University of Porto
Faculty of Engineering

Introducing End-User Reconfiguration on Clinical Knowledge Information Systems

MSc Dissertation

André Tiago Magalhães do Carmo

July 2011

Scientific Supervision by

Hugo Ferreira, PhD, Assistant Lecturer
Ademar Aguiar, PhD, Assistant Professor
Department of Informatics Engineering

University of Porto
Faculty of Engineering

Introducing End-User Reconfiguration on Clinical Knowledge Information Systems

André Tiago Magalhães do Carmo

July 2011

Approved in oral examination by the committee

Chair: Ana Paiva, PhD
External Examiner: Feliz Ribeiro Gouveia, PhD
Supervisor: Hugo Sereno Ferreira, PhD

Supervisor: Ademar Aguiar, PhD

Contact Information:

André Tiago Magalhães do Carmo
Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Informática

Rua Dr. Roberto Frias, s/n
4200-465 Porto
Portugal
Tel.: +351 22 508 1400
Fax.: +351 22 508 1440

Email: andre.carmo@fe.up.pt
URL: http://andrecarmo.net

André Tiago Magalhães do Carmo
"Introducing End-User Reconfiguration on Clinical Knowledge Information Systems"

Areas (based on Association for Computing Machinery's Computing Classification System). D.1.7 Visual Programming, D.2.2 Design Tools and Techniques (User Interfaces), D.2.6 Programming Environments, D.2.11 Software Architectures, D.2.12 Interoperability.
Keywords. adaptive user interfaces, adaptability, architectures, clinical knowledge, incompleteness in software design, health systems, human-computer interaction, metamodelling, openehr, personalization, usability, zk.

This dissertation, "Introducing End-User Reconfiguration on Clinical Knowledge Information Systems", was proposed and funded by *iTGrow - Software e Sistemas, ACE*, which is owned by both *Critical Software, SA* and *Banco BPI*.

Acknowledgements

I am thankful to Hugo Ferreira for the excellent supervision of my work. A dissertation in an industrial environment is, sometimes, very difficult to be supervised. Thank you for your availability whenever I needed.

I am also thankful to Nuno Monteiro, my facilitator from Critical Software, who supervised me in my practical work, and who complied, as well as Catarina Fonseca, the General Manager of iTGrow, with the scientific and academic purpose of this dissertation. To my co-workers in iTGrow and in Critical Software, thank you for helping me whenever I needed.

My father Samuel, my mother Paula and my sister Ana, gave me strength in all situations, and it was very important. Thank you for all the hard work which allowed me to study and to reach this MSc dissertation.

Finally, a special thanks to my girlfriend, Sara, who has been there for me all the time, not only during this dissertation, but also during the my entire Integrated Master in Informatics and Computing Engineering program. Thank you for all you have done for me, and I hope I can do at least the same for you.

"Listen to me now
I need to let you know
You don't have to go it alone"
U2 - Sometimes You Can't Make It On Your Own

This page was intentionally left mostly blank.

Abstract

Change in software requirements is often a problem, as the environment is not static. Methodologies are evolving to support *change*, but engineers should also design to *embrace change*: in other words, besides the ability to respond to change given by these methodologies, systems should also be designed to deal with *change* and *incompleteness* of requirements.

In health area, *incompleteness* is, in fact, present. The domain is sufficiently complicated to be modeled, as there are a great variety of specific needs. There are different health professionals, pathologies and patients. Which content should be presented? How the system should behave? And it is impossible to predict all the situations. *Critical Software, SA*, which proposed this dissertation, had this problem in one of its health related projects. In this dissertation the author studied how to solve this problem, adding some useful functionalities to the current project. That project, and consequently the *Template Builder*, uses two main technologies: *open*EHR, for clinical content, and ZK framework for the UI development.

*Open*EHR is a standard to represent and describe clinical concepts, particularly by using two main artifacts: *archetypes*, which are computable descriptions of some clinical knowledge, and *templates*, which are specialized definitions of clinical content related to more specific situations, and they often correspond to Graphical User Interfaces. These templates are composed by a group of archetypes, with some of its properties being redefined.

The main goal of this dissertation is to address the *incomplete by design* property in health care systems, particularly when using *open*EHR templates. It is, in fact, to create a software component that allows the end-user to build *templates*, that correspond to *screens* for specific purposes, based on *open*EHR *archetypes*, that can be performed by end-users with clinical and *open*EHR knowledge.

Other main goals emerged from the implementation of this solution, like the necessity to link UI elements with the model, or the need of having a set of rules to define some system behavior.

Resumo

A *mudança* dos requisitos é um problema no desenvolvimento de software. Existe uma dificuldade natural em elicitar os requisitos, o que muitas vezes acontece devido à dificuldade que os *clientes* têm em passar a mensagem. Devido a isso, as metodologias de desenvolvimento estão a adaptar-se para responder à mudança. Contudo, é também preciso que os engenheiros adaptem o desenhor do software para este ser lidar com a mudança.

Na área da saúde, o software é constantemente *incompleto*. O domínio que se quer modelar é vasto, e imprevisível. Existem diferentes profissionais de saúde, patologias e pacientes. Estas diferenças de necessidades causam dificuldades ao nível do desenho da interface com o utilizador, e em relação à forma como o sistema deve reagir em determinadas situações. Que conteúdo apresentar? Como o apresentar? Como deve o sistema reagir?

A *Critical Software, SA*, a empresa que propôs esta dissertação, teve este problema num dos seus projectos relacionados com a área da saúde. Esta dissertação teve como objective estudar este problema de *ser incompleto*, completando o actual projecto com novas funcionalidades. O projecto da Critical Software, e consequentemente este *Template Builder*, usa duas tecnologias importantes: o *open*EHR para descrever informação clínica, e a framework ZK para o desenvolvimento da interface com o utilizador. O *open*EHR é um standard para representar informação clínica, usando principalmente dois artefactos: os arquétipos, que permitem descrever conceitos clínicos gerais, e os templates, que permitem descrever conceitos clínicos mais específicos baseados noutros arquétipos. Geralmente estes templates correspondem a interfaces com o utilizador.

O grande objectivo desta dissertação é resolver o problema do software na área da saúde ser incompleto, especialmente quando se está a falar de software baseado no *open*EHR. O que se quer é, de facto, um componente que permita que qualquer utilizador ligado à área da saúde e/ou ao *open*EHR construa os seus próprios *templates*, que na realidade correspondem a novas interfaces com o utilizador.

Outros objectivos surgiram durante a implementação da solução, tal como a necessidade de arranjar uma forma de ligar os elementos da UI com o modelo de dados, ou a necessidade da definição de um conjunto de regras para regular o comportamento do sistema em algumas situações pontuais.

Contents

Abstract i

Resumo iii

1 Introduction 1
- 1.1 Incomplete by Design . . . 2
- 1.2 Design for Incompleteness . . . 3
- 1.3 Incompleteness in Health Care Systems . . . 4
- 1.4 Designing software . . . 4
- 1.5 Graphical User Interfaces in Health Systems . . . 5
- 1.6 Research Challenges and Goals . . . 6
- 1.7 Research Stance . . . 8
- 1.8 How to read this Dissertation . . . 8

2 openEHR 11
- 2.1 Fundamentals . . . 11
- 2.2 Package Structure . . . 13
- 2.3 Archetypes . . . 14
  - 2.3.1 Types of Archetypes . . . 15
  - 2.3.2 Node Types . . . 18
- 2.4 Templates . . . 20
  - 2.4.1 Template's Structure . . . 20
  - 2.4.2 How Templates specialize Archetypes . . . 22
- 2.5 OpenEHR Java Implementation . . . 23
- 2.6 Conclusion . . . 24

3 ZK Framework 25
- 3.1 Fundamentals . . . 25
- 3.2 Architecture . . . 26
- 3.3 Example of usage . . . 26

3.4 Conclusion . . . 28

4 Adaptive Systems **29**
4.1 Overview . . . 29
4.2 Adaptive User Interfaces . . . 30
4.3 Adaptive Health Systems . . . 31
4.4 Examples of the Implementation of AUIs . . . 32
4.5 Conclusion . . . 34

5 Research Problem and Methodology **37**
5.1 Fundamental Challenges and Goals . . . 38
5.1.1 Content Presentation . . . 38
5.1.2 openEHR and ZK connection . . . 39
5.1.3 openEHR incompleteness . . . 39
5.2 Research Methodology . . . 39
5.3 Validation Methodology . . . 41
5.4 Conclusion . . . 42

6 Solution **43**
6.1 Components . . . 44
6.2 Architecture Overview . . . 45
6.3 Architecture in Detail . . . 46
6.3.1 Model . . . 46
6.3.2 View and Controller . . . 48
6.4 Solution Achieved . . . 49
6.4.1 Overview . . . 50
6.4.2 Building a Template by Adding Archetypes . . . 52
6.4.3 (Re)configuring a Template by Setting Rules . . . 57
6.4.4 Other Operations . . . 62
6.4.5 Previewing the GUI that Corresponds to a Template . . . 62
6.5 How are ZK and openEHR Linked? . . . 64
6.6 Reconfigure the system . . . 67
6.7 Conclusion . . . 72

7 Conclusions and Future Work **75**
7.1 Main Results . . . 76
7.2 Problems occurred during research . . . 76
7.3 Lessons Learned . . . 77
7.4 Future Work . . . 78

7.5 Epilogue 79

Nomenclature **81**

References **83**

List of Figures

1.1 Waterfall software development process . . . 2
1.2 Incremental-based software development process . . . 3
1.3 Some clinical concepts, and two ways of combining them . . . 6

2.1 The *open*EHR Specification Project . . . 12
2.2 *Open*EHR deployed in a shared-care context . . . 13
2.3 *Open*EHR's RM, AM and SM package structure . . . 14
2.4 Building Archetypes from Reference Model's components . . . 15
2.5 Entry types: this RM type define the semantics of the clinical knowledge . . . 16
2.6 ITEM_STRUCTURE RM type . . . 17
2.7 A representation of an example of a ITEM_TREE archetype . . . 18
2.8 *constrain_model* with archetype nodes . . . 19
2.9 How *templates* are a group of *archetypes*, and how they apply to data . . . 21
2.10 Template structure represented as an UML class diagram . . . 21
2.11 Relationships between different kinds of artifacts of *open*EHR . . . 22

3.1 ZK client-server architecture . . . 26
3.2 Example of a GUI developed using ZK . . . 27
3.3 The application triggers an event when the user clicks "OK" . . . 27

4.1 How to choose *Archetypes* in *Template Designer* . . . 33
4.2 Customized Health Record in *Bento* . . . 34
4.3 Optional fields that can be dragged into the main window in *Bento* . . . 34

5.1 The Research Methodology followed . . . 40
5.2 Scrum process . . . 41

6.1 The MVC architecture . . . 45
6.2 MVC Architecture of *Template Builder* with components identified . . . 46
6.3 Model layer detailed in a UML class diagram . . . 47
6.4 View layer of *Template Builder* detailed . . . 48

6.5 Controller of *Template Builder* detailed . . . 49
6.6 Main window of *Template Builder* with two work areas . . . 50
6.7 How to search for archetypes . . . 51
6.8 Filtering archetypes for the selected *archetype slot* . . . 51
6.9 The result of the filter for the selected *archetype-slot* . . . 51
6.10 The template's work area contains tabs for multiple template building . . . 52
6.11 Work area for the definition of template properties . . . 53
6.12 The user tries to add a SECTION inside a COMPOSITION . . . 54
6.13 The operation shown in the previous figure succeeds . . . 54
6.14 The operation fails . . . 55
6.15 A virtual *Archetype Slot* is created after an *add operation* . . . 56
6.16 The UML sequence diagram represents the *add archetype* operation . . . 57
6.17 Some operations are made in a template . . . 58
6.18 Some operations are made in a template . . . 59
6.19 Some nodes allow the end-user to choose its data type within the template 59
6.20 Configurations that can be done in an archetype root node . . . 60
6.21 How properties are get . . . 62
6.22 An archetype was deleted in *Checkup Routine* template . . . 63
6.23 Preview (collapsed) . . . 63
6.24 Preview of the *TemplateDissertation* . . . 63
6.25 Each tab has a unique "ID", which is the same as the correspondent template 64
6.26 Instance of the template . . . 64
6.27 Path of an element in ZK . . . 65
6.28 Example of a template . . . 65
6.29 Example of a template after adding . . . 66
6.30 The problem of having two archetypes with the same name . . . 67
6.31 How "pathSlotCounter" allows to distinguish from archetypes . . . 67
6.32 Differences between the specification and the implementation of templates 68
6.33 Some differences between different XML rule's files . . . 69
6.34 Some differences between different XML rule's files . . . 70
6.35 Metamodel UML class diagram . . . 71
6.36 How the metamodel checks the template instance with rules instance . . . 73

List of Source Codes

2.1 Example of an OET Template, a Template with referenced archetypes. It is possible to observe its structure, and the archetypes which make that structure, according to what was explained in Archetype Types and Template Structure sections. It is also possible to observe some *Rules* created, to override and specialize the original archetype's properties. . . . 23

3.1 Example of a simple ZUML interface with some components. 27

3.2 Example of controller written in Java to trigger an event when clicking on a button on source 3.1. 28

6.1 Example of the template properties in the OET file, according to what is defined in the UI by the end-user. 53

6.2 openEHR-EHR-COMPOSITION.problem_list_csw.v1 *archetype slot* representation in the archetype's ADL file. 55

6.3 Template OET file after the last made operation. 56

6.4 Template OET file after the last made operation. 60

6.5 Template OET file after the last made operation. 61

6.6 Template OET file. In bold there is the *path* of the archetype added, which is the same as the path of *SECTION*s *archetype slot*. 66

6.7 Portion of a XML file with rules related to template's structure. 70

6.8 XML rules file general structure. It contains three main sections: (1) one for the definition of terms; (2) other for the definition of terms that inherit other terms previously defined; (3) and a section for the template structure definition. 74

Chapter 1

Introduction

1.1 Incomplete by Design 2
1.2 Design for Incompleteness 3
1.3 Incompleteness in Health Care Systems 4
1.4 Designing software 4
1.5 Graphical User Interfaces in Health Systems 5
1.6 Research Challenges and Goals 6
1.7 Research Stance 8
1.8 How to read this Dissertation 8

Software engineers often have to solve puzzles in their day-by-day. One of the most important factors for these puzzles is the continuous *change* in software requirements, which is one major factor for project failures and overruns [EM03]. Software engineers have to consider that the environment is not static, and that it can actually be so dynamic that that there is no time to keep up with change. And besides *change*, there are even more variables like, cost, schedule and risk [EM03, Fer10, Som04].

These variables have to be considered for the software development process, and that changed the way software is built. It had to change in the past years or decades, and some software development methodologies did not adapt themselves to the world of *change*. Some have disappeared, others are less used nowadays. Some software development processes, like agile methodologies, are intentionally designed to *embrace change*, instead of following a strict plan until the end of the development [ea].

Software requirements, or stakeholders needs, change and evolve over time. In some cases this is because of the nature of the business, which is what happens in health area, where there are many specific needs that are very difficult to predict. There are several subareas, each one with specific requirements. Different health professionals and different

pathologies have different and specific needs, and there are no two equal patients, which increases the difficulty to plan and develop information systems to health area. Moreover, all health processes need to be covered by the software [PS01]. And there is even the possibility that the domain can change or evolve. What if we accept that software is incomplete by nature? What if we accept that the software can't cover all user needs, either at a business logic and Graphical User Interface level?

1.1 Incomplete by Design

As said before, it is difficult for the software to cover the entire domain, in some situations. But it is not only the changing environment that causes these difficulties: there is a natural difficulty while gathering, understanding and formalizing requirements, in most cases because the stakeholders don't really know what they want, or because they can't correctly pass the message of what they really want. Sometimes, even different stakeholders have conflicting requirements [Som04].

In classic development models, there are very defined phases, which must be completed before the next one starts [Som04]. This is shown in Figure 1.1

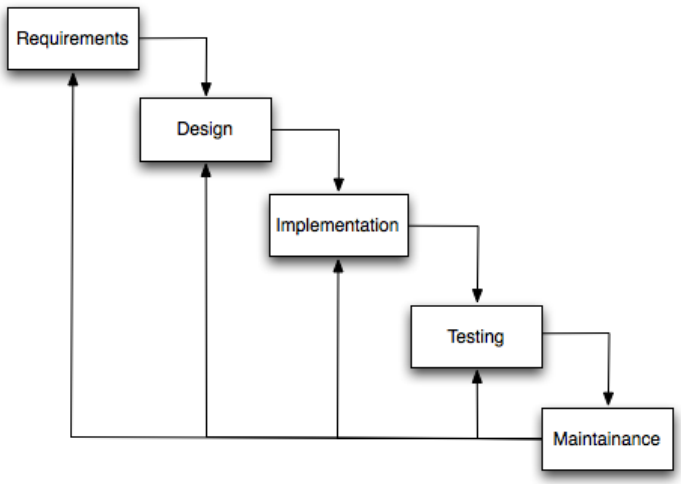

Figure 1.1: Waterfall software development process, based on [Som04].

They are not very flexible, because there is a strict plan, defined at the beginning, which must be strictly followed during the development (even though there is some feedback during the process). Because of this lack of flexibility, a dynamic environment is difficult to be modeled once the requirements elicitation phase is over. They can be effective for stable environments and for stable stakeholders preferences [GJT07].

Nowadays, some software development methodologies have been developed to overcome this difficulty, and to embrace change. These methodologies often use incremental and iterative approach because it is an approach that makes it easier to evaluate what has been done, and to change if it is necessary.

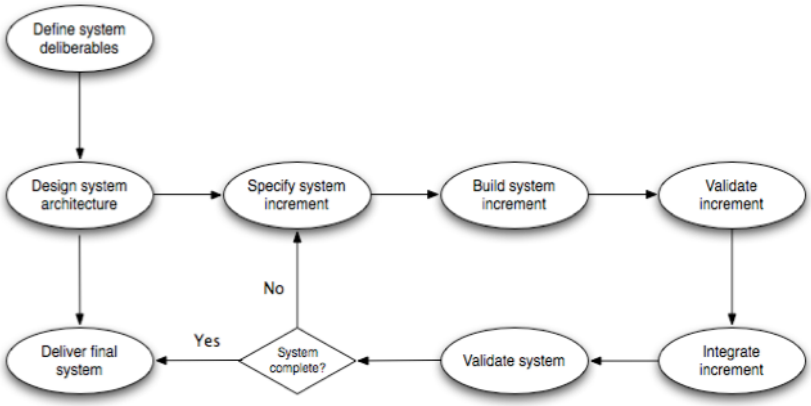

Figure 1.2: Incremental-based software development process, based on [Som04].

Agile methodologies use an incremental delivery approach, as explained in Figure 1.2. They are a good example on how meet customer satisfaction as they respond to change instead of strictly following a plan, even though there is much discussion about this way of developing software. One way to do this is to integrate the customer into the development team so that it is easier and faster to respond to changing requirements [ea]. Actually these methodologies accept that the domain can evolve, and that this is part a common software development.

However, besides this flexibility, agile methodologies may not be suitable for all types of systems [PA02]. Some people think that agile approaches should not be used in critical or safety softwares. Because documentation is not important to agile methodologies [ea], it could not be suitable for projects that require well defined documentation.

The majority of projects have a changing environment, and that must be accepted. The software needs to be adapted to the evolved domain, and it is called *incomplete by design* software.

1.2 Design for Incompleteness

Agile methodologies accept *incompleteness*, having the ability to adjust both the team and the development process. However, they still seam to have difficulties regarding to software artifacts and the way they are designed [Fer10]. Agile methods are based on iterative and incremental approach, in which the process is run several times, integrating

new features or refactoring in each iteration [ea]. But this is done almost as if the iteration would be the last one, which generally is not true.

If the domain is changing or evolving, and it is impossible to predict all the changes, how can we deal with that? If the software is accepted as *incomplete by design*, it should be *designed for incompleteness*. It is not only being able to respond to change, but also to design to *embrace* change [GJT07]. Designing and developing should be take together, and an agile methodology should also expedite software design. However, this *designing for incompleteness* is not exclusive for agile methodologies: other methods should also think about building adaptable software, which can be adapted to new situations.

1.3 Incompleteness in Health Care Systems

Specifically in health area, the domain is far more complicated to be met by the software, because of all those specific needs. It is impossible to predict them. Each health professional have specific needs, as there are experts in different health subareas. And even professionals from the same subarea can have different needs, as they have their own methodology to work and to treat their patients. Moreover, each patient can require specific requirements, as each pathology is different and specific from patient to patient. Considering all these different areas, health professionals, and even different patients, it is impossible to predict all these needs, even though the requirements are well defined. And this is a major problem while designing GUIs. Which information needs to be displayed? How it should be displayed?

If we don't know all the domain, or if we can't predict these specific needs, the system must be *designed for incompleteness*, as explained in § 1.2 (p. 3). And *designing for incompleteness*, in health area, means designing the system to deal with this unknown domain. This can include defining rules outside the system's implementation, or even designing the GUI to be changed based on some specific needs.

1.4 Designing software

Software design is the act of solving and planning a solution for a software problem (adapted from Webster's dictionary [web]). It can also be seen as a deeper thing, like planning software coding, anticipate bottlenecks, anticipate change and evolution, and so on [iSeS]. But what is a good software design? How can we evaluate that?

Some authors propose some software design principles and guidelines, like *modularity* and *abstraction* [RGI75], *strong cohesion and loose coupling* [iSeS], *encapsulation*, or even the usage of *design and architectural patterns* [iSeS, BMR+96]. There are even

some *properties* that allow to evaluate the quality of an architecture, like *reusability*, *maintainability*, runtime qualities like *availability*, *performance*, and others.

However, as said in § 1.2 (p. 3), it seems that there is little attention to designing software to be incomplete, and to deal with that incompleteness. In health area, but not exclusively, some business rules may be difficult to predict, and hardcode them may not be a good idea. So, how it is possible to *design for incompleteness* related to business logic? One way to do so is to use a meta-architectural pattern, in which business logic is stored outside the code. It is stored in a file or a database [RY02]. Instead of putting all the business logic outside the application, we may want to define several rules to determine some aspects of behavior of the application. For example, we may want the application to allow a minimum of information from a patient, but we do not want to hardcode this *rule*. Here is where *design for incompleteness* enter, with the use of XML and metamodels to define rules [HV01].

Of course this is may be difficult to do in the day-by-day, as there are several forces pushing and pulling in the software development process. These forces include, as said, time, cost, quality, scope, and even risks and customer satisfaction[EM03, Fer10, Som04]. Another *problem* may be the team's experience and ability to design such an architecture or such a component to be prepared for incompleteness.

1.5 Graphical User Interfaces in Health Systems

The known domain, or the known needs, leads to incomplete software, as explained in § 1.1 (p. 2) and § 1.3 (p. 4). This incompleteness may be a problem for different layers of the software: we can have incompleteness in the main architecture, but we can also have a user interface incompleteness. And this dissertation is more focussed on user interfaces incompleteness, and on how it is possible to deal with that: how we can *design the system for incompleteness*, in terms of GUIs? Let's see an example on how it is extremely difficult to predict specific needs.

Critical Software, SA, has a problem, related to its existing related project, which can be explained in Figure 1.3 (p. 6). As said before, in health area there are several different areas, health professionals, and even patients, and they require different needs. Health professionals and patients are, in fact, different stakeholders of the system, contributing to requirements elicitation.

In the center of Figure 1.3 (p. 6) we can see seven clinical concepts, as *open*EHR *archetypes* (see Chapter 2, p. 11). In the left side, they are grouped so that they are suitable for a routine checkup, as a *template*. In the right side, they are group to fit for a blood specialist.

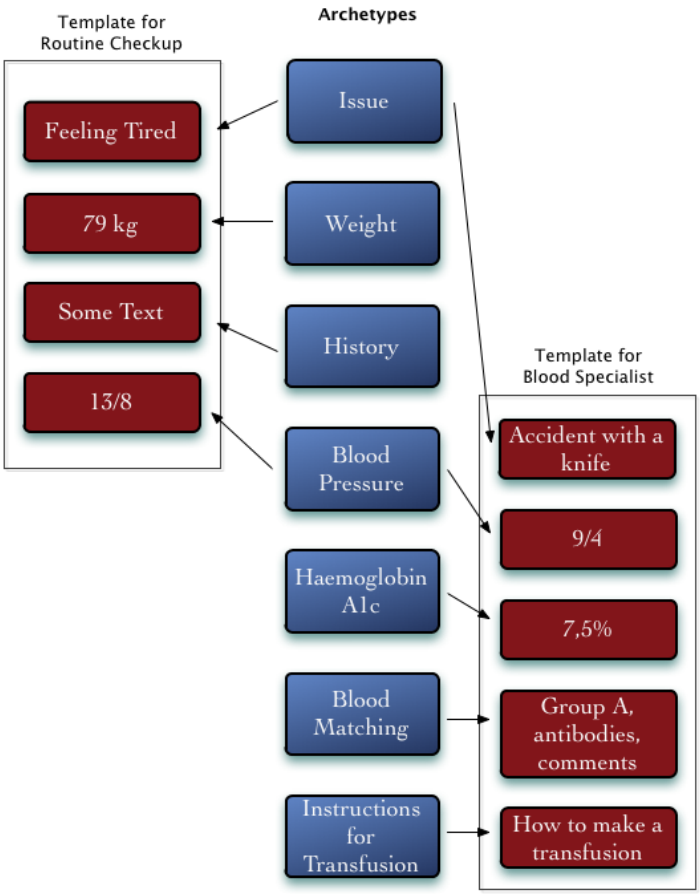

Figure 1.3: Some clinical concepts, and two ways of combining them for different needs.

