Gottfried-Wilhelm-Leibniz-Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Concept and Implementation for Integrating User Interface Descriptions into BPEL Processes

Masterarbeit

im Studiengang Informatik

von

Michael Gutbier

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr. Udo Lipeck
Betreuer: Dipl.-Wirt.-Inform. Daniel Lübke

Hannover, 18. April 2007

Abstract

The Business Process Execution Language (BPEL) is becoming more and more important for companies which want to automatize their workflows. Beyond automation, real-world business processes usually involve human user interaction. However, the recent BPEL specification lacks support for that. This thesis presents an extension-based approach for user interaction in BPEL processes. A new BPEL activity, which allows the definition of flexible tasks—intended for the execution by users—is defined. On this basis, a user interface for the interactive execution of tasks, which connects with BPEL processes through a mediating Web service, is developed.

Zusammenfassung

Die Business Process Execution Language (BPEL) wird für Unternehmen, die ihre Arbeitsabläufe automatisieren wollen, zunehmend wichtiger. Jenseits der Automation erfordern reale Geschäftsprozesse üblicherweise Benutzerinteraktion. Die aktuelle BPEL-Spezifikation unterstützt dies aber nicht. Diese Arbeit stellt einen erweiterungs-basierten Ansatz für Benutzerinteraktion in BPEL-Prozessen vor. Eine neue BPEL-Activity, welche die Definition flexibler Aufgaben — vorgesehen für die Bearbeitung durch Benutzer — ermöglicht, wird definiert. Auf dieser Grundlage wird eine Benutzerschnittstelle für die interaktive Erledigung von Aufgaben entwickelt, welche sich über einen vermittelnden Web-Service mit BPEL-Prozessen verbindet.

Danksagung

Ich danke Herrn Daniel Lübke für die vorbildliche und stets hervorragende Betreuung dieser Masterarbeit.

Contents

1 Introduction **4**

1.1 Motivation 4

1.2 Problem Description 4

1.3 Structure 5

2 Foundations **6**

2.1 Service-oriented Architectures 6

2.2 Web Services 7

2.3 Business Process Execution Language (BPEL) 7

2.3.1 BPEL Execution Engines 8

2.4 XML Schema 9

3 A Concept for User Interaction in BPEL **11**

3.1 Requirements 11

3.1.1 User Resolution 14

3.2 An Extension to BPEL: U-BPEL 15

3.2.1 Definition of People Links 16

3.2.2 Task Types 18

3.2.3 Conversion from U-BPEL to BPEL 20

3.3 Generating the User Interface 21

3.3.1 Traversing Schema Definitions 22

4 Architecture **25**

4.1 BPEL Execution Engine 26

4.2 User Manager 26

4.2.1 Concurrency 27

4.3 User Client 27

4.4 Interaction and Temporal Behavior 28

5 Implementation of a Prototype **30**
5.1 U-BPEL Conversion 30
5.1.1 Prerequisites 31
5.1.2 Transformation 32
5.1.3 Execution of the User Activity 33
5.2 User Manager 34
5.2.1 Operation Interface 35
5.2.2 Program Structure 35
5.3 User Client 36
5.3.1 GUI Generation 37

6 Conclusion and Outlook **40**
6.1 Summary 40
6.2 Conclusion 40
6.3 Outlook 41

A Schema Definitions **47**
A.1 U-BPEL 47
A.2 User Manager 49

B WSDL Definitions **55**
B.1 User Manager 55

C Class Diagrams **57**

Acronyms

BPEL Business Process Execution Language

SOA Service-Oriented Architecture

SOAP originally: Simple Object Access Protocol, recently not longer an acronym

UDDI Universal Description Discovery and Integration

WS Web Service (in acronyms as WSBPEL or WSDL)

WSDL Web Services Description Language

XML Extensible Markup Language

XPath XML Path Language

XSL Extensible Stylesheet Language

XSLT XSL Transformations

Chapter 1

Introduction

1.1 Motivation

The paradigm of Service-oriented Architectures (SOA) is one of the major trends in the development of business software. With the technology of Web services, a flexible and standardized implementation of SOA is available. Web services can be understood as components of software that are remotely callable over a network.

Web services can be composed to build a larger and distributed software system. To facilitate the development of complex workflows, several special languages were developed in the past.

In the year 2002 the Business Process Execution Language (BPEL) was conjointly developed by big software companies to supplant two of their workflow languages: Web Services Flow Language (WSFL) and XLANG. Intended as a successor of these two languages, BPEL combines aspects of both languages and therefore has the potential to become *the* predominant workflow language. In the past years the Organization for the Advancement of Structured Information Standards (OASIS) worked on a new version of BPEL until the committee specification of BPEL 2.0 was released early this year.

However, BPEL was not designed with user interaction in mind. BPEL processes rather execute automatically.

1.2 Problem Description

The objective of this thesis is to develop a concept that combines user interaction and BPEL; after that the concept has to be verified by an implementation.

Combining user interaction with BPEL means that a user interface must be created during the execution of a BPEL process. For this, BPEL has to be extended by a mechanism that allows for easily adding commands for user interaction to a BPEL process.

Each time when such a command for user interaction is reached in a process, one specific user has to be notified so that he can react and perform a task. Once a process reaches a point at which a human user has to act, the question arises which person has to fullfill the task.

Hence, a job message has to be sent to a responsible and suitable person.

1.3 Structure

The thesis has the following structure. Chapter 2 gives a summary of the technological foundations of this work. These encompass Web services, BPEL, and XML Schema. In chapter 3 the concept is described. This includes the specification of the BPEL extension and a method for the user interface generation. Chapter 4 discusses the architecture of the solution, which involves component distribution and the interaction between the components. Finally, chapter 5 shows a prototypical implementation

Chapter 2

Foundations

To understand the results of this thesis certain foundations have to be understood. This chapter introduces to those foundations. One important concept are Service-oriented architectures—actually the context of BPEL and Web services as well. Other vital technologies are XML, XSLT, and XPath.

2.1 Service-oriented Architectures

The idea of Service-oriented architectures (SOA) is one of the latest in the field of software engineering. There are numerous definitions of SOA. According to Dostal et al [DJMZ05] a "SOA is a system architecture that represents multiple, different, and possibly incompatible methods or applications as reusable and openly accessable services, to allow a use that is independent of platform and programming language" (translation by me).

OASIS (the Organization for the Advancement of Structured Information Standards) defines SOA as "a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains." [MBM06]

This indicates the abstractness of the SOA concept. SOA is neither a particular technology nor a standard but an abstract concept that is targeted to make software systems more flexible and reusable. A SOA has a foundation of principles:

- **Simplicity.** Services can be simply reused in different environments.
- **Security** can be accomplished by means of cryptographic solutions. This is a broad field and therefore it is not covered here.
- **Standards.** It is crucial that service consumers understand the service definitions and interfaces by an unknown provider.
- **Distributedness.** It does not matter where a service runs. Every service can be used by invokation over a network.
- **Loose coupling.** A loosely coupled software system is composed of services each of which is found and called at runtime through an implementation-independent interface.

Services are not requested until they are needed. This minimizes dependency among components.

- **Discoverability.** The interface of a service has to be described in a standardized manner so that the service can be found via a directory mechanism.
- **Process-oriented.** A set of services is usually composed to form a larger unit which is commonly viewed as a process.

Because SOA is a paradigm, a concrete implementation is needed to make use of the SOA principles. One possible and popular implementation of SOA is described in the next section.

2.2 Web Services

Today Web services are one of—if not the–most important implementations of a SOA. Web service descriptions,

One good definition of Web services is by the W3C [W3C]:

> *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards [BHM+04].*

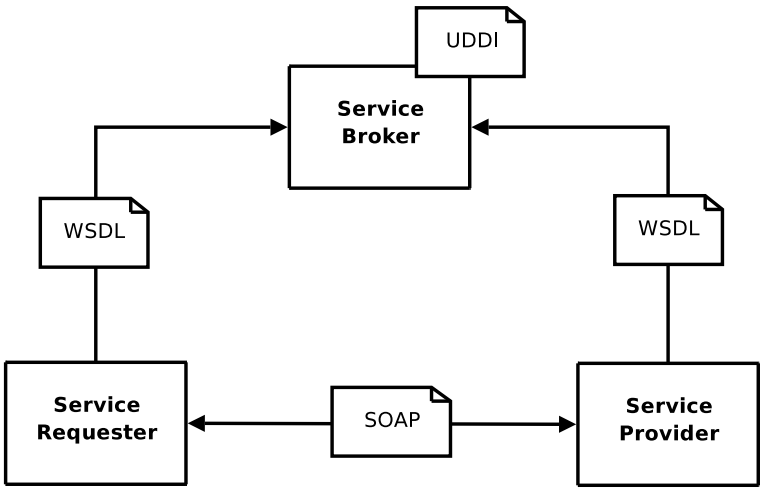

Figure 2.1: Web services roles

2.3 Business Process Execution Language (BPEL)

The Business Process Execution Language is a workflow language which allows the definition of business processes. The first version of BPEL was developed by major software players

in 2002 as a combination of two older workflow languages, Web Services Workflow Language (WSFL) and XLANG. The first usable version of BPEL, version 1.1, was released in April 2003. After that, BPEL was submitted to the Organization for the Advancement of Structured Information Standards (OASIS) for standardization purposes. In early 2007 the OASIS technical committee, which works on BPEL 2.0, released the final specification of this version [ACD+03, JE07].

BPEL is a Web services-based language, which means, that a business process written in BPEL is composed of other Web services. The process itself is a Web service too.

Oracle gives the following explanation for BPEL:

> *BPEL is a programming abstraction that allows developers to compose multiple discrete Web Services into an end-to-end process flow. It has built-in support for asynchronous interactions, flow control and compensating business transactions. It integrates with XPath, XSLT and XQuery for XML data manipulation" (Oracle)*

A usable introduction to BPEL is contained in [Hav05].

2.3.1 BPEL Execution Engines

There are several BPEL execution engines on the market. This section introduces the use of one of them. Though not a truly theoretical basis for this work, BPEL execution engines are heavily involved in the implementation of the approach described in this thesis. To let a BPEL process run in an engine, additional files have to be prepared.

Active Endpoints' ActiveBPEL

This work uses the freely usable BPEL engine "ActiveBPEL" by Active Endpoints [Act07]. One reason is the easy use of this product; others are the ongoing progress and development, and the supporting staff of this software company. In the following the deployment specific to ActiveBPEL is described.

