Universidade do Minho
Escola de Engenharia

André Lopes Barbosa

Pattern based user interface generation

Novembro de 2012

Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Lopes Barbosa

Pattern based user interface generation

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor António Nestor Ribeiro

Novembro de 2012

Abstract

Human Computer Interaction (HCI) is one of the most important aspects in software development. In order to produce valuable products, software companies are focusing more on the users and less on the technology behind their products. This calls for new prospects for development cycles. Traditional methodologies are focused on the internals and there is little support to build a User Interface (UI) in a more iterative manner [12].

Model Driven Development (MDD) [21] is a technique that has been used to increase software quality and boost development time. With MDD organizations are able to implement iterative development methodologies that start with high level models that are iteratively transformed into lower level models and ultimately source code in an automated way. High level models have several advantages because they are platform independent, easier to maintain, easier to reuse and ultimately they serve as documentation for the project.

Unified Modelling Language (UML) is an industry standard language for modelling software. The problem with UML is that it's not fit for UI models [4]. The UI requires a new modelling language that is able to represent UI aspects accurately. The HCI community came up with several solutions for this problems, ITS [28], WISDOM [16], Unified Modeling Language for Interactive Applications (UMLi) [4] and USer Interface eXtensible Mark-up Language (UsiXML) [23] are some examples on this matter.

This work proposes a method to reuse previous UI knowledge using patterns of high level models. The goal of this work is to improve the way developers build UI's and maximize re-usability. Patterns are tested and robust solutions that have been used in other contexts and can even persist between different projects and teams. This work integrates in the Forward Engineering method (FEM) developed by the UsiXML community and uses the UsiXML User Interface Description Language (UIDL) to represent patterns of high level UI models.

We developed a pattern definition using a set of descriptive fields and UsiXML models. With the information provided by the pattern we are able to perform model transformations from the domain and task models to an Abstract User Interface (AUI) model. This gives developers the ability to reuse the structure of a UI developed in other context with a similar objective. This makes it easier for developers with little knowledge in HCI to develop good UI's and also helps development teams to maintain consistency across an application.

Resumo

A Human Computer Interaction (HCI) assumiu-se como um dos aspectos mais importantes no desenvolvimento de software. Por forma a criar produtos com maior qualidade, as empresas de software estão a desviar o foco da tecnologia para os utilizadores, exigindo novas abordagens para os ciclos de desenvolvimento. As metodologias tradicionais são mais focadas em aspectos internos e há pouco suporte para construir a User Interface (UI) de forma iterativa [12].

Model Driven Development (MDD) [21] é uma técnica usada para aumentar a qualidade do software e diminuir o tempo de desenvolvimento. Com a MDD as organizações são capazes de implementar metodologias de desenvolvimento iterativo que começam com modelos de alto nível que são transformados, iterativamente, em modelos de nível inferior e, até chegar ao código-fonte. Modelos de alto nível têm várias vantagens porque são independentes de plataforma, mais fáceis de manter, mais fáceis de reutilizar e, finalmente, servem como documentação do projeto.

Unified Modelling Language (UML) é uma linguagem padrão da indústria para modelação de software. O problema do UML é que não está apto para modelos de UI [4]. A UI requer uma linguagem de modelação nova, que seja capaz de representar os aspectos próprios da UI com precisão. A comunidade à volta da HCI desenvolveu várias soluções para este problema, ITS [28], WISDOM [16], Unified Modeling Language for Interactive Applications (UMLi) [4] e USer Interface eXtensible Markup Language (UsiXML) [23] são alguns exemplos de projetos dentro desta área.

Este trabalho propõe um método para reutilizar conhecimento passado sobre a UI usando padrões de modelos de alto nível. O objetivo deste trabalho consiste em abordar melhorias à forma como se constroem UI 's e maximizar a sua reutilização. Padrões são soluções testadas e robustas, que foram utilizadas em outros contextos e podem até persistir entre diferentes projetos e equipas. Este trabalho integra-se no método Forward Engineering (FEM) desenvolvido pela comunidade do UsiXML e usa a User Interface Description Language (UIDL) do UsiXML para representar padrões de modelos de UI de alto nível.

Neste trabalho desenvolveu-se uma definição de padrão usando um conjunto de campos descritivos e modelos em UsiXML. Com a informação fornecidas pelos padrões é possível realizar transformações de modelos a partir dos modelos de domínio e tarefa e gerar um modelo de Abstract User Interface (AUI). Isto oferece aos engenheiros de software a capacidade de reutilizar a estrutura de uma UI desenvolvida noutro contexto com um objetivo similar. Assim torna-se mais fácil para engenheiros com pouco conhecimento em HCI desenvolver boas UI 's e também ajuda as equipes de desenvolvimento a manter consistência ao longo de uma aplicação.

Acknowledgements

First of all I want to thank my supervisor, Professor António Nestor for his guidance, patience, trust and support, even when that meant working nights. I also want to say thank you to Professor José Creissac for providing me with all the documentation developed by the UsiXML community.

Thank you to all my friends for your support, understanding and good humour. Above all, thank you for helping me keep my sanity. Special thanks to Samuel for his companionship during this long summer spent at the university. Next dinner is on me.

Thank you to my family for always making me feel special. Thank you to my grandfather for teaching me the value of hard work. Thank you to my mother for her support and delicious food. Thank you to my father for never letting me give up. Thank you to my brothers for making me feel so proud. Thank you to my goddaughter for being so cute.

Contents

1 Introduction **1**

1.1 Motivation . . . 2

1.2 Objective . . . 3

1.3 Dissertation Outline . . . 3

2 Background **7**

2.1 Early days . . . 9

2.1.1 ITS . . . 10

2.2 Optimization-based generation of interfaces . . . 11

2.2.1 Gadget . . . 11

2.2.2 Supple . . . 13

2.3 User interface patterns . . . 15

2.3.1 IdealXML . . . 16

2.4 Specification languages . . . 18

2.4.1 UMLi . . . 19

2.4.2 WISDOM . . . 22

2.4.3 UsiXML . . . 23

2.5 Discussion . . . 26

3 UsiXML **29**

3.1 UsiXML semantics . . . 30

3.2 UsiXML method . . . 31

3.2.1 Spem4UsiXML . . . 32

3.2.2 Forward engineering method . . . 36

4 Patterns in UsiXML **39**

4.1 Concept . . . 40

4.2 Pattern definition . . . 41

4.3 Example 43
4.4 Requirements 46
4.5 Algorithm 48

5 UsiXML patterns tool 51

5.1 Package org.usixml 52
5.2 Package org.usixml.aui 53
5.3 Package org.usixml.domain 54
5.4 Package org.usixml.task 56
5.5 Package usixmlpatterns 58
5.6 Implementation details 59

6 Case study 63

6.1 Patterns 63
6.1.1 List Documents Pattern 64
6.1.2 Create document pattern 66
6.1.3 Edit document pattern 69
6.2 Case study models 71
6.2.1 List Prescriptions 72
6.2.2 Create Prescription 74
6.2.3 Edit Prescription 78
6.3 Case Study Implementation 81

7 Conclusions and Future Work 85

A SPEM 2.0 93

List of Figures

2.1 Login window modelled in UMLi 19

3.1 UsiXML specification as an UML class diagram. 30
3.2 Structure of the Spem4UsiXML meta-model. 33
3.3 Structure of the Spem4UsiXML method content meta-model package. . . 33
3.4 Structure of the Spem4UsiXML process structure meta-model package. . 35
3.5 The forward engineering method. 37

4.1 The first stage of the forward engineering method. 40
4.2 Proposal for the first stage of the forward engineering method. 41
4.3 UML specification of user interface conceptual pattern. 42
4.4 UML specification of the *Create document* pattern's domain model. . . . 43
4.5 Task model for the *Create document* pattern. 44
4.6 Abstract User Interface model for the *Create document* pattern. 45
4.7 UML specification of an invoice's domain model. 45
4.8 Task model to create an invoice. 46
4.9 Abstract User Interface model to create an invoice. 47

5.1 The org.usixml package. 52
5.2 The org.usixml.aui package. 53
5.3 The org.usixml.domain package. 55
5.4 The org.usixml.task package. 57
5.5 The usixmlpatterns package. 58
5.6 The UsiXMLPatterns User Interface (UI). 61
5.7 The UsiXMLPatterns UI to choose a path in the local file system. 61
5.8 Junit tests on Netbeans. 62

6.1 UML specification of a document domain model. 64
6.2 *List documents* pattern's task model. 65

6.3 *List documents* pattern's task model. 66
6.4 *Create document* pattern's task model. 67
6.5 *Create document* pattern's task model. 68
6.6 *Edit document* pattern's task model. 70
6.7 *Edit document* pattern's task model. 71
6.8 UML specification of a medical prescription domain model. 71
6.9 List prescriptions task model. 72
6.10 Using UsiXMLPatterns to genera the AUI model to list prescriptions. . . 73
6.11 AUI model to list prescriptions. 74
6.12 Task model to create a medical prescription - part 1. 75
6.13 Task model to create a medical prescription - part 2. 76
6.14 Task model to create a medical prescription - part 3. 76
6.15 AUI model to create a prescription. 77
6.16 Task model to edit a prescription - part 1. 79
6.17 Task model to edit a prescription - part 2. 79
6.18 Task model to edit a prescription - part 3. 80
6.19 Task model to edit a prescription - part 4. 80
6.20 AUI model to edit a prescription. 81
6.21 Prescriptions List. 82
6.22 New Prescription. 83

A.1 Structure of the SPEM 2.0 meta-model. 94
A.2 Structure of the SPEM 2.0 method content meta-model package. 95
A.3 Main classes and associations of SPEM 2.0 process structure package package. 97

Acronyms

AUI Abstract User Interface

BHW Better Hand Writing

CTT Concur Task Trees

DRY Don't Repeat Yourself

FEM Forward Engineering method

GUI Graphical User Interface

HCI Human Computer Interaction

IDE Integrated Development Environment

MB-UIDE Model Based - User Interface Development Environment

MDD Model Driven Development

MDDUI Model Driven Development of User Interfaces

MDE Model Driven Engineering

OMG Object Management Group

SDK Software Development Kit

SPEM Software & Systems Process Engineering Metamodel

SPEM4UsiXML SPEM for UsiXML

UI User Interface

UIDL User Interface Description Language

UML Unified Modelling Language

UMLi Unified Modeling Language for Interactive Applications

UsiXML	USer Interface eXtensible Mark-up Language
WISDOM	Whitewater Interactive System Development with Object Models
XML	eXtensible Mark-up Language

Chapter 1

Introduction

Nowadays what makes a software product stand out is getting less technological and more related to how it handles the Human Computer Interaction (HCI). This may not be true to every software product but it's true for most of them. Usability aspects can't be neglected in any software and this has brought a lot of work in this field by both the industry and academic communities.

Recent years have seen the proliferation of many types of computers and devices. In order to perform their tasks, people now have available a wide variety of computational devices ranging from traditional computers and laptops to mobile phones, tablets and even televisions. While this increasing proliferation of fixed and mobile devices fits with the need for ubiquitous access to information processing, this diversity offers new challenges to the HCI software community like building and maintaining versions of single applications across multiple devices.[3]

Model Driven Development (MDD) defining characteristic is that software development's primary focus and products are models rather than computer programs. The major advantage of this is that we express models using concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain relative to most popular programming languages [21].

Models are easier to maintain than the code itself and, most importantly, they're platform independent. This means that the same model can be used to generate code that runs on a desktop environment, a web environment or even a mobile environment. This makes a lot of sense for user interfaces because modern applications are becoming more and more ubiquitous and it's highly complex and time consuming to build a Graphical User

Interface (GUI) for every supported platform.

MDD is a good solution for the ubiquity problem but alone can't solve all HCI concerns that are as much important. Usability comes from experience and experimentation. Building intuitive applications isn't an easy task. Usually it's an iterative process where developers observe the needs of the users, build a version, put it to test with real users, improve it and repeat the process. Of course this isn't always possible because it takes a lot a time and money.

By combining MDD of user interfaces and promoting the reuse of past developments these two problems can be tackled at once. Patterns are widely used in several fields of engineering (including software engineering) but are still a bit underutilized in the HCI community. Most patterns repositories only contain implementation level models and are designed to be read by humans instead of being processed by computers. This means there is a gap in the utilization of patterns. There's a lack of more high level pattern definitions that could be used in the MDD process and help and serve as a base knowledge to help developers make decisions and automate models transformations.

1.1 Motivation

Most developers spend a lot of their times designing their applications in a way that they won't have to repeat any code line (Don't Repeat Yourself (DRY)) even between separated projects. When it comes to the User Interface (UI) things can sometimes get out of hand. This happens mainly because most developers don't start building UI's the same way they build the core of their applications, that is modelling first. The lack of UI models many times result in poor reuse of previous knowledge and sometimes even in poor UI's. This happens mainly because there isn't a standard language for modelling UI's (like Unified Modelling Language (UML)).

USer Interface eXtensible Mark-up Language (UsiXML) is a new and powerful language that has the potential to change the *status quo* and become a standard for modelling the UI in a MDD compliant way with support for different phases and levels of abstraction.

Having solved the problem of creating models, the next logical step is to create tools suited to help developers reuse models and previously acquired knowledge. In other words, tools to improve existing Model Driven Development of User Interfaces (MDDUI) methodologies using patterns.

1.2 Objective

The main objective of this work is to develop a tool that integrates in the MDDUI methodology developed by the UsiXML community (*Forward engineering method* 3.2.2) and uses patterns to automate the first stage of model transformations.

The first transformation of the *Forward engineering method* receive as input a Task and a Domain models and should output an Abstract User Interface (AUI) model.

The objective is to develop a tool that will help developers with this transformation. The creation of the AUI model is one of the most complex models in UI modelling[22]. That's because developers have to specify what components will appear in every kind of *window*. The objective of this work is to use patterns to help in this stage. Instead of leaving the hard work for developers or to create a complex optimization algorithm, our approach is to leverage from previous knowledge and use it to effectively boost the development time and quality.

With this objective in mind this tool have to meet the following requirements:

- read and interpret Task, Domain and AUI models;
- link the Task and Domain developing models to the Task and Domain models of a pattern;
- generate an AUI model based on the information provided by the developing models and the pattern models.

This work intends to contribute with an alternative path for the first phase of the Forward Engineering method (FEM) (section 3.2.2).

Originally the first phase of the method receives a Task and a Domain models as input and outputs an AUI model through manual transformations. This work will contribute with a tool to automate this process by taking advantage of previous knowledge in the form of patterns.

1.3 Dissertation Outline

This work is divided into 7 Chapters. In the first chapter, the current one, we present an introduction to all the work undertaken for this dissertation.

1. **Chapter 2**: presents some background on MDDUI. In this chapter we present several projects that contributed for the development of the state of the art on this subject. We present some early projects that were very important in an historic manner. Projects that propose alternative approaches that are very different from the traditional. And finally we studied more recent projects with similar characteristics that attempt to solve the problem of introducing models into the development of UI's. At the end of this chapter there's a critic discussion of all the projects mentioned above.

2. **Chapter 3**: presents a detailed view over UsiXML. We begin by presenting an overview about the UsiXML project and its objectives. Afterwards we present the semantics of the UsiXML's User Interface Description Language (UIDL), version 1.8. Finally we present the engineering method developed by the UsiXML project. The method is based on a meta-model, SPEM for UsiXML (SPEM4UsiXML) which is itself based on the Software & Systems Process Engineering Metamodel (SPEM) meta-mode that is approached on appendix A. After the meta-model we present the method itself, the FEM.

3. **Chapter 4**: presents a solution to leverage from UsiXML patterns at a conceptual level. Starts with a concept of how this should integrate in the FEM method. After we specify the contents of a pattern in order to satisfy the concept vision. The next section gives an example of how a pattern should look like and how it should be used. Afterwards we specify a set of requirements that should be met in order to implement the concept. The chapter ends with a description of an algorithm to perform the model transformation specified by the concept.

4. **Chapter 5**: presents the implementation of a prototype application that satisfies the requirements of the concept. In this chapter we start by describing a software architecture to satisfy the requirements mentioned in the previous chapter. We end this chapter with some implementation details like the developed UI for the prototype.

5. **Chapter 6**: presents a case study that uses the prototype application. In this chapter we develop models for the UI of an application called Better Hand Writing (BHW) that is meant to manage digital medical prescriptions. We start by stating the requirements of the application and choose the best suited patterns to use in its development.

In the last chapter we present this work's conclusion and where future projects on this work should focus.

Chapter 2

Background

In this chapter is described the state of the art regarding Model Driven Development of User Interfaces (MDDUI). We'll present several techniques to develop User Interface (UI)'s using Model Driven Development (MDD) principles. There is significant ongoing research in this field since the late 1980s and this chapter makes a short summary of that research.

MDD [21] defining characteristic is that software development's primary focus and products are models rather than computer programs. The major advantage of this is that we express models using concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain relative to most popular programming languages.

Models are easier to maintain than the code itself and, most importantly, they're platform independent. This means that the same model can be used to generate code that runs on a desktop environment, a web environment or even a mobile environment. This makes a very important feature for UI's because modern applications are becoming more and more ubiquitous and it's highly complex and time consuming to build a UI for every supported platform.

There are different kinds of models and several techniques to model UI's. Some tools like Janus[2] only require a domain model to generate a user interface while others like Trident[26, 25] and Adept[9] also use a task model. All these approaches automate part of the development process. Other approaches like ITS[28] require more work from the developers because they don't generate any model.

There are also some recent projects using different techniques to generate UI's. Gadget[6] or Supple[7] use numeric optimization techniques to generate better UI's.

Patterns are widely used in every field of engineering. One of the earlier definitions of patterns can be found in [1]. Almost twenty years later patterns were brought to software engineering by Gamma et al. [8].

Patterns bring many advantages, not only they make the development of a product less time consuming and thus less expensive but can also guarantee a higher level of quality because patterns are solutions that have been tested and used in other projects. IdealXML[14, 13] is a tool that helps developers to take advantage of patterns while developing UI models.

Although there have been a lot of academic work surrounding MDDUI, this work is struggling to be adopted by the industry. There are some theories of what's missing, Molina [12] says that better tool support is needed while Montero et al. [14] says that a common language for representing UI models should the next step.

Unified Modelling Language (UML) [20] is the industry standard for software modelling but, unfortunately, is not completely fit to model UI's. Fortunately, the software engineering community has developed some new modelling languages in the past few years to overcome this problem. Unified Modeling Language for Interactive Applications (UMLi)[4] is an extension to UML that provides an alternative diagram notation for describing Abstract User Interface (AUI) models. Whitewater Interactive System Development with Object Models (WISDOM)[16] is an engineering method for interactive software development that includes a notation for UI models.Concur Task Trees (CTT) [18] aims at task modelling by dividing the task model in three essential parts:

- First a hierarchical logical decomposition of the tasks represented by a tree-like structure;
- Then an identification of the temporal relationships among tasks at the same level;
- And finally an identification of the objects associated with each task and of the actions which allow them to communicate with each other.

USer Interface eXtensible Mark-up Language (UsiXML) [23] is a User Interface Description Language (UIDL) aimed at expressing user interfaces built with various modalities of interaction and independently of them. UsiXML is eXtensible Mark-up Language (XML) compliant to enable flexible exchange of information and powerful communication between models and tools used in UI engineering [23]. One of the great advantages of

UsiXML is platform independence providing a multi-path development of user interfaces [11]. UsiXML characteristics and features give it the potential to become a standard for modelling UI's like UML is for software architecture.

There is a lot of work regarding MDDUI and the idea that models can simplify the development process is becoming more consensual. Section 2.1 focus on projects from the beginnings of MDDUI, namely Its, section 2.1.1, which was one of the most important and innovative projects of its time.

Section 2.2 will describe optimization based techniques and the Gadget and Supple projects will be examined with more depth.

Section 2.3 refers to the usage of patterns in UI design . This section also includes an analysis of IdealXML, a knowledge based tool for designing UI's.

Section 2.4 is about specification languages for UI's. We will look into UMLi, WISDOM and UsiXML with some detail.

2.1 Early days

Research in model-based user interface development comes from the 1980s[22]. Their roots come from user interface management systems (UIMS)[15]. These tools seeked to provide an alternative paradigm for constructing interfaces. Rather than using a toolkit library, developers would write a specification in a specialized, high-level specification language. This specification would be automatically translated into an executable program, or interpreted at run-time to generate the appropriate interface.

Through the 1980s and 1990s specification languages became more sophisticated, supporting richer and more detailed representations that allowed systems to generate more sophisticated interfaces.

Since that time there were essentially two approaches to model driven development of user interfaces. Some tried to minimize the work of developers and generate most of the model from just a domain model like in Janus[2] or a task model as in Trident[26, 25]. The second approach was to give more power to the developer letting him produce all or most of the work like in Its[28].

Section 2.1.1 provides a more detailed study on ITS.

2.1.1 ITS

The Its[28] system was developed in the early 90s by the IBM research and development department. It was successfully used to develop several large applications like the information kiosks for Seville Expo 92.

The Its architecture divides applications in four layers. The action layer implements the application's back-end computations. The dialogue layer defines the content of the user interface independent of its style, much like an abstract user interface model. Content specifies the objects included in each frame of the interface, the flow of control among frames, and what actions are associated with each object. The style rule layer defines how the dialogue is presented to the user in terms of appearance and interaction techniques. Finally the style program layer implements the primitive toolkit objects that are composed by the rule layer into complete interaction techniques.

Before Its came along there were mainly two types of layered architectures that provided the required flexibility in application development. User Interface Management Systems (UIMS) and toolkits. UIMS separate the business layer from the interface. Back-end computations are separated from the dialogue control and style. Style, however, was often treated in a single interface layer. Toolkits separated style from the application. Dialogue control remained in the back-end while the implementation of interaction techniques is hidden in a code library.

The four layers in Its present a series of advantages that separate this tool from its predecessors. Like in previous UIMS there is a separation from back-end computations and the interface itself. By separating the action layer from the dialogue allows actions to be reused in different applications.

Splitting the interface into separate layers for style-independent dialogue, rule base and toolkit also gives some benefits. First, the dialogue remains independent of style. A dialogue can be mapped into any different style simply by firing the appropriate rule. Second, interface designers control style rather than application programmers. The rule layer represents the selection criteria for all interaction techniques.

Each layer in Its architecture corresponds to one of four roles in application development: application programmer, application expert, style expert and style programmer. An application expert is familiar with the domain of the application. The application expert typically is neither trained in software development or part of an information systems department. In Its, the application expert is the author of the dialogue. A style expert may be a graphic artist or a human factors engineer. Rules give them direct control over style in Its.

Its is a specification-based system. The main difference between these tools from automated-design tools like Janus[2] is that the modelling language is open whereas in automated-design tools are closed. By lifting this limitation for the developers the final result can have a higher level of quality although it's dependent from the capabilities of the developer himself.

Even though Its is a specification-based system, this doesn't mean that developers have to specify every feature of every individual window. Developers are forced to specify the content of dialogues which is equivalent to the abstract user interface and this is, as been proved to be by experience, the most difficult model to generate by the automated-design tools. The style rules layer or the concrete user interface model doesn't have to be totally specified. This doesn't mean that ITS generates this model but that developers can reuse rule sets from libraries that contain the abstract to concrete mapping for significant portions of the interface specifications.

2.2 Optimization-based generation of interfaces

Recent work is beginning to reveal that numerical optimization can play a role on modern approaches for generating interfaces and displays. Gadget[6] and Supple[7] are two examples of this tendency.

The first is a framework that aims to provide developers with none or few knowledge on numerical optimization a set of tools that allows them to generate user interfaces through optimization methods.

The second is a tool that aims the generation of personalized user interfaces at run time. The main motivation behind Supple is that current user interfaces are developed with only a limited set of user abilities in mind leaving people with special needs with difficulties to interact with their applications.

2.2.1 Gadget

Although optimization-based techniques appear to offer several potential advantages most programmers are intimidated or uncomfortable by the math required to program an optimization. Although optimization toolkits are available, they typically require substantial specialized knowledge because they have mostly been designed for physics simulations and other traditional optimization problems.

Gadget provides a set of abstractions for many optimization concepts along with a set of mechanisms to help programmers quickly create optimizations, including an efficient lazy evaluation framework, a powerful and configurable optimization structure, and a library of reusable components. A programmer creating an optimization using the Gadget toolkit needs to supply three essential components: an initializer, iterations and evaluations. The initializer creates the initial solutions to be optimized. This might be based on an existing algorithm, or done randomly. Iterations are responsible to transform one potential solution into another, typically using models that are at least partially random. Finally, evaluations are used to judge the different notions of goodness in a solution.

There's a standard framework to abstract the concepts and constructs behind evaluations. Gadget allows programmers to focus on creating evaluations to measure criteria that are important to the problem. Gadget then combines these evaluations and uses them to choose between a set of possible solutions. This process is divided into five stages.

First the framework presents each evaluation with the current potential solution, which is called the prior solution. Each evaluation returns an array of double values representing its interpretation of the prior solution. This collection of arrays of double values is called the prior result.

On the second stage the framework uses an iteration object to modify the prior solution and create a new one, called the post solution.

Third, the framework presents the post solution to each evaluation. Each individual evaluation returns interpretations that are then combined to create a post result.

In the fourth step, the framework uses a method to compare the prior result with the post result. This is possible by requiring each evaluation to be capable of comparing two arrays of double values that it has created and providing a double value in the range of -1 to 1, where -1 indicates the evaluation has a strong preference for the prior solution, 0 indicates that the evaluation is indifferent and 1 indicates that the evaluation has a strong preference for the post solution.

Finally, the result of this comparison indicates whether the framework should go with the post solution or revert to the prior solution. To choose between them, the values retrieved from the fourth step are multiplied by the weight associated with its respective evaluation and then summed. If the sum is greater than 0, the framework prefers the post solution, else it reverts to the prior solution.

2.2.2 Supple

Most of today's software interfaces are designed with the assumption that they are going to be used by an able-bodied person, who is using a typical set of input and output devices, who has typical perceptual and cognitive abilities, and who is sitting in a stable, warm environment. Any deviation from this pattern requires a new design. In [7] is argued that automatic personalized generation of interfaces is a feasible and scalable solution for this challenge. Supple can automatically generate interfaces adapted to a persons devices, tasks, preferences and abilities at run-time. It is not intended to replace user interface designers, instead it offers an alternative user interface for those people whose devices, tasks, preferences and abilities are not sufficiently addressed by the original designs.

Support for users with special needs is often forgotten by interface designers. When this problem is addressed there are three popular patterns that are usually followed, manual redesign of the interface, limited customization support, or by supplying an external assistive technology. The first approach is clearly not scalable, new devices constantly enter the market, and people's abilities or preferences both differ greatly and often cannot be anticipated in advance. Second, today's customization approaches typically only support changes to the organization of tool bars and menus and cosmetic changes to other parts of the interface. Furthermore, even when given the opportunity, most people do not customize their applications. Finally, assistive technologies, while they often enable computer access for people who would otherwise not have it, also have limitations. They're impractical for users with temporal impairments, they do not adapt to people whose abilities change over time and they're often abandoned by people who need them because of factors like cost, complexity, configuration, and the need for ongoing maintenance.

In contrast with this approach, Supple generates personalized interfaces to suit the particular contexts of individual users. In order to be able to generate these personalized interfaces, Supple makes three important contributions:

- Defines an interface generation as a discrete constrained optimization problem and solve it with a branch-and-bound algorithm using constraint propagation. This general approach allows the Supple system to automatically generate "optimal" user interfaces given a declarative description of an interface, device characteristics, available widgets, and a user and device specific cost function.

- Two types of cost functions were developed to guide the optimization process. The first is factored in a manner that enables preference-based personalization as well as fast computation, allowing Supple to generate user interfaces in under 1 second in most cases. The second explicitly models a person's ability to control the pointer, allowing Supple to generate user interfaces adapted to unusual interaction techniques or abilities, such as an input jittery eye tracker or a user's limited range of motion due to a motor impairment.
- Supple also supports two approaches for dynamic personalization of generated interfaces: an automatic system driven adaptation to the current task, and a user driven customization.

Like other automatic user interface generation systems, Supple relies on an interface specification (I). Additionally, Supple also uses an explicit device model (D) to describe the capabilities and limitations of the platform for which the interface is to be generated. Finally, in order to reflect individual differences among usage patterns, Supple additionally includes a usage model, represented in terms of user traces (T).

Supple adopts a functional representation of user interfaces that is, one that says what functionality the interface should expose to the user instead of how to present those features. Like a number of previous systems, Supple represents basic functionality in terms of types of data that need to be exchanged between the application and the user. Semantic groupings of basic elements are expressed through container types which also serve as reusable abstractions. Supple chooses not to adopt a task-oriented approach for two reasons. First, because task-oriented descriptions are typically first compiled into a data-oriented functional description. Second, task-oriented languages are particularly useful for capturing task-oriented processes such as store checkout or making a hotel reservation. Most direct manipulation systems, however, support a broad range of possible tasks and make simultaneously available numerous reversible actions. Such interfaces would not benefit significantly from a task-oriented representation.