But there are not just these two ways of grouping these concepts. Actually, combining one or more of these concepts, we get 127 possibilities. And if order matters, it is possible to get 13699 permutations of one or more of these, with just only seven fields. Of course not all of these 13699 possibilities may be used in real life, but we can see that it is impossible to predict all the needs, and to make software to meet all the domain. The domain is known, but it can be so big that it is impossible to cover it all. It is not possible to design user interfaces for all these requirements, and it is not possible continuously with the health professional to adapt the system.

1.6 Research Challenges and Goals

Given the motivation of this dissertation, related to *incomplete by design* and *design for incompleteness*, particular related to health information systems, this section briefly presents the main research goals. The dissertation aims to contribute to the knowledge of a specific domain of software engineering, related to adaptable systems in health care, but

specifically with *open*EHR standard (see Chapter 2, p. 11). This *incompleteness* in health care systems is related to both business logic and GUIs. The study was done by doing a specific *case study* in an organizational context, with all the constraints and benefits that it brings. The *case study* has the following properties:

- *Critical Software SA* is developing an information system for health care. This system is based on *open*EHR standard to describe and represent clinical information [ope07c], and ZK for rich Graphical User Interfaces.
- They have identified that they had needed some research related to develop which could allow to build *open*EHR templates (see Chapter 2, p. 11);
- The component developed should integrate with Critical's system: it should use some components and technologies (including *open*EHR and ZK), and it should use the same visual design.

These properties were studied, which raised some assumptions.

- Although health area is quite stable in terms of requirements, the system can be widely used, having a high degree of variability - the need of usage of the same software in different contexts [vGBS01];
- *Open*EHR specification takes the enormous possibilities in health area in account. This means that it is flexible enough for the creation of, for instance, new GUI's (named *templates*), based on some more general artifacts (named *archetypes*);
- *Open*EHR official Java implementation is yet to be completely implemented, even though its specification is stable;
- ZK framework version used is not the most recent. In this version some bugs have not been resolved, and workarounds may be needed to overcome specific bugs that were only resolved in future versions.

After the assumptions were studied, this dissertation's fundamental goals were established, and they are presented bellow.

- **The main goal is to develop a *Template Builder* that allows to build *open*EHR templates from archetypes, to cover health specific needs. These templates must specify some archetype's attributes.**
- **These templates must correspond to UI screen, so there is the necessity of previewing the template.**

- **The *Template Builder* must use some ui patterns and design principles so that every health professional can build his own templates/screens.**
- **In terms of *open*EHR and ZK, there must be a way to link the UI elements with the model.**
- **And finally, a minor goals is to develop a way to deal with business logic instability, allowing the system's behavior to be modified in some specific situations.**

1.7 Research Stance

Building software is not easy. In fact, it is complex. Classifying software development as being an engineering is, sometimes, a difficult discussion. One argument for this discussion is that in software engineering things evolve and change so fast [Kru04]. In fact, things evolve, which forces us not be focused only on the practice or tools. It is necessary to go beyond that. And this happens because the technology evolves so fast, but also because it may not be sufficiently mature.

However, Software Engineering inherits some aspects from other areas, and the *the value of generated knowledge is directly dependent on the methods by which it was obtained* [Fer10]. In fact, it is necessary to organize thoughts in theories, in order to obtain knowledge [Shu]. And that is a bit of what is science: thoughts organized into theories, providing an explanation of relationships between entities or *things*.

This dissertation studies a *specific* case, for which the author studies and applies knowledge. The methods chosen were those that seemed more appropriate to the situation and according to some limitations found during this research, and according to the fact that this dissertation was done in an organizational context. Those limitations and this enterprise context aligned this dissertation to a more practical and pragmatic way. In other words, the validation of what was done was pragmatic, based on industrial software validation and satisfaction, instead of other research evaluation techniques.

However, this does not mean that the scope ended where practice ended. The author went beyond practical assets, studying the concepts and the best ways to apply them, even though not all of them could be implemented.

1.8 How to read this Dissertation

This dissertation represents an overview of what was studied and implemented by the author. It is logically structured in a way that is easier to read, composed by three main

parts, which are presented next.

Part 1: Motivation & Fundamental Background. The first part gives an overview of the motivation, giving also a background of the fundamental problem behind this dissertation. It is composed by Chapter 1 (p. 1), "Introduction" which gives an overview about how *change* should be *embraced*, and how this is a key point to this dissertation.

Part 2: State of the Art & Specific Problem. This part gives an overview of the technologies used, allowing to go deeper inside the problem to a more concrete and specific question:

- Chapter 2 (p. 11), "OpenEHR", details *open*EHR, specially its main artifacts like archetypes and templates, and how they are related. Understanding these concepts is very important to understand the goals of this dissertation.
- Chapter 3 (p. 25), "ZK Framework", explains this framework for rich graphical user interfaces, and how it can be used.
- Chapter 4 (p. 29), "Adaptive Systems", goes deeper inside the fundamental problem that was presented. The concepts of Adaptive Systems and Adaptive User Interfaces are presented, and it is explained how this types of systems can solve the *incompleteness* problem, particularly in health information systems. Two examples are also shown.

Part 3: Solution & Conclusions. The final part presents the research problem and methodology, as well as the solution achieved and some conclusions about the work done:

- Chapter 5 (p. 37), "Research Problem and Methodology", explains the main goals of this project after all the background and problem were presented. It details the methodology used to achieve those goals, as well as some difficulties experienced during this research.
- Chapter 6 (p. 43), "Solution", details the implementation of the solution. An overview is first given about its architecture, and then the solution is detailed.
- Chapter 7 (p. 75), "Conclusions and Future Work", represents a retrospective of what was done, giving the conclusions of the work done, as well as future work that can be done.

The document should be read in the order it is presented for better understanding. However, those with knowledge about *open*EHR and/or ZK framework may skip those chapters for a quick reading.

Some typographical conventions are used in order to improve the understanding of this document. Acronyms appear in ALLCAPS. Relevant concepts appear in *italics*. Sometimes, to better point out some concepts, these are put in **bold**. References and citations are typed inside [square brackets], using a highlighting color. They also act as hyperlinks when viewing the document in a computer or a portable device. Source codes appear inside blocks, with a number at the beginning of each line. UML diagrams follow the latest standard [OMG11, Spa11] available at the time this dissertation was written, version 2.3. The document's typographical design was kindly provided by Hugo Sereno Ferreira, with some adaptations.

Chapter 2

openEHR

2.1 Fundamentals 11
2.2 Package Structure 13
2.3 Archetypes 14
2.4 Templates 20
2.5 OpenEHR Java Implementation 23
2.6 Conclusion 24

Having identified the fundamental problem of *incomplete by design* and *design for incompleteness*, and identified the problem of a large number of possibilities to show clinical concepts, it is now suitable to delve deeper. But before showing how to allow to adapt or to build new GUIs, which is the concrete solution, there are some context that must be explained. Like said in § 1.6 (p. 6), this dissertation, proposed by *Critical Software, SA*, represents a research to a concrete problem that they have in a current project related to health software. *Open*EHR is currently used to represent clinical knowledge and concepts. Because it is part of the context of the problem, and also of the solution, it is important to understand it.

2.1 Fundamentals

Nowadays there is a problem in which Informations and Communications Technology cannot effectively represent health *semantics*(the meaning of the information), which is very complex [opea]. *open*EHR is a knowledge-oriented that supports ontology, terminology and semantics, to represent complex and to share complex concepts. Its goal is to effectively support health area at different levels, including medical and administration purposes. Besides that, it allows to create adaptable health computing systems and patient-centric

electronic health records (*EHRs*) [ope07c]. The information in *open*EHR is computable, but also shareable. This means that the information can be:

1. Interpreted and understood in the same way, by different parts of the system, like it is by a human;

2. Used and shared by different systems.

The first characteristic, *semantics* support, is a key point of this standard, which tries to resolve the difficulty of representing the meaning of information. The second means that the system can be distributed, but at the same time that it supports patient-centric EHR.

Figure 2.1 represent the *open*EHR Health Computing Platform, and the relationships with the specifications.

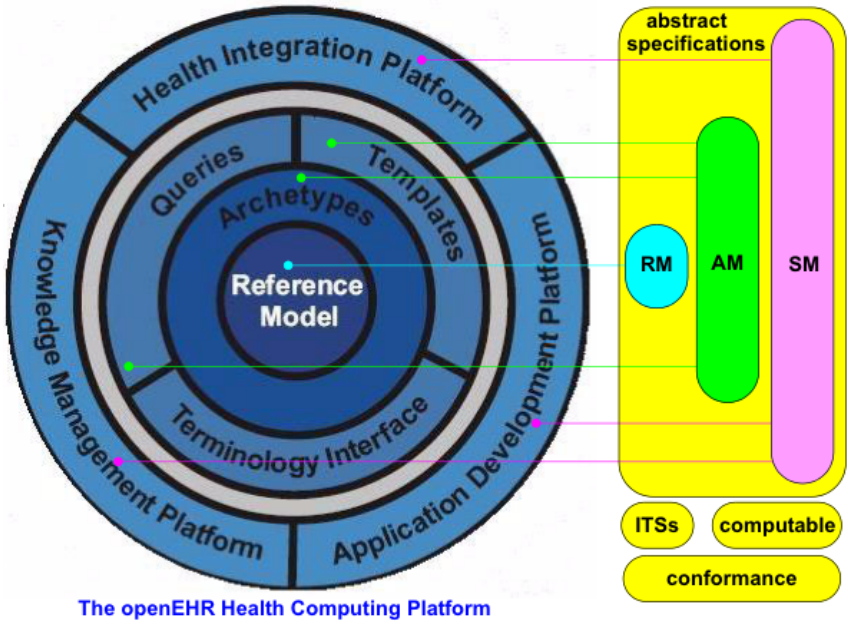

Figure 2.1: The *open*EHR Specification Project, presented on [ope07b].

On the right side of Figure 2.1 we can see some *deliverables*, and to which parts of the computing platform they are mapped. Reference Model (RM), Archetype Model (AM) and Service Model (SM) are described lately, in § 2.2 (p. 13).

*Open*EHR has a generic architecture, meaning that it satisfies more than clinical EHR. Its RM is for concepts related with *service and administrative events relating to a subject of care* [ope07b]. *Archetypes* and *Templates* is what specifies what kind of subjects are defined. This means that *open*EHR can met different requirements, determined by archetypes and deployment. Based on [ope07b], they requirements of *open*EHR are:

1. Suitable for different care views;
2. Support for different clinical data structs, as lists, tables, and other;
3. Support for all aspects of pathology data;
4. Support for natural language
5. Support for privacy;
6. Support for sharing of EHR via interoperability of systems - patient-centric EHR

*Open*EHR supports patient-centric EHR, for a shared patient knowledge between systems. Figure 2.2 shows an example of a context on which *open*EHR can be deployed.

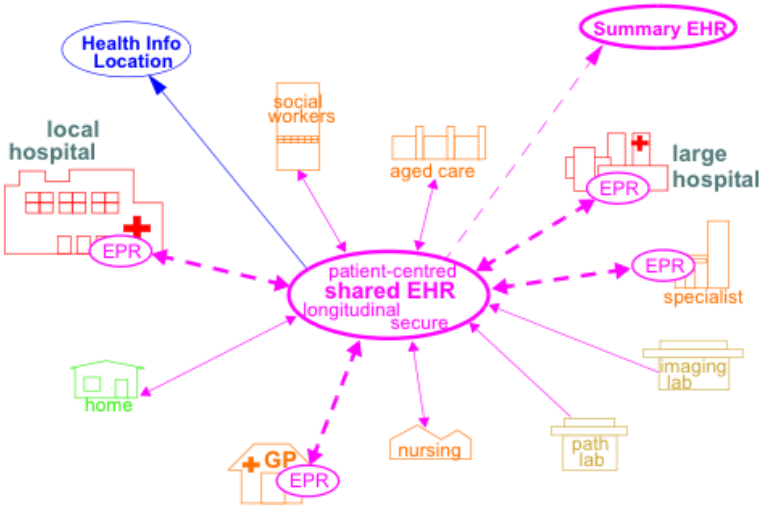

Figure 2.2: *Open*EHR deployed in a shared-care context, presented on [ope07b].

It represents the deployment in a shared health community. The information can be shared between different types of hospitals, as well as with specialists.

2.2 Package Structure

*Open*EHR formal specifications' package structure have three major packages: RM, AM and SM. Figure 2.3 shows RM, AM and SM packages, and the relationships between them.

RM's core provides knowledge access, identification, data types and data structures, that can be re-used in other packages. This re-use provides archetyping support. The *Support Model* describes basic concepts used by all other packages, like identification and terminology. The semantics described here provides other packages the usage of

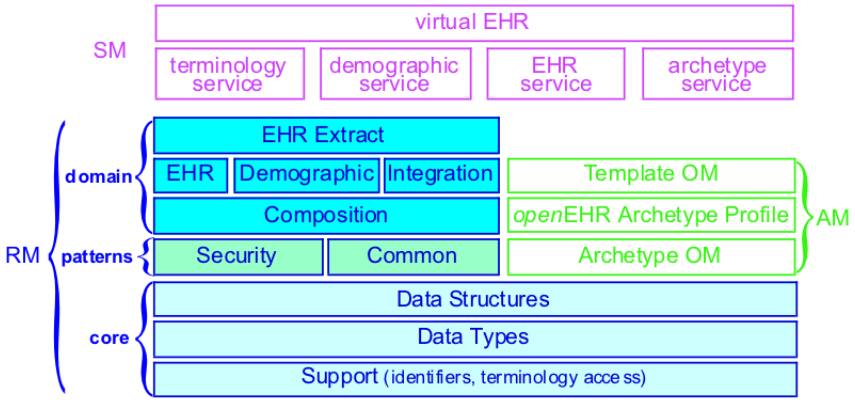

Figure 2.3: *Open*EHR's RM, AM and SM package structure, presented on [ope07b].

identifiers and the access to knowledge like terminology. *Data Types Model* defines data types that can be used by other models, providing specific types to cover all kinds of health information. Data types include *text*, *quantities*, *date and time*, and *basic types* like variables. *Data Structures Model* describes generic data structures to be used in archetypes, for content structure organization. This includes single items, lists, tables, and tree structures. In the *pattern* layer of RM there is access control and privacy settings to information, as well as some concepts that are common to other-level packages. *Domain* layer defines the domain and the semantics of some *open*EHR concepts [ope07b].

AM provides models to describe the semantics of *archetypes* and *templates*, and it includes ADL, the Archetype Definition Language [ope07b]. *Archetypes* are descriptions of valid *Entries*, *Sections* and *Compositions*. These are expressed in a formal manner which enables them to be computable. shared between systems. An archetype represents the description of all the information a clinician might want to report about something. Templates are compositions of archetypes to meet some specific purposes, and they inherit the way archetypes describe information [opea].

SM is a more technical and computer viewpoint of *openEHR* [opea], as it defines services for EHR computing environment [ope07b].

2.3 Archetypes

Archetypes, as well as *templates* § 2.4 (p. 20), are domain-level definitions, and they specify which requirements the *open*EHR implementation satisfies. They are computable expressions of a domain content model, typically health-related concepts. These *archetypes* are based on the RM, and they are expressed using formalisms [ope07a, ope07b]. We can

think that an *archetype* is like a LEGO® object, as different objects can be built using the same bricks. They use RM's components, and like LEGO® bricks, different archetypes can be built with the same components of the RM [Bea07]. Figure 2.4 shows a representation of this.

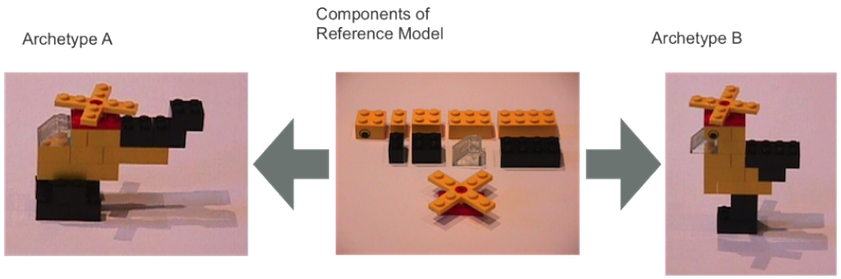

Figure 2.4: Building Archetypes from Reference Model's components. Presented on [Bea07].

Concepts defined by *archetypes* and *templates* include clinical knowledge and concepts like "Blood Pressure", "HbA1c Result" [MS07]. However, its representation is not *ad hoc*. They are from different types, with different purposes, and they have a concrete structure and concrete *content*.

2.3.1 Types of Archetypes

Archetype's represent clinical information of various types, with different purposes. For example, there are archetypes related to clinical domain definition, like the definition for procedures, which may be of type *Instruction*. Other clinical domain definition could be related to observational processes, being of *Observation* RM type [ope08c]. Also, there are archetypes concerned with information structuring. Next in this dissertation several RM types of archetypes are listed and explained.

Domain Level

Domain level archetypes are contained in the *ehr* package, inside RM package of *open*EHR. They are *top level* containers for clinical information, defining its context *semantics*, giving meaning to data [ope07b]. These domain level archetypes could be of the following types:

- COMPOSITION. This type of archetypes represent the *root* point of clinical content. It is used as a *primary container* for data aggregation [ope08c]. Inside a *composition*, data is put inside *content*. Other key information can be found inside *context*, which gives a context of the information recorded inside, and as well in *composer*, which gives a even more context by giving the information of the person who is responsible

for the content inside the *Composition*. *Content* inside *Composition* archetypes are SECTIONs, either directly introduced inside the composition, or as another archetype (see template section § 2.4 (p. 20)). It can also have ENTRYs, even without SECTIONs.

- SECTION. This act as a navigation package, defining an hierarchical navigation structure. They provide a logical structure to arrange *Entries*. Although they are not mandatory in an archetype or a template structure, they can be used to provide a domain hierarchical arrangement of clinical information [ope08c].

- ENTRY. This kind of archetypes represent a *clinical statement* [ope08c]. In order words, they define clinical information, defining semantics of the information content, based on their type [ope07b].

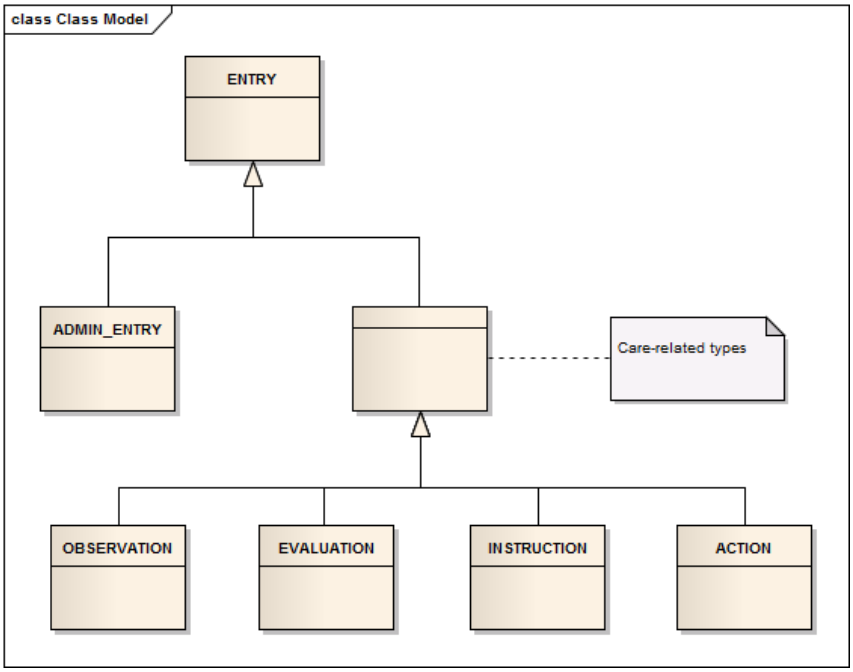

Figure 2.5: Entry types. This RM type define the semantics of the clinical knowledge. Based on [ope08c].

As it is possible to see in Figure 2.5, ENTRY archetype's RM type define clearly the semantics of the information the archetype represents. ADMIN_ENTRY is the only one that is not health care-related. As the name shows, it is used to record administrative information, for instance by non-clinical staff [ope08c]. The other sub types are related to health-care. OBSERVATION is related to observational processes, including test results, while EVALUATION is related to pathology diagnostics, including the diagnostics itself, the context for the diagnostics, or general recommendations for overcome the problem. Moreover, ACTION and INSTRUCTION archetypes specify

the general actions and procedures to be taken to overcome the patient problem. These procedures can be directed to health professionals, and may or may not be passed to the patient [ope08c].

Item Structure

These types of archetypes represent ways to structure clinical information, such as tables, lists or tree representations [ope08b].

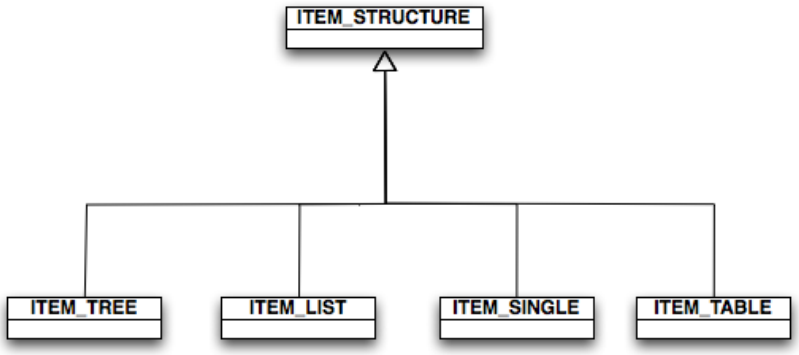

Figure 2.6: ITEM_STRUCTURE RM type. Based on [ope08c].

From the RM types shown in Figure 2.6, only ITEM_TREE archetypes exists. These archetypes are used to represent information which is best represented as a tree, like "biochemistry results" [ope08b]. The other types, ITEM_SINGLE, ITEM_LIST and ITEM_TABLE, are represented as elements from the *Representation* package, as CLUSTERs and ELEMENTs (see § 13).

Representation

Representation archetypes are simple hierarchical representation of any clinical data [ope08b]. They are the most basic and meaningless data structure to represent information in *open*EHR. Representation archetypes can be of two types [ope08b]:

- CLUSTER. These archetypes are groups of ELEMENT. In order words, they contain a list of ordered information.
- ELEMENT. These are the leaf of all archetype structure. They contain the clinical information itself inside it, by the form of *data types*, as explained in § 2.2 (p. 13).

Other types

There are other RM types that can be mapped to archetypes. These archetypes are:

- HISTORY. This RM type represent the concept of time, giving a timeline for clinical information [ope08b].
- DEMOGRAPHICS. This RM type can be mapped or not to archetypes. They represent personal information related to an EHR [ope07b].

It is important to understand that these are *Reference Model* types, which can be, or not, mapped into archetypes. Moreover, an archetype can contain other archetype's type without the concept of being another archetype. For instance, a INSTRUCTION archetype can contain ITEM_STRUCTURE RM type inside it, without having actually another archetype inside. When there is actually an archetype inside another, we may be talking about templates, as it is shown lately.

2.3.2 Node Types

An archetype can contain other archetype's type data, which means that it contains additional clinical information inside. Figure 2.7 represents an example of an ITEM_TREE archetype.