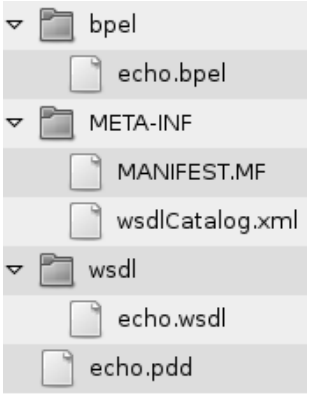

Figure 2.2: Directory structure of a BPR

To deploy a process in ActiveBPEL a "deployment descriptor" must be written. This is a

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="completeName" type"xs:string" />
</xs:schema>
```

Listing 2.1: Example XML Schema definition

configuration for the deployment of the process. It contains all partner links that are used in the process, and their settings.

The second needed file is a catalog of WSDL and XML Schema files, that are used by the process (`wsdlCatalog.xml`).

All files needed for the deployment must be packaged in a JAR file with the file name extension "BPR" and then be put into a special directory in the BPEL engine's installation. The last step completes the deployment. The typical directory structure of such a BPR file is shown in figure 2.2 on the preceding page.

2.4 XML Schema

XML Schema is of special importance to this thesis because all protocols and data formats used are defined using XML Schema. A complete introduction to XML Schema would exceed this chapter, so I give a very brief explanation of XML concepts that are important in this work. Some good introductions to the topic can be found in [HM03], the W3C Primer 0 may also be convenient [FW04].

XML Schema is a W3C recommendation for the formal description of the structural properties of XML documents. An XML schema is itself an XML document. XML Schemas allow to describe of which elements an XML document may consist if it shall be valid relating to the schema. An XML document that validates against a schema definition is called an "instance" of that schema.

An XML Schema definition consists of an XML document that contains the element `schema` as its root element. A very simple schema definition is shown in listing 2.1.

This defines a class of documents that must contain a single element "completeName" which can contain any character string. XML Schema defines a set of pre-defined *simple types*. These are subsets of character sequences which are restricted to represent only defined value sets. To give a few examples for simple types: *string*, *integer*, *decimal*, and *boolean*.

In contrast to simple types, "complex types" are types that define the structure of elements by means of a *content model*, which defines in what way children may be contained in the element. There are three types of *content models*:

sequence The element has children which have to appear in a defined order.

choice One of the defined elements may be the child element.

all The children of the element form a subset of the defined elements.

Two more advanced concepts of XML Schema have to be explained because it is used later in this work: *groups* and *redefinition*. A *group* is a list of element definitions that can be used

```
<xs:redefine schemaLocation="URI_of_the_source_schema">
  <xs:group name="a-group">
    <xs:choice>
      <xs:group ref="a-group"/>
      <xs:element name="new-element" type=""/>
    </xs:choice>
  </xs:group>
</xs:redefine>
```

Listing 2.2: Redefinition of a schema element

to modularize a schema. For example, in a sequence definition a group name can be used to refer to a previously defined group of elements. This is useful, if the same group of elements should be used several times in the schema.

Redefinition: A subset of an schema can be redefined to enhance or restrict some of its element or type definitions. For example, if an element should be added to a group definition, this group can be redefined. Listing 2.2 shows an example. It is important to notice that the group itself must be included in the new group. Otherwise the group would only contain the new element.

Chapter 3

A Concept for User Interaction in BPEL

The BPEL 2.0 specification does not include mechanisms for user interaction. Therefore, if we want to let a user intervene in a BPEL process instance then we have to extend BPEL. In this chapter I describe a concept for an extension to BPEL 2.0. At first the concept's foundations and requirements are described. Then the concept's details are shown.

A BPEL process is executed inside a BPEL engine. As already described in chapter 2, a BPEL process can make calls to other Web services. However, for human intervention, it is necessary to "call" human users from within a BPEL process instance. Because the BPEL specification does not define a mechanism for user interaction, we have to create a way to do this with BPEL to overcome this shortcoming.

We describe how BPEL can be extended to allow interaction with human users. This is done by making the process invoke a dedicated Web service that acts as a mediator between the process and the user. We will call this Web service the "user manager". To allow a human user to edit process data interactively we intend a graphical application program, which we will call "user client". This application will allow human users to perform tasks, for example edit data or make a decision, and send back the result to the process the tasks originate from. From process view, human users—who use the user client—are Web services. Hence users perform a specific task we can speak of a *user-implemented service*.

To make the call to the user manager easier BPEL is extended by a new activity type that takes care of the correct communication to the user manager which then will call the intended human user. The idea of such an activity is described by Kloppmann et al. who call it a "people activity" [KKL+05]. That activity generally represents one or more user tasks to be performed by a human user. For simplicity from now on we speak of a "user activity" instead of a people activity. Although this naming, strictly speaking, includes non-human users as well, we use it here only for human users.

3.1 Requirements

In this section the requirements to the BPEL extension are analyzed.

There are several possible scenarios which imply human user interaction in business processes, as described in [KKL+05]:

people activities These are tasks that are assigned to a user.

people initiating processes Not only other Web services can initiate BPEL processes, but also human users shall be able to start them.

process management While a process instance is active, events that require interventions by a human user can occur. One example is a scenario in which a user has to decide if or how the process should continue.

Here it can be useful to have a business administrator who decides what has to be done to ensure smooth operation of the process.

transitions between human and automatic services Often it does not matter if the service provided by a process is performed automatically or with human user interaction. For the calling client this must be irrelevant. So the transition between human and automatic services must be non-disruptive.

advanced interaction patterns Advanced interaction patterns are business process scenarios which involve more complex conditions to the user interaction. For example this can mean that more than one human user is involved, or that time constraints to the execution of a task have to be met. Some examples are: "four-eyes principle", "escalation", "nominations", and "chained execution". The user manager should be extensible to support more interaction patterns.

This thesis only treats the first scenario described above. The other four scenarios are optional.

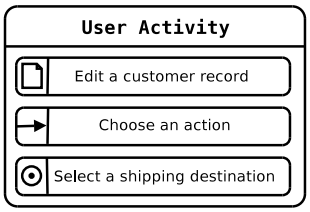

Figure 3.1: User activity with combined tasks

To allow human users to intervene in running BPEL processes the users need a special interface that allows operations on data belonging to the process. A graphical user interface is to be developed that provides all functions necessary for the user interaction with processes.

Sometimes it is desirable to combine several single user activities to form a unity of activities. The user activity has to support this bundling in a simple and flexible way. In order to make the terms clear we speak of a user activity which signifies the whole unity; likewise we speak of an *user task* to signify a single and indivisible task. A representation of a user activity that combines three single user tasks is shown in figure 3.1.

Our concept uses four different types of user tasks, similar to those defined in [Lü05]. These are sufficient to support many possible scenarios of user control. The four tasks are:

edit Let a user edit a record. The record is to be transferred to a user application, e. g. a front end where the user can make changes to the record. A record is any data which can be stored in a single BPEL variable. Thus, it can be any data representable in XML.

visualize Show arbitrary data to a user. To this data the last explanations apply, except that the user cannot change the data.

selection Offer a choice of items to a user. The user is offered a set of items from that he can choose a subset. The subset may be constraint to contain only one element, e. g. the user can only choose one element.

control Let the user choose one action out of a set of actions. The selected action is intended to be executed in the process. This task is meant to let the user take control of the process execution.

Flexibility

A user activity should have properties that make it flexible to use in processes. The following conditions should be fullfilled by the extension.

Independence of BPEL Engines

Today lots of BPEL engines exist (see chapter 2). The extension of BPEL would be quite useless if it could not be used in as many execution environments as possible. Hence, for a BPEL user, customizing an execution engine to support an extended BPEL would be pointless because that engine can—and probably will—change in the future. Therefore our BPEL extension has to work with every BPEL execution engine on the market.

Combination with standard BPEL activities

For an optimal integration of the user interaction into BPEL it has to be compatible with existing standard activities. This means that the use of the user acitivity must not influence the correct behavior of the surrounding process. Likewise, the activity has to support advanced BPEL features like the parallel execution of activities.

Adaptability of the User Interface

A user may have to adjust the user interface to the look and feel of existing applications and domains. One example is one company's corporate identity that demands a certain look and feel. The user interface shall be customizable. This possibly can be obtained by style sheets or other types of configurable parameters to user interface properties.

Automatic Generation of the User Interface

The user interface depends on the user activity and on the data which is to be processed. To simplify the use of user activities it is desirable that no definitions outside the process are

necessary for the creation of the user interface. Therefore the user interface has to depend solely on the user activity and the data.

3.1.1 User Resolution

In the normal case a user activity is assigned to a single user so that it is clear which user has to perform the tasks included in the user activity.

However, during the execution of a process it might be unknown which exact user has to perform a task (or a bundle of tasks, which is regarded the same). A mechanism is needed that allows a more indefinite specification of target users.

One common solution is to define groups or roles that a user can belong to. For our purpose there is no relevant difference between a group and a role because all users that have a certain role can be seen as members of a group which represents that role. From now on we speak of a *role* to denote a group as well as a role. In respect of business processes this is reasonable because for instance an employee might have certain responsibilities that are expressed by roles (think of a manager role).

The usage of roles requires a method to resolve them to actual user names, though. Because a user may have several roles and a role can be owned by several users, the user manager has to decide into which user a role resolves. How this decision is made is not further specified here. A simple algorithm may be to assign tasks to users who do not perform any tasks yet.

Whenever a user activity during the execution of a BPEL process is reached, a message is sent to the user manager. This message has to include

a) information about which user or role has to be addressed,

b) data that is needed to create the user interface, and

c) data that a human user has to process in some way.

After the user manager has received such a message from a process the user manager has to forward that message to an appropriate user. The mapping from users to roles has to be managed by the user manager. It is up to the implementation how this resolution is accomplished.

A process must assign a role or a user to each user activity. As stated by Kloppmann et al. “Particular users who perform a task may be specified at design time, at deployment time, or at runtime.” [KKL+05] This also applies to particular roles, as they are explained above. Design time is the time as a process is developed. If a target user or role for a user activity is defined at design time, this assignment stays static and cannot be changed later. A process has to be deployed, so that it can execute in a runtime environment (e. g. a BPEL engine). It can be necessary to decide during the deployment which user has to be addressed by a user activity. At last the assignment of a user activitry to a user can take place during process execution (e. g. at runtime). The new user activity type has to provide a method to make the assignment to users or roles dynamic.

This analysis leads to the following three types of user assignment:

assignment by username The user manager addresses the user directly by name.

assignment by role The user manager determines a user who owns the role.

assignment by query This is for dynamic user assignment. A user or role can be specified by an XPath expression and an XML document. The XPath expression points to data in the XML document, which must be either a role name or a username (as described under 1. and 2.). This resolution method is necessary for roles or users who depend on runtime data in the process instance.

3.2 An Extension to BPEL: U-BPEL

As mentioned above, BPEL has to be extended by one new activity type to support user interaction. To distinguish the official BPEL version from our extension we call our extension U-BPEL, which is short for "User BPEL". This work uses BPEL 2.0 as a basis because it will become prevalent in the future.

The user activity must allow for the specification of the targeted user or role, and all tasks the user has to process. As demanded in the requirements section of this chapter, it has to be possible to add as many tasks as necessary to the user activity, whereby they can be arbitrary nested. Before the new activity is presented, the different task types and their application are defined.

Aside from the four different task types mentioned above ("edit", "visual", "selection", and "action") a fifth task type is needed. It serves as a container that can contain any number of any other task, which means that it can contain itself as well. This virtual task allows the creation of arbitrarily nested structures of tasks for the greatest possible flexibility in defining user activities inside a process.

Because only one additional BPEL activity type is needed, the only difference between U-BPEL and BPEL is a new activity that is added to BPEL's XML schema definition. This activity's name is `user`. The complete XML Schema definition of U-BPEL is in listing A.1.

The user activity has the following two mandatory child elements:

- The target user or role is given in the child element *peopleLink*. This element has attributes for specifying the target user or role. The exact use is described below.
- All tasks are put into the child element `userTasks`. This element is the virtual container task that has been described above.

The five task types have the following names: `userTasks`, `editTask`, `selectionTask`, `controlTask`, and `visualTask`. A detail of the schema definition for the U-BPEL activity "user" is depicted in figure 3.2 on the following page. In the figure both child elements can be seen. At the top is the `peopleLink` element; it has to come first. The `userTasks` element is at the bottom. As can be seen the `userTasks` element must contain at least one child element. If the child is itself a `userTasks` element it must have a child element as well. Therefore, at least one of the "real" user task elements has to be present in the user activity. This ensures that a request to a user always contains an executable task.

To get a better understanding of the new activity type we regard a simple example now. A company's business process contains in one of its variables a postal address that has to be

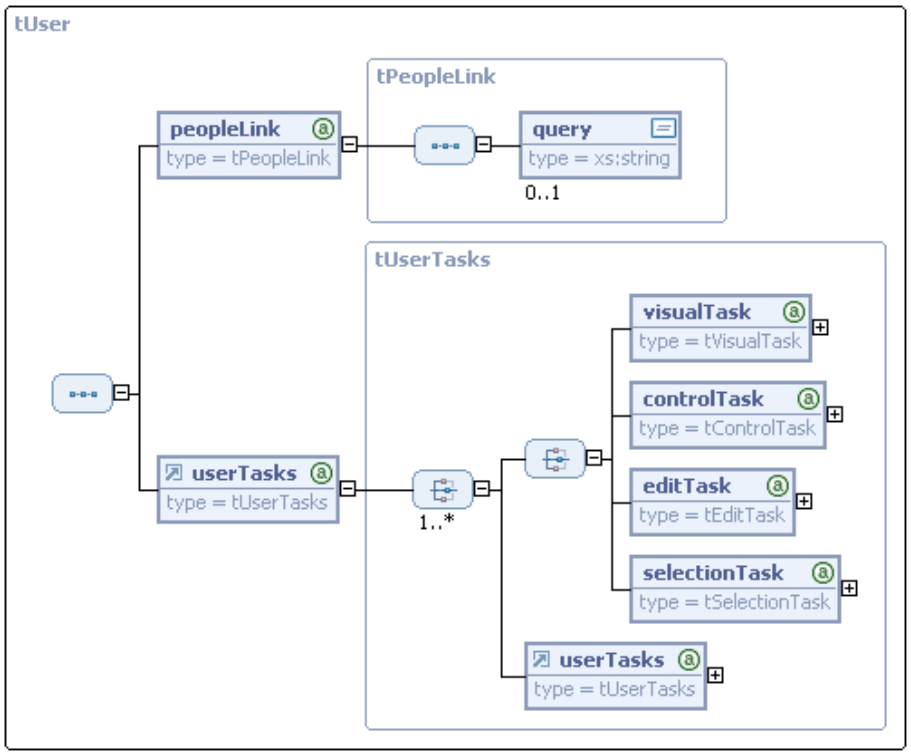

Figure 3.2: View of the schema definition for the U-BPEL activity `user`

reviewed by an employee "John". Let the variable *Address* contain the address in question. During the process execution the address is written into the variable *Address*. To tell John to review the address a user activity has been inserted into the process. Once the user activity is executed the user manager is called by Web service invocation. Transparently to the process, the user manager takes care of delivering the task to John. After John has reviewed (and possibly changed) the address it is sent back to the waiting process and is written back in the variable *Address*. Listing 3.1 shows the U-BPEL code for this example.

