UNIVERSITÀ DEGLI STUDI FIRENZE

Faculty of Engineering
Department of Information Engineering

Ph.D. in Informatics, Systems and Telecommunications
Cycle XXVII

Curriculum: Telematics and Information Society

Coordinator: Prof. Luigi Chisci

# Model and framework for multimodal and adaptive user interfaces generation in the context of business processes development

ING-INF/05

| | | |
|---|---|---|
| *Ph.D. Student* | *Tutor* | *Coordinator* |
| Ivan Zappia | Prof. Dino Giuli | Prof. Luigi Chisci |

*Tutor*
Dr. Federica Paganelli

Years 2012/2014

# Acknowledgements

I would like to express my gratitude to my tutors Professor (and mentor) Dino Giuli and Dr. Federica Paganelli for encouraging my research and for allowing me to grow professionally and personally. I would also like to thank Dr. Paolo Cianchi and Dr. Leonardo Landi for giving me the opportunity to be a Ph.D. student on such an interesting research topic while working at Negentis and for all the stimulating discussions on themes both addressed in this work and not.

I also want to thank my cotutor Dr. Lucia Ciofi for her invaluable support and for still being my friend after a so stressful and intense final month. Moreover, I am grateful to Professor Monica Gherardelli for her great understanding.

A special thanks goes to all the colleagues and workmates who are actually more friends than anything else and contributed to create an enjoyable work environment.

Last but not least, I want to thank my parents who supported me during this long journey.

Firenze, 31 Dicembre 2014

Ivan Zappia

*"Back off man, I'm a scientist."*

**Dr. Peter Venkman**

# Abstract

This thesis addresses issues related to the automatic generation of user interfaces, in order to identify methods to effectively support the constant evolution of processes and, at the same time, to put the emphasis on fundamental concepts for the user such as usability, plasticity, adaptability and multi-modality. The proposed methodology foresees the definition of a set of four meta-models for the design of the various aspects of both the UI and the application development processes, with the definition of the different users involved in the different steps and the indication of the models produced at the end of each step of the process; said methodology is synthesized in a specific framework covering both the design and runtime phases of multimodal adaptive UIs and application life cycles, thus embracing a more holistic model-driven approach foreseeing the integration of methods for the automatic generation of user interfaces with the tools used for business process management. In particular the framework envisions: a Domain Model, to represent all the concepts characterizing each application domain; a Process Model, to represent the tasks fulfilling the application requirements; an Abstract User Interface Model, automatically generated from the two previously introduced models and describing any possible user interface generated for the specific use case; a Concrete User Interface Model, automatically generated from the abstract model and describing the family of concrete user interfaces for a specific use case once a particular target technology has been chosen.

# Contents

**Introduction** **vii**

**I State of the Art** **1**

**1 Approaches to User Interface Development** **2**
1.1 Historical overview: UI Management Systems . . . . . . . . . 3
1.2 Model-Based UI Development . . . . . . . . . . . . . . . . . . 4
1.2.1 Different Generations in MBUID Systems . . . . . . . 6
1.2.2 Cameleon Reference Framework . . . . . . . . . . . . . 7
1.3 Model-Driven Software Development . . . . . . . . . . . . . . 10
1.3.1 Model-Driven UI Development . . . . . . . . . . . . . . 12
1.3.2 Adaptive Model-Driven UI Development . . . . . . . . 14

**2 Languages for User Interface Development** **18**
2.1 User Interface Description Languages . . . . . . . . . . . . . . 19
2.1.1 USer Interface eXtensible Markup Language . . . . . . 20
2.1.2 Useware Markup Language . . . . . . . . . . . . . . . . 20
2.1.3 Dialog and Interface Specification Language . . . . . . 21
2.1.4 ConcurTaskTrees . . . . . . . . . . . . . . . . . . . . . 22
2.1.5 MARIA . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.1.6 User Interface Markup Language . . . . . . . . . . . . 23
2.1.7 eXtensible Interface Markup Language . . . . . . . . . 24
2.2 User Interface Transformation Languages . . . . . . . . . . . . 24
2.2.1 Graph Transformations . . . . . . . . . . . . . . . . . . 25
2.2.2 Atlas Transformation Language . . . . . . . . . . . . . 25

2.2.3 TXL Transformation Language . . . . . . . . . . . . . 26
2.2.4 UIML Transformations . . . . . . . . . . . . . . . . . . . 26
2.2.5 XSL Transformations . . . . . . . . . . . . . . . . . . . . 27

**II Proposed Solution 28**

**3 Approach Overview 29**
3.1 The representation models . . . . . . . . . . . . . . . . . . . . 31
3.1.1 Domain Model . . . . . . . . . . . . . . . . . . . . . . . . 32
3.1.2 Process Model . . . . . . . . . . . . . . . . . . . . . . . . 32
3.1.3 Abstract User Interface Model . . . . . . . . . . . . . . . 33
3.1.4 Concrete User Interface Model . . . . . . . . . . . . . . . 34
3.2 The proposed framework . . . . . . . . . . . . . . . . . . . . . 34

**III Framework Design 39**

**4 Domain Model 40**
4.1 Domain elements description . . . . . . . . . . . . . . . . . . . 41
4.1.1 Concepts . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.1.2 Properties . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.1.3 Property functions . . . . . . . . . . . . . . . . . . . . . 42
4.1.4 Attributes . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.1.5 Relationships . . . . . . . . . . . . . . . . . . . . . . . . 44
4.1.6 Access control lists . . . . . . . . . . . . . . . . . . . . . 45
4.1.7 Persistence . . . . . . . . . . . . . . . . . . . . . . . . . . 46
4.2 Formal representation . . . . . . . . . . . . . . . . . . . . . . . 46
4.3 Instances Model . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.3.1 Buckets and Records . . . . . . . . . . . . . . . . . . . . 49
4.3.2 Bucket filtering . . . . . . . . . . . . . . . . . . . . . . . 50
4.3.3 Notes on persistence . . . . . . . . . . . . . . . . . . . . 51

**5 Process Model 52**
5.1 Process elements overview . . . . . . . . . . . . . . . . . . . . 53
5.1.1 Activities . . . . . . . . . . . . . . . . . . . . . . . . . . 54
5.1.2 Blocks . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
5.1.3 Signaling . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

Signals for transition triggering . . . . . . . . . . . . . 57
Block to block signaling . . . . . . . . . . . . . . . . . . 58
Activity to block signaling . . . . . . . . . . . . . . . . . 59
5.1.4 Distributed workflow . . . . . . . . . . . . . . . . . . . . 59
5.1.5 Access control lists . . . . . . . . . . . . . . . . . . . . . 60
5.2 Formal representation . . . . . . . . . . . . . . . . . . . . . . 61

**6 Abstract User Interface Model 63**
6.1 Model generation . . . . . . . . . . . . . . . . . . . . . . . . . 64
6.2 Abstract UI Model elements . . . . . . . . . . . . . . . . . . . 66

**7 Concrete User Interface Model 68**
7.1 Model generation . . . . . . . . . . . . . . . . . . . . . . . . . 69
7.2 Basic mapping . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

**8 Concrete User Interface 72**
8.1 The client device . . . . . . . . . . . . . . . . . . . . . . . . . 72
8.2 Widgets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
8.2.1 Widgets architecture . . . . . . . . . . . . . . . . . . . . 74
8.2.2 Low-tech clients . . . . . . . . . . . . . . . . . . . . . . . 77
8.3 Widgets composition . . . . . . . . . . . . . . . . . . . . . . . 78

**IV Implementation 80**

**9 Demonstrator 81**
9.1 Demonstrator modules . . . . . . . . . . . . . . . . . . . . . . 81
9.1.1 Domain repository . . . . . . . . . . . . . . . . . . . . . 82
9.1.2 Domain services . . . . . . . . . . . . . . . . . . . . . . . 82
9.1.3 Process repository and workflow engine . . . . . . . . . 84
9.1.4 Text-based user interface client . . . . . . . . . . . . . 86
9.2 Implemented framework features . . . . . . . . . . . . . . . . 87

**Conclusions 90**

**Bibliography 95**

# Introduction

The user interface (UI) layer is considered one of the key components of software applications since it connects their end-users to functionalities. Well-engineered and robust software applications could eventually fail to be adopted due to a weak UI layer [ABY14]. Since the beginning of the personal computing era, a growing interest and a continuous evolution have involved user interfaces; new paradigms have been usually paired with the launch of new input peripherals. At first there were cryptic textual interfaces with the keyboard as the only input device, then graphical interfaces and the mouse; more recently touch and vocal interfaces have been introduced in everyday use. A continuous evolution has contributed to the diffusion of "post-PC devices" to a wide range of users, from tech enthusiast, early adopters and users with an high technological skill level to average people without any prior knowledge in such domain.

The interest has not ceased and the search for new kind of interfaces is still fully active: as an example haptic interfaces are starting to gain attention in the hand-held sector for its adoption by widely appealing commercial products (e.g., the soon to be commercialized Apple Watch).

In the recent years, as a consequence of the great proliferation of mobile devices with different form factors, the same application is required to adapt to different characteristics like, for example, screen resolution and orientation; a concept called *plasticity* [CCT$^+$02]. The approach "one design fits all" [ABY14] has been proved inadequate to answer this call and was abandoned in favor of more adaptive techniques (e.g., responsive web design in the case of graphical web interfaces). Moreover the multiplicity of devices adopted by a single user (e.g., desktop personal computer, laptop, smartphone, tablet) needs to address more advanced behaviors such as, for example, the seamless migration of the application status from a device to another; moreover, it must be kept in mind that each device could operate with different technologies (e.g., graphical, tactile, vocal) so the UI, or its provider, should implement a multimodal approach.

Also new needs for user experience improvements have arisen. First of all there is high demand for more user-friendly designs: a wide public, also composed by users without a technical background, needs interfaces intuitively easy to understand and exploit without the need for a manual or an extensive onboarding phase; this concept is called *usability.* Users diversity also implies that one interface can't possibly satisfy all their needs; users have in fact different necessities, capabilities or impairments and require adaptive interfaces able to change on the base of different user profiles. Such user profiles can't be static but have to change as the context of utilization of the interface changes; as an example the UI should be different whether the user is relaxing on the sofa or is running to catch a bus or just walking in the park. The interface could also adapt to user emotions implementing traits of *affective computing* [Pic00].

Solving all these different problems in an efficient and effective way implies a great effort in terms of costs, time, complexity and the need for ongoing maintenance to keep the pace with a sector in constant evolution. All these aspects are specific to the user interface field and these themes have been usually addressed independently of the rest of the interactive system they are actually part of [MPV11]. Research has paid attention to such issues and with the intent of dealing with them has moved in two main directions: the first one is to focus on the automatic creation of a UI model derived from a formal description of the application tasks (i.e., a formal description of *what* the application has to do), the *tasks model* [GLCV08, EVP01] or *discourse model* [PRK13, PSM+08]; in the second one the focus is on the direct design of the UI model taking into account the application requirements. Both approaches, usually adopting transformational techniques [PFRK12], share the common goal of creating a UI model that can be used to generate the concrete UI model used to instantiate the actual interface [MPV11]. Many domain specific languages were designed for this purpose [MDZ13, BCPS04].

These UI models are intended to represent the entire interface with all its aspects and this can lead to a method, whether automatic or not, of great overall complexity. The resulting interfaces are well structured but poor in terms of usability and general appearance [PRK13]. This latter aspect is traditionally of secondary importance for academics more involved in resolving technical and scientific problems.

On the other hand organizations and companies that design and produce the available software development tools, have only partially adopted the results of the previously introduced researches due to the already explained and still unsolved problems. On the contrary, aesthetic quality is of primary importance on the market [HPBL00] and UIs underwent great enhancements from that standpoint following the wide diffusion of smartphones and tablets; this is due both to the high activity and competitiveness in the sector and to the new capabilities and the always increasing computational power of currently available commercial devices. Consequently the end user expectations have also increased and high quality UI design has become not only essential but also expected.

The most part of UI developers usually still work with a very different technique: custom interfaces are built on top of frameworks; these are supplied in the form of libraries and toolkits (e.g., JavaScript frameworks for web development) or as part of more complete Software Development Kits (SDK) that, for their own nature, results in solutions tightly tied to the addressed target platforms. In either case these tools provide simple building blocks that are used to assemble and control the developed user interfaces. For example, in the mobile operating systems world, all big players (i.e., Apple with iOS, Google with Android and Microsoft with Windows Phone) adopt this very philosophy: the SDK represents the principal, and often only, way to develop full featured applications for their platforms; different languages are employed to write business logic and implement user interfaces making use of the provided UI elements, called *widgets*, in accordance to the classic Model-View-Controller architectural pattern [GHJV95]; said languages are all third-generation programming languages (3GL) [Tha84] so the style adopted is primarily imperative, although object-oriented and therefore are employed to describe *how* everything should work. Also the widgets are integrated with the same programming style and, starting as simple presentation and interaction devices, must be configured and supported by the application control logic. It is also possible that the provided widgets are not suitable for the particular use case and new custom elements have to be created and used.

A really important component of the SDK is represented by the documentation that, among other things, comprises platform specific guidelines the developers have to comply with in order to build applications well integrated, also aesthetically, with the chosen platform and with good levels of usability and accessibility. In this task the Integrated Development Environment (IDE) can only support, even though with really advanced and effective tools in some cases, the developer's work which is substantially left alone with his competence and expertise.

Compared with the aforementioned research approach, the most part of UI developers and companies usually still adopt very different techniques; notwithstanding it appears clear that it isn't favorable at the moment to change such an approach because it lets the developers free to exploit their personal abilities in the production of custom but usable, accessible and appealing UIs in accordance with application level requirements. Obviously such an approach encounters all the limits that the automatic UI generation wants to overcome.

To follow this path, a new widget class is required: the present work proposes the adoption of *complex widgets* that encapsulate the required UI features such as usability, accessibility and context awareness. This implies that said widgets have to work as small applications instead of simple UI components and therefore a technique is needed to compose them in order to shape applications complying with functional and non-functional requirements. Said composition is achieved through the adoption of a framework designed to combine the generative research approach with the widely adopted custom development philosophy. This idea is similar to what already happened in the backend domain with Service-oriented Architectures (SOA) [Erl05]. This new way of thinking distributed computing helped overcome the problems originated from the monolithic client-server approach thus enabling a more efficient way of interoperation in which functionalities exposed as *services* can be combined by other software applications; each service is built in a way that ensures it can exchange information with any other service in the network without human interaction and without the need to make changes to the underlying program itself [Bel10].

