currently contained in it. As for satisfiability, a great deal depends on the requirements of the intended application. In some plausible cases, what is currently offered by the cartridge can be sufficient.

**RQ 4: How do we provide a mapping between a PIM and an AJAX PSM** In this project we have used our metamodel in combination with the PIMs already offered by ANDROMDA. The ANDROMDA PIM is structured using a set of use cases, with each use case containing a set of views. Each view consists of a set of parameters. The corresponding cartridge metamodel elements are `AjaxUseCase`, `AjaxView` and `AjaxParameter`. The mapping between the elements of the PIM and PSM are therefore defined by one-to-one associations. The three AJAX functions which have been introduced are applied on PIMs elements as tagged values, which is the subject of the next research question.

**RQ 5: How do we propagate AJAX choices from the PIM** Three tagged values defined in the PIM are responsible for conveying the AJAX features specified by the user to the generated application. One of these tagged values is applied on an *Actor* element in a use case diagram. This tagged value defines the way the user browses through the use cases in the single-page interface. The other two define the *suggest* and *partial* functions of specific components on the page. These tagged values are ignored when the model is used in conjunction with a cartridge other than the AJAX cartridge.

## 11.2 Implementation Challenges and Contributions

We provide a short summary of our contributions, and the challenges we came across from an implementational point of view.

- A challenging task has been removing ADF from the JSF cartridge without breaking it. The result is a JSF cartridge which is free of any external libraries and can therefore form a basis for many other JSF implementations.

- Reusing the JSF cartridge as much as possible while keeping the metamodel of the AJAX cartridge intact has not always been easy. An example in which we have slightly deviated from our proposed metamodel is the association between `AjaxUseCase` and `AjaxAction` in Figure 9.1, which is nonexistent in the original metamodel. This can be refactored so that the all instances of `AjaxAction` are retrieved through `AjaxView`.

- Implementing the view changing mechanism inside a use case was not trivial, until we discovered the dynamic include feature of FACELETS and thought of the solution to retrieve the current view of a particular use case through its corresponding controller class.
- In order to fulfil its single-page promise, the AJAX cartridge includes the use cases in a single page, which themselves dynamically include the corresponding views. Through the inclusions, it is not possible to create *servlet mappings* or apply *filters* to separate URLs, since there is only one URL, namely that of the single-page. This is very different in the JSF cartridge since the URL changes for each use case, and as a result, URLs are mapped to servlets which handle the initialisation. To solve this, a JSF method binding expression is used in the *index-ajax.xhtml.vsl* template, which initialises the views as well as returning the name of the current page. Hence using *servlet mappings* and *filters* is avoided.

## 11.3 Evaluation of the AJAX Cartridge

### 11.3.1 Limitations

We believe that our cartridge implementation currently has the following limitations:

**Specific user interface** The structure of the user interface the cartridge provides at this point is very specific and limited. Currently, only one type of web application can be modelled and generated, namely a web application that offers a set of use cases whereby each use case consists of a set of sequential views. Hence, at any given time it is possible to display only one use case containing a single view. For an AJAX application, many user interface possibilities are imaginable. The AJAX metamodel presented in Chapter 8 is based on the analogy between desktop user interfaces and AJAX. Using a desktop approach (e.g. Java Swing), components can be precisely positioned and employing layout managers[1] is a possibility, which the AJAX cartridge does not offer at this moment. Moreover, the event-based mechanism is only specified for the *text suggest* and *partial* input components. In an ideal case this is expanded so every input component can be associated with an event, as specified in the ideal metamodel of Chapter 8.

**Testing and production-readiness** Current ANDROMDA cartridges use CodeUnit[2], a JUnit extension for cartridge testing. CodeUnit can be used to compare two APIs or source files at the Abstract Syntax Tree (AST) level. Hence it is used to compare generated output of the cartridge for a given model, with output expected of that

[1] `http://java.sun.com/docs/books/tutorial/uiswing/layout/using.html`
[2] `http://xdoclet.sourceforge.net/xdoclet/tools.html`

model. Describing a test set using CodeUnit or another test framework is a necessity for the AJAX cartridge, should it be expanded and used for production environments.