Most people use only a small subset of functions available in any application, and different users use different subsets. To adapt to a person's tasks and long-term usage patterns, the user interface should be rendered such that important functionalism is easy to manipulate and to navigate to. Instead of relying on explicit annotations by the designer or the user, Supple relies on usage traces, which can correspond either to actual or anticipated usage. Usage traces provide not just interaction frequency for primitive widgets, but also frequencies of transitions among different interface elements. In the context of the optimization framework, traces offer the possibility of computing *expected* cost with respect to anticipated use.

Supple is an optimization-based tool to generate user interfaces. Unlike GADGET it is not a simple framework to be used by programmers, it implements all the optimization related logic. Developers are only obliged to supply the specification. Unlike most of the recent tools in this field, Supple only requires the abstract user interface model and a set of constraints, it doesn't use tasks. The main advantage regarding other solutions is that constraint set is dynamic and depends both on the user and the device. This allows Supple to generate good user interfaces to every user even if the designer doesn't have much information about the users preferences.

2.3 User interface patterns

Patterns are widely used in every field of engineering. One of the earlier definitions of patterns can be found on [1]. Almost twenty years later patterns were brought to software engineering by [8].

Patterns bring many advantages, not only they make the development of a product less time consuming and thus less expensive but can also guarantee a higher level of quality because patterns are solutions that have been tested and used in other projects.

Particularly on user interfaces, these are very important features because building a good user interface is a very complex and time consuming process. On most software projects it takes about half of the time frame allocated to that project, so patterns can help to make this process more efficient. Also there's the problem of usability. This is one of the most important aspects of software projects but its still very difficult to build a user interface compliant with human computer interaction (HCI) rules. By using patterns this can be easily achieved if the patterns are already compliant with these rules.

Patterns are usually stored in catalogues. In [8] a pattern is composed by the following fields:

- The **Pattern name** resumes the pattern in one or two words that we use to refer to named pattern.
- The **Problem** describes in which situations the pattern should be applied.

- The **Solution** describes how the pattern works, what elements it has and how they relate to each other.
- The **Consequences** describe the side effects of using the pattern.

This is the specification used for software design patterns but it's also used in most user interface patterns catalogs. In [27] documentation of patterns is divided in two categories, descriptive patterns and generative patterns. Descriptive patterns are meant to be interpreted by humans so they describe the solution in a generic way so that the pattern can be used in a wide range of contexts while generative patterns maximize *expressivity* over *genericity* thus, they can be used in more restricted range of contexts but the solution is specific enough to be interpreted by machines.

Design patterns like the ones described in [8] are generative patterns because their solution is specified in UML which is a formal language that can be easily interpreted by machines to perform transformations.

A list of catalogues for user interfaces can be found in [5]. Most of these catalogues define their solutions with text and images because there isn't a reference language to specify user interfaces. Thus most of these patterns are descriptive patterns that can only be used by humans.

In order to take full advantage of patterns we need a way to document them. Generative patterns are the most useful in the context of this project but to use them we need to find a language to specify these patterns so that they can be interpreted by a machine to generate a concrete user interface. applications. In section 2.3.1 is described IdealXML a tool for developing user interface models that takes advantages of patterns.

2.3.1 IdealXML

IdealXML[14, 13] is an experience-based environment for user interface design. Experience is the accumulation of knowledge or skills that result from direct participation in events or activities. Developers have a strong tendency to towards reusing designs that worked well for them in the past. Unfortunately, this design reuse is usually limited by personal experience, and there is usually few sharing of knowledge among developers.

IdealXML manipulates a pattern repository, where patterns are organized following a hierarchical structure. At the top, this structure has different models related with a Model Based - User Interface Development Environment (MB-UIDE): domain, task, presentation and mapping, context and user models are left for future work. IdealXML is shipped with a predefined collection of patterns from a variety of sources. These patterns are the initial base of knowledge.

IdealXML is an MB-UIDE and designers can, using several graphical notations, specify domain models, task models, abstract presentation models and mapping models between them. Some of these models are stored in the pattern repository and new ones can always be added.

IdealXML also allows for the animation of a task model to generate a hi-fi prototype of the future user interface while still in the first development stages. This is achieved by using CTT [18], UsiXML [23] and a set of heuristics to transform the task model specification into an abstract UI.

Prototyping consists in the creation of a preliminary version of the future UI (prototype) so that the user and the experts can find possible problems in the design of the UI, both from the functional and from the usability points of view. Prototyping techniques fall into two main categories:

- **Lo-fi:** this family of techniques is mostly used in requirements analysis stage to validate the requirements with the user in user-centred approaches.
- **Hi-fi:** they are aimed at the creation of preliminarily versions of the UI with an acceptable degree of quality. This kind of techniques produces a UI prototype which is closer to final future one.

Abstract prototyping was devised because it was found that the sooner developers started drawing realistic pictures or positioning real widgets, the longer it took them to converge on a good design.

As it was mentioned above, IdealXML uses a set of heuristics to transform the task model into an abstract interface model:

- Each cluster of interrelated task cases becomes an interaction space in the navigation map, so an abstract task is a container.
- A container also can be an interaction task or an application task in a hierarchical task decomposition.
- A component rises when an interaction or application task is found in a hierarchical task decomposition.

- A component can have several facets (input, output, control and navigation). These facets allow the user to interact with the system.

The animation of the abstract user interface that resulted from the designed task model is grounded in the identification of the enabled task set (ETS). Having identified the ETC for a task model, the next step is to identify the effects of performing each task in each ETS. The result of this analysis is a state and transitions occur when tasks are performed. In IdealXML's proposal, the task model specification is split into states. Each state is a set of interrelated tasks, including temporal relationships between those tasks, usually connected to an essential use case.

2.4 Specification languages

In [13] was stated that one of the most important challenges to overcome in model driven development of user interfaces is the creation of a specification language that would be massively adopted and became a common ground between developers like UML is for software architecture.

Over the years many languages were developed to try and overcome this obstacle. In [19] were enumerated some of the problems found with that time user interface models:

- **Partial models**, most models deal only with a portion of the spectrum of interface characteristics. Some emphasize domain, others emphasize tasks, some others emphasize presentation guidelines and so on.
- **Insufficient underlying model**, several model-based systems use modelling paradigms proven successful in other applications areas, but that come up short for interface development. These underlying models typically result in partial interface models of restricted expressiveness.
- **System-dependent models**, many interface models are non-declarative and are embedded implicitly into their associated model-based systems, sometimes at code level. These generic models are tied to the interface generation schema of their system, and are therefore unusable in any other environment.
- **Inflexible models**, experience with model-based systems suggests that interface developers often wish to change, modify, or expand the interface model associated with a particular model-based environment. However, model-based systems do not

offer facilities to such modifications, nor the interface models in question are defined in a way that modifications can be easily accomplished. Thus the inclusion of an open meta-model like in UML could be an important factor of success.

Next, two recent specification languages for user interfaces that try to overcome these problems will be presented. In section 2.4.1 will be presented UMLi, an extension to UML to support the modelling of user interfaces. Section 2.4.1, presents WISDOM [16], an engineering method for small software companies with an UML based notation with support for UI modelling. Finally, section 2.4.3 will be presented UsiXML, a language with potential to become a standard in user interface specification.

2.4.1 UMLi

Although user interfaces represent an essential part of software systems, UML seems to have been developed with little attention to specific details of user interface models. It's possible to use UML to model important aspects of user interfaces but these models usually get widely unnatural.

UMLi[4] doesn't try to replace UML entirely. The UMLi meta-model fully integrates with UML this makes possible to integrate UMLi models with other UML models.

It is possible to model abstract and concrete interfaces using class models in UML. However class models don't provide an intuitive representation of the interface. UMLi provides an alternative diagram notation for describing abstract interaction objects.

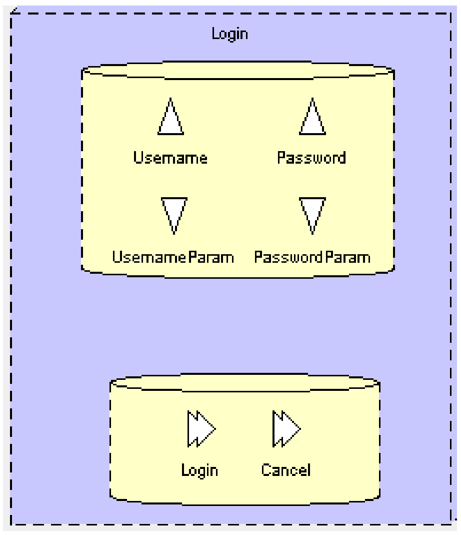


Figure 2.1: Login window modelled in UMLi

Figure 2.1 shows an abstract user interface for a login windows modelled in UMLi. The upper container has four entities, *username* and *password* represented as upward triangles, are indication for what is expected as user input. These are similar to labels. *UsernameParam* and *PasswordParam* are the actual input controllers where the users should write their username and password. These four entities could be replaced by two entities of the type *editors*, rendered as rhombis. On the lower container are represented two possible actions for the users. These are represented as right-pointing arrows.

UMLi's user interface diagram consists of six constructors:

- **FreeContainers** rendered as dashed cubes. A *FreeContainer* is a top-level interaction class that no other interaction class can contain.
- **Containers** rendered as dashed cylinders. A Container is a mechanism that groups interaction classes other than *FreeContainers*.
- **Inputters** rendered as downward triangles. An *Inputter* receives information from users.
- **Editors** rendered as rhombi. An Editor facilitates the two-way exchange of information.
- **Displayers** rendered as upward triangles. A Displayer sends information to users.
- **ActionInvokers** rendered as right-pointing arrows. An *ActionInvoker* receives direct instructions from users.

Tasks are usually represented in a tree notation in which leaf nodes are primitive tasks and non-leaf nodes group and describe relationships between their children nodes. There is a set of three essential features present in most task modelling languages:

- **Hierarchical decomposition**, high-level tasks systematically decompose into less abstract tasks.
- **Temporal relationships**, the order in which a composite task's children are carried out depends on the parent's temporal relation.
- **Primitive tasks**, the lowest-level nodes described in the task model are primitive tasks. An action task, for example, corresponds to an activity the application carries out. An interaction task involves some degree of human-computer interaction.

Use cases and activities in UMLi represent the notion of task with a set of features that include all the elementary ones mentioned above.

Using use cases and their scenarios, it's possible to elicit user interface functionalities required to let users achieve their goals. Possible ways to perform actions that support the functionalities elicited using use cases can be identified using activities. Therefore, mapping use cases into top level activities can help describe a set of interface functionalities similar to that described by task models in other specification languages.

Use-case diagrams in UMLi are UML use-case diagrams. Activity diagrams in UMLi, however, extend activity diagrams in UML. UMLi provides a notation for a set of macros for activity diagrams that can be used to model behaviour categories usually observed in user interfaces: optional, order independent, and repeatable behaviours.

Using these macro notations, activity diagrams in UMLi can cope better with the tendency that activity diagrams have to become complex even when modelling the behaviour of simple user interfaces.

In order to represent relationships between models in UMLi object flows are used in activity diagrams to describe how to use class instances to perform actions in action states. By using object flows, it's possible to incorporate the notion of state into activity diagrams that are primarily used for modelling behaviour. In UMLi, it's also possible to use object flows to describe how to use interaction class instances. However, object flow states—rendered as dashed arrows connecting objects to action states—have specific semantics when associating interaction objects to activities and action states. UMLi specifies categories of object flow states specific to interaction objects:

- The **interacts** object flows relate primitive interaction objects to action states, which are primitive activities. They indicate that associated action states are responsible for interactions in which users invoke object operations or visualize the results of object operations.
- The **presents** object flows relate *FreeContainers* to activities and specify that the associated *FreeContainers* should be visible while the activities are active.
- The **confirms** object flows relate *ActionInvokers* to selection states and specify that selection states have finished normally.
- The **cancels** object flows relate *ActionInvokers* to composite activities or selection states and specify that activities or selection states have not finished normally and that the application flow of control should be rerouted to a previous state.
- The **activates** object flows relate *ActionInvokers* to other activities, thereby trig-

gering the associated activities that start when an event occurs.

In [4] is mentioned a case study specified both in UML and UMLi. A set of metrics were applied to each specification. The results of these metrics show that constructing and maintaining interactive system models should be simpler and easier in UMLi than in UML.

2.4.2 WISDOM

WISDOM [16] is a lightweight software engineering method. Lightweight in the sense that it can be learnt and applied in a couple of days or weeks. WISDOM is object-oriented and it uses the UML to specify, visualise and document the artefacts of the development project. WISDOM is specifically adapted to develop interactive systems, because it uses and extends the UML to support Human Computer Interaction (HCI) techniques. Finally WISDOM is evolutionary, in the sense that the project evolves in a sequence of incremental prototypes, ultimately leading to the end product.

WISDOM's specification is divided in three key issues:

- Process;
- Architecture;
- Notation.

The Wisdom process defines the steps for successfully implementing the Wisdom method in a software development organisation. The WISDOM process is divided in four development phases called evolutionary phases:

- **Inception**, focus on the interiorisation the project, the team should understand the system's context and user's profile and start writing requirements.
- **Elaboration**, focus on requirements, system analysis and UI's architecture design
- **Construction**, in this phase the focus is on designing both the system and it's UI and than implement that design.
- **Transition**, the last phase is focused on evaluate the implementation and gather feedback.

Throughout all phases, there is a constant build, test loop. Every phase includes the construction of a prototype and evaluation of that prototype. Thus the concept of *Evolutionary Prototyping.*

The WISDOM architecture specifies the different models of interest to effectively develop an interactive system under the framework of the WISDOM process. The Wisdom model architecture combines both the internal and user interface architectures. Software engineering methods traditionally focus on the organisation of the internal components and their relationships - the internal architecture of the software system. Since early project inception WISDOM also focus on the external or UI architecture. Like the internal architecture is more than a set of internal components, the UI architecture is also more than a set of UI components. In Wisdom the external architecture not only concerns individual UI components but also (and essentially) the overall structure and organisation of the UI.

The WISDOM notation is a subset and extension profile of UML, the standard object-oriented language for visualising, specifying, constructing and documenting the artifacts of a software-intensive system. WISDOM uses a subset of the UML and extend the language using it's built-in extension mechanism in order to provide some modelling constructs useful to design interactive systems. For a detailed description of the Wisdom notation refer to [16].

2.4.3 UsiXML

UsiXML (USer Interface eXtensible Markup Language) is a User Interface Description Language (UIDL) that uses Model-Driven Engineering (MDE) for specifying a User Interface (UI) at an implementation independent level. The UI specifications are usually specified in different models. Each UI level is described by a model(s). UsiXML is based on the Cameleon reference framework[3]. This framework describes a UI in four main levels of abstraction: task and domain level, abstract UI level, concrete UI and final UI. On the basis of these 4 levels, UsiXML proposes a set of models:

- **Transformation model**: contains a set of rules in order to enable a transformation of one specification to another.
- **Domain model**: describes the classes of the objects manipulated by the users while interacting with the system.
- **Task model**: describes the interactive task as viewed by the user interacting with

the system. The task model is expressed according to the CTT specification [18].

- **Abstract user interface model**: represents the view and behaviour of the domain concepts and functions in platform independent way.
- **Concrete user interface model**: represents a concretization of the abstract user interface model.
- **Mapping model**: contains a series of related mappings between models or elements of models.
- **Context model**: describes the three aspects of a context of use, which is a user carrying out an interactive task using a specific computing platform in a given surrounding environment.
- **Resource model**: contains definitions of resources attached to abstract or concrete interaction objects.

The user interface model in UsiXML consists of a list of component models (described above) in any order and any number. It doesn't need to include one of each model component and there can be more than one of a particular kind of model component. It's also composed by a creation date, a list of modification dates, a list of authors and a schema version.

UsiXML allows designers to apply a multi-path development of user interfaces. In this development paradigm, a user interface can be specified and produced at and from different, and possibly multiple, levels of abstraction while maintaining the mappings between these levels if required. Thus, the development process can be initiated from any level of abstraction and proceed towards obtaining one or many final user interfaces for various contexts of use at other levels of abstraction. In this way, the model-to-model transformation, which is the cornerstone of Model-Driven Architecture (MDA), can be supported in multiple configurations, based on composition of three basic transformation types[11]:

- **abstraction**, is the process of substitution of the input artefacts into more abstract ones;
- **reification**, is the process of substitution of the input artefacts into more concrete ones;
- **translation**, is the process of substitution of the input artefacts aimed at a particular context of use into others that are aimed for a different context.

Multi-path UI development is based on the Cameleon Reference Framework[3], which defines UI development steps for multi-context interactive applications. The development process with this framework is structured in four steps where each developments step is able to manipulate a set of artefacts in the form of models:

- **Final UI (FUI)**: is the operational UI. Any UI running on a particular computing platform either by interpretation or by execution.
- **Concrete UI (CUI)**: it's a transformation of the abstract UI for a given context of use into Concrete Interaction Objects (CIOs). It defines widgets layout and interface navigation. The CUI abstracts a FUI into a UI definition that is independent of any computing platform. Although a CUI makes explicit the final appearance and style of a FUI, it is still a mockup that runs only within a particular environment. A CUI can also be considered as a reification of an AUI at the upper level and an abstraction of the FUI with respect to the platform.
- **Abstract UI (AUI)**: defines interaction spaces by grouping subtasks according to various criteria, a navigation scheme between the interaction spaces and selects Abstract Interaction Objects (AIOs) for each concept so that they are independent of any modality. An AUI abstracts a CUI into a UI definition that is independent of any modality of interaction. An AUI can also be considered as a canonical expression of the rendering of the domain concepts and tasks in a way that is independent from any modality of interaction. An AUI is considered as an abstraction of a CUI with respect to modality.
- **Task and Domain (T&D)**: describe the various tasks to be carried out and the domain-oriented concepts as they are required by these tasks to be performed. These objects are considered as instances of classes representing the concepts manipulated.

UsiXML is also very extensible. At the model level USIXML allows to define any kind of model. In this sense it is possible to instantiate any new model of the above mentioned classes. At meta-model level USIXML offers a modular structure which clearly segregates the models it describes. This facilitates the integration of new classes of models into UsiXML. The model and its concept is simply declared along with its relationships with other models. Rules exploiting this new model can be defined afterwards.

In conclusion, UsiXML solves every problem stated in [19]. UsiXML was designed for user interfaces, it's not an adaptation of some modelling language that was meant for other use. It provides several classes of models with different abstraction levels so that every part of the interface is specified. By allowing the application of multi-path development process, it makes sure that every specification is platform independent. Finally, it's a flexible language, UsiXML provides mechanisms that can be used to extend and modify it's models and transformation rules.

2.5 Discussion

The first tools in this chapter were Janus[2] and Its[28]. These tools represent the beginning of MDDUI and at the same time two different approaches to this subject. Developers using Janus will decrease the development time of their applications. This seems to be the main goal of the project. Because it only needs one input, the domain model, it doesn't take a lot of effort to create a UI. Of course this also means that the generated UI won't have the same quality as one built by hand (depending on the abilities of the developer) and this is the most criticized aspect of Janus. On the other hand, Its can produce high quality UI's. But the amount of work necessary to build something is huge compared to Janus. Its also provides a way to divide work through the development team which is a very interesting feature because it facilitates the implementation of different development methodologies alongside this tool.

By analysing these two projects one can infer that MDD can be used with two objectives in mind:

1. to build low-cost prototypes of an application rapidly;
2. to enhance the quality of the final product.

Janus is clearly a tool that can be used to rapidly create a prototype of an application that can be used to communicate with customers or other stake holders. Its is aimed at building better applications with large teams.

In section 2.2 were introduced a couple of projects that use numeric optimization to generate UI's. The first project was Gadget[6]. This is a framework that abstracts the programmer from the optimization itself. The problem with this project is that it's not a complete solution. Developers still have to produce evaluations which can be a tricky subject because it's not trivial to determine what properties make a user interface "good". The second project, Supple[7], it's more mature and complete. The objective behind Supple is to generate UI's at run time that are optimal for people with special needs.

Although Gadget is meant to simplify numeric optimization of UI's in a way that it could be used by any developer it still looks very complex to use this system.

Supple is a more complete system. It receives three types of input models:

- one related to the domain;
- other related to the user;
- and other related to the platform it's supposed to run on.

This makes Supple easier for developers but doesn't give them much control of the final UI. The great advantage of this system its his capability to dynamically adapt to the needs of every user.

Section 2.3 covered the use of UI patterns in the process of modelling a UI. In this section IdealXML[14, 13] was looked in detail. IdealXML is a tool where designers can, using several graphical notations, specify Domain models, Task models, AUI models and mapping models between them. IdealXML manipulates a pattern repository, where patterns are organized following a hierarchical structure. This enables the usage of patterns at design time.

Unfortunately, like in most projects in this field, IdealXML is an academic project and it's not ready to be adopted by the industry.

Section 2.4 covered "Specification Languages". This is the most important topic approached on this chapter because the lack of a standard language to describe UI's is one of the most setbacks on this field. In this section we've looked into three projects that are trying to solve this issue.

First we analysed UMLi [4]. UMLi is an extension to UML with support for interaction models. The second project was WISDOM [16]. Like UMLi, WISDOM provides an extension to UML in order to support interactive models.

The third project in this section was, UsiXML [11]. UsiXML is a UIDL based on the Cameleon reference framework[3]. UsiXML supports every step in the Cameleon framework. UsiXML is a little different from the first two ones because it doesn't try to extend UML. UsiXML introduces it's own set of models but reuses familiar notations, like UML for domain models and CTT for task models.

Both WISDOM and UsiXML, ship with an engineering method, with different approaches. WISDOM's method focus on the development of the entire product while UsiXML is focused on the UI. Both methods have their advantages and disadvantages. But, if we're looking for a standard tool that can be used by most organizations, UsiXML's approach is more generic and therefore has better probability to find a place in the industry.

UMLi doesn't propose an engineering method. This project seems to be the less featured one. It reuses some of UML models to model interactive features, and only introduces a new notation for AUI models. This can play in favour of UMLi because it's the easiest one to understand and it's the project that would take the least effort to integrate in an organization's workflow.

In conclusion, WISDOM and UsiXML are the most full-featured butUsiXML is more generic then WISDOM. UMLi is less featured but at the same time easier to integrate in an organization.

Trying to guess which of these projects has the most probability of becoming a standard is a difficult task. Although there is one more parameter that can call the tie off which is community. UsiXML is building a large and strong community[1] around it. Taking all things into account, UsiXML shows the best probability of becoming a standard in MDDUI.

[1]http://www.usixml.eu/all-end-user-club

Chapter 3

UsiXML

In section 2.4.3 USer Interface eXtensible Mark-up Language (UsiXML) was referred as part of the state of the art. In this chapter this subject will be studied in more depth. The focus of this chapter will not be just UsiXML as a User Interface Description Language (UIDL) but the whole project behind it.

UsiXML's specification, version 1.8, was released in February 14th, 2007. In April of 2009 was founded a consortium with the objective of lowering the total application costs and development time by adding versatile context driven capabilities to UsiXML that would bring it far beyond the state of the art, up to the achievement of its standardisation.

The UsiXML consortium started with three main goals that would guide the entire project. The first goal is *the UsiXML "μ7" concept elicitation and promotion.* The μ7 concept means that it should be possible to specify a User Interface (UI) for an interactive application that should support multi-device, multi-platform, multi-user, multi-linguality/culturality, multi-organisation, multicontext, or multi-modality capabilities.

The second goal is the *development of theUsiXML language and the Model Driven Development of User Interfaces (MDDUI) method.* The UsiXML language should guarantee interoperability, reusability, and maintainability of interactive applications developed. For this reason UsiXML is an open eXtensible Mark-up Language (XML)-compliant standard UIDL. There should be models to cover every μ7 aspects. Finally, UsiXML should define a flexible methodological framework that accommodates various development paths as found in organisations and that can be tailored to their specific needs.

The third goal is to *set up development tools and demonstration of the validity on applications.* There should be a suite of software tools that support the methodological framework defined in the second goal that can be later integrated or connected to available software environments. A UsiXML service will be defined and developed once so that this service can be deployed many times, especially for all environments requiring them to reduce the total cost of development. UsiXML will provide developers with various knowledge bases containing usability and accessibility guidelines that can be semi-automatically verified on any UsiXML-produced UI so as to guarantee a certain level of quality.

Section 3.1 provides a description of the UsiXML semantics. In section 3.2 refers to the UsiXML engineering method for developing UI's.

3.1 UsiXML semantics

The core component of a user interface specified in UsiXML consists of a uiModel, which is itself decomposed into several models. Not all models should be included. Only those models which are required for the particular UI are included in a UsiXML file. Figure 3.1 shows the components of UsiXML in an Unified Modelling Language (UML) class diagram.

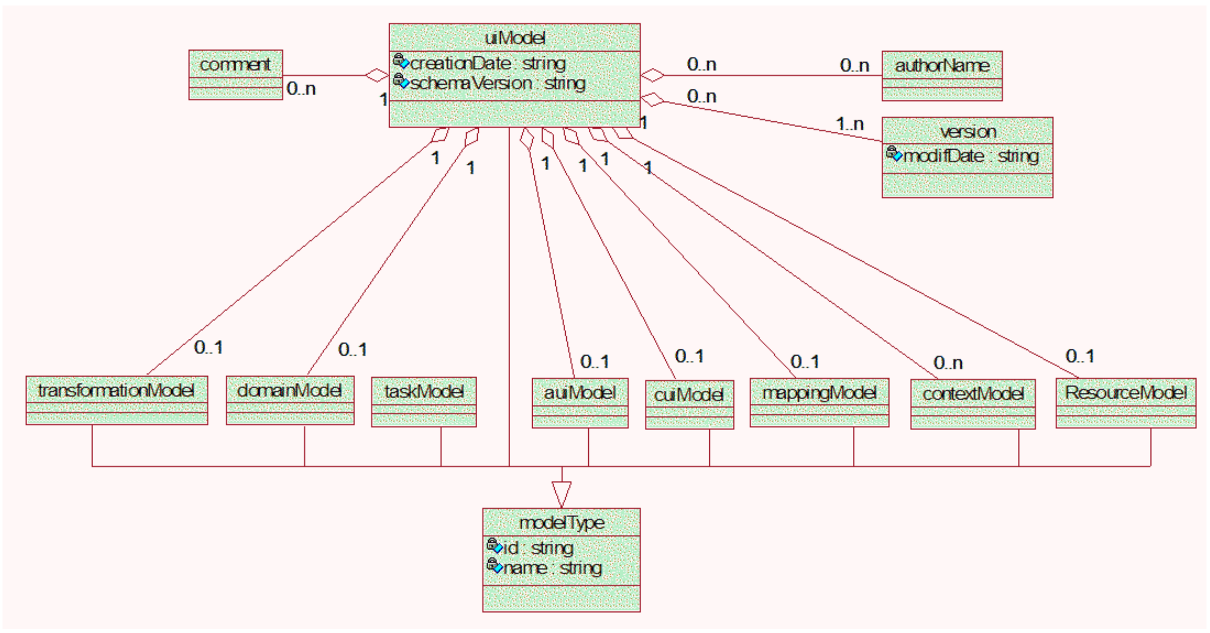


Figure 3.1: UsiXML specification as an UML class diagram.

The uiModel is the topmost superclass containing common features shared by all component models of a user interface. A uiModel may consist of a list of component models in any order and any number:

- **Transformation model**: contains a set of rules in order to enable a transformation of one specification to another.
- **Domain model**: describes the classes of the objects manipulated by the users while interacting with the system.
- **Task model**: describes the interactive task as viewed by the user interacting with the system. The task model is expressed according to the CTT specification [18].
- **Abstract user interface model**: represents the view and behavior of the domain concepts and functions in platform independent way.
- **Concrete user interface model**: represents a concretization of the abstract user interface model.
- **Mapping model**: contains a series of related mappings between models or elements of models.
- **Context model**: describes the three aspects of a context of use, which is a user carrying out an interactive task using a specific computing platform in a given surrounding environment.
- **Resource model**: contains definitions of resources attached to abstract or concrete interaction objects.

The uiModel is also composed by a creation date, a schema version, a set of authors, a set of versions and a set of comments. A complete reference to UsiXML semantics can be found in [23]. This section refer to version 1.8. There is a newer version in development that it's not yet available that might redefine some of the concepts explained here.

3.2 UsiXML method

The Model Driven Engineering (MDE) approach allows developing a UI by transforming progressively UsiXML models in order to obtain specifications that are detailed and precise enough to be rendered or transformed into code.

The usage of the Cameleon reference framework[3] allowed UsiXML development to be divided in a set of phases that can be achieved through a set of transformations (see section 2.4.3). These features supplied a base structure for the development of an engineering method called *Forward engineering method* referenced in section 3.2.2. This method is an instantiation of a meta-model called SPEM for UsiXML (SPEM4UsiXML) that we'll see in section 3.2.1 and can be used to develop other engineering methods. It's based on Software & Systems Process Engineering Metamodel (SPEM) 2.0 a method meta-model specification from the Object Management Group (OMG). In appendix A is an overview of this specification.