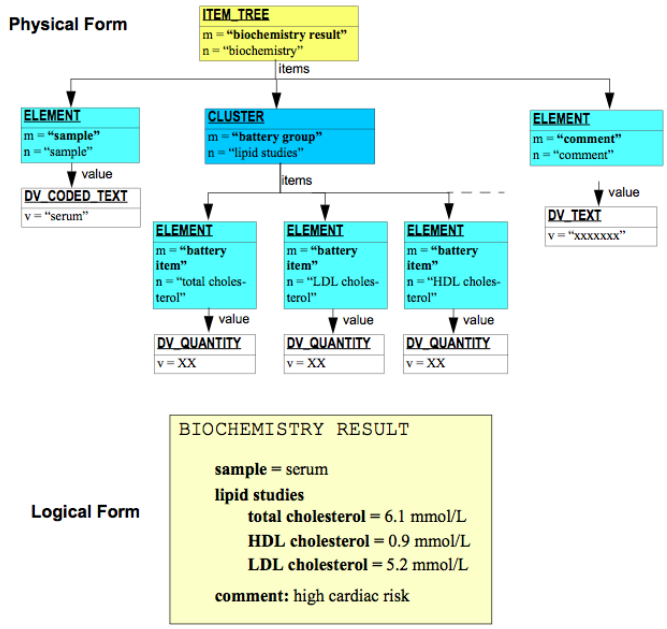

Figure 2.7: A representation of an example of a ITEM_TREE archetype [ope08c] with other RM types included inside.

Each RM type present in the archetype shown in Figure 2.7 is internally represented as a *node*. Each node has a unique *path*, and other attributes depending on the node type. Figure 2.8 (p. 19) represents the archetype nodes available.

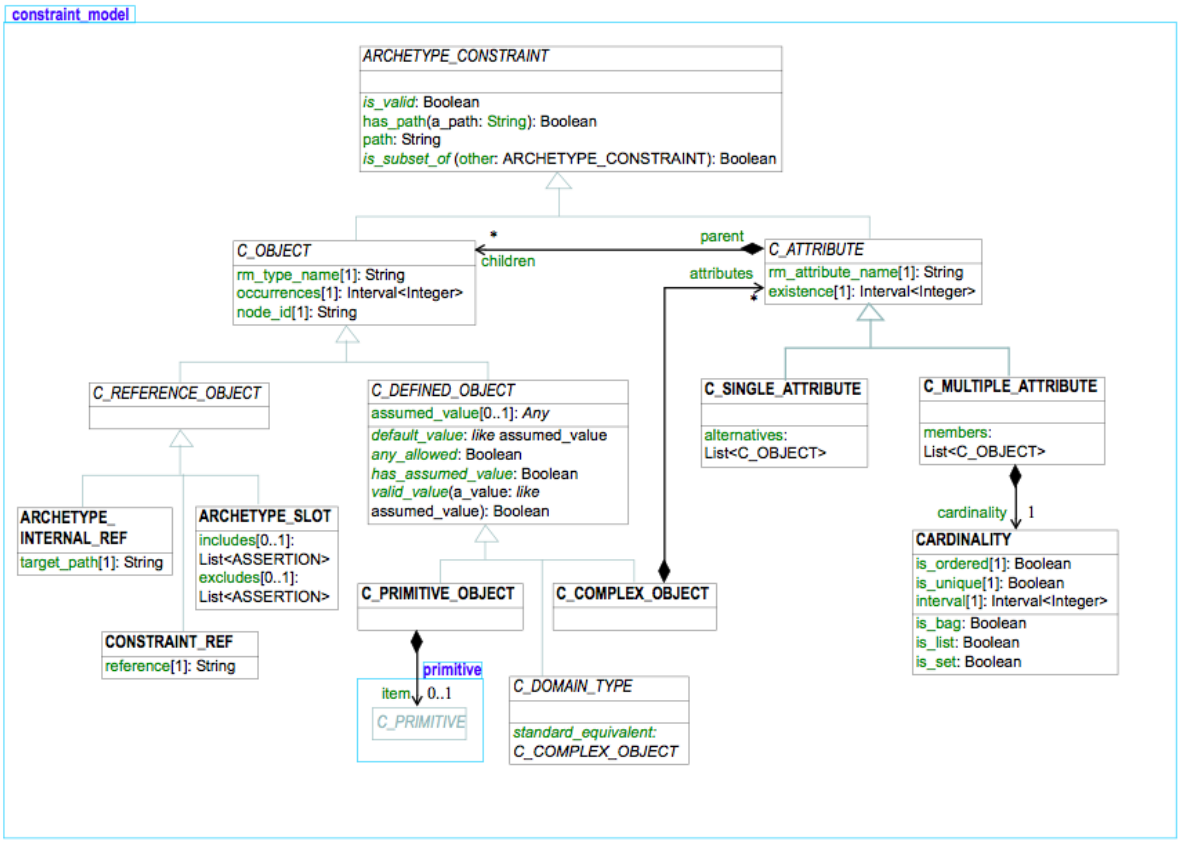

Figure 2.8: *Constrain_model* with archetype nodes [ope08a].

According to what is seen on Figure 2.8, and based on the Archetype Object Model [ope08a], lets take a look at the the most used nodes while parsing archetypes for building templates.

- **CComplexObject.** When navigation through archetype's structure, this kind of node appear many times. They represent a node which contains other nodes inside (they can actually contain other *CObject* objects inside). They can refer to RM types shown in § 2.3.1 (p. 15).

- **CAttribute.** This type of node is an abstraction of other attribute types, either single or multiple attributes. They define constraints on the objects, and they appear right inside attributes are of *CObject* type. They can constraint data itself, or *CComplexObject* for aggregating more nodes inside, or even *Archetype Slots*.

- **Archetype Slot.** Archetypes have the possibility of incorporating more clinical information, either by having it inside the archetype, or by adding other archetypes. This is particular useful when building *templates* (see § 2.4 (p. 20)). These new archetypes are put in these *archetype slots*, which may allow only certain RM types

of archetypes, or either include or exclude specific archetypes. They contain also *cardinality*: in other words, the number of maximum and minimum archetypes that can be inserted in an *archetype slot*.

As said, these types of nodes contain a path relative to archetype root, which is unique. They also contain an internal code (*atcode*), used for internal references and for terminology (for example, a textual representation of a node). Some nodes also contain *cardinality*, as said [ope08a].

2.4 Templates

Templates are specialized and usable definitions, being composed by *archetypes*, and are built and used for specific purposes [ope07a, ope07b]. While building templates, archetypes are specialized by overriding some of its properties, hiding nodes, or by adding new archetypes inside them. It is a process of using a more general description of clinical knowledge (archetypes - see § 2.3 (p. 14)), and specializing it towards some specific necessities. These templates often correspond to graphical user interfaces, reports or messages, for health care usage.

This dissertation aims to study the best way to allow health professional to build these *templates*, by choosing *archetypes* that composes them, and by making the necessary specializations. So, we can think of a *template* as a composition of LEGO's® objects (see Figure 2.4 (p. 15)). Figure 1.3 (p. 6) shows how *archetypes* can be grouped forming *templates* for specific purposes. For instance, it is possible to have a *template* for a "Routine Checkup", being composed by *archetypes* like "Blood Pressure" or "Haemoglobin A1c Result".

2.4.1 Template's Structure

An *archetype* contains data inside it, but is can also contains additional data by having archetypes inside it. So, *archetypes* are composed by structures, some of them being root points when connecting to some other *archetypes* [ope07b]. Figure 2.9 shows how a *archetypes* and *templates* are related.

These archetypes are linked through the archetype root point, and the other archetype's *archetype slot* (§ 2.3.2 (p. 18)). However, they follow a specified structure. For instance, it is not possible to have a template in which its root point is an ELEMENT archetype. Adding to this, it is not also possible to add a COMPOSITION inside an ENTRY archetype. The correct template structure is represented in Figure 2.10 (p. 21) as an UML class diagram.

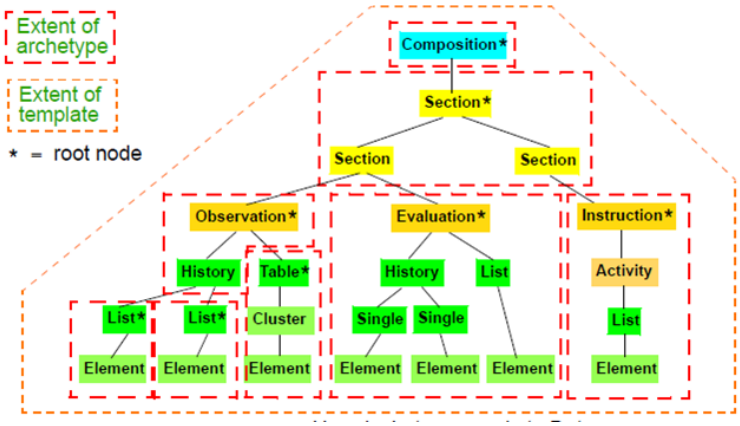

Figure 2.9: How *templates* are a group of *archetypes*, and how they apply to data. Presented on [ope07b].

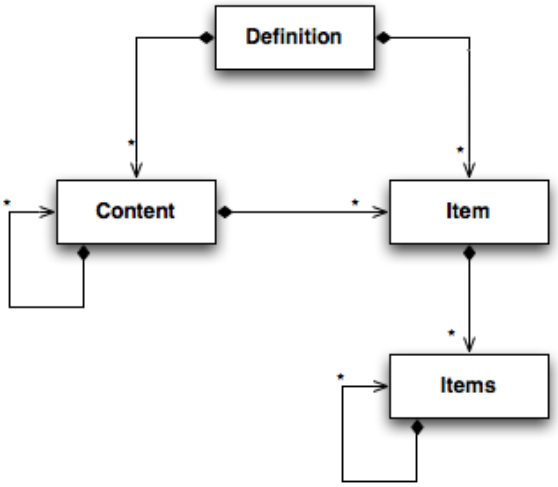

Figure 2.10: Template structure represented as an UML class diagram. It is possible to see a structure, and the cardinality of each relation.

It is possible to see in Figure 2.10 that a Template starts with a *Definition*, which is a COMPOSITION archetype. Inside this *Definition*, it is possible to have either a *Content* with a list of *Item* inside, or a list of *Item* without having a *Content*. It is also possible to have a *Content* with other *Content* inside. Moreover, inside each *Item*, it is possible to have *Items*. These new *terms* correspond to the types of archetypes explained in § 2.3.1 (p. 15),

- **Definition.** Corresponds to COMPOSITION archetypes.
- **Content.** Corresponds to SECTION archetypes.
- **Item.** This point of an archetype corresponds to ENTRY archetypes.

- **Items.** Finally, *Items* corresponds to data structure archetypes, like CLUSTER, ELEMENT, ITEM_TREE or HISTORY.

A more complete relationship between *archetypes* and *templates*, and other different artifacts of *open*EHR is shown in Figure 2.11. It is possible to see that a *template* are more specific that *archetypes*, and that it can take the form of GUIs.

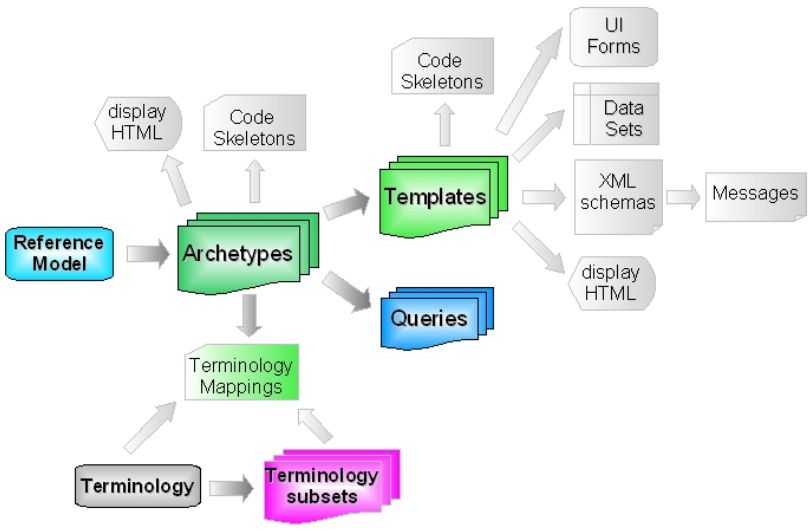

Figure 2.11: Relationships and dependency between different kinds of artifacts - information architecture. Presented on [ope07b]

2.4.2 How Templates specialize Archetypes

Templates are more specific than archetypes: they fit more specific purposes, as it is possible to see in Figure 2.11. Templates are a group of archetypes, with some properties being overwritten. Each archetype is composed by nodes, as it was explained in § 2.3.2 (p. 18), which has its own properties. Moreover, archetype node objects even have attributes, defining its cardinality, its path, its allowed values, and so on.

While adding an archetype into a template, we are actually adding a reference to it. It makes it possible to redefine some of these attributes. For instance, it is possible to define a new minimum of occurrences, changing what was defined in the original archetype. This is done by two ways: (1) Defining attributes and properties in archetype's root node (see § 2.4.1 (p. 20)); (2) By creating rules to override other node's attributes. Source 2.1 shown an example of a template file definition.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<template xmlns="openEHR/v1/Template" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:type="Template">
     <id>...</id>
     <template_id>...</template_id>
     <name>...</name>
     <original_language>en</original_language>
     <description>
         ...
     </description>
     <definition xsi:type="COMPOSITION" archetype_id="openEHR-EHR-
         COMPOSITION.encounter.v1">
          <Content xsi:type="SECTION" path="/content" archetype_id="openEHR-
              EHR-SECTION.adhoc_csw.v1">
               <Item xsi:type="OBSERVATION" path="/items" archetype_id="
                   openEHR-EHR-OBSERVATION.sample_blood_pressure.v1">
                    <Rule min="1" path="/data[at0001]/events[at0006]/data[
                        at0003]/items[at1007]"/>
                    <Rule hide_on_form="true" path="/data[at0001]/events[at0031
                        ]/math_function"/>
                    <Items xsi:type="CLUSTER" path="/data[at0001]/events[at0006
                        ]/state[at0007]/items[at1030]" archetype_id="openEHR-EHR
                        -CLUSTER.symptom-pain.v1">
                         <Rule max="0" min="0" path="/items[at0034]/items[at0058
                             ]/items[at0090]/items[at0143]"/>
                         <Items xsi:type="CLUSTER" hide_on_form="true" path="/
                             items[at0034]/items[at0058]/items[at0090]/items[
                             at0146]" archetype_id="openEHR-DEMOGRAPHIC-CLUSTER.
                             person_birth_data_iso.v1"/>
                    </Items>
               </Item>
               <Item xsi:type="EVALUATION" hide_on_form="true" path="/items"
                   archetype_id="openEHR-EHR-EVALUATION.problem-csw.v1"/>
          </Content>
     </definition>
</template>
```

Source 2.1: Example of an OET Template, a Template with referenced archetypes. It is possible to observe its structure, and the archetypes which make that structure, according to what was explained in Archetype Types and Template Structure sections. It is also possible to observe some *Rules* created, to override and specialize the original archetype's properties.

2.5 OpenEHR Java Implementation

The *open*EHR Java implementation is particularly important, as the *Template Builder* uses it. It is open-source, and allows to implement either client-server or web based systems [CK07]. *Open*EHR concepts are mapped into Java types, including *data types* of the standard. This implementation is aligned with the *openEHR* specification [opeb]. However, not all concepts are implemented. The Reference Model is implemented, as well

as archetypes and archetype utils [CK07], but it lacks of the template implementation.

This dissertation does not aim to implement templates: it uses an existing implementation to build templates, using for that a GUI developed using *ZK*. However, template operations were implemented, particularly when related to the linkage between the GUI and the model. The template's implementation in Java was done by *Critical Software, SA*. See Chapter 6 (p. 43) for further details.

2.6 Conclusion

*Open*EHR is a knowledge-oriented framework that allows to represent and share complex concepts. Although it meets generic requirements, its goal is to represent clinical knowledge, supporting health *semantics*, which is very difficult to represent. This information, in *openEHR*, is computable and shareable, which means that it is understood by the system, as it is understood by humans, and that it is shared between different systems. *Open*EHR also enables patient-centric EHR, making it possible to share the information between several different systems.

The Reference Model (RM) describes basic concepts used by all other packages. In fact, RM's components are the basics of *archetypes*, that describe the domain the be modeled. In this particular context, they can describe clinical concepts in a computable way. *Templates* are compositions of *archetypes*, and in this context they are mostly important and used in GUI, for specific purposes.

This dissertation studied the best way to build those templates from the composition of different archetypes, allowing the health professional or any end-user with *open*EHR knowledge to build them by choosing archetypes and overriding their attributes. The way this is done is explained in Chapter 6 (p. 43). But first, other key background is given, starting by ZK framework, which is used for the GUI.

Chapter 3

ZK Framework

3.1 Fundamentals 25
3.2 Architecture 26
3.3 Example of usage 26
3.4 Conclusion 28

Having explained how clinical concepts are going to be represented, this chapter presents the technology on which they are represented and manipulated. ZK is an AJAX Web application framework for rich GUI for Web applications. It was suggested by *Critical Software, SA*, and it is part of the problem of how to enable *design for incompleteness* on health systems GUIs.

3.1 Fundamentals

ZK is an AJAX event-driven, component-based framework for Web, enabling the creation of rich Web applications. Typically AJAX means working with JavaScript and XML technologies, but not with ZK. This frameworks makes it easier to work with AJAX, as the JavaScript code is automatically generated by it. The programmer only has to think about building the GUI business logic [Mah10, CC07]. This creates an abstraction so that the developer does not have to deal with AJAX's complexity, allowing him to focus on business logic and GUI development [Ins08].

ZK uses a XML-based approach to define the user interface [Mah10]. It has a specific markup language named ZUML, based on ZUL and XHTML. Also, it is possible to design the GUI using Java [ZK], as well as implementing business logic. The fact that it uses ZUML allows for enriched web applications based on XUL and XTML components, making the

design as simples as designing HTML pages. The event-driven engine makes it intuitive to develop, allowing to manipulate XUML components triggered by the user activities [CC07].

3.2 Architecture

ZK has a client-server architecture. In the client side, the user fires events, active like a view [Mah10]. In the server side, it updates the UI and accesses to backend resources, like a database [ZK, CC07, Mah10].

ZK has three major components: ZK Loader, ZK Asynchronous Update Engine, and ZK Client Engine. Figure 3.1 shows the architecture of ZK.

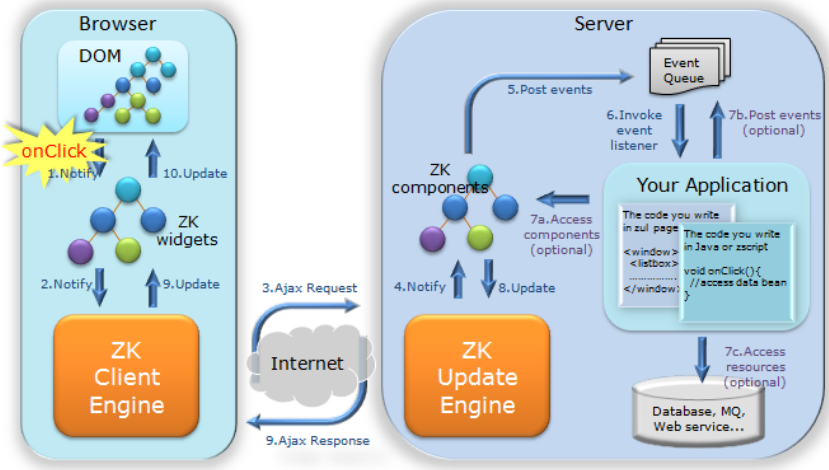

Figure 3.1: ZK client-server architecture, presented in [ZK].

ZK Client Engine is the client side of ZK [Mah10], and it is composed by JavaScript code [CC07]. It sends requests to the server, and interpret the correspondent responses to update the DOM. When there are JavaScript events, it will send AJAX requests to the server side, and will wait for its response [CC07]. ZK Loader is part of the server, and it interprets URL requests, generating complete HTML pages. These HTML pages are sent to the client side, to be interpreted by ZK Client Engine. The other component of the server side of ZK is the Asynchronous Update Engine. It is responsible for receiving and respond to AJAX requests, so that the browser can be updated [CC07, Mah10].

3.3 Example of usage

ZK makes it easier to design GUI because of the usage of a simple markup language. It is possible to use either XUML markup language, or Java, or even to use them together, for the business logic development and even triggering.

The ZUML markup language, with XML and XHTML components, is declared in a `.zul` file. This file is interpreted by the server, and the server can respond to AJAX requests if they are specified.

To illustrate how simple is to design the application GUI, a simple example was developed. This example contains a `window` with two `labels` and a `button`. Figure 3.2 represents the GUI for this application.

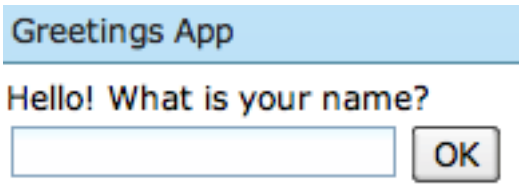

Figure 3.2: Example of a GUI developed using ZK.

Source code 3.1 represents Figure 3.2

```
1 <?page title="Greetings App"?>
2 <window title="Greetings App" border="normal" width="200px">
3   <label value="Hello! What is your name?" />
4   <textbox id="name" />
5   <button label=``OK'' id="buttonok" apply="greetings.Controller" />
6   <label id="lbl" value="" />
7 </window>
```

Source 3.1: Example of a simple ZUML interface with some components.

Designing the GUI is as simple as defining some components. All these components have a `window` as its parents. The objective of this simple application, shown in Figure 3.2, was to enter a name and click "OK". Then, the application greets the user. This is shown by Figure 3.3.

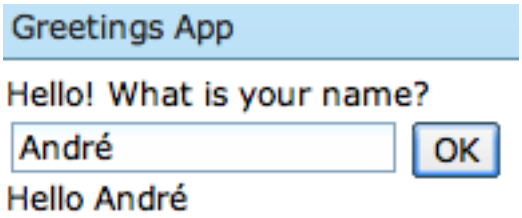

Figure 3.3: The application triggers an event when the user clicks "OK".

As ZK support a Model View Controller architecture, it is possible to separate the View from the Controller. For instance, whenever the user clicks "OK", an event is triggered,

and handled by the Controller: the second `label` with `id="lbl"` is updated with a greeting. This is done using AJAX, as the client send a request, and the server respond with instructions to for the client to update the GUI. In line 5 of source 3.1 there is a `apply` attribute, which enables an event to be triggered by what it is defined in that class. The definition of the behavior can be put inside the `.zul` file. However, in this example it is inside a Java class, as a Controller class, as shown in source code 3.2.