The research community has applied the model-driven approach both on UI specific aspects and in the development of complex interactive systems but has often considered these two fields of work as independent areas, consequently producing different methodologies and tools: a first "technology-oriented" approach led to the creation of formal tools for the description and validation of specific elements; a second "business-oriented" approach centered on the strategic orchestration of services, processes, people and resources. In particular, this second category, close to the enterprise world, has always been more focused on the back-end information flow; since this type of approach is also entering more operative contexts, where there is the need to present not just some forms but more complex data, this aspect is undergoing a marked evolution led by the strong need of advanced user interfaces.

This work wants to bridge the "technology-oriented" and the "business-oriented" approaches introducing an holistic model-driven development process [MPV11] for the whole interactive system development, foreseeing the integration of methods for the automatic generation of user interfaces with the tools used for business process management.

The proposed framework uses the same model-driven engineering (MDE) techniques employed by the generative approaches to formally describe, in a declarative style, the application domain and tasks models. From this knowledge an abstract UI description is derived and used to produce a concrete UI model. The distinction between these two levels of abstraction is needed to create multimodal interfaces since from an agnostic abstract description the transformation process takes as input the specific destination UI type to create the concrete description. Finally this concrete UI model is used to assemble the actual interface combining, as previously introduced, the appropriate complex widgets. This step adds a further layer of flexibility since the concrete UI model refers to widgets through a description of the required features so the party responsible for the actual composition, typically a client device the user is directly interacting with, can adopt a context-aware choice for the actual selection.

This schema identifies at least two different developers profiles: the first type of developer uses declarative techniques to define the elements that constitute the application workflow and delegates the UI composition phase to the framework almost completely; a second type of developer actually creates the complex widgets without constraints, except for the framework integration requirements, and with a complete user-centric approach (i.e., focusing on usability, accessibility and context-awareness concepts and practices).

The proposed approach leverages the declarative MDE techniques employed for UI generation and combines them with the imperative development process predominantly adopted by companies and developers in order to offer a framework fast to exploit but potentially at the state of the art regarding user-centric themes like usability, accessibility, context-awareness and aesthetic appeal.

Following the SOA analogy, it is possible to envision a scenario in which the introduced complex widgets are supplied as services by UI providers, the required business logic is generated from models also obtained through composition from different sources and the final application is pushed to an heterogeneous set of client devices used in diverse situations. Expanding this pattern the resulting applications could be used as building blocks for even more complex systems.

This thesis is articulated in four parts: State of the Art, Proposed Solution, Framework Design and Implementation.

In the State of the Art part, technologies relevant to the topic are discussed; in particular, chapter 1 introduces the different approaches to user interface development adopted by the research community and distinguishing between model-based and model-driven techniques; chapter 2 examines the main user interface description and transformation languages.

In the Proposed Solution part, chapter 3 introduces the proposed methodology and describes the role of each meta-model providing a reference overview for the production system.

The Framework Design part is devoted to the description of the four meta-models; in particular, chapter 4 described the Domain Model, chapter 5 described the Process Model, chapter 6 described the Abstract User Interface Model and chapter 7 described the Concrete User Interface Model. Chapter 8 described the widgets architecture and addresses the composition problem.

Finally, in the Implementation part, chapter 9 introduces the Java demonstrator used to test and validate the framework with a simple but effective use case.

# Part I

# State of the Art

# Chapter 1

# Approaches to User Interface Development

The development of User Interfaces (UI) has been a topic of interest both for academia and industry since the early 1980s. During the time a constant evolution has occurred in the field and such an evolution has been coupled with a continuous effort to develop new methods and strategies to create UIs in effective and efficient ways. This effort can be explained considering that the UI layer is regarded as one of the key components of software applications since it allows users to access the functionalities offered by a specific application [ABY14]. The UI layer is so endorsed to determine or not the success of an application typically without any concern regarding the effective value of the functionalities offered [ABY14]. The main problems encountered during the time have been faced by the HCI community and the industry developing a large quantity of different methods and tools.

Currently, the need to focus on cost-effective development of UIs, both in terms of time and effort, has led to the adoption of principles of Model-Driven Engineering (MDE) resulting in methodologies going under the name of Model Driven UI Development (MDUID). MDUID can be considered as an evolution of the previous approach, the so called Model Based UI Development (MBUID) [MPV11].

It can be of interest to recall also the oldest approach to UI development:

the User Interface Management Systems (UIMS). UIMSs have represented the first historic effort to create UIs in an efficient and effective way. Within this context it will be reported only a brief list of the different generations envisioned in UIMS by Meixner et al. [MPV11] and by Mlađan et al. [MDZ13].

One of the key point of the paradigm in the MDUID is the capability of supporting the so called *context-sensitive* UIs; this term indicates UIs aware of the context of use and able to (automatically) react to context changes in a continuous way (e.g., by changing the UI presentation, contents, navigation and even behavior) [Fon10].

Nowadays such a feature has gained great interest in the domain of application development as a result of the diffusion of pervasive computing and the wide adoption of smartphones and mobile devices among the great public.

The contents of this chapter are organized as follows: first an overview of the beginning of UI development methodologies and tools represented by the User Interface Management Systems, then a presentation of the Model-Based development approach. Then some considerations regarding the new paradigm that led to the Model-Driven development in the UI sector. At last some concepts about the adaptive model driven development where the main focus is in the creation of context-sensitive UIs: a brief explanation of the terms related to context-sensitive UI and context of use will be given.

## 1.1 Historical overview: UI Management Systems

The first historically adopted approach in the UI development was represented by the User Interface Management Systems (UIMS). The term and the first concepts related to UIMS date back to early 1980s [MPV11]. A tool can be considered a UIMS if it satisfies the following definition: “A User Interface Management System is a tool (or tool set) designed to encourage interdisciplinary cooperation in the rapid development, tailoring and management (control) of the interaction in an application domain across varying devices, interaction techniques and UI styles. A UIMS tailors and manages (controls) user interaction in an application domain to allow for rapid and consistent development. A UIMS can be viewed as a tool for increasing programmer productivity. In this way, it is similar to a fourth generation language, where the concentration is on specification instead of coding” [BBF+87].

Within the UIMS approach four different generation have been devised where each one takes into account a specific target audience and adopted methods [Hix90]. The first generation was set about in the period 1968–1984 while the second overlapped part of the first in a period from 1982-1986. Both these two generations concerned only the creation of teletype UIs (text-based UI) in a period when the most of peripherals were keyboard and monitor. As a matter of fact the UIMSs themselves adopted teletype UI to interact with the users and the methodology was centered in the programming of the UI. For this reason, these two generations targeted only programmers as users because the UI were created by means of common programming languages. Otherwise the third (approximately 1985-1988) and fourth (approximately 1988-1990) generation drew their attention to graphical direct manipulation UI and shifted their main concern from UI programming to its design. Furthermore the fourth generation, in order to ease the user interaction with the UIMS, itself adopted the WIMP paradigm. The term WIMP stands for Windows, Icons, Mice and Pointing, and it is used to refer to the desktop, direct manipulation style of user interface. This allowed other new professional figures, such as designers, to start using UIMSs [MPV11].

Notwithstanding the great interest of academic world the UIMSs were not adopted in the industry and remained a research field of HCI area mainly due to three motivations [Mye87]. The first problem lay in the difficulty of use of the UIMS: as a matter of fact only very skilled programmers were able to interact with them. The second one consisted in the fact that UIMS were mainly able to create teletype UI and not WIMP UI which were much more complex to develop. In the late 1980s, GUIs and WIMP UIs had encountered the favor of a wide audience due to their simplicity of use with respect of older teletype UIs. The third problem was that the UI generated by UIMS were deeply bound to a specific platform and it was not possible to use them in different environments other than the one they were created for.

## 1.2 Model-Based UI Development

To overcome the main problems arisen with the UIMS, another approach in UI development gained more and more interest. Its central paradigm was the realization of high-level models for representing the characteristics of a UI in the analysis and design phases: it was the Model-Based UI Development (MBUID). Its origin dates back the late 1980s [MPV11]; two criteria has been identified for a UI tool to be a MBUID environment [Sch96]:

1. a MBUIDE must include a high-level, abstract and explicitly represented (declarative) model about the interactive system to be developed (either a task model or a domain model or both);

2. MBUIDE must exploit a clear and computer-supported relation from the model described in 1 to the desired and running UI. This means that there must be some kind of automatic transformation such as a knowledge-based generation or a simple compilation to implement the running UI.

These two criteria allow to highlight the driving concepts which represent the basis of such an approach. The first criteria has its focus on giving the UI designer the capability to concentrate more on a semantic level than to be distracted by the details involved in the implementation level. The designer can create models without the concern of what will be the tools adopted for the implementation of the UI. Specific languages have to be adopted to create these models enabling the integration within development environments. These specific languages are the so called User Interface Description Language (UIDL) which allow to describe the model in a declarative way without any concern on how the model will be converted in a running UI [Pat05].

The second criteria, instead, focuses on the capability of the MBUIDE to create a running UI starting from the high-level model. In other words, such a criteria implies only that the model is the basis for the realization of the running UI. It implies the capability to generate semi-automatically the code of the UI starting from the model description but no further details are required to describe how the process of UI development has to be made. From the point of view of software engineering there isn't the clear request to define precisely the different steps related to the appropriate software development life cycle [FV10].

Since the late 1980s the approach of model-based development has evolved into different generations bound to different visions 1.2.1. At first, the main idea was to have a single high-level model to describe the UI. Then it was devised the need to define different models to describe the different aspects of a single UI. At last, the issues related to the growing use of many different devices for a single user and the increased user mobility have led to the need to create different UIs specific for different devices and different contexts of use. The production of so many different models, showed its shortcomings and the need to try to fully automate the UI development life cycle increased. Consequently, the effort to define a specific methodology for UI life cycle became of interest and this also led to the definition of the Cameleon Reference Framework (CRF) [CCT$^+$03] which envisions four UI development steps for multi-context interactive applications where each development step is devoted to manipulate any specific artifact of interest as a model or a UI representation [Van05]. The definition of a specific methodology has somehow paved the way to an evolution in the paradigm of UI development which has shifted towards the approach of Model-Driven Engineering which has led to MDUID.

### 1.2.1 Different Generations in MBUID Systems

Four generations have been devised in MBUID systems [MPV11]. The *first generation* (approximately 1990–1996) focused on identifying and abstracting relevant aspects of a UI. Tools in this generation mainly used one universal declarative UI model which integrated all relevant aspects of a UI. The “one design fits all” can summarize the first generation vision. The main trends focused on the fully automatic generation of the UI instead of an integrated holistic MBUID process. Examples for the first generation are UIDE, AME or HUMANOID [MPV11].

The *second generation* (approximately 1995–2000) focused on the extension of the UI model by integration of other distinct models into the MBUID and expressing the high-level semantics of a UI. Therefore, the UI model is structured into several other models like e.g., task model, dialog model or presentation model. With the second generation, developers were able to specify, generate and execute UIs. Much emphasis has been done on the integration of task models (e.g., CTT described in 2.1.4) into MBUID. Furthermore, a user-centered development (UCD) approach for UI on the basis of a task model has been recognized as crucial for the effective design of UIs. Examples for the second generations are ADEPT, TRIDENT or MASTERMIND [MPV11].

The *third generation* (approximately 2000–2004) was mainly driven by the plethora of new interaction platforms and devices. Mobile devices like e.g., smartphones or PDAs, became popular. Indeed, as Myers, Hudson and

Pausch indicated while discussing the future of UI tools, the wide platform variability encourages a return to the study of some techniques for device-independent UI specification [MHP00]. Then, the system might choose appropriate interaction techniques taking all of these into account. Developers and designers had to face the challenge of developing a UI for several different devices with different constraints (e.g., screen size). An expressive integrated MBUIDE became more relevant than in the previous generations. Examples for the third generation are TERESA or Dygimes [MPV11].

The *fourth and current generation* has been mainly interested by context sensitive UIs and has taken into account the lesson learned by the Cameleon Reference Framework (CRF), described in 1.2.2, which has led to new tools and to the shift towards the MDE approach described in 1.3, giving birth to a new approach in the UI development, the so called Model-Driven UI Development (described in 1.3.1) [MPV11]. Examples for this generation are UsiXML and MARIA XML [MPV11].

## 1.2.2 Cameleon Reference Framework

The Cameleon Reference Framework (CRF) serves as a reference for classifying user interfaces supporting multiple targets, or multiple contexts of use in the field of context-aware computing [CCT$^+$03]. A multi-target (or multi-context) UI supports multiple types of users, platforms and environments (as described in 1.3.2 at page 15). Multi-user, multi-platform and multi-environment UIs are specific classes of multi-target UIs which are, respectively, sensitive to user, platform and environment variations [CCT$^+$03].

The CRF is a unified user interface reference framework that is based on two principles [Fon10]: a model-based approach and the coverage of both the design and runtime phases of multi-target user interfaces. As opposed to conventional UI development techniques that merely construct a concrete level (e.g., graphical buttons, text boxes, etc.), CRF introduces additional levels of abstraction that help in building multi-context user interfaces moving in the direction of a new paradigm in the UI development, the one represented by MDE [ABY14].

In figure 1.1, it is reported a simplified version of the Cameleon Reference Framework to ease the comprehension. This simplified version structures the development processes for two different contexts of use into four development steps (each development step being able to manipulate any specific artifact of interest as a model or a UI representation) [Van05]:

- Final UI (FUI) Model: is the operational UI i.e. any UI running on a particular computing platform either by interpretation (e.g., through a Web browser) or by execution (e.g., after compilation of code in an interactive development environment) [Van05]. The actual UI can be rendered with an existing presentation technology such as HTML, Windows Forms, Windows Presentation Foundation, Swing, and so on. [ABY14];

- Concrete UI (CUI) Model: concretizes an abstract UI for a given context of use into Concrete Interaction Objects (CIO) [VB93] so as to define widgets layout and interface navigation. It abstracts a FUI into a UI definition that is independent of any computing platform. Although a CUI makes explicit the final Look and Feel of a FUI, it is still a mock-up that runs only within a particular environment. A CUI can also be considered as a reification of an AUI at the upper level and an abstraction of the FUI with respect to the platform [Van05]. This level is *modality dependent*. For example, it can represent the UI in terms of graphical widgets such as buttons, labels, and so forth. Possible UIDLs for representing concrete user interfaces include TERESA XML, UIML [APB+99], XIML [PE02], UsiXML, and MARIA.