3.2.1 Spem4UsiXML

SPEM4UsiXML is dedicated to UsiXML method modelling. The goal of this meta-model is to propose a set of key elements to define any UsiXML based engineering method. SPEM4UsiXML is an extension of SPEM 2.0 that adds some new classes. SPEM4UsiXML separates the operational aspect of a method from its temporal aspect. This means that SPEM4UsiXML reuses theUML diagrams for the presentation of various UsiXML method concepts. As shown in figure 3.2, the SPEM4UsiXML meta-model uses seven main meta-model packages inherited from SPEM: *Method Content*, *Process Structure*, *Process Behaviour*, *Process With Methods*, *Core Method Plug-in* and *Managed Content.* SPEM4UsiXML extends some of the classes from *Method Content* and *Process Structure* as we'll explain in this section.

A complete reference for the SPEM4UsiXML meta-model can be found in [24]

The *Method Content* and *Process Structure* packages are the only ones that were extended from the original SPEM specification. Hereafter these extensions will be described.

Method Content

The *method content* meta-model package defines the core elements of every method (producer, work unit and work product) independently of any process or project.

As shown in figure 3.4, SPEM4UsiXML adds new classes to the original SPEM *method content* meta-model package in order to specify the several development steps and substeps and also the different kinds of product and producer. The important classes of the SPEM4UsiXML *method content* meta-model are:

- The **Development Step Definition** defines the transformation being performed by *Role Definition instances.* A *Development Step* is associated to an input and an output, *Work Products.* A *Development Step Definition* can be:

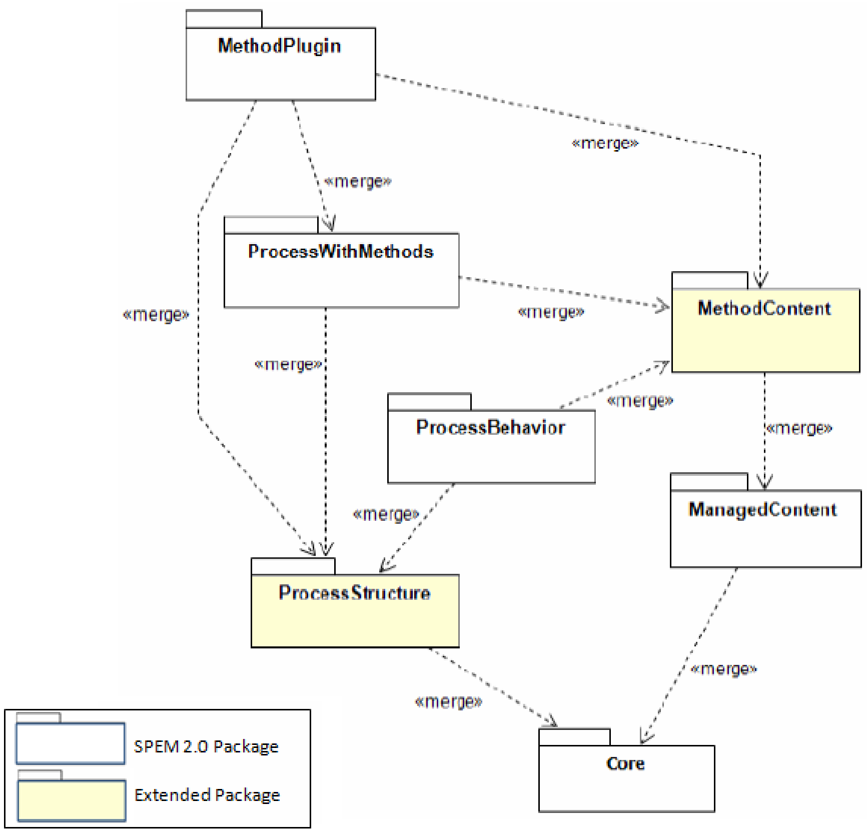


Figure 3.2: Structure of the Spem4UsiXML meta-model.

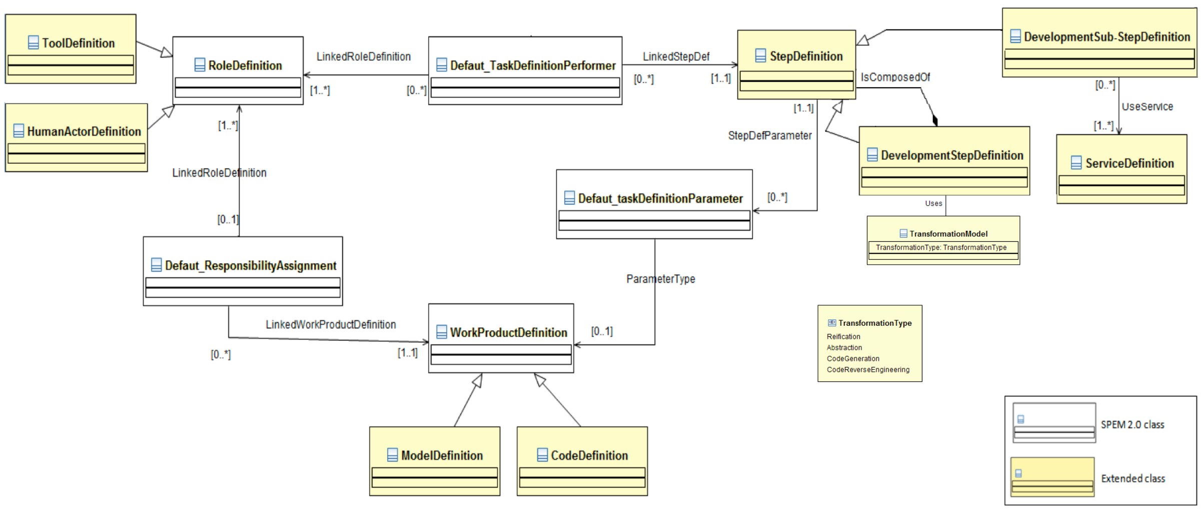


Figure 3.3: Structure of the Spem4UsiXML method content meta-model package.

  - A **Reification Definition** defines the transformation of a *Work Product Definition* of higher-level into a *Work Product Definition* of lower-level.
  - An **Abstraction Definition** defines the transformation of a *Work Product Definition* of lower-level into a *Work Product Definition* of higher-level.
  - A **Translation Definition** defines the transformation a *Work Product Definition* based on context.
  - The **Code generation Definition** defines the transformation of a *Model Definition* into a *Code Definition.*
  - The **Code reverse engineering Definition** defines the transformation of a *Code Definition* into a *Model Definition.*

- A **Development Sub-Step Definition** defines the sub-steps of a *Development Step.* A sub-step can be achieved using a service (*Service Definition*). Each service can be based on a set of transformation rules, a program, the context or a template in order to enact the *Development Sub-Step.*
- A **Step Definition** is an abstract generalization class that defines a set of properties that are inherited by *Development Step*, and *Development Sub-Step.*
- The **Work Product Definition** describes the product which is used, modified, and produced by *Development Steps.* A *Work Product Definition* can be: a UsiXML model (*Model Definition*) or UI code (*Code Definition*).
- A **Role Definition** defines a set of related skills, competencies, and responsibilities of an individual or a set of individuals. Roles are used by a *Development Step* or by a *Development Sub-Step* to define who performs them as well as to define a set of *Work Product Definitions* they are responsible for. A *Role Definition* can be:
  - A **Tool Definition** describes any automation unit that performs the *Development Step* or *Development Sub-Step.*
  - The **Human Actor Definition** describes any person, or organization that performs the *Development Step* or *Development Sub-Step.*

Process Structure

The *process structure* meta-model defines the structure of the method process. This package represents a process decomposed as a set of *Activity* classes that are linked to

Role classes and *Work Product* classes. This structure is useful to express the fact that a life-cycle is composed by set of phases, and each phase is composed by set of activities.

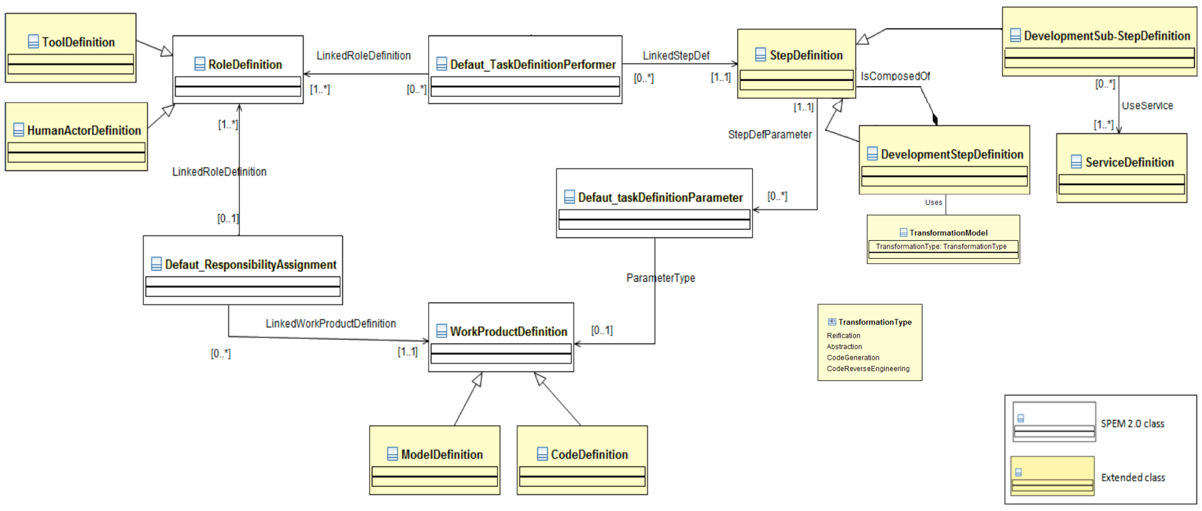


Figure 3.4: Structure of the Spem4UsiXML process structure meta-model package.

As shown in figure 3.4, *SPEM4UsiXML* adds some new classes to the originalSPEM process structure package in order to specify the control flow of the development steps and sub-steps and also the different products and producers used in the method process. The important classes of the *SPEM4UsiXML* process structure package are:

- The **Development Path** defines the properties of a UsiXML method.
- A **Breakdown Element** is an abstract generalization class that defines a set of properties available to the elements of a UsiXML method (*Product*, *Development Step* and *Producer*).
- A **Work Breakdown Element** provides specific properties for *Breakdown Elements* that represent *Development Step* and *Development Sub-Step.*
- A **Step Use** is an abstract generalization class that defines a set of properties available to *Development Step*, and *Development Sub-Step.*
- The **Development Step Use** defines the transformation steps of the method that are being performed by *Role Use* instances. A *Development Step Use* is associated to an input and an output *Work Product Use.* A *Development Step Use* can be: a reification , an abstraction, a translation, a code generation or a code reverse

engineering.

- A **Development Sub-Step Use** defines the sub-steps of a *Development Step Use.*
- A **Role Use** represents a performer of a *Development Step Use* or a *Development Sub-Step Use.*
- The **Work Product Use** represents an input and/or output type for a *Development Step.* It can concern a model (*Model Use*) or code (*Code Use*).
- The **Control Flow** represents a relationship between two *Work Breakdown Elements* in which one *Work Breakdown Element* depends on the start or end of another *Work Breakdown Element* in order to begin or end.

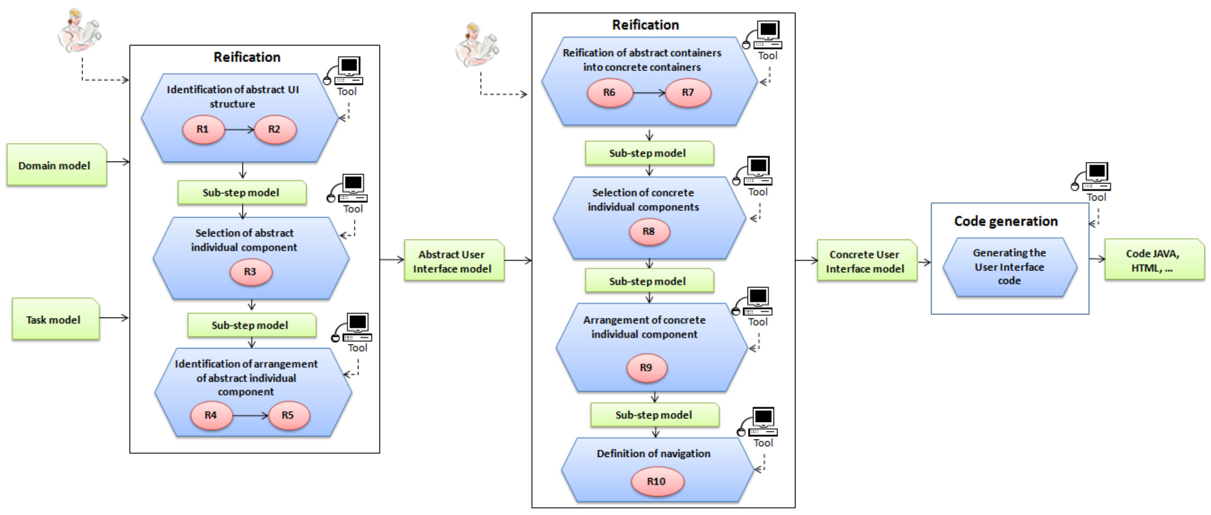
3.2.2 Forward engineering method

The starting point of the Forward Engineering method (FEM) is a task and a domain model (products). These models are then transformed (work) into an Abstract User Interface (AUI) model (product) which is then transformed (work) into a *Concrete user interface model* (product). Finally the code (product) is generated (work).

In order to achieve these transformations, a sequence of development step, reifications and code generation, are performed. Each development step may involve a set of development sub-steps. For example, the first development step involves the development sub-step, *Identification of Abstract user interface structure.* This sub-step consists in the definition of groups of an abstract interaction (an element of the abstract user interface). Each group corresponds to a group of tasks (in task model) tightly coupled together. To achieve its work, the sub-step can use a sequence of rules. For example, the sub-step: *Identification of Abstract UI structure* uses the sequence of two rules:

1. For each leaf task of a task tree, create an Abstract Individual Element;
2. Create an Abstract Container structure similar to the task decomposition structure.

Every development step takes as input a UsiXML model and transforms it into another UsiXML model by involving a set of development sub-steps, which in turn, manipulate sub-steps models by using a set of rules. Note that, a development sub-step can use templates of transformations instead of rules. For example, the step *Generating the user interface code* can use a template based approach in order to generate the UI code. Another note is that, each development step and development sub-step has a producer responsible of its execution. For example, the first development step can have a human actor who verifies the transformation done in this step. In turn, the sub-step: *Identification of Abstract UI structure* can have a transformation tool that can execute the rules sequence of this sub-step.



Reification
Identification of abstract UI structure
R1
R2
Tool
Domain model
Sub-step model
Selection of abstract individual component
R3
Task model
Identification of arrangement of abstract individual component
R4
R5
Abstract User Interface model
Reification
Reification of abstract containers into concrete containers
R6
R7
Selection of concrete individual components
R8
Arrangement of concrete individual component
R9
Definition of navigation
R10
Concrete User Interface model
Code generation
Generating the User Interface code
Code JAVA, HTML, ...

Figure 3.5: The forward engineering method.

More information in deeper detail about the FEM can be found in[10].

Chapter 4

Patterns in UsiXML

In chapter 2 was introduced some background on Model Driven Development of User Interfaces (MDDUI). In that chapter we described several projects related to this field and, most importantly, we identified some issues related to this subject that should be solved in order for the industry to advance:

- the lack of a common language to describe user interface models;
- the lack of tools to enable the reuse of these models at a conceptual level.

The first issue can be solved by USer Interface eXtensible Mark-up Language (UsiXML) (chapter 3). UsiXML is a User Interface Description Language (UIDL) with support for multiple development phases. With all the advantages granted by UsiXML and the strong community it's building, UsiXML has the potential to become an industry standard for describing User Interface (UI) models.

The lack of tools to enable the reuse of these models at a conceptual level is still a problem. There are several libraries of user interface patterns at a concrete level but there's very low support for conceptual user interface patterns.

In the course of this chapter we'll describe a solution that takes advantage of UI patterns at a conceptual level during the first stage of the Forward Engineering method (FEM). More specifically, we're aiming at the use of generative patterns[27]. This kind of patterns are more specific and more bound to be interpreted by machines. The goal is to use patterns to automate the first stage of the FEM. Hence, diminish the development time and increase the quality of the final product.

4.1 Concept

In section 3.2.2 we described the FEM. This methodology was introduced as a path to develop UI's using UsiXML and passing through multiple stages and levels of abstraction. In this section we'll be focusing on the first stage of the FEM.

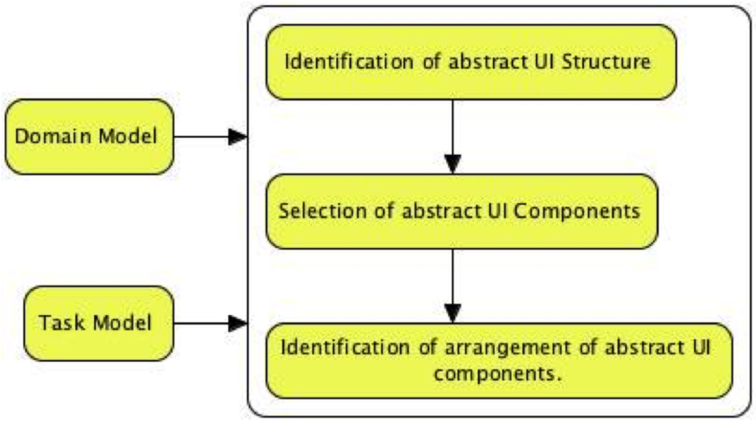

Identification of abstract UI Structure
Domain Model
Selection of abstract UI Components
Task Model
Identification of arrangement of abstract UI components.

Figure 4.1: The first stage of the forward engineering method.

The first phase of the FEM receives two inputs, a domain model and a task model. The goal of this phase is to work on these models in a way that the output is an Abstract User Interface (AUI) model.

The first phase of the FEM is divided in three steps:

- Identification of abstract UI structure;
- Selection of abstract UI components;
- Identification of arrangement of abstract UI components.

The identification of abstract UI structure consists on the definition of groups of abstract interaction. Each group corresponds to a group of tasks tightly coupled together.

The selection of abstract UI components consists on finding the best abstract individual component type to support one or several tasks.

The identification of arrangement of abstract UI components consists on the spatio temporal arrangement of elements in the AUI. This structure should respect the temporal relations defined in the task model.

The goal of this concept is to provide a solution to automate this process. Because this is a very complex transformation, developing a algorithm to accomplish this work it's not feasible. Maximize usability should always be the focus of any technique that attempts to build UI's and usability usually comes from experience and user's feedback. With this in mind we concluded that the best way to automate this transformation is by taking advantage of previous works and tested solutions. In other words, giving the nature of the problem, the best way to solve it, is using patterns.

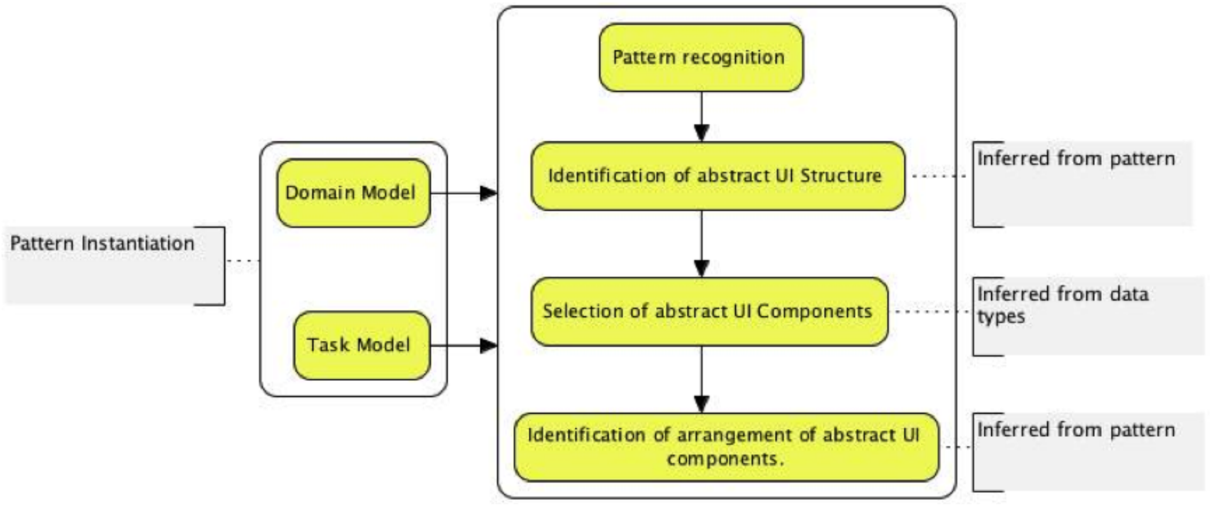

Pattern recognition
Identification of abstract UI Structure
Selection of abstract UI Components
Identification of arrangement of abstract UI components.
Domain Model
Task Model
Pattern Instantiation
Inferred from pattern
Inferred from data types
Inferred from pattern

Figure 4.2: Proposal for the first stage of the forward engineering method.

Figure 4.2 shows the first stage of the FEM with some changes. The input models should form a pattern instantiation. This means that for this method to work, the input can't be random, it should follow a structure that will provide additional information to the system. Another step was introduced in the first stage of the method. The first step is now the pattern recognition. This will allow the system to gather additional information related to the input models. By combining the information gathered from the pattern with it's instantiation the system should now be capable of following the other steps on its own without any need for human intervention.

4.2 Pattern definition

In section 4.1 we defined a concept to automate the first stage of the FEM using patterns. In this section we'll define a pattern to be used with the concept.

Most pattern catalogues describe patterns with the following set of key attributes:

- **Name**, is a way to reference the pattern. Should be a short and descriptive set of words.

- **Problem**, is a description of what problem is that particular pattern trying to solve. This will constraint the set of contexts where the pattern should be used.
- **Solution**, describes how the problem should be solved. This is the heart of the pattern.
- **Consequences**, describes a set of side effects that can happen by using this pattern.

There are other fields that are often included as well like an alias, the author name or the version. These attributes are very generic and used on most patterns catalogues.

The *Name* and *Consequences* are descriptive fields that should be interpreted by humans. So they should be represented in natural language.

The *Problem* defines the context of use for the pattern. In section 4.1 we defined that the input models should form a pattern instantiation. This means that the problem should be defined as a domain and a task models.

The *Solution* should represent the end result of using a pattern. We are studying the first stage of the FEM and the result of this stage is an AUI model. Thus the solution in the pattern should be the resulting AUI model for the task and domain models provided by the problem. This AUI model will serve as a template for the pattern instantiation's AUI model.

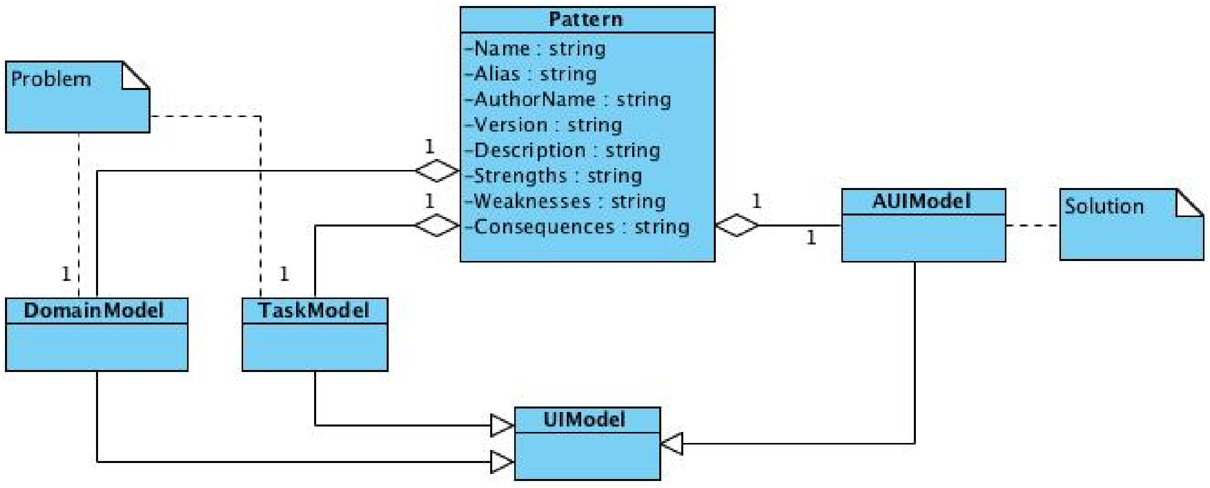


Figure 4.3: UML specification of user interface conceptual pattern.

Figure 4.3 shows the UML specification of a pattern. It contains a set of descriptive fields, a Task and Domain models (problem) and an Abstract User interface model (solution).

The structure of a pattern for this concept is similar to the one found in [8]. A pattern has a set of descriptive elements, namely:

- **Name**, the name of the pattern;
- **Alias**, a secondary name for the pattern, this field is optional;
- **AuthorName**, the name of the pattern's author;
- **Version**, the current version of the pattern;
- **Description**, a short text description of the pattern, meant to be read by humans;
- **Strengths**, a short text describing the best features of the pattern, this field is optional;
- **Weaknesses**, a short text describing what's worse about the patter or what could be improved, this field is optional;
- **Consequences**, a short text describing side effects that can occur while using this pattern, this field is also optional.

The "problem" that the pattern is trying to solve is represented by a domain and a task model. The "solution" is represented by an AUI model.

4.3 Example

In section 4.1 was introduced a concept to automate the first stage of the FEM based on patterns. Section 4.2 defined the structure of the patterns needed for the concept to work. In this section we'll see an example of a pattern. Also we'll use this pattern to demonstrate the work-flow of the concept described in section 4.1.

For the sake of this example we'll use a pattern for the creation of documents.

Table 6.2 shows the descriptive fields for the *Create Document* pattern.

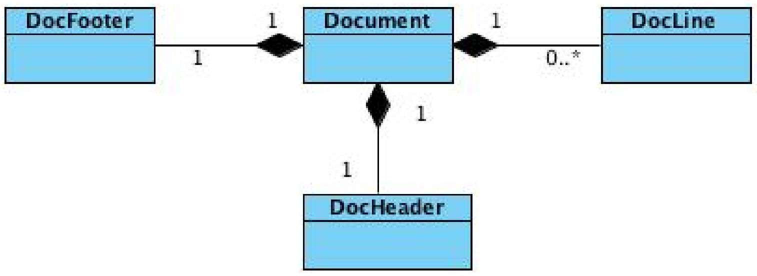


Figure 4.4: UML specification of the *Create document* pattern's domain model.

Name	Create Document
Alias	Build Document
Author	André Barbosa
Version	1.0
Description	This pattern describes the creation of a document. It goes from the structure of the document itself in the domain model to the specification of the task. The resulting interface contains a container for the every different section of the document and a separate container for the document lines that should only appear after the user call the "New Line" function.
Strengths	The usage of this pattern results in a familiar interface that most users will recognize and understand without any external help.
Weaknesses	The usage of this pattern may cause a poor user experience in smaller screens.
Consequences	There aren't any consequences by the use of this pattern besides user's satisfaction.

Table 4.1: Descriptive fields for the Create Document pattern.

Figure 6.1 shows the domain model for the *Create document* pattern specified in UML. The domain model for this pattern represents a document and it can and should be reused in other document related patterns. This model contains a *Document* class that is composed by a header represented by the *DocHeader* class, a set of lines represented by the *DocLine* class and a footer represented by the *DocFooter* class.

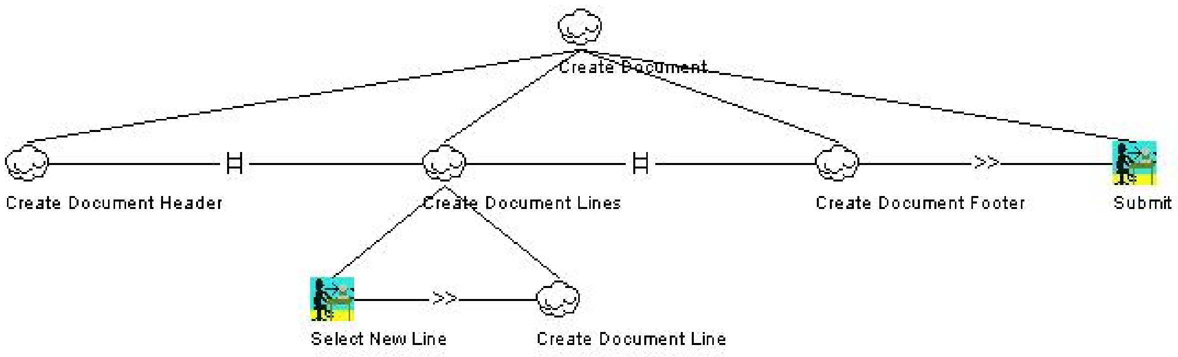


Figure 4.5: Task model for the *Create document* pattern.

Figure 4.5 shows the task model for the *Create document* pattern. There's a main abstract task named *Create Document* at the top followed by other three abstract tasks and an interaction task named *SubmitDocument*. The *Create Document Header* and *CreateDocumentFooter* abstract tasks don't have any child tasks because there isn't enough details in the domain model to go further. The *CreateDocumentLines* task has two children tasks, an interaction task named *ClickNewDocumentLine* that gives access to the *CreateDocumentLine* abstract task that also doesn't have any child.