```
1  package greetings;
2
3  import org.zkoss.zk.ui.util.GenericForwardComposer;
4  import org.zkoss.zul.Label;
5  import org.zkoss.zul.Textbox;
6
7  public class Controller extends GenericForwardComposer {
8
9    Textbox name;
10   Label lbl;
11
12   public void onClick$buttonok(){
13     lbl.setValue("Hello " + name.getValue());
14   }
15 }
```

Source 3.2: Example of controller written in Java to trigger an event when clicking on a button on source 3.1.

`onclick$buttnok` method defines the behavior for *onClick* action on button "OK", setting the value of the `lbl`. As it is possible to see, the developer do not have to deal with JavaScript part of AJAX. He only has to deal with GUI design, event triggering and business logic.

3.4 Conclusion

ZK is an AJAX framework based on Java, for Web applications. It abstracts the developer from the JavaScript's AJAX side, allowing him to focus on GUI and business logic. ZK is a very good candidate for AUI, since it allows design and implement an editor to create GUIs for health professionals' specific needs. And because it allows rich web applications, it was a good technology to build an intuitive and rich *Template Builder* to manipulate clinical concepts, in order to create specific GUIs.

Chapter 4

Adaptive Systems

4.1 Overview 29
4.2 Adaptive User Interfaces 30
4.3 Adaptive Health Systems 31
4.4 Examples of the Implementation of AUIs 32
4.5 Conclusion 34

As said before, requirements can *change* or *evolve* over time, or they can be difficult to predict, especially in health systems. Additionally, there is a need of complete traceability of all health care activities [PS01]. It is clear that it is not possible to have a developer continuously working on the software to cover all health needs as they are identified. Designing to deal with this *incompleteness* is a solution to overcome this problem. But how is possible to do that? What concepts need to be applied?

This dissertation's main goal is to study and build a tool to allow to deal with this *incompleteness*. As said in Chapter 2 (p. 11), *open*EHR's templates can take the form of GUI's, fitting specific purposes. These *templates* are built typically by editing a XML-like file, or they can be built using specific tools. These tools exist, but are they a real solution for the problem? This research aims not only to develop a software component, but also to study how to make it a good solution to build *templates*, which may correspond to GUI.

4.1 Overview

Adaptable Systems, often mentioned as Adaptive Systems, are systems which can be adapted or reconfigured to different purposes [Fer10]. It is different from being the developer to change or deploy the system using a different configuration or with a different implementation, to respond of what is called *Software variability*: the need of usage of

the same software in different contexts [vGBS01]. Instead, *adaptability* means allowing a non-technological user, with limited skills, to adapt the system to different situations, without breaking the system.

Adaptive Object-Model (AOM) it a way of allowing this, giving the user the possibility to adapt the system in runtime, without recompiling the source [Fer10]. With AOM, business rules are described and stored externally to the program, using XML or databases, and these are interpreted in runtime. This allows the user to change the system in runtime, with immediate effect, in a controlled way to not compromise the system. AOM can be called an architectural pattern. It is based on metamodelling, as the system is represented by metadata instead of classes [TUI05]. It can be also defined as based on object-oriented design, and as having the property of being reflective [Fer10]. AOM architecture is composed by different *design patterns*. *Design patterns* are proven solutions for recurrent patterns [BMR+96]. These solutions, studied and experienced by skilled software engineers, promote good design practice, in this case for specific questions and problems of the overall architecture.

Some authors also refer *adaptivity*, which is the ability for the system to modify its behavior based on the environment [CLG+09]. These type of systems adapt themselves, in a pro-active way [Fer10]. One example is the creation of user models, which represents how a specific user usually interacts with the system, making it possible to adapt itself to him. However, this *adaptivity* is outside the scope of this dissertation.

4.2 Adaptive User Interfaces

Human-computer interaction is having great attention, and more and more researching in done in this area. This happens because of the increasing attention given to user interfaces, and to the interaction with the end user. There are interactive products everywhere, from cellphones to computers, or even remote controls, coffee machines, and so on [PSR07].

There is lot of researching on how to present contents to the user, and on the usability of the products. How are they usable? Are they enjoyable to use? Or do they present badly the contents? Besides that, it is important to think about, not only *how* the content is present, but also *which* content is shown to the user. This issue has received less attention, but it is important [Lan] because different users have different computer interactions, and different needs. The variety of needs have to be considered while defining usability criteria [Ben], to satisfy all users. And that is what this dissertation is about: cover all users needs, allowing the end user to change the GUI towards his own needs.

An Adaptive User Interface is an User Interface which can be adapted in runtime, without compromising the system. They *comply with changing rules*, and *provide more*

flexility to *deal with changes* [Ram09]. AUIs have an advantage over static GUIs, as they can reach a large number of possible interactions without having to be deployed using different configurations or a different source code [Ben]. Some authors also include the ability to create a user model to automatically present suggestions to the end user into its definition [Lan, Ram09]. But the research presented in this dissertation is focussed mostly on the possibility for the end-user to create its own GUIs, based on *open*EHR templates.

Due to more complex and sophisticated systems it is becoming more and more difficult to achieve satisfaction for all end users. A goal of Adaptive User Interfaces is to present the end user with an interface that is easy, effective and efficient to use, according to his requirements and needs [GG]. Actually, AUIs are a serious solution to different usability preferences [Ben]: with an interface that can have different kinds of interactions, it can be adapted to the user, allowing him to be more productive, and efficient on what he is doing. And to reach this, it is necessary to provide the end user a way to adapt the user interface in terms of the way content is shown, but also which content is going to be presented.

4.3 Adaptive Health Systems

In health area, as said earlier in this dissertation, there is a large number of different requirements. Accepting that health-related softwares are mostly *incomplete by design*, because it is very difficult to predict and to design software to meet specific needs, we can design it to deal with incompleteness. Adaptive User Interface and Adaptive Object Models allow to deal with incompleteness, as the system can be changed according to the user needs. And it is not only about allowing the end user to change *how* the content is present, but also *what* is shown, and *how* the system should behave in some situations according to health professionals and/or patient needs. So, the following problems exist:

- Different health professionals have different needs;
- Different patients have different needs
- Each End-user has its human-computer interaction preferences;

The first two are related to health area, as said earlier. These problems lead to a large variety of needs and combinations of clinical concepts. And because it is impossible to cover all requirements with a single or with pre-compiled user interfaces, allowing the health professional to change it is a good way of overcome this problem. The third problem is a common human-computer interaction problem: as in every thing, each user is a different users. Not all users like the content to be displayed in a single way: different

individuals might prefer different ways of seeing content, and they can prefer one way to another depending on the context of use.

To overcome these problems, the system must be designed in a way to allow *change*. In fact, in this particular case, *open*EHR (see Chapter 2, p. 11) allows this change with its *templates*. Also, the user can give specific meanings to the information. But this has to be completed with an AUI and an AOM approach that deals with software *variability*, by allowing *adaptability* in a way that it can be possible to manipulate clinical concepts, based on *open*EHR ARCHETYPES, to build new GUIs, represented by *templates*. Also, in terms of business logic, an AOM approach would be a good idea to deal with some non-GUI system properties that could be manipulated and changed without having to *hard code*.

4.4 Examples of the Implementation of AUIs

Some real examples of the implementation of Adaptive Systems and Adaptive User Interfaces exist. In this section two commercial software solutions are presented to better understand the concept of being *adaptable*.

Template Designer

Template Designer is a software developed by *Ocean Informatics* company, and it is an editor to create *open*EHR *templates*, from the aggregation of *archetypes*. Actually, this commercial software represents a solution to the main goal of this dissertation, which is to develop a component to build templates. However, the fact that it is a commercial software, which is impossible to integrate into Critical Software's project, made it impossible to consider it *the solution*. Moreover, as explained, this dissertation aims also to study the concepts explained in this chapter, in order to apply them in the best way possible.

Figure 4.1 (p. 33) shows the GUI of the application. It is possible to see three main windows: the template view, and the repository of archetypes, on the right side, and the *template* properties. To select some *Archetypes* from the repository, it is necessary to expand elements of the tree view. Expanding these nodes of the tree, it is possible to drag-and-drop some *Archetypes* to the main window, and put them inside the nodes as shown in Figure 4.1 (p. 33), like it is claimed by Ocean Informatics [Inf08].

The chosen *Archetype* (in this case a *Template* composed by a single *Archetype*) can be dragged into a green node. It is also possible to choose the possible *Archetypes* by right clicking on a node of the *Template* tree representation.

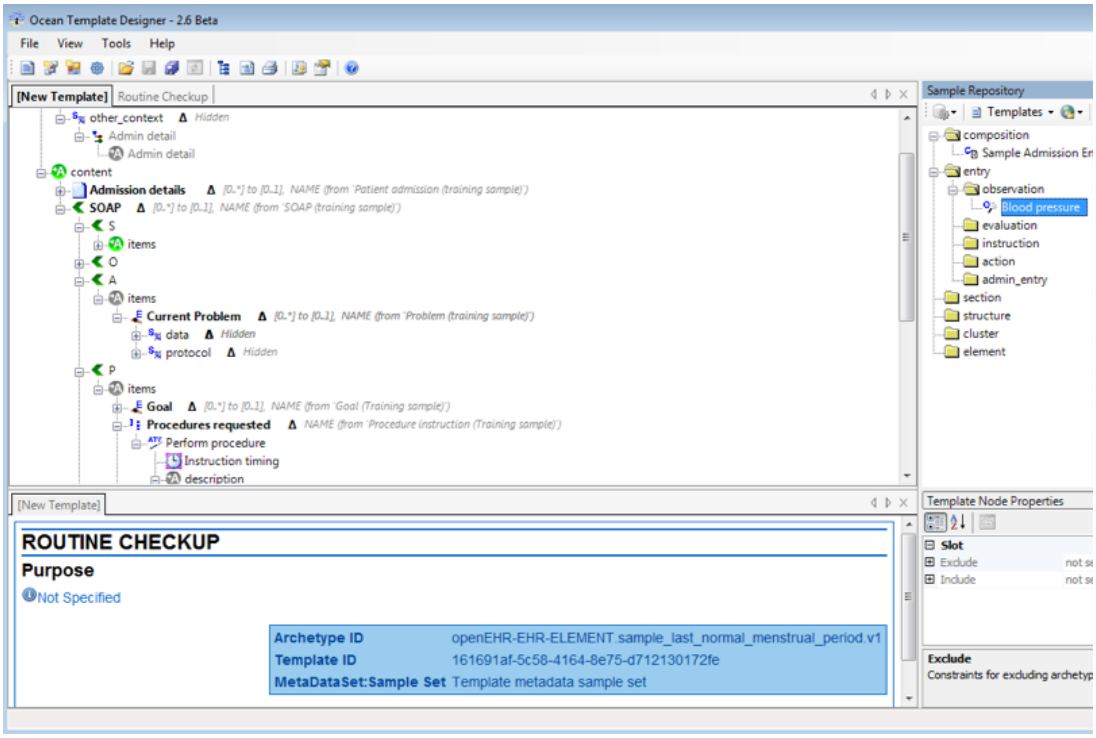

Figure 4.1: How to choose *Archetypes* in *Template Designer*.

Bento

Bento is a software developed by *FileMaker*, for Apple Mac OS X operating systems, with the objective of being a personal database software [Fil]. It is an excellent example of an AUI, being very intuitive and fitting for different purposes. It uses the concept of *Templates*, which is generically the same as *open*EHR *Templates*. There are a several of predefined *Templates*, to manage people, events, photos, or even health records. The problem of *Bento* is being so generic that it has a lack of focus on any particular subject [CNE].

Selecting the Health Record *Template*, it is possible to customize it, adding, creating or deleting fields. Figure 4.2 (p. 34) shows a customized Health Record. It is possible to customize the GUI, while the way data is stored is completely managed by *Bento*. However, it seams like it is impossible to customize it differently for different records in the same *Library*. In order to customize its GUI, Bento allows the user to drag-and-drop items from a menu into the form. This field menu is shown on Figure 4.3 (p. 34).

The GUI gives feedback, helping the user to place fields in the right place. Thus, this feedback is very a important design principle, for better interaction with the user [PSR07]. If the user wants a custom field, *Bento* allows this to be created, which makes it possible to customize the GUI as the user wants.

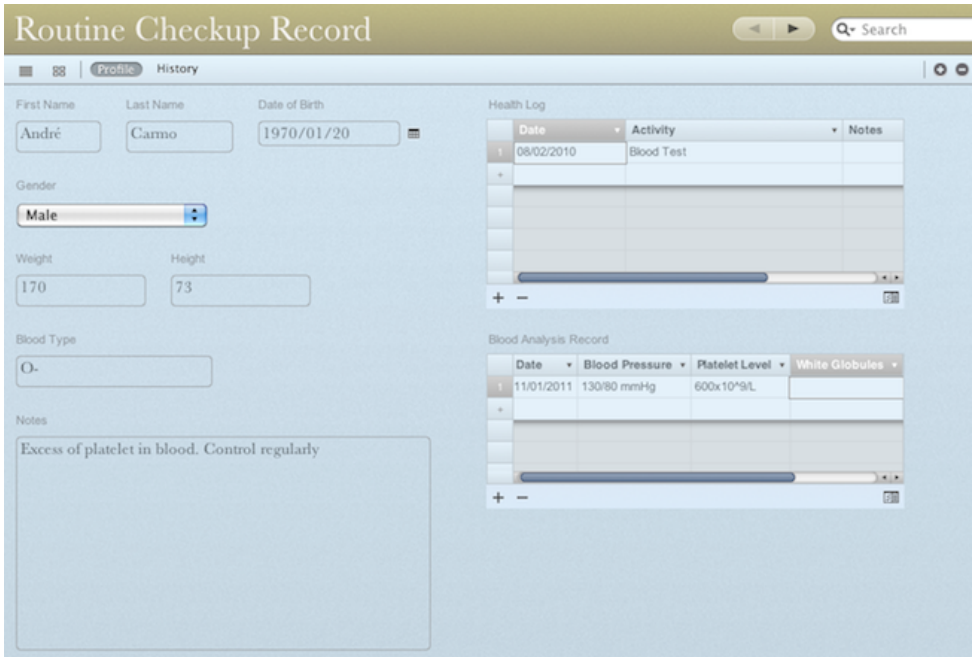

Figure 4.2: Customized Health Record in *Bento*.

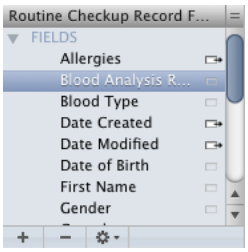

Figure 4.3: Optional fields that can be dragged into the main window in *Bento*.

4.5 Conclusion

In software *change* is part of the development process, and software development methodologies are becoming aware of this problem. These methodologies have to *design for incompleteness*, particularly in health systems. Users have different needs and ways of interaction, related to GUIs, and it is very difficult to define a usability criteria that covers all types of users. Adaptive systems are a good way of solving this problem, as they allow *adaptability* to respond to allow the software to be used in different contexts. AOM is a good way of allowing this, with the business logic being stored externally to the program code. It has a kind of an interpreter, which processes new business logic rules to adapt the system. In GUI, the concept of AOM can also be used, implementing what is called AUI. This provides the end user a way to change the GUI towards his needs. And this is very important to health systems, where there are some problems that cause an explosion

of different possibilities of interacting with the health professional, depending on his needs. AUI are, in fact, a serious solution to this problem, as they allow *change* to happen in an efficient way.

Two existing solutions were presented to better understand this concept of being *adaptable*. *Template Designer* is, in fact, very close to the solution that it is wanted, but it has some problems which made this need to create another software component, which was the starting point to this dissertation. The *Template Builder*, which is presented in Chapter 6 (p. 43), implements some of these concepts, like metamodel and even AUI at some point, but not so deeply as studied in theory - neither *Template Designer* does it. However, the author decided to study further than what was in fact implemented. It is important to notice that the *time* variable did not allow to implement these concepts more deeply.

Since the problem theoretical problem is presented, the next chapter talks the *methodology* followed to solve this problem.

Chapter 5

Research Problem and Methodology

5.1 Fundamental Challenges and Goals 38
5.2 Research Methodology 39
5.3 Validation Methodology 41
5.4 Conclusion 42

OpenEHR is gaining attention through developers and through health care professionals [LMT]. However, its usage with technologies like ZK has not been studied or documented. Its Java implementation implements some stable specifications [CK07], but lacks of the implementation of some key parts like *templates*. This obligates developers to make the necessary adaptations and even some implementations from scratch.

Adaptive User Interfaces are GUI's that can be adapted in runtime. And this can be applied to *open*EHR, using ZK framework. This chapter's goal is to give an overview of the main research questions, as well as the methodology followed on this research. The approach taken is presented with a certain level of abstraction.

The, *Template Builder*, which is the main goal of this dissertation, should integrate with Critical's project, which is being developed using *open*EHR, ZK for GUI, and Java for business logic. It is, at the same time, an industrial component, and a theoretical research about the best ways to present the information to the end user, and make him create and reconfigure GUI'S content. This need of creation and reconfiguration of screen for clinical usage lead to the Adaptive User Interface concept, which UIs can be dynamically changed in runtime. In limit, this was the ideal way of allowing an health professional to change the screen's content towards his needs.

Besides this GUI question, other goals emerged. *Open*EHR's lack of complete implementation obligates developers to implement what's left in a not so standard way. In fact, company's implement towards what they want and need, which lead to different

implementation, as opposite to the specification. An easy way to handle this was to define a set of rules to control the system behavior in some situations. A metamodel was needed to provide runtime system status verification and validation according to the rules defined.

5.1 Fundamental Challenges and Goals

This dissertations starts from the idea that *change* is present all over the software development process. It was seen that some systems are *incomplete by design* and should be *designed for incompleteness*. Some key concepts were explained, and the idea of having a component to build *open*EHR templates was explained.

As explained in early chapters of this dissertation, *open*EHR templates can take the form of Graphical User Interfaces. They are representation of clinical concepts and knowledge for specific purposes, by grouping different archetypes and redefining properties. In fact, with these two concepts of *open*EHR it is possible to allow the end-user to create a screen or change an existing one based on these *open*EHR artifacts. So, the main goal of this dissertation is **to build a component to allow end-users, even health professionals without technical knowledge, to build screens based on *open*EHR templates.**

Having the main goal of this dissertation, let's see other achievements raised by this main goal. They are strongly related to each other, as all of them are raised by the necessity of *embrace change*.

5.1.1 Content Presentation

As said before, AUI allows the end-user to adapt the GUI in runtime. In this specific case, it is wanted to create and (re)configure templates, which may correspond to screens, based on *open*EHR standard and ZK framework. The goal of creating and reconfiguring templates leads to other sub-goals including:

- Choose the best GUI patterns to allow the creation of templates, including the best ways to display either archetypes and templates, and the ways to add archetypes into templates;
- The usage of the AUI concept to provide a way to see the (re)configured GUI that is being created, based on *open*EHR templates.

5.1.2 openEHR and ZK connection

OpenEHR is highly supported by models [opeon], as ZK treats of the View part of the system. This makes it necessary to find a way to link these two technologies: the model must be mapped to the UI elements. It gains more importance as there are no known implementation of these two technologies together.

5.1.3 openEHR incompleteness

The Java implementation does not completely implements the *open*EHR specification, which makes the necessity for the developers to implement what is left. This is a potential problem as developers and companies may make necessary adaptation based on their specific needs, and that was what happened with Critical Software's implementation: the template's implementation was a bit different from the specification of *open*EHR. This led to the necessity of dealing with those differences. To overcome that, a metamodel was developed to allow the system to behave as it is defined in some rules. So, related to this incompleteness, the main goals are:

- Develop a way to adapt some system behavior according to what is defined somewere. To do so, a metamodel approach was thought to be the best way.
- Develop a way to store rules so that the metamodel can validate the system state.

5.2 Research Methodology

This research followed a general approach, a methodology proposed by Quivy [Qui08], adapted to a bottom-up approach. And even if it is presented as for *Social Sciences*, it can be applied to much more areas, even to software engineering.

However, the research process was not so straight. There was not a fixed and stable direction, as in software engineering requirements change or evolve [Som04, EM03, DL99]. And because of this, it is difficult to follow a strict top-down approach. Some problems that made not practical are documented in § 7.2 (p. 76). Figure 5.1 (p. 40) represents a methodology followed during this dissertation, based on the more general methodology presented by [Qui08]. As it is possible to see, it seems like an agile software development process, in which there is a constant solution redefinition.

The *Rupture* phase is to think beyond pre-assumptions. It is to study, starting from an initial question, with an open mind. Pre-assumptions are good, as they guide the researcher to investigate and validate his theories [dMA10]. However, an open mind is

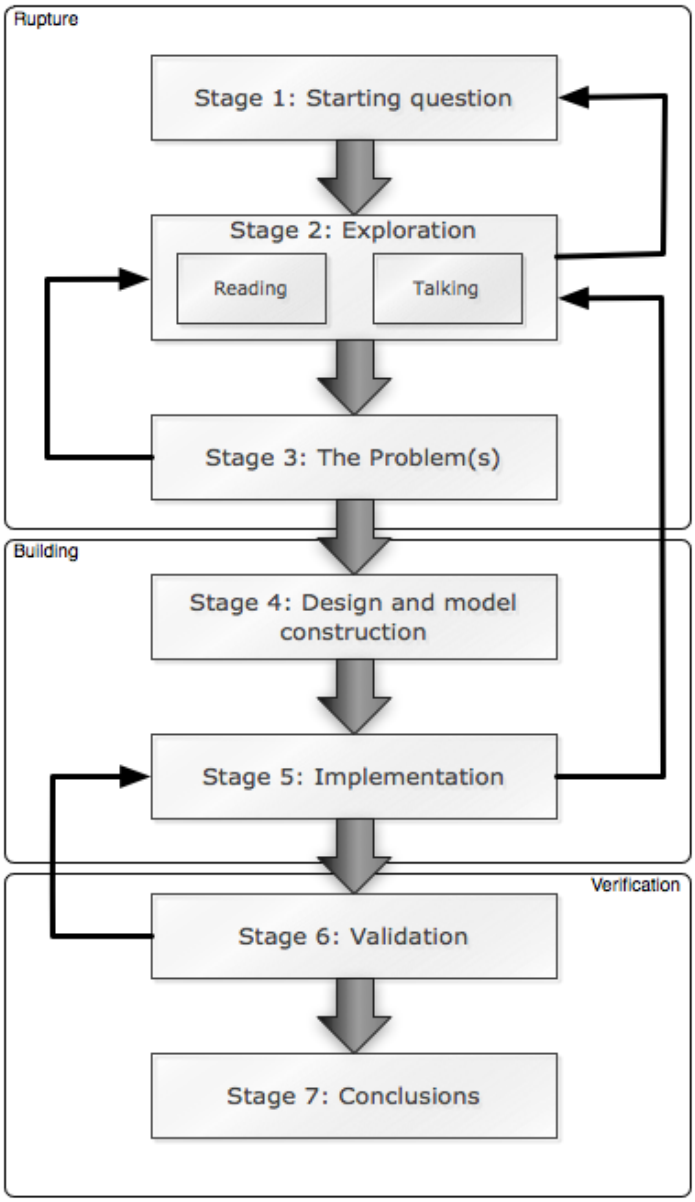

Figure 5.1: The Research Methodology followed, adapted from [Qui08].

necessary, as *unexamined pre assumptions* can be problematic [Bar04] if they don't allow the researcher to view more than what was initially thought.

This dissertation started with a propose made by *Critical Software, SA*. They wanted to study the best way to implement a *Template Builder*, to build *open*EHR templates. This was the starting point to choose documentation, and to obtain research questions, which was done also through conversations and through knowledge transfer, which tend to be a efficient way of understanding specific requirements [MMRG04]. That was a continuos process, which concluded with a Technical Report of what was studied.

The building and exploratory phase started with a revision of what was done, and

with some more documentation reading. Prototypes were done to better understand what was wanted to be implemented. As said before, this research process is not so top-down as it should be, at least in theory. When the exploratory phase started, it was seen that it could be necessary to go back and rethink the problem. It is possible to think of this as a bottom-up approach: as the implementation/design was in progress, new problems raised, and old problems were left behind. Actually, the development phase followed an agile approach, with the usage of Scrum. Figure 5.2 represents the 7 days *Sprint* (each iteration is called a *Sprint* in Scrum) cycle applied in this particular case.

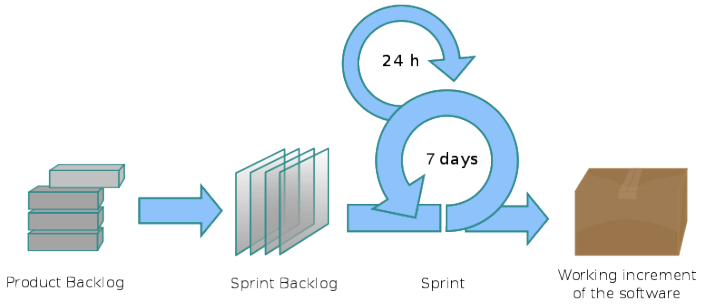

Figure 5.2: Scrum process, adapted from [Wik].

As seen in Figure 5.2, in each *Sprint* some functionalities of the *Product Backlog* were taken, and decomposed into small features to be implemented. In the end of each sprint, the work was analyzed and sometimes new problems were raised. This made the approach to be bottom-up-like, even though there were some initial thoughts and an initial global objective. The goal was specified and reworked as the project was implemented.

The last phase of this research was to validate what was done and to write conclusions. This validation was done through acceptance tests, following Critical's internal quality assurance process, which could guarantee the quality of the solution, and that it fits the requirements. This validation was also done qualitatively by people who had contact with some other software alternatives. The last part was the dissertation writing, which is a retrospective of what was done [dMA10].

5.3 Validation Methodology

The dissertation results have to be validated in some way, and the validation methodology should be the one that best fits the research problem. This particular dissertation was an industrial case study, with a concrete solution for a concrete problem, and the validation was as pragmatic as possible. It followed an *engineering method*, as the solution was

developed and evolved based on the regular results made [ZW98]. And this results and evaluation of the solution was done in a programatic way, using software testing methods and based on Critical Software satisfaction.

Critical Software internal quality assurance program includes, besides other testing phases, a rigorous accepting test phase, in which tests are planned and run to verify the software quality. Because they have knowledge about all the technologies used, this can guarantee that the software produced with quality, solving the problem and going towards the dissertation goals that were presented in § 5.1 (p. 38).

5.4 Conclusion

This particular dissertation joins the *design for incompleteness* with *open*EHR, and ZK framework. Its main goal is to study and develop a software component to deal with incompleteness in health systems based on *open*EHR, in which new templates can be built from archetypes.

To achieve this goal, a methodology was taken. The bottom-up approach, in which the problem and solution, as long with software testing, were redefined as the system was built, was a good idea mainly because of the short time-frame of this dissertation. This allowed to be more agile while understanding the problem and developing a solution. The validation for the solution was as pragmatic as possible, based on Critical Software satisfaction, and based on its internal quality assurance. The fact that this is an industrial problem obligated to take a specific approach both to the development and the validation.

After these methodologies were presented, it is time to shown some existing solutions, explaining why they are not really a solution to the existing problem. Next, in Chapter 6 (p. 43), the developed solution is presented and explained in detail.

Chapter 6

Solution

6.1 Components 44
6.2 Architecture Overview 45
6.3 Architecture in Detail 46
6.4 Solution Achieved 49
6.5 How are ZK and openEHR Linked? 64
6.6 Reconfigure the system 67
6.7 Conclusion 72

In this chapter the solution is detailed and explained. The objective of developing a component to create reusable *open*EHR templates, mainly for GUI purposes, was studied and accomplished. This *Template Builder* handles *open*EHR, and uses a GUI in ZK to present the information to the end-user. It builds these GUIs based on *open*EHR *archetypes*, grouping them and overriding attributes to produce *templates* (which mainly correspond to GUIs) for specific needs. This chapter explains how this is done, showing:

- Which components are used to build this solution;
- Which architecture is used, and how these components fit in it. It is also explained how these components are related;
- A running example to explain the most important aspects of the solution. Mainly, it is explains how to build and configure a specific template corresponding to a GUI;
- How there is a match between the system model status and what is shown to the end-user;
- How it is possible to (re)configure some system behavior, which is a consequence of this need to adapt health-care systems.

6.1 Components

Template Builder is not only a component itself, but it also takes advantage of other components through interoperability. It can be seen as a union of some different components and two different technologies. Firstly, two technologies are joined: ZK, the GUI framework, and *open*EHR, the framework to represent health-related information. ZK is based on Java, as said in Chapter 3 (p. 25). *Open*EHR's implementation was written in Java too, as explained in § 2.5 (p. 23), with some little adaptations by Critical Software, SA from *open*EHR Specification 1.0.2. (see [opeb]). Another component presented is a portion of a Critical's project, which implements some missing features in the official Java implementation.

- **ZK.** It is the framework for developing the GUI, and it allows to create rich user interface, with great usability. ZK typically act as the view, in a Model View Controller architecture, as it is going to be explained in 6.2. However, in this particular case, controllers also contains ZK API's calls to create the view elements.

- **Preview.** The preview component is used to render templates as long as they are built. This means that the end-user can see the GUI that he is creating and configuring. This *Preview* is a component from Critical Software's current project, and it was also developed using ZK framework.

- **openEHR.** This component represents the implementation of *open*EHR specification in Java. It contains the implementation of Archetypes and Reference Model, and methods to handle them. Archetypes are built from RM's parts, to represent clinical information. They are used, typically grouped, to build Templates. It is important to notice that templates are not implemented in the current version (1.0.2) of the *open*EHR Java Implementation (see [opeb]).

- **CSW-Component.** This component is a portion of a Critical Software's project. From now on, this is going to be named **CSW-Component**. The portion used contains utilities to handle Archetypes, like input and output streams, and the implementation for *open*EHR Templates, as well as its structure.

- **Template Builder.** Template Builder is the central component of the application, and it is what was developed by the author. It uses ZK to build the user interface, and *CSW-Component* and *open*EHR to manipulate and build templates from archetypes. It also uses the *Preview* to show the corresponding GUI of the template. It can be seen as a component that joins all other components, providing the ability to create templates in real time by any end-user.

6.2 Architecture Overview

Template Builder's architecture follows a kind of Model View Controller (MVC) architecture. MVC is a pattern that separates the application in three different layers [ZK], as explained in 6.1.

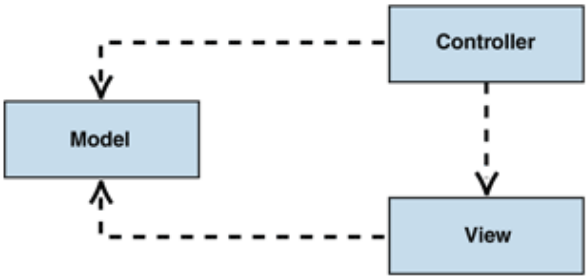

Figure 6.1: The MVC architecture, as shown in [mic].

These three layers are explained next.

- **Model.** It represents the data and the application domain [mic], and provides access to get and change information.
- **View.** The View manages the appearance of the information [mic].
- **Controller.** It links Model and View. For example, if there is an event or an action in the view, it updates the model and gets the information back to update the view according to the model [mic].

With MVC explained, it is possible to talk more specifically about Template Builder's architecture, and how its components are placed within the MVC architecture. Figure 6.2 (p. 46) represents this architecture and how components fit in it.

ZK and *Preview* components represent the view, and render what is happening in the system. Actually, they are linked, as *Preview* was developed using ZK. However, ZK here refers to the *Template Builder*'s GUI to create and configure templates. *OpenEHR* and *CSW-Component* represent the model, with their respective actions.

Template Builder fits in the three layers. It uses View components to build the GUI: ZK for the main GUI of the application, and *Preview* for to shown the template current status and representation in as a GUI. It also uses Model components, and even extends some of them, to complete the application model. This model represents archetypes and templates, having methods to manipulate them.

Then, there is the controller, which is taken care only by Template Builder's core. This controller handles GUI events and actions, updating the model, and reflecting it back to the UI.

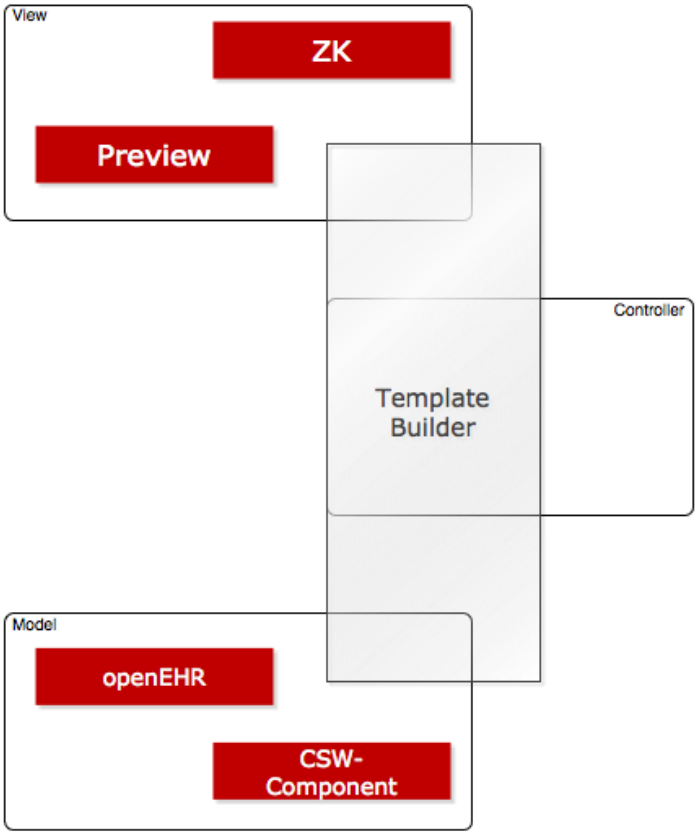

Figure 6.2: MVC Architecture of *Template Builder* with components identified.

The next section explains in even more detail each layer of the architecture. Moreover, it is explained why this MVC architecture is a *kind of*, as there is no clear separation between some views and some components.

6.3 Architecture in Detail

6.3.1 Model

The model uses some components that already exists, but also new *classes* to persist and access information that is present in memory. Figure 6.3 (p. 47) shows model components in detail and how they are linked.

Template Builder uses the other components, and it is the *core* of the system. It contains core data, and stores Templates and Archetypes. These Archetypes, from the **openEHR** component are read and parsed, in order to build the structure in memory, using methods from the **CSW-Component**. These archetypes are then used by *Templates*, being referenced and not actually copied to the template.

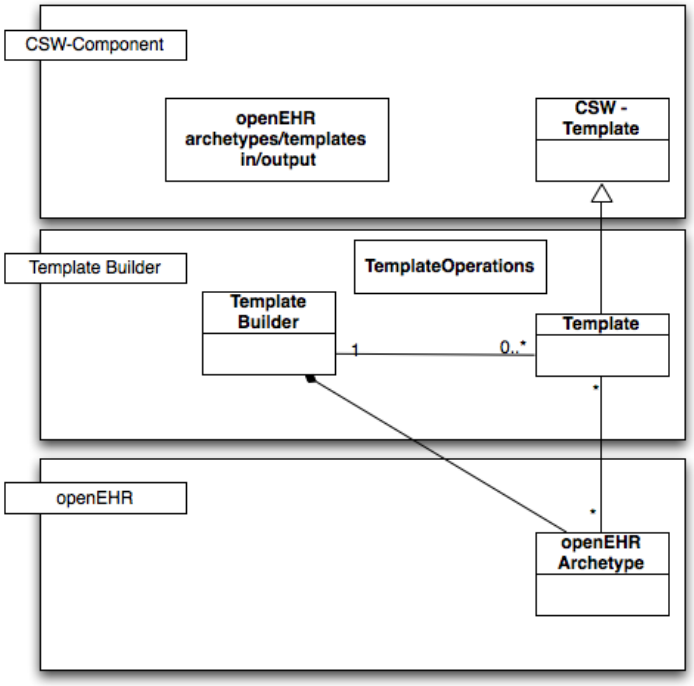

Figure 6.3: Model detailed in a UML class diagram, with classes clearly separated by components. Some *classes* may correspond to multiple classes in the actual implementation.

Template class inherit Template from CSW-Component (named **CSW-Template**). This **Template** class contains the implementation of *openEHR* templates (as said, inherited from *CSW-Template*), as well as some specific attributes and methods to operate in it because of the linkage between them and ZK component. These methods include ways to easily define template's properties, or operations to be made in templates that have match the UI element with the model element. These operations are actually contained inside **TemplateOperations**, and include:

- **Adding operations**, in which an archetype is added inside a template in a specific location according to the information passed from ZK view. It is important to understand that these archetypes are actually references to archetypes present in *Template Builder*. In fact, and as explained in Source 2.1, an OET template only contains a reference to an archetype that exists in a repository.
- **Removing operations**, in which an archetype is removed from the template in a specific location also passed from the view.
- **Rules operations**, in which *Rules* that override original archetype's node attributes, as explained in § 2.4.2 (p. 22). This is also done according to the information passed from the view.

That information passed from the view is interpreted and these operations are made in the correct template nodes. Further details on this information is shown in the chapter that explains the way ZK and *open*EHR are linked (see § 6.5 (p. 64)).

6.3.2 View and Controller

Typically View and Controller are separated, even if they are strongly related. In this particular case, some Views and Controllers are actually merged, giving less complexity to the project. In ZK, GUI elements (XUL or XHTML elements) can be developed either using a ZUL file, in a markup language similar to XML, or either in Java files, using the Java's ZK API. As this is a dynamic application, only the basics of the layout are in ZUL files, which correspond to a typical View layer. Elements present in these ZUL files are then *controlled* in Java files, that act as the *Controller* layer. As these Java files can, in fact, update the UI, it becomes less complex to mix the View and Controller layers within these files, making it easier to reflect what is happening in the UI (user actions), which the model current status, and vice-versa. However, there is a separation in some situations.

View

The View layer of the architecture is composed by ZK and *Preview* components, as said before. *Preview* is the component that takes care of rendering the templates, as a GUI, being built in the Template Builder. ZK is used to build both the main GUI, and the template's corresponding GUI, and to reflect the state of the model. Figure 6.4 represents the View layer, with each component.

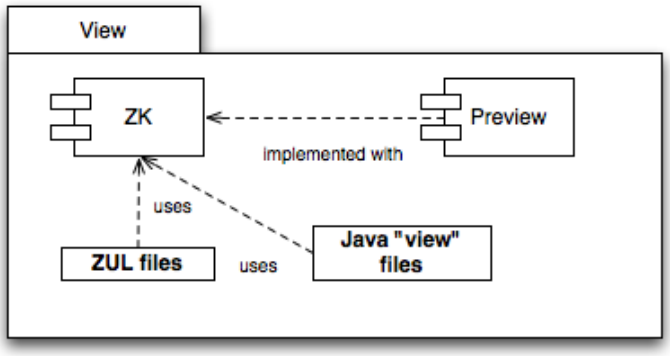

Figure 6.4: View layer of *Template Builder* detailed.

ZUL files contain the ZK markup language to build simple user interfaces. Then, Java controller files contain some aspects related to View, as they build the GUI based on the

system status. With ZK, it is possible to build UIs either using ZUL and Java files, or also by combining them.

Controller

The controller takes care of users thrown events by their interaction with the system. For example, when a user adds an archetype to a template, the controller must handle it and must call the appropriate methods to update the model. Then, that will reflect on the view.

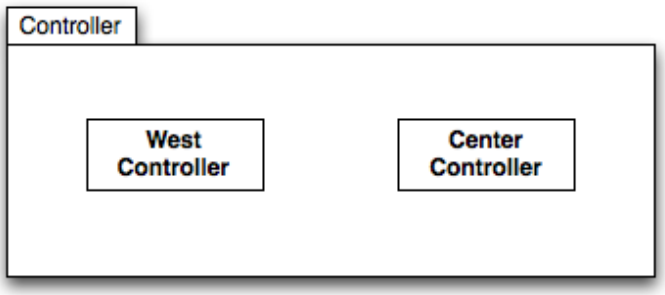

Figure 6.5: Controller of *Template Builder* detailed.

As seen in Figure 6.5, there are two main controllers: *WestController* and *CenterController*. They handle the archetype and template work areas, from where and to where the user can drag and drop archetypes onto templates. UIs are built either using ZUL or Java files. These Java files may be *Controllers*, as they handle events that occur on XHTML and XUL elements defined in the ZUL file. For the author, this specific property of ZK makes it simpler to merge the View and the Controller in most cases.

6.4 Solution Achieved

After explaining the several components that are part of the final system, and after explaining the architecture and how these components fit it in, this section presents the solution in a pragmatic way. This solution was achieved after the studying of this *incompleteness* with Adaptive User Interfaces: or at least a system that allows to (re)configure GUIs for specific purposes. In terms of technologies, the study of *open*EHR allowed to understand it better, in order to handle this complex health standard.

The solution, named *Template Builder*, consists in a component that allows to create *open*EHR template for specific purposes. This is achieved (re)using archetypes, group them, and override or define some proprieties in order to build those templates. These

templates, as explained in § 2.4.2 (p. 22), can take the form of GUIs. So, it is possible to say that this *Template Builder* allows the end-user to create or reconfigure reusable screens, as long as the health professional needs.

6.4.1 Overview

The *Template Builder* is a component for building templates, in a form of a GUI. To do so, it was identified that it was necessary to have two main work areas with different content: (1) the first one should be a *source* for building and add new *features* to the GUI; (2) as the second should consist in the actual working area when the GUI is being built. Passing this to *open*EHR's reality, the *Template Builder* consists on two main work areas: one for archetypes, and one for templates. This allows to have two different contents on each panel. The objective is to have archetype's panel as a source, and to drag them into template's work area to build these templates.

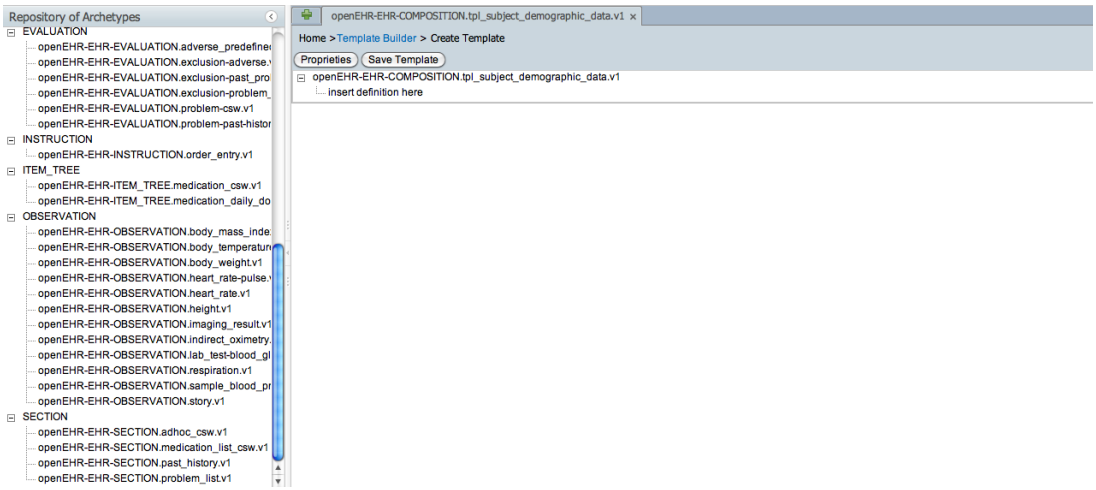

Figure 6.6: Main window of *Template Builder* with two work areas.

Figure 6.6 shows the main window of *Template Builder*, with these two work areas. It uses the *Many Workspaces* GUI pattern, in which multiple panels are used to display context-related but different content [Tid10]. It is also possible to adjust the width, and to even hide archetype's work area (*Collapsible Panels* pattern [Tid10]). Another implemented pattern was the *Center Stage*, in which the template's work area is given the most space and the most visibility on the screen [Tid10]. Note that this might not be the final UI in terms of style.

Next, these two work areas are better explained and detailed.

Archetype's Work Area

The archetype's work area, being controlled by the *WestController*, contains all archetypes grouped by they RM type. They are found on the file system, through a configuration string that corresponds to a system directory. This work area can be filtered, or in other words, it is possible to search for archetypes (see Figure 6.7). It is also possible to filter archetypes that are suitable for a certain *archetype slot* in a template, right-clicking on it and choosing to filter in the archetype's work panel (see Figure 6.8 and Figure 6.9).

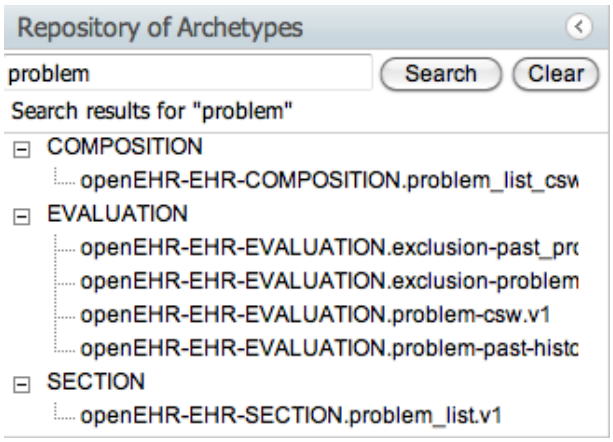

Figure 6.7: How to search for archetypes.

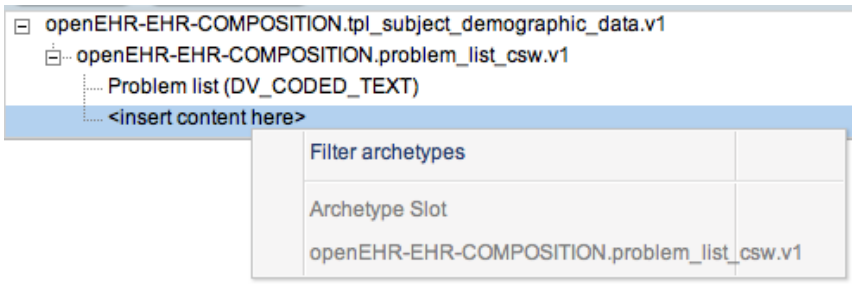

Figure 6.8: Filtering archetypes for the selected *archetype slot*. This will take action in the archetype's work area.

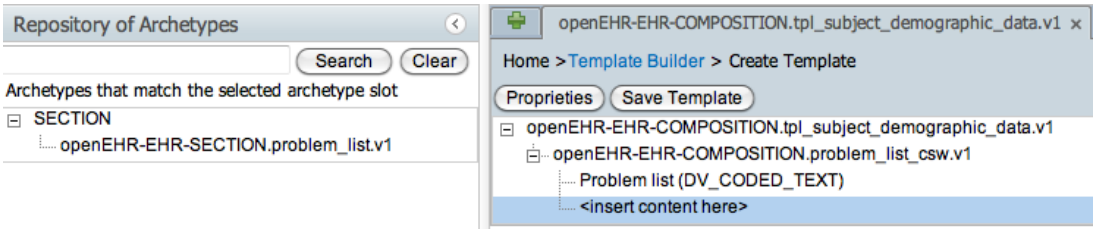

Figure 6.9: The result of the filter for the selected *archetype-slot*.

Template's Work Area

The other main panel is the template's work area, which is controlled by the *CenterController* (and associated classes). In here it is possible to find, one more time, the *Many Workareas* pattern [Tid10], as it is composed by tabs, making it is possible to have more than one template being built using the same session. This pattern that allows some kind of multitasking can be seen in Figure 6.10.

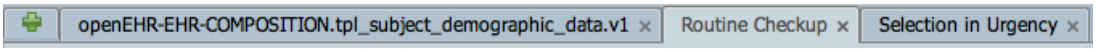

Figure 6.10: The template's work area contains tabs for multiple template building.

Each template work area displays the current template status in a tree form, which represents the internal structure, and each node is represented as a ZK *Treeitem*. *Archetype Slots* are places where new archetypes can be dropped. The system only allows to drop some types of archetypes, depending on what is defined in the Archetype definition. It was not possible to integrate the *Preview* component within this work area. Later in this chapter it is seen how the *Preview* represents the current template as a GUI.

In this work area there is also a possibility of defining template's specific properties, like "name", "ID", and many others, as shown in Figure 6.11 (p. 53). These properties are then updated on the model, and are reflected on the template output file when the user saves it. Source 6.1 shows an example of the OET representation of a template after its properties were defined. More features of these two work areas are explained in the next sections of this chapter.

6.4.2 Building a Template by Adding Archetypes

Templates are built from archetypes, as they are grouped and its properties may be overwritten for specific purposes. This makes the *adding archetypes to templates* one of the key features of the *Template Builder*. *Archetype Slots* represent archetype's nodes in which new archetype's can be placed within a template. In these *slots*, sometimes only some archetypes can be placed, depending on the archetype definition.

The previously presented Figure 6.6 (p. 50) shows the *Template Builder* main interface. When the user wants to add an archetype, he must choose one and drag it into an available *archetype slot* in the template. This drag-and-drop process may have two consequences:

- **The operation succeeds.** If the operation is allowed (Figure 6.12 (p. 54)), the *Controller* updates the model, and the view is consequently updated with the new archetype (Figure 6.13 (p. 54)).

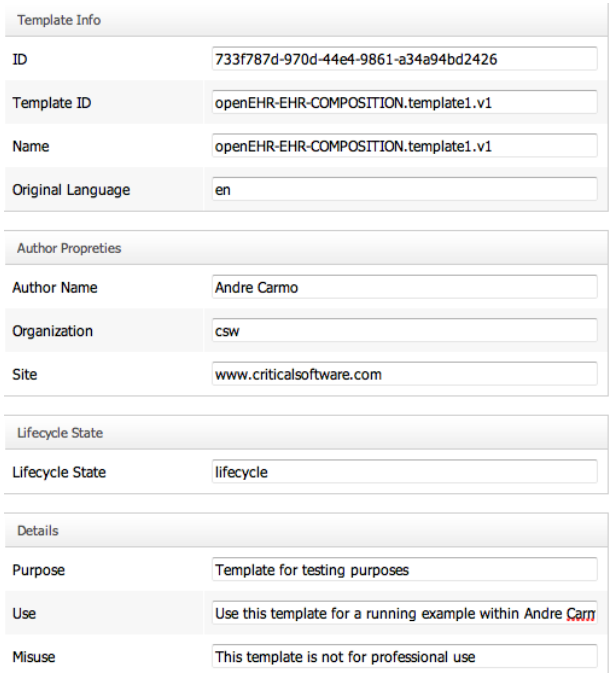

Figure 6.11: Work area for the definition of template properties.