- Abstract UI (AUI) Model: defines abstract containers and individual components [LVM+05], two forms of Abstract Interaction Objects [VB93] by grouping subtasks according to various criteria (e.g., task model structural patterns, cognitive load analysis, semantic relationships identification), a navigation scheme between the container and selects abstract individual component for each concept so that they are independent of any modality. An AUI abstracts a CUI into a UI definition that is independent of any modality of interaction (e.g., graphical interaction, vocal interaction, speech synthesis and recognition, video-based interaction, virtual, augmented or mixed reality). An AUI can also be considered as a canonical expression of the rendering of the domain concepts and tasks in a way that is independent from any modality of interaction. An AUI is considered as an abstraction of a CUI with respect to interaction modality. At this level, the UI mainly consists of input/output definitions, along with actions that need to be performed

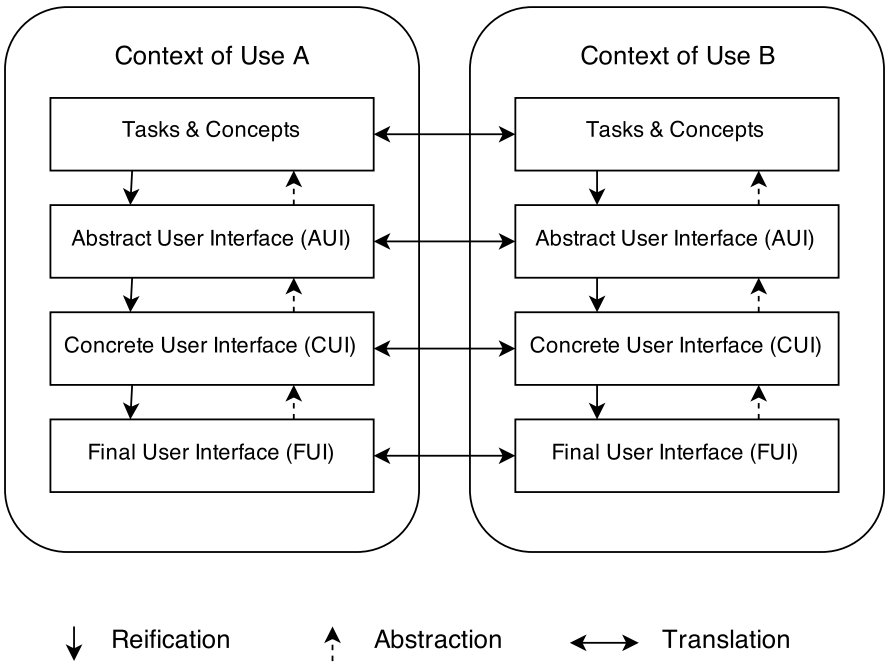

Context of Use A
Context of Use B
Tasks & Concepts
Abstract User Interface (AUI)
Concrete User Interface (CUI)
Final User Interface (FUI)
Tasks & Concepts
Abstract User Interface (AUI)
Concrete User Interface (CUI)
Final User Interface (FUI)
Reification
Abstraction
Translation

Figure 1.1: *The simplified Cameleon Reference Framework*

on this information [Van05]. The AUI model can be represented using UIDLs such as TERESA XML [BCPS04], UsiXML [LVM+05], and MARIA [PSS09].

- Task and Domain Models: describe the various user's tasks to be carried out and the domain-oriented concepts as they are required by these tasks to be performed. These objects are considered as instances of classes representing the concepts [Van05]. The task model is the highest level of abstraction that represents UI features as tasks. One possible representation for task models is the ConcurTaskTrees [PMM97] notion that allows tasks to be connected with temporal operators. The domain model denotes the application universe of discourse and can be represented using UML class diagrams.

Between these levels exist different relationships [MPV11]:

- *Reification* covers the inference process from high-level abstract descriptions to runtime code. The CRF recommends a four-step reification process: a Concepts-and-Tasks Model is reified into an Abstract UI which in turn leads to a Concrete UI. A Concrete UI is then turned into a Final UI, typically by means of code generation techniques.

- *Abstraction* is an operation intended to map a UI representation from one non-initial level of abstraction to a higher level of abstraction. In the context of reverse engineering, it is the opposite of reification.
- *Translation* is an operation that transforms a description intended for a particular context into a description at the same abstraction level but aimed at a different context. It is not needed to go through all steps: one could start at any level of abstraction and reify or abstract depending on the project.

## 1.3 Model-Driven Software Development

The model-driven paradigm is gaining more and more interest over recent years in the field of software engineering [BF14]. Specifically in the development of a software system Model-Driven Engineering (MDE) main concern is represented by models which are abstract representations of the system or product under construction [BF14]. Nevertheless the idea of using models doesn't represent any novelty in software engineering industry where models have been normally used in the analysis and design phases but they have been neglected in the implementation and maintenance phases [BF14]. As a matter of fact the models created in the design phase were simply handled to software developers which had the task to create the application code taking them as a reference. Such an approach can be considered model-based but not model-driven where instead models represent the key drivers of the creation of an application in each phase of the development process making also possible the automatic generation of code without any concern of human intervention.

The model-based approach has had a greater usage than model-driven in the past years and the motivations to such a situation can be found in the nature of the software product itself and in the life cycle of its development process; as a matter of fact it is extremely simple to make changes during the maintenance phase directly on the source code without any concern in adjusting the corresponding models and often a software is released and then iteratively fixed while in use. Notwithstanding the previous considerations, MDE has been increasingly adopted in the past two decades mostly due to the proliferation of different platforms, the scarcity of skilled developers and the large agreement upon few standards defined by the Object Management Group (OMG) [BF14].

Since 1997, the OMG has launched an initiative called Model-Driven Architecture (MDA) to support the development of complex, large, interactive software systems providing a standardized architecture with which:

- systems can easily evolve to address constantly evolving user requirements;
- old, current and new technologies can be harmonized;
- business logic can be maintained constant or evolved independently of the technological changes;
- legacy systems can be integrated and unified with new systems; in MDA, a systematic method is recommended to drive the development life cycle to guarantee some form of quality of the resulting software system [Van05].

Four principles underlie the OMG view on MDAs [MSUW04]:

1. models are expressed in a well-formed unified notation and form the cornerstone to understanding software systems for enterprise scale information systems. The semantics of the models are based on meta-models [Van05];
2. the building of software systems can be organized around a set of models by applying a series of transformations between models, organized into an architectural framework of layers and transformations: model-to-model transformations. A MDA-compliant environment for developing UIs of information systems support any change between models while model-to-code transformation are typically associated with code production, automated or not [Van05];
3. a formal underpinning for describing models in a set of meta-models facilitates meaningful integration and transformation among models, and is the basis for automation through software [Van05].

4. acceptance and adoption of this model-driven approach requires industry standards to provide openness to consumers, and foster competition among vendors [Van05].

Specifically, MDE approaches provide techniques and tools for dealing with models in an automated way in order to generate executable software [MDZ13]. Models, meta-models and model transformations are the key issues of MDE to increase productivity of software development. If models can be seen as abstract representation of a system, the meta-models represents a set of rules and constraints which a model has to undergo to be formally correct. Model transformations are the process which allow to generate lower-level models from higher-level models, the model-to-model (M2M) transformations or eventually to generate code, the model-to-code (M2C) transformations. At present however the MDE approach is adopted only to generate code from models in specific tiers because the quality of a fully generated application is so far lower than one directly developed. Moreover it is still too complex and expensive creating models which allow the automatic generation of all aspects of a software application [BF14].

Current MDE approaches mostly rely on Unified Modeling Language (UML) notation to describe models. UML is a widely adopted industrial standard used in a large number of software engineering fields and with rich tool support [MDZ13]. UML is actually a collection of languages, including collaboration diagrams, activity diagrams, as well as use case support. These languages are intended to cover all aspects of specifying a computational system. While they have been used for the interactive part as well, they have not been expressly designed to support it and they tend to ignore some specific aspects related to the user interaction [Pat05]. On the other hand, the Human-Computer Interaction field has brought specific notations for describing user interfaces such as task models before UML had been proposed. With the advent and the acceptance of UML, existing notations for user interface descriptions were shaped in UML setting [MDZ13]. Thus far, several UML models for user interface description were introduced, such as USIXML [LVM$^+$05].

### 1.3.1 Model-Driven UI Development

The basics concepts of MDE have been taken into account in the UI development resulting in a new approach called Model-Driven UI Development

(MDUID) [MDZ13]. This approach can be considered the natural evolution of MBUID, described in 1.2, which has been the main paradigm in the UI development sector since the early 1990s [MPV11]. As a matter of fact the fourth and last generation devised in MBUID systems is considered model-driven and not model-based any more as models and transformations among models are at the heart of the development process [ABY14].

Generally speaking in the MDUID several models are used to describe different aspects of user interface when the level of detail varies but more specifically it is possible to see different trends within this approach which can be summarized in [ABY14]: Static modeling, Generative runtime modeling and Interpreted runtime modeling. The first trend adopts static models for UI design and these models don't change at runtime so they are not employed for the code generation and they are adopted only in the analysis and design phases. In the second trend, models are also adopted to generate code and are therefore used in the development phase. The third trend doesn't require code generation and models are interpreted directly at runtime to create the UI [ABY14].

The main advantages derived from MDUID are represented by enhancing traceability, technology independence [ABY14] and reduction of development costs [Flo06]. The latter two advantages are mainly due to the fact that from a single high level model UIs are automatically generated for many different platform and devices. However, there are also some drawbacks in this approach and the main one is represented by the low usability of automatically generated UIs; this is a consequence of the difficulty in specifying the details concerning the layout of the UI in the higher-level models [RPV12].

As it has been introduced when speaking of MBUID generations, it is possible to say that current approaches in the fourth generation are model-driven and they can't be considered model-based anymore [MPV11]. This shift of paradigm has been made possible by the seminal work in the Cameleon Reference Framework which has provided an abstraction guidance for devising Uis with a model-driven approach [ABY14]. This generation (approximately 2004–today) is focusing on the development of context-sensitive UIs for a variety of different platforms, devices and modalities, called multi-path development, and the integration of web-applications. Central elements of most of the current approaches are models which are mostly stored as XML-based languages to enable easy import and export into authoring tools. Furthermore, one research focus is on the optimization of the automatically generated

UIs by ensuring a higher degree of usability [MPV11].

### 1.3.2 Adaptive Model-Driven UI Development

Recently great interest has raised about situations where an high context variability is present. Such interest is mainly due to the increased mobility of the users of software systems and to the large use of many different devices by a single user which spans his attention from the laptop to his smartphone in different moments of his or her daily life. In this scenario the users should not adapt to applications but the applications should adapt to the different contexts of use. This consideration implies the creation of many different UIs for the same functionality in order to answer the needs of the different situations; such an approach can be extremely expensive whereas the UIs should be "hand-made" developed. On the contrary, the MDE approach seems to offer the right solution to satisfy such needs as it introduces a level of abstraction in the software applications which allows to describe the required UI without the concern of taking into account the context of use. Moreover, MDE promotes the automatic generation of code starting from the models created during the design phase. This approach has been considered as the most promising in literature for such a problem leading to Adaptive Model-Driven UI development (AMDUID) [ABY14] which aims to create context-sensitive UI. In order to describe in a plain way such a theme, it is important to give some details of what "context" means and which are the properties a UI should posses to be a context-sensitive UI. Moreover, there are several different types of context-sensitive UIs and it is worth trying to give some glossary to explain such differences:

- context-awareness "indicates that a system is aware of its context, which is its operating environment" [ST09]. If the UI is aware of its context and is able to detect context changes, then it can trigger adaptations in response to those changes (e.g., based on a set of rules) in order to preserve its usability.

- self-configuring "is the capability of automatically and dynamically re-configuring in response to changes" [ST09]. To keep the UI adaptation rules up to date with an evolving context of use (e.g., if a user's computer skills improve), there is a need for a mechanism that can reconfigure these rules by monitoring such changes. Another type of rule reconfiguration could be based on the end-user's feedback; for example, the end-user may choose to reverse a UI adaptation or select an alternative. Keeping the end-users involved in the adaptation process could help in increasing their awareness and control, thereby improving their acceptance of the system.

- self-optimizing "is the capability of managing performance and resource allocation in order to satisfy the requirements of different users" [ST09]. To adapt this definition to user interfaces, we can say that a UI can self-optimize by adapting some of its properties, for example, adding or removing features, changing layout properties (e.g., size, location, type), providing new navigation help, and so forth.

Context literally refers to the meaning that must be inferred from the adjacent text. As a result, to be operational, context can only be defined in relation to a purpose, or finality [CCRR02]. In the field of context-aware computing, a definition of context that has been largely used is provided by Anind Kumar Dey in [Dey00]: "Context is any information that can be used to characterize the situation of entities (i.e., whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves. Context is typically the location, identity and state of people, groups and computational and physical objects."

While the above definition is rather general, thus encompassing many aspects, it is not directly operational. Hence, Fonseca in [Fon10] defines the Context of Use of an interactive system as a dynamic, structured information space that includes the following entities:

- a model of the User $U$ who is intended to use or is actually using the system;
- the hardware-software Platform $P$ which includes the set of computing, sensing, communication and interaction resources that bind together the physical environment with the digital world;
- the social and physical Environment $E$, where the interaction is actually taking place.

Thus, a context of use is a triple composed by $(U, P, E)$. The User represents the human being, or a human stereotype, who is interacting with the system. The characteristics modeled or relevant can be very dependent on the application domain (e.g., age, level of experience, preferences, tastes, abilities and disabilities, short term interests, long term interests). In particular, perceptual, cognitive and action disabilities may be expressed in order to choose the best modalities for the rendering and manipulation of the interactive system [Fon10].

The definition of platform can accommodate physical devices (e.g., phone, tablet, laptop, etc.), operating systems and different types of application platforms (e.g., web, desktop, rich Internet application, etc.). Variability in screen size and the available UI widgets are common examples of aspects that could spur platform-related adaptive UI behavior [ABY14].

The Environment denotes the set of objects, persons and events that are peripheral to the current activity but that may have an impact on the system and/or users behavior, either now or in the future [CR02]. According to our definition, an environment may encompass the entire world. In practice, the boundary is set up by domain analysts whose role is to elicit the entities that are relevant to the case at hand. Specific examples are: user's location, ambient sound, lighting or weather conditions, present networks, nearby objects, user's social networks, level of stress and so on [Fon10].

