Specification and Generation
of User Interfaces
Based on Method-Oriented Dialog Flows

by
Artun Subasi

in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

at the University of Kaiserslautern
Faculty of Computer Science
May 10, 2011

Thesis Supervisors:
Prof. Dr. Arnd Poetzsch-Heffter
Dr. Gerhard Pews
Dipl.-Inf. Patrick Michel
Abstract

Considerable time and effort is spent on implementing user interfaces and assuring that the implementation conforms to the user interface architecture. Although existing user interface generation approaches can be used to reduce the implementation effort, the generated user interfaces are not tailored to user interface architectures. This thesis shows how dialog flow specifications can be used for the automatic generation of user interfaces with regard to user interface architectures. An abstract user interface specification approach is proposed which enables platform independent specification of dialog flows. In addition, a Java-based user interface specification language is used to investigate whether dialog flows can be practically defined using the elements of the Java language, such as interfaces and annotations.

As a proof of concept, a new software tool is presented which is able to generate web based graphical user interfaces with regard to a concrete user interface architecture which was developed by the “Bundesstelle für Informationstechnik des Bundesverwaltungsamtes” in cooperation with Capgemini sd&m. The evaluation results show that the developed user interface specification and generation method optimizes the workflow for the creation of the user interfaces.
Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Kaiserslautern, May 10, 2011

__________________________
Artun Subasi
Acknowledgements

I would like to express my appreciation to Prof. Dr. Arnd Poetzsch-Heffter and Dr. Gerhard Pews for their efforts in the organization of this thesis, and for the insightful conversations to set the scope of the work.

My utmost gratitude goes to Patrick Michel for his very helpful comments on the text, for his great advices on the structuring of the work, and for his fast responses to my e-mails, even on weekends.

I would like to thank Andy Scherzinger and Simon Spielmann for their continuous feedback during the development of the ideas in this thesis.

I would also like to thank Marco Rose for his effort on the evaluation of the generator component which was developed in this thesis.

I would like to express further thanks to Florian Strauß for his help on the installation of the needed software environment.

I thank Capgemini sd&m for providing me the opportunity to participate in the company’s training program, and for the compensation of my travel costs.

I thank the “Arbeitsgruppe Softwaretechnik” for providing me a state of the art laptop for the thesis. Additional thanks go to Hans Schmitt and the Service Center Informatik for providing the software used in the thesis.

I thank the German Research Center of Artificial Intelligence (DFKI) for allowing me to use the company’s HiWi room during the thesis, and for the delicious coffee.

The most special thanks go to my family and friends for their support throughout the thesis.
List of Figures

2.1 The mapping problem in the interface models ........................................... 6
2.2 Coarse architecture of an application with a dialog controller ..................... 11
2.3 An example of a dialogue net ................................................................. 13
2.4 The layers of Naked Objects .................................................................... 17
2.5 An example GUI for Naked Objects .......................................................... 17

3.1 Generation of UI layers based on MODF ..................................................... 24
3.2 Method orientation ..................................................................................... 25
3.3 Data flows between dialogs ....................................................................... 26
3.4 Hierarchical decomposition of flows ......................................................... 28
3.5 Removing redundancies from flows ........................................................... 29
3.6 Limits of decomposition ............................................................................ 30
3.7 Directed acyclic graph of a flow with hierarchical decomposition ............... 30
3.8 The three axes of MODF .......................................................................... 33
3.9 The relations between the MODF meta model, models and specifications .... 37
3.10 UML Class Diagram of the MODF Meta Model ...................................... 38
3.11 The scenario for demonstrating the MODF model .................................... 43
3.12 An example of a MODF Model .................................................................. 44

4.1 Overview of flow interfaces ....................................................................... 46
4.2 Overview of the scenario .......................................................................... 50

5.1 Overview of the application landscape ....................................................... 60
5.2 The technical architecture of the Register Factory applications .................. 61
5.3 The GUI components of the Register Factory applications and their relations . 63
5.4 Overview of the relations between the generated artifacts ......................... 67
5.5 The visualization of MODF models in Eclipse ........................................ 73

6.1 Overview of the classes used by the example scenario ................................. 76
6.2 Demonstration of a possible flow execution using screenshots ..................... 77
6.3 A screenshot of the generated view of the PurchaseCd flow ....................... 78
6.4 A screenshot of a customized view of the ListCds flow ............................... 79
List of Figures

List of Tables

1.1 Terminology ...................................................... 4
4.1 Definition of the MODF Java annotations ....................... 56
4.2 Flow and FlowWithMultipleStates annotations ............... 57

Listings

4.1 Flow interface of the PurchaseCd flow ......................... 47
4.2 Flow interface of the ListCds flow ........................... 48
4.3 Flow interface of the ListCDs flow (extended) .................. 49
4.4 Flow interface of the DeleteCd flow .......................... 49
4.5 Flow Interface of the Show CD Details flow ................... 51
4.6 Flow interface of the SearchCd flow using nested interfaces ........................................ 51
4.7 Flow interface of the SearchCD flow without nested interfaces ........................................ 52
5.1 Backing bean data types ........................................ 69
5.2 Definition of an application logic layer Spring bean .......... 71
5.3 Example of a generated application logic layer wrapper ...... 72
5.4 Definition of an application logic layer wrapper Spring bean ........................................ 72

List of Algorithms

4.1 Transformation of a Java-based MODF specification to a MODF model ........ 58
Chapter 1

Introduction

Implementation of user interfaces is a time consuming and repetitive process which involves creating and linking different layers, components, libraries, configuration files and similar artifacts. Automatic generation of user interfaces aims to speed up the development process. Nevertheless, generation of user interfaces is a vague term in the sense that it is not clear what is generated. The contents of a user interface implementation highly depends on the user interface architecture. Various approaches have been developed in order to generate user interfaces. However, the user interface architecture of the target application is neglected during the generation process. The generated code has to be modified and extended according to the user interface architecture.

Existing enterprise applications, such as Register Factory business applications [7], have fixed architectures which cannot be changed because of the requirements of the used frameworks. Instead, the used technologies should be chosen based on the architecture of the system. The Register Factory was designed by “Bundesstelle für Informationstechnik des Bundesverwaltungsamtes” (BIT)[8] in cooperation with Capgemini sd&m [10]. A goal of this thesis was to find a practical way to generate graphical user interfaces (GUI) for Register Factory business applications. Although the user interfaces of Register Factory applications are usually simple, the implementation of the GUI is time consuming and error prone because of the need to make sure that the GUI implementation conforms to the GUI architecture. Therefore, a new user interface specification and generation method is presented which optimizes the work process of the user interface creation.

Existing user interface generation approaches usually focus on the generation of the presentation aspects. For example, user interface builders can be used to design the look and feel of the GUI by composing windows, text fields, buttons and similar widgets [61][14]. They can be used to generate GUI code which can be modified to bind the application logic to the GUI. However, GUI builders only deal with the generation of the user interface elements which are visible to the user, i.e. the presentation. Every other part of the user interface, which lies between the presentation and the application logic layer, must be implemented manually. Considerable effort must be spent in order to ensure that the implementation conforms to the user interface architecture, in particular for enterprise applications such as the Register Factory applications.

There are existing user interface generation approaches which can generate more than the presentation aspects. For example, rapid application development frameworks exist which can generate user interfaces and other parts of applications based on domain models [55][30][69].
Chapter 1 Introduction

However, such frameworks can only be used for applications which fulfill specific requirements which are demanded by the used framework. For existing applications, which already have a fixed architecture, the fulfillment of the requirements may be difficult or impossible depending on the restrictions of the architecture.

There is a need for a user interface generation approach which can cooperate with existing applications without restricting or changing their architecture. Not only the presentation of the user interface should be covered, but also other aspects, such as the navigation in the user interface and the data flow. This thesis proposes the usage of dialog models for generation purposes and sets the focus to the generation of user interface artifacts, with regard to user interface architectures. The contributions of the thesis are as follows:

- It gives an overview of the existing user interface models and analyzes their suitability for the generation of user interfaces for Register Factory applications.
- A platform independent user interface specification language was created using UML specifications.
- A Java-based user interface specification language was designed which enables the declarative specification of user interfaces for Java applications.
- A user interface generator was developed which is able to generate web user interfaces with regard to the Register Factory reference architecture.
- The user interface specification and generation approach was evaluated both by the author and Capgemini sd&m.

Chapter 2 introduces the related work in the area of user interface development approaches. In addition to the academic research in the area, state of the art frameworks are also briefly introduced, including open source and commercial frameworks.

Chapter 3 presents the Method Oriented Dialog Flows (MODF), the abstract user interface specification method which was designed during the thesis.

Chapter 4 introduces the user interface specification language based on Java annotations.

Chapter 5 covers the Register Factory Web GUI Generator, which is able to generate user interfaces with regard to the Register Factory reference architecture.

Chapter 6 shows the examples of the generated view and covers the evaluation of the Register Factory GUI generator as well as the MODF specification language based on Java annotations.

Chapter 7 presents the conclusions reached and new insights that arose from this work.
1.1 Terminology

This thesis contains statements about models, dialogs and other terms, which are not always used consistently in different contexts. Some of the terms used in this paper, such as beans and registers, are domain specific. This section introduces the commonly used terms and explain what they mean in the context of this thesis in order to avoid confusion about their meaning.

<table>
<thead>
<tr>
<th>Term</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>A model is an abstract representation of an item or concept. In the context of this thesis, the term model usually refers to a user interface model, which is a simplified representation of certain aspects of the user interface.</td>
</tr>
<tr>
<td>Meta model</td>
<td>A meta model describes what constitutes a model. It highlights properties of the model itself. A model must conform to its meta model.</td>
</tr>
<tr>
<td>Presentation model</td>
<td>In the context of this thesis, a presentation model is an abstract representation of the look and feel of the user interface. It describes the elements of the user interface, their layouts and other presentation aspects. An article published by Fowler [29] uses the term in a different sense. Because the both definitions are needed in the thesis, the term backing bean will be used for the latter, even though backing bean is a term that originates from the J2EE terminology. In the terms of the Fowler’s article, a presentation model pulls the state and behavior of the view out into a model class that is part of the presentation. In other words, the presentation model contains all the data that is needed to display a presentation. These will be called backing beans.</td>
</tr>
<tr>
<td>Dialog</td>
<td>Dialog is also a term that is used in different meanings. In this thesis, a dialog represents a conversation between a user and a user interface. The fact that dialog boxes or dialog windows are sometimes abbreviated as dialogs may cause confusion. In order to avoid confusion, the term view is used to describe the concrete user interface elements which are presented to users. Dialogs describe the interaction possibilities between the user and user interface, but not the presentation of the user interface.</td>
</tr>
<tr>
<td>Dialogue</td>
<td>Dialog and dialogue are American and UK spellings of the same word respectively. For consistency, dialog will be used in this thesis with the exception of dialogue nets, which is a fixed term that describes a concept in the related work.</td>
</tr>
<tr>
<td>Dialog flow</td>
<td>A dialog flow describes how the dialogs of the user interface change according to the user interaction. A dialog flow includes the navigation possibilities between the dialogs and may also include data flow.</td>
</tr>
</tbody>
</table>
Chapter 1 Introduction

<table>
<thead>
<tr>
<th>Term</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow</td>
<td>For simplicity, the dialog flows can be abbreviated as <em>flows</em>.</td>
</tr>
<tr>
<td>View</td>
<td>A view is a presentation part of the user interface, which is used to gather information from the user or to present information to the user.</td>
</tr>
<tr>
<td>Bean</td>
<td>A bean is a type written in the Java programming language conforming to a particular convention.</td>
</tr>
<tr>
<td>Backing bean</td>
<td>A <em>backing bean</em> is an object that contains all the data that is needed to display a view in the user interface. In addition to the content that should be displayed, backing beans can also contain state information to determine the status of the view, e.g. to set the number of displayed items or enable/disable widgets.</td>
</tr>
<tr>
<td>Register Factory</td>
<td>The <em>Register Factory</em> is a standard for the creation and management of IT systems for electronical registries. The Register Factory is introduced in Section 5.1.</td>
</tr>
<tr>
<td>Register Factory application</td>
<td>The registry applications, which are developed according to the Register Factory architecture, will be called <em>Register Factory applications</em>.</td>
</tr>
</tbody>
</table>

Table 1.1: Explanation of the commonly used terms in the thesis.
Chapter 2

Related Work

Many different approaches have been proposed for the development of user interfaces. Depending on the requirements of the user interface, each approach has its own advantages. This chapter introduces commonly known approaches and related work in order to reason about the benefits of Method-Oriented Dialog Flows.

A straight-forward approach to create user interfaces is programming them with toolkits. User interface toolkits, also known as widget toolkits, provide an application programming interface (API) which can be used to construct user interfaces. The APIs consist of methods for creating windows and widgets such as buttons, menus, text fields and trees. Most toolkits also offer layout managers as well as event handling mechanisms in order to support loose coupling of the user interactions (events) and the code that should handle the events.

Using user interface toolkits is an explicit approach. Every widget on the user interface has to be programmed explicitly. When the user interface gets more complex, the number of frames and widgets increases along with the lines of code. Since this thesis covers the generation of user interfaces in order to speed up the user development process, toolkits are mainly relevant because they can be used by user interface generators depending on the type of the generated user interface.

In contrast to programming user interfaces with toolkits, model-based user interface design follows the idea of capturing different aspects of a user interface with one or multiple models and constructing the user interface using (semi) automatic model transformations. In the context of model-based user interface design, a model is a simplified representation of certain aspects of the UI. Various kinds of models exist which capture different aspects. Kinds of user interface models include task, domain, dialog, presentation and user models [70][60]. This list is non-exhaustive and a model does not necessarily fall into exactly one kind.

In the process of user interface design, more concrete models are derived from abstract ones. The final step would be the generation of a concrete UI implementation from a presentation model, in general in combination with other models. Research on model-based UI design has been focused on domain, task, dialog and presentation models which will be explained in the following sections in order to distinguish the differences between them.

In 1998, Puerta and Eisenstein mentioned the limited success of the tools in model-based systems that attempt to generate concrete user interfaces from abstract representations such as task and domain models [62]. They argued that the main reason for the limited success was the lack of a general computation framework which bridges the gap between abstract and
concrete interface models. They dubbed this problem of linking abstract and concrete models the *mapping problem* (Figure 2.1).

![Figure 2.1: The mapping problem in the interface models (Figure taken from [62].)](image)

They proposed an approach called Model-Based Interface Designer (MOBI-D) to solve the mapping problem in order to design a wide variety of user interface types. Since then, more research has been done in this area. The concepts and frameworks that follow the domain-driven design principles, which will be introduced in Section 2.3, can be seen as solutions to the mapping problem from the domain model to concrete presentation and dialog models. It is harder to bridge the gap between task models and concrete user interfaces because task models are in general more abstract than domain models. Nevertheless, various research has been done in this area as well [17][47]. Research in the area of task models will be briefly explained in Section 2.4 after an introduction to the more concrete models, namely presentation and dialog models.

# 2.1 Presentation Models

Presentation models capture information about the look and feel of the UI. They describe generic but concrete user interface elements (widgets). This means, although the elements are generic and possibly platform independent, they are concrete enough so that a concrete user interface can be generated from the description. In general, presentation models do not include content (for example text, images, sounds), but they rather specify a mapping between widgets and content. In addition to the structure of output, presentation models also describe layouts as well as bindings to operations [70].

When the generation of user interfaces is mentioned, one may first imagine that the directly visible elements are generated, e.g. views, buttons, input fields, etc. Although Method-Oriented
Dialog Flows cover the generation of user interfaces including the presentation aspects, the focus is not on the presentation, but on the dialog flows. As mentioned before, the presentation models include bindings to content (e.g. backing beans) and operations (e.g. concrete methods or abstract actions). Nevertheless, Method-Oriented Dialog Flows can be used to generate simple presentation models, which already include these bindings and serve as a starting point for further refinement. The user interface developers can modify the generated models in order to create the desired layout. Therefore, the rest of the section will give an overview about existing presentation models.

Similar to most of the other model kinds, presentation models can be specified at different abstraction levels. Imagine a simple dialog where the user is able to select one element out of multiple elements. A presentation model describes the dialog and a widget inside the dialog which enables the selection. An abstract presentation model would describe this widget as “a widget that allows 1-out-of-N selection”. The possible values for the selection would not be specified in the description, but the widget would be mapped to data. A user interface generator can then process the abstract widget description and construct a concrete user interface by choosing a suitable widget, for example a list or radio buttons. A more concrete presentation model would specify the widget and its layout more precisely.

2.1.1 Languages for Specifying Presentation Models

The following sections present examples of languages that can be used to specify presentation models.

**XAML**

XAML (Extensible Application Markup Language) [49] is an XML-based language provided by Microsoft in order to create user interfaces for .NET Framework applications. Widgets are created by the declarative XAML markup language and mapped to backing types and operations in .NET assemblies. XAML user interface definitions can be also created visually using Microsoft Visual Studio, Expression Blend or other editors. Other user interface builders may also have their own UI description languages [15][61]. These languages can be used to specify presentation models even though the described user interfaces are actually tailored to toolkits.

**XUL**

Another XML-based user interface language is XUL (XML User Interface Language) [3], which can be used for building cross-platform applications. Applications written in XUL can be run in all platforms on which Mozilla runs, analogous to HTML documents which can be rendered by all platforms where a browser runs. XUL provides widgets like windows, labels and buttons to build the structure of the user interface and facilitates CSS (Cascading Style Sheets) [5] for layouting purposes. XUL uses another language called XBL (eXtensible Binding Language) in order to bind data and operations to widgets.
Chapter 2 Related Work

Jade

Jade [72] is an interactive tool developed in 1990 for generating graphical user interfaces from presentation models. One of Jade’s goals was to be look-and-feel independent by separating dialog’s widgets and look-and-feel. An application programmer writes a textual specification for the dialog’s widgets without any graphic information. Then, Jade uses the textual specification in combination with look-and-feel databases in order to generate a graphical user interface. The generated GUI can be modified using a graphical editor. Jade can maintain these modifications even if the textual specification changes. Jade is written in Lisp and the textual specifications may facilitate Lisp expressions in order to enable or disable widgets. The textual specification may also be used to create a mapping between buttons and Lisp functions.