```
    ...
    <id>733f787d-970d-44e4-9861-a34a94bd2426</id>
    <template_id>openEHR-EHR-COMPOSITION.template1.v1</template_id>
    <name>openEHR-EHR-COMPOSITION.template1.v1</name>
    <original_language>en</original_language>
    <description>
        <original_author>
            <item>
                <key>name</key> <value>Andre Carmo</value>
            </item>
            <item>
                <key>organisation</key> <value>csw</value>
            </item>
            <item>
                <key>site</key> <value>www.criticalsoftware.com</value>
            </item>
        </original_author> <lifecycle_state>lifecycle</lifecycle_state>
        <details>
            <purpose>Template for testing purposes</purpose>
            <use>Use this template for a running example within Andre Carmo
               's disseration</use>
            <misuse>This template is not for professional use</misuse>
        </details>
        ...
```

Source 6.1: Example of the template properties in the OET file, according to what is defined in the UI by the end-user.

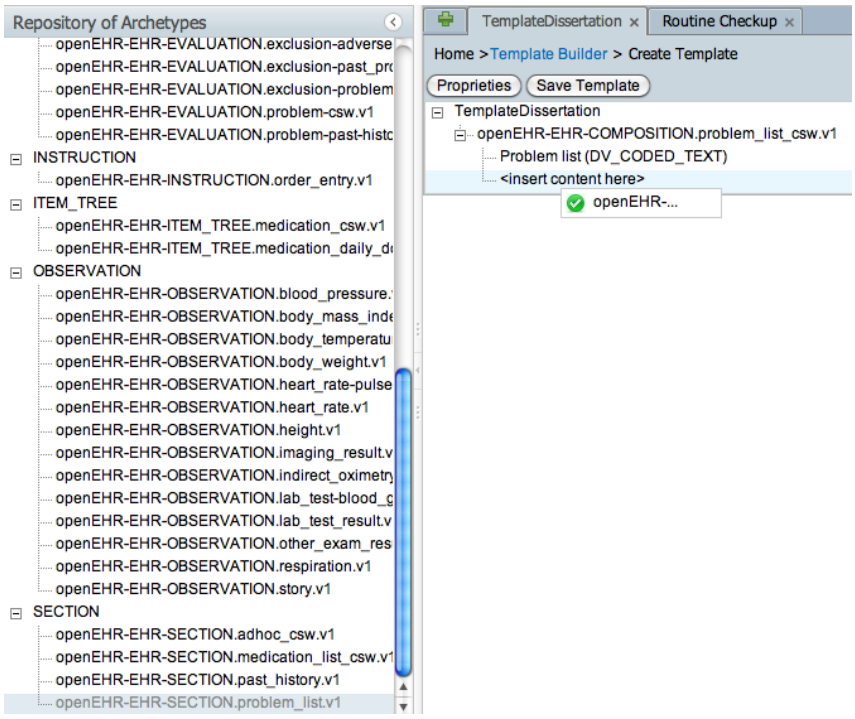

Figure 6.12: The user tries to add a SECTION inside a COMPOSITION. This operation is allowed by the *Archetype Slot* attributes.

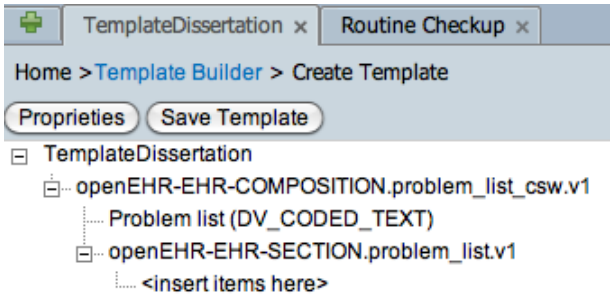

Figure 6.13: The operation shown in the previous figure succeeds, and the View is updated.

- **The operation is unsuccessful.** An unsuccessful operation can happen in two different case: (1) by disabling the possibility of making an operation, when it is detected as not possible (Figure 6.14 (p. 55)); (2) or by an error while updating the model, even if the application allows the operation (see Figure 6.12). When an operation is not allowed by one of these two ways, the View is not updated by the Controller, and a message is shown to the end-user.

The allowed archetypes in a given *Archetype Slot* follow the structure presented in § 2.4.1 (p. 20). However, it could be said the specific archetypes are the only allowed, or even that some are excluded and consequently not allowed there. The operation in Figure 6.12 succeeds because the *Archetype Slot* allows that specific archetype, based on a

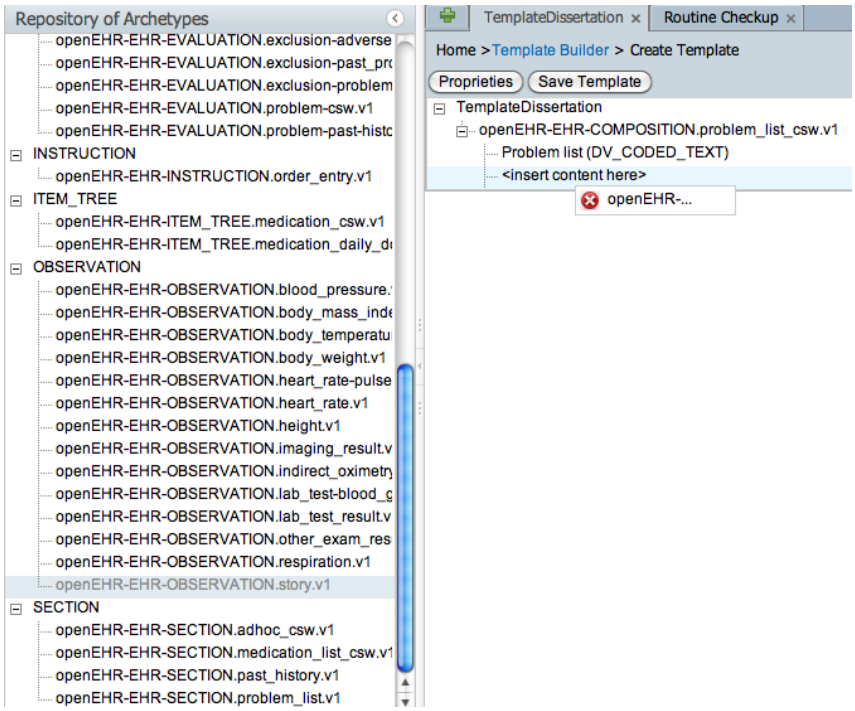

Figure 6.14: The operation fails because the RM type is not allowed there.

regular-expression match, as shown in Source 6.2. Other archetypes cannot be added to that archetype slot. That *path* in that Source represents the path of this *archetype slot*, and it is used to link ZK and the model, as it going to be lately explained.