The relationship between a UI and its context of use leads to the following definitions:

- *A multi-target UI* supports multiple types of users, platforms and environments. Multi-user, multi-platform and multi-environment UIs are specific classes of multi-target UIs which are, respectively, sensitive to user, platform and environment variations [CCT+03].
- *An Adaptive UI* refers to a UI capable of being aware of the context of use and to (automatically) react to changes of this context in a continuous way (for instance, by changing the UI presentation, contents, navigation or even behaviour).
- *An Adaptable UI* can be tailored according to a set of predefined options. Adaptability normally requires an explicit human intervention. We can find examples of UI adaptability on those word processors where the set of buttons contained by toolbars can be customized by end users.
- *A Plastic UI* is a multi-target UI that preserves usability across multiple targets. Usability is not intrinsic to a system. Usability can only

be validated against a set of properties set up in the early phases of the development process [CCT+03].

# Chapter 2

# Languages for User Interface Development

The model-driven user interface development has focused its attention around the concept of model; the production of a model gains a central role in the UI life cycle: all the components are models and all models have to be consistently defined according to a meta-model, the underlying language has to be based on a meta-language, and the software should be mainly based on model-to-model (M2M) transformation and model-to-code generation (M2C) [FV10]. In this context it is possible to speak about different languages used for UI description and transformations between UI models, namely UIDL (User Interface Description Language) and UITL (User Interface Transformation Language) [MDZ13].

A User Interface Description Language (UIDL) is hereby defined as a language for describing any kind of User Interface (UI) at a higher level of abstraction than the code used to program it, whether it is a programming language or a markup language [FV10]. In theory, a UIDL can be defined according to any programming paradigm and its syntax can be specified according to any formal scheme. In practice however, most UIDLs are declarative and are defined as a markup language, typically based on XML [FV10]. In the model-driven approach, even transformations among models and relationships are described in terms of a meta-model. A mapping model defines the relationships between the models; this mapping model allows the specification of the link of elements from heterogeneous models and viewpoints. Several relationships can be defined to explicit the relationships between models [GLCV08].

User Interface Transformation Languages (UITL) are languages that allow to describe transformation mechanisms used to map a model onto another one but the logic and the definition of the transformation rules are completely hard coded, with little or no control by designers. In addition, the definition of these representations is not independent of the transformation engine [LVM+05].

## 2.1 User Interface Description Languages

A User Interface Description Language (UIDL) consists of a high-level computer language for describing characteristics of interest of a UI with respect to the rest of an interactive application in order to be used during some stages of the UI development life cycle. Such a language involves defining a syntax (i.e., how these characteristics can be expressed in terms of the language) and semantics (i.e., what do these characteristics mean in the real world). It can be considered as a common way to specify a UI independently of any target language (e.g., programming or markup) that would serve to implement the UI [GGGCVMA09].

The introduction of UIDLs has allowed professional figures other than programmers to enter the UI development process. This has been possible mainly due to the adoption in most UIDL of a declarative approach in place of the classical imperative paradigm more familiar to a software developer. These new figures, as for example UI designers, has permitted to pay more attention to usability and accessibility aspects of the user interface. The adoption of models described by UIDLs has eased the process of communication among the different stakeholders of the development process and has paved the way to semi-automatically generate the UI code. Since the end of 90s, UIDLs have quickly multiplied due to these interesting features and thanks to the great simplicity to create languages based on XML [GGGCVMA09].

In the present chapter only some UIDL languages will be presented and the choice made has been driven by the capability of the languages to describe models with respect of the Cameleon Reference Framework (CRF) described in 1.2.2. This choice has been done mainly because CRF represents a suitable reference for the adoption of a model-driven paradigm (MDUID) [ABY14].

### 2.1.1 USer Interface eXtensible Markup Language

The USer Interface eXtensible Markup Language (UsiXML) is structured according to the four levels of abstraction defined by the CRF. UsiXML relies on a transformational approach that progressively moves among the levels to the Final UI [LVM+05]. The transformational methodology of UsiXML allows the modification of the development sub-steps, thus ensuring various alternatives for the existing sub-steps to be explored and/or expanded with new sub-steps. As such, UsiXML supports model-driven engineering of UIs as defined by the Object Management Group (OMG). Designers can shape the UI of any new interactive application by specifying and describing it in the UIDL, without requiring programming skills usually found in markup and programming languages [MPV11]. UsiXML allows cross-toolkit development of an interactive application. A UI of any UsiXML-compliant application runs in all toolkits implementing it.

UsiXML supports *device independence*: a UI can be described in a way that remains autonomous with respect to the devices used in the interactions (e.g., mouse, screen, keyboard, voice recognition system). In case of need, a reference to a particular device can be incorporated. UsiXML supports platform independence: a UI can be described in a way that remains autonomous with respect to the various existing computing platforms (e.g., mobile phone, Pocket PC, Tablet PC, kiosk, laptop, desktop, wall screen). In case of need, a reference to a particular computing platform can be incorporated.

Moreover UsiXML supports *modality independence*: a UI can be described in a way that remains independent of any interaction modality (e.g., graphical interaction, vocal interaction, 3D interaction, virtual reality interaction). In case of need, a reference to a particular modality can be incorporated. UsiXML allows reusing elements previously described in other UIs in order to leveraging them in new applications. Historically, the first version of UsiXML resulted from the EU-funded FP5 Cameleon Project.

### 2.1.2 Useware Markup Language

The Useware Markup Language (UseML) 1.0 refers to the Task and Concepts level of the CRF and was developed to support a user-centered development (UCD) process (ISO 9241-210) with a modeling language representing the results of the initial task analysis. Accordingly, the use model (task model) abstracts platform-independent tasks into use objects (UO) that make up a hierarchically ordered structure. Furthermore, the leaf tasks of a use model are described with a set of elementary use objects (eUO) representing atomic inter-active tasks: inform, trigger, select, enter and change. In Version 2.0, UseML was extended with five temporal operators to support temporal relationships as well as it provides the possibility to define multiple executions or conditions that can be attached to tasks of the model [MSB11]. This information can be used later in the transformation process to derive a dialog model. UseML is supported by Udit, an interactive editor and simulator for task models which is also able to transform use models into Dialog and Interface Specification Language models (language introduced in 2.1.3).

### 2.1.3 Dialog and Interface Specification Language

The abstract UI level of the CRF can be modeled with the Dialog and Interface Specification Language (DISL) [SBM07], which is a User Interface Markup Language (UIML) subset that extends the language in order to enable generic and modality independent dialog descriptions. Modifications to UIML mainly concerned the description of generic widgets and improvements to the behavioral aspects. Generic widgets are introduced in order to separate the presentation from the structure and behavior, i.e., mainly to separate user- and device-specific properties and modalities from a modality-independent presentation. The use of generic widget attributes enables to assign each widget to a particular type of functionality it ensures (e.g., command, variable field, text field, etc.). Further, a DISL rendering engine can use this information to create interface components appropriated to the interaction modality (e.g., graphical, vocal) in which the widget will operate.

The global DISL structure consists of an optional head element for Meta information and a collection of templates and interfaces from which one interface is considered to be active at one time. Interfaces are used to describe the dialog structure, style and behavior, whereas templates only describe structure and style in order to be reusable by other dialog components [GGGCVMA09].

### 2.1.4 ConcurTaskTrees

The ConcurTaskTrees (CTT) notation [Pat99] addresses the Task and Concepts level of CRF and has represented an important contribution towards engineering task models and making them exploitable in various contexts in both design and runtime phases. It has a set of features that make it suitable to easily represent activities that need to be carried out to reach the user goals: hierarchical structure, temporal relations, icons to indicate task allocation and a set of attributes to indicate various aspects (e.g., task type, task objects, relevant platforms for task execution). Recently, the possibility of better specifying preconditions has been added. Such preconditions can also be considered by the associated interactive simulator, which is included in the ConcurTaskTrees Environment, a publicly available tool for editing and analyzing task models.

The CTT specifications can be saved in XML format in order to include and exploit them in other tools and environments. CTT and the associated tool have been exploited over time in various application domains e.g., interactive safety-critical systems, enterprise resource planning applications and service engineering [MPV11].

### 2.1.5 MARIA

The Modelbased lAnguage foR Interactive Applications (MARIA) language [PSS09] addresses different abstraction layers of CRF: in particular the Abstract UI and the Concrete UI levels [MPV11]. It is associated with a publicly available tool (MARIAE). This language has been developed following the experiences gathered with previous approaches in order to: *i*) support a data model, which is useful for specifying the format of input values and the association of various data objects to various interactors; *ii*) specify events at abstract or concrete level, which can be property change events or activation events (e.g., access to a web service or a database); *iii*) include an extended dialog model, obtained through conditions and CTT operators for event handlers thus allowing specification of parallel input; *iv*) support UIs including complex and Ajax scripts with the possibility of continuously updating fields without explicit user request; and *v*) describe a dynamic set of UI elements with conditional connections between presentations with the possibility of propagating changes to only a part of the UI.

The associated tool supports the editing of Abstract UIs in the MARIA language, which can be derived from a task model or created from scratch. The editor supports browsing the specification through an interactive tree view and a graphical representation of the elements of a selected presentation, in addition to showing the XML specification. The editor allows the editing through drag-and-drop of the elements and their attributes.

From the abstract description, it is possible to derive concrete descriptions for various platforms (e.g., desktop, mobile, vocal, multimodal). Each concrete description can be presented and edited with modalities similar to those for the abstract specifications. From the concrete descriptions, it is possible to obtain implementations for various implementation languages (e.g., XHTML, HTML5, JSP, VoiceXML, X+V, SMIL) through associated transformations [MPV11].

### 2.1.6 User Interface Markup Language

User Interface Markup Language (UIML) [APB$^+$99] is an XML-based language addressing the Concrete UI level of the CRF [MPV11]. UIML provides: *i*) a device-independent method to describe a UI; and *ii*) a modality-independent method to specify a UI.

UIML allows describing the appearance, the interactions and the connections of the UI with the application logic. The following concepts underlie UIML [GGGCVMA09]:

- UIML is a meta-language: UIML defines a small set of tags, used to describe a part of a UI, that are modality-independent, target platform-independent (e.g., desktop, mobile) and target language-independent (e.g., Java, VoiceXML). The specification of a UI is done through a toolkit vocabulary that specifies a set of classes of parts and properties of the classes. Different groups of people can define different vocabularies: one group might define a vocabulary whose classes have a 1-to-1 correspondence to UI widgets in a particular language (e.g., Java Swing API), whereas another group might define a vocabulary whose classes match abstractions used by a UI designer;

- UIML separates the elements of a user interface and identifies: *i*) which parts are composing the UI and the presentation style; *ii*) the content of each part (e.g., text, sounds, images) and binding of content to external resources; *iii*) the behavior of parts expressed as a set of rules with conditions and actions; and *iv*) the definition of the vocabulary of part classes;

- UIML logically groups the user interface elements in a tree of UI parts which dynamically changes over the lifetime of the interface itself. UIML provides tools to describe the initial tree structure and to dynamically modify it;

- UIML allows UI parts and the aforementioned part-trees to be packaged in templates: these templates may then be reused in various interface designs.

### 2.1.7 eXtensible Interface Markup Language

The eXtensible Interface Markup Language (XIML) [PE02], is a language developed by Redwhale Software, derived from XML and able to store the models developed in MIMIC [Pue96]. MIMIC is a meta-language that structures and organizes interface models. It divides the interface into model components: user-task, presentation, domain, dialog, user, and design models. The design model contains all the mappings between elements belonging to the other models. The XIML is thus the updated XML version of this previous language.

The XIML language is mainly composed of four types of components: models, elements, attributes, and relations between the elements. The presentation model is composed of several embedded elements, which correspond to the widgets of the UI, and attributes of these elements representing their characteristics (e.g., color, size). The relations at the presentation level are mainly the links between labels and the widgets that these labels describe.

XIML supports design, operation, organization, and evaluation functions; it is able to relate the abstract and concrete data elements of an interface; and it enables knowledge-based systems to exploit the captured data.

## 2.2 User Interface Transformation Languages

Model-driven engineering of user interfaces assumes that various models describe different aspects of user interface. Relations between these models are established through model transformations. In this way, the development of user interfaces can be seen as a transformation chain starting with models at high level of abstraction and ends with executable versions of user interface. An extensive taxonomy of model transformation approaches has been proposed in [CH06].

Variability of semantics between different models, their formats and tools produced various transformational approaches in the context of model-driven development of user interfaces. Some of them operate directly upon models, while others work with their derived formats; some are integrated in models, while others are applied externally; finally, some are editable and modifiable, while others are integrated in tools and cannot be modified [MDZ13].

### 2.2.1 Graph Transformations

GT (Graph Transformations) presents a formal, declarative approach for transformations of models with a structure of directed graph [CH03]. UsiXML is a candidate language to use this type of transformation. The models formed with UsiXML are based on graphs and therefore, the model mappings of UsiXML are specified with graph transformations consisting of a set of transformation rules [LVM$^+$05, SVM08]. Each rule consists of a Left Hand Side (LHS) matching a graph G, a Negative Application Condition (NAC) not matching G and a Right Hand Side (RHS) which is the result of the transformation. The transformation is performed by searching LHS templates in source model and replacing found matching patterns with RHS, while taking into account the NAC. The main limitation of the approach is that it requires models with an underlying graph structure [MDZ13].

### 2.2.2 Atlas Transformation Language

The Atlas Transformation Language (ATL) is an hybrid language for transformations of UML models [JK06]. In this sense, the user can choose whether to use the pure declarative features of the language, or to employ the additional imperative features. The declarative approach is realized by a system of matching rules, where a source pattern is described through a set of source types and constraints on provided types. The target pattern is specified in a similar way by specifying a set of target types together with a set of bindings used to initialize the target types features. The declarative aspect offers a pretty straightforward way to specify transformation rules however, it may be difficult to specify more complex rules; in this case, ATL provides imperative constructions organized in action blocks. These blocks can be added to declarative rules, or even call external code for transformation logic [MDZ13].

ATL is a good candidate for model transformations according to the following arguments: it is an open-source software with a large user community, a solid developer support and a rich knowledge base of model transformations [JABK08].