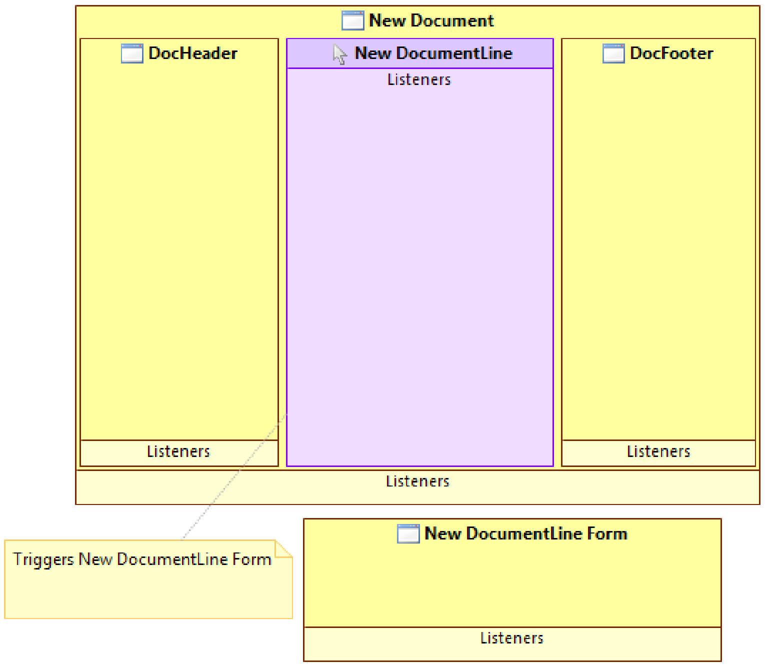


Figure 4.6: Abstract User Interface model for the *Create document* pattern.

Figure 4.6 shows the Abstract User Interface model for the *Create document* pattern. There is a container called *NewDocument* composed by three elements. *DocHeader* and *DocFooter* are two containers that are related to the *CreateDocumentHeader* and *Create-DocumentFooter* abstract tasks respectively. The *NewDocumentLine* listener is related to the *ClickNewDocumentLine* interaction task and should trigger the *NewDocumentLine* container that displays a form for the *CreateDocumentLine* task.

To continue with this example, let's create an instantiation of the *Create Document* pattern. A classic type of documents is an invoice.

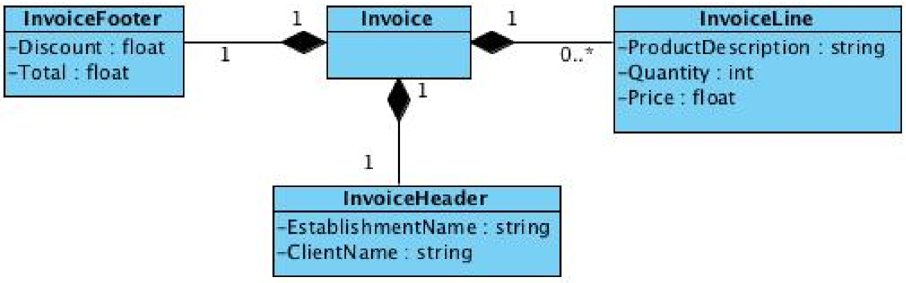


Figure 4.7: UML specification of an invoice's domain model.

Figure 4.7 shows the domain model of an invoice. As expected it's very similar to the domain model of the *Create Document* pattern. There's a class named *Invoice* that is composed by an *InvoiceHeader*, an *InvoiceFooter* and a set of *InvoiceLine* objects. The main difference between this model and the early one is that this one has more details.

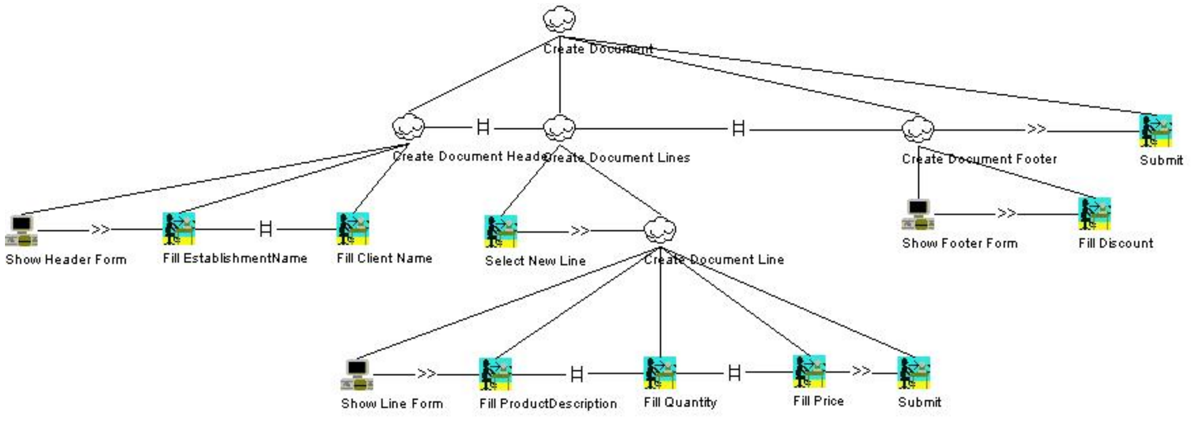


Figure 4.8: Task model to create an invoice.

Figure 4.8 shows the task model to create an invoice. *CreateDocumentHeader*, *CreateDocumentFooter* and *CreateDocumentLine* were childless tasks in the pattern. This is an instantiation of the pattern and because the domain model is also more detailed, the task model carries more details as well.

By using the method described in section 4.1 we reach the Abstract User Interface mode from figure 4.9. The Abstract User Interface from the pattern in figure 4.6 serves as a template for the new model. The structure defined in the pattern is kept but is now more detailed. The empty containers from the pattern are now filled with input controllers. This is possible because the task model has more information.

4.4 Requirements

In order to satisfy the concept described in section 4.1 the application should implement the requirements listed below.

- **Read and interpret Task, Domain and AUI models.**

  The application needs to be able to load Task, Domain and AUI models from a file to memory. This is essential because these are the components of a pattern as explained on section 4.2. Also the input models for the transformation are a Task

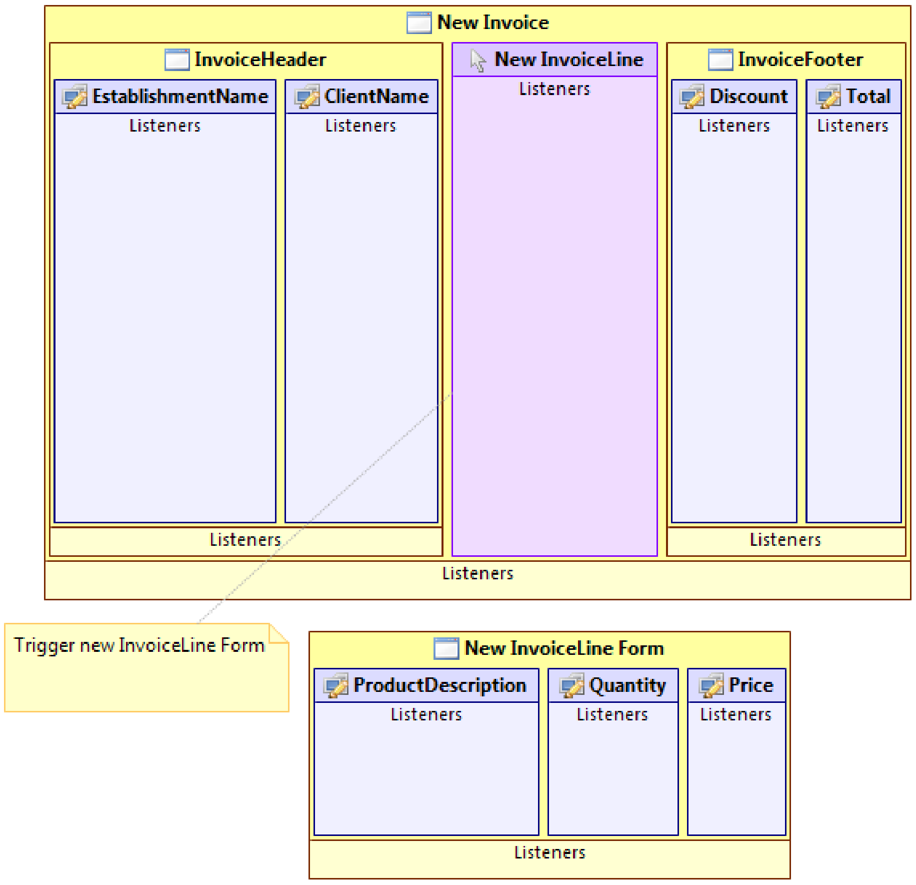


Figure 4.9: Abstract User Interface model to create an invoice.

and a Domain model. The files containing the models should be in the format used by the UsiXML tool set.

- **Validate that the input models are an instantiation of the input pattern.**

  It's essential for the application to validate that the inputs are correct. Otherwise the transformation would give unexpected results.

- **Link the developing models to the pattern models.**

  The application needs a way of linking the input models with the pattern. This is essential for the transformation to occur.

- **Generate an AUI model based on the information provided by the developing models and the pattern models.**

  The main objective of the application is to transform the provided Task and Domain models into an AUI model. this process is achieved by combining the information contained on both the input models and the pattern.

- **Write the resulting AUI model to a file.**

  The AUI model that results from the transformation should be stored in a file compatible with the UsiXML tool set.

Above was described a list of requirements that should be implemented by the application. Bellow is a description of the application's features in the form of user stories.

- "As a developer, I want to load a Task, a Domain and an AUI models from my hard drive so that they form a pattern."
- "As a developer, I want to load a Task and a Domain models from my hard drive so that they can be matched against a pattern."
- "As a developer, I want to select the output directory for the generated AUI model in my hard drive so that I can use it with other tools."

4.5 Algorithm

This section explains the algorithm used to generate AUI model from the input models and a pattern.

There are two restrictions on the models that need to be followed for the algorithm to work. Elements should follow a naming convention and the models that instantiate the pattern should indicate the associated elements in the pattern.

First, there are a few naming conventions for the UsiXML elements used to describe the different models. This conventions serve to match the different elements of different models.

- All names should come in the *CamelCase* format.
- The name of a container (*AbstractCompoundIU*) should equals the name of the class in the domain model that it represents.
- The name of a task should start with a word that represents the action of that task. Supported actions are: "Show", "Click" and "Fill".

An example of this name conventions can be found on the models shown in section 4.1.

The second restriction is that the elements of the pattern instantiation that have an associated element in the pattern should indicate that element. That is accomplished by adding an extra attribute called *pattern_id*, the value should be the *id* of the corresponding element. To demonstrate this, we'll use eXtensible Mark-up Language (XML) representation of the models instead of the graphical one like in previous sections.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<org.usixml.task:TaskModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:org.usixml.task="http://org.usixml.task/2.0">
  <tasks id="1" name="CreateDocument">
    <expressions xsi:type="org.usixml.task:TemporalRelationship" type="ORDERINDEPENDENCE">
      <succ xsi:type="org.usixml.task:TaskDecoration" nature="INTERACTIVE" target="3"/>
      <pred xsi:type="org.usixml.task:TaskDecoration" nature="INTERACTIVE" target="2"/>
    </expressions>
    <expressions xsi:type="org.usixml.task:TemporalRelationship" type="ORDERINDEPENDENCE">
      <succ xsi:type="org.usixml.task:TaskDecoration" nature="INTERACTIVE" target="4"/>
      <pred xsi:type="org.usixml.task:TaskDecoration" nature="INTERACTIVE" target="3"/>
    </expressions>
    <expressions xsi:type="org.usixml.task:TemporalRelationship">
      <succ xsi:type="org.usixml.task:TaskDecoration" nature="INTERACTIVE" target="5"/>
      <pred xsi:type="org.usixml.task:TaskDecoration" nature="INTERACTIVE" target="4"/>
    </expressions>
    <part id="2" name="CreateDocumentHeader"/>
    <part id="3" name="CreateDocumentLines">
      <expressions xsi:type="org.usixml.task:TemporalRelationship">
        <succ xsi:type="org.usixml.task:TaskDecoration" nature="INTERACTIVE" target="7"/>
        <pred xsi:type="org.usixml.task:TaskDecoration" nature="USER" target="6"/>
      </expressions>
      <part id="6" name="ClickNewDocumentLine"/>
      <part id="7" name="CreateDocumentLine"/>
    </part>
    <part id="4" name="CreateDocumentFooter"/>
    <part id="5" name="ClickSubmitDocument"/>
  </tasks>
</org.usixml.task:TaskModel>
```

Code block 4.1: Task model from Create Document pattern