UIML and UsiXML

Although the above mentioned languages can be used to describe presentations of user interfaces declaratively, which can be in principle rendered in different platform, they were designed for specific platforms. Out of the box, XAML widgets can be only bound to .NET assemblies. Applications written in XUL depend on the Mozilla platform and similar constraints exist for the other mentioned languages. Nevertheless, more abstract languages exist which are completely vendor neutral. UIML [1] and UsiXML [67] are examples of vendor neutral languages which can be used for specifying abstract user interfaces. They are both XML-based and can be used to specify different aspects of user interfaces, not only the presentation model. They do not constrain the API of the business logic that the UI is connected. They both support multimodal user interfaces, where the user interface enables multiple methods of communication between the end user and interaction devices [67].

2.1.2 Presentation Models for Web Applications

Different kinds of presentation models can be used for web applications. Web applications are usually developed using web application frameworks which also provide ways to specify presentation models. The next sections introduces common web application frameworks and their approaches to specify presentation models.

HTML: Presentation model or not?

Since there is no formal definition of a presentation model, one may argue that even HTML [39] can be seen as a way to specify a presentation model. HTML is a markup language which adds semantics to its content. Although HTML does not specify how documents should be exactly displayed, it can be used to describe widgets such as buttons and input fields as well as layout aspects to some degree. An HTML document can be rendered differently on a PC browser, a PDA, a beamer, etc. However, an HTML document contains its content staticly. Although HTML provides an abstraction with regard to presentation, the embedding of content prevents HTML from being used as a modeling language which solely captures presentation information.
Chapter 2 Related Work

HTML can be seen as a concrete target language for generators that try to generate a user interface that should run in a browser.

**JSP and PHP**

Languages such as JavaServer Pages (JSP) [52] and PHP [33] can be used to embed scripts into HTML, which can be used to provide dynamic content to documents. However, intermixing of HTML and scripts often leads to code being unstructured. Moreover, the freedom to use arbitrary scripts within view definitions creates the risk that the business logic gets mixed with the view definitions. This can be prevented by setting design principles so that the logic in view definitions is used for presentation only. Tag libraries (JSTL [53]) or templating engines (Smarty [32]) can be used to enforce such design principles in order to separate the application and presentation logic.

The definition of web pages using tag libraries or template engines is still an imperative approach. Even if the logic inside the view definitions is used solely for presentation purposes, the programmers still have to explicitly construct loops to create lists and tables or if-then-else statements to construct widgets depending on the underlying model.

**JSF and ASP.NET**

JSF [51] and ASP.NET [48], which can be seen as state of the art web application frameworks, follow a component-oriented approach for defining views. Each view is composed of a tree of components. Loops, branches and other simple logic can be completely removed from the view definitions. Data and operations can be mapped to components. The component-oriented approach also enables an easy integration of WYSIWYG editors, both ASP.NET and different JSF vendors include WYSIWYG editors.

JSF is a specification. In addition to the reference implementation (Mojarra) provided by Oracle [54], Apache also provides an implementation called MyFaces [21]. On top of those two implementation, so called component libraries go beyond the JSF specification and provide their own components, for example IceFaces [40], JBoss RichFaces [36] and Apache Tomahawk [22]. Moreover, although JSF is a web framework, different renderkits can be integrated in order to convert the view definitions into languages other than HTML. XulFaces [64] includes a renderkit which generates XUL definitions. Renderkits may be written for mobile devices as well. Apache Trinidad [23] supports mobile devices and is able to render JSF view definitions in different ways depending on the device and browser from which the page is requested.

Presentation models are an important part of the user interface design because they describe the whole look and feel of the user interface. They describe the visible part of the user interface. However, generators that are solely based on presentation models can only generate user interfaces which are non-functional. The generated presentation elements can be manually connected to the operations and data in the application logic layer. A direct connection between the presentation elements and the application logic layer is only possible for rapid prototyping or application with very simple GUI architectures. Enterprise applications may have multiple layers and components between the presentation elements and the application logic layer.
Chapter 2 Related Work

One of the goals of this thesis is the generation of a complete user interface, including the infrastructure underlying the presentation model. Therefore, presentation models are not appropriate as an input to user interface generators. The generators based on presentation models suggest a top-down modeling approach in the sense that UI designers begin the development of the user interface from the top by designing the look and feel. Then the generated non-functional presentation elements must be connected to the underlying layers. For the purposes of this thesis, a bottom-up approach is needed. It is not desirable to start from the presentation. Instead, it is suggested that the complete user interface should be generated based on the application logic layer. This means, the presentation models can be also generated based on the underlying layers.

The Register Factory Web GUI Generator, which was developed during this thesis, is able to generate simple presentation models using JSF and Facelets. The focus of the generator is not on the created layout. Nevertheless, the generated presentation models contain widget definitions which are connected to the generated backing beans and dialog flows. Thanks to the templating mechanism of Facelets, the desired layout can be set using the generator configurations or by modifying the generated views manually.

2.2 Dialog Models

While presentation models deal with the look and feel of user interfaces, dialog models deal with the dynamic behavior. Dialog models specify the flow between the views, or in other words, how a user navigates within the user interface. The possible sequences of views are usually specified as a state-transition graph, where states correspond to views and transitions correspond to the navigation between views. In addition to navigation possibilities, dialog models may also specify the data flow between the views.

Even if the developers do not explicitly model dialog flows, they always exist in every application which has a dynamic user interface. The flow may have been defined informally on paper mockups, by graphs, verbally or even implicitly in the developers mind. A dialog model is a formal specification of the dialog flow of an application.

The user interface description languages, which are used to specify the presentation models, do not provide a way to define dialog flows declaratively. Dialog flows, which are often intuitive to the developers, may become buried in the presentation layer. In client applications, clicking a button may open a new frame or change parts of the existing frame. Such changes must be explicitly coded in the handlers. In web applications, the flow is generally defined by hyperlinks or buttons that are in general embedded in the presentation layer of the application. This leads to a situation where the navigation links are scattered across the code. The explicit specification of dialog flows reduces the problem of navigation links being embedded in the presentation layer.

As the name of Method-Oriented Dialog Flows suggests, they are required to capture the dialog flows of applications. Therefore, the related work in this area is important to reason about the need for the Method-Oriented Dialog Flows.
2.2.1 Dialog Frameworks

Dialog modeling languages are often interrelated with dialog frameworks. Unless a dialog model is used solely for generation purposes, a dialog controller is needed which controls the flow of the user interface according to the dialog model. The dialog controller can choose which view to display and which application code to call. A dialog framework provides means to specify dialog models and a dialog controller. Figure 2.2 depicts the coarse architecture of an application with a dialog controller. Presentation and dialog models are registered with the dialog controller. The requests for user interface changes are received by the dialog controller. Figure 2.2 explicitly does not enumerate the interaction steps because they may differ in dialog frameworks. Depending on the dialog model and the context, the dialog controller finds out which view to display and which application methods to call. In other words, the dialog model acts as a bridge between the presentation and application layer. The dialog controller coordinates them.

Figure 2.2: Coarse architecture of an application with a dialog controller

Figure 2.2 also depicts the references from the dialog model to the presentation model (1) as well as to the application logic (2). Most dialog models associate states of the user interface...
with views in the presentation models (1), but the binding between the dialog model and application logic (2) is not possible in all dialog frameworks.

Dialog frameworks are structurally similar to workflow management systems which are used to manage business processes. Although their purposes are different, a dialog model may be seen analogous to a workflow definition, and a dialog controller analogous to a workflow engine.

2.2.2 Related Frameworks

Method-Oriented Dialog Flows (MODF), the specification method that was designed during this thesis, is mainly used for creating dialog models. Therefore, this section will introduce related frameworks and dialog modeling languages so that the main differences can be shown.

Terminal State Transition Diagrams

One of the first researches on dialog models has been made by David L. Parnas in 1969 when the focus of user interface design was the command language [56]. The introduced “terminal state transition diagrams” describe the dynamic aspects of a user interface. A terminal represents a connection point where the user can connect to the system. A terminal is at first in an initial terminal state which characterizes the possible input messages and their interpretations. According to the input of the user, the terminal may interpret the input, for example, by calling a program and/or changing its state. The paper also introduces a graphical notation for the state transition diagrams as well as common errors when defining the states. Although the main ideas of a dialog model are shown, the derivation of a concrete user interface from the dialog model is not explained.

Dialogue Nets

“Dialogue nets” are another technique for specifying the dynamic behavior of user interfaces [41]. Dialogue nets are dialog models based on petri nets, which support rapid prototyping of user interfaces by generating an output for an existing user interface management system. Among others, dialogue nets suggest solutions for handling parallelism and modal dialogs, hierarchical decomposition of flows and binding to application logic. In this context, parallelism means that multiple views can be opened and processed at the same time. Traditional state transition diagrams cannot be used to express parallelism because only one state can be active at the same time. Because parallelism can be expressed with petri nets, dialogue nets can also handle parallelism. When multiple views can be shown at a given point of time, modal dialogs may be needed, which are child dialogs which block the interaction with other dialogs until the modal dialog is closed. Modal dialogs can be expressed by dialogue nets using a special notation. In order to allow better structuring of dialog flows, dialogue nets allow the definition of abstract views called complex places. A complex place is not associated with a view, but it rather allows the partitioning of the diagram, as seen in Figure 2.3.

Figure 2.3 shows an application of the mentioned diagram partitioning technique. Round shapes are places which are associated with views. Rectangles are transitions. The place
Chapter 2 Related Work

Figure 2.3: An example of a dialogue net. The above Customer place is a complex place which is refined by a subdialogue net. (Figure taken from [41].)

called Customer has a double border which indicates that it is a complex place. The thick border of the Message place indicates a modal dialog. The above Customer place is not associated with a view. It is associated with a subdialogue net that shows the behavior of the complex Customer place. The subdialogue net is started when the complex place is marked (in petri net terms). The subdialogue net ends if a the token is removed from the complex place or all tokens are removed from the subdialogue net.

Dialogue nets use two steps for the binding to the application logic. At first, the transitions are unspecified transitions as in Figure 2.3. In the first step, the unspecified transitions are manually converted into fully specified transitions, which may include concrete conditions and actions. In the second step, the fully specified transitions are converted to the rules of the underlying user interface management system.

One of the requirements of the Method-Oriented Dialog Flows is the handling of the data flow. Dialogue nets do not provide the means to specify how the data is transferred between the dialog. They are used to specify coarse-grain dialogue specification, i.e. the navigation possibilities between the dialogs.

The Guilet Dialog Model and Dialog Core

The Guilet Dialog Model (GDM) [63] facilitates Guilets, which are abstract behavioral building blocks, in order to model the behavior of graphical user interfaces. GDM aims to be independent of the presentation and application layer. An optional component called Guilet Dialog Core (GDC) can be seen as a dialog controller, as introduced in Section 2.2.1.

GDM can be used to specify fine grained dialog flows compared to dialogue nets, which can describe coarse grained dialog flows. Coarse grained dialog flows include sequencing of views and the calls to the application logic layer [41], whereas GDM can be used to describe the behavior of widgets (inner Guilets) inside a dialog, as well as data flow and application calls.
between dialogs and widgets. The possibility of composing Guilets increases reusability. Once a Guilet is defined, it can be inserted into the dialog flow or used as an inner guilet. In order to define grained dialog flows, GDM provides seven major language elements, which are used to create Guilets with a graphical editor. The number of major language elements are low, considering that Guilet dialog models can express detailed behavior of graphical user interfaces. Nevertheless, the GDM tends to become very large depending on the level of modeled presentation details [63]. Therefore, the authors of GDM argue that Guilets should only be used to model the presentation parts that require an event processing by the GDM.

GDM is an expressive model as it captures fine grained dialog flows, data flows as well as the possibility for the decomposition of flows. GDM is not a pure dialog model in the sense that it also covers presentation aspects such as widgets and the relations between dialogs and widgets. The expressiveness of GDM also increases the complexity of its notation. Method-Oriented Dialog Flows aim to provide a simpler notation in order to speed up the user interface development process. Furthermore, Guilet Dialog Models have to be interpreted by a special dialog controller in runtime, such as the Guilet Dialog Core, in order to control the flow of the user interface. This implies that GDM requires a special GUI architecture with a dialog controller. Method-Oriented Dialog Flows can be used in the same way. However, in general, Method-Oriented Dialog Flow specifications are not interpreted directly. Instead they are used to generate the user interface. This way, MODF abstracts from the exact implementation of the user interface architecture.

The Dialog Flow Notation and Dialog Control Framework

The Dialog Flow Notation (DFN) [4] can be used to specify dialog models which can express sequences of dialogs, the data flow between them and calls to application logic. DFN aims to be flexible so that the specified user interfaces can be adapted to different rendering devices without requiring redundant specifications. Therefore, DFN can be used to create variants of flows for different presentation channels using special notations.

DFN also enables structuring and composition of flows using hierarchical dialog modules. Every dialog flow graph must be encapsulated in a dialog module which has initial and terminal events in order to make nesting and connection to another modules possible.

The Dialog Control Framework (DCF) coordinates the flow of dialog models, presentation and application layers. The concept of DCF corresponds to the dialog controller that was defined in Section 2.2.1.

Last but not least, the authors of DFN [4] also specify the formal semantics of both DFN and DCF, in contrast to most of the dialog meta models which are explained informally or semi-formally using natural language and graphs.

The Dialog Flow Notation is specific for web applications and needs the Dialog Control Framework to be integrated into the GUI architecture. As mentioned in the previous section, Method-Oriented Dialog Flows are not restricted to web applications and can be used to generate different user interface implementations based on different GUI architectures.
Chapter 2 Related Work

Spring Web Flows

The Spring Web Flow (SWF) [66] is a framework which uses the concept of dialog models for web applications that are based on the Spring Framework. Dialog flows can be defined using a XML-based language. Flows may contain view states that are linked to a web page which can be rendered with different technologies such as plain HTML, JavaServer Pages (JSP) [52] or JavaServer Faces (JSF) [51]. Flows also define transitions between states, actions or subflows to support modularity. The SWF framework acts a dialog controller to manage flows.

The GUI architecture of Register Factory makes use of the SWF framework. The XML-based flow definition language of SWF is very expressive but also complex. The manual creation of SWF flow definitions can become cumbersome. Furthermore, the creation process becomes repetitive for simple user interfaces. The Register Factory Web GUI Generator, which was implemented in this thesis, is able to generate the SWF flow definitions based on MODF specifications, which are simpler in contrast to the SWF flow definitions.

JavaServer Faces

JSF [51] provides a navigation framework which partially fulfills the goal of separating the flows from the presentation. By defining navigation rules using XML, one can specify a target view depending on an abstract outcome string and the current view. This way, JSF pages may have links that define abstract outcome strings instead of pointing to existing pages. Thus, the navigation between the dialogs is extracted from the dialogs to an external configuration file. However, no actions can be called on these transitions unless JSF pages directly call application logic. Furthermore, structuring and composition of flows are not possible because the flows cannot be encapsulated. Action definitions for calling application logic and composition of flows belong to the requirements of Method-Oriented Dialog Flows.

JBoss Seam

JBoss Seam [38] is a development platform for building rich Internet applications in Java. Along other features, it provides improvements to the navigation framework of JSF. It offers a mechanism to replace ordinary JSF navigation rules in order to allow the navigation rules to include action definitions using an expression language.

As an alternative, Seam can be also used with the jBPM Process Definition Language (JPDL), which is similar to the XML-based flow definition language of SWF. However, JPDL is much more complex because its capabilities go beyond the definition of dialog flows. Among others, JPDL can be used to define traditional workflow management as well as the orchestration of services in a SOA environment [37].

Together with JDPL, Seam is a state of the art platform for building web applications. Although the thesis focuses on the generation of the user interfaces for Register Factory applications, which use the SWF framework, generation of user interfaces based on Seam and JPDL would be a very interesting topic as well.
2.3 Domain Models and Domain-Driven Design

A domain model captures business concepts and constraints. Concepts, objects, attributes and operations of objects describe the domain for which the software is developed. Domain models may be specified using UML, the Entity-Relationship-Model (ERM), database schemas and similar notations. Source code, which captures domain information, is also a valid representation of a domain model. For example, in object-oriented languages classes can be used to represent domain entities, relations and operations.

Domain-driven design (DDD) is an approach to software development that suggests that the primary focus should be on the domain and domain logic. DDD is not a formal methodology, but a set of principles pioneered by the book of Eric J. Evans named “Domain-Driven Design: Tackling Complexity In the Heart of Software” [18]. DDD is an interesting topic in the context of this thesis because encapsulating the business logic in the domain model allows the automatic generation of user interfaces. The idea of generating user interfaces from underlying business models was also proposed by Alan Kay in 1990 [45] (referenced from [59]).

2.3.1 Naked Objects

Richard Pawson introduced Naked Objects in his doctoral thesis [59], a concrete approach which follows the domain-driven design principles. According to Pawson, Ole-Johan Dahl and Kristen Nygaard, who received an A. M. Turing award for their role in the invention of object-oriented programming, conceived ‘objects’ as representations of the entities that model a chosen domain. This approach proposes that each object should encapsulate its attributes and relationships to other objects as well as with the behaviors associated with that entity. Such ‘objects’ are also called ‘behaviorally complete’. Pawson argues that newer object-oriented systems, which deliberately separate domain entities and their behavior, are ‘behaviorally weak’. Pawson quotes Firesmith to describe this pattern as:

“dumb entity objects controlled by a number of controller objects” [19].

‘Naked objects’ is a concept where the business application is written solely in terms of domain entities. Domain entities are not solely data structures, but they also encapsulate all domain-related operations. Because of the behaviorally completeness of the domain objects, a generic presentation layer permits the users to view and manipulate the domain objects (depicted in Figure 2.4).