### 11.3.2 Potentials

The cartridges provided by ANDROMDA, or code generated by other MDA frameworks involving applications with (web) user interfaces, will almost always need adjustments when used for a specific product. In ANDROMDA adjustments to the generated user interface code can be made through CSS or manual alteration of the generated output. Another less trivial option however is to modify the templates or the transformation in order to achieve the required results. For generating AJAX web applications, the AJAX cartridge forms a good basis for further specialisation. By customising the cartridge and adding more AJAX components which are desired, the cartridge can prove very useful. Creating or customising a cartridge for ANDROMDA is however not a trivial task, which we further discuss below.

## 11.4 ANDROMDA

Even though embodying a novel approach to MDA and modelling user interfaces, ANDROMDA suffers from several shortcomings:

**Steep learning curve** A major weakness of ANDROMDA is the shortage of documentation, making it a difficult tool to use for developers who want to employ it for generating a web application, or for those who want to build a cartridge for it. Although documentation exists, it is often incomplete, outdated or scattered where it might not be found easily. In the case of the JSF cartridge, simply, no documentation exists. As a modelling guide for the JSF cartridge, the documentation of the Struts cartridge can be used, which is not coincidental as the input models have to be platform-independent. The implementation of the cartridge itself however is not documented at all and the guide featured on the website for creating a new cartridge is incomplete. This leaves going through the source-code as the only option to gain an understanding of the framework, which is what we had to do. A forum is available which can be used for support, however receiving an answer to a question posted is not guaranteed.

**Modelling tool** The UML modelling tool almost exclusively used by the developers of ANDROMDA and its users is MagicDraw[3]. MagicDraw is very complete in terms of the amount of functionality it offers and in terms of implementing the UML standard. The tool is commercial but a community edition can be used with limited support. The disadvantage to MagicDraw being actively used by the developers is that other modelling tools are to some extent neglected and the problems that may occur with those tools are not documented. There is a considerable amount of support for Ar-

[3] `http://www.magicdraw.com/`

goUML[4] which is an open-source CASE tool but in general for serious development with ANDROMDA MagicDraw is the recommended UML tool.

**Development** ANDROMDA development is at this moment not very active[5]. A new version of ANDROMDA has been started but development seems to be halted for now. As stated on the ANDROMDA website, the new version (4) has a higher level of conformance with the MDA standard, in that it should be able to incorporate multiple levels of model to model transformations with each level being modifiable by the user.

**Platform-Specific Model and Model-Driven AJAX** For the current stable version of ANDROMDA (3.3), the fact that the PSM models are only accessible in memory as bytecode and cannot be tweaked before further generation is a major limitation. The consequence of this restriction for the AJAX cartridge is that any AJAX-specific user interface detail has to be incorporated in the PIM in the form of *marks*. One disadvantage to this is that the number of marks can easily grow and make the PIM more specific than desired. Another disadvantage is that adding *marks* in the PIM may not be as intuitive as enriching a generated PSM model which already has some similarity to the desired code. For instance, specifying *events* in the model is much better done in the example model of Figure 8.2 than in an ANDROMDA PIM, such as the one depicted in Figure 7.5.

Reflecting on our experiences gained during this project, we would recommend using an MDA toolkit wherein generated models are modifiable, and one which is sufficiently documented and used by an active community. At this moment, openArchitectureWare[6] (discussed in Chapter 2) seems to embody these benefits and it would make an interesting study the compare the results of this thesis and a similar approach based on openArchitectureWare.

## 11.5 ICEFACES

Our experience with ICEFACES has been very positive. The development is very active and documentation and examples are adequate. In terms of AJAX functionality, it is one of the most complete AJAX implementations of JSF. The forum is well observed and according to our experiences, main developers as well as users respond to questions in a short amount of time.

[4] `http://argouml.tigris.org/`
[5] `http://galaxy.andromda.org/forum/viewtopic.php?t=5643`
[6] `http://www.openarchitectureware.org/`

# Chapter 12

# Conclusions and Future Work

Model-driven software development is gaining increasing popularity, mainly due to the growing complexity of software systems. A main goal for model-driven development is reducing complexity and maintenance efforts in software development, by abstracting away from code and working at the model level. Numerous frameworks have appeared in recent years to assist in realizing this concept and the OMG has released several standards for model-driven software development under the hood of Model-Driven Architecture (MDA).