### 2.2.3 TXL Transformation Language

TXL is designed as a general purpose transformation language [Cor06]. Among other things, it allows transformations of programming languages since it is not confined to any source or target format. In general, the language comprises the following specifications [MDZ13]:

- specification of a structure to be transformed based on grammars;
- specification of transformation rules based on source/target replacement rules.

TXL is intended to transforming models which have syntax tree structure; this is the case of most of the programming languages.

### 2.2.4 UIML Transformations

An important feature of UIML is its capability to define connections to the back-end logic and to provide mappings to other UIML instances or target languages. Therefore, language specification includes transformation features that define explicit mappings of UIML primitives to target format constructs. A separate section defines connections to the application logic; in particular, specification prescribes mappings to VoiceXML and HTML formats. However, these mappings are not necessarily restricted to XML formats, but may also be defined for other languages (e.g., Java). Considering UIML mapping technique based on explicit matches to target format primitives, it can be seen as declarative.

The obvious advantage of the UIML approach is that user interface definition and transformation are specified in the same language. On the other hand, transformation rules are too simple to support more complex transformation tasks [MDZ13].

### 2.2.5 XSL Transformations

The eXtensible Stylesheet Language Transformations (XSLT) is a language for transforming the XML document submitted as input into a textual, in most cases XML, output [Kay07]. This language can be used to generate documents written in languages different from XML. XSLT is comprised of templates rules. Each rule includes a matching pattern, construction elements (template) and additional optional attributes. The matching pattern consists of expressions evaluated against currently processed node of the input XML document. Transformation executes starting from the document route node and continues until each node is traversed and processed according to the specified rules. When a pattern is matched, the template is recursively executed and the target element is generated. Considering rules processing, XSLT provides constants, variables and literals together with conditions, iterations, recursion and sorting as control structures. In addition, XSLT offers a powerful set of built-in string functions for advanced text processing.

While the XSLT transformation mostly follows a declarative style, it also allows imperative constructs such as conditions, iterations and recursion. Therefore, the language can be considered to be an hybrid [MDZ13].

# Part II

# Proposed Solution

# Chapter 3

# Approach Overview

The research community has applied the model-driven approach both on UI specific aspects and in the development of complex interactive systems but has often considered these two fields of work as independent areas, consequently producing different methodologies and tools: a first "technology-oriented" approach led to the creation of formal tools for the description and validation of specific elements; a second "business-oriented" approach centered on the strategic orchestration of services, processes, people and resources. In particular, this second category, close to the enterprise world, has always been more focused on the back-end information flow; since this type of approach is also entering more operative contexts, where there is the need to present not just some forms but more complex data, this aspect is undergoing a marked evolution led by the strong need of advanced user interfaces.

This work wants to bridge the "technology-oriented" and the "business-oriented" approaches introducing an holistic model-driven development process [MPV11] for the whole interactive system development, foreseeing the integration of methods for the automatic generation of user interfaces with the tools used for business process management. More specifically, the context for this thesis was selected after focusing on the available enterprise platforms possessing, among their features, the capability to render a user interface for their business processes; the attention was placed on the NEGENTIS Enterprise Software Platform [NEG]; the NEGENTIS Platform is an applications infrastructure for the Internet of Everything (IoE) able to integrate, from a process perspective, people, applications and devices in the context of distributed and net-centric systems.

The NEGENTIS Platform has been effectively adopted in various production and research projects such as the SIMOB Project in which it was employed to shape an InfoMobility Integrated Platform [GPCC13] and, more recently, the SITMar Project in which it has been used to provide innovative real-time services for goods monitoring in multimodal transport [ZCA+14]. Both these projects contributed to the beginning of a tight collaboration, still active, between the Department of Information Engineering of the University of Florence and NEGENTIS s.r.l.; this collaboration is also based on shared staff such as the author of the present work of thesis.

From the analyses carried out during the work on these projects, the urge of overcoming the limitations of the class of tools the NEGENTIS Platform belongs to, arose. The main aim of the present work is to define a new methodology for model-driven user interfaces development. This methodology allows to create interactive applications integrating UI development techniques within the context of enterprise platform for the orchestration of business processes.

Such a methodology foresees the definition of a set of representation models and it is synthesized in a specific framework covering both the design-time and runtime phases of multimodal and adaptive UIs life cycle. This framework describes a set of levels and each level addresses different aspects of the development process leveraging suitable representation models.

Moreover, such a framework envisions a production system made of two elements: the back-end subsystem and the front-end subsystem.

The proposed framework is divided into four different abstraction levels and the instances handled by each level belongs to a specific model envisioned in the representation models depicted in the present work.

This chapter contains the foundation of the vision at the basis of the present work in relation to the approaches currently available in literature. Then the representation models and the framework are introduced.

## 3.1 The representation models

This thesis proposes a representation model constituted by four core models each one focused on a specific aspect of the user interface generation global process; this process includes both automatic phases and phases requiring the involvement of a developer user who is expert in the field of the specific element to which is called to contribute. The four envisioned models are:

- the *domain model* has the purpose to describe the application domain by means of the concepts constituting it; this model is the result of the work of an user who is expert in the domain of interest;
- the *process model* describes the tasks or actions to be performed on the concepts contained in the domain model in order to meet the application requirements; also this model is user-defined;
- the *abstract user interface model* contains a description of the UI by means of the interaction elements associated to each task of the process model; this model is automatically generated from the process model;
- finally, the *concrete user interface model* describes the concrete components, called *widgets*, that have to be used in the final UI; this model is automatically generated from the abstract UI model while widgets are designed and implemented by a specific developer.

This approach, based on the separation of concerns principle, favors decoupling between each aspect thus enabling an effective description of each application scenario: the two authored models, the domain and the process models, respectively declare the data involved and the tasks the process is made of, thus stating, in a declarative style, *what* the application must do in order to meet specific application requirements. On this basis, the abstract and concrete models are generated in order to produce the final UI through widgets composition. This clear separation between specific tasks also implicitly suggests an analogous separation between the developer profiles associated to each authored artifact. In particular, as it will be clarified in the following chapters, it is possible to envision three profiles to address the three user-defined areas: domain, process and widgets.

In the following sections each core model is described in order to introduce the features they target and understand their role in the framework.

### 3.1.1 Domain Model

The domain model aims at representing the application domain through the detailed description of every concept directly (to be shown to or manipulated by the application end user) or indirectly (to be used or manipulated by the process itself as a hidden variable) involved in the process.

This model is tightly coupled with the application domain but completely decoupled from the specific tasks the application is required to perform. This means the domain model should be created with the intent to describe the scenario as much in details as possible instead of being just a support tool for the tasks described in the process model. With this premise, the same model could represent the domain for any number of processes.

The main idea consists in defining "enhanced" data types bearing more context information about the data they contain; these can be used by the application in place of simple and plain variables; the application domain is mainly represented by a graph in which the nodes correspond to the concepts of the domain called *entities* and characterized by a set of properties and the arcs correspond to different types of relationships among the concepts; one particularly interesting relationship is the *structural relationship* which conveys the "has a" relationships.

Knowing how the domain elements relate to each other can be useful to better describe the application context with the aim of exploiting this knowledge during both the design phase and the generation phase.

### 3.1.2 Process Model

The process model declares the tasks required to satisfy the application requirements. A process is modeled at its core as an activity diagram; each activity could be of different types to accommodate different needs; activities are used to define the application behavior in terms of the tasks to be executed, also without the user's supervision, and the UI to be generated in order to enable the desired user interaction with data. In particular said interaction is defined by means of *presentation activities* and in terms of the data that, according to the application requirements, the user needs to examine and manipulate; these especially designed presentation activities are defined as a graph of *blocks*; in this context the arcs that connect the blocks represent a logical hierarchy; this graph abstractly describes the activity structure thus complying with the approach intent of describing the application without coupling with any target technology: as for the structural relationships defined among the entities of the domain model, the relationships among the blocks can be exploited during the generation process. Blocks, as activities, are of different types and are used to describe the simple elements that will constitute the final UI; in particular they are used to present to the end user specific domain entities or other data (e.g., temporary variables needed by the process execution) but also to execute unsupervised tasks; with this tasks it is possible to define a workflow directly embedded into the UI and leveraging the data contained in any of the blocks of the same presentation activity; this workflow complements the one defined among the activities and represents an important trait of the UI itself since enables this usually passive component to execute part of the application global business logic.

The flow among activities and blocks is triggered by messages called *signals* which are defined at design-time in the process model and sent at run-time when certain predefined conditions are met.

As anticipated in 3.1.1, the process is defined leveraging the entities contained in the underlining domain model in order to declare operands exploiting the relationships existing among the entities; these operands are then used inside the declaration of certain types of blocks to describe presentation activities required by the particular application requirements to enable interaction with data by the end-user of the application; on the whole, the process model combines tasks and data to represent the application that the user will use.

### 3.1.3 Abstract User Interface Model

Once the domain and the process have been described, an abstract user interface model is automatically generated; it is abstract since it represents any possible UI originated from the same application requirements, which, as already explained, lead to the definition of both the domain model and the process model; being an agnostic representation, it is not dependent on a specific UI paradigm or technology and must, adopting a specific language, therefore describe each element in terms of its user interaction *intent* and not focusing on its "appearance", in a broad sense given the multimodality requisite, of the final user interface.

In a sense, the abstract model is a destructured representation of the UI: it collects all elements without any grouping in order to provide a raw view of the UI that can be exploited in the following generation step. For this reason, this model is considered as an internal representation of the user interfaces that needs to be further transformed before it can be effectively used to describe one of the possible user interfaces.

### 3.1.4 Concrete User Interface Model

The concrete user interface model is automatically generated directly from the abstract UI model but this transformation is parametrized in respect of a specific target technology: as explained in 3.1.3 the abstract model is completely decoupled from any UI paradigm therefore this step is needed to fix a particular UI family among those available for the specific use case. For this reason, from the “intent-based” description of the abstract model, the concrete UI model must give tangibility to said description: this is achieved by means of specific concrete UI components called *widgets*; in more detail, the concrete model does not reference the actual concrete components but identifies widgets by means of a list of required features, a so called *abstract widget*. When the model is parsed the most suitable concrete widget, or simply widget, will be used. With this approach, the concrete model is able to identify a whole “family” of user interfaces all sharing the same target technology: UIs in the same family could in fact be used on a great variety of devices, each one with its peculiar characteristics (e.g., form factor, I/O features), and it is then needed a mechanism to acknowledge this variety with a single concrete UI model; the adoption of the abstract widgets addresses this issue since it delegate its resolution to device composing the final UI which posses the required context knowledge required to perform the concrete widgets selection.

Regardless of the particular UI paradigm, widgets are able to implement the front-end workflow introduced in 3.1.2 in combination with the back-end workflow executed on the server.

## 3.2 The proposed framework

The models introduced in 3.1, which will be described in more detail in chapters 4, 5, 6 and 7, can be associated to four corresponding logical levels. As previously introduced, each model, and hence each level, is focused on a specific area and the applied separation actually decouples the sub-tasks the

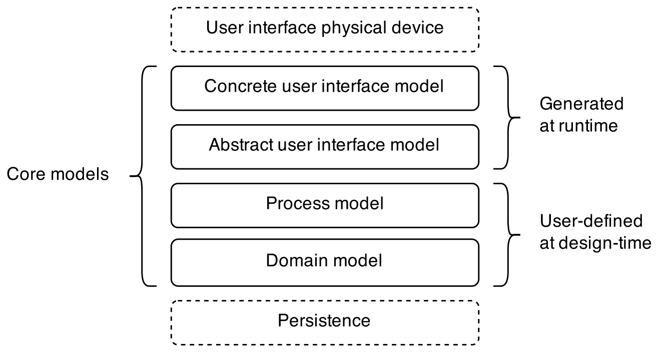

User interface physical device
Concrete user interface model
Abstract user interface model
Process model
Domain model
Persistence
Core models
Generated at runtime
User-defined at design-time

Figure 3.1: *Logical levels and models*

main generation problem is divided into; from this analysis it is possible to produce the representation shown in figure 3.1.

Beyond the areas addressed by these four core models and levels, at least two other topics deserve to be examined: persistence and application fruition. As for persistence, the domain model defines the entities involved in a certain process execution and, as will be explained in 4.3.1, proposes an *instance model* defining the way data can be represented; on the other hand, although not of secondary importance, it is considered out of scope the problem of the actual data persistence which could be assigned to a level placed under the domain level.

Instead, the application fruition theme refers to the set of problems related to the actual platform or device the generated UI can be accessed from in order to enable the intended interaction with the process: UI and process together represent in fact the whole application generated from the models for the end user. Once it is generated, the UI must be submitted to the user and this is done by means of a specific hardware device; this device receives the generated concrete user interface model and assembles the available widgets to produce the interface. This key component could be implemented with very diverse *target technologies* (e.g., graphical, vocal, haptic) in a multimodal approach and the same process could be accessed from different devices with different interaction paradigms. More details on this subject are to be found in chapter 8).

A more architectural outlook on the system elements offers the opportunity to understand how each component implementing one or more of the aforementioned level functionalities interacts with other parts to achieve the desired outcomes (figure 3.2). The system as a whole can be divided in two main elements; one, the back-end system, is devoted to all functionalities related to the actual generation process and to services providing while the other, the front-end system, is quite exclusively represented by the device used to access the services offered by the back-end system and present the application to the end users. Without loosing in generality the couple can be seen also based on a client-server relationship without binding it with any implementation architecture.

From an information flow standpoint, two phases are required for the production of a full application: a design phase and a run phase. The typical use case starts with the design phase and in particular with definition by the domain developer of a domain model representing a specific applications context; then a process developer chooses one among the available domains as the basis for the process model implementing a particular application workflow. Both the domain and the process models are stored in corresponding repositories on the server; these repositories offer services enabling the required interaction with the contained artifacts (e.g., services to list the available domains, to edit the domains and their concepts, to list the available processes, to edit the processes). It is also possible to envision a configuration step in which a deputed user declares specific processes available for certain target technologies: for example a process designed to access a photo library could be made available for a graphical UI but not for a vocal UI, otherwise a process designed to access an audio samples library could be available for both. At runtime, the client interacts with the server and requests the desired process to be "prepared" for the client; the back-end system, from the process and domain models, generates the abstract user interface model first and the concrete user interface model then, customizing the latter on the basis of the characteristics of the client in use. The concrete model is then forwarded to the client which builds the actual UI and enables the user to interact with the system. Contextually, since the process could include also some execution activities that must be run on the server, the back-end starts the process execution carried on by the workflow-engine and waits for signals from the client or from other active processes. The UI on the client interacts with the running process and contributes with its own UI workflow to the global process execution.