The generic presentation layer provides an object-oriented user interface by introspecting the domain objects. The concept was given the name ‘naked objects’ because the users can directly see the domain objects and their methods in the generic presentation view without adornment. Figure 2.5 shows a GUI rendered by the Naked Objects Framework as proposed in [59]. Even though the presentation layer can be provided in different platforms, all GUIs that use the proposed generic presentation view look similar. Classes and objects (instances) are represented by separate windows. Users see a list of classes in the domain model. Objects methods can be accessed by choosing an object and using the context menu or drag and drop operations.
Constantine criticised naked objects in 2002, soon after the OOPSLA conference where naked objects were presented [12]. In particular the usability aspects of the generic object-oriented
presentation view were criticized. Constantine argues that the ‘object-action’ (or ‘noun-verb’) style of interaction, which is forced by naked objects, is not always natural to the end users. Picking an action first and then the object can be more suitable depending on the situation. Constantine also points out that the presentation of information should be tailored based on different aspects:

“The greatest usability problem with Naked Objects is the one-size-fits-all premise on which the approach rests. Instead of tailoring the presentation of information and the operation of the user interface to fit the unique aspects of the context, the application, and the user needs, one solution is presumed to fit all problems, provided all the relevant domain objects are properly identified with all their important behavior fully modelled [12].”

With no use-case controllers permitted, all business functionality must be encapsulated within entity objects rather than within use-case controllers that sit on top of those entities [59]. This approach is suitable for domain modeling because it is consistent and logical. Imagine a transactional banking system which offers operations such as money transfers, withdrawals and deposits. Using the object-oriented user interface, the user must first create an object that represents the transaction, for example a ‘transfer’ object. After setting the source, target accounts and the transfer amount in the transfer object, the users may commit the transaction or even rollback using the object methods. This simple scenario is actually similar to existing verb-noun interaction scenarios, for example online banking sites where users first open the ‘transfer’ page and then fill in a form. There is a little twist however. In traditional systems, a user would log in with an account, then transfer money by solely specifying a target account and the transfer amount. The system would use a session to pass the source account to the ‘transfer’ operation. The user would not see the fact that the underlying ‘transfer’ operation needs the source account as input.

The above scenario is simple and understandable. However, interacting with such ‘business objects’ can get very unnatural to end users when the business logic gets more complex. Calling methods on the object, which represents the business logic, may change the state of multiple objects. Their presentations adapt accordingly. In particular the availability of the methods in context menus may change. During a long workflow, the end users may have to call different methods on multiple objects in a certain order. The users may not always see the path clearly, which leads to the users goal. On the other hand, a use-case based user interface would guide the user using wizards and possibly hide the steps that are needed in domain model. Nevertheless, the applications developed using the naked objects concept come with a free graphical user interface, even though the usability of the UI is questionable.

2.3.2 Related Frameworks

Generating user interfaces or even whole applications based on domain models is not only an abstract research topic. Various frameworks exist in this domain. The next sections will introduce some of these frameworks which aim to support “rapid development” of applications using the domain-driven development principles.
Chapter 2 Related Work

OpenXava

OpenXava [55] is a tool for rapid Java web development that follows the domain-driven design principles. The development process is driven by writing simple annotated domain classes. The presentation and persistence layers are generated based on the domain model. The generated artifacts may be customized using Java annotations. The generated user interface is mainly capable of create, read, update and delete (CRUD) operations.

Roma <Meta> Framework

The Roma <Meta> Framework [30] also follows the domain-driven design approach for the development of enterprise level Java applications. The authors called Roma a “meta” framework because it is, in contrast to OpenXava, independent of the technologies that are used. Both OpenXava and Roma are so called full-stack frameworks because all parts of a fully functional application, particularly presentation and persistence layers, are generated solely based on the domain model which is generally specified using plain old Java objects (POJOs).

Frameworks that follow the naked objects pattern

Numerous modern frameworks adapted the naked objects pattern. Naked Objects MVC [31] uses Microsoft’s ASP.NET MVC 3 [48] to realize the concept. Apache Isis [20], JMatter [69] and Tynamo [11] also follow the naked objects pattern using Java technologies. The main differences between these full-stack frameworks are the technologies that are used to provide the layers around the domain model. JMatter uses Java Swing [50] for presentation and Hibernate [35] for persistence. Tynamo (formerly known as Trails) uses Apache Tapestry [24] as a web framework and Hibernate for persistence. Apache Isis (formerly known as Naked Objects for Java) follows a more flexible approach by allowing the binding of different presentation layers providing both desktop and web applications. Isis also offers the same flexibility for persistence, testing or even the programming language, similar to the Roma framework [30]. Isis foresees the usability issues in the generated OOUIs and provides ways to customize the generated UI:

“The OOUIs generated by Apache Isis are especially suitable to “expert” users, typically those internal to your organization who have a good understanding of the domain and just want to get their job done. But for less expert users (or for a webapp deployed on the internet), a more scripted UI may be called for. Isis therefore lets you customize the user interface according to your users’ needs [20].”

The DDD frameworks promote “rapid development of applications” by generating almost every needed software component around the domain model. Rapid development is also one of the goals of this thesis, with one key difference: This thesis focuses on the generation of user interfaces based on an existing application. It is neither desired nor allowed to generate software components other than the user interface. Nevertheless, the fact that the generation of user interfaces are based on an existing application, implies that the application already has a domain model. Therefore, Method-Oriented Dialog Flows also make use of the domain
model. The concepts that are used by the above mentioned frameworks can be partially used by the Method-Oriented Dialog Flows.

## 2.4 Task Models and Others

While domain models focus on the domain entities and their operations, task models try to describe what the users want to do and why. The main focus of task models is the users’ goals and the activities that are performed to reach the goals [70]. Task models are used for different software development activities such as design, analysis and evaluation. Task models can also be used to derive concrete user interfaces. Therefore, this section introduces commonly used task modeling notations and approaches which aim to derive user interfaces from task models. [57].

Task models can be expressed with different notations and at different abstract levels. Commonly used task modeling notations include GOMS (Goals, Operators, Methods, and Selection rules) [42], UAN (User Action Notation) [34], ConcurTaskTrees [58] and TKS (Task Knowledge Structure) [43]. All mentioned notations allow the hierarchical structuring of tasks. Most abstract activities are specified as high-level tasks and can be refined by decomposition. In the context of this thesis, the exact notations of the task models are not very relevant. Adept [44] and TRIDENT (Tools foR an Interactive Development EnvironmeNT) [2] provide processes and tools in order to bridge the gap between task models and concrete user interfaces. Because this thesis does not propose a solution for the “mapping problem” from task models to more concrete models, they will not be explained here in more detail.

In addition to presentation, dialog, domain and task models, other kinds of models exist which describe different aspects of user interfaces. User models characterizes the users that will use the system. They try to describe users’ knowledge of the system, age, motor abilities and similar properties that can affect the design of the system. User models should be specified early in the design process because they may affect all design decisions. Platform models can be used to specify the input/output capabilities of devices. Platform models can particularly affect the presentation models because widgets and layouts should be selected depending on the capabilities of the input/output devices [70]. Additional kinds of models may exist in order to capture different aspects of user interfaces.

Since this thesis covers the topic of user interface generation, user models and platform model are also related because they can be used to influence the generation process. Depending on the user and platform, generators may choose different widgets, colors, fonts, sizes, layouts or even split a specified dialog into multiple dialogs. Mainly the presentation of the user interface is affected. Method-Oriented Dialog Flows aim to generate complete user interfaces. Presentation is only a small part of the user interface and this thesis does not focus on generating the presentation. Therefore, Method-Oriented Dialog Flows do not make use of user and platform models in order to keep the specification language as simple as possible.
Chapter 3

Method-Oriented Dialog Flows

The idea of creating a new user interface specification technique originated while trying to find a practical way to generate graphical user interfaces for applications based on the Register Factory architecture. Such applications will be called Register Factory applications. The graphical user interfaces of Register Factory applications seem to be simple, but the GUI architecture consists of multiple layers which have distinct purposes. The manual creation of graphical user interfaces is a time consuming and repetitive process which is also error prone because it is not easy to make sure that the implementation of the GUI artifacts conform to the GUI architecture.

The method-oriented dialog flows were designed to fulfill the requirements of the user interfaces of Register Factory applications while being as generic as possible so that they can be applied to other applications. In order to determine the scenarios, in which the specification and generation method can be applied, the following coarse requirements were analyzed:

1. The architecture of the existing application cannot be changed for the purposes of making GUI generation possible. It is not desirable to adapt the underlying application logic layer to the GUI. Instead, the GUI should be specified on top of the existing application logic layer.
2. It must be possible to specify dialog flows. It is not enough to generate separate dialogs which are not connected.
3. It must be possible to relate the dialog flows with the methods in the application logic layer.
4. It must be possible to specify the data flow between the dialogs.
5. The dialog flows should be modular in order to assure reusability.

The research on related work shows that the generators, which generate graphical user interfaces based on the domain model, are not suitable for Register Factory applications because they do not fulfill the first requirement. Most domain-driven design (DDD) frameworks are full stack frameworks. They generate not only the user interface, but also the persistence layer and possibly other components in order to create a fully functional application. The generation of other components is not always optional. In particular, the frameworks depend on the generation of the persistence layer even though the developers can choose which implementation should be generated. The generation of the persistence layer is required because it is used
by the generated components to perform create, read, update and delete (CRUD) operations. Operations other than CRUD can only become automatically available if the domain objects are “behaviorally complete”, as explained in Section 2.3.1. This is not the case in Register Factory applications, i.e. operations are not encapsulated in the domain objects, but rather provided by components that control the domain objects.

The first requirement implies that the application logic layer and the GUI should be independent in the sense that it must be possible to change the GUI behavior without changing the application logic layer. In contrast to this implication, DDD enforces that the domain model and the user interface behave the same. This means, the user interface directly reflects the domain model. Any change in the domain model influences the user interface. The DDD frameworks require that the domain model satisfies certain requirements that are determined by the framework. They are not suitable for generating the user interfaces for existing systems, in particular not registers.

All of the above coarse requirements can be fulfilled using dialog models. In this section, method-oriented dialog flows (MODF) are presented which can be used to design modular dialog models in order to generate user interfaces in a practical way. Method-oriented dialog flows fulfill the above requirements without restricting the concept to registers only.

### 3.1 The Characteristics of MODF

As introduced in Section 2.2.2, various dialog models and frameworks already exist. The Guilet Dialog Model (GDM), the Dialog Flow Notation (DFN) and Spring Web Flows (SWF) can all be used to describe modular flows, bindings to the application logic layer and the data flow. The notations of these dialog models are powerful in the sense that they can express complex dialog flows. However, the main goal of method-oriented dialog flows was the conception of a simpler meta model in order to support rapid development of user interfaces. Method-oriented dialog flows define a set of ideas for how the dialog models can be used for the generation of concrete user interfaces. The key characteristics of method-oriented dialog flows are explained in the next sections.

#### 3.1.1 Generation of User Interfaces

All mentioned dialog frameworks provide the means for separation of concerns and implementing a clearly layered architecture, but they do not restrict the developers to do so. It is in the developers hands that the implementation in fact uses the advantages of the frameworks. The developers have to make sure that the GUI is implemented according to the architecture. If the architecture consists of multiple layers with precise interaction patterns, implementation and validation of the layers can be a time consuming process.

The following paragraph from the Seam reference [46] advocates this property as one of the characteristics of Seam:
“You can layer your application according to whatever architecture you devise, rather than being forced to shoehorn your application logic into an unnatural layering scheme forced upon you by whatever combination of stovepipe frameworks you’re using today.”

Although the Seam reference [46] mentions that the Spring framework is more restricting with regard to architecture, this does not apply to the dialog models defined with SWF. A flow can access any component that is defined as a bean in the context of the flow. Not only the application layer beans, but also backing beans or any bean that resides in a lower layer, can be accessed. The beans can not only be accessed from the flow definitions, even the JSF pages can access the same beans.

The authors of DFN and DCF suggest an architecture which is based on the Model-View-Controller (MVC) pattern [4]. The Guilet Dialog Model introduces an optional Guilet Dialog Core that can be used to enable loose coupling of presentations [63]. Nevertheless, the dialog model and the dialog controller are only a part of the GUI architecture. All other layers and the components, which conform to the GUI architecture, have to be implemented manually.

Method-oriented dialog flows propose the generation of all architectural elements of the user interface based on the dialog model. The idea is to specify an abstract dialog model using method-oriented dialog flows on top of the application logic layer. The dialog model directly accesses the application logic layer methods, ignoring any layers that should be implemented in the concrete GUI implementation. And all architectural elements (e.g. controllers, backing beans, concrete dialog models, views) can be generated automatically based on the method-oriented dialog flows. Figure 3.1 shows the idea of generating complete UI layers based on MODF specifications. Even if the GUI layer involves multiple layers and components, the developers only have to specify the behavior of the user interface using a MODF specification language. The MODF specifications and the existing application logic layer are given to a GUI generator as input.

Different GUI generators may be implemented to generate user interfaces conforming to different GUI architectures. Two examples are shown in Figure 3.1. On the left side, a GUI layer is created which uses a dialog controller and a dialog model. In addition to the presentation model, another component is generated between the dialog model and the application logic layer. On the right side, a GUI layer is created which does not make use of a dialog model. The method-oriented dialog flows are used to embed the behavior of the GUI into the generated components. Note that generation of GUI architectures, which do not use dialog controllers, may imply restrictions on the MODF specification. Section 3.1.4 covers this topic in more detail.

If the GUI architecture is to be changed, there is no need to modify every element across the GUI layer. Instead, the changes can be applied centrally in a single place by modifying the GUI generator. As good as this may sound, the regeneration of the GUI layer does not always cope well with components that were modified after the generation. Generated elements may need to be modified at some point of time because they cannot completely replace hand-crafted code. This regeneration problem is a known issue with code generators in general and will not be handled in the context of this thesis.
As mentioned, the generators need the application logic layer components and the MODF specifications as input. In order to explain how complete GUI layers can be generated based on those inputs, we introduce the term of *method-orientation* in the context of dialog models.

### 3.1.2 Method-Orientation

Most dialog frameworks provide a mechanism to call application logic code. Dialog models contain *actions, expressions* or similar terms to bind methods of application logic code to events of the user interface. On the occurrence of the defined events, the methods are called by the dialog controller. Method-oriented dialog flows go beyond this simple behavior. The basic
idea is to use the information contained in method signatures in order to generate elements of a user interface. Using the domain model of the existing application, the methods in MODF specifications can be used to generate backing beans, controllers, UI widgets, views and similar elements.

The name *method-oriented dialog flows* was dubbed to this concept because the generation of user interface artifacts is based on the methods that are referenced by the dialog flow specifications. Figure 3.2 shows an example to demonstrate the idea.

Consider a dialog which should display a list of items, let’s say CDs. The developer specifies a ListCds flow which calls a getCds method on entrance of the flow. A GUI generator processes the MODF specification and may generate UI artifacts such as a backing bean that stores the result of the getCds method, a controller that is called by the dialog controller when the dialog is requested, as well as a view that is able to display the list of CD objects from the backing bean. The exact artifacts that will be created depend on the target UI architecture. In the example, the continuous lines show the artifacts that are generated based on the ListCds flow, i.e. the backing bean, controller and the view. The dotted lines roughly demonstrate how the contents of the generated artifacts are yielded from the method signature.
3.1.3 Data Flow

The `getCds` method in the above example has no parameters. In more realistic scenarios, methods with parameters will need to be called. In some cases, the arguments for the methods are obtained by the user input. In other cases, they have to be passed from the previous dialogs. Method-oriented dialog flows provide a way to specify the data flow based on the methods which are referenced in the MODF specifications.

MODF specifications can connect the methods which are used by different dialogs.

Figure 3.3 demonstrates the following scenario: Just like in the previous example, the first dialog displays a list of CDs. The `getCds` method is called on the entrance of the flow. In this dialog, the user can choose a CD and search for related CDs. For this purpose, a `searchRelatedCds(cd)` method is called on the entrance of the next dialog.

The `searchRelatedCds` method has a `cd` parameter, which needs to be obtained from the selected CD in the previous dialog. This is a typical scenario where the result of one method needs to be passed to another method as argument. Such scenarios can be handled by MODF specifications. In order to specify the connections between methods, MODF specifications assign variables to method parameters and results. The variables are then used to specify how methods are connected.

In the example in Figure 3.3, the result of the `getCds` method is stored in a result variable. The result variable is connected to the `cd` parameter of the `searchRelatedCds` method. Note that the `getCds` method returns a list of CDs whereas the `searchRelatedCds` method expects exactly one CD. MODF specifications can handle such cases by using path expressions. In this example, the "`selectedItem`" path expression means that only the selected row from
the list of CDs should be passed as argument. The details of data flow specifications will be explained in Section 4.3.

3.1.4 Reusability of Flows

In a discussion board about the problem of typical projects having too much special-purpose GUI code, Philippe Detournay, a fellow computer scientist, points out an interesting topic:

“You can reuse a button, you can reuse a tabbed panel, but whatever more sophisticated than this doesn’t deal with the presentation itself, but with the flow. And the flow can’t be reused from one application to another because, well, the flow is more or less what defines an application from a user’s point of view.” [16]

If the flow of a whole application is considered, it is true that the flow is tailored to the application and hard to reuse in a different context. However, dialog models enable the reusability of flows by allowing modular flow definitions. That is one of the great advantages of the modular dialog models. One can define an arbitrarily complex flow in order to handle a specific task and use the same flow in different contexts of the application, or even within different applications.

For dialog models, reusability can be achieved in different ways. In this thesis, the following approaches were identified:

- Static hierarchical decomposition of flows
- Subflow calls
- Flow inheritance

All approaches are orthogonal. The MODF meta model allows the definition of static hierarchical decomposition of flows and subflow calls. The next sections explain the mentioned approaches.