In this thesis we have explored an MDA approach for the development of AJAX web applications. Specifically, we attempted an approach wherein there is a clear separation between input models as to their level of platform independence.

As an initial step, we discussed ANDROMDA, an open-source MDA toolkit which is primarily directed towards web application development. We discussed ANDROMDA's structure as well as the form of its required input models.

In Chapter 8, we introduced a platform-specific metamodel for AJAX user interfaces which is based on UML. The metamodel captures a tree of containers, components and their associated events.

Later, we extended the introduced metamodel for it to be usable as part of an AJAX transformation for ANDROMDA. Also, UML tagged values were defined for the platform-independent models of ANDROMDA, in order to convey AJAX-specific meaning to the AJAX cartridge. As for the platform-independent metamodel, the current mechanism of ANDROMDA which is used to model web applications is used.

The result of our implementation work is an AJAX cartridge for ANDROMDA that can be used to generate simple single-page AJAX applications, conforming to a narrow set of AJAX-related requirements. In Chapters 10 and 11 of this thesis, we report on the results of our work in terms of:

- Completeness of the generated web application.
- The choices that were made during the project regarding the MDA toolkit and modelling language.
- Modelling AJAX user interfaces in general.

Possible future work based on this project can exist in one of two possible categories: either improving the current work which is based on ANDROMDA, which has its limitations as explained in Chapter 11; or employing another MDA framework which allows for modifiable generated models.

For the first category, future work can exist in several fields:

- Introducing role-based security to in the front-end. This feature is found in other cartridges but has been left out in the AJAX cartridge in order to focus on the AJAX user interface.
- Completing missing functionality and testing. The partial submit and auto-complete/suggestion features specified in the models are not functional. Testing of the current functionality using CodeUnit.
- Adding more AJAX functionality in the form of components, and at the metamodel level.
- Conducting more case studies besides PERSONMANAGER.

In order to be able to modify a generated model based on the AJAX metamodel, an MDA toolkit other than ANDROMDA should be used. At this moment, the open-source openArchitectureWare appears as an interesting option, however, it does not contain a set of ready to use cartridges to be used for the back-end.

# Bibliography

[1] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: An appliance-independent XML user interface language. In *WWW '99: 8th International Conference on World Wide Web*, pages 1695Ű–1708, 1999.

[2] J. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The unified modeling language user guide*. Addison-Wesley Reading Mass, 1999.

[4] Juan Carlos Preciado, Marino Linaje, Sara Comai, and Fernando Sanchez-Figueroa. Designing Rich Internet Applications with Web engineering methodologies. In *Proceedings of the 9th IEEE International Symposium on Web Site Evolution (WSE'07)*, pages 23–30. IEEE Computer Society, 2007.

[5] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web modeling language (WebML): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157, 2000.

[6] Jim Conallen. *Building Web Applications with UML (2nd Edition)*. Addison-Wesley, 2003.

[7] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[8] A. van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1994.

[10] Jesse Garrett. Ajax: A new approach to web applications. Adaptive path, 2005. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

[11] Vahid Gharavi. Modelling AJAX User Interfaces for the Purpose of Code Generation, 2007. Literature Study.

[12] Vahid Gharavi, Ali Mesbah, and Arie van Deursen. Modelling and Generating Ajax Applications: A Model-Driven Approach. In *7th International Workshop on Web-Oriented Software Technologies (IWWOST'08)*, pages 38–43, July 2008.

[13] Jonathan Joubert. From REST to Rich: Retargeting a DSL to Ajax. Master's thesis, Delft University of Technology, September 2007.

[14] N. Koch and A. Kraus. The expressive power of UML-based web engineering. In *IWWOST '02: 2nd International Workshop on Web-oriented Software Technology*, pages 105Ű–119. CYTED, 2002.