The two workflow nodes communicate with each other at runtime exchanging commands and data; this behavior ensures that *i*) each party is able to trigger an action on the other one thus enabling a workflow distributed between the back-end and the front-end and *ii*) data can flow in both directions in a pull or push fashion among the front-end and the back-end thus enabling the UIs and hence the application to offer an always updated application status.

Although logically separated the two workflow nodes cooperate to deliver an application perceived as a whole from the user point of view even when the user changes the device used to access the application. The client side workflow can be forwarded to the server in order to keep an always up to date application status that can be exploited to create recovery checkpoints and/or a seamless client migration.

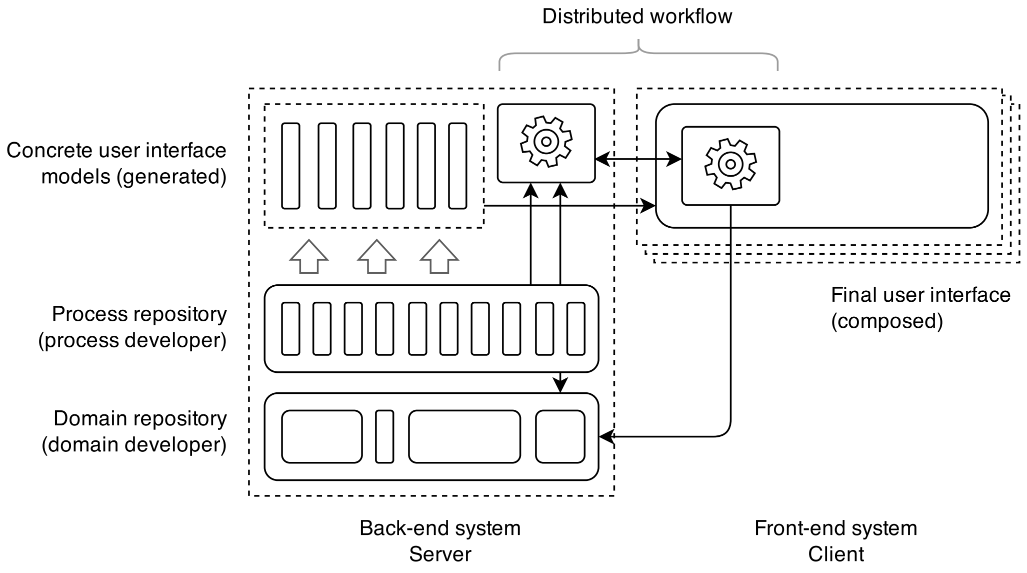

Distributed workflow
Concrete user interface models (generated)
Process repository (process developer)
Domain repository (domain developer)
Final user interface (composed)
Back-end system Server
Front-end system Client

Figure 3.2: *Architecture overview*

In summary, with a system as the one delineated in this section, it is possible to produce complete applications in a multimodal context from the declarative description of domain elements and the desired business processes. The automatically generated UI enables users to interact with a workflow distributed among a back-end system and front-end systems implementing different UI technologies.

Given this reference overview of the whole system, it is plausible to imagine two degenerate and complementary variants, each one collapsing on one of the two architecture endpoints.

The first variant is represented by a back-end only process and is obtained when only execution activities are used for authoring and the process is completely defined by its business logic; in this case the workflow is not distributed and relies only on the server side component. This type of process can be started either by a user or not: in the former case, the client is still required for process start but the actual workflow is run only on the server; for the latter case it is possible to start the process on event, for example, following the reception of a message from another process or from any active communication channel.

The second variant, more interesting in the context of this work, is represented by a front-end only process and is obtained when only presentation activities are used for authoring; in this case the workflow is not distributed and relies only on the client-side component; while still requiring the back-end system for transition triggering, non actual workflow is run on the server; moreover it is possible to envision processes condensed in a single presentation activity thus not requiring interaction with the server but to start the process itself; this could be anyway useful, for example, to ensure that the version of the UI in use is actually the most recent. This style, especially if independent from the back-end, could be useful for situations in which is not assured a reliable communication channel between the client and the server. Moreover, with a similar approach, could be possible to envision a cloud application distribution system; the distributed applications would be always updated and not requiring any installation.

# Part III

# Framework Design

# Chapter 4

# Domain Model

This model describes, from a structural and relational point of view, all the entities a specific application domain can be populated by. As previously stated in chapter 3, this model defines the structured data types each business process, based on said domain, can present and manipulate. This is useful also to keep separate contexts for different business processes; this particular aspect can be effectively exploited to streamline the development and the management processes. Moreover, this model should convey as much detail as possible: every piece of information that could be used to better describe domain entities should be added to this level so to assure that information is placed where belongs instead of where needed for processing.

Since the main requisite is to represent a graph of entities, each of them defined by means of their properties and the connection with other entities, with the intent of formally describe various application contexts, the adoption of knowledge representation techniques is the natural choice. In particular, ontologies represent a viable candidate especially due to their wide adoption and the resulting availability of tools; for this reason this model adopts a similar approach.

This chapter is dedicated to the description of how specific application domain knowledge is represented; this is obtained by means of a model, the Domain Model; a new model must be defined when a new context for business processes is required. In order to define the Domain Model itself, which is actually an instance of all possible domain models, a Domain Meta-model is required; this chapter is dedicated to the description of this meta-model.

Finally, this chapter introduces the problem of representing actual data conforming to a given domain model: in fact, the domain model describes the data types relevant to a specific application domain therefore the domain model is like a schema for the data, borrowing a terminology typical of databases modeling; the actual data are instances of the data types. This theme is addressed in 4.3.

## 4.1 Domain elements description

This model describes the application domain as a graph of concepts; already introduced entities are just one of the available concept types although they probably are the most distinctive and important. Each concept is described by means of its properties and other "tools" (i.e., property functions, attributes, access control lists).

The arcs correspond to different types of relationships among the concepts; one particularly interesting relationship is the *structural relationship* which conveys the "has a" relationships.

### 4.1.1 Concepts

Concepts are the nodes of the domain graph and can be of different types:

- *Entities* are used to describe the elements populating each application domain and are defined by a name and a set of properties; for example, in a domain named Warehouse designed to represent data for a logistics use case, could be useful to design an entity named Item with the properties Name, Code and Price and an entity named Order with the properties Code, Date, Items and TotalValue;

- *Enumerations* are special entities; are used to describe concepts with a predefined and limited set of values. These concepts typically represent knowledge that is not directly manipulated by the process but instead used as special data types; enumerations are used to create reusable "dictionaries" shared by entities, even among different domains, in order to assure consistency and uniformity throughout the model which, as a consequence, results as more compact and simpler to manage; a typical usage for enumerations is represented by and address entity that could leverage global Country, City and PostCode enumerations;

- *Data sources* are meant to be used as proxies for external systems as a way to directly integrate data provided from other systems in the model. With such a tool could be possible, for example, to keep updated entity values through a web service.

### 4.1.2 Properties

Properties are used to characterize each concept in the domain; each property represents a component of the concept. There are three different types of properties, namely:

- *Terminal Properties* are used to define properties with values of a specified "flat" non-structured data type (e.g., strings, numbers, dates, vectors); each type can be configured to meet specific entity requirements since the same string property can be shaped in different ways for different entities; more data types could be defined and used (e.g., complex numbers and time series);

- *Enumeration Properties* are used to define properties with values in a predefined and limited set establishing a connection with a domain-visible enumeration;

- *Entity Properties* are used to define properties based on an entity; are used to establish a connection with other domain entities.

Entities can be composed of any type of properties. Enumerations are described just by terminal properties; this limitation ensures that it is always possible to determine a value for the property. Each of this concept classes describes what features should posses an instance of this class to be considered valid.

### 4.1.3 Property functions

For each entity is possible to define custom functions that, taking as input an entity instance, return as result an enriched view on its selected properties.

Even thought more can be defined, two types of property functions were introduced:

- *Text Functions* are used to apply a template string to an entity instance; referencing particular properties is possible to obtain an interpolated representation of a subset of the instance data. The resulting output string could be assimilated to a compound property. For example, recalling the example introduced in 4.1.1, a text property function like the following could be employed to obtain a string describing an order.

```
[<Date>] <Code> : EUR <TotalValue>
```

- *Evaluation Functions* are used to apply a specific expression to an entity instance in order to obtain a result in return. This result in calculated instead of assembled. The resulting output string could be assimilated to a compound property. For example, the following function returns a boolean value stating whether the Value property exceeds the given threshold.

```
<TotalValue> > 1000
```

### 4.1.4 Attributes

To better characterize the domain entities it is possible to mark one or a set of its properties with attributes:

- *Unique.* Unary attribute. This attribute indicates that the entity or terminal property which is applied to must be considered as a "key" for the entity: as in databases theory a key is used to identify rows of the same table, a unique property is used to identify instances of the same entity; this means that cannot exist more than one instance with the same value for the property;
- *Required.* Unary attribute. This attribute is used to mark a certain property as required for an entity instance to be considered valid;

- *Label.* Unary attribute. This attribute marks the entity or terminal properties that should be used to "represent" an entity instance when presented in an application context; for example, a property representing the name of an entity instance is more suitable to be displayed that its unique identifying code;
- *Group.* Nary attribute. This is used to group different terminal properties that are related from a semantic standpoint; particular use cases could required this attribute, for example, to group the First and Last Name in a Person entity;
- *Bind.* Binary attribute. This attribute is used to declare a group of two enumeration properties in which the values of a controlled property depends on the values of the controlling one; for example, in a use case regarding an address, a Province enumeration controls the City enumeration;
- *Master.* Unary attribute. This attributes marks the controlling property in a bind group; in the previous example the Province enumeration;
- *Slave.* Unary attribute. This attributes marks the controlled property in a bind group; in the previous example the City enumeration;
- *Secure.* Unary attribute. This attributes indicates that the terminal property should be handled with particular security policies; typically this attribute is used to mark properties used to store passwords;
- *External.* Unary attribute. When imported from an external system, properties are marked with this attribute to point out that the corresponding value must be retrieved and saved.

Attributes can be used to implement special use cases (e.g., the bind attributes) and to clarify logical or semantic relationships among the properties of a given entity (e.g., the group attribute). More could be defined in order to add details to the domain model.

### 4.1.5 Relationships

Relationships, the arcs of the domain graph, can be of two types, namely:

- *Structural relationship.* This is the fundamental relationship and describes the "has a" relationship. It is used to highlight the components each element of the model is made of. This type of relationship is called "structural" because is used to define the structure of the application domain itself in regard to those features that characterize the domain and not to particular, and hence variable, requirements of a selected application scenario. This type of arc is always directed.

  When a "parent" node A is linked with a "child" node B through a structural arc directed from A to B, it means that A has, among its components, a set of elements of B type. The cardinality of the relationship is expressed with two numbers respectively indicating the minimum and the maximum size of the child set. A visual representation is shown in figure 4.1).

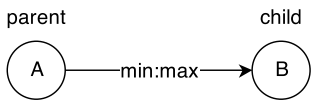


Figure 4.1: *Structural relationship*

  In more detail the structural relationship is actually used to link an entity property of the node A with the node B. In this regard it is like that node A "contains", as a property, a set of B nodes.

- *Semantic relationship.* When there is the need to describe a relation between two structurally disconnected groups of entities, is possible to define special arcs; different *colors* can be used to give arcs different meanings, accordingly to the specific application domain, in order to describe a wide range of relationships. For any other aspect this type of relationship is equivalent to the structural one.

### 4.1.6 Access control lists

Adopting an ACL-based security model, each concept in the domain is considered a resource and therefore is associated to a list of permissions. Each access rule indicates whether it refers to a group or a specific user, the name of the actual group or user and the permitted operations. This resembles the typical approach to file systems ACLs.

The types of interaction managed for this level is limited to read and write operations that translate in permissions to read and write instances of specific enumerations and entities.

*Ah hoc* rules are set for users qualified to edit the actual model but this subject is more deployment and platform oriented and therefore less relevant to this work.

### 4.1.7 Persistence

Persistence *per se* is not a problem directly addressed at this level, hence it is sufficient to indicate which concepts should make use of the storage layer with a persistence flag.

## 4.2 Formal representation

The formal representation of the domain meta-model is written using a slightly modified version of the Extended Backus–Naur Form (EBNF) [CO08] notation: usually used to describe context-free grammars, some elements were introduced to make it more suitable to describe sets. For this reason this representation is intended to be used only to offer a more organized view on the model elements.

More in detail, sets are represented by curly brackets; a *u* on the lower right indicates that the set is *unordered* while a *o* that is *ordered*. A plus sign on the upper right indicates that the set must have at least one item (or a precise number of elements if specified).

As in standard notation, nonterminal symbols are written in plain text while terminal symbols are in italic. In this particular case, terminal symbols are data types (e.g., *string*, *number*, *boolean*).