```
1     ...
2     allow_archetype SECTION[at0001] occurrences matches {0..*} matches {
          -- path: /content[at0001]
3     include
4       archetype_id/value matches {/openEHR-EHR-SECTION\.problem(_[a-zA-Z0-9
            _]+)*\.v1/}
5   }
6         ...
```

Source 6.2: openEHR-EHR-COMPOSITION.problem_list_csw.v1 *archetype slot* representation in the archetype's ADL file.

In some cases, it is possible to have more than one archetype in an *Archetype Slot*. When this happens, another virtual *Archetype Slot* is created in the UI for the end-user to understand that it is possible. Figure 6.15 (p. 56) shows a new one after an add operation.

After these sequences of operations, that ended in Figure 6.15 (p. 56), lets take a look at the template OET file. After the save operation, the file looks as shown in Source 6.5. It is possible to see that the XML-like structure matches what is shown in Figure 6.15 (p. 56): the template is in fact being built.

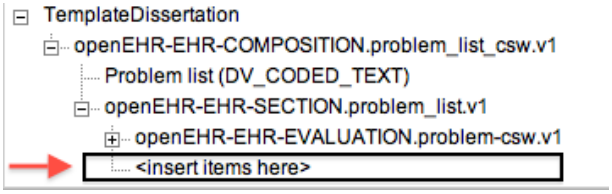

Figure 6.15: A virtual *Archetype Slot* is created after an *add operation* for the end-user to understand that new archetypes can be added here.

```
1    ...
2    <definition xsi:type="COMPOSITION" archetype_id="openEHR-EHR-
        COMPOSITION.problem_list_csw.v1">
3          <Content xsi:type="SECTION" path="/content[at0001]" archetype_id=
              "openEHR-EHR-SECTION.problem_list.v1">
4              <Item xsi:type="EVALUATION" path="/items[at0001]"
                  archetype_id="openEHR-EHR-EVALUATION.problem-csw.v1"/>
5          </Content>
6    </definition>
7    ...
```

Source 6.3: Template OET file after the last made operation.

After showing how these operations are done, it is important to understand the process behind this *add archetype*, and the sequence of operations that take place between some components or entities described in the architecture § 6.2 (p. 45). Figure 6.16 (p. 57) shows this add operation, showing the sequence of information passed between different entities of the system. When an *onDrop* event is detected by the *Controller*, the process described in Figure 6.16 (p. 57) begins.

When the user drops an archetype, the Controller detects that *onDrop* event. Both the treeitem that corresponds to the archetype and the *treeitem* (which is a ZK element) that corresponds to the template node are detected. This allows the Controller to get either the corresponding template and archetype being dropped, which are get as a reference from the *TemplateBuilder* class. After having them, *TemplateOperations* methods for adding the archetype to the template are called, to make changes on the model. Based on what it returns, the Controller decides to update the View to reflect the change that happened, or to warn the user that an error occurred. As it is possible to see, there is a flow between View and Controller, and the model. It starts by an event detection by the Controller, and getting back to the View and Controller, ending in a View change based on the change on the model.

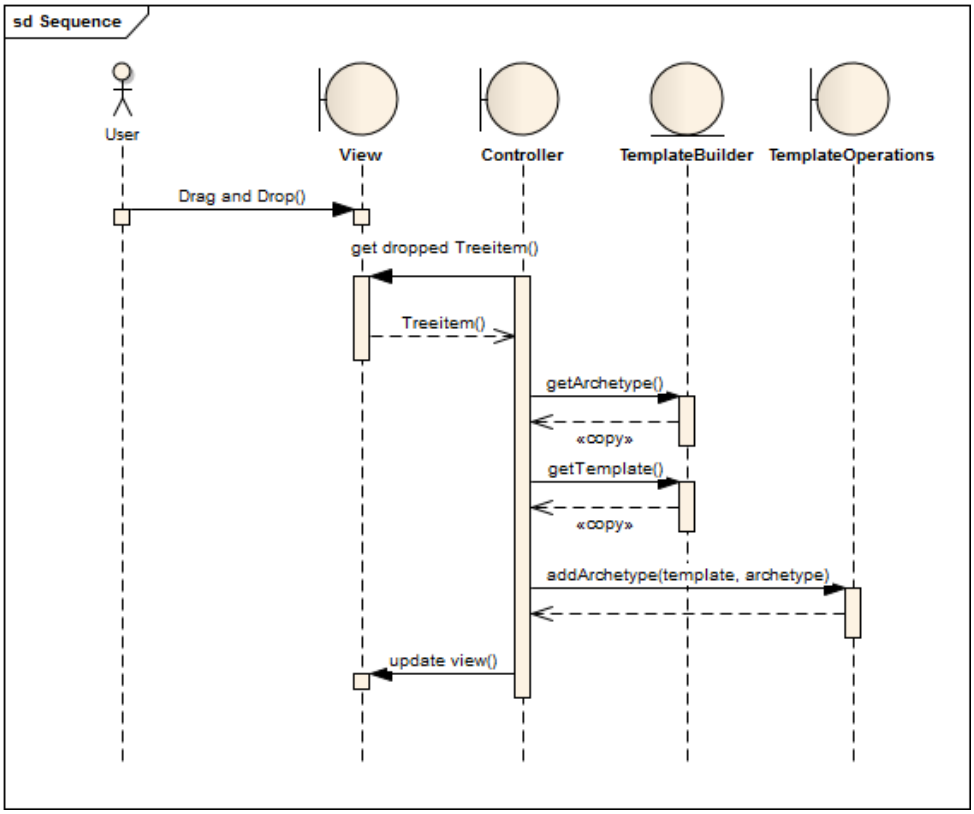

Figure 6.16: The UML sequence diagram represents the flow of information when the user adds an archetype.

6.4.3 (Re)configuring a Template by Setting Rules

Besides this *add operation*, there are other operations that can be done inside templates. As explained, a template can override archetype's attributes of specific nodes to cover specific needs by the health professional. This is done by two ways:

- **Rules.** In this case, a rule is added inside in the template's OET file, which means that a node's attributes of an archetype is being overwritten in the template.
- **Archetype reference attributes.** In case that we are talking about overriding attributes of the actual archetype root, this is done using attributes in the archetype reference in the OET file, and not by *Rules*.

These Rules are added via a context menu, which is built dynamically based on the archetype node that is right-clicked. However, some nodes have no possibility of having rules, like *Archetype Slots* and the template root node. So, it is important to understand which nodes support these *Rules* Figure 6.17 (p. 58). Number **1** represents a template root node, which contains all other nodes. Number **2** represents an archetype root node, or in other words, the root of a referenced archetype. **3** represents a node from an archetype, but

that is not a root node. Numbers **2** and **3** are internally represented as *CComplexObject*s and *CAttribute*s, as explained in § 2.3.2 (p. 18). Finally, number **4** is an *Archetype Slot* node. Only numbers **2** and **3** have can have attributes redefined by *rules*.

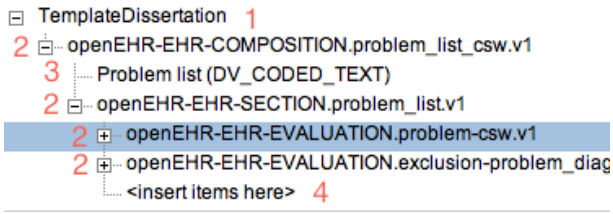

Figure 6.17: A template example with numbers to understand which support *rules*. Only **2** and **3** nodes support them, representing archetype root nodes and archetype general nodes. **1** represents a template node, and **4** represents *archetype slots*.

Rules

One way to (re)configure a template by overriding archetype node's attributes is to create new Rules. These rules override archetype node's (represented by number **3** in Figure 6.17) properties, by only for nodes that aren't the archetype root node itself. This happens because the end-user may want add an archetype, but it may be necessary to override some attributes to match his specific needs. Remember that archetypes are generic and reusable clinical knowledge with the ability of being specified and reconfigured while inside templates. Figure 6.18 (p. 59) shows some possible operations that can be done via context menu.

Figure 6.18 (p. 59) shows cardinality-related operations, or a *Hide On Form* property. This *Hide On Form* means that the end-user wants to hide the archetype's node while the template is rendered in a GUI form. In this case, cardinality related operations (1, 2, 4) means that the end-user wants to set the number of time a node will appear in the GUI. This figure also shows some numbers, that represent the operations that were made to the corresponding nodes. Source 6.4 shows these operations within an OET files.

Besides these operations, there are some nodes that can have different data types, allowing the user to specify it while building the template. The operation presented in Figure 6.19 (p. 59) is also is also reflected in the OET file presented in Source 6.4.

Source 6.4 represents the corresponding OET file after Figure 6.18 (p. 59) and Figure 6.19 (p. 59) were done. It is possible to see that these *Rules* where added inside the *openEHR-EHR-EVALUATION.problem-csw.v1*. They override this archetype's nodes that have the same *path* as shown in this OET file.

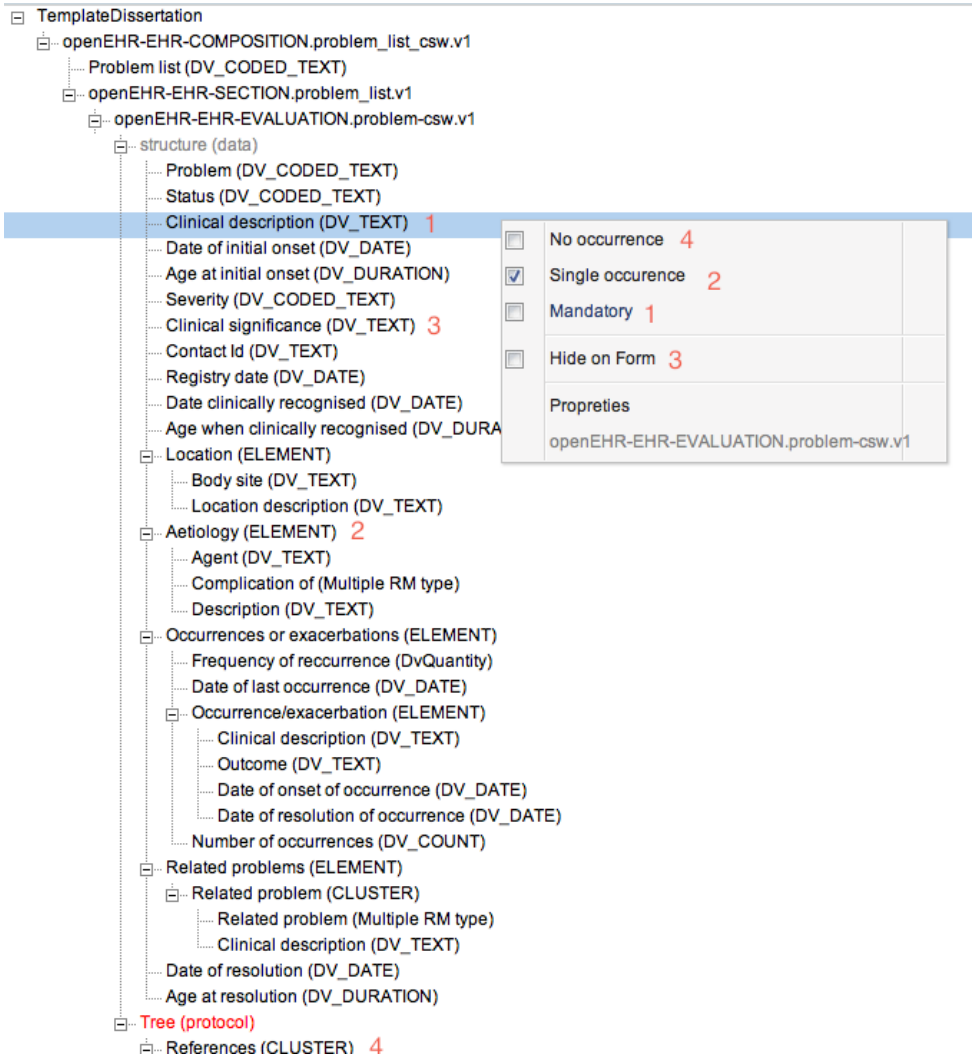

Figure 6.18: Numbers represent the operations made in the corresponding nodes in the template.

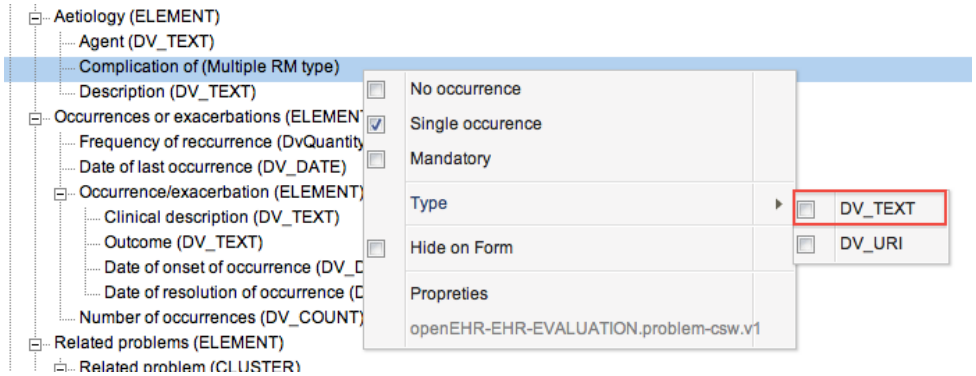

Figure 6.19: Some nodes allow the end-user to choose its data type within the template.

Archetype Reference Attributes

Source 6.5 showed a template OET file. In that source it is possible to see that there is a reference for three templates, which represent the archetypes shown in Figure 6.15 (p. 56).

```
1     ...
2 <Item xsi:type="EVALUATION" path="/items[at0001]" archetype_id="openEHR-EHR
     -EVALUATION.problem-csw.v1">
3       <Rule min="1"
           path="/data[at0001]/items[at0009]"/> <!-- Clinical Descritpion -->
4       <Rule max="1" min="0"
           path="/data[at0001]/items[at0014]"/> <!-- Aetiology -->
5       <Rule max="0" min="0"
           path="/protocol[at0032]/items[at0033]"/> <!-- References -->
6       <Rule hide_on_form="true"
           path="/data[at0001]/items[at0038]"/> <!-- Clinical Significance -->
7       <Rule path="/data[at0001]/items[at0014]/items[at0016]">
8          <constraint xsi:type="multipleConstraint">
9               <includedTypes>DV_TEXT</includedTypes>
10         </constraint>
11    </Rule> <!-- TYPE -->
12 </Item>
13    ...
```

Source 6.4: Template OET file after the last made operation.

When the end-user wants to (re)configure some properties of these archetype's root nodes (represented by number **2** in Figure 6.17 (p. 58)), these new attributes are put within the archetype's reference, like shown in the referred source. Figure 6.20 shows the possible configurations for these root nodes.

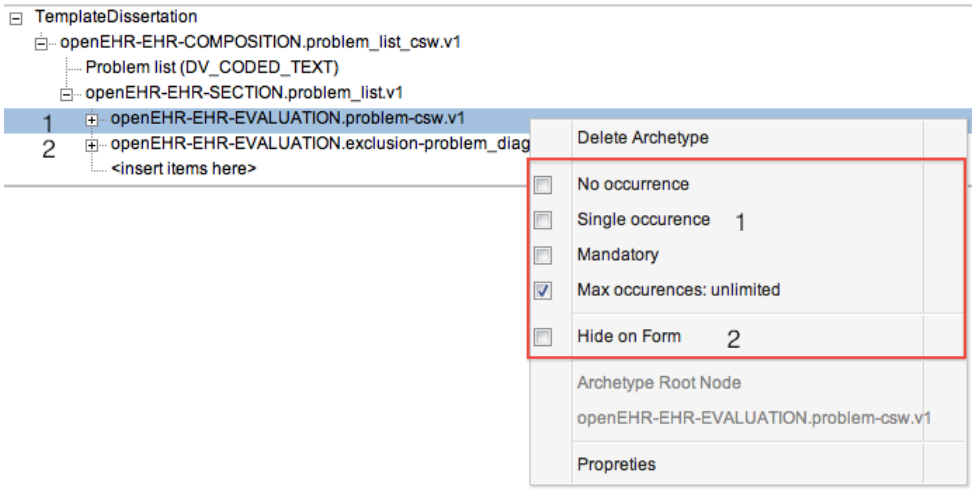

Figure 6.20: Configurations that can be done in an archetype root node. Those numbers indicates the operations made, and to which archetype they were applied to.

These operations can include cardinality-related operations, or a *Hide On Form* property. Cardinality operations are related to *Archetype Slots* maximum and minimum "slots". This possibility of configuration allows the end-user to say that he wants the archetype to appear between a minimum and a maximum number of times, allowing him to decide

when using the GUI. It is also possible to give a name to an archetype, so that it is possible to better understand why it was placed there, inside that template. This is done via *Properties*.

In Figure 6.20 (p. 60) there are some numbers, which means that the operation *Single Occurrence* was applied to archetype with the same number (1), and that *Hide On Form* was applied to the second archetype in that slot.