```
<user>
  <peopleLink method="name" value="John" />
  <userTasks title="Address Verification" type="tabs">
    <editTask variable="Address" />
  </userTasks>
</user>
```

Listing 3.1: Example of a U-BPEL user activity

3.2.1 Definition of People Links

Each user activity has to be linked to either a user or a role. In this section the definition of target users or roles is described in detail. The user activity must have exactly one element *peopleLinks* as the first child. As described above there are two methods to address a user:

by the user's name or by a role. Furthermore, I described the possibility to decide at process runtime which user or role a user activity has to address. For this purpose the *peopleLinks* element has different attributes and one child element. Depending on the method to specify the user or role respectively, only a subset of the attributes and child elements may be used.

There are four attributes:

method Specifies if a user or a role shall be addressed. This attribute may have one of the values "user" or "role".

value Contains the name of either a user or a role depending on the previous attribute.

variable Specifies the name of a BPEL variable that contains the user's or role's name, again depending on the attribute "method". Together with this attribute the *peopleLinks* must contain a child element *query* that contains an XPath expression. This expression points to a node in the XML data that is contained in the BPEL variable. The node has to contain the user's or role's name respectively.

part If the *variable* attribute is used and that variable has a WSDL message type then the part of that WSDL message type can be specified in the *part* attribute. Otherwise this attribute must not be used.

We have to note that the use of the attributes "value" and "variable" is mutually exclusive. Only one of them may be used at the same time. In the case a variable is used to specify the user or role, the *query* element is only necessary if the variable or its WSDL part is not of a (XML Schema) simple type. The query is used as in BPEL's *from-spec* or *to-spec*. However, the query language is fixed to XPath 1.0.

This leads to four possibilities to specify the target user. First, a target user or role can be addressed with a literal value, either by name or role. Second, the the value for user or role can be extracted by an XPath expression from a BPEL variable.

With the possibility of using BPEL variables to specify names of users or roles the flexibility mentioned in the requirements section is met.

In listing 3.2 a very simple example is shown. A user with name "mrsmith" is addressed. The tasks are omitted for brevity.

```
<user>
  <peopleLink method="name" value="mrsmith" />
  <userTasks><!-- omitted tasks --></userTasks>
</user>
```

Listing 3.2: Addressing a user directly

Listing 3.3 gives an example of how an XPath expression can be used with a people link. It is assumed that the variable *Managers* contains a list of managers. This user activity chooses the username of a manager whose last name is "Smith".

```
<user>
  <peopleLink method="name" variable="Managers">
    <query>/managers/manager/username[name='Smith']</query>
  </peopleLink>
  <userTasks><!-- omitted tasks --></userTasks>
</user>
```

Listing 3.3: Addressing a user with an XPath expression

3.2.2 Task Types

This section describes how the U-BPEL's five task types are used. Listing 3.9 on page 20 shows a user activity that contains all possible task types together in one user activity.

userTasks

As described above, this task type is a container for other task elements, including another *userTasks* element. It can have two attributes:

title Gives the container a title that is shown in the user client.

type Determines how the children of the container will be arranged in the user client. A value of "simple" will arrange the children tasks one below the other. A value of "tabs" will cause the child tasks to be put in tabs. If this attribute is omitted, a value of "simple" is assumed.

The exact syntax of the task *userTasks* is shown in listing 3.4.

```
<userTasks title="xsd:string"? type="tContainerType"?>
  any-task-type+
</userTasks>
```

Listing 3.4: Syntax of task type *userTasks*

editTask

With this task a user can edit process data. A BPEL variable has to be specified in the attribute *variable*. This variable's contents are sent to the user and written back after the user's response is received. The syntax of the task *editTask* is shown in listing 3.5.

```
<editTask variable="BPELVariableName" />
```

Listing 3.5: Syntax of task type *editTask*

selectionTask

A user may choose one or more items from a set of items. This task enables a process to let a user make a selection. Because there is a choice of element and the resulting selection, two

variables have to be specified for this task element. One variable must contain the choice of items and in another the chosen selection (result) will be written after the user has finished the selection.

As a default only one item may be chosen. It is possible to extend the selection to multiple elements by setting the attribute *multiple* to "true". (The default is "false".)

The exact syntax is as follows:

```
<selectionTask choiceVariable="BPELVariableName"
  resultVariable="BPELVariableName"
  multiple="true|false"? />
```

Listing 3.6: Syntax of task type *selectionTask*

controlTask

With this task a set of actions can be offered to a user. One of the actions can be selected. The selected action's ID number is returned to the process by writing it into the variable that is specified in the task's `resultVariable` attribute. Thus, a user can take control over some point in the process execution.

The syntax of *controlTask* is given in listing 3.7. Such an element must contain two or more `action` elements each of which has the following attributes:

id: An id for this action (must be an integer number).

description: A description for this action. The text is displayed to the user.

name: An optional name for this action. This text may further explain the action to the user.

```
<controlTask resultVariable="BPELVariableName">
  <action description="xsd:string" id="xsd:integer"
          name="xsd:string"? />
  <action description="xsd:string" id="xsd:integer"
          name="xsd:string"? />+
</controlTask>
```

Listing 3.7: Syntax of task type *controlTask*

visualTask

This task's purpose is to show the user a set of information. One use of this can be to let a user notice something, maybe a reminder to do something. Another application can be to display information supplemental to other (sub) tasks.

Only one attribute is mandatory for this task type. The attribute *variable* must contain the name of a variable that contains the data which shall be displayed in the user client. The syntax is as shown in listing 3.8.

```
<visualTask variable="BPELVariableName" />
```

Listing 3.8: Syntax of task type *visualTask*

Listing 3.9 gives a complete example of how all task types can be combined in a single user activity. Note that the tasks are nested, i. e. the first *userTasks* element contains not only "real" task but also another *userTasks* element.

```
<user>
  <peopleLink method="name" variable="Var1" part="parameters">
    <query>/nsp:value</query>
  </peopleLink>
  <userTasks title="To do" type="tabs">
    <controlTask resultVariable="Action">
      <action description="Notify the customer" id="1" />
      <action description="Do nothing" id="2" />
    </controlTask>
    <userTasks title="Notice" type="frame">
      <editTask variable="Book" />
      <selectionTask choiceVariable="SubsidiaryChoice"
        resultVariable="SubsidiarySelection" multiple="true" />
    </userTasks>
    <visualTask variable="CustomerNote" />
  </userTasks>
</user>
```

Listing 3.9: Different task types in a user activity

3.2.3 Conversion from U-BPEL to BPEL

Existing BPEL engines obviously do not support the described BPEL extension. To execute a U-BPEL process with user activities in an engine it would be necessary to adapt the engine to support U-BPEL. This contradicts any flexibility because on the one hand there are too many BPEL engines, on the other hand, not all BPEL engines are distributed with their source code so a modificiation would be impossible. We have to look for something else. The solution is to translate U-BPEL to BPEL after the process is written. This will enable the process to run inside any standard compliant BPEL engine.

One useful fact is that BPEL is an XML language. Therefore every BPEL process can be transformed with XSLT. In fact only the user activity has to be transformed into a sequence of valid BPEL activities. All other parts of the process can be left untouched.

For a conversion we need to define valid BPEL code as a result. A user activity in U-BPEL involves several single activities in BPEL. The actual Web service call to the user manager has to be prepared by creating a message that contains all instructions and data necessary for the execution of the user activity's tasks. After the user has finished all tasks of a user activity its result is sent back to process in forms of a message. After the process has received that response message its contents have to be handled so that the process can use it.

Essentially the following steps have to be executed for a user activity:

1. Create a message that will be sent to the user manager. All data required for the tasks must be included in the message. This includes not only the data that has to be processed in the user client, but also meta data that is needed for the GUI generation. The generation of the user interface is described in the next section.
2. Make a Web service call. The previously prepared message is sent to the user manager service. This is done by a BPEL `invoke` activity.
3. Wait until the user has finished the tasks contained in the message. The user manager will then send back a response message. This is accomplished by a BPEL `receive` activity.
4. Move all data from the response message to the appropriate BPEL variables so that the process can use the data.

A converted U-BPEL user activity basically has the structure shown in listing 3.10. Implementational details are left out. The first step described above is done in the first *assign* activity (lines 1 to 11). Here, the request message to the user manager is built. After this, step two is executed by means of an *invoke* (line 12). Step three is done by the *receive* activity that waits for a response message (line 13). Once the response is received step four is finished by an *assign* activity (lines 14 to 16).