Static Hierarchical Decomposition of Flows

Static hierarchical decomposition of flows provides an abstraction mechanism in order to help the modeling process. Using decomposition, flows are partitioned into modular flow definitions. High level flows can be specified without going into the details. High level flows can include abstract states whose exact behavior are left unspecified in the modular flow definition. The behavior of the abstract states are specified by referencing other flows. Decomposition can consist of higher hierarchy levels depending on the need for abstraction. The modular flow definitions can be statically merged into one flat flow which does not have any decomposition.

Figure 3.4 shows an example to demonstrate hierarchical decomposition of flows. The flat flow on the left side does not have a hierarchical decomposition. The scenario is as follows: At the start, a list of CDs is shown, where the user can choose a CD to purchase. If the user chooses to purchase a CD, a dialog is shown to enter the billing data. When the user wants to proceed to checkout, a confirmation dialog is shown. If the user confirms, the CD is purchased and the
system shows recommendations of other CDs. In this scenario, purchasing a CD consists of a sequence of two dialogs: one for entering the billing data and one for confirmation. Hierarchical decomposition is used on the right side to abstract from the exact dialog sequence to purchase the CD. An abstract Purchase CD state, depicted with a thicker border and a light blue hue, is defined without specifying the concrete dialogs. With the help of this abstract state, the flow designers do not have to specify how exactly the purchasing occurs. The designers specify the transitions from the abstract state depending on the outcome of the abstract state.

Hierarchical decomposition can be also used for removing redundancies. In the flow of an application, it is very probable that some parts of the flow appear multiple times in the same way. Instead of specifying the repeated parts redundantly, the flow can be decomposed so that the repeated parts are extracted and referenced. Figure 3.5 shows a scenario for demonstration purposes. The previous scenario is slightly modified. After the list of CDs is shown, it is possible to show the details of a CD in the list or purchase one directly. When the details of a CD are shown, it is also possible to purchase the displayed CD. Purchasing fires a sequence of dialogs which is needed in both cases. This causes redundant specification of the dialog sequence and transitions. The redundancy is removed via hierarchical decomposition.

On the left side in Figure 3.5, note that the “List CDs” and “Show CD details” cannot reference the same “Enter billing data” state because the “cancel” transition needs to target the previous state. This follows from the fact that state transition diagrams do not keep track of the state history. If the same state is referenced from different places, one cannot specify “how to go back” without duplicating the states. The Figure 3.6 shows a shorter scenario in order to demonstrate the problem clearly. After purchasing a CD, the user can choose purchase further CDs from the recommended CDs. If the “Show recommended CDs” state targets the same “Purchase CD” state, one cannot specify if the control should return to “Purchase CD” or “Show recommended CD”, because it is dependent on the previous state.
Chapter 3  Method-Oriented Dialog Flows

Figure 3.5: On the left side, purchasing steps are defined redundantly. On the right side, redundancies are removed via hierarchical decomposition.

The problem in Figure 3.6 can be solved by adding a duplicate “Purchase CD” state after “Show recommended CDs”. However, this solution is only possible if the flows are acyclic, i.e. the flows must form a directed acyclic graph (DAG), where each flow corresponds to a vertex in the DAG and each transition to an abstract state corresponds a directed edge from the container flow to the referenced flow.

Figure 3.7 shows an example of a directed acyclic graph obtained from flows with hierarchical decomposition. The “List CDs Flow” contains two abstract states which reference the “Purchase CD Flow”. The two references to the “Purchase CD Flow” can be seen in the directed acyclic graph, on the right side. The directed acyclic graph can be also seen as an overview of the structural decomposition of flows. If the “Purchase CD Flow” would contain an abstract state that references the “List CDs Flow”, the directed graph would have a cycle. Thus, it would not be possible to define transitions that go to the previous state.

As an implication of DAGs, the flows with hierarchical decomposition can be flattened so that they do not contain any partitioning, without changing their semantics. For example,
Figure 3.6: The “cancel” transition cannot be specified without adding another duplicate “Purchase CD” state after “Show recommended CDs”.

Figure 3.7: The directed acyclic graph also shows an overview of the structural decomposition of flows.

in Figure 3.5, both flows are semantically equivalent. The partitioned flows on the right side can be derived from the flat flow. In the other direction, the flat flow on the left side can be reconstructed from the flows on the right side. The relation is bidirectional. Flattening would not be possible if the directed graphs had cycles.

The possibility to go back to the previous dialog is an important requirement in user interfaces, particularly in web user interfaces. The fact that hierarchical decomposition does not keep track of the previous states makes it hard to achieve this requirement. In particular, it is not possible if the flows contain cyclic references.
Subflows

Subflows add a dynamic behavior to the hierarchical decomposition, which is achieved by keeping track of the previous states. In order to do this, modular flows can be defined, where the flows are allowed to call other flows. Calling another flow is referred as a subflow call. When a subflow is called, it starts the execution from its initial state and returns back to the caller after it reaches an end state. Since the modular flow definitions do not know in their caller states, this approach needs a dialog controller that keeps the track of subflow calls.

Stacks are appropriate data structures to keep track of the subflow calls. When the main flow starts, the dialog controller pushes the flow to the stack. Whenever a subflow is called, the called flow is pushed to the stack along with its contents. When the subflow reaches an end state, the flow is removed from the stack and the control returns to the last flow in the stack. The usage of a stack not only allows the possibility to go back, it can also keep track of the content history. The same dialogs can appear multiple times in the stack with different contents. This means, each time a subflow ends, the contents of the last flow can be restored. This cannot be done without a controller which manages the history in runtime. If such a controller is not present, the contents of a state must be recalculated each time the state is entered.

The approach using subflows is similar to hierarchical decomposition with the following differences. While hierarchical decomposition is purely static, subflows add a dynamic behavior to the flows and needs a controller in runtime. Using hierarchical decomposition, the navigation between dialogs can be statically embedded in the generated user interface layer. However, the user interfaces are not aware of the navigation and content history. Although flows with subflow calls cannot be handled statically, they can be used to express every scenario which be expressed with hierarchical decomposition. In addition, they can keep track of the navigation and content history, and they allow cyclic subflow calls, and thus, also recursive calls.

In order to remain flexible, the MODF meta model does not include special constructs for the abstract states or the subflow calls. The generators, which generate user interfaces based on MODF models, can interpret whether the transitions between different flows are subflow calls or references in the sense of structural decomposition. This way, the generated user interfaces are not restricted. As a consequence of this flexibility, the semantics of the model is loose in the sense that the exact interpretation of transitions is left unspecified. The loose semantics is justifiable because it is desirable to be able to generate user interfaces with or without dialog controllers. The semantics is not fixed purely based on the MODF model, but also the target interface architecture, and thus, the corresponding user interface generator.

The Register Factory Web GUI Generator interprets transitions between the flows as subflow calls because the generated user interface is backed by the Spring Web Flow framework, which also serves as a dialog controller. Nevertheless, other generators can be implemented which generate a user interface without a dialog controller, by embedding static navigations links in the presentation layer.
Flow Inheritance

The reusability of flows can also be increased using the concept of flow inheritance, similar to type inheritance in the object-oriented programming languages. Within the dialog models, which were introduced in the related work, only the Spring Web Flow (SWF) framework deals with flow inheritance, although the flow inheritance in SWF has differences compared to the type inheritance in Java. Although the MODF meta model does not capture flow inheritance, the Register Factory Web GUI Generator is able to make use of the flow inheritance capabilities of SWF in order to increase the modifiability of generated flow definitions.

3.2 The Flexibility of MODF

Since the goal of MODF is to generate complete user interface layers, a high degree of flexibility is desirable. MODF aims to remain language and platform independent so that MODF can be applied to different kinds of applications in order to generate various user interfaces. Figure 3.8 shows the three axes of method-oriented dialog flows.

3.2.1 User Interfaces

MODF specifications do not predetermine the user interfaces that can be generated. In the context of this thesis, the Register Factory Web GUI Generator was implemented, which is able to generate Spring-based web user interfaces by taking MODF specifications as input. The Register Factory Web GUI Generator is explained in Section 5. The type of the generated user interface is determined by the generator component and not by the MODF specifications.

3.2.2 Application Requirements

Since method-oriented dialog flows aim to generate user interfaces for existing applications, it is desirable that it can be applied to different types of applications. As mentioned before, method references are one of the driving aspects of the generation process. The generators make use of the type information within method signatures. Therefore, the language, in which the methods are defined, must have a type system in order for MODF to work.

Generic presentation layers, as in the Naked Objects concept [59], require that the underlying programming language has introspection capabilities, which enable a computer program to observe its own structure at runtime. Since MODF propose the generation of user interfaces, introspection is not required. Type information can be inspected statically during the generation process. The frameworks, which can be used to observe type information from source or intermediate level code, can significantly reduce the implementation effort for creating generators.

The way to execute the referenced methods is another very important topic. The method references in MODF specifications must be precise enough in order to resolve and call the
method. For example, if the underlying application is implemented in an object-oriented
programming language, the method reference must include the fully qualified name of the
method, i.e. the fully qualified name of the containing class, the method name and the fully
qualified type names of the parameters. Even if the method reference is precise enough so that
the method can be resolved, it is not trivial to determine how to instantiate the containing
class. The following approaches were identified in order to deal with the problem:

- A trivial solution can be achieved by requiring that the referenced methods must be
  static. However, this is too restricting.

- The factory method pattern can be applied. In this case, the types, in which the refer-
  enced method are defined, must provide a factory which can be used by the generated
  user interface to create an instance of the type. As a simpler alternative, the user in-
  terface can call the (parameterless) constructor of the type. In both cases, it is hard
  to manage the dependencies of the types. If the factory method or the constructor of
  the type needs parameters, this approach cannot be applied without further modifica-

Figure 3.8: The three axes of MODF: (1) Generated user interfaces, (2) Application requirements, (3)
Specification languages.
Chapter 3 Method-Oriented Dialog Flows

Last but not least, the above restrictions can be overcome using the inversion of control principles. The generated user interface should not be responsible for creating the needed type. Instead, it should be able to request it from the underlying application. There are different ways to achieve this. One option is using the service locator pattern. The Register Factory Web GUI Generator uses the dependency injection mechanism provided by Spring. In both cases, the application creates and registers instances in a registry. Then, the generated user interface may request an instance of the needed type from the registry.

The above approaches work if MODF specifications reference methods, which are defined in object-oriented programming languages. For other kinds of languages, different requirements apply. But the inversion of control principles can be applied for different scenarios as well.

MODF specifications can also refer to web services. In this case, the binding is easier because web services are per definition loosely coupled. However, type information must be acquired from the web service reference. Therefore, a prerequisite for using web services is the binding to a WSDL [71] or a similar service description document, which includes the parameter and return types of the referenced web service operation.

3.2.3 Specification Languages

The meta model of MODF was designed in an abstract way so that different languages can be used to create MODF specifications. The following three approaches are proposed:

1. Annotated Java interfaces (Flow Interfaces)
2. XML-based specification languages
3. Graphical specifications

Annotated Java interfaces were successfully used for the generation of the services of Register Factory applications [68]. The idea is to annotate the source code so that a user interface can be generated for the application. However, the requirements state that the application logic layer cannot be modified, which implies that the annotations have to be written somewhere else. To overcome this conflict, new annotated Java interfaces (abbreviated as flow interfaces) and methods can be created instead of annotating the existing layers. The concept of creating new annotated Java interfaces, which was also applied in the service generator, is an unusual way of using Java annotations and may cause confusion when first learning its usage. Nevertheless, it also has its advantages. Therefore, the following sections compare the mentioned specification approaches from different perspectives.

**Type references:** When using flow interfaces, simple type names can be used in the annotations thanks to import declarations. Using XML, types have to be referenced by fully qualified names. However, the XML schema can be extended so that it allows import declarations similar to the Java language. This way, XML can also include simple type names. In graphical
specifications, types also have to be referenced by fully qualified names. In contrast to flow interfaces and XML, import declarations cannot be used in an intuitive way. Nevertheless, graphical editors, which are integrated into an IDE, can be used to choose the referenced types within the IDE, which is even better than handling it with imports.

**Ease of processing:** One argument for flow interfaces was the easy processing possibilities for annotations. The Register Factory Web GUI Generator uses the Eclipse Java development tools (JDT) [26] to process annotations. Processing annotations is not as easy as expected. In particular, the referenced types can not be obtained easily. Given an annotation that references a class object, e.g. `@foo(A.class)`, one can expect that an object can be retrieved that represents the referenced type. This is not the case in JDT. One can only obtain the name of the type as a string, “A”, which is not even the fully qualified name. Then, different utility methods must be used in order to resolve the “A” string into a fully qualified name using the import declarations. Finally, the fully qualified type name can be used to resolve a class object. Helper methods can be defined to ease the processing, but in the end, the processing of annotations with JDT is not easy.

Although XML-based specifications are not used in this thesis, the author thinks that processing XML files would not be harder in comparison to the flow interfaces. The process of resolving class objects from type references would remain almost the same. The existence of various languages and tools, which can be used to read and query XML documents, is also an advantage when processing XML specifications. Graphical specifications are stored in an internal format which is generally XML. Therefore, the same comments about the ease of processing apply to the graphical specifications.

**Validation:** In this context, static type checking is the biggest advantage of using flow annotations. Type references are checked statically by the compiler without any additional work. The code cannot be compiled successfully if non existing types are referenced. All referenced types must be existent during compile time. If using an IDE like Eclipse, the type errors are shown directly and dynamically within the source code editor. XML and graphical editors do not have this property out of the box. Custom validators can be implemented for XML and graphical editors, but the effort would be higher in comparison to flow interfaces.

**Learning curve:** When using flow interfaces, the Java language is being used for a purpose other than programming, i.e. one is actually specifying a dialog flow by creating annotated Java interfaces and methods. The annotations are not being used to give additional properties to existing Java elements, which is the usual way of using annotations. Specifications using flow interfaces are unusual in this sense. After learning how the concept works, it can be used to specify flows efficiently. However, the unusual usage of Java elements and annotations causes it to be hard to learn in the beginning.

XML-based schemas would be structurally very similar to the flow interfaces. Nevertheless, XML documents are commonly used for specification purposes [6][49][3][66][51]. Therefore, it would not cause the same confusion in comparison to flow interfaces.

Graphical specifications would be the easiest to understand because dialog flows are essentially graphs and graphs can be comprehended easier when they are visualized. In fact, when explaining what a concrete flow interface or XML specification does, one of the best ways would be visualizing the specification so that readers can see the flow.
Consistency to the service generator: Because the service generator [68] also uses the concept of annotated interfaces, it is more consistent to use a similar approach from the perspective of Capgemini, where both the service generator and the Register Factory Web GUI Generator will be used together. Although not as consistent as flow interfaces, XML and graphical specifications would still remain compatible with the service generator.

In the context of this thesis, a flow specification language was defined based on Java annotations in order to make use of static type checking and be consistent with the service generator. This way, the user interface developers can use plain Java within their favorite IDE to specify the flows, without having to use XML. Furthermore, the MODF meta model is designed with the Eclipse Modeling Framework (EMF) [27] in order to allow easy integration of future graphical editors. Because the Graphical Modeling Project [28] provides tools for developing graphical editors based on EMF, the implementation and integration efforts of graphical editors can be reduced.

3.3 The MODF Meta Model

The previous chapter explained the key characteristics of MODF. The MODF meta model is designed in order to define MODF models which possess the key characteristics. Before discussing the MODF meta model in detail, the terminology should be made clear. Figure 3.9 visualizes the relations between the following terms:

- **MODF model**: A MODF model is a formal description of a user interface. It describes the dialogs, the transitions between the dialogs, data flow, calls to application logic and other elements. User interface generators use MODF models in order to generate the user interface implementations.

- **MODF meta model**: The MODF meta model formally specifies what a MODF model can consist of. From the perspective of the class-object paradigm of object-oriented languages, the MODF meta model can be seen as a class representing the properties of MODF models. The MODF models can be seen as objects, i.e. instantiations of the MODF meta model. Therefore, Section 3.3.1 explains the MODF meta model using an UML class diagram. Section 3.3.2 explains example MODF models using UML object diagrams.

- **MODF specification languages**: Using MODF specification languages, one can create more abstract MODF specifications in order to build MODF models in a declarative way. MODF specification languages are abstractions of the MODF meta model.

- **MODF specifications**: MODF specifications are more abstract representations of MODF models. For one MODF model, different specifications can exist.

To summarize, the MODF models possess the following properties:

- **Modularity**: MODF models define flows which can be used in different contexts independently.
Figure 3.9: MODF specification languages establish a language abstraction to the MODF meta model. The MODF model can be specified using MODF specification languages.

- **Binding to application logic**: The flows can define calls to the methods (or functions, operations, etc. depending on the context) of the underlying application.
- **Presentation independency**: The model does not predetermine the presentation that will be generated.
- **Data flow**: MODF models define how the data is transferred between flows by associating method parameters and results.

The following section explains the MODF meta model in detail.
3.3.1 UML Specification of the MODF Meta Model

The MODF meta model formally specifies what a MODF model can consist of. Figure 3.10 shows an UML class diagram of the MODF meta model which visualizes the contained classes, attributes and their relations. The actual MODF meta model used by the Register Factory Web GUI Generator is modelled using EMF Ecore [27], which is a dialect of UML. Since UML is more common, UML was chosen to explain the meta model in detail.

![UML Class Diagram of the MODF Meta Model](image)

Figure 3.10: UML Class Diagram of the MODF Meta Model. The diagram is logically partitioned into three parts: (1) Flows, dialogs and navigation, (2) Application binding, (3) Data flow.

The UML class diagram in Figure 3.10 is logically partitioned into three parts. This informal partitioning only serves for structuring purposes and does not have any influence on the MODF models. The parts called flows, dialogs and navigation shows the classes that are used for the creation of the modular flows, dialogs and the navigation between them. The application
binding part provides the classes, which are used to call application logic code at certain points of the flows. The data flow part is used to specify how the data is transferred between the flows.

ModelRoot

The ModelRoot is a container for the flows of the application. It is the entrance point to MODF models. The ModelRoot can be passed to a user interface generator in order to start the generation process.