```xml
<?xml version="1.0" encoding="UTF-8"?>
<org.usixml.task:TaskModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:org.usixml.task="http://org.usixml.task/2.0">
  <tasks id="1" pattern_id="1" name="CreateInvoice">
    (...)
    <part id="4" pattern_id="4" name="CreateInvoiceFooter">
      <expressions xsi:type="org.usixml.task:TemporalRelationship">
        <succ xsi:type="org.usixml.task:TaskDecoration" nature="USER" target="16"/>
        <pred xsi:type="org.usixml.task:TaskDecoration" nature="SYSTEM" target="15"/>
      </expressions>
      <part id="15" name="ShowInvoiceFooterForm"/>
      <part id="16" name="FillDiscount"/>
    </part>
    <part id="17" pattern_id="5" name="ClickSubmitInvoice"/>
  </tasks>
</org.usixml.task:TaskModel>
```

Code block 4.2: Create Invoice task model

As you can see in code block 4.2, some tasks (main task represented with an element *tasks* and the others in an element *part*) have an attribute *pattern_id*. This value of *pattern_id* is an integer that can be found as an *id* in a task from code block 4.1.

The algorithm itself is divided into the following steps:

1. Match the elements of the pattern's task and AUI models;
2. Iterate through the elements of pattern's AUI model and for each:
   (a) Get the correspondent task element from the pattern task model;
   (b) Get the correspondent task element from the input task model using the *pattern_id* attribute;
   (c) Replicate the AUI element to the generated AUI model, with a name inferred from the task element;
   (d) Recursive call through child AUI elements;
   (e) Iterate through current input task model elements without *pattern_id* and try to generate a correspondent AUI element using the naming convention.

At the end of this algorithm the application has generated a new AUI model based on the provided pattern. The pattern's template provides the structure and disposition of the final AUI model, while the action word in the name of each task element provides the type of AUI element to use.

Chapter 5

UsiXML patterns tool

Given the requirements for the project presented on section 4.4 we are now able to start developing the application that will support the model transformation. In this chapter we'll present a prototype for this concept.

The architecture for this project is divided in six packages with different objectives that will be introduced below:

- ***org.usixml***

  Contains abstract classes to represent USer Interface eXtensible Mark-up Language (UsiXML) models and elements.

- ***org.usixml.aui***

  Contains classes to represent both the Abstract User Interface (AUI) model and it's elements.

- ***org.usixml.domain***

  Contains classes to represent both the Domain model and it's elements.

- ***org.usixml.task***

  Contains classes to represent both the Task model and it's elements.

- ***usixmlpatterns***

  It's the main package of the application. Contains the class representation for a pattern and the *Main* method that runs the interface.

- ***utils***

This package contains some utility classes and methods that are used throughout the application but aren't part of the core of the project.

In the subsections that follows we'll go into detail for each package and it's components.

5.1 Package org.usixml

Figure 5.1 shows the *org.usixml* package. This package contains some abstract classes to help represent the UsiXML models and elements.

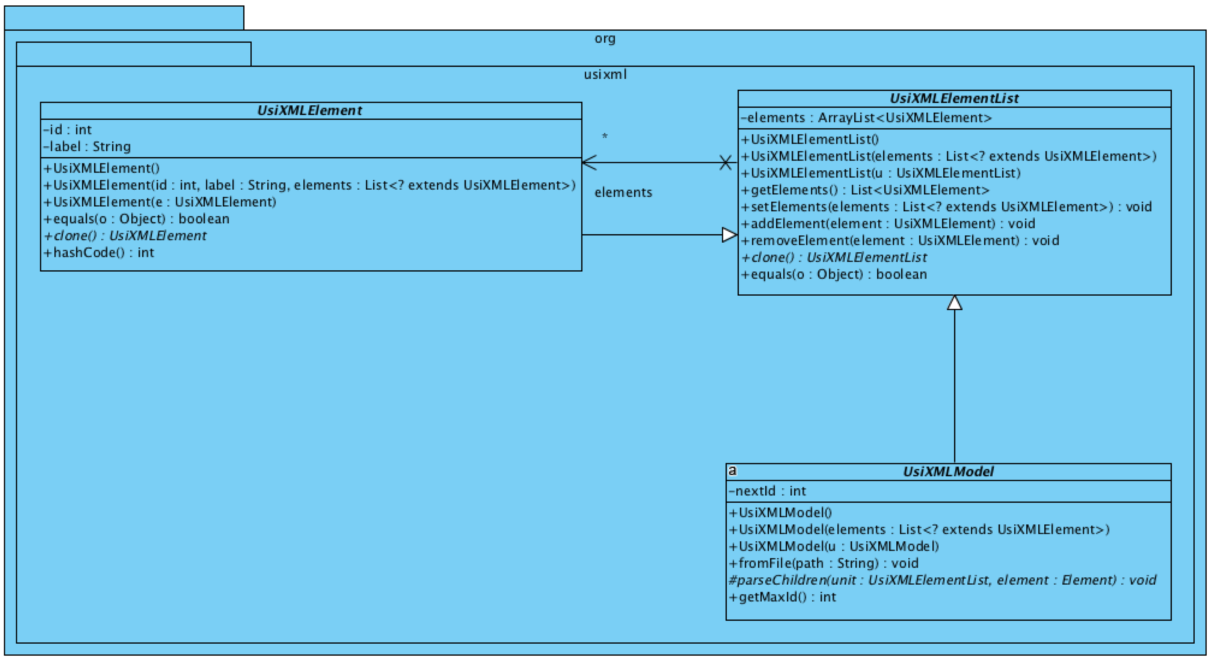

org
usixml
UsiXMLElement
-id : int
-label : String
+UsiXMLElement()
+UsiXMLElement(id : int, label : String, elements : List<? extends UsiXMLElement>)
+UsiXMLElement(e : UsiXMLElement)
+equals(o : Object) : boolean
+clone() : UsiXMLElement
+hashCode() : int
*
elements
UsiXMLElementList
-elements : ArrayList<UsiXMLElement>
+UsiXMLElementList()
+UsiXMLElementList(elements : List<? extends UsiXMLElement>)
+UsiXMLElementList(u : UsiXMLElementList)
+getElements() : List<UsiXMLElement>
+setElements(elements : List<? extends UsiXMLElement>) : void
+addElement(element : UsiXMLElement) : void
+removeElement(element : UsiXMLElement) : void
+clone() : UsiXMLElementList
+equals(o : Object) : boolean
a
UsiXMLModel
-nextId : int
+UsiXMLModel()
+UsiXMLModel(elements : List<? extends UsiXMLElement>)
+UsiXMLModel(u : UsiXMLModel)
+fromFile(path : String) : void
#parseChildren(unit : UsiXMLElementList, element : Element) : void
+getMaxId() : int

Figure 5.1: The org.usixml package.

This package is composed by three classes. The *UsiXMLElementList* is the class at the top of the hierarchy. This class manages a list of objects of the type *UsiXMLElement*. The class *UsiXMLElement* inherits from *UsiXMLElementList*. This means that every UsiXML element can have a set of child elements. Finally the class *UsiXMLModel* also inherits from *UsiXMLElementList*, meaning that every UsiXML model is basically a list of UsiXML elements.

5.2 Package org.usixml.aui

Figure 5.2 shows the *org.usixml.aui* package. This package contains a class representation for the AUI model and it's elements.

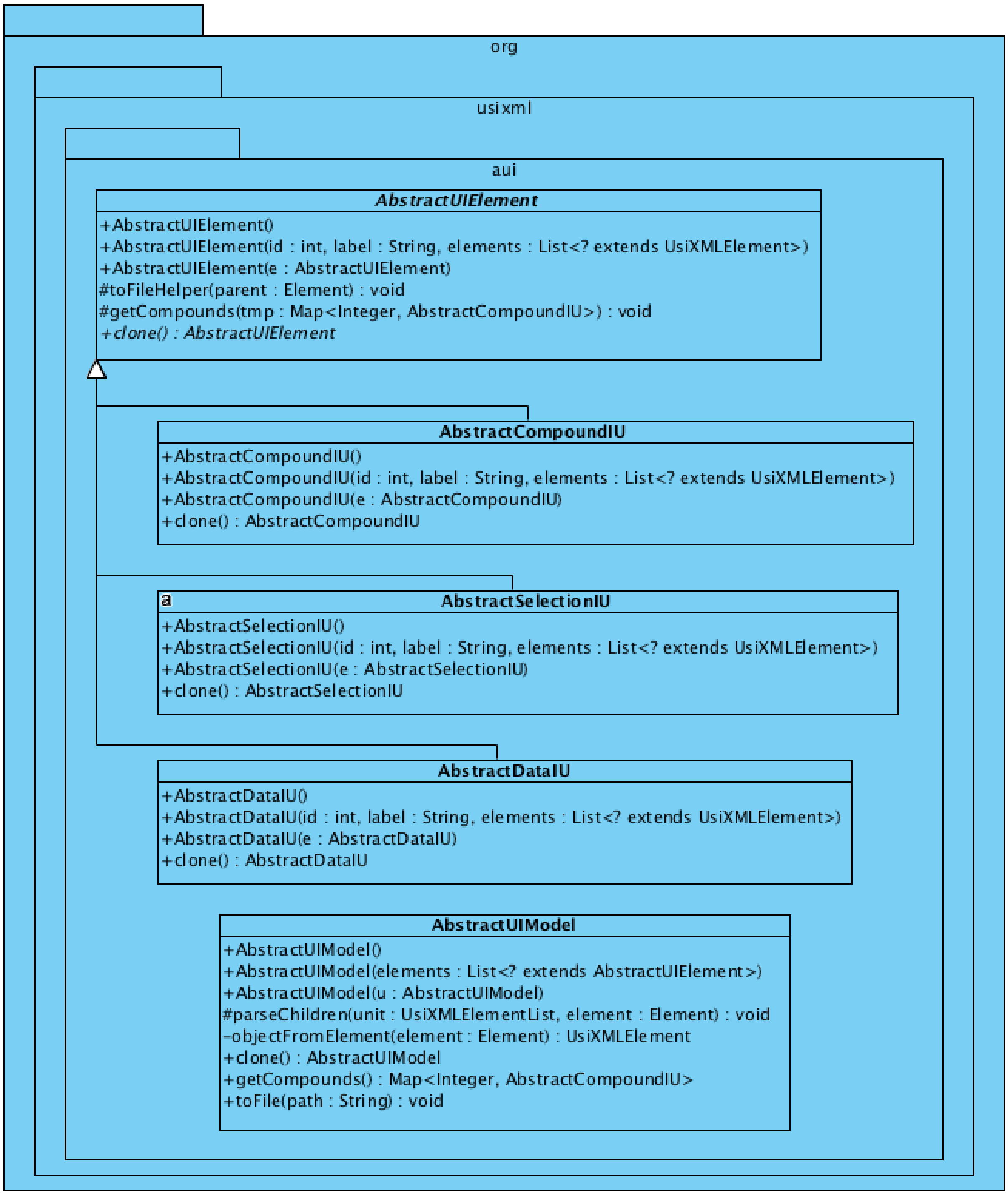

org
usixml
aui
AbstractUIElement
+AbstractUIElement()
+AbstractUIElement(id : int, label : String, elements : List<? extends UsiXMLElement>)
+AbstractUIElement(e : AbstractUIElement)
#toFileHelper(parent : Element) : void
#getCompounds(tmp : Map<Integer, AbstractCompoundIU>) : void
+clone() : AbstractUIElement
AbstractCompoundIU
+AbstractCompoundIU()
+AbstractCompoundIU(id : int, label : String, elements : List<? extends UsiXMLElement>)
+AbstractCompoundIU(e : AbstractCompoundIU)
+clone() : AbstractCompoundIU
a
AbstractSelectionIU
+AbstractSelectionIU()
+AbstractSelectionIU(id : int, label : String, elements : List<? extends UsiXMLElement>)
+AbstractSelectionIU(e : AbstractSelectionIU)
+clone() : AbstractSelectionIU
AbstractDataIU
+AbstractDataIU()
+AbstractDataIU(id : int, label : String, elements : List<? extends UsiXMLElement>)
+AbstractDataIU(e : AbstractDataIU)
+clone() : AbstractDataIU
AbstractUIModel
+AbstractUIModel()
+AbstractUIModel(elements : List<? extends AbstractUIElement>)
+AbstractUIModel(u : AbstractUIModel)
#parseChildren(unit : UsiXMLElementList, element : Element) : void
-objectFromElement(element : Element) : UsiXMLElement
+clone() : AbstractUIModel
+getCompounds() : Map<Integer, AbstractCompoundIU>
+toFile(path : String) : void

Figure 5.2: The org.usixml.aui package.

This package is composed by five classes. The *AbstractUIElement* abstract class inherits from *UsiXMLElement* and it's the superclass of every AUI element. *AbstractCompoundIU*, *AbstractSelectionIU* and *AbstractDataIU* are the AUI elements represented here.

- ***AbstractCompoundIU*** is a container of elements. This element can be used to show an object's properties or a form or other kind of interface element that serves as a container for other elements.
- ***AbstractSelectionIU*** is an interactive element that triggers an event. It's mostly used to represent objects like buttons or links.
- ***AbstractDataIU*** is an input/output element. Should be used to get or set the value of some property.

Lastly, the *AbstractUIModel* represents a AUI model. This class inherits directly from *UsiXMLModel*.

5.3 Package org.usixml.domain

Figure 5.3 shows the *org.usixml.domain* package. This package contains a class representation for the domain model and it's elements.

This package seems more complex then the previous one, but it follows the same structure. The class *Domain Element* inherits from *UsiXMLElement* and it's the superclass of every domain model elements. The class *Class* represents a class composed by attributes an relationships (a class should also have methods but they weren't implemented for simplicity and because they're not as relevant for the interface). The class *Relationship* is an abstract class that inherits from *DomainElement* and serves as the superclass for every type of relationship. The classes *Generalization* and *Association* represent two types of relationships and thus inherit from *Relationship*. Finally there's the class *Attribute* that represents a class attribute. An *Attribute* is also associated with the enumerator *Visibility* that specifies the visibility of that given attribute.

- ***Class*** is a class contained on the domain model. It has the same definition as in all object oriented programming or modelling languages.

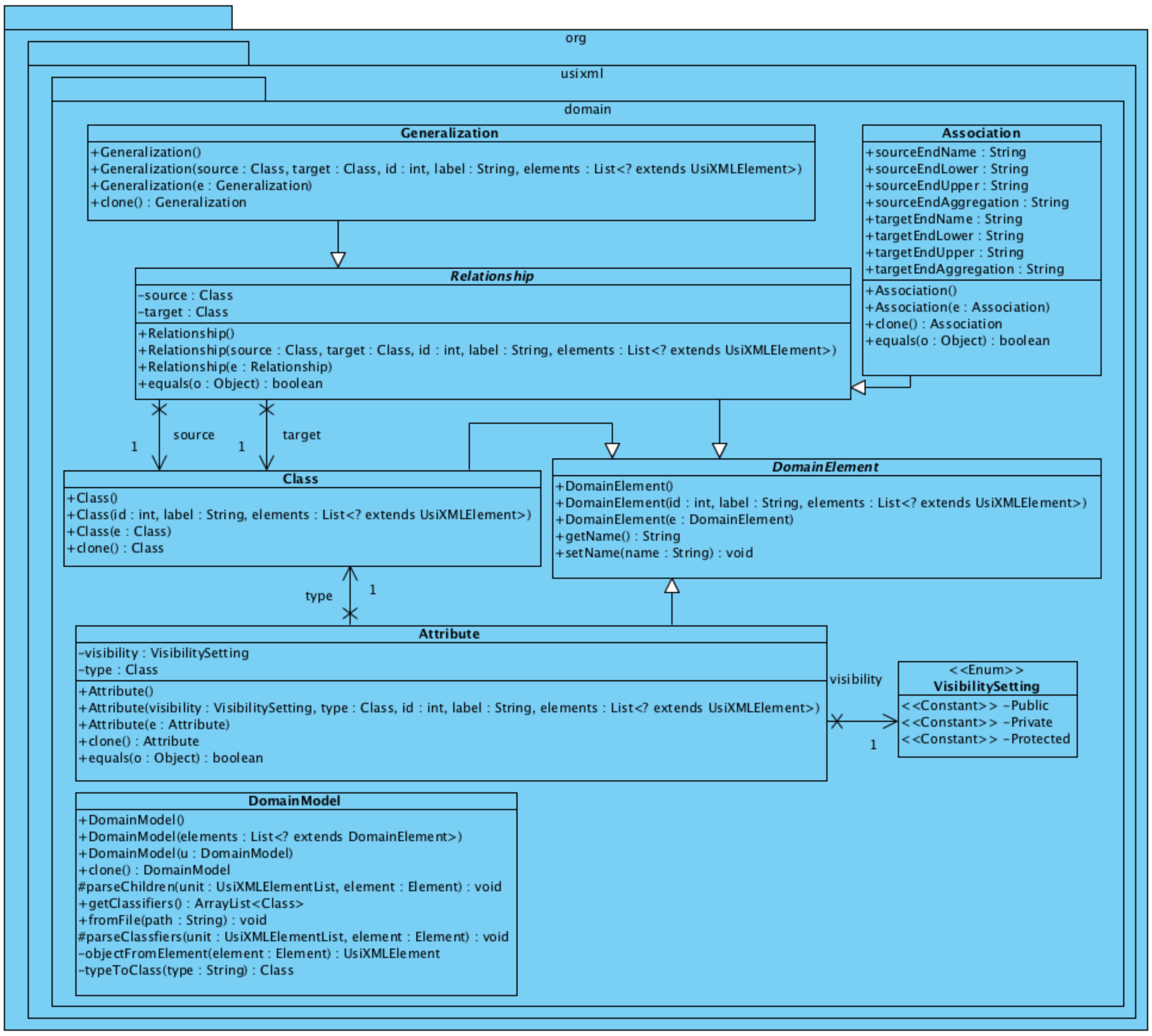

org
usixml
domain
Generalization
+Generalization()
+Generalization(source : Class, target : Class, id : int, label : String, elements : List<? extends UsiXMLElement>)
+Generalization(e : Generalization)
+clone() : Generalization
Association
+sourceEndName : String
+sourceEndLower : String
+sourceEndUpper : String
+sourceEndAggregation : String
+targetEndName : String
+targetEndLower : String
+targetEndUpper : String
+targetEndAggregation : String
+Association()
+Association(e : Association)
+clone() : Association
+equals(o : Object) : boolean
Relationship
-source : Class
-target : Class
+Relationship()
+Relationship(source : Class, target : Class, id : int, label : String, elements : List<? extends UsiXMLElement>)
+Relationship(e : Relationship)
+equals(o : Object) : boolean
source
1
target
1
Class
+Class()
+Class(id : int, label : String, elements : List<? extends UsiXMLElement>)
+Class(e : Class)
+clone() : Class
DomainElement
+DomainElement()
+DomainElement(id : int, label : String, elements : List<? extends UsiXMLElement>)
+DomainElement(e : DomainElement)
+getName() : String
+setName(name : String) : void
type
1
Attribute
-visibility : VisibilitySetting
-type : Class
+Attribute()
+Attribute(visibility : VisibilitySetting, type : Class, id : int, label : String, elements : List<? extends UsiXMLElement>)
+Attribute(e : Attribute)
+clone() : Attribute
+equals(o : Object) : boolean
visibility
1
<<Enum>>
VisibilitySetting
<<Constant>> -Public
<<Constant>> -Private
<<Constant>> -Protected
DomainModel
+DomainModel()
+DomainModel(elements : List<? extends DomainElement>)
+DomainModel(u : DomainModel)
+clone() : DomainModel
#parseChildren(unit : UsiXMLElementList, element : Element) : void
+getClassifiers() : ArrayList<Class>
+fromFile(path : String) : void
#parseClassfiers(unit : UsiXMLElementList, element : Element) : void
-objectFromElement(element : Element) : UsiXMLElement
-typeToClass(type : String) : Class

Figure 5.3: The org.usixml.domain package.

– ***Relationship*** describes how two classes relate to each other. This relationship can be of two types:

  ∗ *Generalization* means that the *source* class is a subclass of the *target* class;

  ∗ *Association* means that the *source* class has an attribute with the type of the *target* class.

– ***Attribute*** is a property of a class. Every attribute as a *visibility* setting, it can be:

  ∗ ***Public*** means that the attribute can be accessed from other classes;

  ∗ ***Protected*** means that the attribute can only be accessed from classes of the same package;

    * ***Private*** means that the attribute can only be accessed from inside the class.

The last class in this package is *DomainModel* that represents the domain model and thus inherits from *UsiXMLModel*

5.4 Package org.usixml.task

Figure 5.4 shows the *org.usixml.task* package. This package contains a class representation for the task model and it's elements.

This package has the same structure has the previous ones. The *TaskElement* class inherits from *UsiXMLElement* and serves as superclass for every element of the task model. The class *Task* represents a task and thus inherits from *TaskElement*, the attribute *pattern_id* is optional and should only be defined in the cases described in section 4.5. The *TemporalRelationship* class represents a temporal relationship between two tasks. It's composed by a *type* which comes from the *TemporalRelationshipType* enumeration, and a predecessor (*pred*) and successor (*succ*). The *pred* and *succ* attributes are instances from the class *TaskDecoration*. A *TaskDecoration* includes a *target* (*Task*) and a *nature* that comes from the *NatureSetting* enumeration.

- ***Task*** is a task that can be accomplished either by the user, the system or both. This element is used to describe any interactive action of the application.
- ***TemporalRelationship*** describes how two tasks are related. There's six types of relation:
    * ***Order Independence***, doesn't matter which task is executed first;
    * ***Enabling***, the execution of the predecessor task enables the execution of the successor task;
    * ***Choice***, the user should choose between the two tasks and execute one of them;
    * ***Disabling***, the execution of the predecessor task disables the execution of the successor task;

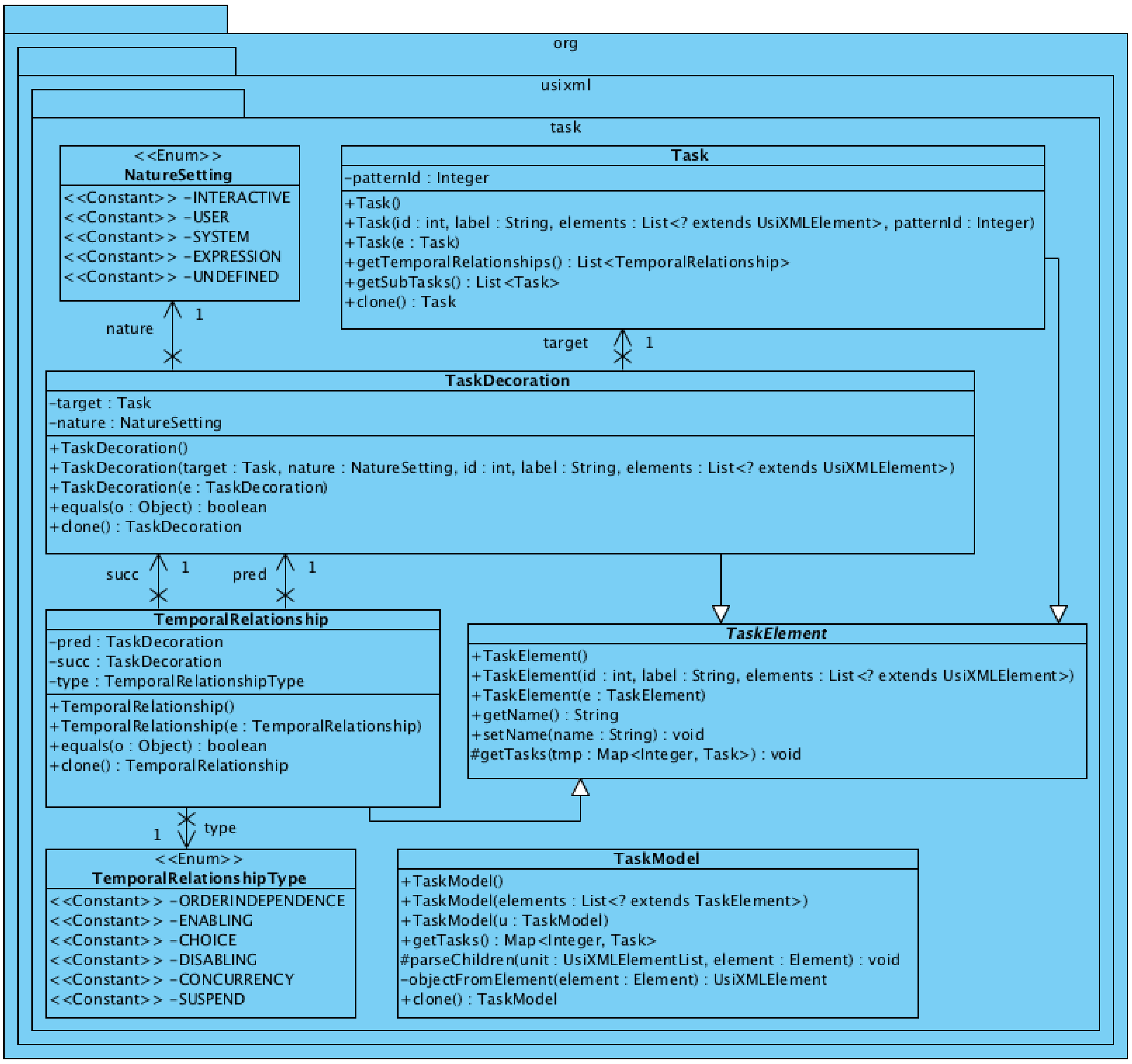

org
usixml
task
<<Enum>>
NatureSetting
<<Constant>> -INTERACTIVE
<<Constant>> -USER
<<Constant>> -SYSTEM
<<Constant>> -EXPRESSION
<<Constant>> -UNDEFINED
Task
-patternId : Integer
+Task()
+Task(id : int, label : String, elements : List<? extends UsiXMLElement>, patternId : Integer)
+Task(e : Task)
+getTemporalRelationships() : List<TemporalRelationship>
+getSubTasks() : List<Task>
+clone() : Task
nature
1
target
1
TaskDecoration
-target : Task
-nature : NatureSetting
+TaskDecoration()
+TaskDecoration(target : Task, nature : NatureSetting, id : int, label : String, elements : List<? extends UsiXMLElement>)
+TaskDecoration(e : TaskDecoration)
+equals(o : Object) : boolean
+clone() : TaskDecoration
succ
1
pred
1
TemporalRelationship
-pred : TaskDecoration
-succ : TaskDecoration
-type : TemporalRelationshipType
+TemporalRelationship()
+TemporalRelationship(e : TemporalRelationship)
+equals(o : Object) : boolean
+clone() : TemporalRelationship
TaskElement
+TaskElement()
+TaskElement(id : int, label : String, elements : List<? extends UsiXMLElement>)
+TaskElement(e : TaskElement)
+getName() : String
+setName(name : String) : void
#getTasks(tmp : Map<Integer, Task>) : void
1
type
<<Enum>>
TemporalRelationshipType
<<Constant>> -ORDERINDEPENDENCE
<<Constant>> -ENABLING
<<Constant>> -CHOICE
<<Constant>> -DISABLING
<<Constant>> -CONCURRENCY
<<Constant>> -SUSPEND
TaskModel
+TaskModel()
+TaskModel(elements : List<? extends TaskElement>)
+TaskModel(u : TaskModel)
+getTasks() : Map<Integer, Task>
#parseChildren(unit : UsiXMLElementList, element : Element) : void
-objectFromElement(element : Element) : UsiXMLElement
+clone() : TaskModel

Figure 5.4: The org.usixml.task package.

  ∗ ***Concurrency***, the predecessor and successor tasks can be executed simultaneously;

  ∗ ***Suspend***, the execution of the predecessor task suspends the execution of the successor task.

– ***TaskDecoration*** wraps the task inside a *TemporalRelationship*. This is useful to add a *nature* to the task that is specific for that order of execution. The nature can be one of the following five types:

  ∗ ***Interactive***, the task is executed with the collaboration of both the user and the system;

* *User*, the task is executed by the user;
* *System*, the task is executed by the system;
* *Expression*;
* *Undefined*.

Lastly, the *TaskModel* class represents the task model and thus inherits directly from *UsiXMLModel*.

5.5 Package usixmlpatterns

Figure 5.5 shows the usixmlpatterns package. This is the most important package in the project, because it contains all the logic that makes the model transformation.

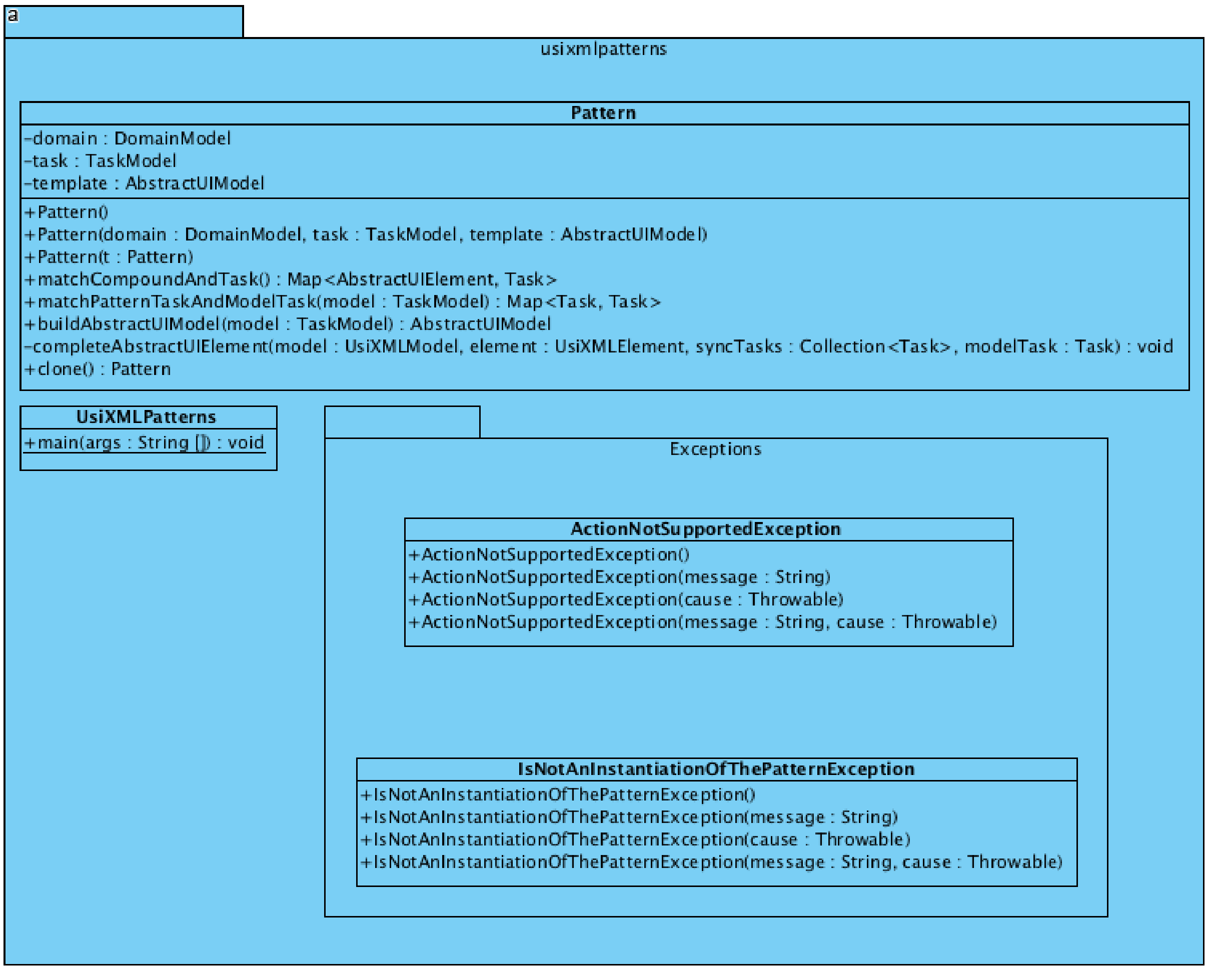

a
usixmlpatterns
Pattern
-domain : DomainModel
-task : TaskModel
-template : AbstractUIModel
+Pattern()
+Pattern(domain : DomainModel, task : TaskModel, template : AbstractUIModel)
+Pattern(t : Pattern)
+matchCompoundAndTask() : Map<AbstractUIElement, Task>
+matchPatternTaskAndModelTask(model : TaskModel) : Map<Task, Task>
+buildAbstractUIModel(model : TaskModel) : AbstractUIModel
-completeAbstractUIElement(model : UsiXMLModel, element : UsiXMLElement, syncTasks : Collection<Task>, modelTask : Task) : void
+clone() : Pattern
UsiXMLPatterns
+main(args : String []) : void
Exceptions
ActionNotSupportedException
+ActionNotSupportedException()
+ActionNotSupportedException(message : String)
+ActionNotSupportedException(cause : Throwable)
+ActionNotSupportedException(message : String, cause : Throwable)
IsNotAnInstantiationOfThePatternException
+IsNotAnInstantiationOfThePatternException()
+IsNotAnInstantiationOfThePatternException(message : String)
+IsNotAnInstantiationOfThePatternException(cause : Throwable)
+IsNotAnInstantiationOfThePatternException(message : String, cause : Throwable)

Figure 5.5: The usixmlpatterns package.

This package is composed by two classes and an exceptions sub-package. The *UsiXMLPatterns* class is the class that contains the *Main* method that starts the application and runs the Graphical User Interface (GUI). The package *Exceptions* contains two classes, *ActionNotSupportedException* and *IsNotAnInstantiationOfThePatternException*. The first one can occur when the program finds a task that it's not part of the pattern and don't know what kind of element will correspond in the AUI model. The second one occurs when the input models don't correspond to an instantiation of the pattern. The *Pattern* class is the most important one. All the logic needed for the models transformation. We'll go into detail of it's components (attributes and methods) below.

As we specified in section 4.2 a pattern is composed by three models. A domain model, a task model and an AUI model that serves as template for the pattern's instantiations. Thus the *Pattern* class has three attributes:

- **domain** with the type *DomainModel* that represents the domain model of the pattern;
- **task** with the type *TaskModel* that represents the task model of the pattern;
- **template** with the type *AbstractUIModel* that represents the AUI model of the pattern and will serve as template to transform pattern's instantiations into other AUI models.

The most important method of the class is the *buildAbstractUIModel* method. This method does the work of generating the AUI model for the provided models using the patterns template as a start point. The details of the algorithm used to generate the AUI model are presented in section 4.5.

5.6 Implementation details

The UsiXMLPatterns application was implemented in Java 7. The only restriction regarding the programming language was that it had to be object oriented because the architecture was based on that principle. From all the available choices, Java was the chosen language for the following reasons:

- familiarity with the language;
- comprehensive set of frameworks, including frameworks for GUI's and for manag-

ing eXtensible Mark-up Language (XML) files which was essential to handle the UsiXML models;

- comprehensive documentation available online free of charge.

We used the most recent version of the language (7), although it wont compile for everybody using an ancient Software Development Kit (SDK) due to its syntactic changes because it's packed with features[1] that help building cleaner code like *Type Inference for Generic Instance Creation* (Code Block 5.1) or *Strings in switch Statements* (Code Block 5.2).

```java
Map<AbstractUIElement, Task> taskAuiModel = new HashMap<>();
```

Code block 5.1: Type Inference for Generic Instance Creation

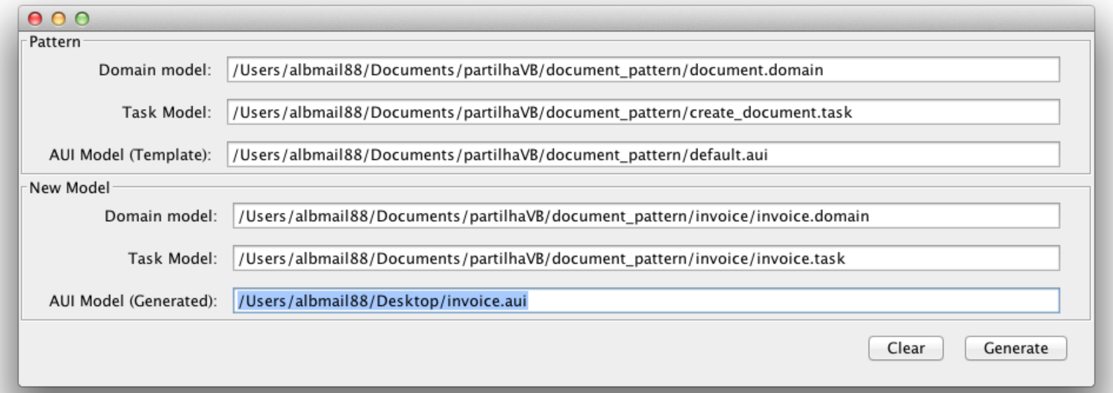
```java
switch(actionName){
    case "Click":
        element.addElement(new AbstractSelectionIU(
                model.getNextId(), nameWithoutAction,
                new ArrayList<UsiXMLElement>()));
        break;
    case "Fill":
        element.addElement(new AbstractDataIU(
                model.getNextId(), nameWithoutAction,
                new ArrayList<UsiXMLElement>()));
        break;
    default:
        throw new ActionNotSupportedException(actionName
                + " "
                + "is not supported. Supported actions "
                + "are: \"Show\", \"Click\" and \"Fill\"");
}
```

Code block 5.2: Strings in switch Statements

The User Interface (UI) for the application was built using the SWING framework. The UI is very small and simple. It consists on a GUI with a set of inputs where the user can insert the path to the pattern's model, the input models and to the location where the generated AUI model should be stored. Figure 5.6 shows the UsiXMLPatterns application UI.

If we click on one of the inputs, the application shows a modal to choose file location, using the `JFileChooser` class from the SWING framework as shown in figure 5.7.

[1]http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html


Pattern

Domain model: /Users/albmail88/Documents/partilhaVB/document_pattern/document.domain

Task Model: /Users/albmail88/Documents/partilhaVB/document_pattern/create_document.task

AUI Model (Template): /Users/albmail88/Documents/partilhaVB/document_pattern/default.aui

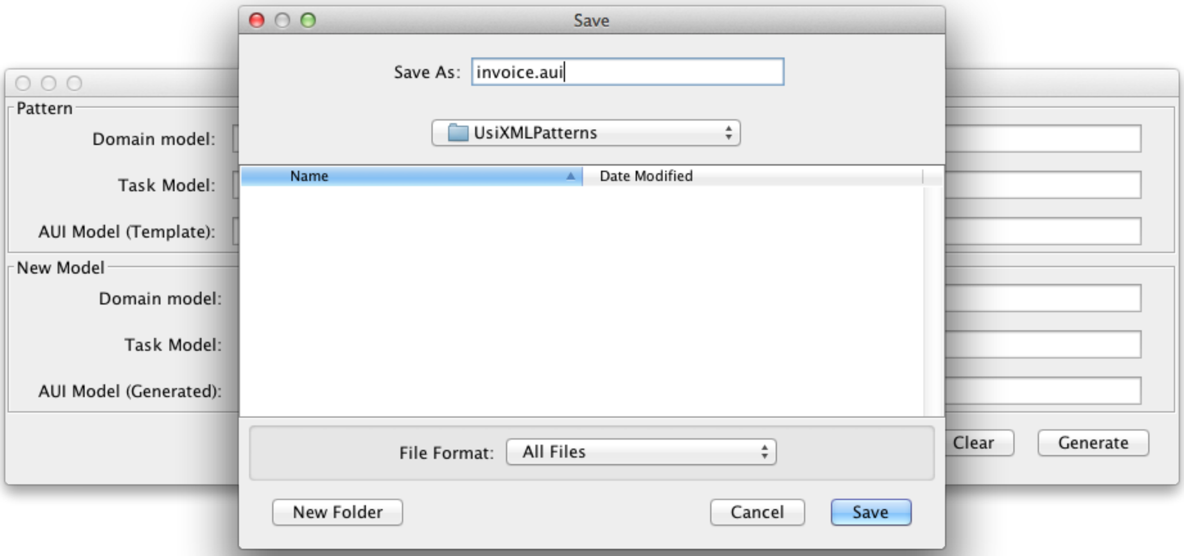
New Model

Domain model: /Users/albmail88/Documents/partilhaVB/document_pattern/invoice/invoice.domain

Task Model: /Users/albmail88/Documents/partilhaVB/document_pattern/invoice/invoice.task

AUI Model (Generated): /Users/albmail88/Desktop/invoice.aui

Figure 5.6: The UsiXMLPatterns UI.


Save

Save As: invoice.aui

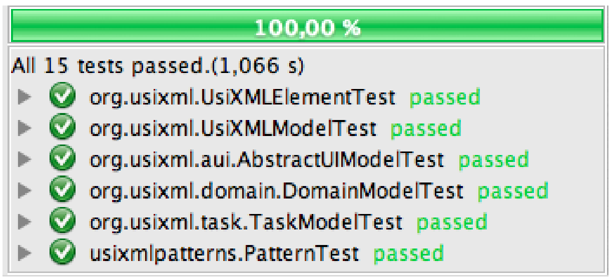
File Format: All Files

Figure 5.7: The UsiXMLPatterns UI to choose a path in the local file system.

In order to ensure that all implemented features are working properly, a test suite was developed using JUnit[2]. This framework was chosen in detriment of others like TestNG[3] because it integrates very well with Netbeans (one of the most popular Java IDE's) and it was already familiar for everyone involved in the development. Figure 5.8 shows the

[2]http://www.junit.org
[3]http://testng.org/doc/index.html

Junit's plugin for Netbeans with all tests passing.