```
1  <assign>
2    <copy>
3      <from>
4        <literal>
5          literal XML data
6        </literal>
7      </from>
8      <to variable="UMrequest" ... />
9    </copy>
10   copy activities
11 </assign>
12 <invoke inputVariable="UMrequest" ... />
13 <receive variable="UMresponse" ... />
14 <assign>
15   copy activities
16 </assign>
```

Listing 3.10: U-BPEL user activity converted to BPEL code. The contents of the input variable is prepared before the invoke is executed. After the receive possible response data is copied back to variables.

The conversion will be realized with an XSLT stylesheet. Further details are a matter of implementation, which is presented in chapter 5.

3.3 Generating the User Interface

The user client has to render information coming from BPEL processes into a graphical user interface that allows the user to execute defined actions on this data. Every action is represented by one of the four task types that are described above under requirements. We call this

created user interface a "task form". Aside from the task form, a frame for the task forms is needed. Because a user activity allows nested tasks the task forms have to support nesting.

Information from the process has to be transferred to the user client in a properly defined manner. BPEL processes use variables to store data. A BPEL variable's type usually is defined by XML Schema. Another option is to use WSDL message types. Therefore information that should be processed by human users can be easily transferred to the user client as normal XML data. The user client can then use the XML data to render a user interface out of it.

However, this does not suffice for rendering the graphical user interface. In addition to the data an XML schema definition for the data is needed as well. One reason for that is the fact that XML data alone contains no meta information about itself. Such type information is needed to validate possible input by the user, though.

The second reason why the XML schema definition is needed is because of the fact that an XML schema complex type can contain optional elements and multiple children of elements. It is possible that such elements are not in the data which is to be edited by the user. So, without the schema definition the user client would have no chance to find out if the user can add elements at certain positions in the document.

Third, prior work exists that describes a concept for creating user interfaces out of XML schema definitions [LLK04]. It is suggested to transform schema definitions into a Java Bean representation of Swing components. To create a GUI, these "encoded" Swing components can then be "decoded" into actual Swing components.

However, I follow a different approach: The schema definitions are directly interpreted to create GUI components. One principle remains, though: To make dynamic user interface creation possible both the process data and its XML schema definition have to be used. In the next section my approach to generate an editor GUI from schema data is explained.

3.3.1 Traversing Schema Definitions

How can an XML schema definition be used to create a graphical user interface? I show a method that traverses a schema definition tree to create GUI elements that match the schema definition.

First we consider the simple schema definition in listing 3.11 on the next page. This schema defines an element "book" that must contain a sequence of five child elements. All of them are of an XML Schema simple type. Note that the shown schema definition is not a valid XML Schema of its own, because it has no parent element "schema". Thus the processing of the definition becomes easier (see the algorithm below).

One possible instance document for this element definition is shown in listing 3.12 on the following page.

Let us say the typing error in the title of this *book* shall be corrected by a user. ("Busines" lacks an "s".) A suitable user interface has to be created. This user interface has to show the data in a way that enables the user to edit it. It is common to arrange the elements of a record of data—as like this book—in a tabular fashion. An editor GUI may look like sketched in figure 3.3 on the next page.

To build this form out of the corresponding schema definition, the definition tree has to

```
<xsd:element name="book">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string" />
      <xsd:element name="author" type="xsd:string" />
      <xsd:element name="publisher" type="xsd:string" />
      <xsd:element name="year" type="xsd:gYear" />
      <xsd:element name="pages" type="xsd:positiveInteger" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Listing 3.11: Schema definition for an element *book*

```
<book>
  <title>Essential Busines Process Modeling</title>
  <author>Michael Havey</author>
  <publisher>O'Reilly</publisher>
  <year>2005</year>
  <pages>332</pages>
</book>
```

Listing 3.12: Example *book* element

be traversed until an element that has an XML Schema simple type is reached. Every simple type can be assigned to a suitable GUI component that exists in most GUI toolkits.

My concept for a GUI generation algorithm is shown in listing 3.13 on the following page. It uses a blend of Java syntax and pseudo code that should be easy to understand.

The function expects as an argument an XML node that contains the schema definition, and a GUI component object. After the schema definition is completely traversed the GUI component will contain the generated GUI. The example in listing 3.11 shown above would be traversed in the following steps:

1. An element *complexType* is found because it is the first child of *element*. The GUI component is not changed. The function is recursively called with the current child and the unchanged GUI component.

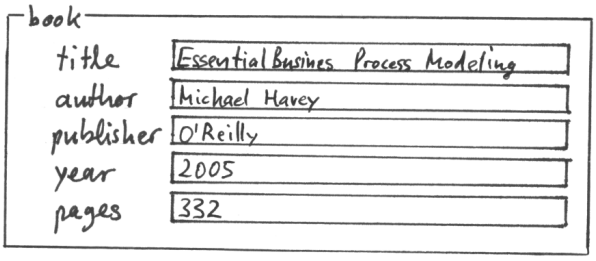

Figure 3.3: Sketch of an edit form

2. Now the element *sequence* is found. Again the GUI component is not changed. The call is the same.

3. For each of the five elements ("title", "author", "publisher", "year", "pages") a new GUI element, that is suitable for the XML Schema simple type, is created and then added to the GUI component.

```
1  void traverse (Node node, Component comp) {
2    for (child in node.children) { // traverse all children
3      if (child.name == "complexType")
4        traverse (child, comp);
5      else if (child.name == "sequence")
6        traverse (child, comp);
7      else if (child.name == "element")
8        if (child is of simple type)
9          comp.add (new GUI component for this simple type);
10       else if (child is of complex type) {
11         subcomp = new Component ();
12         traverse (child, subcomp);
13         comp.add (subcomp);
14       }
15   }
16 }
```

Listing 3.13: Principle of an algorithm for GUI generation using schema traversal

Chapter 4

Architecture

This chapter describes the architectural requirements of an implementation for the problem. At first, the structure of the architecture is described. After that, some non-functional requirements are defined.

The architecture's structure is derived directly from the concept. Chapter 3 defined three parts that communicate with each other:

1. A BPEL process is executed by an execution engine.
2. A Web service, that is called user manager, receives requests containing user tasks from the BPEL process. The user manager forwards each request to exactly one user.
3. Human users use an application called user client that receives task requests from the user manager. With the user client a user is able to perform tasks and send them back to the user manager which causes the tasks to be returned to the process they originated from.

These three responsibilities can be considered as layers of an architecture. This makes perfect sense because the user client application will never interact directly with a BPEL process. All communication is handled by the user manager. It works as a mediator between processes and users. This relationship is shown in figure 4.1. The layer concept requires that every layer only communicates with a neighboring layer.

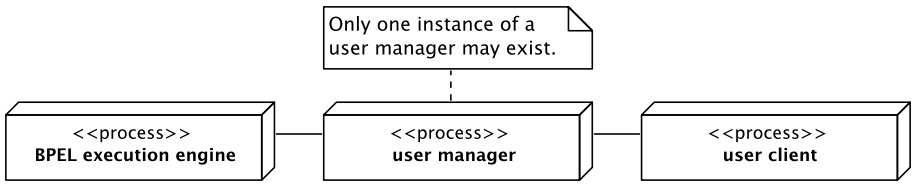

Figure 4.1: Overview of the architecture. The layers are *BPEL execution engine*, *User Manager*, and *User Client*.

Because I divided the solution into layers, they have to be made independed of each other—strictly speaking, each layer has to be located on a different machine. In the realization it must

therefore be possible to put all layers on different machines. Of course, this is not necessary in every case; for example, all layers can be run on the same machine for testing purposes.

In the following sections I will describe each of the single layers with their respective responsibilities.

4.1 BPEL Execution Engine

To narrow the scope we make the following definition. A process only pertains to this layer if it makes use of the *user activity* that was described as an extension to BPEL in chapter 3. As described there, a U-BPEL process, that utilizes the user activity, has to be converted to BPEL before it can run in a standard engine. All other processes can also run in the same engine but they are of no interest here.

This layer is not actively involved in this work because we use already existing BPEL engines. Actually the only necessary work for this layer is done outside of the process execution because a process that contains one or more user activities (e. g. a U-BPEL process) has to be converted to BPEL before it can be deployed in the BPEL engine.

Messages to the user manager, that are created in converted processes, have to comply with the user manager's interface description. The message format is determined in the conversion process because code for the user activity is created that uses literal XML data to build the message.

Every time a running process (e. g. a process instance) reaches the generated code of a user activity, the process has to make an asynchronous Web service call to the user manager. After that, the process instance has to wait for a response. The call to the user manager must be asynchronous because a user cannot perform their tasks in a period of time that is short enough to be accepted by a BPEL engine's timeouts. It is possible that a user activity "runs" for days or even months until it is finished by the user.

4.2 User Manager

This layer constitutes an exceptional in this architecture, because only one instance of the user manager may run at a time. This is necessary for the following reasons. Certainly it would be possible to use several autonomous user managers out of BPEL processes by specifying a user manager's partner link in the user activity. The different user managers would have to synchronize to build a joint task pool. This would complicate the architecture very much. However, user clients must know in advance which user manager they have to contact to get undone tasks. This can be accomplished by using a service broker, e. g. utilizing UDDI. Without service lookup it is necessary to provide a central service for the user client, though. To simplify the solution, I decided to limit the number of user manager instances to one. By this means all task handling can be done in one place and no additional communication between user manager instances is necessary.

The user manager has three major jobs:

- Receive requests containing user tasks from processes and send the result back to them.

As described earlier this is realized by an asynchronous Web service call invoked by the inquiring process.

- Maintain (e. g. store) tasks until they are finished and sent back to their processes. After a task has been received it has to be assigned to the right user (see section 3.1.1 on page 14).
- Handle user clients. This involves user registration like login and logout but also, more importantly, operations for sending task messages to the clients and receiving finished tasks.

Since the user manager lies between two layers, namely *BPEL execution engine* and *user client*, it has to communicate with both of them. The user manager's connection to the other layers is realized by means of a Web service interface. This interface defines operations for the three major jobs described above.

A typical procedure of handling a user task message consists of three steps:

1. A process is sending a user task message. The user manager is looking up a user who can perform the task, assigns the task to him, and saves it appropriately.
2. The user manager waits that this user logs in and fetches his undone tasks.
3. While the user works on his tasks, the user manager is waiting for a response. When the user responds with his finished task the user manager immediately tries to send the task message back to the process it originated from. If this was successful the procedure ends.

4.2.1 Concurrency

In real world operation it is possible that several processes send task messages to the user manager at the same time. For this case the user manager has to be able to handle tasks concurrently. To make real concurrency possible, several user manager services, running in parallel, would be necessary. However, this would require all those services to be synchronized in some way. In the time scope of this thesis this would be too complicated to handle. In practice a Web service need not have to be better than a usual Web server. Therefore it is sufficient here to run only one Web service.

However, it is important that the user manager handles requests as fast as possible. If the processing of user task requests exceeds a sensible threshold of time the user manager would be blocked for other processes. Other requests could not be received and the concurrency issue would become serious. After all, the user manager must not be blocked by any operation for a longer time than absolutely necessary.

4.3 User Client

The user client is used on a individual computer. This application allows users to look for new tasks, to perform them, and to return them to the user manager. All communication with the user manager is realized by Web service calls, all of which are made by the user client. As a

consequence, the user manager cannot notify a user of newly arrived tasks. Therefore users have to connect to the user manager independently to get noticed of new tasks.