Flow

A Flow represents a modular process which can be executed in different contexts in the user interface. Each flow has a unique name and a list of states, which are associated with dialogs. Each flow must have at least one state. Flows may also have input parameters in order to allow data transfer from other flows.

State

Each State in the flow shows a graphical view to the user and expects a user interaction. This graphical view is a web page for web user interfaces or a kind of window for desktop applications (e.g. Java Swing). States have transitions to other states. States may have an arbitrary number of actions which are executed at specific points of flows.

EndState

An EndState is a special kind of state which represents the end of a flow execution. In combination with the SubflowOutcomes, the end states determine how the flow execution continues after the parent flow terminates.

Transition

The transitions define how states are connected to each other. A transition can be triggered by user interactions in the view, for example clicking of a button.

The target state can be a state of another flow. In this case, the exact behavior depends on how a generator interprets the model. If subflows are used as mentioned in Section 3.1.4, such transitions are named subflow calls. If a subflow call occurs, a subflow execution starts and the parent flow is blocked until the subflow terminates by reaching an end state.

Each transition, which target a state of another flow, must define a SubflowOutcome for each EndState of the target flow so that the next state can be determined after the target flow terminates.
In addition, transitions may have arguments in order to specify the data, which is required to be able to navigate to that transition.

**SubflowOutcome**

Each transition, which targets a state of another flow, must define a SubflowOutcome for each EndState of the target flow. After the target flow terminates, the SubflowOutcome determines the next state and implicitly the next flow in the user interface. Each SubflowOutcome references two states, one end state and one target state. A SubflowOutcome has the following meaning: if the target flow terminates with the referenced EndState, then the next state is the referenced target State.

**Action**

Each Action corresponds to a call into the application logic layer. The type of the signature attribute is intentionally left unspecified because it may change according to the underlying application. For object-oriented languages, the signature must be a fully qualified method signature which contains the method name, the fully qualified names of the owner type and the method parameters.

Action is an abstract class and its concrete subclasses determine when the action will be executed.

**EntryAction**

EntryActions are executed when the state is entered. In general, the result of entry actions will be used as the content of the view.

**TransitionAction**

Each TransitionAction references a list of transitions which determine when the action is executed. A TransitionAction is only executed if one of these transitions take place.

TransitionActions have a guarded attribute which determines whether the result of the action should restrict the transition to take place. The TransitionActions, whose guarded attribute is set to true, will be called guarded TransitionActions.

A guarded TransitionAction can be a validator method which returns false if the transition to the next state is not allowed. Guarded TransitionActions have restrictions on the signature of the action. For example, if a guarded TransitionAction returns a string, then it is not clear, for which values the transition can take place. For this purpose, the meta model could be extended so that guarded TransitionActions could contain a list of allowed or disallowed values. To keep the meta model as simple as possible, it is assumed that the guarded TransitionActions will have a boolean or a similar return type, which can be used to decide whether the transition is possible.
**InputParameter**

Flows may also have **InputParameters** in order to allow data transfer from other flows. Each input parameter has a unique name and a type. The type of the `type` attribute is intentionally left unspecified. It must be customized depending on the type system of the application logic layer. For object-oriented languages, a string can be used to contain a fully qualified type name.

The input parameters can be used by the actions which are defined in the same flow. The connection between an input parameter and an action parameter is done implicitly using their names. For example, if the flow has an input parameter named `cd` and an action with a signature of `purchaseCd(cd)`, then the action can use the `cd` input parameter. If the action is a `EntryAction`, it is called directly with the needed input parameter. If the action is a `TransitionAction` and the view provides input fields so that the parameters can be filled by the user, the input parameter can be used as an initial value for the input fields. Since a flow can contain different actions which have the same parameter names, the input parameter can refer to an action so that the desired parameter name can be uniquely resolved.

**Variable**

Actions may define **Variables** so that **Arguments** can be used to refer to them. Variables can represent the data in the method parameters. For example, imagine an `EntryAction` with a signature of `getCdDetails(isbn)`. The action will be called when the flow starts in order to display the details of the CD. If another flow should be called which needs an `isbn` as an **Argument**, a variable can be defined to represent the value of the `isbn` parameter. The defined **Variable** can then be used to give the value of the `isbn` to another flow.

**ResultVariable**

Actions can have **ResultVariables** which correspond to the result value of the action. A **ResultVariable** can be used to pass the method result to another flow as argument.

A **ResultVariable** can have a path expression which defines what part of the result will be assigned to the variable. If a path expression is not defined, the **ResultVariable** refers to the complete result value of the method. A path expression can determine if a field of the result value should be used. Nested expressions can also be used in order to navigate through variable fields. For example, imagine that the method returns an object of the `User` class which has an `address` field. Let the `Address` class have a `city` field. Then, the **ResultVariable** can have the "address.city" path expression which points to the city of the user.

A special case applies if an `EntryAction` returns a collection. As mentioned before, the result of entry actions are generally used as the content of the view. If an `EntryAction` returns a collection, this generally implies that the view will show this collection as a list or table. It is desirable to be able to pass only a selected item from the collection as an input, for example to display the details of the selected item. In some cases, it is also desirable to pass multiple selections as an input, for example to delete all selected items. To handle selections,
two keywords `selectedItem` and `selectedItems` are introduced. These keywords can be added to the path expressions in order to specify that the variable refers to the selected row or multiple selected rows respectively.

For example, imagine a `ListCDs` flow has a single state and an `EntryAction` that returns a list of CDs. The view displays a list of CDs. Imagine a `PurchaseCd` flow which expects a single CD as input. In order to define a transition from `ListCDs` to `PurchaseCd`, a `ResultVariable` is declared for the `EntryAction` in `ListCDs`. The “`selectedItem`” path expression is used so that the result variable refers to the selected CD in the view. When the transition takes place, the selected CD will be passed to `PurchaseCd` as input. If the “`selectedItem`” path expression is not used, the whole collection of CDs would be passed as input.

**Argument**

An Argument is used by transitions to define which data should be passed to another flow as argument. Each Argument refers to a Variable in order to define the data that should be passed. Each Argument also refers to an InputParameter of the target flow in order to define a mapping from the variable to an input parameter of the target flow.

Arguments can also have path expressions similar to the path expressions in `ResultVariable`. If a path expression is not defined, than the value of the variable will be used. A path expression can specify if a field of the value should be used.

### 3.3.2 An Example Scenario

In order to demonstrate how the MODF meta model is used to construct MODF models, this section will present the MODF model of an example scenario. Because MODF models can get quite large, a very simple scenario is used, which is depicted in Figure 3.11. The scenario is a simplified version of the scenarios which were presented in Section 3.1.4. The first dialog shows a list of CDs. The user is able to select a CD in the dialog and start a new flow which is responsible for the purchasing process. When the `Purchase CD` flow is started, the selected `cd` from the first dialog is passed to the `Purchase CD` flow. The `Purchase CD` flow has a single state which shows an input form so that the billing information can be filled, as well as buttons for confirming and cancelling the purchasing operation. If the user confirms, the `purchaseCd` method is called with the filled data and the `cd`, which was passed from the caller flow. Both confirming and cancelling ends the `Purchase CD` flow and the control returns to the `List CD` state.

Figure 3.12 shows an UML object diagram of the MODF model which represents the above scenario. Even for simple scenarios like the one in Figure 3.11, the MODF models get considerably large. Therefore, the UML diagram was partitioned into logical segments for structuring purposes. The gray vertical bar in the middle logically separates the objects of the two flows. The horizontal layers logically separate the roles of the objects.
Despite the size of the diagram, its semantics is the same as the above textual description of the scenario. The MODF model describes the scenario formally so that it can be processed by a machine, in the context of this thesis, a user interface generator.

Starting from the top, the model root has two flows: ListCds and PurchaseCd. The objects to the left of the gray vertical separator are the elements of the ListCds flow. The other objects are the elements of the PurchaseCd flow.

The ListCds flow has a single state. The state has an EntryAction which is executed when the state starts and a purchase transition that leads to the other flow. Since the purchase transition leads to another state, SubflowOutcomes were defined for each EndState of the PurchaseCd flow. They SubflowOutcomes map both end states of the referenced flow to the ListCds state. This implies, no matter how the called flow terminates, the control goes back to the ListCds state.

In order to specify the data flow between the flows, the EntryAction contains a result variable which corresponds to the selected cd within the result. This is specified with the selectedItem path expression of the result variable. If a path expression is not specified, than the ResultVariable would refer to the collection of CDs that is returned by the getCds method. The specification of the selectedItem path expression also implies that the purchase transition is only possible if a cd is selected.

The mapping between the result variable and the input parameter of the PurchaseCd flow is handled by the Argument, which is contained in the purchase transition. In other words, the Argument connects the result of the getCds method with the input parameter of the PurchaseCd flow.
Chapter 3 Method-Oriented Dialog Flows

On the right side of the vertical separator, the elements of the PurchaseCd flow are defined. The PurchaseCd flow has one regular state which corresponds to a dialog, and two end states which represent the outcome of the flow. The confirm and cancel transitions connect the regular state with the corresponding end states. The regular state of the PurchaseCd flow also has a TransitionAction, which references the confirm transition in order to specify that it will be only executed on that transition.

The InputParameter itself does not define how it will be used. Therefore, it is connected to the TransitionAction which means that the value input parameter (cd) will be used as a parameter in the TransitionAction.
Chapter 4

MODF Specifications using Java Annotations

MODF models can get very large even for simple scenarios. Programmatically constructing MODF models is not practical. Nevertheless, MODF models are intentionally designed that way so that they can be easily processed by user interface generators. Because one of the goals of this thesis is the practical specification of user interfaces, MODF models are only used as intermediate models. Declarative user interface specification languages should be used to specify MODF models practically and MODF models should used to generate the user interfaces.

A user interface specification language based on annotated Java elements was created, which enables user interface developers to specify MODF models in a declarative way. 3.2.3 explained the reasons for choosing Java annotations. To summarize, using Java annotations allows user interface developers to:

- specify user interfaces using plain Java, without using XML or other languages
- make use of the static type checking capabilities of the Java compiler
- use their favorite IDE and take advantage of code completion and other capabilities
- be consistent with other annotation based specifications, in particular the service generator [68].

As mentioned in Section 3, existing packages in the application logic layer (and the layers below) are not allowed to be modified. This also applies for annotations. Existing classes, methods and other Java entities should not be annotated directly in the existing packages. To define the flows, a new Java interface is created and annotated for each flow. These annotated Java interfaces are called flow interfaces.

In each flow interface, a new annotated method is defined for each method that should be used in the application logic layer. In order to resolve the actual location of the method, the signature of the method must be exactly the same as its signature in the application logic layer and the method must be annotated with the type where the method resides. Method annotations specify whether the method should be called on the flow entrance or on the execution of a transition. The transitions between the flows are specified by annotating the flow interfaces.
Figure 4.1 shows an overview of a flow interface. On the left side, a simple flow is shown where a list of CDs is shown. If the user navigates to the search transition, a subflow is started for searching cds. On the right side, the flow interface is shown which represents the same model.

```java
public interface ListCds {
    @OnEntry(inClass = CdManagement.class)
    public List<Cd> getCds();
}
```

Figure 4.1: An overview of a flow interface.

In order to define the flow, a flow interface named ListCds is created, which is a plain Java interface annotated with the Flow annotation. The transition to the subflow is defined as part of the Flow annotation. The name and to annotation members are used to specify the name of the transition and the flow interface of the subflow respectively. The getCd method, which is called on the flow entrance, is written as a method definition in the flow interface. The getCd method is annotated in order to specify when the method should be called and where the method is located. The next section will use scenarios in order to explain how Java annotations are used for MODF specifications. Section 4.2 will give a detailed description of the annotations.

### 4.1 Example MODF Specification

In order to demonstrate the basic concepts and data flow, the scenario in Section 3.3.2 will be reused, which was also depicted in Figure 3.11. Because two flows are involved, two flow interfaces are created. In all listings in the following sections, the import statements are omitted for simplicity.

Listing 4.1 shows the flow interface of the PurchaseCd flow. A plain Java interface is created and annotated with the Flow annotation. The name of the interface defines the name of the Flow. Flow interfaces, which are annotated with the Flow annotation, have implicitly exactly one state and one view as a consequence. Handling of multiple states are done with the FlowWithMultipleStates annotation, which will be explained later.
Chapter 4 MODF Specifications using Java Annotations

The Flow annotation has a transitions member which specifies the transitions to the end states or the other flows. In the PurchaseCd flow, Confirm and Cancel transitions are defined. Their names are specified using the name member of the Transition annotation. Transition names must be unique within a flow. The target of the transition is defined using the to member of the Transition annotation, by referring to the class object of a flow interface. The class objects are referenced in order to make use of the the static type checking capabilities of the Java compiler. In the PurchaseCd flow, other flows are not referenced. Both transitions lead to an end state. In order to denote that the transition leads to an end state, the void type is used. This means, the transitions that lead to an end state must refer to the void.class in the to member.

```java
@Flow(transitions = {@Transition(name = "Confirm", to = void.class),
                    @Transition(name = "Cancel", to = void.class))})
public interface PurchaseCd {

    @OnTransition(inClass = CdManagement.class, callOnTransitions = "Confirm")
    public void purchaseCd(Cd cd, BillingData billingData);
}
```

Listing 4.1: Flow interface of the PurchaseCd flow

For each transition that targets void.class, an end state will be created implicitly. This reduces the specification effort because the end states do not have to be defined explicitly. However, as a consequence, other flows cannot target the end states to specify how the outcome of a subflow should be handled. The problem is solved by targeting the transitions by name, which lead to an end state. Although this solution is not ideal because of the semantical differences, it is more practical because the end states do not have to be defined in an explicit way.

The purchaseCd method is declared within the flow interface. The OnTransition annotation and its callOnTransitions member denote that the method will be called on the Confirm transition. The values of the callOnTransitions member must match the unique transition names that are defined in the Flow annotation.

The signature of the purchaseCd method and the inClass member of the method annotation are used to resolve the actual location of the method to call. The inClass member targets the class object of the type that contains the method. This way, the static type checking mechanism of the Java compiler can be used. The CdManagement type must exist so that the the flow interface can be successfully compiled. With the usage of Java imports, which are ommitted in the listings for simplicity, the type names can be written as a simple type names. This way the specifications remain more readable.

Although the Java compiler can check the existence of the CdManagement type as well as the parameter types, it will not generate a warning if the <importedPackage>.CdManagement#purchaseCd(Cd, BillingData) method does not exist. The check can only be done during the processing of the specification. This is the case because the compiler does not know which method is referenced by the annotated method. For the compiler, it is a regular method declaration with an annotation. During the processing of the specification, the referenced method can be resolved and the existence of the resolved method can be checked.
Listing 4.2 shows the flow interface of the ListCds flow. The structure of the flow interface is the same. Since the purchaseCd method in the PurchaseCd flow expects a cd parameter, which must be inputted from the caller flow, additional annotations are used to specify the data flow.

```java
@Flow(transitions = @Transition(name = "Purchase", to = PurchaseCd.class, arguments = "cd"))
public interface ListCds {
    @OnEntry(inClass = CdManagement.class)
    @ResultVariables("cd=selectedItem")
    public List<Cd> getCds();
}
```

Listing 4.2: Flow interface of the ListCds flow

The getCds method has the OnEntry annotation which means that the method will be called on flow entrance. The result of the method is a collection of Cd objects and the PurchaseCd flow expects exactly one cd as an input argument. The ResultVariables annotation is used to bind the result of the method to a parameter. The ResultVariables annotation contains a text in the following syntax: `variableName[ = pathExpression]` where the pathExpression is a string separated with dots. If a path expression is not used, then the method result is assigned to the variable name as it is. If a path expression is used (e.g. “fieldA”, “fieldA.fieldB”, “selectedItem.fieldC”), then only the specified field is assigned to the variable. The selectedItem path expression can be used for collection types in order to denote that only the element, which is selected by the user, will be assigned to the variable. In this case, the cd variable refers to the Cd object that is selected by the user.

The arguments member of the Transition annotation is used to give the cd variable to the PurchaseCd flow as input. Since the PurchaseCd flow contains a method which has a parameter named cd, an input parameter named cd will be created implicitly for the flow. If the variable name and the input parameter name are different, a mapping can be specified in the arguments member of the Transition annotation. The arguments has the following syntax: `variable[.pathExpression][ > inputParameterName]`. The pathExpression is optional and can be used in the same way as the pathExpression in the ResultVariables annotation. inputParameterName is also optional and can be used to specify a mapping from the variable name to the input parameter name. For example, if the input parameter name was purchasedCd, then the following argument could be used: “cd>purchasedCd”.

Note that the purchasedCd method has a second parameter billingData. Since the purchasedCd method is called on a transition and not on the flow entrance, the method parameters do not have to be necessarily inputted from the caller flows. The parameters can be filled by the user using forms. The Register Factory Web GUI Generator is able to generate views which contain input fields based on the method signatures. Getting the input from the users is not possible if the method is called on the flow entrance. This implies, all parameters of the methods, which are annotated with OnEntry, are required input parameters for the flow.

The previous example scenario introduced the basic structure of flow interfaces, the definition of transitions, methods as well as simple argument passing. The previous scenario will be extended in order to demonstrate
Chapter 4 MODF Specifications using Java Annotations

- the definition of flows with multiple states,
- the handling of the outcome of subflows,
- examples of cases where path expressions are needed.

The extended scenario is shown in Figure 4.2. Note that the subflow calls (depicted with thick borders) do not have transitions that go back to the caller state because the control goes per default back to the caller state when a subflow call terminates. If this behavior is not wanted, then a transition can be specified to change the default flow. The scenario still contains the ListCds and PurchaseCd flows. The ListCds flow was extended for showing the details of a CD, deleting a CD as well as searching CDs.