[15] N. Koch and A. Kraus. Towards a Common Metamodel for the Development of Web Applications. In *ICWE '03: International Conference on Web Engineering*, pages 495Ű–506. Springer, 2003.

[16] Nora Koch. Transformation techniques in the model-driven development process of UWE. In *MDWE '06: 2nd Model-Driven Web Engineering Workshop*, page 3. ACM Press, 2006.

[17] F. J. Martìnez-Ruiz, Jaime Munoz Arteaga, Jean Vanderdonckt, and J. M. Gonzàlez-Calleros. A first draft of a model-driven method for designing graphical user interfaces of Rich Internet Applications. In *LA-Web '06: Proceedings of the 4th Latin American Web Congress*, pages 32–38. IEEE Computer Society, 2006.

[18] Stephen Maryka. Enterprise ajax - transcend the hype. `http://www.icefaces.org/docs/whitepapers/Enterprise_Ajax_WP.pdf`, 2006.

[19] Martin Matula. NetBeans Metadata Repository. *NetBeans Community. In: http://mdr.netbeans.org, Accessed in*, 25, 2008.

[20] Ali Mesbah. Ajaxifying classic web applications. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering, Doctoral Symposium*, pages 81–82. IEEE Computer Society, 2007.

[21] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling Ajax by inferring user interface state changes. In *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)*. IEEE Computer Society, 2008.

[22] Ali Mesbah and Arie van Deursen. A component- and push-based architectural style for Ajax applications. *Journal of Systems and Software (JSS)*, 2008. To appear.

[23] Sun Microsystems. Java Metadata Interface (JMI) home. `http://java.sun.com/products/jmi/index.jsp`.

[24] J. Miller, J. Mukerji, et al. MDA Guide Version 1.0.1, 2003. `http://www.omg.org/docs/omg/03-06-01.pdf`.

[25] OMG. MDA, 2008. http://www.omg.org/mda.

[26] Object Management Group (OMG). Meta Object Facility (MOF) Core Specification, 2003. formal/06-01-01, http://www.omg.org/cgi-bin/apps/doc?formal/07-02-03.pdf.

[27] Douglas C. Schmidt. Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[28] Marino Linaje Trigueros, Juan Carlos Preciado, and F. Sánchez-Figueroa. A method for model based design of Rich Internet Application interactive user interfaces. In *ICWE '07: Proceedings of the 7th International Conference Web Engineering*, pages 226–241. Springer, 2007.

[29] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008.

[30] W3C. The XMLHttpRequest Object, 2006. http://www.w3.org/TR/XMLHttpRequest/.

[31] J. Warmer and A. Kleppe. *The object constraint language: precise modeling with UML.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1998.