```
100,00 %
All 15 tests passed.(1,066 s)
▶ ✓ org.usixml.UsiXMLElementTest  passed
▶ ✓ org.usixml.UsiXMLModelTest  passed
▶ ✓ org.usixml.aui.AbstractUIModelTest  passed
▶ ✓ org.usixml.domain.DomainModelTest  passed
▶ ✓ org.usixml.task.TaskModelTest  passed
▶ ✓ usixmlpatterns.PatternTest  passed
```

Figure 5.8: Junit tests on Netbeans.

For versions control we used git[4]. Git is free, open-source and thanks to github[5] it's very easy to create a repository. All the code produced for UsiXMLPatterns can be found at `https://github.com/nata79/UsiXMLPatterns`.

[4]http://git-scm.com
[5]https://github.com

Chapter 6

Case study

In this chapter we will use the USer Interface eXtensible Mark-up Language (UsiXML) Patterns prototype to build a case study. As a proof of concept we will build medical prescription management application. A medical prescription is a document so it's possible to reuse the document concept used in Chapter 4. The medical prescription management application will be called Better Hand Writing (BHW) and should cover the following requirements:

- create a new prescription;
- edit an existing prescription;
- remove an existing prescription;
- list all prescriptions.

This chapter will start to document a documents patterns library. These patterns will then be used on the case study application models in order to transform domain and task models into Abstract User Interface (AUI) models as shown in chapter 4 with the tool developed in chapter 5. Ultimately we will present the final result of the application's User Interface (UI) based on the developed models.

6.1 Patterns

A medical prescription is a document type. In that sense we should use a library of document related patterns in order to meet our requirements. In this section we will describe a pattern library with patterns to list, create and edit document(s).

A domain model can be used within different patterns. As this is a document patterns library, all the patterns will have the same domain model that describes a document. This domain model is shown in Figure 6.1 using the Unified Modelling Language (UML).

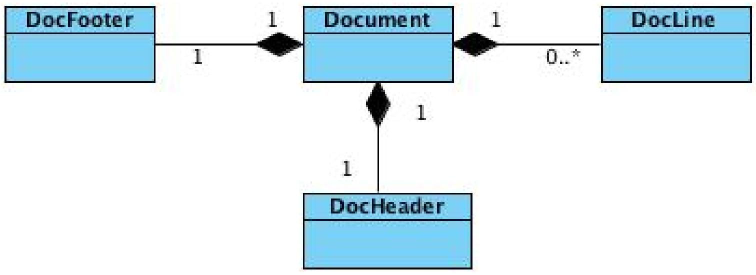


Figure 6.1: UML specification of a document domain model.

A document is composed by a header represented by the class *DocumentHeader*, a footer represented by the class *DocumentFooter* and a set of lines represented by the class *DocumentLine*. Note that this domain model was already used in this document in section 4.3.

6.1.1 List Documents Pattern

In this subsection we will describe the *List Documents Pattern*. This is a pattern for listing a set of documents that can be easily edited by adding new items, editing existing ones or deleting them. As specified in section 4.2 a pattern is composed by a domain model, a task model, an AUI model and a set of descriptive attributes. This subsection starts by the descriptive attributes, then we will describe the task model and ultimately the AUI model.

Table 6.1 shows the descriptive attributes for the *List Documents Pattern*. As these attributes are meant to be read by humans they are written in plain English.

The task model should be created using the tool provided by UsiXML community. This tool uses a notation similar to Concur Task Trees (CTT) [18] and will generate a `task` file that later will be used by the tool developed in this work. The task model developed for this pattern is shown in Figure 6.2.

This task model has a main task called *DocumentList*. This task has five child tasks: *ShowDocumentList*, *SelectDocument*, *EditDocument*, *DeleteDocument* and *NewDocument*.

The *ShowDocumentList* is a system task and is responsible to render the document list to the user. *SelectDocument* is a user task and represents the act of selecting an item on the

Name	List Documents
Alias	Documents Index
Author	André Barbosa
Version	1.0
Description	This pattern describes a list of documents. It shows a list of documents with options for editing or deleting an element of the list and to create a new document.
Strengths	The usage of this pattern results in a familiar interface that most users will recognize and understand without any external help.
Weaknesses	The usage of this pattern may cause a poor user experience in smaller screens.
Consequences	There aren't any consequences by the use of this pattern besides user's satisfaction.

Table 6.1: Documents list pattern description.

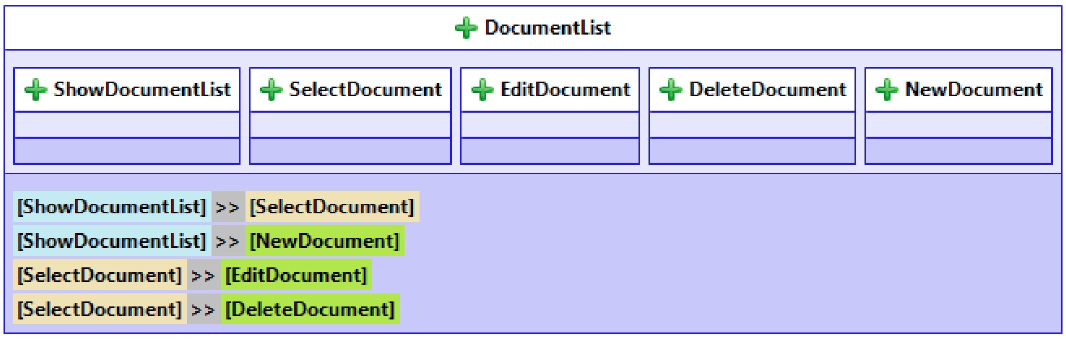

DocumentList
ShowDocumentList
SelectDocument
EditDocument
DeleteDocument
NewDocument
[ShowDocumentList] >> [SelectDocument]
[ShowDocumentList] >> [NewDocument]
[SelectDocument] >> [EditDocument]
[SelectDocument] >> [DeleteDocument]

Figure 6.2: *List documents* pattern's task model.

list by a user. *EditDocument* is an interactive task, meaning that it involves interaction of the user with the system, and it should invoke the *EditDocumentPattern*, with the selected document, that will be described later on this section. *DeleteDocument* is also an interactive task because by performing this task, the user will invoke some behaviour in the system, and in this case, the system will delete the selected document. Finally the *NewDocument* task will invoke the *Create Document Pattern* that will also be described later on this section.

At the lower end of the model, one can see a set of relationships between task decorations. A task decoration is reference to a task with additional information saying who performs the task. It can be performed by the user, the system or both (interactively). In the graphical representation of the model this is represented by a color code, blue for system, orange for user and green for interactive. There is only one type of relationship used in this model. The symbol ">>" means "enables". This means that the *ShowDocumentList* task enables the user to perform the *SelectDocument* and *NewDocument* tasks and the

SelectDocument task enables the user to perform the *EditDocument* and *DeleteDocument* tasks.

For the pattern to be complete we still need to specify an AUI model. This model will serve as template to generate the AUI of the pattern instantiations. Like the task model, the AUI model should also be developed using the appropriate tool provided by the UsiXML community. This tool will generate a `aui` file that will later be used by the tool developed in this work. The AUI model developed for this pattern is shown in Figure 6.3.

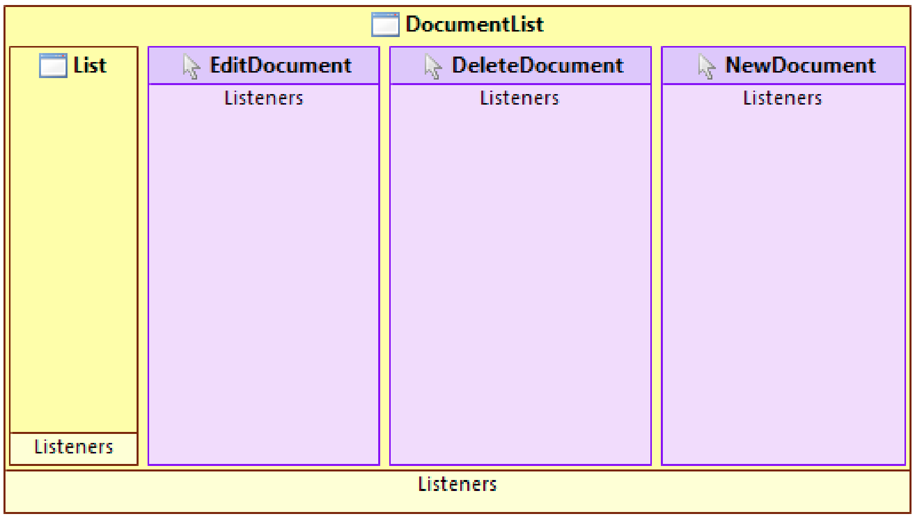


Figure 6.3: *List documents* pattern's task model.

The AUI model is composed by one main *CompoundIU*, which is a container that can be filled with any kind of controllers, a child *CompoundIU* and three *SelectionIUs*, a *SelectionIU* is an interaction unit that in a lower level should be mapped to something like a button or a link. The child *CompoundIU* labelled "List" will be the place where the concrete document list will appear and provides the functionality for a user to select one item. The first *SelectionIU* labelled "EditDocument" should launch the *Edit Document Pattern* with the selected document. The second *SelectionIU* labelled "DeleteDocument" deletes the selected document. Finally, the third *SelectionIU* labelled "NewDocument" should launch the *New Document Pattern*.

6.1.2 Create document pattern

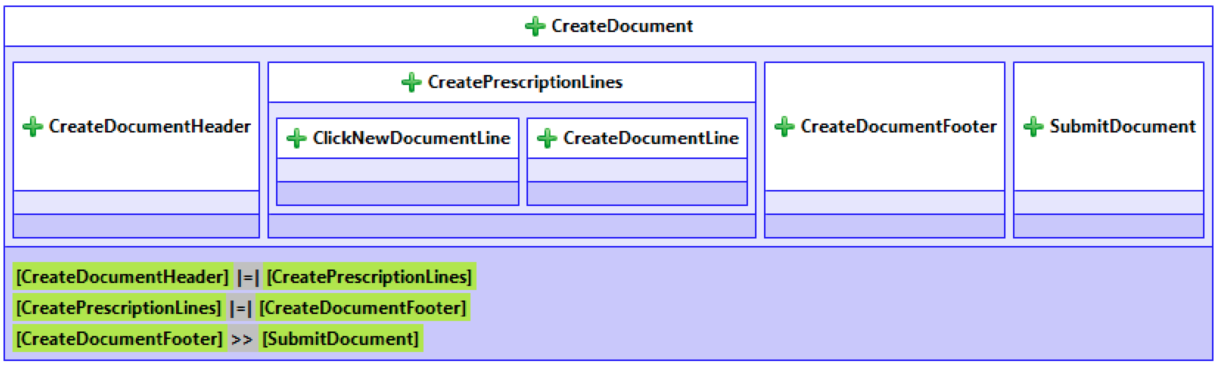
In this subsection we will describe the *Create Document Pattern*. This is a pattern for creating a new document. As in the previous pattern the *Create Document Pattern* is composed by a domain model, a task model, an AUI model and a set of descriptive attributes. This subsection starts by the descriptive attributes, then we will describe the task model an ultimately the AUI model. Note that in section 4.3 was already a definition of a *Create Document Pattern*.

Name	Create Document
Alias	Build Document
Author	André Barbosa
Version	1.0
Description	This pattern describes the creation of a document. It goes from the structure of the document itself in the domain model to the specification of the task. The resulting interface contains a container for the every different section of the document and a separate container for the document lines that should only appear after the user call the "New Line" function.
Strengths	The usage of this pattern results in a familiar interface that most users will recognize and understand without any external help.
Weaknesses	The usage of this pattern may cause a poor user experience in smaller screens.
Consequences	There aren't any consequences by the use of this pattern besides user's satisfaction.

Table 6.2: Descriptive fields for the Create Document pattern.

Table 6.3 shows the descriptive attributes for the *Create Document Pattern*. As these attributes are meant to be read by humans they are written in plain English.

As with the previous pattern, the *Create Document Pattern's* task model should be created using the tool provided by the *UsiXML* community in order to later use the `task` file. The task model developed for this pattern is shown in Figure 6.4.



[CreateDocumentHeader] |=| [CreatePrescriptionLines]
[CreatePrescriptionLines] |=| [CreateDocumentFooter]
[CreateDocumentFooter] >> [SubmitDocument]

Figure 6.4: *Create document* pattern's task model.

This task model has a main task called *CreateDocument*. This task has four child tasks:

CreateDocumentHeader, *CreateDocumentLineList*, *CreateDocumentFooter* and *Submit-Document*.

The *CreateDocumentHeader* task is a user task that provides the user the ability to create the header of a document. *CreateDocumentLines* is an interactive task that allows the user to create document lines for a document. This task is has two child tasks, *ClickNewDocumentLine* and *CreateNewDocumentLine*, the first one enables the second allowing the user to create a new document line. *CreateDocumentFooter* is a user task that allows the user to create the contents of the document footer. Finally the *SubmitDocument* task is an interactive task that allows the user to submit the new document.

On the bottom of the *CreateDocument* task there is the relationships between the child tasks. Note that there is a symbol that has not been used before, "$| = |$", which means that the right and left task can be executed in any order.

To complete the pattern we now present the specified AUI model. As with the previous pattern, this model will serve as template to generate the AUI of the pattern instantiations. And as for the other models, we used the tool provided by the UsiXML community in order to benefit from the `.aui` file that will later be used by the tool developed in this work. The AUI model developed for this pattern is shown in Figure 6.5.

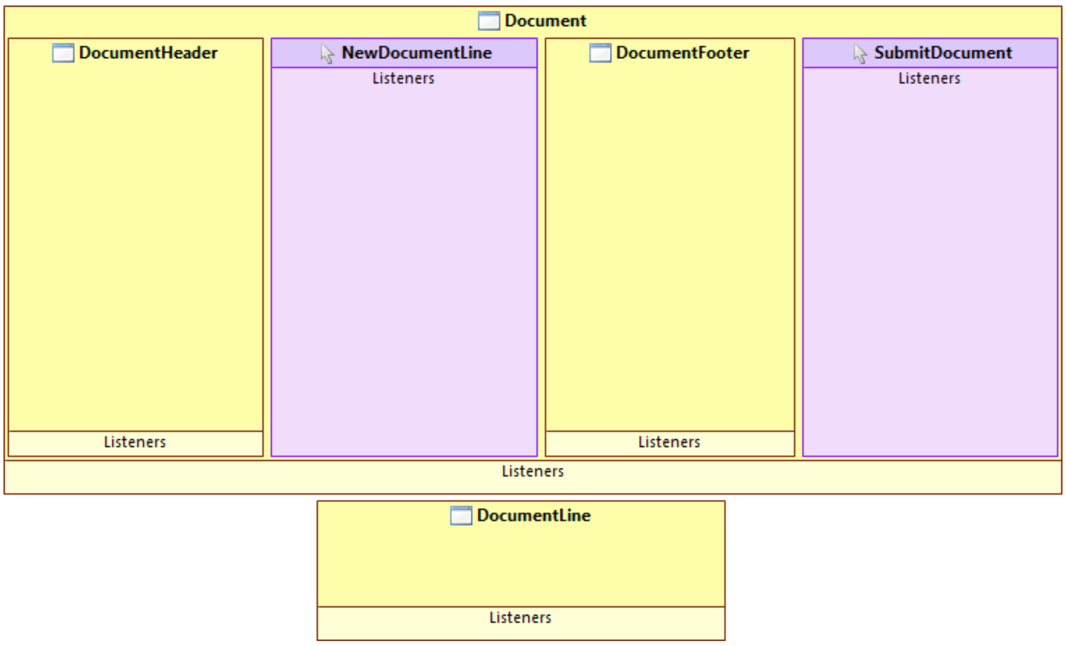


Figure 6.5: *Create document* pattern's task model.

The AUI model is composed by two main *CompoundIUs*. The first one refers to the document itself and second refers to an individual document line. The *CompoundIU* labelled *Document* is composed by two child *CompoundIUs* and two *SelectionIUs*. The first *CompoudIU* refers to the *DocumentHeader* and should contain inputs to create the *DocumentHeader* attributes, the second *CompoudIU* refers to the *DocumentFooter* and should contain inputs to create the *DocumentFooter* attributes. The *SelectionIU* labelled *NewDocumentLine* should trigger the *CompoundIU* labelled *DocumentLine* that will allow the user to create a new *DocumentLine*. Finally the *SelectionIU* labelled *SubmitDocument* should submit the changes the user made to the document.

6.1.3 Edit document pattern

In this subsection we will describe the *Edit Document Pattern*. This is a pattern for editing a document. As the previous pattern the *Edit Document Pattern* is composed by a domain model, a task model, an AUI model and a set of descriptive attributes. This subsection starts by the descriptive attributes, then we will describe the task model an ultimately the AUI model.

Name	Edit Document
Alias	Update Document
Author	André Barbosa
Version	1.0
Description	This pattern describes the edition of a document. It allows a user to edit an existing document.
Strengths	The usage of this pattern results in a familiar interface that most users will recognize and understand without any external help.
Weaknesses	The usage of this pattern may cause a poor user experience in smaller screens.
Consequences	There aren't any consequences by the use of this pattern besides user's satisfaction.

Table 6.3: Edit document pattern.

Table 6.3 shows the descriptive attributes for the *Edit Document Pattern*. As these attributes are meant to be read by humans they are written in plain English.

As with the previous pattern, the *Edit Document Pattern's* task model should be created using the tool provided by the *UsiXML* community in order to later use the `task` file. The task model developed for this pattern is shown in Figure 6.6.

This task model has a main task called *EditDocument*. This task has four child tasks: *EditDocumentHeader*, *EditDocumentLineList*, *EditDocumentFooter* and *SubmitDocument*.

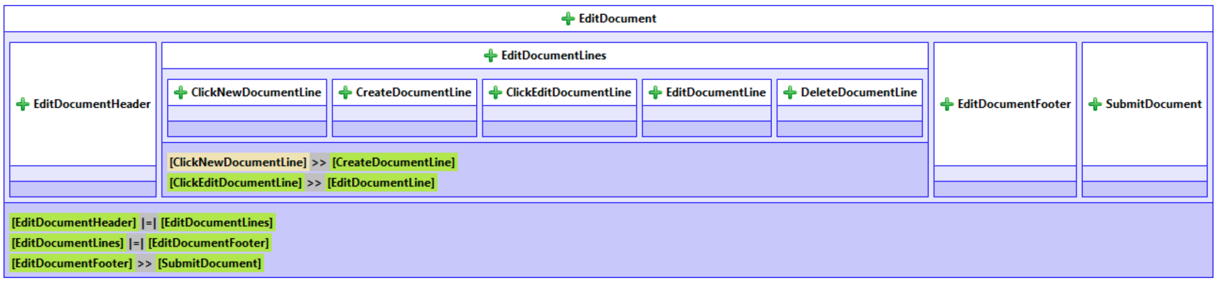

EditDocument
EditDocumentHeader
EditDocumentLines
ClickNewDocumentLine
CreateDocumentLine
ClickEditDocumentLine
EditDocumentLine
DeleteDocumentLine
EditDocumentFooter
SubmitDocument
[ClickNewDocumentLine] >> [CreateDocumentLine]
[ClickEditDocumentLine] >> [EditDocumentLine]
[EditDocumentHeader] |=| [EditDocumentLines]
[EditDocumentLines] |=| [EditDocumentFooter]
[EditDocumentFooter] >> [SubmitDocument]

Figure 6.6: *Edit document* pattern's task model.

The *EditDocumentHeader* task is a user task that provides the user functionality to edit the header of a document. *EditDocumentLines* is an interactive task that allows the user to edit the lines of a document (this task is more complex and will be explained in detail after this paragraph). *EditDocumentFooter* is a user task that allows the user to edit the contents of the document footer. Finally the *SubmitDocument* task is an interactive task that allows the user to submit the changes made to the document.

The *EditDocumentLines* task is composed by five child tasks. These tasks enable the user to select the option of editing and deleting a selected document line or create a new one. The defined relationships indicate the order on which these tasks should be executed.

To complete the pattern we now present the specified AUI model. As with the previous pattern, this model will serve as template to generate the AUI of the pattern instantiations. And as for the other models, we used the tool provided by the UsiXML community in order to benefit from the `.aui` file that will later be used by the tool developed in this work. The AUI model developed for this pattern is shown in Figure 6.7.

The AUI model is composed by two main *CompoundIUs*. The first one refers to the document itself and the second one refers to an individual document line. The *CompoundIU* labelled *Document* is composed by two child *CompoundIUs* and four *SelectionIUs*. The first *CompoudIU* refers to the *DocumentHeader* and should contain inputs to edit the *DocumentHeader* attributes, the second *CompoudIU* refers to the *DocumentFooter* and should contain inputs to edit the *DocumentFooter* attributes. The *SelectionIU* labelled *NewDocumentLine* should trigger the *CompoundIU* labelled *DocumentLine* that will allow the user to create a new *DocumentLine*. The *SelectionIU* labelled *EditDocumentLine* should trigger the *CompoundIU* labelled *DocumentLine* with the selected *DocumentLine* that will allow the user to edit a that *DocumentLine*. The *SelectionIU* labelled *Delete-DocumentLine* should delete the selected *DocumentLine*. Finally the *SelectionIU* labelled *SubmitDocument* should submit the changes the user made to the document.

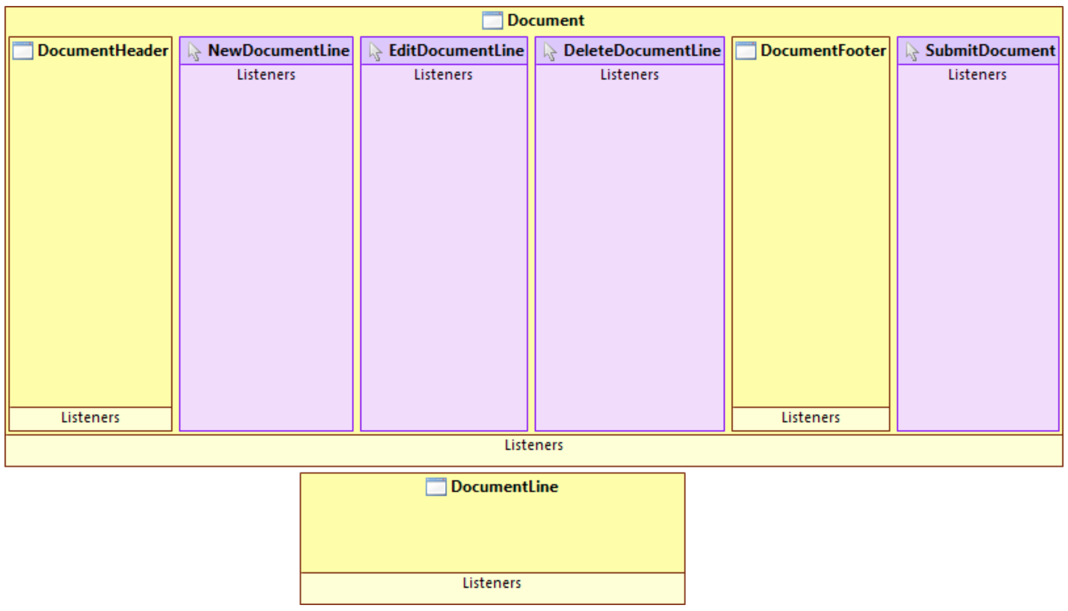


Figure 6.7: *Edit document* pattern's task model.

6.2 Case study models

In this section we will introduce the models specific to our case study, the BHW application. We will build a domain model and three task models that will instantiate the patterns from the previous section and use those patterns to generate the AUI models for the BHW application. Figure 6.8 shows the UML specification of medical prescription.

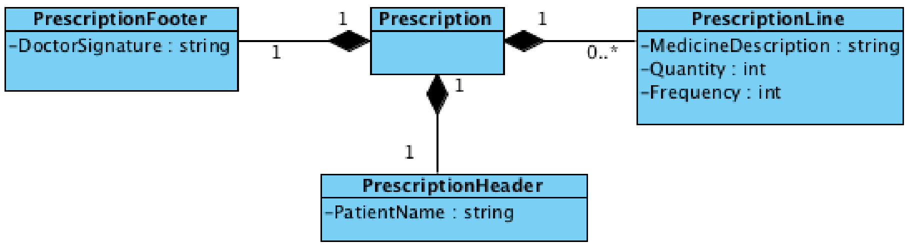


Figure 6.8: UML specification of a medical prescription domain model.

A prescription, like any document is composed by a header represented by the class *PrescriptionHeader*, a footer represented by the class *PrescriptionFooter* and a set of lines represented by the class *PrescriptionLine*.

A *PrescriptionHeader* contains a string attribute for the *PatientName*. The *Prescription-Footer* contains a string attribute for the *DoctorSignature*. Finally, each *PrescriptionLine* is composed by string attribute for the *MedicineDescription*, an integer attribute for the *Qunatity* and an integer attribute for the *Frequency*.

6.2.1 List Prescriptions

To implement the *List Prescriptions* feature we need to start by developing a task model. As we did for the patterns, we have to use the tool provided by the UsiXML community in order to get the `task` file. Figure 6.9 shows the task model to list a set of medical prescriptions.

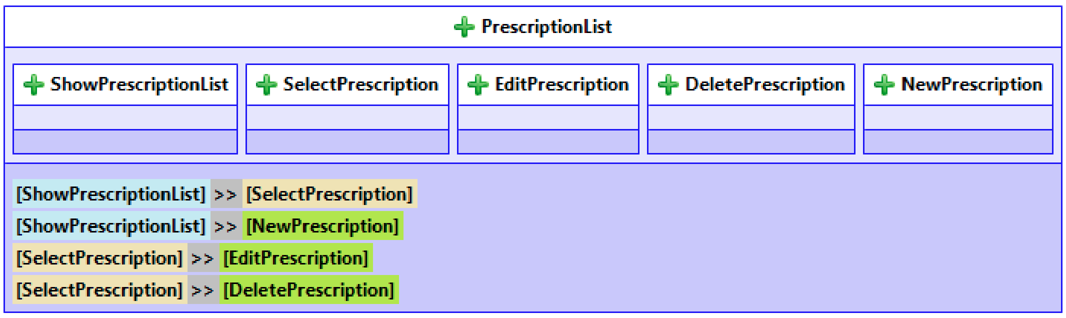

PrescriptionList
ShowPrescriptionList
SelectPrescription
EditPrescription
DeletePrescription
NewPrescription
[ShowPrescriptionList] >> [SelectPrescription]
[ShowPrescriptionList] >> [NewPrescription]
[SelectPrescription] >> [EditPrescription]
[SelectPrescription] >> [DeletePrescription]

Figure 6.9: List prescriptions task model.

This task model is very similar to the one from *List Documents Pattern*, as expected. There is a main task called *PrescriptionList*, that refers to *DocumentList* in the pattern. This task has five child tasks: *ShowPrescriptionList*, *SelectPrescription*, *EditPrescription*, *DeletePrescription* and *NewPrescription*. There is an obvious connection between these tasks and the ones from the pattern's task model. Because this feature only displays a set of items it is fairly simple and doesn't add more complexity from the pattern. Usually the complexity comes when the tasks interacts with more details from the domain model as in the features that we will present below.

The first step to generate the AUI model from the pattern is to link the *List Prescriptions* task model to the task model from the *List Documents Pattern*. Currently the prototype application doesn't support any "smart" way to do this, thus, it should be done directly editing the `task` file with a text editor. For each task that has a correspondent task in the pattern we should add the `pattern_id` property with the value of the `id` of the correspondent task in the pattern's task model. The final result of this process should look like the code block 6.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<org.usixml.task:TaskModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:↩
    xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:org.usixml.task="http://org.↩
    usixml.task/2.0">
  <tasks id="1" name="PrescriptionList" pattern_id="1">
    <expressions xsi:type="org.usixml.task:TemporalRelationship">
      <succ xsi:type="org.usixml.task:TaskDecoration" nature="INTERACTIVE" target="3"/>
      <pred xsi:type="org.usixml.task:TaskDecoration" nature="SYSTEM" target="2"/>
    </expressions>
    <expressions xsi:type="org.usixml.task:TemporalRelationship" type="CHOICE">
      <succ xsi:type="org.usixml.task:TaskDecoration" nature="INTERACTIVE" target="5"/>
      <pred xsi:type="org.usixml.task:TaskDecoration" nature="INTERACTIVE" target="4"/>
    </expressions>
    <part id="2" name="ShowPrescriptionList" pattern_id="2"/>
    <part id="3" name="SelectPrescription" pattern_id="3" />
    <part id="4" name="EditPrescription" pattern_id="4" />
    <part id="5" name="DeletePrescription" pattern_id="5" />
    <part id="6" name="NewPrescription" pattern_id="6" />
  </tasks>
</org.usixml.task:TaskModel>
```

Code block 6.1: Task model list prescriptions

As an example, one can see that the *PrescriptionList* task has the property `pattern_id` with the value of "1" which is the `id` of the *DocumentList* task in the pattern's task model.

After setting the task model we can now use the prototype application from the last chapter to generate the AUI model to the list prescriptions feature as shown in figure 6.10.

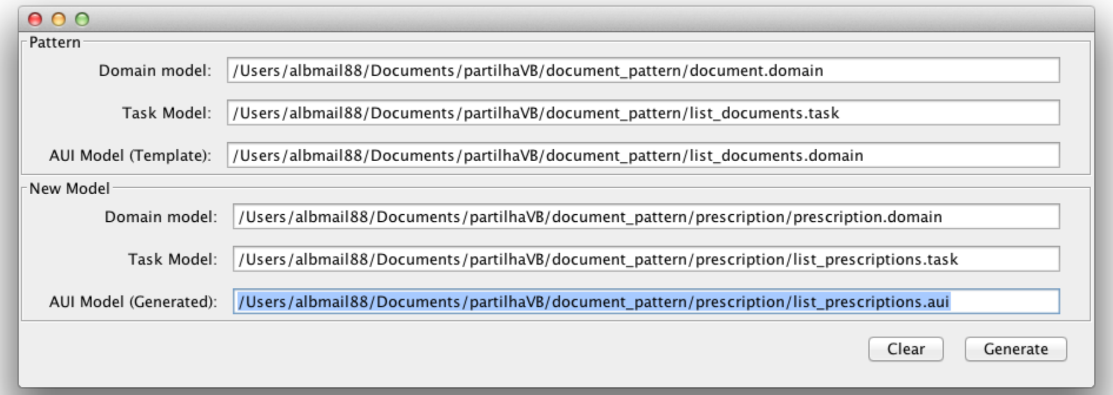

Pattern
Domain model: /Users/albmail88/Documents/partilhaVB/document_pattern/document.domain
Task Model: /Users/albmail88/Documents/partilhaVB/document_pattern/list_documents.task
AUI Model (Template): /Users/albmail88/Documents/partilhaVB/document_pattern/list_documents.domain
New Model
Domain model: /Users/albmail88/Documents/partilhaVB/document_pattern/prescription/prescription.domain
Task Model: /Users/albmail88/Documents/partilhaVB/document_pattern/prescription/list_prescriptions.task
AUI Model (Generated): /Users/albmail88/Documents/partilhaVB/document_pattern/prescription/list_prescriptions.aui

Figure 6.10: Using UsiXMLPatterns to genera the AUI model to list prescriptions.

The application should receive as input the paths for the models that make the pattern (domain, task and AUI models), the paths for the input models (domain and task models) and the path to where the resulting AUI model should be saved.

After clicking the "Generate" button, the application will apply the transformation and save the resulting model on the provided path. The end result of the AUI model is shown in Figure 6.11.

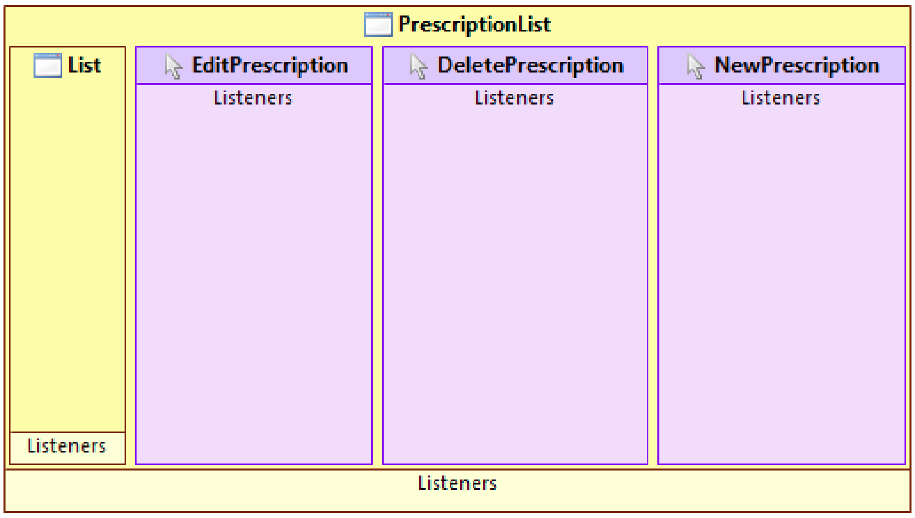


Figure 6.11: AUI model to list prescriptions.

The resulting AUI model follows the exact same structure from the pattern's AUI model. The AUI model is composed by one main *CompoundIU*, a child *CompoundIU* and three *SelectionIUs*. The child *CompoundIU* labelled "List" will be the place where the concrete prescription list will appear and should provide the functionality for a user to select one item. The first *SelectionIU* labelled "EditPrescription" should launch the *Edit Prescription* feature with the selected prescription. The second *SelectionIU* labelled "DeletePrescription" deletes the selected prescription. Finally, the third *SelectionIU* labelled "NewPrescription" should launch the *New Prescription* feature.