Other jobs of the user client were already described in chapter 3, as displaying user tasks, and the like.

4.4 Interaction and Temporal Behavior

This section gives an overview and an exact definition of the interaction between the layers. Most parts of the interaction were already mentioned above.

Every time a process instance executes a user activity, it delegates the execution to the user manager. The user manager determines the user which has to perform the task.

Figure 4.2 on the next page shows the interaction between the three layers exemplarily. This scenario has the following order of events.

- A user starts the user client (1).
- A new process is created by Web service call (2). This process has two sequential user activities, which means that the second user activity cannot execute before the first one was finished. Further details about the process or its user activities are not of concern here.
- The process sends user tasks to the user manager (3), which then waits for the user to perform the task.
- The user client logs in to the user manager (4, 5), and fetches new tasks (6). Because just one task is waiting until then, only this task is returned to the user client (7).
- The user finishes the task (8) and sends it back to the user manager (9). The user manager triggers a callback to the process (10) and acknowledges the user client's message (11).
- Because the process needs another user interaction, after a while a second *userCall* is done (12). Then, the same procedure as explained above is passed through.
- Some time after the second callback to the process has happened, the process finishes and sends a callback message to the Web service.

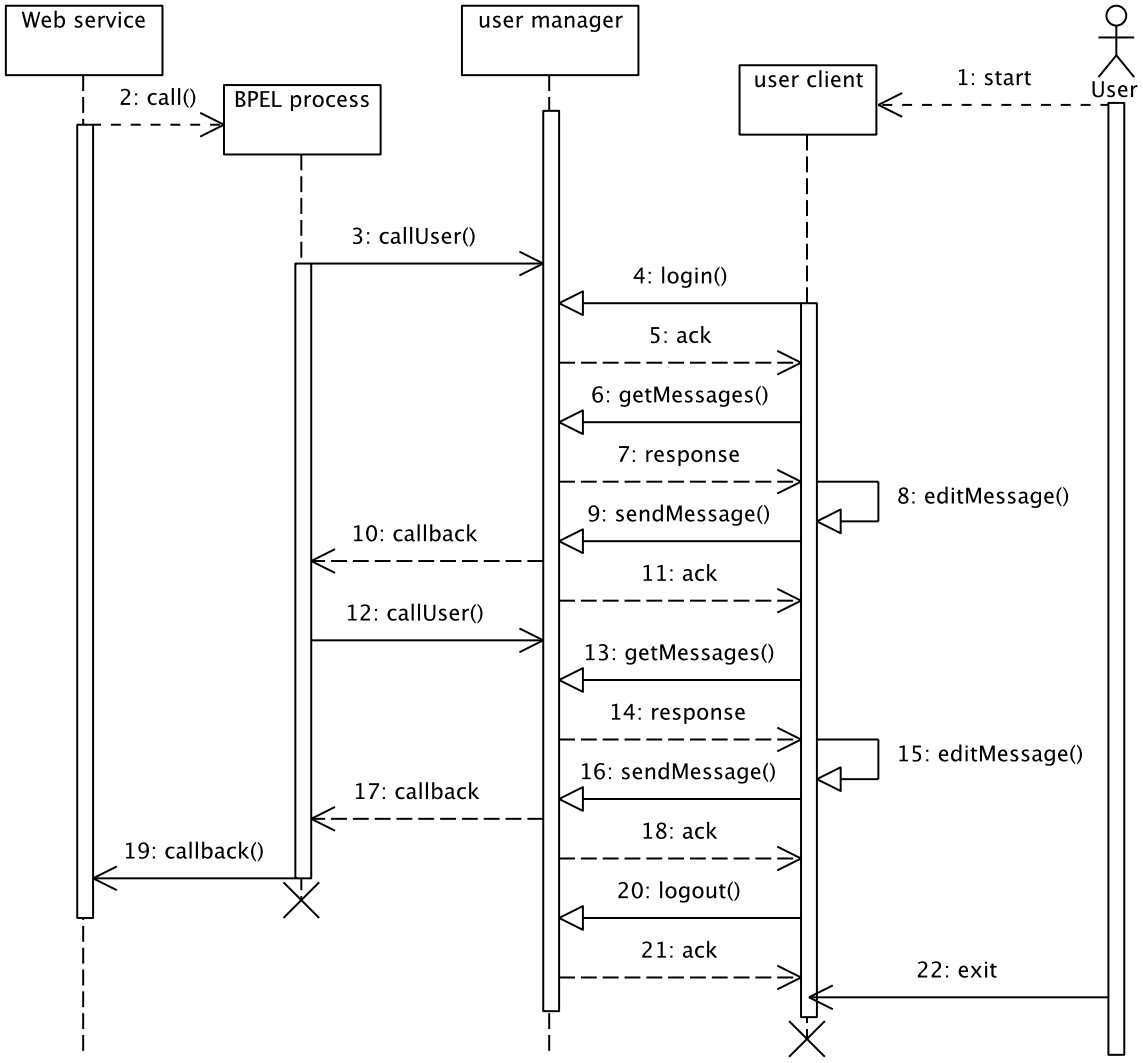

Figure 4.2: Interaction between the three layers *BPEL engine*, *User Manager*, and *User Client*

Chapter 5

Implementation of a Prototype

This chapter describes the implementation of a) the conversion from U-BPEL to BPEL, b) the user manager Web service, and c) the user client application. I present how the solution works and which problems still exist.

At first we define a convention:

- We call the *userTasks* portion in a user activity simply a "task" to express that a user has to do something with it. The contents of the task does not matter if it is transmitted between the layers of the architecture. However, we still speak of "task types" when it is necessary to distinguish.

5.1 U-BPEL Conversion

In chapter 3 I introduced the idea of converting U-BPEL processes into standards-conform BPEL processes. In this section I will present a realization of that concept. The principle of the conversion of the U-BPEL process is depicted in figure 5.1.

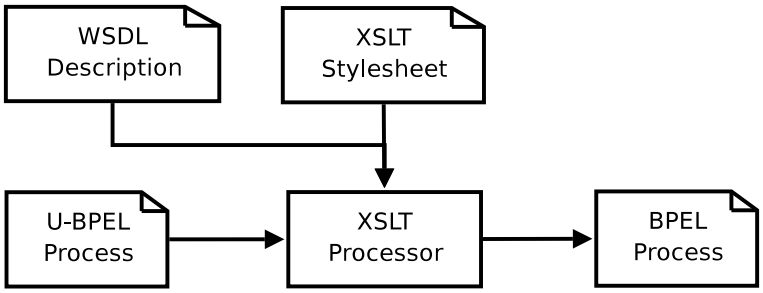

Figure 5.1: Principle of the U-BPEL conversion. The XSLT processor transforms the U-BPEL process into a BPEL process by executing the XSLT stylesheet. Type definitions for variables are extracted from the WSDL file of the process.

5.1.1 Prerequisites

Before the conversion can be done, we must analyze the prerequisites that have to be fullfilled in order to create BPEL code that works and builds a valid request message.

Correlation Sets

As a user may respond after a long period of time, a user activity must use an asynchronous Web service call. The response from the user manager must be correlated with the request message. In BPEL, this is typically done with "correlation sets". This correlation set must be added to the process.

Auxiliary Files and Deployment

Except for the BPEL file there are a few additional files that are important for the deployment and the execution of the process. Some of these files must be altered, too, in order to leave behind a deployable process after the conversion. The files and what has to be changed on them is described now.

WSDL file: The process' WSDL file has to be changed because a) the user manager's WSDL must be imported to let the process use definitions from there, b) a port type and a binding for the callback from the process must be added, and c) two partner link types for the user manager must be added.

WSDL catalog file: An entry for the WSDL file and the separate XML schema file of the user manager must be added to the catalog.

PDD file: The process deployment descriptor must contain partner link definitions for the user manager and references (locations) to the user manager's WSDL and schema files.

Locating Schema Definitions

To include schema definitions in the request message, they have to be found first. This is not as easy as it may seem because there are several ways to specify the type of a BPEL variable or a WSDL message. The schema definition, that is used for the type of a variable, can be defined in the types section of the WSDL declaration, or it can be included from a separate schema file. If the latter is the case, that separate file must be found first to access the type definition.

Most of the task types (see section 3.2.2 on page 18) cause the contents of variables (data) to be transmitted to the user client. As described before, the schema definitions of that variables' types are needed to display the data in the user client's GUI.

A distinction between variables which content is transmitted to the user client, and variables whose content is not transmitted, would complicate the conversion unnecessarily. Therefore, the U-BPEL converter assumes that each variable's type definition is needed—and present.

This convention leads directly to the next section.

Restrictions on Schema Definitions

XML Schema allows very complex (and complicated) type definitions. For example, an element definition can have a complex type that is defined elsewhere in the schema file. In such a case the element's type is a reference to a named complex type.

```
<xsd:element name="entry" type="tns:aComplexType" />
<xsd:complexType name="tns:aComplexType">
  <xsd:attribute name="key" type="xsd:string" use="required" />
  <xsd:attribute name="value" type="xsd:string" use="required" />
</xsd:complexType>
```

Listing 5.1: Schema definition with referenced complex type

Listing 5.1 shows an example. In this definition, the element *entry* has a complex type which is defined below. Now there is a problem. The user client needs a complete schema definition to show process data in the GUI. Hence the schema definition should be simple and contiguously defined. Otherwise it had be culled from the whole schema definition, or even from different schema files. This would complicate the conversion so much that I decided to put limits on the types of variables used in a U-BPEL process.

Restrictions on U-BPEL Processes

To ensure that each variable, which is used in a user activity, has a reachable and valid XML schema definition, we define the following restrictions on U-BPEL processes:

1. A variable that is used in a user activity must have its type definition located in the WSDL file of the process—and nowhere else.

2. Type definitions of variables must be contiguous, i. e. they must not contain any references to named types. An example of a contiguous type definition is given in listing 5.2. It is an adapted version of the previous example.

```
<xsd:element name="entry">
  <xsd:complexType>
    <xsd:attribute name="key" type="xsd:string" use="required" />
    <xsd:attribute name="value" type="xsd:string" use="required" />
  </xsd:complexType>
</xsd:element>
```

Listing 5.2: Contiguous schema definition

5.1.2 Transformation

Now the realization of the U-BPEL conversion is presented. In contrast to prevalent applications of XSLT, where XML documents are transformed from one document type to an other, in our case the XML documents only have to be changed partially. For example, to convert the U-BPEL process to BPEL, it only must be copied except for the user activity, which has to be replaced with new code.

To copy an XML document with an XSLT stylesheet, a template that matches every possible element in the document has to be written. This is realized by a template rule that recursively copies all elements from the input document to the output document. The code for such a template is shown in listing 5.3. However, all elements that have to be substituted, for example the *user* element, have to be matched by other templates. Because that "copy" template would override every other template, its priority has to be lowered (to -1). By that, the other templates are considered first by the XSLT processor.

```
<xsl:template match="*|comment()|processing-instruction()|@*"
  priority="-1">
  <xsl:copy>
    <xsl:apply-templates
      select="*|text()|comment()|processing-instruction()|@*" />
  </xsl:copy>
</xsl:template>
```

Listing 5.3: A XSLT template which recursively copies all elements of a XML document. Only elements for that a template exists are left out.

Each file mentioned above is treated like this. For each part of the files that has to be altered or substituted by new elements template rules are defined.

All needed type definitions of variables are copied into the process during the conversion. The result of the conversion is an executable BPEL process.