Listing 4.3 shows the extended flow interface of the ListCDs flow. In comparison to Listing 4.2 in the previous section, it has three additional transitions defined within the Flow annotation. The usage of path expressions can be demonstrated in the Purchase and Delete transitions. Both PurchaseCd and DeleteCd flows expect an isbn of a CD as input. As in the previous example, the cd variable refers to the selected CD in the user interface. Using the isbn path expression, the cd.isbn argument gives the value of the isbn field of the CD as input.

Another point to mention is the mapping that is used in the arguments member of the transitions. Since the DeleteCd flow has an input parameter named cdIsbn, a mapping has to be done from isbn to cdIsbn. This is done with the cd.isbn>cdIsbn expression in the argument member of the Delete transition.

```java
@Flow(transitions = {
    @Transition(to = PurchaseCd.class, name = "Purchase", arguments="cd"),
    @Transition(to = ShowCdDetails.class, name = "Details", arguments="cd.isbn"),
    @Transition(to = DeleteCd.class, name = "Delete", arguments="cd.isbn>cdIsbn"),
    @Transition(to = SearchCd.class, name = "Search")})

public interface ListCds {
    @OnEntry(inClass = CdManagement.class)
    @ResultVariables("cd=selectedItem")
    public List<Cd> listCds();
}
```

Listing 4.3: Extended flow interface of the ListCDs flow

Listing 4.4 shows the flow interface of the DeleteCd flow. This flow can be used to delete a CD after showing a confirmation view. Its flow interface does not introduce a new concept. Nevertheless, it can be used to demonstrate the modularity of the flow interface. Although it is referenced by the ShowCdDetails flow, it does not contain any reference to the ShowCdDetails flow or any other flow.

```java
@Flow(transitions = {
    @Transition(to = void.class, name = "Confirm"),
    @Transition(to = void.class, name = "Cancel"))

public interface DeleteCd {
    @OnTransition(inClass = CdManagement.class, callOnTransitions = "Confirm")
    public void deleteCd(String cdIsbn);
}
```

Listing 4.4: Flow interface of the DeleteCd flow
Figure 4.2: The extended scenario contains three new flows for showing the details of a single CD, searching CDs and deleting a single CD.

In Figure 4.4, the transitions, which lead to void.class, are end transitions which return the control to the caller flow. This way, the DeleteCd flow remains independent from other flows and can be used in different contexts.

Listing 4.5 shows the flow interface of the ShowCdDetails flow. On the start of the flow, the getCdDetails method is called with an isbn argument in order to load and show the details of a CD. The user can choose to go back or delete the CD. The ShowCdDetails flow demonstrates two points: the handling of the outcome of subflows and mapping of arguments.
to input parameters.

```java
@Flow(transitions = {
    @Transition(to = void.class, name = "Back"),
    @Transition(to = DeleteCd.class, name = "Delete",
               arguments = "isbn=cdIsbn",
               outcomes = @Outcome(on = "Confirm", to = void.class)))
public interface ShowCdDetails {

    @OnEntry(inClass = CdManagement.class)
    public CdDetails getCdDetails(String isbn);
}
```

Listing 4.5: Flow Interface of the Show CD Details flow

The delete transition is a subflow call to the DeleteCd flow. After the DeleteCd flow terminates, the default behavior is to go back to the caller state. In this case, the CD details will be displayed per default. If the CD is deleted after the DeleteCd flow, it does not make sense to show the details of the deleted CD. This case shows an example where the handling of subflow outcomes is needed. If the CD gets deleted, the ShowCdDetails flow should end. If the CD is not deleted, the details of the CD can be shown. Since the latter is the default behavior, only the exception outcomes must be specified, i.e. only the cases must be specified where the control of flow must be redirected to another flow.

The handling of the subflow outcomes can be specified with the outcomes member of the Transition annotation. The nested Outcome annotation has two members. The on member refers to the end transition of the subflow, which should be handled. The to member references a flow interface in order to redirect the control to the first state of the referenced flow. As usual, void.class can be used to end the flow, which is used in this case. @Outcome (on = "Confirm", to = void.class) can be read as: “If the subflow call terminates with the Confirm transition, go to void.class, i.e. end the flow.”.

Definition of flows with multiple states is an issue because of its increased complexity. One solution is to use nested interfaces to define the states within the interface. Listing 4.6 shows the flow interface of the Search CD flow using nested interfaces to define the states.

```java
@FlowWithMultipleStates
public interface SearchCd {

    @State(transitions = {
        @Transition(to = SearchResults.class, name = "Next"),
        @Transition(to = void.class, name = "Back")})
    public interface SearchForm {

        @OnTransition(inClass = CdManagement.class, callOnTransitions = "Next")
        @ResultVariables("cd=selectedItem")
        public List<Cd> searchCds(String string);
    }

    @State(transitions = {
        @Transition(to = SearchForm.class, name = "Back"),
        @Transition(to = ShowCdDetails.class, name = "Details",
                    arguments = "cd.isbn"),
        @Transition(to = void.class, name = "Finish")})
    public interface SearchResults {

    }
}
```

Listing 4.6: Flow interface of the SearchCd flow using nested interfaces
In Listing 4.6, a new annotation named \texttt{@FlowWithMultipleStates} is used because this flow interface has different annotation members such as \texttt{globalTransitions} as well as nested interfaces to define the states. The transitions that are defined with \texttt{globalTransitions} are available from each state. Each state may also define additional transitions. In this case, no global transitions are defined. The flow has two nested interfaces for the two states. The first state represents a view, where the user can formulate a search query using a string. On the transition to the next state, the \texttt{searchCds} method is called. The second state represents a view where the result is displayed. The user can choose to go back to reformulate the search, finish the search process or display the details of a CD.

Although the above approach using nested interfaces works, it is not practical to use. Using the fact that flows with multiple states usually consist of wizard-like views where the user can only go forward and backward between states, a shorter approach can be used by stating the number of states in a flow. Listing 4.7 shows the flow interface of the \texttt{SearchCD} flow using this shorter alternative. No nested interface is defined. Instead, the flow interface has a new annotation member called \texttt{numberOfStates}.

\begin{verbatim}
@FlowWithMultipleStates(
   numberOfStates = 2,
   globalTransitions = (@Transition(to = void.class, name = "Cancel"),
   @Transition(to = ShowCdDetails.class, name = "Details",
                  arguments = "cd.isbn"))

public interface SearchCd {
   @ResultVariables("cd=selectedItem")
   public List<Cd> searchCds(String string);
}
\end{verbatim}

\begin{verbatim}
Listing 4.7: Flow interface of the SearchCD flow without nested interfaces
\end{verbatim}

The above definition using the shorter alternative is almost equivalent to the definition in listing 4.6. Generic transitions are defined implicitly for navigating forward and backward between the states, as well as a generic final transition from the last state to an end state. Those generic transitions are named implicitly so that \texttt{OnTransition} methods can be used to reference them. The final transition is named \texttt{Finish}. The forward and backward transitions are named \texttt{<stateNumber> next} and \texttt{<stateNumber> back} respectively. For example, if a method should be called when the user navigates from first state to second state, the method should have the \texttt{OnTransition} annotation with the following attribute: \texttt{callOnTransitions = "1 next"}. Or if a method should be called when the user navigates from the second state back to the first state, the following attribute is used: \texttt{callOnTransitions = "2 back"}. For the finishing transition, which ends the flow, the following attribute is used: \texttt{callOnTransitions = "Finish"}.

### 4.2 Definition of the Java Annotations

Table 4.1 contains a detailed explanation of all available annotations and their members.

* denotes optional members.
<table>
<thead>
<tr>
<th>Annotation</th>
<th>Member</th>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow interface annotation</td>
<td>caption*</td>
<td>String</td>
<td>Caption of the flow. This can be displayed as a title, in bread crumbs etc.</td>
</tr>
<tr>
<td></td>
<td>help*</td>
<td>String</td>
<td>Help identifier of the flow. Using this identifier, the help context can be resolved.</td>
</tr>
<tr>
<td></td>
<td>transitions</td>
<td>Transition[]</td>
<td>Transitions which can be navigated to. See Transition annotation for details.</td>
</tr>
<tr>
<td>Flow With Multiple States interface annotation</td>
<td>numberOfStates</td>
<td>int</td>
<td>Defines the number of states in the flow which are chained together with forward and backward transitions. Minimum value is 2.</td>
</tr>
<tr>
<td></td>
<td>globalTransitions*</td>
<td>Transition[]</td>
<td>Global transitions that are available in all of the states defined in this flow.</td>
</tr>
<tr>
<td>Transition</td>
<td>to</td>
<td>Class</td>
<td>References the flow interfaces where the transition will lead to. Possible values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• If another flow interface is referenced, then a subflow is started.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• If the transition refers to the itself (i.e. the same flow interface where this transition annotation is defined), the transition leads to the first state of this flow. No subflow is started.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• If <code>void.class</code> is referenced, the transition leads to an end state. I.e. this flow terminates and the parent flow continues.</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>String</td>
<td>The unique name of the transition. The name is used for referencing the transition.</td>
</tr>
</tbody>
</table>
## Annotation Member Types

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Member</th>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>arguments*</td>
<td>String[]</td>
<td>Passes an input argument to the referenced flow. The argument name must be the name of a variable that is declared in this flow. Variables are declared implicitly for each method parameter in the flow. Explicit variables can be declared using the <code>ResultVariables</code> annotation. If the referenced flow has an input parameter with the same name, this input argument will be mapped to that parameter. If the names are different, a mapping must be specified using the <code>&gt;</code> character. Examples:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- <code>arguments = &quot;cd&quot;</code> if a result variable named <code>cd</code> is defined in this flow and the referenced flow has an input parameter named &quot;cd&quot;.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- <code>arguments = &quot;cd&gt;requiredCd&quot;</code> if a result variable named <code>cd</code> is defined in this flow and the referenced flow has an input parameter named <code>requiredCd</code>. The value of <code>cd</code> will be mapped to <code>requiredCd</code>.</td>
</tr>
<tr>
<td>outcomes*</td>
<td>Outcome[]</td>
<td>Optimal member for transitions that start subflows. Outcomes can be used to redirect the control to a different flow depending on how the subflow terminated. If no outcome is specified, the subflow terminates and the parent flow continues from the state, where the subflow call was made. For example, consider a flow which displays information about a CD. A subflow is called which can eventually delete the CD. If the CD is deleted, the information about the CD should not be shown. Therefore, an outcome can be specified to end the flow if the CD is deleted. See the Outcome annotation for details.</td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 4 MODF Specifications using Java Annotations

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Member</th>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outcome</td>
<td>on</td>
<td>String</td>
<td>References a subflow transition name that leads to an end state. It may also reference a subflow call in the subflow, which has an outcome that leads to an end state. In other words, it must reference a transition of the subflow, where the transition can lead to an end state.</td>
</tr>
<tr>
<td></td>
<td>to</td>
<td>Class</td>
<td>References a flow interface. If the subflow terminates with the transition that is specified with the <code>on</code> member, this referenced flow will be called. See the <code>to</code> member of the <code>Transition</code> annotation for possible values.</td>
</tr>
<tr>
<td>OnEntry</td>
<td>inClass</td>
<td>Class</td>
<td>The class object of the type where this method resides.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The targeted method will only be called if the user navigates to the transitions defined by this member. The transitions are referenced by name.</td>
</tr>
<tr>
<td>guarded</td>
<td>boolean</td>
<td>Determines whether the result of the action should restrict the transition to take place. A guarded action must have a boolean return type. If a guarded action returns false, the transition cannot take place. For example, a guarded action can be a validator method which returns false if the transition to the next state is not allowed.</td>
<td></td>
</tr>
<tr>
<td>Outcome</td>
<td>on</td>
<td>String</td>
<td>References a subflow transition name that leads to an end state. It may also reference a subflow call which has an outcome that leads to an end state.</td>
</tr>
<tr>
<td></td>
<td>to</td>
<td>Class</td>
<td>References a flow interface. If the subflow terminates with the transition that is specified with the <code>on</code> member, this referenced flow will be called. See the <code>to</code> member of the <code>Transition</code> annotation for possible values.</td>
</tr>
</tbody>
</table>
Chapter 4 MODF Specifications using Java Annotations

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Member</th>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResultVariables</td>
<td>value</td>
<td>String</td>
<td>Used to declare variables for the result of the method. The variables can then be used as input parameters in the called flows. Examples for variable definitions:</td>
</tr>
<tr>
<td></td>
<td>method annotation</td>
<td>can be omitted</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- <em>x</em> - Defines a variable named x that references the returned object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- <em>x=f</em> - Defines a variable named x that references the $f$ field of the returned object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- <em>x=f.h</em> - Defines a variable named x that references the $f.h$ field of the returned object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- <em>x=selectedItem</em> - Defines a variable named x that references the object in the selected row of a collection. If the returned object is a list and $selectedItem$ is not used, then the variable will reference the whole collection.</td>
</tr>
</tbody>
</table>

Table 4.1: Definition of the MODF Java annotations

4.3 Steps to Define Dialog Flows

The following steps can be followed in order to specify the dialog flows systematically:

Create a package  First, a Java package should be created, which will contain the flow interfaces. Flow interfaces can be created in any package. But as a convention, they should be created in an own package. For example: `de.capgemini.cdmanagement.gui.generator`.

Define the flow interfaces  Next, the flow interfaces should be defined by creating Java interfaces and annotating them with either the `@Flow` or `@FlowWithMultipleStates` annotations. One flow interface should be created for each view in the user interface unless using the `@FlowWithMultipleStates` annotation. The flow annotations can then be chosen according to the following guidelines:
Chapter 4 MODF Specifications using Java Annotations

Annotation Usage

@Flow With Multiple States

- For displaying or filling the contents of a backing bean using multiple pages so that the users are not overwhelmed by a single large page. Validation can be added separately to each page and the user can navigate forward and backward between pages.

- For search pages: If the value of the numberOfStates attribute is set to two, the first page can show the input fields, the second page can show the search results.

@Flow

In general, the Flow annotations can be used for the other cases. Each flow interface that is annotated with the Flow annotation will have a single view. These are usually used to show a list of entries, display details of a single entry or provide forms.

Table 4.2: Guidelines for choosing between the Flow and FlowWithMultipleStates annotations

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Flow</td>
<td>Define the transitions After the follow interfaces are defined, the transitions between them can be specified. This is done by using the transitions or globalTransitions members of the flow annotations. A unique name should be given for each transition in a flow interface. Then, the flow interface of the targeted flow should be referenced using the to member of the transition annotation.</td>
</tr>
<tr>
<td>@Flow</td>
<td>Define the actions The actions should be defined that will be executed during the flows: The signature of the method, which will be called in the application logic layer, must be declared in the flow interface. The throws clauses for exceptions can be omitted because they are not necessary to resolve the methods. Then, the method is annotated with the OnEntry or OnTransition annotations and the inClass member of the annotation is defined in order to specify in which application logic layer class the method resides.</td>
</tr>
<tr>
<td>@Flow</td>
<td>Define the arguments and the subflow outcomes If a called subflow needs input parameters, the corresponding transition must be extended with the argument member in order to define how the input will be given. If the flow should be redirected depending on the outcome of the subflow, the outcome member should be defined.</td>
</tr>
</tbody>
</table>

4.4 Transformation to MODF Models

After the dialog flows are specified with the Java annotations, they can be converted into the intermediate MODF model so that it can be easily processed by tools. During the transformation to the MODF model, the created objects occasionally need to set references to objects.
which do not exist yet. For example, during the processing of the first flow interface, if a transition annotation refers to another flow, the corresponding Transition object must reference a state which is not created yet. This initialization problem can be solved with different approaches and data structures. The following solution involves iterating over the flow interfaces multiple times until all dependencies are created. Each iteration over the flow interfaces creates the objects that are needed in the next iterations.

Algorithm 4.1 Transformation of a Java-based MODF specification to a MODF model

Create model root

for all flow interfaces do
    Create the flow and the states
end for

for all flow interfaces do
    if the flow interface has the FlowWithMultipleStates annotation then
        Create the implicit transitions between the states
        Create the global transitions in each state
    else
        Create the transitions
    end if
    if transition leads to void.class then
        Create an end state
    end if
end for

for all flow interfaces do
    if a subflow outcome exists, which leads to end state then
        Create an end state
    end if
end for

for all flow interface do
    Create the subflow outcomes (both implicit and explicit)
end for

for all flow interface do
    for all annotated methods in the flow interface do
        Create the actions, the input parameters and the variables
    end for
end for

for all created transitions do
    Create the arguments
end for
Chapter 5

The Register Factory Web GUI Generator

The Register Factory Web GUI Generator is a tool which was developed during the thesis in order to generate graphical user interfaces based on method-oriented dialog flows. The Register Factory Web GUI Generator proves that MODF models can be used to generate complete user interface layers according to GUI architectures. The generated user interfaces conform to the Register Factory architecture. Therefore, the first section will give an overview of the Register Factory. Section 5.2 will explain the GUI architecture of the Register Factory applications. The following sections will cover the Register Factory Web GUI Generator.

5.1 Introduction to the Register Factory

The Register Factory was designed by “Bundesstelle für Informationstechnik des Bundesverwaltungsamtes” (BIT)[8] in cooperation with Capgemini sd&m [10] in order to provide easier implementation of registries. Registries are used to manage information about entries in a structured way, for example about foreign citizens, visa applications, driver’s licences or weapons. Cadasters are also an example of public registers showing the details of ownership and value of land.

Because most registries have similar functionalities, the Register Factory was created to help the creation of registries by providing a reference architecture as well as reusable components that can be used in each registry. The registry applications, which are developed according to the Register Factory architecture, will be called registers. Registers have a service oriented architecture. They do not provide a graphical user interface. Applications can access the registers using the services which they provide. Figure 5.1 shows an overview of the application landscape, which depicts how the registers are used by the surrounding applications and users. Apart from the registers, two types of systems are presented:

- **Business applications** contain business logic and processes. In contrast to registers, they also provide a user interface in order to present the data from the registers and allow user interaction.