6.2.2 Create Prescription

To implement the *Create Prescription* feature we need to start by developing a task model. As we did previously, we are going to use the tool provided by the UsiXML community in order to get the `task` file. Figures 6.12, 6.13 and 6.14 shows the task model to create a medical prescription. Note that the model had to be split in three images to fit on the pages.

This task model has a main task called *CreatePrescription*. This task has four child tasks: *CreatePrescriptionHeader*, *CreatePrescriptionLines*, *CreatePrescriptionFooter* and *SubmitPrescription*.

The *CreatePrescriptionHeader* task is a user task that provides the user functionality to create the header of a prescription. *CreatePrescriptionLines* is an interactive task that allows the user to edit the lines of a document. *CreatePrescriptionFooter* is a user task that allows the user to create the contents of the prescription footer. Finally the *SubmitPrescription* task is an interactive task that allows the user to submit the new prescription.

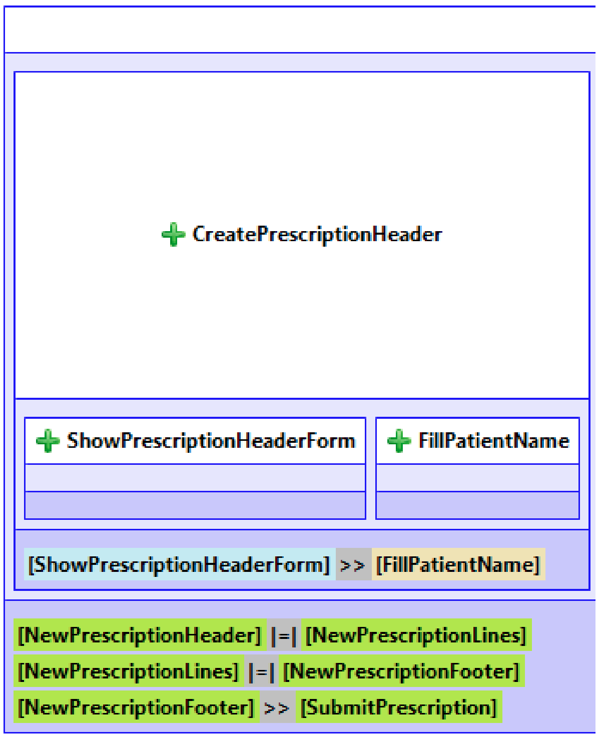

CreatePrescriptionHeader
ShowPrescriptionHeaderForm
FillPatientName
[ShowPrescriptionHeaderForm] >> [FillPatientName]
[NewPrescriptionHeader] |=| [NewPrescriptionLines]
[NewPrescriptionLines] |=| [NewPrescriptionFooter]
[NewPrescriptionFooter] >> [SubmitPrescription]

Figure 6.12: Task model to create a medical prescription - part 1.

The *CreatePrescriptionHeader* task has two child tasks. The *ShowPrescriptionHeader-Form* is a system task that means that the system should render the form before the user interact with it, this temporal relationship is represented using the symbol ">>" at the lower end of the task graphical representation. The second child task is *FillPatient-Name*, this is a user task and means that the user should fill the name of the patient in a form. These child tasks do not have correspondents on the pattern's task model because they are related with attributes present on the prescription's domain model but not on the document's domain model. When the application applies the transformation, it will generate de AUI elements accordingly to the algorithm specified in section 4.5.

The *CreatePrescriptionLines* task is composed by two child tasks. The *ClickNewPre-scriptionLine* is an interactive task that means that the user can ask the system for a form that will enable him to create a prescription line which is the second child task, *CreatePrescriptionLine*. The *CreatePrescriptionLine* task starts with a system task, *ShowPrescriptionLineForm*, then, the user should fill the form as specified by the tasks *FillMedicineDescription*, *FillQuantity* and *FillFrequency*. After filling the form the user

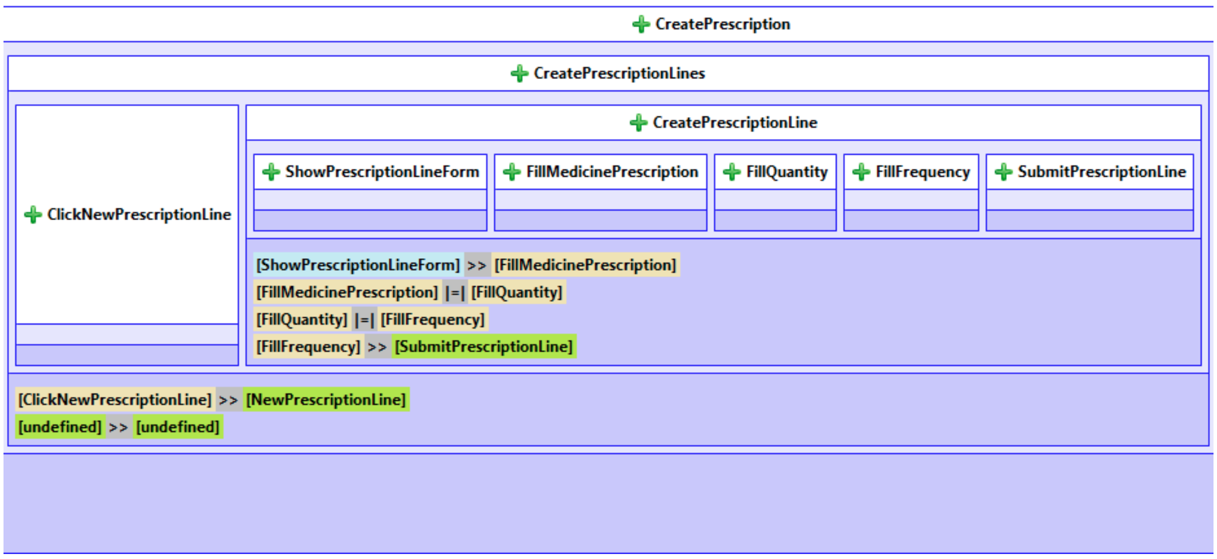

CreatePrescription
CreatePrescriptionLines
ClickNewPrescriptionLine
CreatePrescriptionLine
ShowPrescriptionLineForm
FillMedicinePrescription
FillQuantity
FillFrequency
SubmitPrescriptionLine
[ShowPrescriptionLineForm] >> [FillMedicinePrescription]
[FillMedicinePrescription] |=| [FillQuantity]
[FillQuantity] |=| [FillFrequency]
[FillFrequency] >> [SubmitPrescriptionLine]
[ClickNewPrescriptionLine] >> [NewPrescriptionLine]
[undefined] >> [undefined]

Figure 6.13: Task model to create a medical prescription - part 2.

is enabled to submit the prescription line by the task *SubmitPrescriptionLine*. As we saw earlier in the *CreatePrescriptionHeader* task, there are tasks here that also don't correspond to any task on the pattern's task model. The application will handle them the same way according to transformation's algorithm.

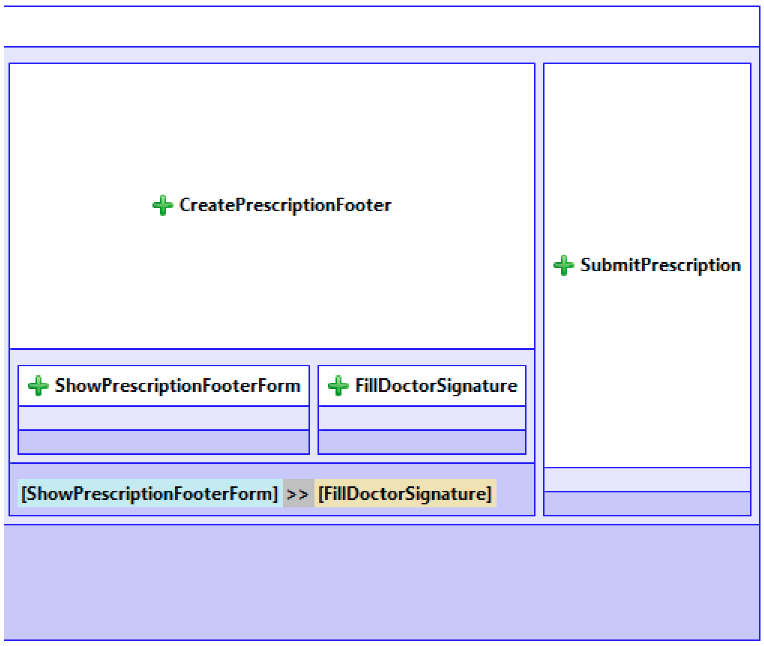

CreatePrescriptionFooter
SubmitPrescription
ShowPrescriptionFooterForm
FillDoctorSignature
[ShowPrescriptionFooterForm] >> [FillDoctorSignature]

Figure 6.14: Task model to create a medical prescription - part 3.

The *CreatePrescriptionFooter* task is very similar in structure with the *CreatePrescriptioHeader* task, we have a *ShowPrescriptionFooterForm* system task that enables a *Fill-*

DoctorSignature user task.

Finally, the *SubmitPrescription* task is enabled after the user completes the earlier ones and let's the user tell the system that he can save this prescription.

As we did for the *List Prescriptions* feature, now we have to link this task model with the task model from the pattern. Again, we have to open the `.task` file with a text editor and add the `pattern_id` property to the tasks that have a direct correspond on the pattern's task model.

After completing the linking process we are able to load the models into the application and perform the transformation to generate the AUI model for the *Create Prescription* feature. Figure 6.15 shows the resulting AUI model.

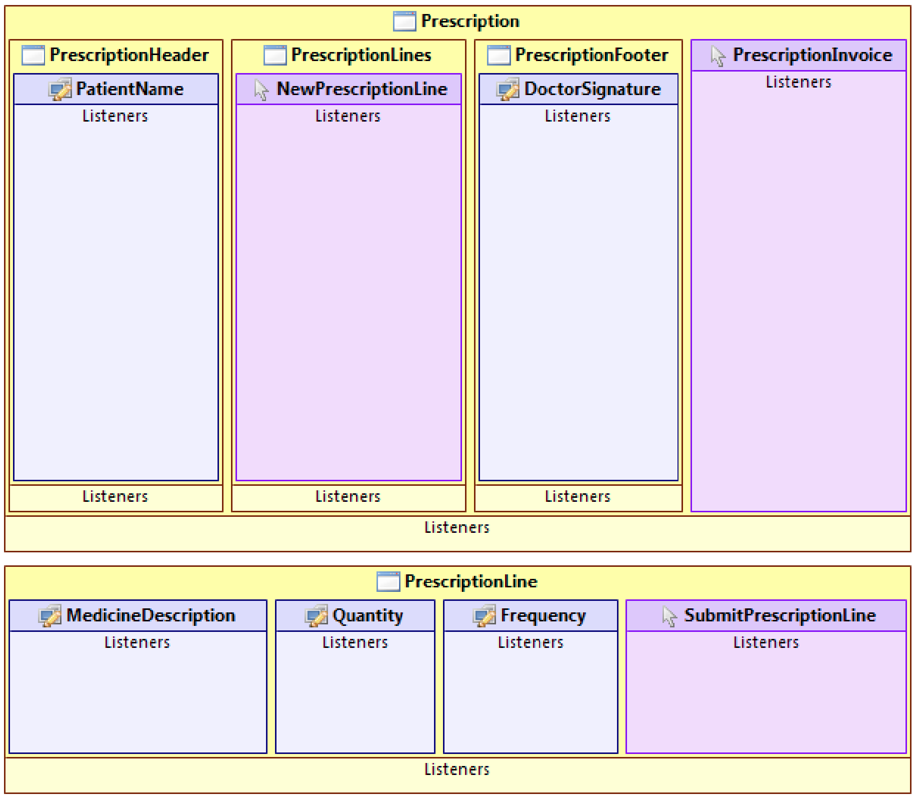


Figure 6.15: AUI model to create a prescription.

This AUI model is very interesting because we can see here some details that we haven't seen with the *List Prescriptions* feature. Comparing it with the AUI model from the *Create Document Pattern* we can notice that they follow the same structure but this one has elements that were not present in the pattern. The *PrescriptionHeader CompoundIU* corresponds to the *DocumentHeader* in the pattern, yet, in the current model, we have a *DataIU* element labelled *PatientName*. A *DataIU* element is controller that expects user input, it should be mapped to some kind input field. This happens because in the task model, there is a task named *FillPatientName* under the task *CreatePrescription-Header* and the application inferred that there should be an *DataIU* with this name in the *PrescriptionHeader CoumpoundIU*. The same behaviour can be observed in the *PrescriptionFooter* and *PrescriptionLine CompoundIUs*.

6.2.3 Edit Prescription

To implement the *Edit Prescription* feature we need to start by developing a task model, following the same process as the above features. Figures 6.16, 6.17, 6.18 and 6.19, show the task model to edit a medical prescription. Note that the model had to be split in four figures to fit on the pages.

This task model has a main task called *EditPrescription*. This task has four child tasks: *EditPrescriptionHeader*, *EditPrescriptionLines*, *EditPrescriptionFooter* and *SubmitPrescription*.

The *EditPrescriptionHeader* task is a user task that provides the functionality for a user to edit the header of a prescription. *EditPrescriptionLines* is an interactive task that allows the user to edit the lines of a document. *EditPrescriptionFooter* is a user task that allows the user to edit the attributes of the prescription footer. Finally the *SubmitPrescription* task is an interactive task that allows the user to submit the changes made to the prescription.

The *EditPrescriptionHeader* task has two child tasks. The *ShowPrescriptionHeaderForm* is a system task that means that the system should render the form before the user interact with it, this temporal relationship is represented using the symbol ">>" at the lower end of the task graphical representation. The second child task is *FillPatientName*, this is a user task and means that the user should fill the name of the patient in a form.

The *EditPrescriptionLines* task is composed by five child tasks. The *ClickNewPrescriptionLine* is an interactive task that means that the user can ask the system for a form that will enable him to create a prescription line. The *ClickEditPrescriptionLine* is an interactive task that means that the user can ask the system for a form that will enable him to edit a selected prescription line. The *ClickDeletePrescriptionLine* is an interactive task that means that the user can ask the system to delete a selected prescription line. Finally *CreatePrescriptionLine* and *EditPrescriptionLine* are meant to let the user fill a

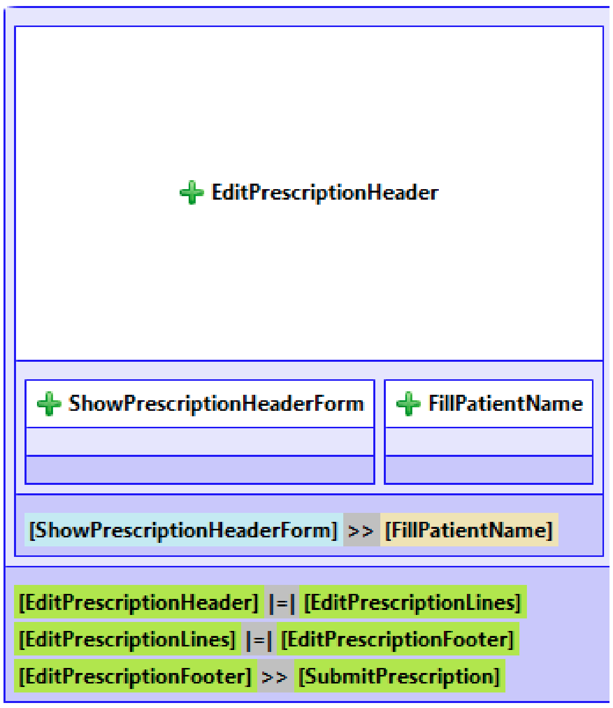

EditPrescriptionHeader
ShowPrescriptionHeaderForm
FillPatientName
[ShowPrescriptionHeaderForm] >> [FillPatientName]
[EditPrescriptionHeader] |=| [EditPrescriptionLines]
[EditPrescriptionLines] |=| [EditPrescriptionFooter]
[EditPrescriptionFooter] >> [SubmitPrescription]

Figure 6.16: Task model to edit a prescription - part 1.

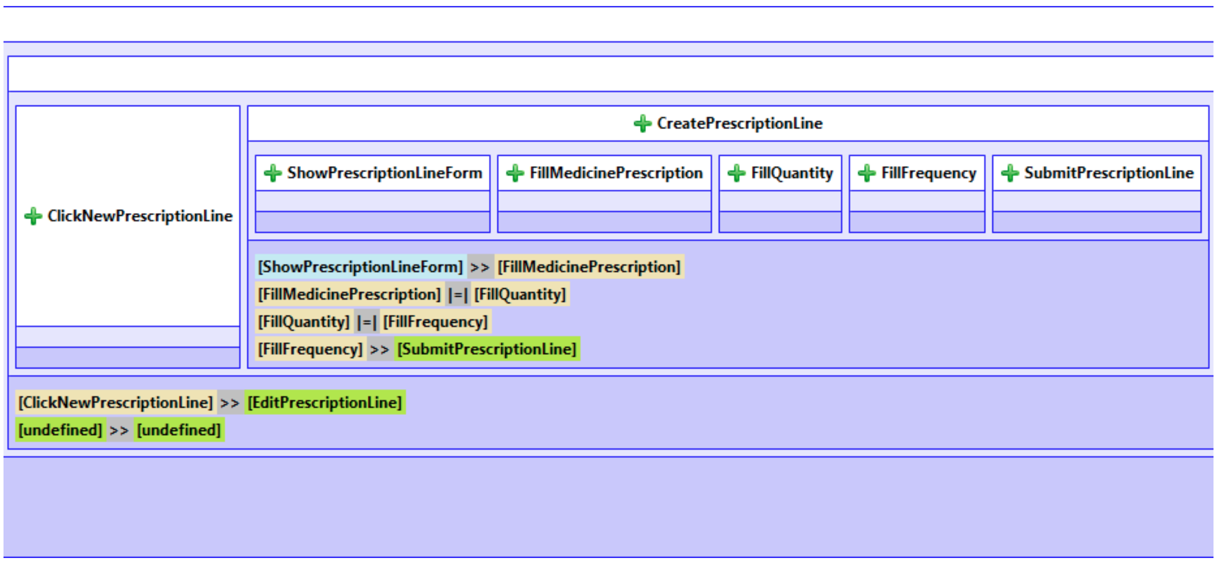

CreatePrescriptionLine
ClickNewPrescriptionLine
ShowPrescriptionLineForm
FillMedicinePrescription
FillQuantity
FillFrequency
SubmitPrescriptionLine
[ShowPrescriptionLineForm] >> [FillMedicinePrescription]
[FillMedicinePrescription] |=| [FillQuantity]
[FillQuantity] |=| [FillFrequency]
[FillFrequency] >> [SubmitPrescriptionLine]
[ClickNewPrescriptionLine] >> [EditPrescriptionLine]
[undefined] >> [undefined]

Figure 6.17: Task model to edit a prescription - part 2.

form for a description line. The difference is that the first is to create a new prescription line and the second to edit an existing one.

The *EditPrescriptionFooter* task is very similar in structure with the *EditPrescriptio-Header* task, we have a *ShowPrescriptionFooterForm* system task that enables a *FillDoctorSignature* user task.

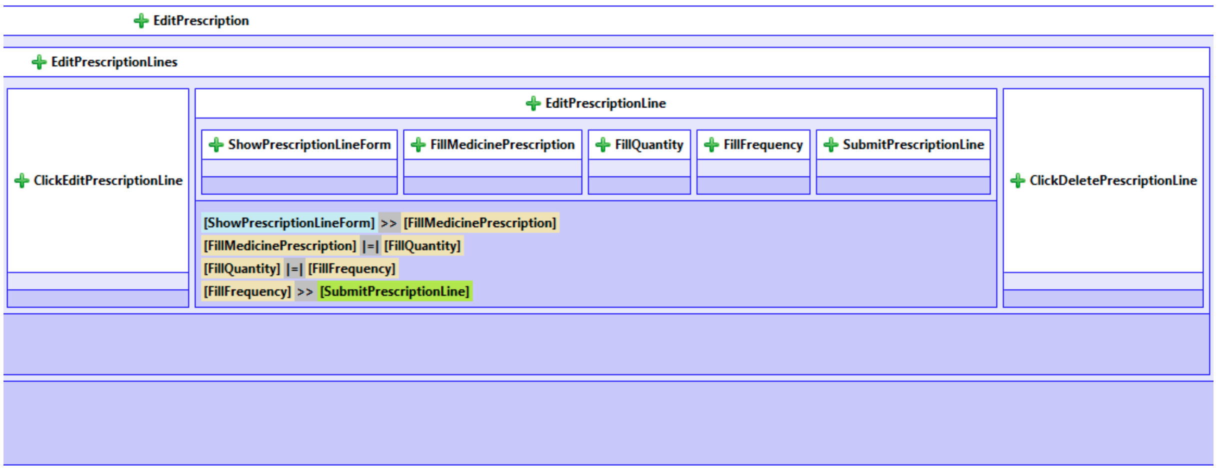

EditPrescription
EditPrescriptionLines
ClickEditPrescriptionLine
EditPrescriptionLine
ShowPrescriptionLineForm
FillMedicinePrescription
FillQuantity
FillFrequency
SubmitPrescriptionLine
ClickDeletePrescriptionLine
[ShowPrescriptionLineForm] >> [FillMedicinePrescription]
[FillMedicinePrescription] |=| [FillQuantity]
[FillQuantity] |=| [FillFrequency]
[FillFrequency] >> [SubmitPrescriptionLine]

Figure 6.18: Task model to edit a prescription - part 3.

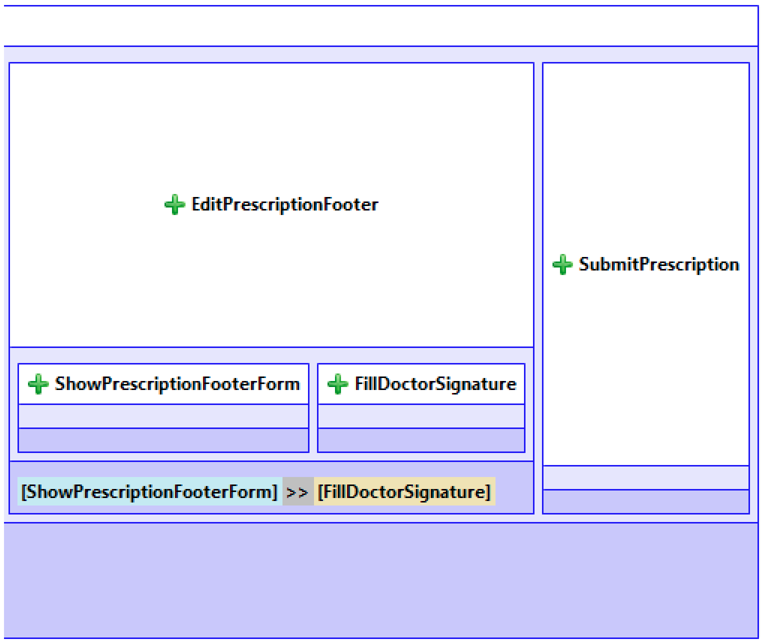

EditPrescriptionFooter
SubmitPrescription
ShowPrescriptionFooterForm
FillDoctorSignature
[ShowPrescriptionFooterForm] >> [FillDoctorSignature]

Figure 6.19: Task model to edit a prescription - part 4.

Finally, the *SubmitPrescription* task is enabled after the user completes the earlier ones and let's the user tell the system that he can save the changes made to the current prescription.

As we did for the *List Prescriptions* and *Create Prescription* features, now we have to link this task model with the task model from the pattern. Again, we have to open the `.task` file with a text editor and add the `pattern_id` property to the tasks that have a direct correspond on the pattern's task model.

After completing the linking process we are able to load the models into the application and perform the transformation to generate the AUI model for the *Edit Prescription* feature. Figure 6.20 shows the resulting AUI model.

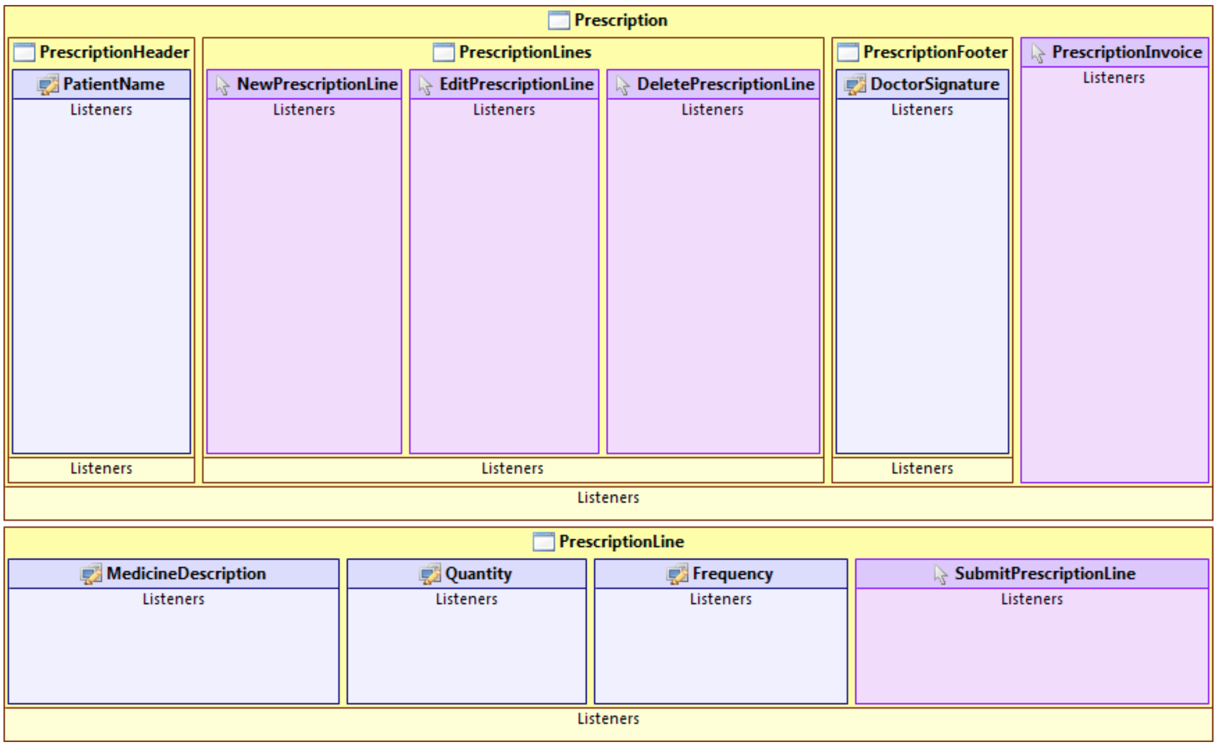
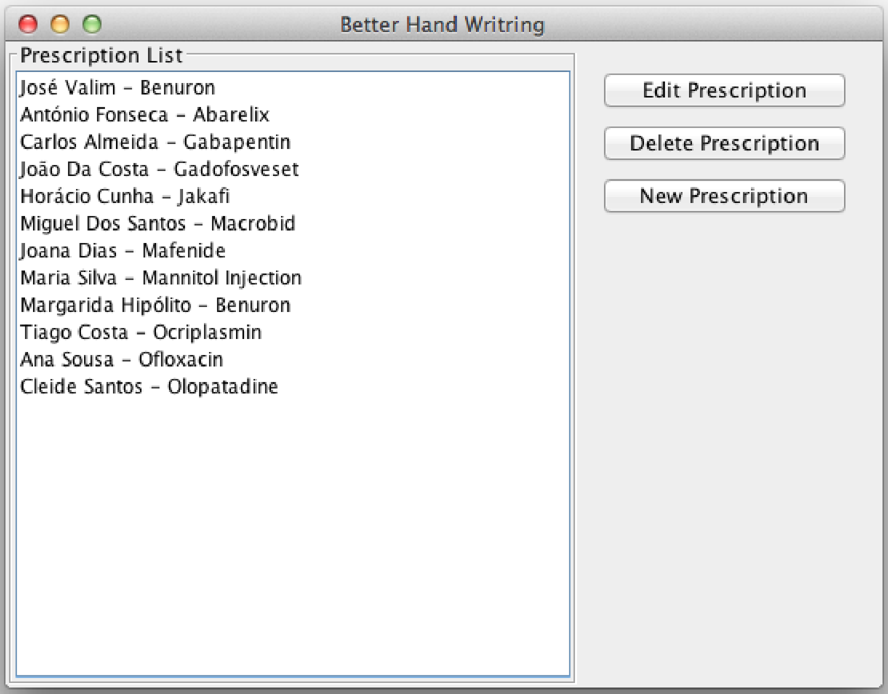


Figure 6.20: AUI model to edit a prescription.

In this AUI model we can see the same effect that we saw earlier with the *Create Prescription* feature. Comparing it with the AUI model from the *Edit Document Pattern* we can notice that they follow the same structure but this one has elements that were not present in the pattern. As in the previous feature, the new elements are related to attributes that are not present in the pattern domain model but exist in the prescription domain model.