5.1.3 Execution of the User Activity

In section 3.2.3 on page 20 the BPEL code for the user activity already was described in a more general way. In this section specifics for the execution of the user actitivity are discussed.

The format of the request message to the user manager is defined in the XML schema definition of the latter (see listing 3.2.3 on page 20 for reference). Each user activity must be replaced with a sequence of BPEL activities that carry out the necessary actions. These actions were described as four steps in section 3.2.3.

ID for Correlation, Messages, and Tasks

Before the afore mentioned steps are executed a correlation ID must be created, because it will be copied into the request message. The correlation ID must be a unique number because more than one user activity may be executed in one process at the same time. However, BPEL does not provide a facility for the creation of unique (random) numbers. Therefore, an extra service operation, that returns a randomly created UUID (Universally Unique Identifier), was added to the user manager. This UUID is used as a correlation id for the request and response messages, and as an id for the task(s).

It is useful to have a unique ID for every task because the user manager and the user client can use the ID as a key to access task objects.

Scope Isolation

A user activity must be applicable in any context of a process. This means that this activitiy also can occur in a `flow` activity, which allows two user activities to be executed concurrently. To make the user activities independent from each other, they must not use the same variables, for this would lead to collisions and inconsistency. Since BPEL provides `scope` activities, it is possible to encapsulate a user activity (read: the generated code for it) so that all resources used by the activity will be isolated from the surrounding process. The conversion with XSLT realizes this.

Because of technical problems, the partner link definitions for the user manager have to be defined outside of the scope, though, that is to say, at process scope. At least, this does not affect the functioning of the user activity.

Restore Response Data

Finished tasks contain response data, i. e. altered process data or new input from a user. To make that data available to the process it must be extracted from the response message, and then be copied to BPEL variables that were specified with the respective task types.

The extraction is done by `copy` activities that are executed after the response message was received. (See the second `assign` activity in listing 3.10 on page 21.)

5.2 User Manager

Now the implementation of the user manager is introduced. It provides a Web service interface for both the BPEL processes and the user client. An overview of the operations is shown in table 5.1.

Operation name	Purpose
Operations for processes	
callUser	Send a request message (i. e. a task).
getRandomUUID	Get a random UUID.
Operations for the user client	
login	Log in. Returns a session id.
logout	Log out.
getMessages	Get all unfinished messages.
sendMessage	Send a finished message.

Table 5.1: Web service operations provided by the user manager

The Web service declaration (WSDL) and XML Schema definition for the user manager are in listings B.1 on page 55 and A.2 on page 49 respectively.

For the implementation the Web service engine Apache Axis2 is used [Fou07]. In Axis2, a Web service is implemented by a Java class. Every operation of the Web service is realized by a method of this class. Usually a new instance of this class is created when a call is received. However, Axis2 allows to configure the scope of a deployed service. This is the type

of session the service runs in. There are four types of scope: “Application”, “SOAPSession”, “TransportSession”, and “Request”. “Request” scope means that a service class instance is created for each request, so that every information that was present in an instance will be lost after the request is finished. “Application” scope means that the service class runs as long as the Axis2 engine is running.

Since the user manager service has to run all the time, it must be set to “application” scope. Otherwise, (user) tasks would not be preserved.

5.2.1 Operation Interface

The above listed operations have the following semantics.

callUser: Lets processes send a user task request. The request must contain a correlation ID (as explained above), and a task structure with two elements: `peopleLink` and `userTasks`. The task structure is contains all data necessary for the user client, i. e. tasks with variable data and its type definition.

getRandomUUID: Returns a random UUID. (The purpose has been described above.)

login: Let a user log in with username and password. The operation returns a session ID that is used to authorize all other requests subsequent to the login.

logout: Logs the user out from the user manager. (A valid session ID is required.)

getMessages: Get all unfinished task for this user. (A valid session ID is required.) This operation returns all unfinished tasks for this user. If there are no such tasks the response is empty. The user client has to check for its own and by means of the UUID if a task has already been received.

sendMessage Send a finished task back. (A valid session ID is required.) Directly after the task is received by the user manager it tries to send the task back to the right BPEL process.

5.2.2 Program Structure

A registry class in the user manager maintains objects of the class *User*. This class holds information about a user and what tasks he has to perform, if any. A diagram of the interface is shown in figure 5.2 on the following page. In this implementation the user registry very simple. Per default it has two fixed users to allow studies. Of course, the registry can be extended to support access to external user directories.

Incoming tasks from processes have to be assigned to a user. If a responsible user was found, an instance of the class *Task* is created and added to the respective user object. This class holds a) the incoming task structure, which has to be processed by a user (`content`), b) correlation data ID (field `uuid`), c) a finished task that has to be sent back to the process (`reply`), and d) a reply address to the BPEL process (`processEPR`). The class diagram is in figure 5.3 on the next page.

A class diagram with all important classes of the user manager is on page 57 in the appendix.

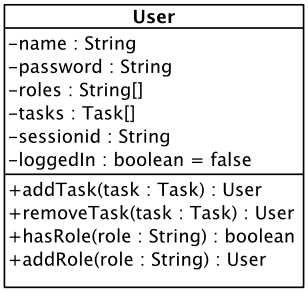

Figure 5.2: User manager: This class maintains information about a single user, which encompasses—among others—a name, password, roles the user belongs to, and a list of tasks.

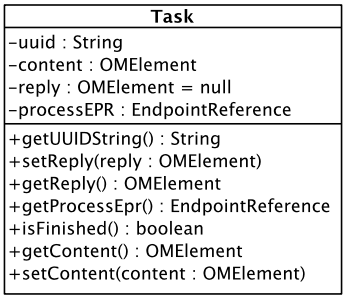

Figure 5.3: User manager: This class maintains task data that came in from a process, and the finished version of the task.

5.3 User Client

While the previous two chapters do not mention how the user client has to be implemented, now a decision must be made. There were two alternative implementation options to choose from:

1. Web application using J2EE
2. Java Swing application

I opted for Java Swing because of the following reasons:

- Java Swing provides a reasonable set of ready GUI components, whereas Web applications must be laboriously coded in HTML.
- A Web controller is harder to manage than a Java Swing controller.
- Java Swing applications are far more responsive than Web applications.

The user client consists of a framing user interface which can receive tasks from the user manager to display them to the user. A generated user interface for a task is called “task editor”.

5.3.1 GUI Generation

The user interface for the tasks is created, as described in chapter 3, by recursive traversal of the XML schema definitions. It is suggestive to construct the GUI similar to the structure of the schema definition. To gain flexibility the composite design pattern was chosen for this purpose [GHJV95].

We define a base class for the composite hierarchy: `AbstractUIComponent`. A UML diagram of this class is shown in figure 5.4 on the following page. The two abstract methods have the following purpose:

getSwing() This method must return a Swing component that displays the contents of the component. For editable data a sub class can create an Swing input component, e. g. a `JTextField`.

getResponseXML() This method must return an XML node that represents the contents of the component. The format of the returned XML data complies with the message type definition of the user manager. A component for the task type `userTasks` will create the complete response data for the process.

Because these two methods are abstract, a sub class must implement them. The other methods are mostly intended for composite sub classes, not for leaf classes.

If a subclass of `AbstractUIComponent` is a composite, then it will call the getSwing() or getResponseXML() methods of all its children to build its Swing component or response XML data, respectively. The effect is that the complete component tree is traversed if one of these two methods is called in the root component.

As described in the algorithm in listing 3.13 on page 24, for each XML schema element a GUI component is created. Each XML simple is mapped to a sub class of `AbstractUIComponent` that is suitable to display this type. The described algorithm is part of the class `XMLSchemaGUIBuilder` (see the UML diagram on page 58 for reference).

After a schema definition was processed, the algorithm returns one sub class of `AbstractUIComponent`. Now the framing user interface can display the generated GUI by calling the method `getSwing()`.

Has the user finished a task, the framing user interface will call the method `getResponseXML()` to get the response message and send it to the user manager by calling the operation `sendMessage()` (see section 5.2 on page 34).

An example view of the user client is shown in figure 5.6 on page 39. The framing user interface is displaying a task that has nested subtasks. The main task is convfigured to arrange its subtasks in a tabbed pane.

Software used for the Implementation

The following software is used for the implementation:

- For the conversions, the Saxon 8.8 XSLT processor is used [Kay07].

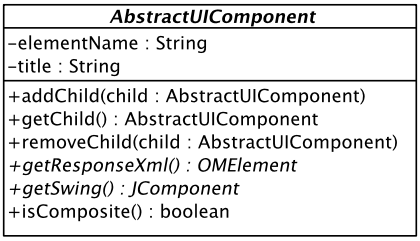

Figure 5.4: User client: This abstract class is the base class of a composite structure that stores the generated GUI of a task.

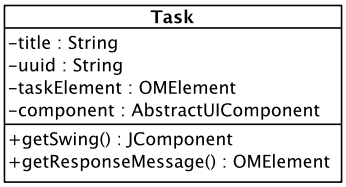

Figure 5.5: User client: This class maintains a task, i. e. its generated GUI and administrative data like the ID. The GUI component can be retrieved by the main GUI of the user client.

- Apache Ant is used for the automation of deployment and conversion [Fou06].
- The user manager is implemented in Java, using the Axis2 Web service engine [Fou07].
- The user client is implemented in Java 1.6, using the new `GroupLayout` layout manager for the Swing GUI and Axis2 for Web client functions.
- As a BPEL engine, ActiveBPEL 3.1 is used [Act07].

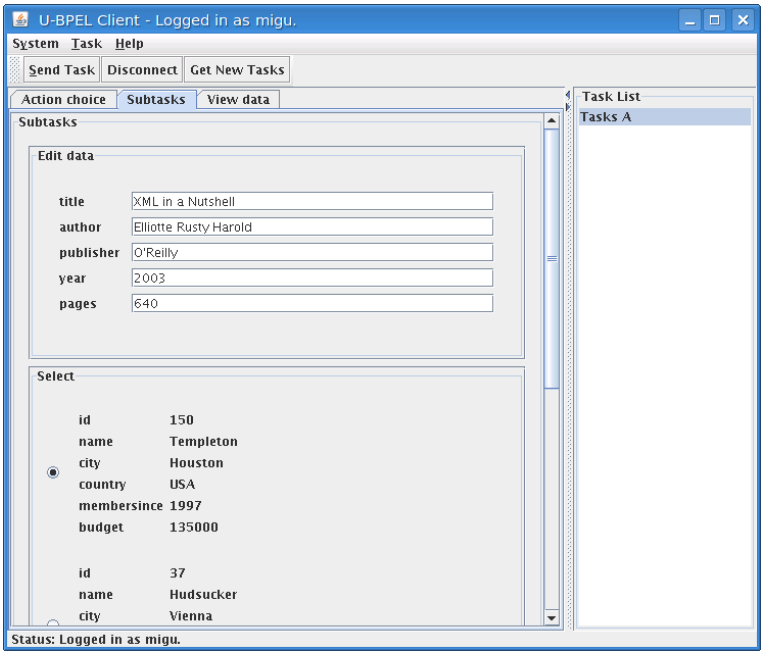

Figure 5.6: User client displaying a nested example task.

Chapter 6

Conclusion and Outlook

In this chapter, the concept and the prototypical implementation are summarized. After the summary, an outlook to possible future work given.