- **Cross-cutting systems** provide common services for the registers and business applications, for example user management and administration [7].
Chapter 5 The Register Factory Web GUI Generator

The users can interact with the registers in two ways: using the graphical user interfaces of the business applications directly or the portal of the Register Factory. The portal provides public access to the contained business applications. Among others, the portal provides a start page which can be used to navigate to the business applications as well as a Single Sign on mechanism. Even though the portal integrates the user interfaces of the business applications, it does not replace them. The users still use the user interfaces of the business applications, even if they access them indirectly over the portal of the Register Factory. External applications can use the service gateway in order to access business applications via web services.

For this thesis, the business applications are the most relevant elements in the application landscape because the user interfaces are provided by the business applications. Since the goal of the Register Factory Web GUI Generator is the generation of user interfaces, the technical architecture of the business applications also needs to be introduced. The application landscape only shows a conceptual view of the interactions between the users and the applications in the Register Factory platform. Nevertheless, the Register Factory reference architecture also covers the technical architecture of registers and business applications, which is shown in Figure 5.2. The layers are based on a three tier architecture. In contrast to the classical three tier architectures, where the top layer is usually the presentation layer, the top layer is split into three parts: GUI layer, batch layer and service layer. Since registers do not have user interfaces, registers do not have the GUI layer. In addition to the three layers, the cross-cutting functionalities such as authorization and logging are covered with a vertical layer.

The persistence layer is responsible for accessing databases and providing functions for creation...
and management of data. The application logic layer contains the business logic. The top layers
provide the interface for communication with users and other applications. The Bachelor’s
thesis of Strauß (2010) [68] successfully captured the topic of generating the service layer for
register applications, based on source code annotations in the application logic layer. The
Register Factory Web-GUI Generator covers the generation of the GUI layer for the business
applications (the top left layer in 5.2).

In order to show the artifacts that are generated, the GUI architecture should be explained in
more detail. The Register Factory reference architecture also captures the details of the GUI
layer, which will be explained in the next section.

### 5.2 The GUI Layer of the Register Factory Applications

The GUI layer of the Register Factory applications aims to separate the presentation views and
dialog flows. The flows are specified using the Spring Web Flow’s XML-based flow definition
device. The Spring Web Flow framework is also used to manage the flow executions in
runtime and the user sessions. For the presentation aspects, JSF and Facelets are used. Before
explaining the components of the GUI layer, the used frameworks are briefly introduced in the
next section.
5.2.1 Used Frameworks

The frameworks, which are introduced in this section, are very comprehensive and cannot be covered in detail in the thesis. Nevertheless, this section gives a short overview about the frameworks since they are used by the GUI components of the Register Factory platform.

Spring

Spring [65] is an open source framework which provides a wide range of modules for the creation and management of Java components (beans), aspect-oriented programming, development of web services, security, messaging and more. The GUI layer mainly uses the Inversion of Control (IoC) mechanism of Spring, which enables beans to be centrally configured and wired together. Most of the GUI components, which will be explained in Section 5.2.2 are configured as beans and wired together via Spring.

Spring Web Flow

The Spring Web Flow framework (SWF) [66], which was already introduced in Section 2.2.2, is the module of Spring for implementing dialog flows. SWF is used to separate views, dialog flows and application logic code. The flow controller provided by SWF uses the flow definition files in order to coordinate the flow of the application. User sessions are also managed by SWF.

JavaServer Faces and Facelets

JavaServer Faces (JSF) [51] is a framework for the development of web applications. JSF follows the model-view-controller pattern and a component oriented approach for developing web applications. JSF provides components such as data tables, buttons, calendars which can be used to build a component tree. The components can use expressions and refer to beans in order to load their contents.

Facelets is a view technology for JSF, which aims to increase the reusability of JSF components. Facelets is used to render HTML pages based on the component trees built by JSF.

5.2.2 The GUI Components

The components of the GUI layer, which follow the method-view-controller pattern, are shown in Figure 5.3.

The following GUI components must be created by the user interface developers:

- Facelets components are XHTML documents which are used to specify the views using templates and component composition.
• Flow configurations are SWF XML-based flow definition files that specify the dialog flows, i.e. logical views, the transitions between the views and actions that should be executed at certain points in the flow.

• Backing beans: Views use the backing beans to read and write the data during sessions. In other words, the backing beans contain all domain data that is needed to visualize a page. Their scopes are managed by SWF. These components are actually called view models in the Register Factory terminology. Because models are used in a different meaning in the thesis, the backing bean term is used to avoid confusion.

• Application logic layer wrappers are positioned in the middle of the GUI layer and the application logic layer. As a requirement of the Register Factory architecture, the data that originates from application logic layer cannot be stored in the backing beans as is. This prevents aliasing issues and makes sure that the GUI layer cannot directly modify the contents of objects from the lower layers. The application logic layer wrappers make sure that the components in both layers operate transparently without seeing the data conversion between the layers. Therefore, the wrappers are used to map the data from the application logic layer to the data format of the backing beans.

Figure 5.3: The GUI components of the Register Factory applications and their relations
and the other way around. Furthermore, the application logic layer wrappers handle the exceptions that are thrown from the application logic layer so that internal exceptions are not visible in the user interface.

- **Controllers** are stateless components that are referenced by the flow configurations in order to access the application logic layer. The controller methods receive backing beans as parameters. They do not directly call the application logic layer code. Instead, they use the application logic layer wrappers. The returned values of the application logic layer wrappers are written back to the backing beans.

The following components are provided by the used frameworks or the existing application logic layer:

- **DispatcherServlet** is a part of the Spring MVC framework, which serves as an entry point for all web request that come to the server. It dispatches the requests to their corresponding handlers according to its configuration. When it receives a request about a flow execution, it dispatches the request to the flow controller.

- **FlowController** is the handler for the requests about flow executions. The flow controller is part of the Spring Web Flow framework. When a request arrives at the flow controller, it finds the flow configuration file that corresponds to the request and checks whether a new flow execution should be created or an existing flow execution should be continued. Depending on the request and the state of the flow execution, it resolves the view to display and evaluates expressions if needed.

- **FaceletViewHandler** is a part of the Facelets library, which is responsible for the creation of the HTML documents by facilitating JSF and facelets. When the flow controller resolves the logical view name from the flow configurations, it uses the FaceletViewHandler in order to construct the concrete HTML documents from the logical view name.

- **Application logic layer beans** contain the data which is used in the application logic layer. The application logic layer beans are not allowed to be given to the GUI layer.

- **Business services** are Spring beans in the application logic layers that provide the business logic that is used by the user interface. Business services operate using the application logic layer beans.

### 5.2.3 Interactions Between the GUI Components

To demonstrate the interactions between the components, consider a simple flow which starts by displaying a list of CDs. A client uses a regular browser and requests a web page which corresponds to a flow. The request arrives at the Spring DispatcherServlet which is the central component that dispatches all incoming requests to their handlers. In this case, the DispatcherServlet dispatches the request to the FlowController.

The FlowController finds the flow configuration file that corresponds to the request and checks whether a new flow execution should be created or an existing flow execution should be continued. Because the user requested a new flow, the FlowController will start a new flow.
execution and create a new backing bean for the flow. The FlowController then resolves the view to display and evaluates expressions if needed.

In this case, let’s assume that the flow configuration contains an expression in order to load a list of CDs using the view controller. The FlowController calls the corresponding view controller to load the list of CDs and gives the created backing bean as a method argument. The controller calls the corresponding application logic layer wrapper in order to load the list of CDs. The application logic layer wrapper calls the actual business method in the application logic layer bean, converts the results into the data format of the GUI layer and returns the converted data to the controller. The controller stores the returned data in the backing bean.

After the evaluation of the controller call, the list of CDs are stored in the backing bean of the flow. FlowController uses the FaceletViewHandler in order to create a concrete view using the logical view name that was resolved by the flow configuration. Facelets and JSF are used to render the HTML document, which is then returned as a response to the client.

### 5.3 Generation of the GUI Components

Creating the GUI layer with regard to the Register Factory architecture consists of many steps. Even though most steps may be easy, it is a time consuming and error-prone process. The binding to the GUI is done with a configuration file which is found automatically by Spring. The existing implementation is not modified. The following elements must be created:

- A webapp directory which contains the web application specific files. E.g. configuration files, flow definitions, Facelets templates and view definitions, images, stylesheets.
- Configuration files in the spring directory for configuring the Spring Web Flow framework and defining new Spring beans. A Spring bean must be defined for each view controller and application logic layer wrapper.
- Java resources which contain the source code for the Spring beans. In particular, application logic layer wrappers, backing beans, new data types for backing beans and view controllers need to be created.

The existing Java source files are not modified at all. The new types, which are created in new packages, are wired to the application using the Spring bean definitions.

When the generation of Web-GUIs is mentioned, one could imagine the generation of basic HTML pages. One can see that this is not the case, although the complexity of the GUI elements is not visible from the above explanations. The Web-GUI does not only consist of simple presentation aspects, but rather a complete infrastructure with regard to the Register Factory architecture.

The Register Factory Web-GUI Generator uses the MODF models in order to generate the following artifacts, which will be explained later in more detail:

- An application logic layer wrapper is generated for each bean in the application logic layer, which is used by the MODF models.
• A **controller** and a **backing bean** is generated for each flow in the MODF model.

• **Backing bean data types** are generated which are used in the backing beans.

• A SWF specific **flow definition file** (XML) is generated for each flow in the MODF model.

• Various **view widgets** are generated for each action (method) in the MODF model, e.g. data tables for displaying the result of the methods.

• **View definitions** are generated for each state in the MODF model.

• **Bean definition files** (XML) are generated which contain bean definitions for the generated Spring beans, i.e. for controllers and application logic layer wrappers.

• An **exception type** is generated, which is used by the generated beans. The Register Factory Web-GUI Generator also allows the binding of an existing exception type.

The Register Factory Web-GUI Generator also includes a component which transforms annotation based MODF specifications into MODF models. In order to show how the generated artifacts relate to the flow interfaces (annotated Java interfaces according to 4, Figure 5.4 visualizes the relations between them.

The following sections presents the details about the generation.

### 5.3.1 Generation of the Backing Bean Field Names

The backing beans contain the data that is needed to display the views of a flow. In order to do that, a backing bean field is generated for each parameter and return type of the methods in the corresponding flow. For example, let’s say that a flow contains a method with the following signature: `List<Cd> searchCd(String text, int limit)`. The generated backing bean of the corresponding flow contains three fields: one field to store the result of the method, two fields to store the `text` and `limit` parameters. Because the flows can contain multiple methods, the naming of the generated backing bean fields is problematic. In particular, the following problems should be solved:

1. How are the backing bean result field names generated if multiple methods with the same name exist (i.e. if a method is overloaded)?

2. How are the backing bean parameter field names generated if multiple methods have the same parameter name? For example, if the flow contains the methods `deleteCd(String isbn)` and `purchaseCd(String isbn)`, then two fields are needed to contain separate `isbn` parameters.

In both cases, the backing bean field names must be extended so that they become unique. However, extending the backing bean field names only on demand (only if conflicts exist) would make the field names unstable in the following sense. Imagine that the flow contains a single method `deleteCd(String isbn)`. Because there is only one method, the generator creates a backing bean field named `isbn`. After a while, a new method `purchaseCd(String isbn)` is inserted into the flow. Now, a conflict exists because two methods have the same parameter
name. The generator will have to extend the field names, for example the new field names could be isbn-deleteCd and isbn-purchaseCd. In this case, any (manually created) parts of the user interface, which uses the isbn field, would not function without being updated accordingly. As an alternative, the generator can leave the old part unchanged and only extend the new inserted parts. However, this would make the field names inconsistent. Furthermore, the generator would have to keep track of history of the old generation sessions. As a consequence, the on demand extension of file names was not used. Each generation should result in the same field names even if no conflict exists.

In order to solve the problems that are mentioned above, two restrictions were applied to the MODF specifications. The first restriction disallows the overloading of methods within a flow. I.e. if multiple methods are declared within a flow, the methods must have different
names. The overloading of methods is restricted in order to simply the generated artifacts. In the generated backing beans, result fields are generated which store the method results. The generated result fields are named after the method, whose result they contain. For example, if the flow contains a searchCd(String text, String author) method, then the generated result field is called resultSearchCd. The generator displays an error message if a conflicting parameter name is chosen, e.g. searchCd(String resultSearchCd) is disallowed. If method overloading was allowed, then the name of the result fields would have to include the parameter types as well, for example resultSearchCd-String-String. For each method that has a non-null return type, a view widget is generated which is able to display the result of the method. The same naming problem also applies to the generated view widgets. As a consequence of the restriction of method overloading, the generated names remain readable.

The second restriction concerns the parameter names. If multiple methods are declared within a flow, all parameter names must be unique within the flow. For example if there is a deleteCd(String isbn) method, there can’t be another method which has a parameter named isbn. This restriction is applied because a parameter field is generated in the backing bean for each parameter. To avoid this restriction, the field names would have to be automatically extended by appending the method name to the field name. Because overloading is restricted, the automatic extension by method names would always yield a unique field name. Nevertheless, the resulting parameter names would be too long and unreadable.

Even though the method signatures in the flows are used to resolve the methods, which are called in the application logic layer, the parameter names are not necessary to resolve the methods, because a fully qualified method signature includes only the method name and the parameter types. This means, if the flow contains two methods, which normally have the same parameter names in its signature in the application logic layer, the parameter names can be changed in the flow without modifying the signature in the application logic layer. Therefore, this restriction could be safely applied in order to simplify the generated backing bean field names. As an alternative solution to this restriction, a manual mapping from parameter names to field names could be defined. Nevertheless, the creation of a mapping does not comply with the requirements about simplicity because it increases the specification overhead.

5.3.2 Generation of the Backing Bean Data Types

In order to prevent aliasing issues, the Register Factory architecture restricts the referencing of objects from the GUI layer to the application logic layer. The application logic layer wrappers have to convert the data between application logic layer beans and the backing beans. In general, the data is copied during the conversion so that the GUI layer cannot directly modify the data from the lower layers. In other words, the data is transferred between the GUI and the application logic layers by value instead of by reference.

When the data is transferred from the application logic layer to the GUI layer, it is stored in the backing beans so that it can be visualized and modified by the user interface. One approach would be to clone the data before passing it to the GUI layer. In this case, the backing beans can use the same types that are used in the application logic layer. However, the solution is not that simple because every type that is used in the backing beans must be transitively serializable, i.e. they must implement the Serializable interface. Because the types that are
used in the application logic layer are not necessarily *serializable*, they cannot be used in the GUI layer. Furthermore, the Register Factory conventions state that types in the application logic layer cannot be used in the GUI layer even if they are serializable.

Since the types of application logic layer cannot be used in the GUI layer, one cannot simply clone the data from the application logic layer and pass it to the GUI layer. A corresponding *backing bean data type* must be generated for each type which must be stored in a backing bean, with the exception of immutable and serializable types. In general, if a type in the application logic layer is immutable and serializable, there is no need to generate a corresponding backing bean data type. Since immutable types cannot be modified after they are created, aliasing would not be an issue. Nevertheless, as a convention, the types in the application logic layer cannot be used in the GUI layer, even if they are immutable and serializable. Furthermore, determining the immutability of a random type exceeds the context of the thesis. Therefore, a simpler solution was agreed upon:

- The serializable types, which reside in the *java* package (e.g. *java.lang.String*, *java.util.Date*), can be used in the backing beans.
- Primitive types can be used in the backing beans because they are implicitly serializable and do not cause aliasing problems.
- For any other type, a corresponding backing bean data type is generated. The backing bean data type can be seen as a serializable copy of its corresponding type.

Listing 5.1 shows an example of the generated backing bean types. Let’s assume that a flow contains a method which returns a *User* object. Then the backing beans data types are generated as seen on the right side of Listing 5.1. The *User* class contains a *name* field of the type *String* and an *address* field of the type *Address*. Because *String* is a serializable type in the *java.lang* package, it can be used in the generated *UserGuiObject* backing bean data type. The *field* cannot be used in *UserGuiObject* because *Address* is not in the *java* package. Therefore the same generation process is applied to *Address* and its fields recursively. Because both fields of the *Address* class are either primitive or in the *java* package, no additional backing bean data type is created other than the *UserGuiObject* and *AddressGuiObject* classes.

```java
package core;
package gui;
import java.io.Serializable;

public class User {
    String name;
    Address address;
}

public class UserGuiObject implements Serializable {
    String name;
    AddressGuiObject address;
}

public class Address {
    String street;
    int streetNumber;
}

public class AddressGuiObject implements Serializable {
    String street;
    int streetNumber;
}
```

Listing 5.1: Application logic layer beans (left) and the corresponding backing bean data types (right)
The GuiObject suffix was appended to the type names in order to demonstrate the differences in a clear way. The suffix could be omitted because the types are contained in different packages. However, the application logic layer wrappers have to use both classes for conversion purposes. If the suffix is omitted, application logic layer wrappers cannot import both types at the same time. Instead wrappers have to use fully qualified type names in order to distinguish between the types. In order to avoid the usage of fully qualified type names, the Register Factory Web-GUI generator appends a suffix to the generated backing bean data types. Among others, the suffix can be configured using preferences.

5.3.3 Mapping of Collections

As mentioned in the previous section, the data from the application logic layer has to be mapped to the data types of the GUI layer. A special case applies to the collections. Although the collection types, which are provided by the Java API, can be used and stored in the GUI layer, the content of the collections must still be converted. The Register Factory Web-GUI Generator is able to handle lists and arrays. The handling of lists proves that collections can be handled. Nevertheless, other collection types are not supported in order to reduce the implementation effort for the generator.