6.3 Case Study Implementation

To complete the case study we will present a possible implementation for the BHW application. The UI was developed based on the AUI models developed in this chapter. This implementation was developed using the *Java* language and the *Swing* framework. Figure 6.21 shows the *List Prescriptions* screen and Figure 6.22 shows the *New Prescription* screen.

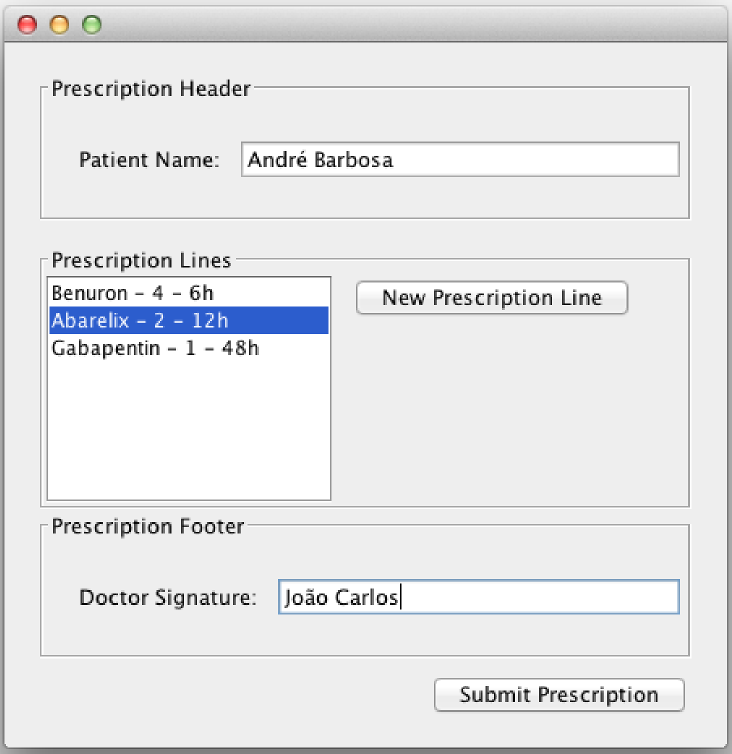


José Valim - Benuron
António Fonseca - Abarelix
Carlos Almeida - Gabapentin
João Da Costa - Gadofosveset
Horácio Cunha - Jakafi
Miguel Dos Santos - Macrobid
Joana Dias - Mafenide
Maria Silva - Mannitol Injection
Margarida Hipólito - Benuron
Tiago Costa - Ocriplasmin
Ana Sousa - Ofloxacin
Cleide Santos - Olopatadine

Figure 6.21: Prescriptions List.

The *Prescription List* screen is composed by a list of prescriptions and three buttons, *Edit Prescription*, *Delete Prescription* and *New Prescription*. Comparing this screen with the AUI model to list prescriptions, one can see that the List *CompoundIU* was mapped to the list aligned on the right of the screen and the three *SelectionIU* were mapped to the three buttons placed at the left of the list. Clicking the *Edit Prescription* or the *Delete Prescription* buttons without selecting a prescription from the list won't yield any result. The *Edit Prescription* button should launch a window compliant with the *Edit Prescription* AUI model passing the selected prescription as argument in order to be edited. The *DeletePrescription* button will delete the selected prescription from the list. Finally the *New Prescription* button will launch the window shown on Figure 6.22 to allow the user to create a new prescription.

The *New Prescription* screen has three *titled panels*, *Prescription Header*, *Prescription Lines* and *Prescription Footer*. By looking at the *New Prescription* AUI model, one can see three *CompoundIUs* in the main *CompoundIU* labelled *Prescription*. We can easily notice that these three *CompoundIU* were mapped into each of the three *titled panels*. Inside each panel, we can also note that each controller were mapped with the *DataIU*


Prescription Header

Patient Name: André Barbosa

Prescription Lines

Benuron - 4 - 6h
Abarelix - 2 - 12h
Gabapentin - 1 - 48h

Prescription Footer

Doctor Signature: João Carlos

Figure 6.22: New Prescription.

and *SelectionIU* elements present on the AUI model. Pressing the *New Prescription Line* button would launch a window with the form to create a new *PrescriptionLine* as specified in the AUI model. The *Submit Prescription* button corresponds to the *SelectionIU* labelled *SubmitPrescription* and, on click, should save the new prescription and close the *New Prescription* window.

Chapter 7

Conclusions and Future Work

The introductory chapter (chapter 1) of this dissertation started by stating the importance of Human Computer Interaction (HCI) on today's technological industry. Usability is often one of the most important aspects of a software product. And it's also true that nowadays HCI can come in many forms (or platforms). The proliferation of mobile equipments takes a huge impact on application's development. Different sizes, different resolutions and different input methods need different User Interface (UI)'s.

Model Driven Development of User Interfaces (MDDUI) was presented as a possible solution to help maintain consistency of UI's, boost their development time and even improve their quality. Models are easier to maintain and faster to build than code. And, if we're dealing with ubiquitous problems, models can also be platform independent. By using models it's easier to promote re-usability of previous knowledge on different projects.

The main goal of this project was to promote the reuse of previous knowledge in the form of patterns. The approach we choose was to create patterns at a conceptual level using USer Interface eXtensible Mark-up Language (UsiXML) models. We opted by this approach because:

1. models at a higher level of abstraction are more platform independent and thus are easier to reuse in different contexts;
2. in a Model Driven Development (MDD) approach, most decisions are made at a conceptual level;
3. lower level patterns tend to take concerns about design and other presentational

aspects that should not be a concern of an engineering method.

The way we choose to achieve this goal was by integrating into an established engineering method, developed by the UsiXML community, the Forward Engineering method (FEM). By doing this we could more easily specify what should be the inputs and outputs of the developing tool. As presented on section 3.2.2, the FEM starts with a domain and a task models that should originate an Abstract User Interface (AUI) model. This is a hard transformation to achieve and thus very susceptible to human errors and at same time critical for how the final UI will perform in terms of usability. The concept we designed acts on this stage. Section 4.2 defines a pattern as a set of these input and output models, among other descriptive attributes. We took this approach towards patterns because:

1. allows the development of a tool capable of understanding the patterns and apply them to developing models without human interference;
2. patterns can be easily produced using UsiXML modelling tools;
3. thanks to the set of descriptive attributes, patterns are still very accessible for humans even if their not part of a technical team and/or are not familiar with UsiXML.

Developers can benefit by using these patterns because they get the identification of abstract UI structure and the spatiotemporal arrangement of elements in the AUI for free. Meaning that using the information gathered from a pattern an automated tool can easily arrange the AUI elements in the right way as specified by the pattern. A tool can infer which elements are related and should stay close and in which order elements should be presented.

The biggest fragility with this approach relies on the selection of abstract UI elements. The fittest element for a task depends o the nature of the task. In the way UsiXML specifies tasks, based on Concur Task Trees (CTT) [18], it's easy for a human to understand what's the best AUI element for a given task but it's not well optimized for a machine to infer this information. Thus we specified a naming convention with support for common actions that are than mapped into AUI elements. This solution works for set of situations but it's not scalable. It doesn't support every type of possible actions and limits developers to use the English language in their models.

After designing this concept, we implemented a prototype that serves as prove of concept for the specified features. The prototype had to be compliant with UsiXML tools released in July, 2012. This raised a problem because this set of tools are not compliant with the version 1.8 of the UsiXML's specification. Which is the most recent version publicly available. Although the changes weren't too significant this prevented the UsiXML models and elements class representations to be generated automatically from the specification. For this reason, the prototype doesn't support every UsiXML's elements (although it supports most of them).

As future work, there are several improvements that could be made on the prototype:

- in order to match elements from the pattern models with the elements from the input models, input model's elements need a *pattern_id* property that indicates the corresponding element in the pattern. A matching algorithm that took leverage of the models hierarchy complemented with a *drag and drop* UI would be a more elegant solution;
- the prototype doesn't help a developer to find a suited pattern for the input models, only validates if the input models are in fact an instantiation of the pattern. A better user experience could be provided if the system were able to search for suited patterns in a given repository. A possible implementation would be through integration with a tool like IdealXML [13];
- the UsiXML language is still under development and constant change. Probably further work will necessary for the tool to be compliant with the next versions of UsiXML's tools.

Bibliography

[1] Alexander, C., Ishikawa, S., and Silverstein, M. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, New York, 1977.

[2] Balzert, H., Hofmann, F., Kruschinski, V., and Niemann, C. The janus application development environment - generating more than the user interface. In *Computer-Aided Design of User Interfaces I, Proceedings of the Second International Workshop on Computer-Aided Design of User Interfaces, June 5-7, 1996, Namur, Belgium* (1996), J. Vanderdonckt, Ed., Presses Universitaires de Namur, pp. 183–208.

[3] Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. A unifying reference framework for multi-target user interfaces. *INTERACTING WITH COMPUTERS 15* (2003), 289–308.

[4] da Silva, P. P., and Paton, N. W. User interface modeling in umli. *IEEE Softw. 20* (July 2003), 62–69.

[5] Erickson, T. The interaction design patterns page. `http://www.visi.com/~snowfall/InteractionPatterns.html`, Nov. 2011.

[6] Fogarty, J., and Hudson, S. E. Gadget: a toolkit for optimization-based approaches to interface and display generation. *ACM Trans. Graph. 23*, 3 (Aug. 2004), 730–730.

[7] Gajos, K. Z., Weld, D. S., and Wobbrock, J. O. Automatically generating personalized user interfaces with supple. *Artif. Intell. 174*, 12-13 (Aug. 2010), 910–950.

[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns.* Addison-Wesley, Boston, MA, 1995.

[9] Johnson, P., Johnson, H., and Wilson, S. Scenario-based design. John Wiley & Sons, Inc., New York, NY, USA, 1995, ch. Rapid prototyping of user interfaces driven by task models, pp. 209–246.

[10] Limbourg, Q., and Vanderdonckt, J. Multipath transformational development of user interfaces with graph transformations. In *Human-Centered Software Engineering*. 2009, pp. 107–138.

[11] Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and López-Jaquero, V. Usixml: a language supporting multi-path development of user interfaces. Springer-Verlag, pp. 11–13.

[12] Molina, P. J. A review to model-based user interface development technology. In *MBUI* (2004).

[13] Montero, F., Lozano, M., and González, P. Idealxml: an experience-based environment for user interface design and pattern manipulation. Tech. rep., 2005.

[14] Montero, F., and López-Jaquero, V. Idealxml: An interaction design tool. In *CADUI'06* (2006), pp. 245–252.

[15] Myers, B. A. User interface software tools. *ACM TRANSACTIONS ON COMPUTER-HUMAN INTERACTION 2* (1993), 64–103.

[16] Nunes, N. J., and e Cunha, J. F. Object modeling and user interface design. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001, ch. Wisdom - Whitewater interactive system development with object models, pp. 197–243.

[17] Object Management Group, Inc. *Software & Systems Process Engineering Metamodel Specification v2.0*, omg ed., Oct. 2007.

[18] Paternò, F., Mancini, C., and Meniconi, S. Concurtasktrees: A diagrammatic notation for specifying task models. In *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction* (London, UK, UK, 1997), INTERACT '97, Chapman & Hall, Ltd., pp. 362–369.

[19] Puerta, A. The mecano project: Comprehensive and integrated support for model-based interface development. In *In Computer-Aided Design of User Interfaces* (1996), Namur University Press, pp. 5–7.

[20] Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modeling Language Reference Manual.* Addison-Wesley Professional, Jan. 1999.

[21] Selic, B. The pragmatics of model-driven development. *IEEE Softw. 20* (September 2003), 19–25.

[22] Szekely, P. Retrospective and challenges for model-based interface development. In *Design, Specification and Verification of Interactive Systems '96* (1996), Springer-Verlag, pp. 1–27.

[23] Usixml Consortium. Usixml, user interface extensible markup language.

[24] Usixml Consortium. Usixml method specification.

[25] Vanderdonckt, J. Knowledge-based systems for automated user interface generation: the trident experience. Tech. rep., IN PROCEEDINGS OF THE CHI '95 WORKSHOP ON KNOWLEDGE-BASED SUPPORT FOR THE USER INTERFACE DESIGN PROCESS, 1995.

[26] Vanderdonckt, J. Readings in intelligent user interfaces. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998, ch. Automatic generation of a user interface for highly interactive business-oriented applications, pp. 516–520.

[27] Vanderdonckt, J., and Simarro, F. M. Generative pattern-based design of user interfaces. In *Proceedings of the 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems* (New York, NY, USA, 2010), PEICS '10, ACM, pp. 12–19.

[28] Wiecha, C., Bennett, W., Boies, S., Gould, J., and Greene, S. Its: a tool for rapidly developing interactive applications. *ACM Trans. Inf. Syst. 8*, 3 (July 1990), 204–236.

Appendix A

SPEM 2.0

Software & Systems Process Engineering Metamodel (SPEM) 2.0 is an Object Management Group (OMG) standard dedicated to software method modelling. This section is an overview of the main features and meta-models defined in SPEM 2.0. A complete specification can be found in [17].

The goal of SPEM is to propose minimal elements necessary to define any software and systems development method, without adding specific features to address particular domains. As a result, this meta-model supports a large range of development methods of different styles, levels of formalism, and life-cycle models.

SPEM is a Unified Modelling Language (UML) profile, meaning that SPEM reuses UML whenever possible. Consequently, SPEM uses UML for various software model concepts presentations.

The current version of SPEM is 2.0 and it was completely reformulated from the previous one in order to separate the operational aspect of a method from the temporal aspect of a method. As shown in figure A.1, the SPEM 2.0 meta-model uses seven main meta-model packages:

- **Method Content** package describes the static aspect of a method;
- **Process Structure** and **Process Behaviour** packages describe the dynamic aspect of a method, **Process With Methods** package describes the link between these two aspects;
- **Core package** provides the common classes that are used in the different packages;
- **Method Plug-in** package describes the configuration of a method and Managed

Content package describes the documentation of a method.

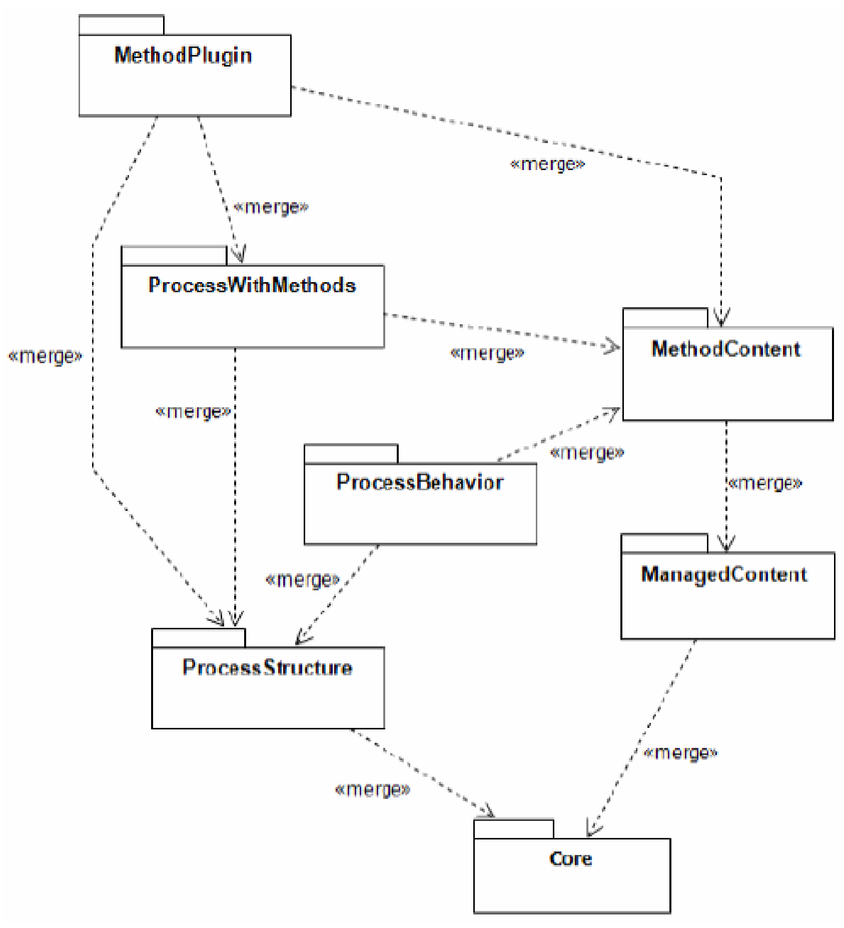


Figure A.1: Structure of the SPEM 2.0 meta-model.

In the following subsections we'll go into some detail for each package in order to get a better understanding of the SPEM 2.0 method meta-model.

Core package

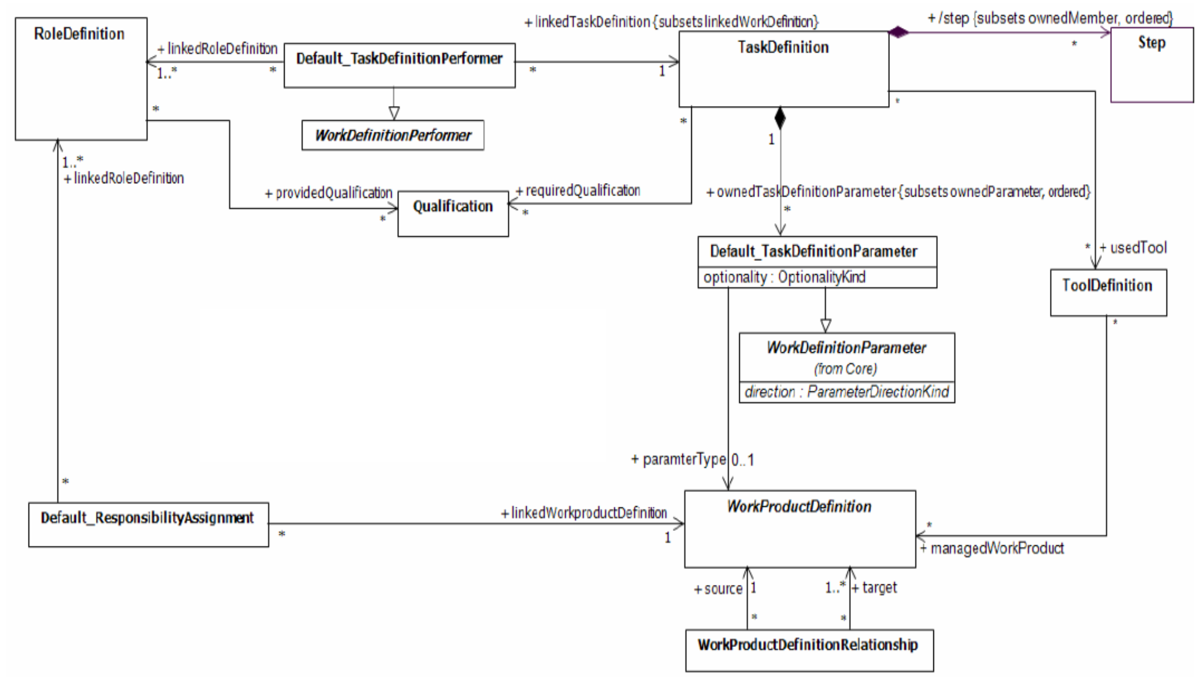
The Core meta-model package contains abstract generalization classes that are specialized in the other meta-model packages. These abstract generalization classes are used to define common properties of their specialized classes. The main elements of the core package are:

- The **Kind class**, that expresses a refined vocabulary specific to a method;
- **Work Definition** is an abstract generalization class that represents the work being performed by a specific role, or the work performed throughout a life-cycle.

- **Work Definition Parameter** is an abstract generalization class that represents parameters for Work Definitions.
- **Work Definition Performer** is an abstract generalization class that represents the relationship of a work performer (role) to a Work Definition.

Method content

The method content meta-model package defines the core elements of every method (producer, work unit and work product) independently of any process or development project. It describes the specific development steps that are achieved by which roles with which resources and results, without specifying the placement of these steps within a specific development life-cycle.



RoleDefinition
+linkedRoleDefinition
Default_TaskDefinitionPerformer
+linkedTaskDefinition {subsets linkedWorkDefinition}
TaskDefinition
+/step {subsets ownedMember, ordered}
Step
WorkDefinitionPerformer
+linkedRoleDefinition
+providedQualification
Qualification
+requiredQualification
+ownedTaskDefinitionParameter {subsets ownedParameter, ordered}
Default_TaskDefinitionParameter
optionality : OptionalityKind
+usedTool
ToolDefinition
WorkDefinitionParameter
(from Core)
direction : ParameterDirectionKind
+paramterType
Default_ResponsibilityAssignment
+linkedWorkproductDefinition
WorkProductDefinition
+managedWorkProduct
+source
+target
WorkProductDefinitionRelationship

Figure A.2: Structure of the SPEM 2.0 method content meta-model package.

Figure A.2 shows the Method Content meta-model package. The main classes of the Method content meta-model are:

- The **Task Definition** defines the work being performed by *Roles Definition instances*. A Task is associated with input and output *Work Products*.
- A **Step** describes a meaningful and consistent part of the overall work described for

a *Task Definition.* The collection of Steps defined for a *Task Definition* represents all the work that should be done to achieve the overall development goal of the *Task Definition.*

- The **Work Product Definition** describes the product which is used, modified, and produced by *Task Definitions.*
- The **Role Definition** designs a general reusable definition of an organizational role. It defines a set of related skills, competencies, and responsibilities of an individual or a set of individuals. *Roles* are used by *Task Definitions* to define who performs them as well as to define a set of Work *Product Definitions* they are responsible for.

Process Structure

The process structure meta-mode defines the structure of the method process. This package represents a process decomposed as a set of Activity classes that are linked to Role classes and Work Product classes. This structure is useful to express the fact that a life-cycle is composed by set of phases, and each phase is composed by set of activities.

Figure A.3 illustrates the Process Structure meta-model package. The most important classes of this meta-model are:

- A *WorkDefinition* (coming from the Core package) is performed by a **Work Definition Performer**, which is a role, and, through this role, by a *Process Performer.*
- A **Breakdown Element** is an abstract generalization class that defines a set of properties available to all of its specializations.
- A **Work Breakdown Element** provides specific properties for *Breakdown Elements* that represent work.
- An **Activity** defines basic units of work within a process as well as a process itself. Every activity can represent a process in SPEM 2.0. It relates to *Work Product Use* instances via instances of the *Process Parameter* class and *Role Use* instances via *Process Performer* instances. An activity can be used by another activity, so that the structure of the source activity is copied into the target activity.
- The **Role Use** represents a performer of an *Activity* or a participant of the *Activity.* A *Role Use* is only specific to the context of an *Activity.* It is not a general reusable definition of an organizational role like the *Role Definition* of the Method Content

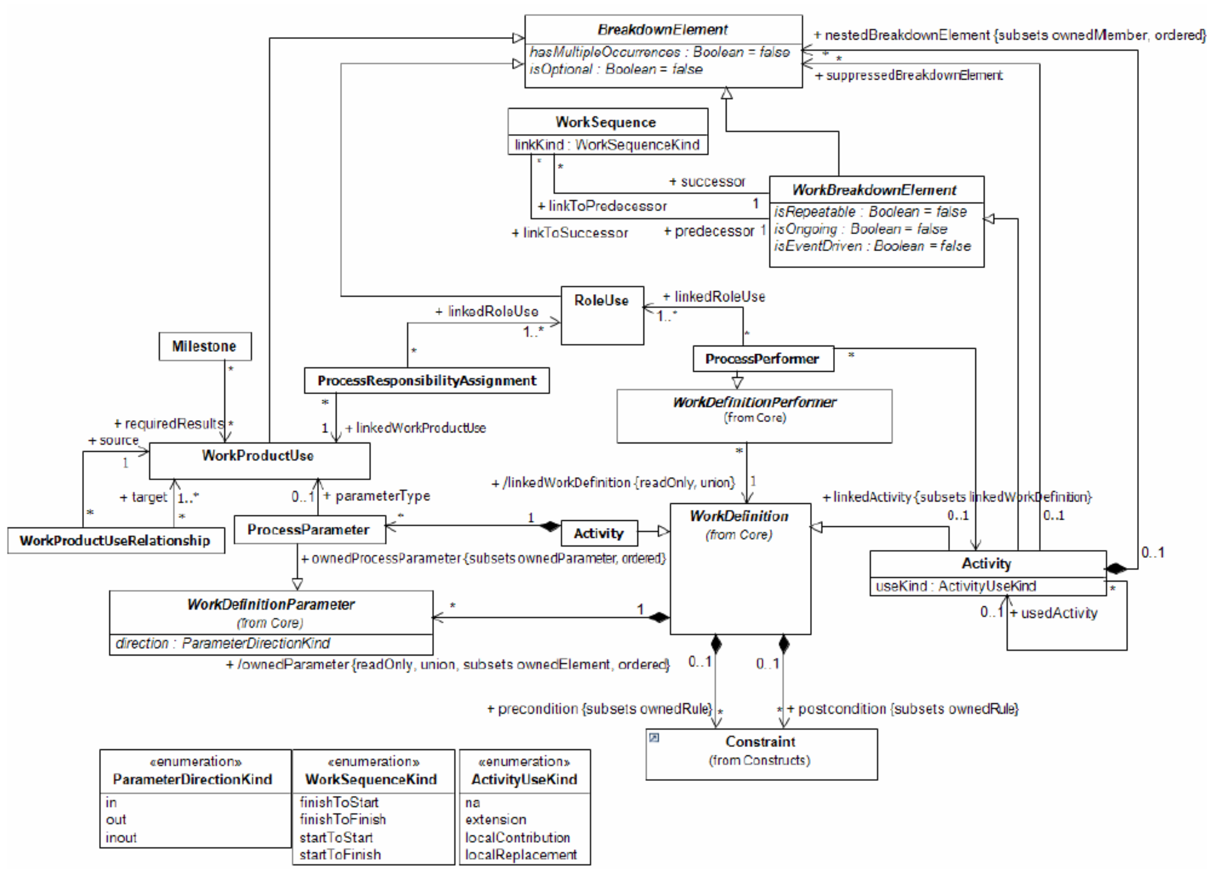

BreakdownElement
hasMultipleOccurrences : Boolean = false
isOptional : Boolean = false
+ nestedBreakdownElement {subsets ownedMember, ordered}
+ suppressedBreakdownElement
WorkSequence
linkKind : WorkSequenceKind
+ successor
+ linkToPredecessor
+ linkToSuccessor
+ predecessor
WorkBreakdownElement
isRepeatable : Boolean = false
isOngoing : Boolean = false
isEventDriven : Boolean = false
RoleUse
+ linkedRoleUse
Milestone
ProcessResponsibilityAssignment
ProcessPerformer
WorkDefinitionPerformer
(from Core)
+ requiredResults
+ source
WorkProductUse
+ linkedWorkProductUse
+ target
+ parameterType
WorkProductUseRelationship
ProcessParameter
Activity
+ /linkedWorkDefinition {readOnly, union}
WorkDefinition
(from Core)
+ linkedActivity {subsets linkedWorkDefinition}
Activity
useKind : ActivityUseKind
+ usedActivity
+ ownedProcessParameter {subsets ownedParameter, ordered}
WorkDefinitionParameter
(from Core)
direction : ParameterDirectionKind
+ /ownedParameter {readOnly, union, subsets ownedElement, ordered}
+ precondition {subsets ownedRule}
+ postcondition {subsets ownedRule}
Constraint
(from Constructs)
«enumeration»
ParameterDirectionKind
in
out
inout
«enumeration»
WorkSequenceKind
finishToStart
finishToFinish
startToStart
startToFinish
«enumeration»
ActivityUseKind
na
extension
localContribution
localReplacement

Figure A.3: Main classes and associations of SPEM 2.0 process structure package package.

package.

- A **Work Product Use** represents an input and/or output type for an *Activity* or represents a general participant of the *Activity*.

- The **Work Sequence** represents a relationship between two *Work Breakdown Elements* in which one *Work Breakdown Element* depends on the start or finish of another *Work Breakdown Elements* in order to begin or end.

The **Process Behaviour** meta-model package allows extending these process structures with behavioural models. However, it does not introduce its own formalism for behaviour models, but instead provides *links* to existing externally-defined behaviour models, enabling reuse of these approaches from other OMG or third party specifications.

Process With Methods

SPEM 2.0 separates reusable core method contents (expressed using the *Method Content* meta-model) from its temporal aspect (expressed using the *Process Structure* meta-model).

The Process With Methods meta-model package allows the integration of the process definition with instances of the core method content elements. This integration allows the specification of how and which method elements will be applied in a particular state of the process.

For example, a *Task Definition* can be invoked many times throughout a development process. Each invocation is defined with an individual element of the *Process With Methods* meta-model which is called *Task Use.* The latter manages the *Task Definition* invocation by changing for example the roles involved in performing the task or an omission of specific work product input types. A *Task Use* represents a binding for a *Task Definition* in the context of one specific *Activity.* Therefore, one *Task Definition* can be represented by many *Task Uses*, each within the context of an *Activity* with its own set of relationships.

The SPEM 2.0 specification also contains a **Managed Content** and **Method Plug-in** meta-models. The first one introduces concepts for managing the textual documentation of a method while de latter defines concepts for designing and managing repositories of method content and processes.