6.1 Summary

The presented BPEL extension allows for user interaction in BPEL processes by using a new type of activity which is named “user”. Five types of “user tasks” can be joined to a nested structure of tasks, of which the user client generates a GUI to let the user edit or view process data.

In order to execute a process with user activities, the process first must be converted to real BPEL. The different files necessary for process deployment, like the WSDL, the deployment descriptor, and the WSDL catalog, also must be converted for this purpose. The conversion is done by several XSLT tranformation programs, each for the respective file type.

The user interface is automatically generated from schema definitions.

6.2 Conclusion

Both the conversion and the user interface generation work. However, there are some open issues that have to be solved.

1. If a process contains more than one user activity, the first is finished without problems, but the second call to the user manager fails because of a wrong SOAP header. The reason for this behavior is unknown.
2. Only a limited subset of schemas is supported by the UI generation algorithm.

6.3 Outlook

There are some possibilities for future work:

- The development of processes with user interaction would be easier if tools like eclipse or the ActiveBPEL Designer would support the conversion step. For now, a fair amount of hand work is needed to get the process running.
- A real BPEL extension would save the conversion. An extension could be developed on the basis of BPEL's extension features.

Bibliography

[ACD+03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. *Business Process Execution Language for Web Services - Version 1.1*, May 2003.

[Act07] ActiveEndpoints. ActiveBPEL Engine. http://www.activebpel.org, 2007. 2007-04-10.

[BHM+04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. http://www.w3.org/TR/ws-arch/, February 2004.

[DJMZ05] Wolfgang Dostal, Mario Jeckle, Ingo Melzer, and Barbara Zengler. *Service-orientierte Architekturen mit Web Services*. Spektrum Akademischer Verlag, 2005.

[Fou06] Apache Software Foundation. Apache Ant. http://ant.apache.org/, December 2006. 2007-02-20.

[Fou07] Apache Software Foundation. Apache Axis2. http://ws.apache.org/axis2/, January 2007. 2007-03-10.

[FW04] David C. Fallside and Priscilla Walmsley. *XML Schema Part 0: Primer Second Edition*. W3C, October 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison Wesley, 1995.

[Hav05] Michael Havey. *Essential Business Process Modeling*. O'Reilly, 1st edition, August 2005.

[HM03] Elliotte R. Harold and W. Scott Means. *XML in a Nutshell. Deutsche Ausgabe.* O'Reilly, 2003.

[JE07] Diane Jordan and John Evdemon. *Web Services Business Process Execution Language Version 2.0 - Committee Specification*, January 2007.

[Kay07] Michael Kay. Saxon xslt processor. http://saxon.sourceforge.net/, January 2007. 2007-02-20.

[KKL+05] Matthias Kloppmann, Dieter Koenig, Frank Leymann, Gerhard Pfau, Alan Rickayzen, Claus von Riegen, Patrick Schmidt, and Ivana Trickovic. WS-BPEL Extension for People - BPEL4People. July 2005.

[LLK04] Patrick Lay and Stefan Lüttringhaus-Kappel. Transforming XML schemas into Java Swing GUIs. In Peter Dadam and Manfred Reichert, editors, *GI Jahrestagung (1), INFORMATIK 2004 - Informatik verbindet, Band 1, Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20. September - 24. September 2004 in Ulm*, volume P-50 of *LNI*, pages 271–276. GI, 2004.

[Lü05] Tim Lüecke. Development of a concept for creating and managing user-interfaces bound to business processes. Master's thesis, Universität Hannover, November 2005.

[MBM06] C. Matthew MacKenzie, Ken Laskey Francis McCabe Peter F. Brown, and Rebekah Metz. *Reference Model for Service Oriented Architecture 1.0.* OASIS, http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html, 1.0 edition, October 2006.

[W3C] W3C. World Wide Web Consortium. http://www.w3.org/. 2007-04-16.

List of Figures

2.1 Web services roles 7
2.2 Directory structure of a BPR 8

3.1 User activity with combined tasks 12
3.2 View of the schema definition for the U-BPEL activity `user` 16
3.3 Sketch of an edit form 23

4.1 Architecture Layers 25
4.2 Interaction between the three layers 29

5.1 Principle of the U-BPEL conversion 30
5.2 User manager: This class maintains information about a single user, which encompasses—among others—a name, password, roles the user belongs to, and a list of tasks. 36
5.3 User manager: This class maintains task data that came in from a process, and the finished version of the task. 36
5.4 User client: This abstract class is the base class of a composite structure that stores the generated GUI of a task. 38
5.5 User client: This class maintains a task, i. e. its generated GUI and administrative data like the ID. The GUI component can be retrieved by the main GUI of the user client. 38
5.6 User client displaying a nested example task. 39

C.1 Classes of the user manager 57
C.2 Classes and packages of the user client 58

Listings

2.1 Example XML Schema definition 9
2.2 Redefinition of a schema element 10
3.1 Example of a U-BPEL user activity 16
3.2 Addressing a user directly 17
3.3 Addressing a user with an XPath expression 18
3.4 Syntax of task type *userTasks* 18
3.5 Syntax of task type *editTask* 18
3.6 Syntax of task type *selectionTask* 19
3.7 Syntax of task type *controlTask* 19
3.8 Syntax of task type *visualTask* 20
3.9 Different task types in a user activity 20
3.10 Converted user activity 21
3.11 Schema definition for an element *book* 23
3.12 Example *book* element 23
3.13 Principle of an algorithm for GUI generation 24
5.1 Schema definition with referenced complex type 32
5.2 Contiguous schema definition 32
5.3 A XSLT template which recursively copies all elements of a XML document. Only elements for that a template exists are left out. 33
A.1 U-BPEL schema definition 47
A.2 Schema definition for the user manager 49
B.1 WSDL file of the user manager 55

List of Tables

5.1 Web service operations provided by the user manager 34

Appendix A

Schema Definitions

A.1 U-BPEL

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE types [
  <!ENTITY bpel20 "http://docs.oasis-open.org/wsbpel/2.0/">
]>
<xs:schema
  targetNamespace="&bpel20;process/executable"
  xmlns="&bpel20;process/executable"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:bpel="&bpel20;process/executable"
  elementFormDefault="qualified" attributeFormDefault="unqualified"
  blockDefault="#all">

  <xs:redefine
    schemaLocation="&bpel20;process/executable/ws-bpel_executable.xsd">
    <xs:group name="activity">
      <xs:choice>
        <xs:group ref="activity" />
        <xs:element name="user" type="tUser" />
      </xs:choice>
    </xs:group>
  </xs:redefine>

  <xs:complexType name="tUser">
    <xs:sequence>
      <xs:element name="peopleLink" type="tPeopleLink" />
      <xs:element ref="userTasks" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="userTasks" type="tUserTasks" />
  <xs:complexType name="tUserTasks">
    <xs:choice maxOccurs="unbounded">
      <xs:choice>
        <xs:element name="visualTask" type="tVisualTask" />
```

```
      <xs:element name="controlTask" type="tControlTask" />
      <xs:element name="editTask" type="tEditTask" />
      <xs:element name="selectionTask" type="tSelectionTask" />
    </xs:choice>
    <xs:element ref="userTasks" />
  </xs:choice>
  <xs:attribute name="title" type="xs:string" use="optional" />
  <xs:attribute name="type" type="tContainerType" use="optional"
    default="simple" />
</xs:complexType>

<xs:complexType name="tVisualTask">
  <xs:attribute name="variable" type="bpel:BPELVariableName" />
</xs:complexType>

<xs:complexType name="tControlTask">
  <xs:sequence>
    <xs:element name="action" minOccurs="2" maxOccurs="unbounded">
      <xs:complexType>
        <xs:attribute name="name" type="xs:string" use="required" />
        <xs:attribute name="description" type="xs:string" />
        <xs:attribute name="id" type="xs:positiveInteger"
          use="required" />
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="resultVariable" type="bpel:BPELVariableName" />
</xs:complexType>

<xs:complexType name="tEditTask">
  <xs:attribute name="variable" type="bpel:BPELVariableName" />
</xs:complexType>

<xs:complexType name="tSelectionTask">
  <xs:attribute name="choiceVariable" type="bpel:BPELVariableName"
    use="required" />
  <xs:attribute name="resultVariable" type="bpel:BPELVariableName"
    use="required" />
  <xs:attribute name="multiple" type="xs:boolean" default="no" />
</xs:complexType>

<xs:complexType name="tPeopleLink">
  <xs:sequence>
    <xs:element name="query" minOccurs="0" maxOccurs="1"
      type="xs:string" />
  </xs:sequence>
  <xs:attribute name="method" type="tPeopleMappings"
                use="required" />
  <xs:attribute name="value" type="xs:string" />
  <xs:attribute name="variable" type="bpel:BPELVariableName" />
  <xs:attribute name="part" type="xs:NCName" />
</xs:complexType>
```

```
  <xs:simpleType name="tPeopleMappings">
    <xs:restriction base="xs:string">
      <xs:enumeration value="name" />
      <xs:enumeration value="role" />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="tContainerType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="simple" />
      <xs:enumeration value="frame" />
      <xs:enumeration value="tabs" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Listing A.1: U-BPEL schema definition

A.2 User Manager

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://www.ubpel.org/userManager/"
  xmlns="http://www.ubpel.org/userManager/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:um="http://www.ubpel.org/userManager/"
  elementFormDefault="qualified">

  <!-- request for a user activity to the user manager -->
  <xsd:element name="request">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="correlationData" type="xsd:string" />
        <xsd:element name="peopleLink" type="um:tPeopleLink" />
        <xsd:element name="userTasks" type="um:tUserTasksRequest" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="tUserTasksRequest">
    <xsd:choice maxOccurs="unbounded">
      <xsd:choice>
        <xsd:element name="visualTaskRequest">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="object" type="xsd:anyType" />
              <xsd:element name="schema" type="um:tAnySchema" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="controlTaskRequest">
          <xsd:complexType>
```

```
            <xsd:sequence>
              <xsd:element name="action" minOccurs="2"
                maxOccurs="unbounded">
                <xsd:complexType>
                  <xsd:attribute name="name" type="xsd:string"
                    use="required" />
                  <xsd:attribute name="description"
                                 type="xsd:string" />
                  <xsd:attribute name="id" type="xsd:positiveInteger"
                    use="required" />
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="editTaskRequest">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="object" type="xsd:anyType" />
              <xsd:element name="schema" type="um:tAnySchema" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="selectionTaskRequest">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="choice" type="xsd:anyType" />
              <xsd:element name="schema" type="um:tAnySchema" />
            </xsd:sequence>
            <xsd:attribute name="multiple" type="xsd:boolean"
              default="no" />
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
      <xsd:element name="userTasks" type="um:tUserTasksRequest" />
    </xsd:choice>
    <xsd:attribute name="title" type="xsd:string" use="optional" />
    <xsd:attribute name="type" type="um:tContainerType" use="optional"
      default="simple" />
  </xsd:complexType>

  <xsd:complexType name="tPeopleLink">
    <xsd:attribute name="method" type="um:tPeopleMappings"
                   use="required" />
    <xsd:attribute name="value" type="xsd:string" use="required" />
  </xsd:complexType>

  <xsd:simpleType name="tPeopleMappings">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="name" />
      <xsd:enumeration value="role" />
    </xsd:restriction>
```

```
</xsd:simpleType>

<xsd:simpleType name="tContainerType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="simple" />
    <xsd:enumeration value="frame" />
    <xsd:enumeration value="tabs" />
  </xsd:restriction>
</xsd:simpleType>