Depending on the type of the data that is transferred between the layers, the application logic layer wrappers map the value in the following way:

- For non-generic, non-collection, serializable types from the java package: No mapping is performed and the object reference is passed as is (e.g. java.lang.String or java.util.Date).
- For non-generic types that are not in the java package: The type is mapped to the backing bean data type which is generated for that type.
- For generic types that are not a list: No mapping is performed. Generic types are not supported in order to reduce the complexity of the generator. The generation process is terminated with a corresponding error message.
- For collection types that are not lists (types that implement java.util.Collection but not java.util.List): No mapping is performed. Those types are not supported in order to reduce the complexity of the generator. The generation process is terminated with a corresponding error message.
- For lists that do not have a generic parameter: No mapping is performed because the generator can not infer how to map the members of the list. The generation process is terminated with a corresponding error message.
- For lists (types that implement java.util.List) that have a generic parameter: A new ArrayList is created. Each member of the list is mapped using the above rules and added to the array list. Nested lists are not supported (e.g. List<List<String>>) in order to reduce the complexity of the generator. If nested collections are needed, wrapper classes can be created which contain the inner collections.
For arrays: An new array with the same size is created. Each member of the array is mapped using the above rules and added to the new array. Arrays with multiple dimensions are not supported.

### 5.3.4 Binding to the Application Logic Layer

Because the actions, which are defined in the MODF models, reference the application logic layer methods, the generated elements must communicate with the application logic layer. The generated elements must be able to instantiate the referenced class in order to call the methods. For this purpose, the Register Factory Web GUI Generator uses the dependency injection mechanism provided by Spring, as mentioned in Section 3.2.2. The application logic layer classes, which are referenced from the user interfaces, are declared as Spring beans according to the Register Factory reference architecture. This means that the application context already has bean definitions which refer to the needed application logic classes. Listing 5.2 shows an example of a bean definition.

```xml
<beans>
  <bean id="cdManagement"
       class="de.capgemini.cdmanagement.core.cdmanagement.impl.CdManagementImpl"/>
</beans>
```

Listing 5.2: Definition of an application logic layer Spring bean

Let’s assume that a MODF model refers to the interface of the `CdManagementImpl` class, for example `de.capgemini.usermanagement.core.cdmanagement.CdManagement`. The goal is to use the above bean which has the identifier `cdManagement`. However, there is no direct mapping between the interface type and the bean identifier. In order to resolve the bean identifier from a referenced type name, the following options would be possible:

- One may require that the application logic layer beans must be named according to a naming convention, which can be used to resolve the bean name. For example, one may require that the bean identifier must be same as the interface name, however starting with a lower letter. This is the case in the above bean definition. The identifier is `cdManagement` and the interface name is `CdManagement`. However, this requirement is too restrictive.

- One may provide an explicit mapping from the interface names to the bean names. This may be done with an additional annotation in the methods of the Java-based MODF specifications. However, the same bean identifier would need to be specified in each annotated method. As an alternative, an additional mapping file can be created which relates the type names to bean identifiers. Nevertheless, this option has a configuration overhead.

- Last but not least, the problem can be solved in a simple way using the `autowiring by type` mechanism of Spring. Given a type, Spring can automatically find “the” existing bean, which has the given type. As a prerequisite, Spring must find exactly one bean matching the type. Otherwise, the bean cannot be created.
Chapter 5 The Register Factory Web GUI Generator

In order to keep it simple, the last approach is used and it is assumed that the application logic layers will not have multiple beans referring to the same service. If this is the case, the problem can be solved by a slight modification on the generated bean definitions. Autowiring works in the following way. As Listing 5.3 shows, the generated application logic layer wrapper classes contain an autowired field that refers to the type in the application logic layer and corresponding getter/setter methods.

```java
public class CdManagementAwkWrapperImpl implements CdManagementAwkWrapper {
    @Autowired
    private CdManagement cdManagement;

    public CdManagement getCdManagement() {
        return cdManagement;
    }

    public void setCdManagement(CdManagement cdManagement) {
        this.cdManagement = cdManagement;
    }
}
```

Listing 5.3: Example of a generated application logic layer wrapper

The Register Factory Web GUI Generator also generates a bean definition for each of the generated application logic layer wrappers. Listing 5.4 shows the generated bean definition, which is autowired by type. As a consequence, whenever the CdManagementAwkWrapper bean is instantiated, all of its fields, which are annotated with the Autowired annotation, will be autowired by type. For example, Spring will find that the cdManagement field should be autowired and set the value of the field to the bean with the identifier cdManagement (in Listing 5.2), because it matches the CdManagement type.

```xml
<beans default-autowire="byType">
    <bean id="cdManagementAwkWrapper" class="de.capgemini.cdmanagement.gui.awkwrapper.CdManagementAwkWrapperImpl"/>
</beans>
```

Listing 5.4: Definition of an application logic layer wrapper Spring bean

5.4 Implementation of the Register Factory Web GUI Generator

The Register Factory Web GUI Generator is implemented as an Eclipse [25] plug-in. The following three Eclipse plug-ins were implemented:

- **MODF Models:** The MODF meta-model is specified in this plug-in using an EMF Ecore [27] model. EMF was used to generate the Java code of the interfaces and classes, which represent the MODF meta-model. The MODF models plug-in also integrates a view to Eclipse, which can visualize the MODF models using a tree widget (Figure 5.5).
• **MODF Annotations:** The *MODF annotations* plug-in defines the Java annotations which are used to define the flow interfaces. Applications, in which the MODF Java annotations will be used, must add the jar-file of the *annotations* plug-in to its classpath so that the annotations can be imported. The *MODF annotations* plug-in also includes a *flow builder* component, which transforms Java-based MODF specifications into MODF models. Therefore, the *MODF models* plug-in is a dependency.

• **Register Factory Web GUI Generator (RF):** The *RF* plug-in contains the Register Factory workspace specific parts of the implementation. It integrates property pages to the Eclipse workspace to configure the generation as well as an action to start the generation. The generation can be started using the context menu of a Java project. When the generation is started, the *RF* plug-in first uses the *MODF annotations* plug-in to transform the flow interfaces into a MODF model. Then, it generates the user interface artifacts based on the MODF model and the configuration of the plug-in. Both the *MODF Models* and the *MODF Annotations* plug-ins are dependencies.

![MODF Model](image)

**Figure 5.5:** The depicted Eclipse view uses a tree widget in order to visualize the MODF model, which is transformed from the flow interfaces.
Chapter 5 The Register Factory Web GUI Generator

The features of the implementation can be summarized as:

- Integration with Eclipse
- Configuration using the Eclipse properties windows
- Progress monitor during the generation process
- Result summary after the generation process (generated artifacts, warnings, errors)
- Logging system for the generation (detailed explanations of the warnings and errors)
- Visualization of the MODF models using a tree widget
- Usage of the code generation templates
- Automatic formatting of the generated source code using the users’ Eclipse formatting preferences
- Automatic generation of Javadocs based on the Javadocs of the referenced methods

The preferences pages can be used to configure the following aspects:

- Names, prefixes and suffixes of the generated packages and types
- Location of the generated flow, view and bean definition files
- Overwriting of the generated artifacts
- Custom exception type which can be used by the generated types
- Custom Facelets template which can be integrated to the generated views

The following frameworks and plug-ins were used:

- Java Development Tools (JDT) [26]: To process and generate Java elements, such as classes, interfaces, annotations, etc.
- Eclipse Modeling Framework (EMF) [27]: To define the MODF meta-model and generate Java code of the interfaces and classes which represent the meta-model. EMF was also used in order to develop the Eclipse view, which displays the MODF models, in an efficient way. Because of EMF’s interoperability with GMF [28], it also makes the future implementation of graphical editors easier.
- Apache Velocity: To generate source code and other files using templates.
- Apache Log4j: To implement the logging system.
- Apache Commons: In particular for advanced collection types such as bidirectional maps.
- Eclipse Core UI plug-in: To add menu actions to Eclipse via extension points.
- Eclipse Core Runtime plug-in: To use plug-in properties and to add progress monitors to Eclipse.
- Eclipse Resources plug-in: To navigate through the Eclipse workspace for generation purposes.
In addition, all three plug-ins include relevant utility methods in order to increase modularity. In total, the implementation contains around 10500 lines, 4100 statements and 70 classes with a comment to line ratio of 37%, including the Javadocs and excluding the files, which were generated by EMF.
Chapter 6

Evaluation

Various dialog flows were created for the evaluation of the MODF specifications and the Register Factory Web GUI generator. The evaluation was not only done by the author, but also by Capgemini sd&m [10]. In order to demonstrate the generated views, the example dialog flows will be used, which were explained in Section 4.1. Then, the feedback from Capgemini sd&m will be presented.

Although the focus of the Register Factory Web GUI Generator is not the generation of views, the demonstration of the generated views is interesting because it is the only part of the user interfaces that is directly visible to the users. Figure 6.1 gives an overview of the existing classes which were referenced by the example dialog flows in Section 4.1.

![Diagram of classes used by example scenario](image)

Figure 6.1: Overview of the classes used by the example scenario

Figure 6.2 shows a possible flow execution using screenshots. The flow execution starts from the ListCds flow on the top. Note that the Purchase, Details and Delete transitions are available for each entry because their corresponding transitions in Listing 4.3 are referring to a single selected item, which was specified with the selectedItem keyword. The Search transition is available globally. In the SearchCd flow, a single input field called string
can be seen, where the user can enter a query. The field is generated because of the string parameter of the searchCd method in the SearchCd flow. The search results are shown in the second state of the SearchCd flow. Again, the Details transition is available for each item. Other transitions are available globally. The ShowCdDetails flow displays all fields of the CdDetails object that is returned by the getCdDetails method in the ShowCdDetails flow. The DeleteCd flow shows a confirmation view.

The view for the PurchaseCd flow is depicted separately in Figure 6.3. Six input fields are displayed. The purchaseCd method in the PurchaseCd flow has two parameters called cd and billingData. The first two input fields are generated from the cd parameter. Since
a Cd object is passed as an input argument to the flow, the fields are already filled when the flow starts. Other fields are generated because of the billingData parameter. Not only the fields are displayed, which are direct children of the class, but also the indirect ones. The BillingData class has references the Address class, which has two fields of type String. Input/output fields are generated for all indirect fields recursively. Per default, the recursion is terminated when a recursion level of three is reached or a type is found which can be displayed directly (e.g. String, Date or primitive types).

![PurchaseCdFlow](image)

Figure 6.3: A screenshot of the generated view of the PurchaseCd flow

The generated views form a good starting point for the design of the views, but the following modifications may be needed:

- Input/output fields may need to be removed or disabled. E.g. in the DeleteCd flow, the input field should not exist. Instead, a confirmation message should be added. In the PurchaseCd flow, the input fields about the cd should be removed or disabled because the users are not supposed to modify their content.

- Labels should be renamed. Currently the generated labels are derived from field names. It is also possible to convert the camel case notation of the fields into more readable labels. E.g. songTexts can be converted to Song texts. Optimally, the generator should create properties files for the labels so that they can easily modified and internationalized without changing the view definitions.

- The layout may need to be customized. Currently the generator uses a simple Facelets template for initial layouting. In addition, the generator allows the configuration of a custom Facelets template. However, if a custom template is used, the generated widgets may need to be modified to match the style sheets (CSS) [5] that are defined in the template.
Chapter 6 Evaluation

Figure 6.4 shows the view of the ListCds flow using a custom Facelets template. The template, the referenced stylesheets and the images were provided by Capgemini sd&m. Other than configuring the template in the generator properties, no other manual changes were needed.

As mentioned, the Register Factory Web GUI Generator was also evaluated by Capgemini sd&m [10]. The generator was used in a project which was developed for the “Bundesverwaltungsamt” [9]. The estimated scope of the mentioned project is 300 man-days. The created user interface consists of six flows, where two of them were specified by MODF specifications using Java annotations. Based on the specifications, the following user interface artifacts were generated with regard to the Register Factory architecture: Spring Web Flow’s XML-based flow definition files, simple view definition and widget files (JSF/Facelets), Spring bean definition files in XML format, as well as Java interfaces and classes of controllers, backing beans and application logic wrappers. The generated artifacts were integrated into the project after manual customization.

During the implementation of the Register Factory Web GUI Generator, change requests were issued by Capgemini sd&m due to the modifications in the Register Factory user interface architecture. The Register Factory Web GUI Generator could be modified in short time in order to reflect the changes in the Register Factory user interface architecture, which shows the flexibility of the generator and the possibility of generating implementations with regard to other architectures, as mentioned in Section 3.2.1. In order to increase the flexibility further, the usage of code generation templates are suggested, which can be configured without changing the generator implementation.

The following aspects of the Register Factory Web GUI Generator were rated as very successful by Capgemini sd&m:

- Generation of the user interface artifacts with regard to the Register Factory architecture
- Integration to the Eclipse platform
- Configuration possibilities
- Monitoring of the generation process

Figure 6.4: A screenshot of the view of the ListCds flow using a custom Facelets template.
Chapter 6 Evaluation

• Report about the generated artifacts
• Logging system for the generation process

According to the feedback from Capgemini sd&m, the definition of flows using Java annotations is not intuitive from the perspective of the user interface developers. The usage of annotated interfaces and methods for flow definition purposes is unusual in the sense that the Java language is being used for specification purposes instead of programming. The problem can be overcome by supporting the graphical definition of MODF models, as proposed in Section 3.2.3.

As a conclusion, the evaluation report by Capgemini sd&m states that the Register Factory Web GUI Generator increases the efficiency of the user interface creation process to a great degree, which is affected positively depending on the:

• simplicity of the flows,
• complexity of the generated user interface artifacts,
• familiarity of the user interface developer with the specification technique.

According to Capgemini sd&m, the Register Factory Web GUI Generator can reduce the user interface development time by 20% to 50%.
Chapter 7

Conclusion

In this thesis, Method-Oriented Dialog Flows (MODF) were presented, which can be used to specify modular dialog models in a practical way. As explained in Section 3, Method-Oriented Dialog Flows fulfill the requirements which are needed to specify user interfaces for Register Factory applications. Nevertheless, The MODF meta-model is designed in a platform independent way using UML so that MODF models can be applied to applications other than the Register Factory applications.

The implementation and the evaluation of the Register Factory Web GUI Generator showed that MODF models can be successfully used to generate user interfaces with regard to the Register Factory reference architecture.

The Register Factory Web GUI Generator can be modified in short time in order to reflect the changes in the target architecture, which shows the flexibility of the generator and the possibility of generating implementations with regard to other architectures, as mentioned in Section 3.2.1. The flexibility was shown by the successful modifications to the Register Factory Web GUI Generator in reasonable time, which were done according to the change requests which were issued by Capgemini sd&m due to the modifications in the Register Factory user interface architecture.

Section 4 demonstrated an approach to specify dialog flows using Java language elements such as interfaces and annotations. The Register Factory Web GUI Generator is able to use the Java-based MODF specifications for generation purposes. The evaluation of the Java-based specification language showed that it was not an intuitive specification approach for user interface designers, although it still reduced the time and effort for the user interface creation process. The process of user interface creation can be optimized further if MODF models can be specified using a more suitable notation, such as a graphical notation, as proposed in Section 3.2.3.

The Java-based user interface specification language and the Register Factory Web GUI Generator were not only evaluated by the author, but also by Capgemini sd&m in a customer project. The evaluation resulted in improvement ideas which can be useful for the future work.
Chapter 7 Conclusion

7.1 Future Work

The following ideas can serve as a starting point for future research.

Expressiveness The MODF models can describe the dialog flows of user interfaces to a great degree. However, some situations cannot be expressed, for example deciding the next flow purely based on a method result. In such cases, the generated user interface can be modified manually in order to resolve the issue. However, if it is desirable to be able to express more complex dialog flows, the following ideas can be used:

1. The MODF meta-model can be extended in order to increase its expressiveness. The following features can be added in order to enhance the meta model: decision states, handling of flow output data, flow inheritance and exception handling. The details of the suggested enhancements can be found in dialog model references in Section 2.2.

2. The MODF meta-model can be replaced with an existing dialog modeling notation. Method-orientation and the generation of user interfaces, which were introduced in Section 3.1, can be applied to other dialog models as well. This way, the MODF characteristics can be combined with the expressiveness of other dialog modeling notations. Possible candidates are the notations which are used in the Spring Web Flow framework (SWF) [66], the Dialog Flow Notation (DFN) [4] and the Guilet Dialog Model (GDM) [63]. Since SWF and DFN are specific for web applications, a dialect of the dialog modeling notation may need to be created in order to support platform independency. GDM is platform independent in its nature. Due to the complexity of its notation, GDM may have a higher learning curve before dialog flows can be specified practically.

Generator extensions The user interfaces of Register Factory applications, which can be generated by the Register Factory Web GUI Generator, are based on the Spring Web Flow and the Spring MVC frameworks. As mentioned in Section 3.2.1, MODF was designed to allow generation of other user interface implementations. Therefore, other user interface generators can be developed in order to generate user interfaces for other platforms such as ASP.NET MVC [48] and JBoss Seam [38].

Web services Last but not least, a very interesting topic is the handling of web service calls within MODF specifications. The Register Factory Web GUI Generator is only able to handle the calls to methods which reside in an existing Java application. If a user interface generator can handle web service calls in MODF specifications, the requirements for the underlying application are completely eliminated. Instead of an existing application, a formal web service description (e.g. a WSDL document [71]) would be needed. If this idea is combined with a web service generator, as covered in the Bachelor’s Thesis of Strauß (2010) [68], user interfaces can be generated for applications that are implemented in a completely different platform. For example, web services can be generated for a Java application and a C++/QT-based user interface [13] can be generated based on a MODF specification and web service description. The solution would bridge the heterogeneities between the application logic layers and user interfaces.
Bibliography

http://www.uiml.org/.

[2] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, Isabelle Provot, and  
Jean Vanderdonckt. A model-based approach to presentation: A continuum from task  

[3] Peter Bojanic. The Joy of XUL.  

multi-channel web applications. In COMPSAC, pages 161–168. IEEE Computer Society,  
2006.


http://www.bit.bund.de/.


[12] Larry Constantine. The emperor has no clothes: Naked objects meet the interface.  


Bibliography


Bibliography

[37] Red Hat. jBPM process definition language (JPDL).


http://download.oracle.com/javase/tutorial/uiswing/.


Bibliography


Date of access to the referenced URLs: May 10, 2011