```
1  ...
2  <definition xsi:type="COMPOSITION" archetype_id="openEHR-EHR-
       COMPOSITION.problem_list_csw.v1">
3     <Content xsi:type="SECTION" path="/content[at0001]" archetype_id=
          "openEHR-EHR-SECTION.problem_list.v1">
4        <Item xsi:type="EVALUATION" max="1"
             min="0" path="/items[at0001]" archetype_id="openEHR-EHR-
             EVALUATION.problem-csw.v1"/>
5        <Item xsi:type="EVALUATION" hide_on_form="true" path="/items[
             at0001]" archetype_id="openEHR-EHR-EVALUATION.exclusion-
             problem_diagnosis.v1"/>
6     </Content>
7   </definition>
8  ...
```

Source 6.5: Template OET file after the last made operation.

These new attributes (in **bold**) were added within the archetype reference inside the template.

To generate the context menu for these rules (shown in Figure 6.18 (p. 59) and Figure 6.20 (p. 60)), the node from the original template is get, and this informations is combined with existing rules or attributes for template nodes. This process is explained in Figure 6.21 (p. 62), where we can see that a node of a **template** has two types of properties, as said before:

- ***Rule*s and root node attributes in archetype references**, depending if the *path* represents a root of an archetype or not.
- **Properties from the node of the original archetype**. Remember that when an archetype is added to a template, only a reference or a *copy* of it is added.

What happens is that when the context menu is built, it analyses the type of node and the builds the data using these two properties explained. *Rule*s have priority over the original node attributes, because rules represent the template (re)configuration over a more general archetype used to build the current template. Also, when the end-user creates a new *Rule*, or reconfigures an archetype, it is not the archetype that is changed:

a new *Rule* or attributes are created or changed. These *Rule*s have a *path* that links them to the original node in the archetype. This process is explained in Figure 6.21.

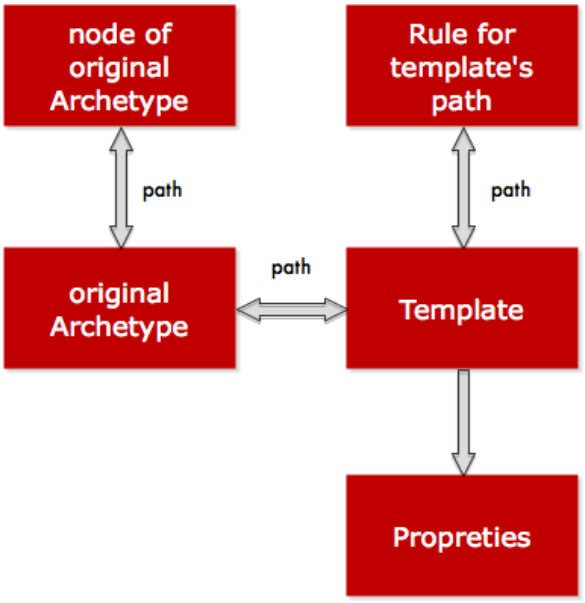

Figure 6.21: How properties are get.

6.4.4 Other Operations

Template Builder contains other operations. These operations include:

- **Delete Archetypes.** It is also possible to delete an archetype from the template, like shown in Figure 6.20 (p. 60). When an archetype is deleted, the view is updated according to the state of the model. This means that new virtual (and visual) *archetype slots* may be created, as shown in Figure 6.22 (p. 63).
- **Clear template.** It is possible to clear a template by selecting its root node.
- **Save and import templates.** The *Template Builder* allows to save and event import existing templates for reconfiguration.

6.4.5 Previewing the GUI that Corresponds to a Template

*open*EHR templates can take the form of GUIs. In fact, it was initially thought to include the *Preview* within the *Template Builder* in order to have a WYSIWYG editor. It was not possible due to time frame problems, but it was possible to integrate an asynchronous PREVIEW. This PREVIEW is a component that is **currently** being developed by *Critical*

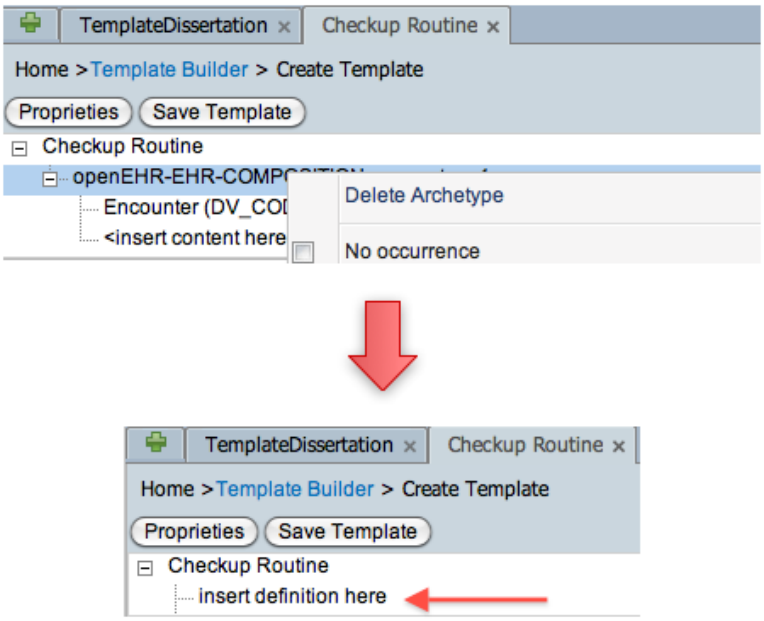

Figure 6.22: An archetype was deleted in *Checkup Routine* template, and a new archetype slot was created in the UI.

Software, SA, so it is expected that not all features are implemented. Figure 6.23 and Figure 6.24 represent the general *Preview* based on the operations made until Source 6.4.

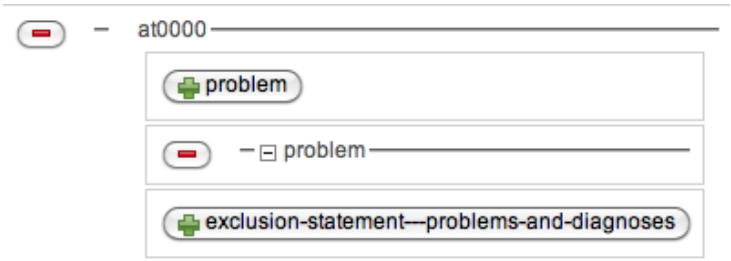

Figure 6.23: Preview (collapsed) of the *TemplateDissertation* until Source 6.4.

Figure 6.24: Preview of the *TemplateDissertation* until Source 6.4. It is possible to see the effect of **mandatory** in *Clinical Description*.

6.5 How are ZK and openEHR Linked?

It is now necessary to understand how exactly ZK and *open*EHR were linked. This is important because in MVC there is a clear separation between the Model and the View, which makes imperative to establish a way of coordination. They way the Model and the View were linked was thought and implemented based on ZK and *open*EHR specific constraints and properties. When the end-user adds an archetype to a template, and when the user edits template's properties, or in any other operation, it is necessary to know which Model object correspond to the View element. And this is done, generally, with identifiers or paths.

When building a template, the first thing to do this is to know which template is the end-user dealing with, as there are more than one tab, and consequently more than one template being built. As ZK elements contain a unique identifier, each template instance from *Template* (see Figure 6.3 (p. 47)) contains the same unique identifier as the tab that is currently selected, allowing to link the template in ZK with the model. Figure 6.25 shows the *uuid* for the templates presented during this running example, and Figure 6.26 shows the templates instantiated.

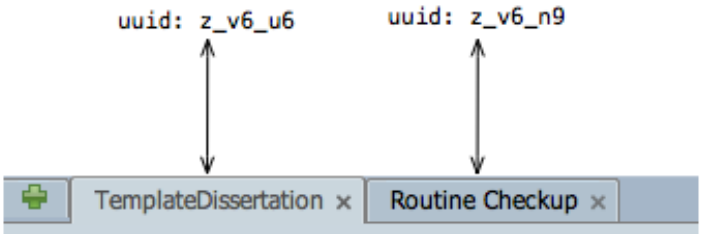

Figure 6.25: Each tab has a unique "ID", which is the same as the correspondent template.

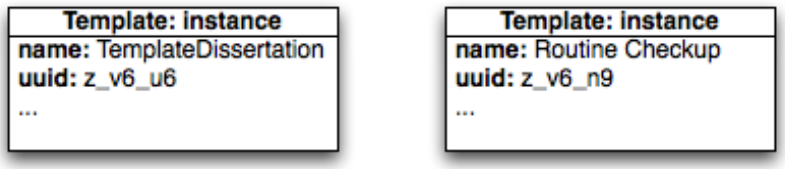

Figure 6.26: Instance of the template. They inherit *openEHR* properties, like name, and have a **uuid** property to link with ZK. Note that **uuid** is the same as shown in Figure 6.25

After knowing which template are the user dealing with, it is necessary to make ZK operations to take effect in the model. When a user drags an archetype into an *archetype slot*, within a template, or when an attribute is set to "mandatory", it is necessary to know to which template node this operation should take effect. This is also done with a

unique identifier, that has the structure shown in Figure 6.27. The path structure shown is unique because of the last part, shown as ZK element `uuid` (remember that each ZK element has a unique identifier). This part is not considered to find the correct node. As seen, archetypes are separated by a **double slash**, and each archetype path has a first element, that is **its name**. When navigating inside a template, this is used to know what archetype should the method deal with. Then, it is removed, and the rest is the real path that is used to make the operation.

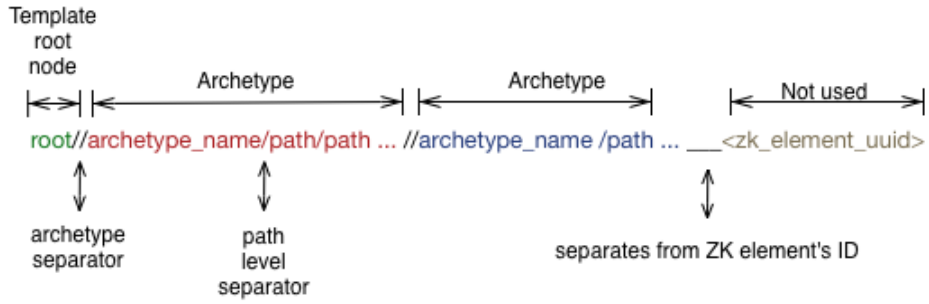

Figure 6.27: Path of an element in ZK.

To better understand how this works, Figure 6.28 shows the same *template* shown in Figure 6.13 (p. 54), with either the **zk identifier**, and **relative node path** (relative to the archetype root).

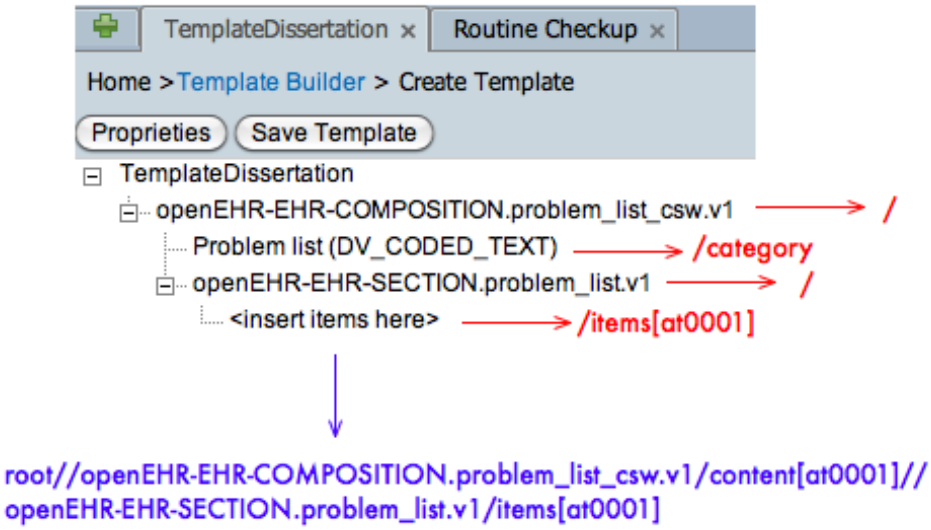

Figure 6.28: Example of a template.

When a user adds an archetype, what happens is that the *archetype slot* ZK's identifier is passed to the method to know the correct path to make changes in the model. After

adding (see Figure 6.15 (p. 56)), **zk's identifiers** stay like shown in Figure 6.29. The *archetype slot* **path**, shown in Figure 6.28 (p. 65), is added as attribute in the archetype reference within the template, as shown in Source 6.6 (in **bold**).

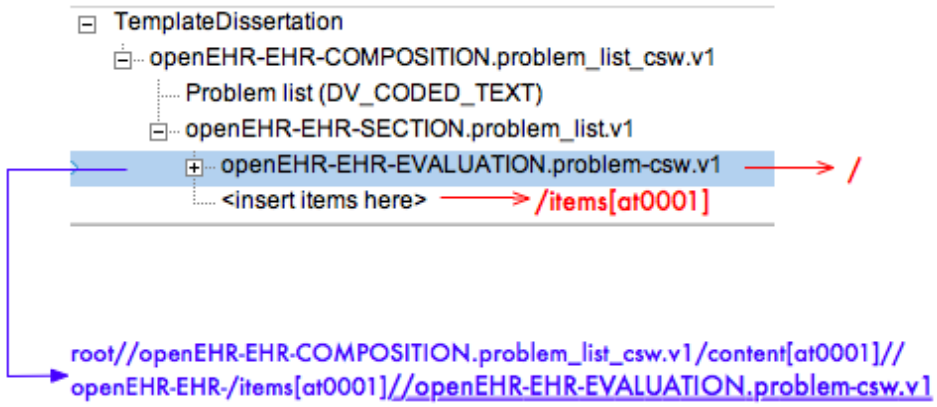

Figure 6.29: Example of a template after adding.

```
1    ...
2    <definition xsi:type="COMPOSITION" archetype_id="openEHR-EHR-
        COMPOSITION.problem_list_csw.v1">
3          <Content xsi:type="SECTION" path="/content[at0001]" archetype_id=
              "openEHR-EHR-SECTION.problem_list.v1">
4               <Item xsi:type="EVALUATION" max="1" min="0" path="/items[
                   at0001]" archetype_id="openEHR-EHR-EVALUATION.problem-csw.
                   v1"/>
5               <Item xsi:type="EVALUATION" extbf{hide_on_form="true"
                   path="/items[at0001]" archetype_id="openEHR-EHR-EVALUATION
                   .exclusion-problem_diagnosis.v1"/>
6          </Content>
7       </definition>
8    ...
```

Source 6.6: Template OET file. In bold there is the *path* of the archetype added, which is the same as the path of *SECTION*s *archetype slot*.

However, there can be one problem with this, and Figure 6.30 (p. 67) explains why. If there are two archetypes with the same name at the same archetype slot, the ZK element identifier is exactly the same. there must be something else that allows us to know which archetype are we dealing with. Having the ZK's element *uuid* in each archetype class was impossible, due to a large number of different archetypes (as shown in Chapter 2 (p. 11). To do so, each ZK element has an attribute called ***pathSlotCounter***, that act as a counter. With this, it is possible to know the exact archetype in an archetype slot, even if they have the same name and the same internal path. Figure 6.31 (p. 67) takes the

example shown in Figure 6.30 to demonstrate how it is possible to distinguish archetypes with the same name within the same *archetype slot*.

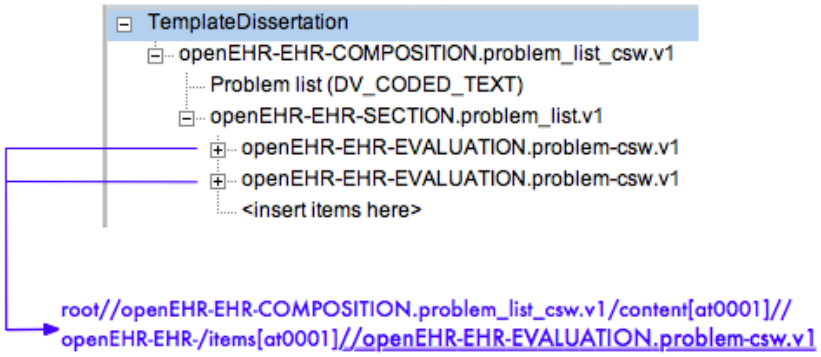

Figure 6.30: Two archetypes with the same name within the same archetype slot will have the same ZK identifier (excepting the last part, as shown in Figure 6.27 (p. 65)).

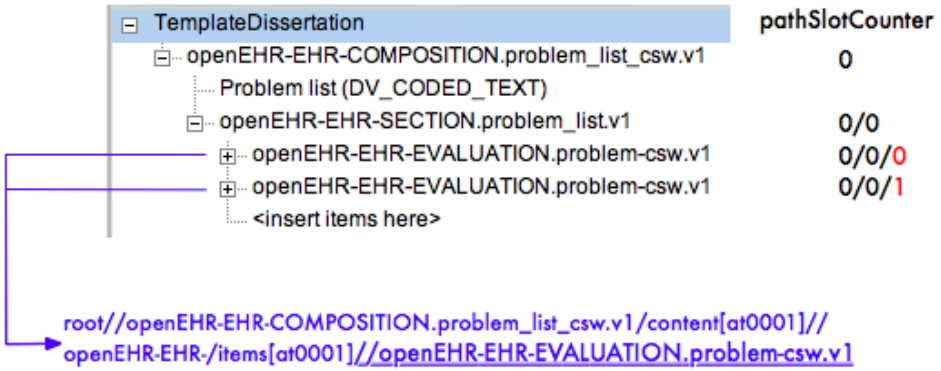

Figure 6.31: How "pathSlotCounter" allows to distinguish from archetypes with the same name.

At the right side it is possible to see the ***pathSlotCounter***, which allows to distinguish between archetypes in the same *archetype slot*. Notice that this counter has levels so that sons can have the correct counter of his parents.

6.6 Reconfigure the system

The *open*EHR specification can be implemented in different programming languages and technologies. This *Template Builder* uses the oficial Java implementation [CK07], which is not completely finished. This implementation lacks of the template's implementation, as well as other aspects of *openEHR*. This lack of *completeness* obligates developers and companies to implement what is left. In this particular case, there was a clear need of

having template's implemented, as well as some utils to handle template's and archetypes from *open*EHR.

To overcome this incompleteness, *Critical Software* implemented many other features that are included in the specification (see [opeb]), but that are not included in the implementation. Besides many others that are not so important to this disseration, template's and its structure were implemented, as explained before, and they are included in the component called **CSW-Component**.

This necessity to implement whats left lead to some differences between the specification and Critical's implementation, due to business constraints related to other projects. So, of course, these differences were a constraint to the *Template Builder* and this dissertation. This dissertation started form the *incomplete by design* and *design for incompleteness* points, particulary related to health systems. So, instead of adapting the system in terms of artifacts, why not designing it to handle this incompleteness, or in this case, these implementation differences? Figure 6.32 shows the differences between the specification and Critical's implementation of templates. As it is possible to see there are many differences, specially related to *Definition* and *Content*. Critical's implementation only holds one *Content* inside *Definitions*, and it is impossible to have one *Item* inside it without having a *Content* in the middle.

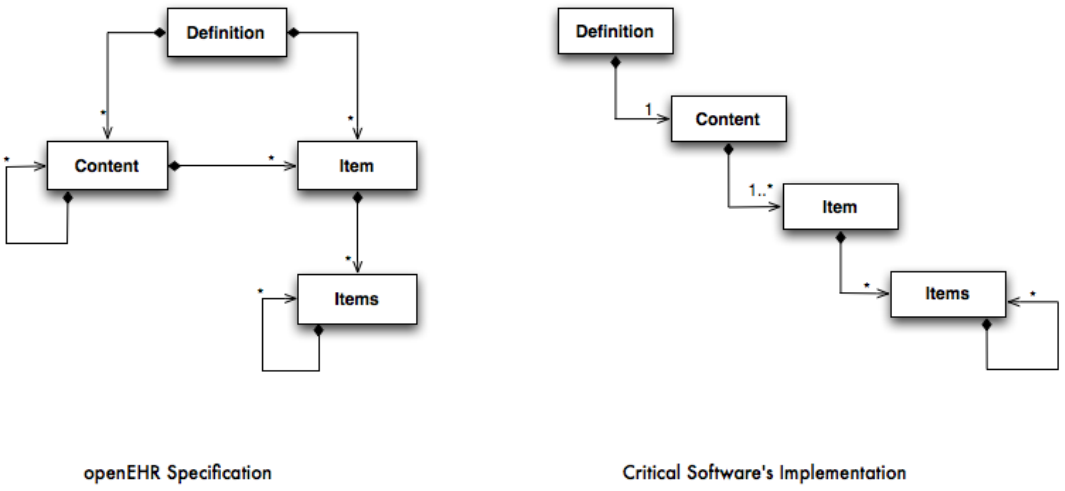

Figure 6.32: Differences between *openEHR* specification and Critical Software's implementation of templates. It is possible to see that there are structural and multiplicity differences.

It is not so difficult to handle these differences. But what if the implementation changes and starts to allow more than one *Content* inside a *Definition*?

To overcome this, a way of changing the system behavior in these situations was implemented, and this was done using an AOM-based approach: a system with rules

specified in a XML, which are interpreted in runtime [HV01]. There is also a metamodel to match the rules and the system state, allowing or not to make certain changes to the template's model based on the specified rules. The main goals of this metamodel are:

- **Deal with different implementations.** In most cases, if the implementation changes, by changing the XML rules file the application still works. This allows to rapidly change the behavior based on different rules, at least in the best case. In the worst case, only minimal changes on the way the API is used needs to be done, because the rest of the system is prepared to handle different behaviors, depending on what is defined.
- **Don't allow operations which may take the system into impossible states.** With this metamodel we can make it impossible to make certain actions just with this XML file.

These main goals are illustrated in Source 6.7. **Bold** attributes represent the main differences shown the UML class diagrams in Figure 6.32 (p. 68). In Figure 6.33 it is possible to understand the consequences of having **numSons** set to **1** or **more**. Moreover, in Figure 6.34 (p. 70) it is possible to see what happens when the second **term** (which is an *Item*) is there, which represents being or not compliant with the specification of *open*EHR.

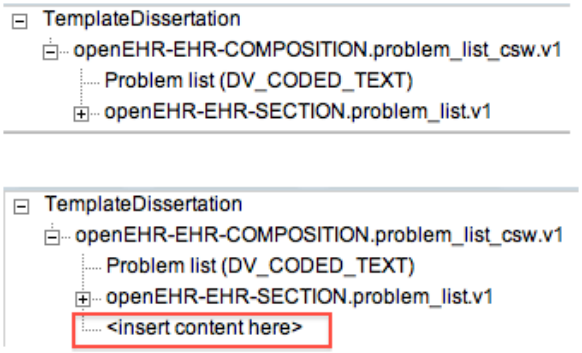

Figure 6.33: Differences between having **numSons** set to 1 and set to more than 1, as shown in Source 6.7. At the top it is possible to see that only one *Content* is allowed inside a *Definition*. At the bottom it is possible to see that having **numSons** set to more than one allows the system to have more than one *Content*s inside. Please note that in some cases some changes need to be done to handle with different APIs implementation, although this eliminates this need or at least diminishes that need.

This *Rule*'s implementation is composed by three main parts, which are explained bellow. These parts are named from **M0** to M3, and the number represents the increase

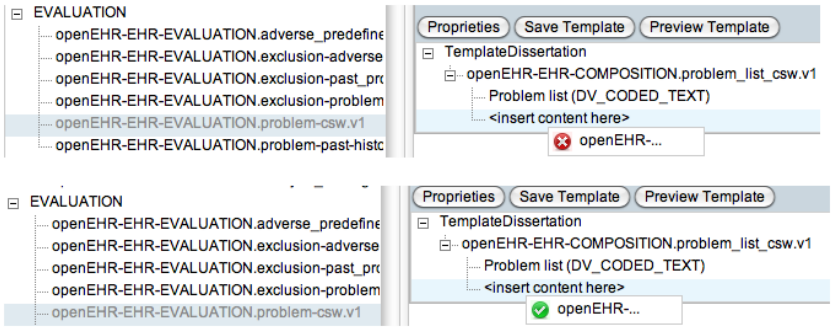

Figure 6.34: Differences between having *Item* **term** inside *Definition*, in Source 6.7. At the top it is possible to see that there it is not possible to add an *Item* inside a *Definition*. In the bottom image that is possible by adding the term in **bold** in that Source.