# Appendix A

# Diagrams

## A.1 Remove Person Activity Diagram

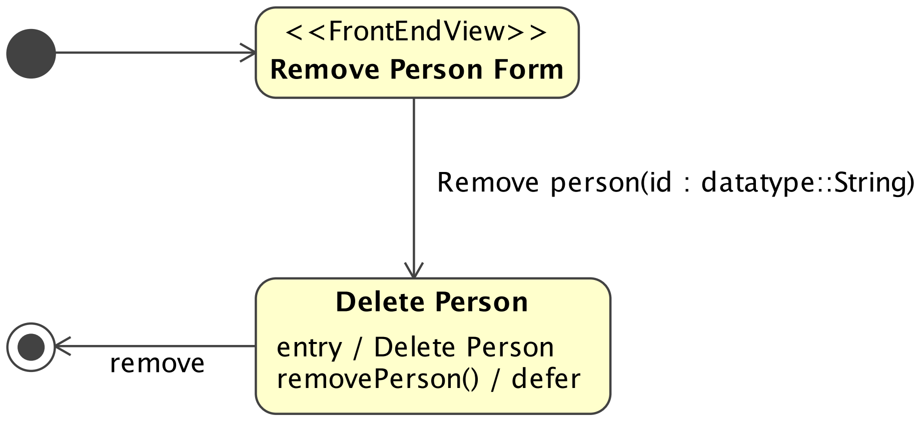


Figure A.1: ANDROMDA Remove Person activity diagram

## A.2 List Person Activity Diagram

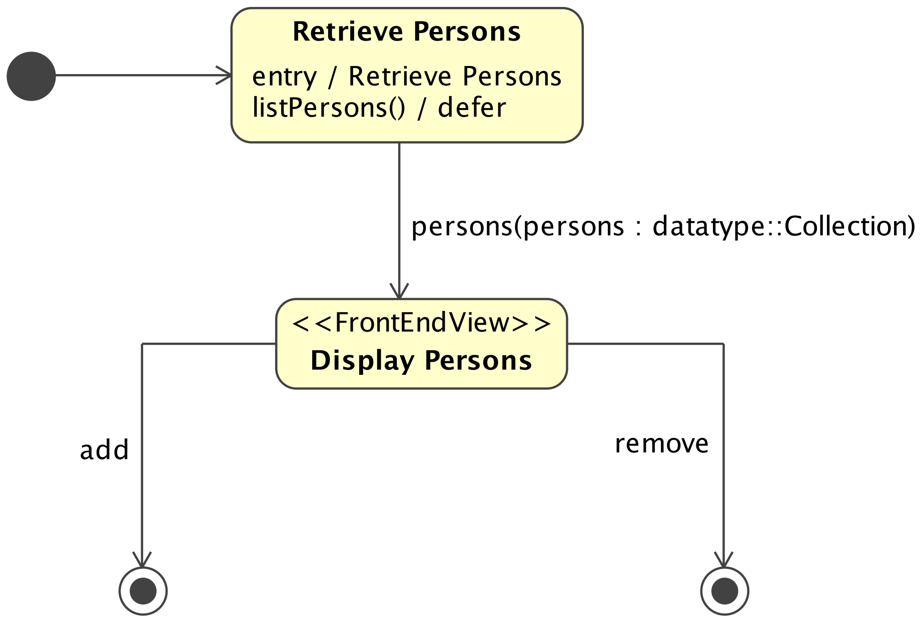


Figure A.2: ANDROMDA List Persons activity diagram

# Appendix B

# Source Codes

## B.1 Remove Person Controller Source Code

```
public class RemoveControllerImpl
    extends RemoveController
{
    public void removePerson(RemovePersonForm form)
    {
        try {
          this.getPersonService().removePerson(form.getId());
        } catch (Exception e){
                e.printStackTrace();
                throw new RuntimeException(e);
        }
    }
}
```

Figure B.1: Remove controller class after manual implementation

## B.2 List Person Controller Source Code

```
public class ListControllerImpl
    extends ListController
{
    public void listPersons(ListPersonsForm form)
    {
        form.setPersons((java.util.List) this.getPersonService().listPersons());
    }
}
```

Figure B.2: List controller class after manual implementation

# Appendix C

# Glossary

| | |
|---|---|
| **API** | Application Programming Interface |
| **CRUD** | Create-Read-Update-Delete |
| **CSS** | Cascading Style Sheets |
| **DOM** | Document Object Model |
| **DSL** | Domain Specific Language |
| **EJB** | Enterprise JavaBeans |
| **GUI** | Graphical User Interface |
| **GWT** | Google Web Toolkit |
| **HTML** | HyperText Markup Language |
| **JMI** | Java Metadata Interface |
| **JDBC** | Java Database Connectivity |
| **JSF** | JavaServer Faces |
| **JSP** | JavaServer Pages |
| **MDA** | Model-Driven Architecture |
| **MOF** | Meta Object Facility |
| **MVC** | Model-View-Controller |
| **OCL** | Object Constraint Language |
| **OMG** | Object Management Group |
| **POM** | Project Object Model |

| | |
|---|---|
| **PIM** | Platform Independent Model |
| **PSM** | Platform Specific Model |
| **SQL** | Structured Query Language |
| **UML** | Unified Modelling Language |
| **UWE** | UML-based Web Engineering |
| **XHTML** | Extensible Hypertext Markup Language |
| **XMI** | XML Metadata Interchange |
| **XML** | Extensible Markup Language |
| **XSLT** | Extensible Stylesheet Language Transformations |