<!-- response to the process -->
<xsd:element name="response">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="correlationData" type="xsd:string" />
      <xsd:element name="userTasks" type="um:tUserTasksResponse" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="tUserTasksResponse">
  <xsd:choice maxOccurs="unbounded">
    <xsd:choice>
      <xsd:element name="visualTask">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="object" type="xsd:anyType" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="controlTask">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="action" type="xsd:positiveInteger" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="editTask">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="object" type="xsd:anyType" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="selectionTask">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="selection" type="xsd:anyType" />
          </xsd:sequence>
          <xsd:attribute name="multiple" type="xsd:boolean"
            default="no" />
        </xsd:complexType>
```

```
      </xsd:element>
    </xsd:choice>
    <xsd:element name="userTasks" type="um:tUserTasksResponse" />
  </xsd:choice>
</xsd:complexType>

<!-- general type -->
<xsd:complexType name="tAnySchema">
  <xsd:sequence>
    <xsd:any minOccurs="1" maxOccurs="1"
      namespace="http://www.w3.org/2001/XMLSchema" />
  </xsd:sequence>
</xsd:complexType>

<!-- UUID request for correlation data in the process -->
<xsd:element name="UUIDrequest">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="in" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="UUIDresponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="out" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<!-- PART 2: communication between user manager and user client -->

<xsd:element name="login">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="username" type="xsd:string" />
      <xsd:element name="password" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="loginack">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="username" type="xsd:string" />
      <xsd:element name="sessionid" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="logout">
  <xsd:complexType>
```

```
      <xsd:sequence>
        <xsd:element name="username" type="xsd:string" />
        <xsd:element name="sessionid" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="logoutack">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="sessionid" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="getMessages">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="sessionid" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="getMessagesReply">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="sessionid" type="xsd:string" />
        <xsd:element name="task">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="uuid" type="xsd:string" />
              <xsd:element name="task" type="um:tUserTasksRequest"
                minOccurs="0" maxOccurs="unbounded" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="sendMessage">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="sessionid" type="xsd:string" />
        <xsd:element name="task">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="uuid" type="xsd:string" />
              <xsd:element name="task" type="um:tUserTasksResponse" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
```

```
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="sendMessageAck">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="sessionid" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing A.2: Schema definition for the user manager

Appendix B

WSDL Definitions

B.1 User Manager

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions
  targetNamespace="http://www.ubpel.org/userManager/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:um="http://www.ubpel.org/userManager/"
  xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
  xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.ubpel.org/userManager/"
        schemaLocation="UserManagerService.xsd" />
    </xsd:schema>
  </types>
  <message name="UserRequestMessage">
    <part name="request" element="um:request" />
  </message>
  <message name="UserResponseMessage">
    <part name="response" element="um:response" />
  </message>
  <message name="getUUIDMessage">
    <part name="msg" type="xsd:string" />
  </message>
  <message name="login">
    <part name="login" type="um:login" />
  </message>
  <message name="loginack">
    <part name="loginack" type="um:loginack" />
  </message>
  <message name="logout">
    <part name="logout" type="um:logout" />
```

```
  </message>
  <message name="logoutack">
    <part name="logoutack" type="um:logoutack" />
  </message>
  <portType name="UserManagerPT">
    <operation name="callUser">
      <input message="um:UserRequestMessage" />
    </operation>
    <operation name="getRandomUUID">
      <input message="um:getUUIDMessage" />
      <output message="um:getUUIDMessage" />
    </operation>
  </portType>
  <binding name="UserManagerBinding" type="um:UserManagerPT">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="callUser">
      <soap:operation soapAction="urn:callUser" />
      <input>
        <soap:body use="literal" />
      </input>
    </operation>
    <operation name="getRandomUUID">
      <soap:operation soapAction="urn:getRandomUUID" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>
  <service name="UserManagerService">
    <port name="UserManagerPT" binding="um:UserManagerBinding">
      <soap:address location="..." />
    </port>
  </service>

  <vprop:property name="umCorrelationData" type="xsd:string" />
  <vprop:propertyAlias propertyName="um:umCorrelationData"
    messageType="um:UserRequestMessage" part="request">
    <vprop:query>um:correlationData</vprop:query>
  </vprop:propertyAlias>
  <vprop:propertyAlias propertyName="um:umCorrelationData"
    messageType="um:UserResponseMessage" part="response">
    <vprop:query>um:correlationData</vprop:query>
  </vprop:propertyAlias>
</definitions>
```

Listing B.1: WSDL file of the user manager

Appendix C

Class Diagrams

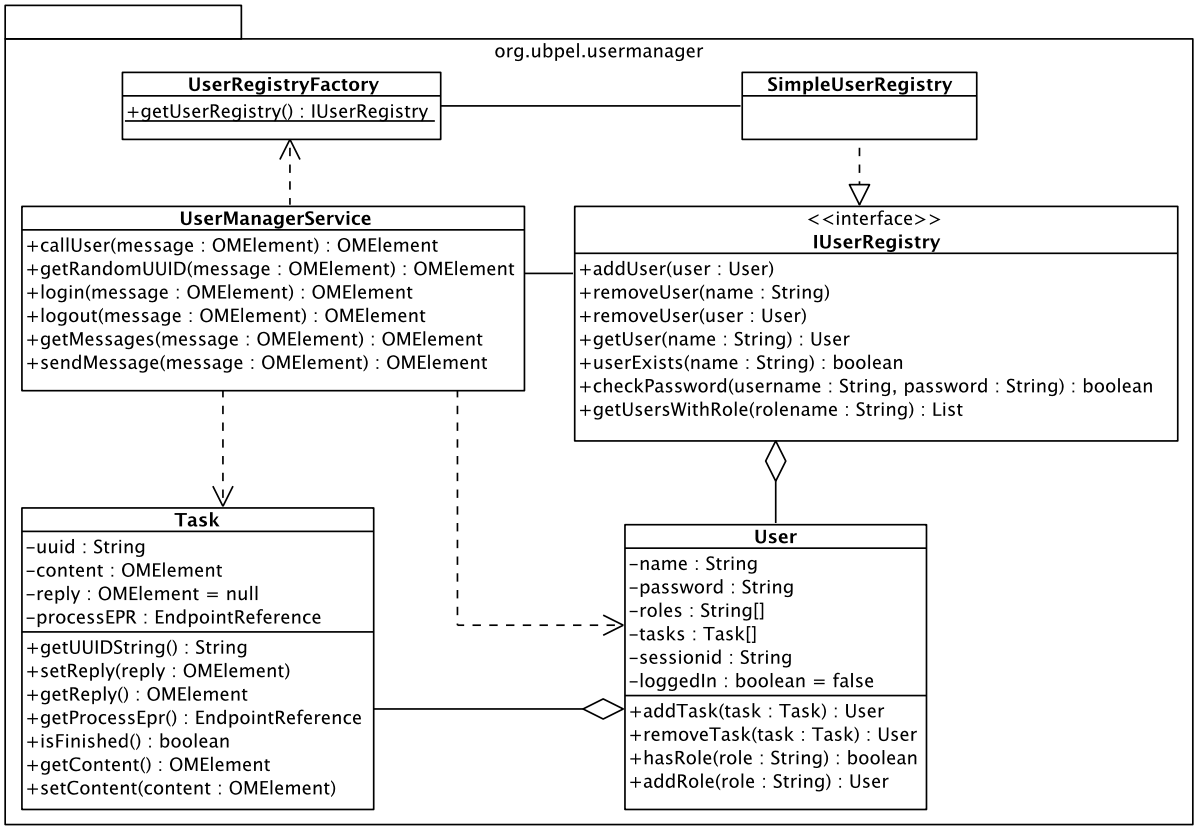

Figure C.1: Class structure of the user manager

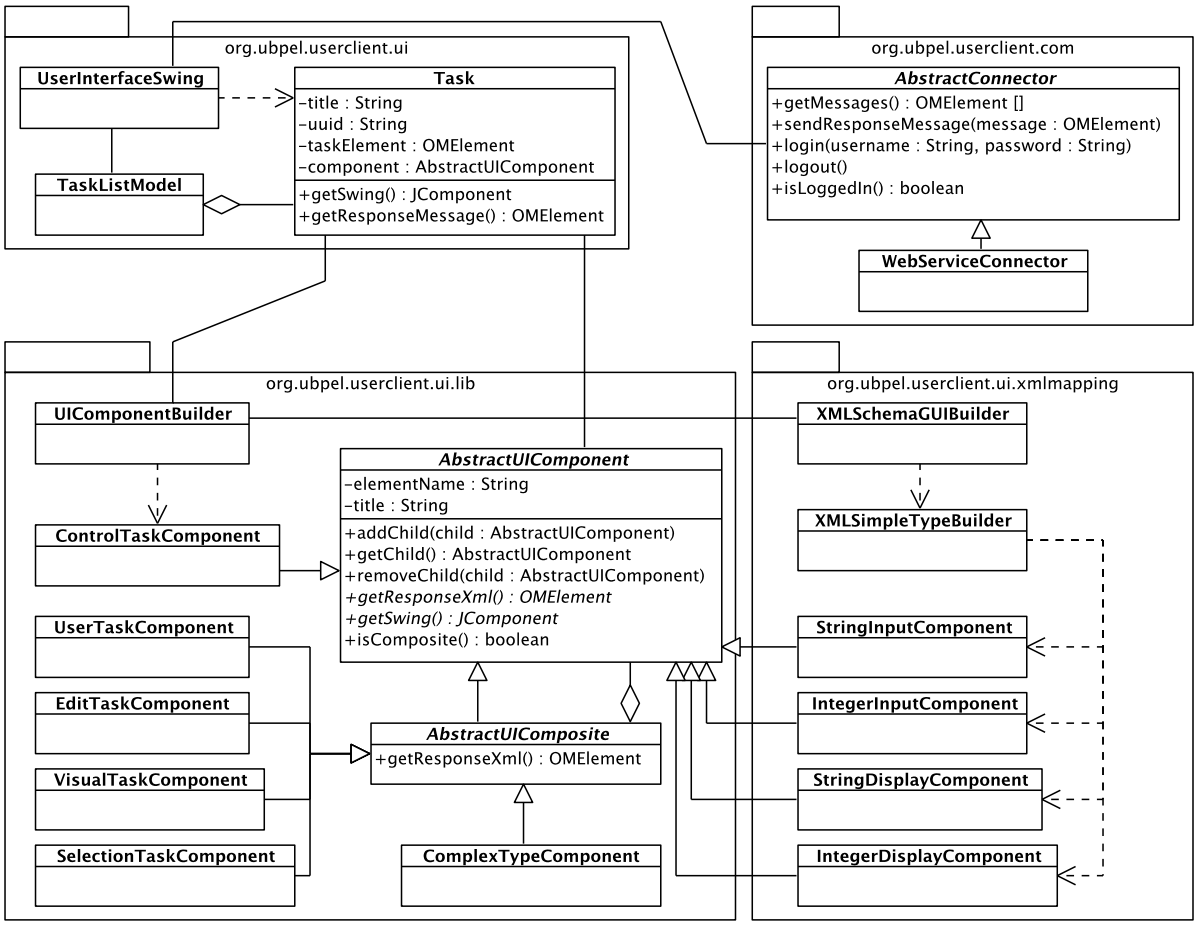

Figure C.2: Class and package structure of the user client

Erklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe.

Hannover, 18. April 2007

_____

Michael Gutbier