```
1  <structure>
2    ...
3    <term>
4      <id>1</id> <!-- Definition -->
5      <cardinality>1</cardinality>
6      <numSons>1</numSons>
7      <mandatory>true</mandatory>
8      <compositions>
9        <term> <!-- Content -->
10         ...
11       </term>
12       <term> <!-- Item -->
13         ...
14       </term>
15     </compositions>
16   </term>
17 </structure>
```

Source 6.7: Portion of a XML file with rules related to template's structure.

of abstraction: *M1* represents the data itself, as **M3** is for an abstract representation of the template's structure without *thinking* about the implementation.

- **Model M0.** Represents the existing data, which are template instances.
- **Model M1.** Corresponds to the template's structure that is implemented.
- **Model M2.** Rules specified in a XML file;
- **Model M3.** Represents Metamodel, that is hardcoded. It is though that the main structure of rules and implementation never changes that much.

Rules XML file

The first three models are explained. **M2**, the rule's file, follows a structure like shown in Source 6.7. This Source shows how the template's structure is defined, by indicating the **term**, its attributes, and which **term**s compose them. The attributes include the possibility of setting it to mandatory and to indicate the number of terms that compose it. Source 6.8 shows the typical structure of this rules XML file. There are also to places to define both the *main* terms, and the terms which inherit those main terms. In this source it can be seen that a **Composition** inherits a **Definition**.

Metamodel

The metamodel, **M3**, represents the last model to be presented. It is an abstract representation of the template's structure, in this case. This metamodel is presented in Figure 6.35, as an UML class diagram. As it is possible to see, terms have two different kinds of associative retaliations with other terms:

- **Composition.** This relation indicates that a term is composed by other terms, or in other words, it tells which terms as contained within a term.
- **Inheritance.** This means that a term is inherited by other terms. This is important because, as shown in § 2.4.1 (p. 20), there are archetypes that inherit a *group* of archetypes. For instance, all ENTRY archetypes inside templates inherit *Item*.

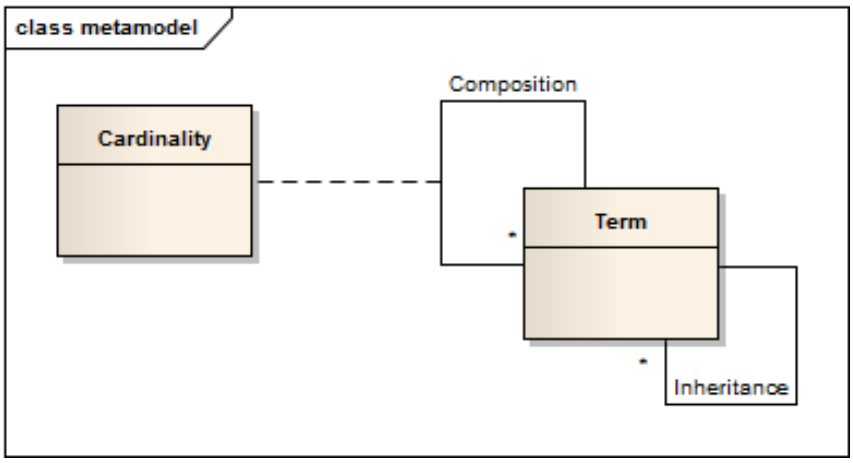

Figure 6.35: Metamodel UML class diagram.

This metamodel allows to prevent unwanted system states, based on the rules. When there is some user action, or even before it happens, the template instance of the metamodel is compared with the rule's metamodel. With the result of this comparison, only allowed

actions take effect. Figure 6.36 (p. 73) shows how this process works. **M0 and M1** represents the template shown in Figure 6.34 (p. 70). In red, it is possible to see the EVALUATION archetype being added to the template. What happens in this specific case is that it checks if the operation is possible agains the instance of rules, **M2**, which were defined in the XML file. According to Critical Software's rules, the operation fails. But according to *open*EHR specification based rules it succeeds (see Figure 6.32 (p. 68) for both rules instances). This is an easy way of changing the system behavior only by changing XML rules.

6.7 Conclusion

This chapter presented the solution for the problem of incompleteness in health systems, particularly related to *open*EHR. Since this standard has templates, which can be mapped to screens, there was the necessity of allowing every end-user to change the GUI towards his needs. To allow that, the *Template Builder* was designed and developed, allowing to create templates from the specialization one or more archetypes. This *Template Builder* follows a Model View Controller architecture, and integrates other components like the *Preview*, allowing the end-user to see the current state of the template being built, mapped to a UI screen. To allow the (re)configuration of templates, some GUI patterns were used in order to give good usability.

The solution found to the problem of displaying to different contents, either archetypes and templates, was found to be the best way to make this happen. There are two different work areas (the *Many Workspaces* GUI pattern [Tid10]), and the user can drag archetypes into templates. This solution allows to keep an eye on the repository of archetypes without loosing the key point: the current state of the template.

Some other problems needed to be solved, including the way ZK and *open*EHR are linked, and the differences between the *open*EHR specification and Critical Software's implementation. Regarding to the linkage between these two technologies, the use of *identifiers* and *counters* allowed to link UI elements to the model. Since there was no information about on how to link them, this is thought to be the best way to do it, since standard attributes of ZK elements provided by its API are used to store those *identifiers*. Related to the different implementations, the lack of a complete official Java implementation showed to be a problem, because developers and companies sometimes make some adaptations to their *world*. It is natural. This differences lead to the necessity of having a set of defined rules that could be interpreted by the system, making it behave like the way it is defined. It was obvious interests regarding to predict impossible system states, based on API constraints, but also to have less hardcoded business rules that could

difficult *open*EHR's API evolvement. The usage of a metamodel to validate the system state according to rules proved to be a good way to make the system change easily, even though not all changes are possible due to APIs constraints.

Also, the problems found and presented in § 7.2 (p. 76) made hard to integrate both the *Preview* and the template editor, essentially due to time-frame issues. However, the *Preview* was, in fact, integrated, allowing the end-user to check the corresponding screen. This *Preview* issue is better explained in the next chapter, which presents the conclusions taken and the future work that can be done.

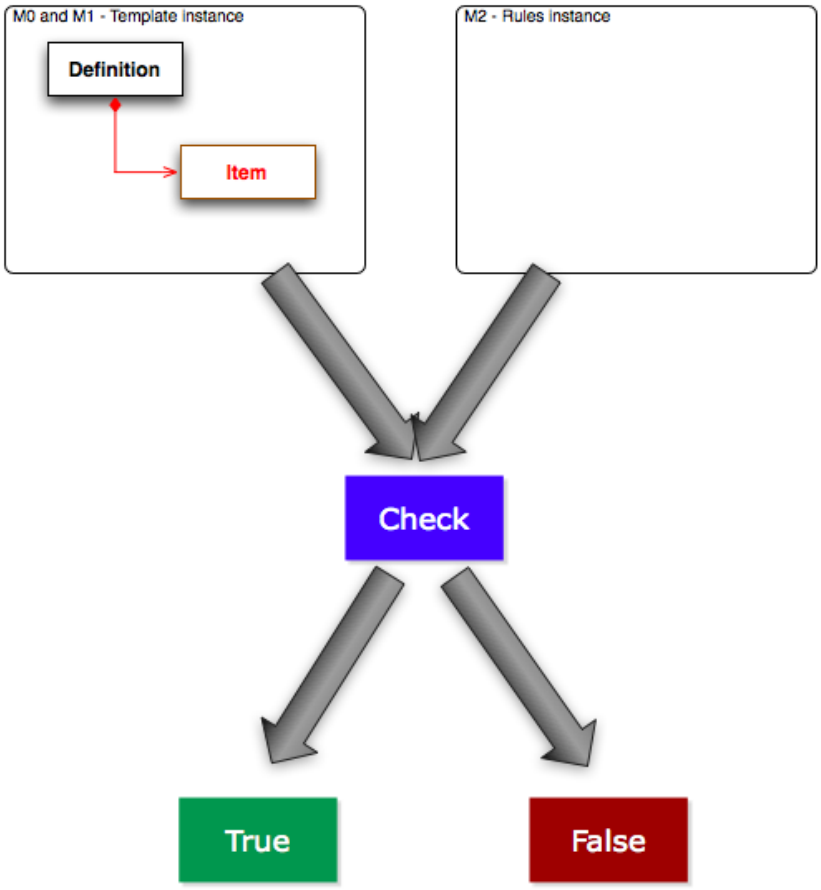

Figure 6.36: How the metamodel checks the template instance with rules instance. It returns true or false if the operation is allowed.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <termsDef>
    <term>
      <id>1</id>
      <name>definition</name>
        </term>
        ...
  </termsDef>

  <inheritanceTermAux>
    <inheritanceTerm>
      <id>1</id>
      <name>composition</name>
      <parent>1</parent> <!-- definition -->
    </inheritanceTerm>
      ...
  </inheritanceTermAux>

  <structure>
    <numSons>1</numSons>
    <term>
      <id>1</id> <!-- definition -->
      <cardinality>1</cardinality>
      <numSons>1</numSons>
      <mandatory>true</mandatory>
      <compositions>
        <term>
          ...
        </term>
        ...
      </compositions>
    </term>
  </structure>
</root>
```

Source 6.8: XML rules file general structure. It contains three main sections: (1) one for the definition of terms; (2) other for the definition of terms that inherit other terms previously defined; (3) and a section for the template structure definition.

Chapter 7

Conclusions and Future Work

7.1 Main Results 76
7.2 Problems occurred during research 76
7.3 Lessons Learned 77
7.4 Future Work 78
7.5 Epilogue 79

Change is, in fact, a major factor that influence software development, and sometimes this *change* is complemented by the *incompleteness*: not only the environment or the domain to be modeled can change, but it can also be difficult or even impossible to predict. And that is what happens in the health area, where there are many different necessities that are difficult to predict. This dissertation studied the best ways to cover all the necessities, particularly, but not only, related to GUIs.

Critical Software had a running project in which they used some technologies presented in this dissertation, like *open*EHR and ZK. ZK's templates can take the form of GUIs, but they have to be built somehow. Since they can be defined using a XML-like syntax, every one with experience in XML and *open*EHR can build them. But what about the end-user, typically a health professional? Or even an engineer that wants to build it fast? A software component for this purpose would cover this *incomplete by design* property of health systems, in particular when based on *open*EHR, as the end-user could build its own screens whenever he needs.

Some software exists to that, but why it is not a solution for this problem? Critical Software wanted this new component to completely integrate with its system. Also, because the Java official implementation does not cover all the specification, there are some differences that make it impossible to use the existing solution. And because this is

a Master dissertation it is also wanted to study the best way of implement this kind of software components.

7.1 Main Results

According to the main problem, a solution was developed. As long as it was being developed, the problem was redefined, ending up with some new minor problems. This iterative methodology lead to the main results and contribution of this dissertation:

- **Development of the Template Builder.** The main contribution of this dissertation was, in fact, a *Template Builder* to deal with *open*EHR artifacts, especially related to Critical Software needs. This *Template Builder* allows end-users to build their own templates, corresponding to GUIs, according to their specific needs. And they can actually do it just in time. Also, this GUI creator implements the AUI concepts, having a *Preview* to show the current screen state.
- **Contribution to *open*EHR.** The *Template Builder* contributes to the *open*EHR standard, as there is not much study on template builders.
- **How to best integrate *open*EHR with zk.** This achievement was important to this dissertation as there are no records of integration between *open*EHR and ZK. These two technologies can fit together, and this dissertation showed how to link UI elements with the model.
- **How to define the system behavior by rules.** This problem and achievement was raised from the problem of having different implementations for some *open*EHR artifacts, like template. A metamodel was developed to validate the system state, allowing, at best, to change the system behavior by changing its rules, instead of hardcoding.

7.2 Problems occurred during research

This research was highly influenced by the time variable. It was conducted between February and June, in an industrial environment, which gave a significant overhead for the author to adapt to company's way of developing software, even though its internal development process was not strictly followed. Instead, it was used a Scrum-like approach, as specified in § 5.2 (p. 39).

This research could be divided into three parts: (1) a **state-of-the-art study**, which started even in November, and ended at the end of this dissertation; (2) the **implementation**, which included some designing, coding and testing and validation; (3) and dissertation writing. The **implementation** phase was what took longer, and that most influenced the result of this dissertation.

While implementing some problems raised, as stated before. However, not all of them could be overcame, and they influenced the research. Those problems are described here.

- **Preview not completed in time.** In software, time delay happens, even if software engineers do all for this not to happen. There are even methodologies and principles for this not to happen, but the truth is that is happens. The **Preview** component was not ready in time, being a real problem in this dissertation direction. The idea of taking AUI concept to the next level, and implementing a WYSIWYG editor, could not be completed. This lead to less focus on AUI, and lead to a direction change later in this dissertation on how to implement the AUI concept and the *Preview*.
- **zk problems.** ZK's version used was 5.0.0, which is not the latest version. This version contains some unresolved bugs, that needed to be worked around. Those problem took some overhead in some crucial phases of the development, and were considered as a risk.
- **openEHR lack of documentation.** Considering the time frame that the author had for this dissertation, the lack of documentation of *open*EHR-related API's introduced a big overhead, and costed precious time during the development.
- **openEHR API specific properties.** Some *open*EHR specific properties were a big problem, either by lack of documentation, as said, or by unimplemented things. This took difficulty to handle *open*EHR, and to later develop a way of connecting the UI (ZK) with it.

All these problems took time in a very tight time-frame for this dissertation. This lead to less evolved system, as opposite to the theoretical study done and demonstrated in this dissertation. In fact, the theoretical study went further than the implementation, for the reasons explained.

7.3 Lessons Learned

During the dissertation period, which included the development of the *Template Builder*, some lessons were learned. These lessons are:

- **Technology problems.** Technologies are always evolving, essentially to enhance features, add new ones, or to correct some existing problems. The usage of non standard and not the latest version of a software may bring issues to the development. It was seen that the fact that ZK's version was not the latest took some problems that needed some workarounds, which introduced some significant overhead in some problematic situations. Also, the fact that the Java official implementation of *open*EHR was not completed was a big problem, because some differences were not predicted. Those problems need to be predicted and assumed as a risk to the project, or to the dissertation in situation, partially when the there is little time to implement the solution.

- **Different time-frames.** One of the key factor for this dissertation was the time variable. In fact, there was a lot of ideas, but there was no time for that. Moreover, there was a delay between the end of the development of the *Preview* component (see Chapter 6 (p. 43)) and the time when the *Template Builder* development started. This was a major problem, which made it impossible to have a WYSIWYG editor. However, an asynchronous way of previewing the screen was introduced within the *Template Builder*. These time related problems have to be predicted and studied before the project starts, especially when the dissertation occurs in an industrial way.

- **Industrial issues.** Some industrial related issues were already shown. Dissertations that take place in an industrial environment gain industrial attributes, and are highly influenced by the culture of the organization. This includes the development process, which may follow some properties that are a bit incompatible with the time-frame for the dissertation. For instance, Critical Software develops critical systems, having a very mature testing phase, which is difficult to balance with the academic and dissertation interests. However, those interests were balanced as possible to provide a robust solution.

7.4 Future Work

Although the most important goals were achieved and completed, some further work can be done. This work include:

- **WYSIWYG editor.** The integration between the *Preview* component and the editor itself is one of the major future working possibilities. The reasons why this was not done during this dissertation were explained before.

- **UI for rules definition.** Currently, rules for change the system behavior are defined in a XML file, without having the possibility of changing them in runtime with an editor. An editor for this would be a interesting future work, taking even further the concept of *embrace change*.
- **Deeply study AUI and GUI design patterns**. With the presence of a WYSIWYG editor, it could be possible to study even deeply the AUI concept, as well as GUI design patterns. Related to GUI design patterns, new patterns could be proposed.
- **Empirical Studies.** Empirical Studies, particularly with health professionals with knowledge with *open*EHR would be a good study in order to improve the quality of this *Template Builder*.

These points are the main future work that can be done nearly, and that could be particularly interesting to this particular industrial case. However, other work could be also done, like improving visual capabilities using more GUI principles.

Even though there is this future work, it is considered that the main goals were achieved, and that the study made and the solution implemented fits the problem proposed.

7.5 Epilogue

This dissertation started from a proposal by Critical Software to implement a software component to build templates. But even though the proposal was as much concrete as possible, the problem was redefined during this dissertation that started by the middle of November, 2010. The author tried not to become stuck in practical assets, but also the research and study even further. Not all studied concepts could be applied, but they were in fact studied.

The way to get to this point was hard: understanding *open*EHR, and its concepts, was certainly the most difficult part of the dissertation. This standard is very powerful, but it also is very complex. Even now, it is not a problem to say that many concepts behind *open*EHR were not completely acquired, even though the key points were deeply studied and understood. The implementation of the solution was not trivial, and took the most of the time. And those problems that were referred did not allow to implement things such as a WYSIWYG. However, the main goals were accomplished, and that resulted in a satisfactory work both for the author and the company. Like in Agile methods, an iterative and constant feedback were a key point to accomplish those goals.

Nomenclature

ADL Acronym for **A**rchetype **D**efinition **L**anguage. It is also an the file extension in which *open*EHR archetypes are defined.

AJAX Acronym for **A**ssyncronous **J**avascript **A**nd **X**ML.

AM Acronym for **A**rcherype **M**odel.

AOM Acronym for **A**daptive **O**bject-**M**odel.

Archetype *Open*EHR artifact that allows to describe clinical concepts.

AUI Acronym for **A**daptive **U**ser **I**nterface.

DOM Acronym for **D**ocument **O**bject **M**odel.

EHR Acronym for **E**lectronic **H**ealth **R**ecord.

End-user In software engineering, it refers to the group of persons who will ultimately operate a piece of software, i.e., the expected user or target-user. [Fer10].

Framework Frameworks are reusable software system components that provide a set of generic functionalities.

GUI Acronym for **G**raphical **U**ser **I**nterface.

HTML Acronym for **H**yper**T**ext **M**arkup **L**anguage.

Metamodel In software engineering, it is a model that interprets a set of rules, checking it against the current system state to verify if it is valid or not.

MVC Acronym for **M**odel **V**iew **C**ontroller..

OET Template It is a template file extension in which archetypes are referenced inside a template. These files follow a XML-like structure.

OpenEHR It is a computer standard for clinical knowledge representation [ope07c].

OPT Template An Operational Template is an expanded template, where archetypes are actually expanded inside the template instead of being only referenced.

Requirement It is a statement that identifies a necessary attributes system in order for it to have value and utility to a stakeholder [Fer10].

RM Acronym for **R**eference **M**odel.

SM Acronym for **S**ervice **M**odel.

Template *Open*EHR artifact that allows to describe specific clinical knowledge, based on the composition of archetypes, specifying their properties. Templates often correspond to UI forms.

UML Acronym for **U**nified **M**odeling **L**anguage [OMG11].

URL Acronym for **U**niform **R**esource **L**ocator.

WYSIWYG Acronym for **W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et editor. It is an editor in which the end-user sees immediately, in the same window, what he is editing.

XHTML Acronym for e**X**tensible **H**yper**T**ext **M**arkup **L**anguage.

XML Acronym for e**X**tensible **M**arkup **L**anguage.

XUL Acronym for **X**ML **U**ser Interface **L**anguage.

ZK Framework for rich web based application that creates an abstraction over the Javascript part of AJAX. The user only has to focus on business logic and GUI.

ZUL Extension for ZK User Interface Markup Language (ZUML) files.

ZUML Acronym for **Z**K **U**ser Interface **M**arkup **L**anguage.

References

[Bar04] David Baronov, *Navigating the hidden assumptions of the introductory research methods text*, Radical Pedagogy (2004). Cited on p. 40.

[Bea07] Thomas Beale, *openehr: a primer*, Slides, 2007. Cited on p. 15.

[Ben] David Benyon, *Adaptive systems: a solution to usability problems*. Cited on pp. 30 and 31.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-oriented software architecture, volume 1: A system of patterns*, Wiley, Chichester, UK, 1996. Cited on pp. 4 and 30.

[CC07] Henri Chen and Robbie Cheng, *Zk™ - ajax withou javascript™ framework*, firstPress, 2007. Cited on pp. 25 and 26.

[CK07] Rong Chen and Gunnar Klein, *The openehr java reference implementation project*, 12th International Health (Medical) Informatics Congress, Medinfo (Brisbane, Australia) (Klaus A. Kuhn, James R. Warren, and Tze-Yun Leong, eds.), August 20 - 24, vol. 129, IOS, 2007, pp. 58–62. Cited on pp. 23, 24, 37, and 67.

[CLG+09] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle, *Software engineering for self-adaptive systems*, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 1–26. Cited on p. 30.

[CNE] CNET, *Bento for mac*, http://download.cnet.com/Bento/3000-2065_4-175260.html [Last checked 22 January 2010]. Cited on p. 33.

[DL99] Don Widrig Dean Leffingwell, *Managin software requirements*, 1st ed., Addison Wesley, October 1999. Cited on p. 39.

[dMA10] Álvaro Jorge Loureiro de Melo Albuquerque, *Estudo de viabilidade de redes sociais como prática colaborativa interna no banco bpi*, July 2010. Cited on pp. 39 and 41.

[ea] Kent Beck et al, *Manifesto for agile software development*, http:agilemanifesto.org/ [Last checked: 10 January 2011]. Cited on pp. 1, 3, and 4.

[EM03] Kweku Ewusi-Mensah, *Software development failures: Anatomy of abandoned projects*, MIT Press, Cambridge, MA, USA, 2003. Cited on pp. 1, 5, and 39.

[Fer10] Hugo Sereno Ferreira, *Adaptive object-modeling - pattherns, tools and applications*, Ph.D. thesis, Faculty of Enginnering - University of Porto, December 2010. Cited on pp. 1, 3, 5, 8, 29, 30, and 81.

[Fil] FileMaker, *Bento 3*, http://www.filemaker.com/products/bento/ [Last checked: 21 January 2011]. Cited on p. 33.

[GG] Andrina Granic and Vlado Glavinic, *Functionality specification for adaptive user interfaces*. Cited on p. 31.

[GJT07] Raghu Garud, Sanjay Jain, and Philipp Tuertscher, *Incomplete by design and designing for incompleteness*. Cited on pp. 2 and 4.

[HV01] Zaijun Hu and Gerhard Vollmar, *Towards xml metamodel patterns for xml data modeling*, Database and Expert Systems Applications, International Workshop on **0** (2001), 0071. Cited on pp. 5 and 69.

[Inf08] Ocean Informatics, *Template designer*, 2008. Cited on p. 32.

[Ins08] SEIRAND Institute, *An introduction to zk*, 2008, http://www.zkoss.org/support/training/webinar/zkintro.dsp [Last checked: 20 January 2011]. Cited on p. 25.

[iSeS] André Carvalho: iTGrow: Software e Sistemas, *Software design*, Internal Document. Cited on p. 4.

[Kru04] Philippe Krunchten, *The nature of software: What's so special about software engineering?* Cited on p. 8.

[Lan] Pat Langley, *User modeling in adaptive user interfaces*. Cited on pp. 30 and 31.

[LMT] Ocean Informatics Lisa M Thurston, *Flexible and extensible display of archetyped data: The openehr presentation challenge*. Cited on p. 37.

[Mah10] Sachin K. Mahajan, *Rich internet applications using zk*. Cited on pp. 25 and 26.

[mic] Cited on p. 45.

[MMRG04] Neil Maiden, Sharon Manning, Suzanne Robertson, and John Greenwood, *Integrating creativity workshops into structured requirements processes*, Proceedings of the 5th conference on Designing interactive systems: processes, practices, methods, and techniques (New York, NY, USA), DIS '04, ACM, 2004, pp. 113–122. Cited on p. 40.

[MS07] Kwak Mi-Sook, *Openehr archetype*, Presentation, October 2007. Cited on p. 15.

[OMG11] OMG, *Unified Modelling Language (UML)*, 2011, http://www.uml.org/ [Online; accessed 13-July-2011]. Cited on pp. 10 and 81.

[opea] openEHR, *The openehr health computing platform*, http://www.openehr.org/201-OE.html [Last checked 13 January 2011]. Cited on pp. 11 and 14.

[opeb] _____, *The openehr health computing platform*, http://www.openehr.org/releases/1.0.2/roadmap.html [Last checked 13 January 2011]. Cited on pp. 23, 44, and 68.

[ope07a] _____, *Archetype definitions and principles*, The openEHR Foundation, 1.0 ed., March 2007. Cited on pp. 14 and 20.

[ope07b] _____, *Architecture overview*, The openEHR FoundationopenEHR, 1.1 ed., April 2007. Cited on pp. 12, 13, 14, 15, 16, 18, 20, 21, and 22.

[ope07c] _____, *openEHR :: future proof and flexible EHR specifications*, 2007, http://www.openehr.org/ [Last checked 14 January 2011]. Cited on pp. 7, 12, and 81.

[ope08a] _____, *The openehr archetype model: Archetype object model*, November 2008. Cited on pp. 19 and 20.

[ope08b] _____, *The openehr reference model: Data structures information model*, August 2008. Cited on pp. 17 and 18.

[ope08c] _____, *The openehr reference model: Ehr information model*, August 2008. Cited on pp. 15, 16, 17, and 18.

[opeon] Technology and architecture challenges in UI implementation. Cited on p. 39.

[PA02] Jussi Ronkainen Juhani Warsta Pekka Abrahamsson, Outi Salo, *Agile software developement methods: Review and analysis*. Cited on p. 3.

[PS01] Jean-François Quaranta Dominique Fieschi Marius Fieschi Pascal Staccini, Michel Joubert, *Modelling health care processes for eliciting user requirements: a way to link a quality paradigm and clinical information system design*, International journal of medical informatics **64** (2001), 129–142. Cited on pp. 2 and 29.

[PSR07] Preece, Sharp, and Rogers, *Interaction design*, Wiley, 2007. Cited on pp. 30 and 33.

[Qui08] Raymond Quivy, *Manual de investigação em ciências sociais*, Gradiva, 2008. Cited on pp. 39 and 40.

[Ram09] Krish Ramachandran, *Adaptive user interfaces for health care applications*. Cited on p. 31.

[RGI75] D.T. Ross, J.B. Goodenough, and C.A. Irvine, *Software engineering: Process, principles, and goals*, Computer **8** (1975), no. 5, 17 –27. Cited on p. 4.

[RY02] Nicolas Revault and Joseph Yoder, *Adaptive object-models and metamodeling techniques*, Object-Oriented Technology (Ákos Frohner, ed.), Lecture Notes in Computer Science, vol. 2323, Springer Berlin / Heidelberg, 2002, pp. 149–175. Cited on p. 5.

[Shu] Gary Shute, *The nature of software engineering*. Cited on p. 8.

[Som04] Ian Sommerville, *Software engineering (7th edition)*, Pearson Addison Wesley, 2004. Cited on pp. 1, 2, 3, 5, and 39.

[Spa11] Sparx Systems, *UML 2 Tutorial*, 2011, http://www.sparxsystems.com.au/resources/uml2_tutorial/ [Online; accessed 13-July-2011]. Cited on p. 10.

[Tid10] Jenifer Tidwell, *Designing interfaces, patterns for effective interaction design, 2nd edition.pdf*, O'Reilly, 2010. Cited on pp. 50, 52, and 72.

[TUI05] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama, *An adaptive object model with dynamic role binding*, Proceedings of the 27th international conference on Software engineering (New York, NY, USA), ICSE '05, ACM, 2005, pp. 166–175. Cited on p. 30.

[vGBS01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg, *On the notion of variability in software product lines*, WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society, 2001, pp. 45–54. Cited on pp. 7 and 30.

[web] *Webster's online dictionary*. Cited on p. 4.

[Wik] Wikipedia, *Scrum (development)*. Cited on p. 41.

[ZK] ZK, *Zk essentials for zk 5*. Cited on pp. 25, 26, and 45.

[ZW98] Marvin V. Zelkowitz and Dolores R. Wallace, *Experimental models for validating technology*, Computer **31** (1998), 23–31. Cited on p. 42.