The meta-model is represented by the production rules reported in listing 4.1. Rule n.1 declares and describes the root element of the meta-model: each domain is identified by an id and consists of sets of different concepts: entities, enumerations, data sources. Entity groups provide a useful tool for entity management. Each entity (rule n.2) is identified by an id and consists of a non empty ordered set of properties, a set of property functions, a set of access rules and a flag which specifies how the instances of this entity must be treated regarding persistence (rule n. 9). Enumerations (rule n. 3) are designed to define sets of entities in a structured although "flat" domain ::= id, {entity}$^{+}_{u}$, {enumeration}$_{u}$, {dataSource}$_{u}$, {entityGroup}$_{u}$

entity ::= id, {property}$^{+}_{o}$, {propertyFunction}$_{u}$, {accessRule}$^{+}_{u}$, persistenceRule

enumeration ::= id, {terminalProperty}$^{+}_{o}$, {propertyFunction}$_{u}$, {accessRule}$^{+}_{u}$

dataSource ::= id, sourceType, *endpoint*, {*option*}$_{u}$, {externalInterface}$^{+}_{u}$

entityGroup ::= {entity.id}$^{2+}_{u}$

property ::= terminalProperty | enumerationProperty | entityProperty | semanticProperty

propertyFunction ::= id, {entity.property}$^{+}_{u}$, *script*

accessRule ::= accessProfile, accessName, accessLevel

persistenceRule ::= *boolean*

terminalProperty ::= id, type, {attribute}$_{u}$

enumerationProperty ::= id, enumeration.id, {attribute}$_{u}$

entityProperty ::= id, entity.id, {attribute}$_{u}$, cardinality

semanticProperty ::= entityProperty

cardinality ::= minOccurrence, maxOccurrence

id ::= *string*

type ::= *string* | *number* | *currency* | *vector* | *timeSeries* | ...

attribute ::= *unique* | *required* | *label* | *enumeration* | *group* | *bind* | *master* | *slave* | *secure* | *external* | ...

accessProfile ::= *group* | *user*

accessName ::= *string*

accessLevel ::= *r* | *w*

sourceType ::= *webService* | ...

externalInterface ::= {exportProperty}$^{+}_{o}$, {importProperty}$^{+}_{o}$

exportProperty ::= property.id

importProperty ::= property.id

minOccurrence ::= *integer*

maxOccurrence ::= *integer* | $\infty$

Listing 4.1: Domain formal representation

way: each element is an entity with only terminal properties and can be used to model sets. The enumeration access rules dictates whether it has a global scope (for general purpose enumerations) or a domain scope (for enumerations specific to the current domain). Each dataSource (rule n. 4) has an id, a sourceType (rule n.21) for proper interaction, an endpoint, a set of options to acknowledge specific parameters and a non empty set of externalInterfaces. An externalInterface, as stated in rule 22, represents input and output parameters of the remote service. Entity groups (rule n. 5) can be used to correlate two or more entities by means of their ids; at least two ids are required to define a group; the dotted notation *entity.id* refers to entities ids. A propertyFunction (rule n. 7) is a script defined on a subset of its entity properties and returning a terminal symbol, for instance a string value. A terminalProperty (rule n. 10) is a “flat”, non structured property described by an id and a type (described in rule n. 16); it’s a single value field even though it is possible to define advanced data types (e.g., complex numbers or time series). An enumProperty (rule n. 11) is a property defined through a domain or global enumeration. With this kind of property is possible to leverage on known sets of data; said sets can be defined once and used in any domain. An entityProperty (rule n. 12) is a property defined through another domain entity. Using this kind of property a structural relationship is defined between the two entities involved. With this approach the domain, regarding its entities portion, consists of at least one island of structurally connected entities. This kind of property can be imported with different cardinality (rule n. 14). Rule n. 13 defines a semanticProperty simply as an entityProperty; the rename comes from the necessity to define relations between two entities but not in a structural way; this allows to connect entity islands with different semantics compared to the structural ones in order to obtain, when useful, a completely connected graph. Cardinality (rule n. 14) is used to characterize the relation between the entity and each attribute: it’s important to define the number of property item each entity instance refers to. As described it’s possible to specify the minimum and maximum numbers.

## 4.3 Instances Model

The domain model, through its meta-model, enables the definition of entities, enumerations and so on but without offering any tool to define instances.

This layer should be kept distinct from an actual persistence layer in order to implement a decoupled approach. For this reason the concepts introduced in this section need to expand the domain model without trying to solve problems related to specific persistence techniques. The two problems are in fact very diverse: domain data must be represented in order to be read and edited without the need to to be persistently stored; optionally, this feature could be enabled for specific entities with the already introduced persistence flag.

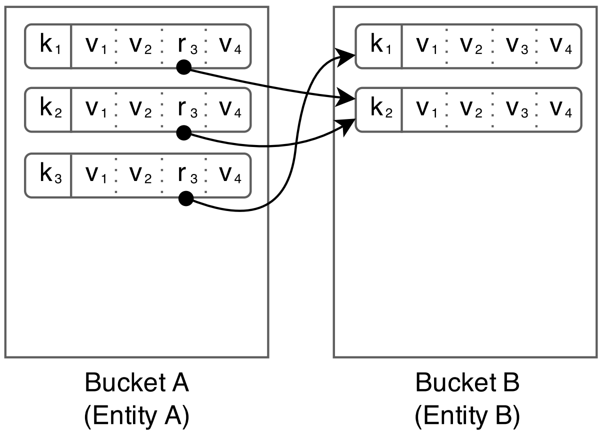
### 4.3.1 Buckets and Records

Each entity requires the creation of a proper container to hold its instances: the *Bucket*. Each bucket is characterized by its reference entity, which indicates which domain entity originated the bucket, and a set of instances. Each instance is represented by a *Record*; to uniquely identify each record in the bucket a key is required and introduced: the RecordKey. Typically, buckets behave like associative arrays and a key is required to retrieve the desired record.

Records are created to store instance data and therefore are composed by a set of values and a set of references. Values are used to store terminal properties values while reference are used to track other linked records; this is needed to follow the relationships defined through the use of entityProperty and semanticProperty elements. So, as entities are nodes of a directed graph in which arcs link containing nodes to the contained ones, records relative to the containing entity directly reference one or more records from the contained entity.

In figure 4.2 two generic buckets are visible; records are identified by their corresponding keys ($k_1$, $k_2$, $k_3$); each record is composed by values ($v_1$, $v_2$, $v_4$) and references ($r_3$). The references declare, in this particular example, that the records of the Bucket A with keys $k_1$ and $k_2$ reference the record with key $k_2$ of the Bucket B while the the record of the Bucket A with key $k_3$ reference the record with key $k_1$ of the Bucket B.

These aforementioned elements are described in terms of requirements in an abstract way to delegate to specific implementations any detail or choice.



$k_1$ $v_1$ $v_2$ $r_3$ $v_4$
$k_2$ $v_1$ $v_2$ $r_3$ $v_4$
$k_3$ $v_1$ $v_2$ $r_3$ $v_4$
$k_1$ $v_1$ $v_2$ $v_3$ $v_4$
$k_2$ $v_1$ $v_2$ $v_3$ $v_4$
Bucket A (Entity A)
Bucket B (Entity B)

Figure 4.2: *Instance model, buckets and records*

### 4.3.2 Bucket filtering

Buckets are per-entity homogeneous sets of records; a method to efficiently identify a subset of those records is required in order to interact with data in an effective way; for this reason a query system must be defined. Through this service would be possible to request particular instances satisfying one or more conditions; the service then returns a filtered bucket with just the corresponding records.

The proposed filtering approach is based on just two types of filter: the GreaterThan filter and the Equals filter. Both receive as input two parameters: the property path to be evaluated and a terminal value. The path could imply the need to follow records references in order to apply the filter to the desired terminal property. Is then possible to combine them through three logical operators: the AND, OR and NOT operators.

With this approach is possible to apply a compound filter to a bucket and obtain a subset of its records; each record is validated by conditions on the record itself and on any contained one.

For the sake of completeness, alongside the aforementioned QueryService, a WriteService is required to enclose all functionalities needed to create buckets and records.

### 4.3.3 Notes on persistence

This work is focused on the architectural framework enabling the creation of user interfaces so the problem of persistence, although important, is not explicitly addressed. Nonetheless could be useful to highlight some points:

- the instance model closely recalls the way NoSQL techniques approach the storage problem; the adoption of this style for the persistence layer could represent the more natural choice;
- relational databases use a different approach to data organization so some kind of adaptation process must be employed; the domain meta-model constraints help build models with limited variability; this trait could be exploited to implement automatic management techniques (e.g., the object-relational mapping programming technique).This aspect offers an interesting research topic;
- finally, the interaction with legacy systems is a topic of great value; a solution to this problem may be represented by the introduction of an abstraction layer exposing an SQL interface.

# Chapter 5
# Process Model

This model describes the actual workflow each business process implements. As previously stated in chapter 3, this model defines the exact sequence of activities and, for each presentation activity, the desired user interface elements from a structural point of view. Moreover, this model leverages the underlying domain layer in order to take advantage of its structured elements and bring domain level information (e.g., data types, entity contents, structural relationships) up to process definition level for further exploitation. In fact, one or more processes can be defined on top of a specific domain therefore importing and using its entities and others model concepts.

In this model are used typical elements of business process management theory and others that will be introduced in this chapter; the objective is to build a workflow enabling users, also represented by other processes, to interact with domain concepts in accordance with specific application requirements.

This chapter is dedicated to the description of how specific business process are represented and interact with domain models; this is obtained by means of a model, the Process Model; a new model must be defined when a new workflow is required, whether it requires interaction with a domain or not. In order to define the Process Model itself, which is actually an instance of all possible process models, a Process Meta-model is required; this chapter is dedicated to the description of this meta-model.

## 5.1 Process elements overview

This model describes the process as a graph of different types of elements. Said elements provide a declarative description of the tasks required to fulfill particular application needs and of the UI to be generated in order to enable the desired user interaction with data. All elements are introduced avoiding any coupling with other models elements (e.g., specific representation techniques).

To ease model elements comprehension is useful to provide a basic description of the main concepts a basic workflow and business process engine should possess delineating its execution model. The focus of this work is not in the business process management problem itself and therefore this topic is examined just as deeply as needed to provide a context for the main subject.

The supposed workflow engine can be described as similar to a Finite State Machine (FSM): a machine of this type can be in one of a finite number of *states* and can change its state when a triggering event is received or a certain condition is met; this is called a *transition*. The machine alternates execution states and wait states: during the former tasks are ran while during the latter the process is kept on hold until the required event is received. A representation of this behavior is reported in figure 5.1. When execution results have to persist outside of the origin task scope, variables should be used to hold generated values; this variables are said *operands*.

The workflow delineated in this work is "distributed" among different systems: when needed by the particular application scenario, a business process could require that part of it's activities are performed on a client system and not just in the main execution environment as already explained in 3.2.

As previously mentioned a process is represented as a graph, an activity diagram: nodes of this graph are the activities and will be executed by the workflow engine; directed arcs represent the conditions or events needed to trigger the transition required to reach the destination activity of each arc. The execution cycle of the activity diagram should not be confused with the workflow engine cycle on which it is run; the relation between the two is as follows: a process execution provides a sequence of activities; when activities require interaction with the user the workflow engine reaches a wait state and wait for the required signal to trigger the process transition; between

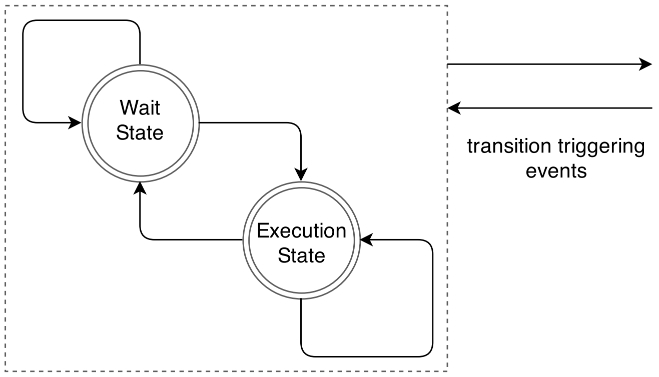


Figure 5.1: *Workflow engine life cycle*

the wait states the workflow engine in in execution state in which are run unsupervised tasks.

### 5.1.1 Activities

Activities represent the building blocks of any process and the nodes of the process model; the arcs represent the available transitions and therefore define the available "routes" that can be followed; which transitions will be actually executed is decided at runtime. Activities are of four basic types:

- the *Start activity* is required as first activity in the process; it is used by the workflow engine to start the execution loop;
- the *Execution activity* is the typical activity and represents a task to be completed before the workflow engine continues the executes loop triggering the related transitions;
- the *Wait activity* is used to hold the execution loop; after the associated tasks are completed the workflow engine enters the wait state until the required event is received or a certain conditions are met; at that point a transition is executed;
- the *Stop activity* is used to declare that the process has ended and should be terminated; each process requires this activity to be included once but many transitions could lead to it.

While Start and Stop activities are required to correctly implement the execution loop, they are not actually meaningful for the specific application. Execution and Wait activities on the other hand convey actual business logic; however these are just generic blocks that must be specialized.

In particular two are the activities that are prominent for this dissertation:

- the *Custom activity* is an execution activity that can be loaded with code and can interact with domain elements through the query and write services introduced in 4.3;
- the *Presentation activity* is a wait activity that is used to mark the need to present an interface to the user and wait until the required prerequisites for transition are fulfilled, typically user interaction; this activity must be configured with an abstract description of the activity itself from which, as will be explained, the interface is generated.

Obviously many other activities could be created as required by specifications and to implement specific tasks in relation to flow management (e.g., a split activity that creates two in parallel execution pipelines or a choice activity that leads the execution flow only on one of the available branches in accordance to some condition) or different actions (e.g., sending a message to an activity or another business process).

### 5.1.2 Blocks

As previously stated, presentation activities must be configured with an abstract description of its contents in order to avoid coupling with specific presentation technologies. This descriptions takes the form of a graph in which the nodes are called *blocks*; arcs represent the structural relationship existing between blocks in a similar way to the relationship defined for domain entities (as shown in 4.1.5).

Blocks can be of various types, each with a specific task ranging from data representation to logical evaluation:

- *Entity blocks* are used to present domain entities thus creating domain-level operands presentation blocks; each block of this type can be populated with a connected set of entities and a filter; also in this case, selected domain elements must be arranged in a tree in order to form a non ambiguous to visit structure; this sets can be considered as structured operands;

- *Process blocks* are used to define process-level operands presentation blocks; this type of blocks can be used to present data, loaded as a domain instance or produced during process execution, in different forms; to do so specialized process blocks can be defined as, for example:
  - *Field Blocks* are used to present single field data (e.g., a string, a number); must be configured with a reference domain data type or with a reference process operand; this could be useful, for example, to implement data validation;
  - *List Blocks* are used to present ordered sets of data; must be configured with a reference domain entity or data type;
- *Execution blocks* are used to define activity-level executable code blocks not destined for data presentation: instead of defining an execution activity at process level is possible to define a presentation activity with a variable number of execution blocks in it;
- *Signaling blocks* are used to create blocks that are configured to send messages to other blocks in the same activity or to an activity (more on this aspect is explained in 5.1.3).

Blocks are used to interact with domain elements and can be configured for usage as ouput as well as input elements in order to enable workflow of different application requirements.

The resulting activity model defines a hierarchy of blocks; the structural relationship should be intended as a way to visit the blocks: from a topological standpoint each activity should be built as an acyclic undirected graph, a tree, in which the first block is the root of the tree.

### 5.1.3 Signaling

Signaling is the method used by activities and blocks to broadcast events and send data to other domain elements. Exchanged messages are called *signals* and it is possible to identify different cases on the basis of different source/target combinations.

### Signals for transition triggering

Signaling can be used to trigger a transition in the process; as previously introduced in the process model, activities are "linked" by arcs that represent the available "routes" the execution can take. These routes are associated to particular events; when these events are generated the workflow engine can follow the matching route and execute the corresponding activity. Signals can be used to deliver those events.

Source of this type of communication can be either a custom activity or a presentation activity. After the execution of its tasks, each custom activity has to produce an output in the form of a signal; this is received by the workflow engine which will use this information to select the appropriate route; it is important to highlight the difference between the routes statically defined as arcs of the process model and the signals produced at runtime by each activity that will be used to select the path the execution will actually follow. The same approach can be used to send signals to other processes which are in a wait state.

Presentation activities can be configured with signaling blocks; such a block can send messages to its parent activity and therefore trigger a transition in a similar way as what custom activities do. The only difference is that presentation activities place the workflow engine in a wait state so this creates a "window" for user interaction.

In either cases the following activity could be a custom or a presentation activity. A chain of only custom activities could be used to build a conditional network of processing units while the use of presentation activities can add the required UI layer. A combination is also possible: a custom activity could precede or follow a presentation activity in order to pre-process or post-process data. Finally, the possibility for blocks to send their signals also to other processes can give further flexibility enabling processing and presentation tasks synchronization among different processes and devices.

For example, in figure 5.2, is reported an example process: each available route is labeled; the custom activity could produce a different signal depending on the result obtained by its task thus advancing the process execution to the Presentation Activity (OK) or to the End Activity (FAIL). Also, from the Presentation Activity, the user could activate the Retry or Quit signaling block.

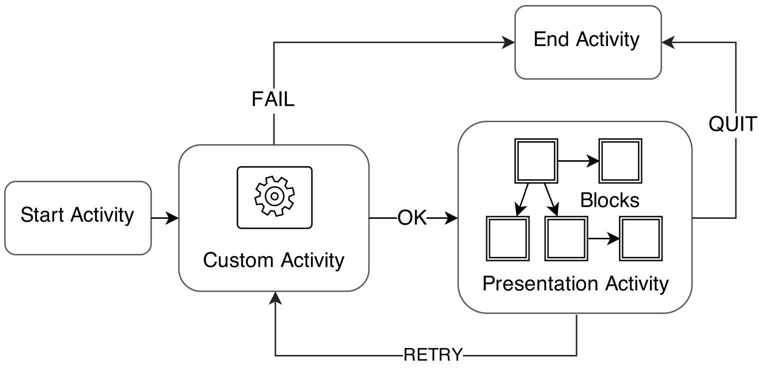


Figure 5.2: *Signaling for transition triggering*

### Block to block signaling

This type of communication occurs between blocks defined in the same activity and it is used to trigger block transitions or to move data among different blocks.

Since it is possible to define execution blocks it is also possible to create a workflow inside the activity in a similar way to what already explained for process-level workflow; signaling blocks are used to trigger also those transitions. In order to do so, signaling blocks can be configured with conditions that, when fulfilled, trigger the signaling action.

There is also the need to move data in order to implement activity-level manipulation. Signaling blocks are used to perform this action and, in this case, the communication payload is represented by an operand.

Interaction between the blocks is enabled by especially designed interaction event/action couples; in particular were designed four trigger events:

- the SELECT/DESELECT events couple is used to execute or negate a selection for a specific record in an entity block or a process block;
- the ADD/REMOVE events couple is designed to add or remove a record to or from an entity block or a process block.

For each of these verbs it is possible to define an action that is executed once the corresponding triggering event is received; with this technique it is possible to chain actions among the blocks.

In figure 5.3 is represented a presentation activity in which are depicted both approaches to inter-block signaling: the chain of execution blocks is triggered when the appropriate conditions are met; then an ADD action is performed on the process block; other two signaling blocks execute the selection and deselection action on the same process block.

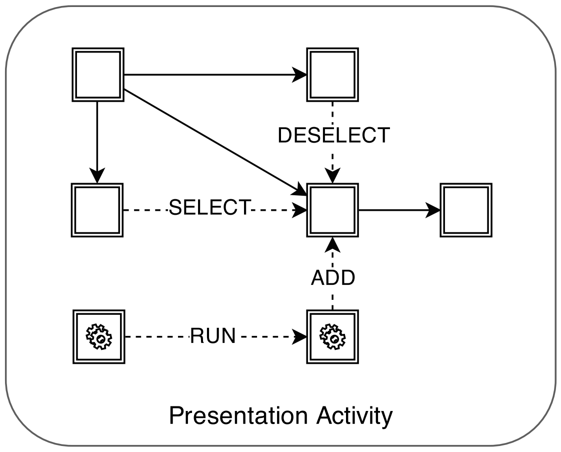


Figure 5.3: *Block to block signaling*

**Activity to block signaling**

For specific applications could be useful to present processing results as soon as they are generated and without an explicit action by the user. This type of interaction adhere to the “push” communication style which is contrasted with pull where the request for the transmission should be initiated by the receiver [BMvD07].

In this case, any custom activity could be the source of this type of communication; specific blocks could be used as receivers of push messages that could be presented to the user or trigger a side effect in the presentation activity workflow; the adoption of this communication adds flexibility and expressivity to the process model.

### 5.1.4 Distributed workflow

From the description of this model is apparent that the workflow is designed to run, when needed by the particular application scenario, on different classes of systems: a back-end system and a front-end system.

The back-end system is used to execute the main workflow, essentially composed by custom activities; the front-end system is primarily aimed at user interaction and therefore used for presentation activities. It is possible to describe the relationship between back-end and front-end systems as a client-server relationship; for this reason the main workflow can be called *server-side workflow.* The client is the system on which the presentation activity is executed and also represents the platform hosting the activity workflow which is then called the *client-side workflow.*

The combination of server-side workflow and client-side workflow forms the *distributed workflow.* In figure 5.4 is shown a process during execution: the back-end workflow is now in a wait state since the user is interacting with the presentation activity P on the client.

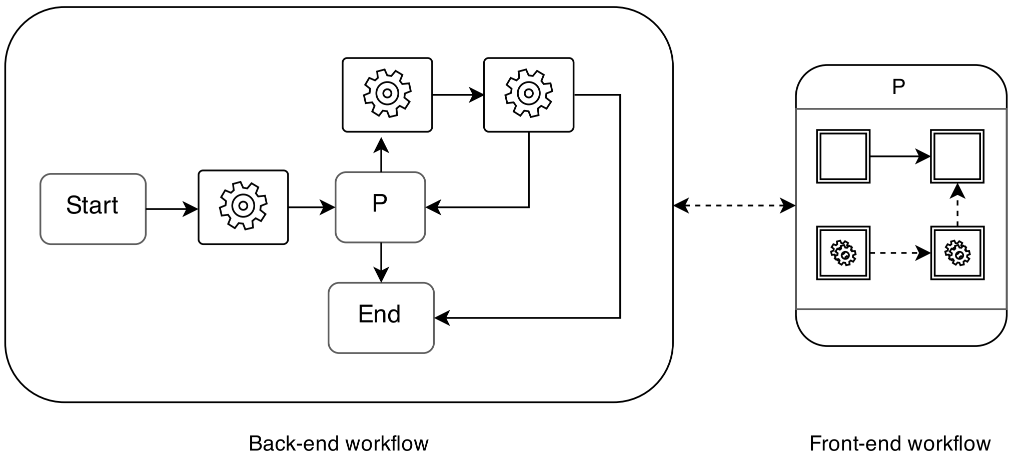


Figure 5.4: *Distributed workflow*

### 5.1.5 Access control lists

Adopting an ACL-based security model, each activity and block in the process is considered a resource and therefore is associated to a list of permissions. Each access rule indicates whether it refers to a group or a specific user, the name of the actual group or user and the permitted operations. This resembles the typical approach to file systems ACLs.

The types of interaction managed for this level are read, write and execution operations; the execution operation translates in the capability to trigger signaling while read and write operations are to be intended as an override, specific to the particular process, in respect of what stated by the domain model rules, which are imported in the process model.

*Ah hoc* rules are set for users qualified to edit the actual model but this subject is more deployment and platform oriented and therefore less relevant to this work.

## 5.2 Formal representation

This section is dedicated to a formal representation of the elements constituting the process model. The same considerations expressed in 4.2 are valid also for this section. The meta-model is represented by the production rules reported in listing 5.1.

Rule n.1 declares and describes the root element of the meta-model: each process is identified by an id and consists of sets of different activities, signals, operands and access rules; start and stop activities are required to implement the workflow entities. A really important element is the reference to a domain, through its id, enabling the use of domain concepts inside the process.

Rules n.2, 3 and 4 set the activity hierarchy; the relative specialized activities are the custom activity (rule n.5) and the presentation activity (rule n.6). In particular, the latter is described by means of a set of signals, an access rule and the relative presentation activity model (rule n.7) which defines a presentation activity as a set of blocks (rule n.8); the different types of blocks are described with rules n.9, 10, 11, 12, 13 and 14. Entity blocks are defined by an id and the id of the domain entity used to populate it as well as the filter identifying the required subset of records. Field blocks and list blocks don't require the filter since are process blocks populated at runtime; anyway they are characterized by the entity.id to enable records validation.

Operands described by rule n.15 are process operands defined from a type (rule n.17) which are shared with the domain model.

```
process ::= id, startActivity, {activity}$_u^+$, stopActivity, {operand}$_u$, {signal}$_u^+$, domain.id, accessRule
activity ::= executionActivity | waitActivity
executionActivity ::= customActivity | ...
waitActivity ::= presentationActivity | ...
customActivity ::= id, processingUnit, {activationRule}$_u^+$
presentationActivity ::= id, presentationActivityModel, {signal}$_u^+$, accessRule
presentationActivityModel ::= {block}$_u^+$
block ::= entityBlock | processBlock | executionBlock | signalingBlock
entityBlock ::= id, {domain.entity.id}$_u^+$, filter
processBlock ::= fieldBlock | listBlock | ...
fieldBlock ::= id, {process.operand.id}$_u^+$
listBlock ::= id, {domain.entity.id}$_u^+$
executionBlock ::= id, processingUnit, {activationRule}$_u^+$
signalingBlock ::= id, signal
operand ::= id, type, {attribute}$_u$, accessRule
activationRule ::= {condition}$_u^+$
type ::= string | number | currency | vector | timeSeries | ...
signal ::= id, accessRule
id ::= string
attribute ::= length | pattern | min | max | ...
accessRule ::= accessProfile, accessName, accessLevel
accessProfile ::= group | user
accessName ::= string
accessLevel ::= r | w
```

Listing 5.1: Process model formal representation

# Chapter 6

# Abstract User Interface Model

This model describes, in abstract terms, all the elements needed to implement a specific workflow and to shape the required user interface; all these elements are generated and composed on the basis of the contents of the domain and process models. As already introduced, given a single domain is possible to define a certain number of processes that can leverage the same application data context, the domain; on the other hand from each process only one abstract UI model will be generated even though, sharing the same domain model, some part may be actually very similar.

The abstract UI model represents a really important step for the framework: it works as a decoupling point since it captures the idea of a generic UI not bound to a specific target technology, platform or language. In a sense, it is possible to state that the abstract UI model potentially contains every possible interface for the specific business process. For this reason it is essential for meeting the multimodality requirement.

In order to define the Abstract User Interface Model itself, which is actually an instance of all possible abstract UI models, an Abstract User Interface Meta-model is required; this chapter is dedicated to the description of this meta-model.

## 6.1 Model generation

This model describes the user interface as a graph of abstract elements. Since it is used to generate any concrete user interface, its elements should represent any component needed to effectively present operands and to enable interaction with operands themselves and with the application in general. Aiming at being the base for the subsequent derivation of UIs of any implementation technology, the abstract model describe each component in terms of the intent of interaction with users and not focusing on the "appearance" of the final UI. Each element is also enriched with all the information required by the concrete application to execute its tasks; this means this model must incorporate all the references needed to obtain and send data to the server-side infrastructure. Finally, the client-side workflow defined at process level and introduced in 5.1.3 must be included to keep the model independent in the form of trigger events, messages and transitions.

Since the abstract model is based on both the domain and the process models, the procedure required to generate it has to deal with data produced at different levels: for this reason the domain model needs to be integrated with the activity structure defined in the process model in order to produce a single abstract model.

The process model, with its activities, represents the *macro workflow layer* of the process and defines the macro sections of the resulting application, regardless of the specific target technology employed. For each activity, the contained blocks are used to define the *micro workflow layer* executed in the UI itself. Inside each block and activity, domain concepts are used to populate and exploit the particular block features; this level could be seen as a *data layer* placed below both the workflow levels. In figure 6.1 is reported this three levels representation. The transformation process must take into account all these details in order to compile an effective abstract model of the application.

In relation to these three levels, the transformation process maps the user-defined elements to abstract elements of different types:

- activities are seen as macro steps in the global workflow execution; this steps can be seen as different *moments* inside the application; for example, a web-based graphical user interface could later translate these elements as different pages in its concrete model; in the abstract interface only the presentation activities will be modeled and must be

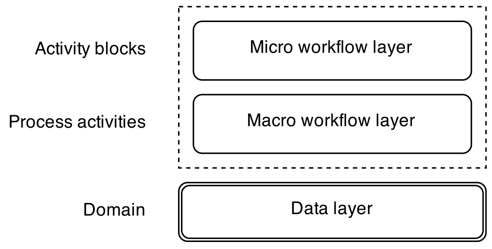


Figure 6.1: *Three levels structure*

kept track of the signaling needed to implement the transitions between every type of activity: even though the execution activities are not directly represented in the abstract model, this must contain all the information needed by the generated UI to trigger their execution;

- blocks inside each activity are used to structure the single activity; each block has a place in the hierarchy of the activity and this information must be preserved in order to be exploited during the following transformation phases and produce layout data; moreover, all client-side related aspects must be reflected in the abstract model in order, for the generated UI, to trigger transition execution and data flow; in general, blocks are of different types (as described in 5.1.2) and each must be treated accordingly to its function and to other information contained in the process model such as the relative ACLs; briefly, for each block type:
    - *Entity blocks* are used to present domain entities; this block is interpreted as a *abstract container* for entity records; said records are homogeneous since they are associated to the same domain entity and since the reference to the domain entity is kept at this abstraction level, it is possible to associate to this abstract container at least two *actions*: one to add and one to remove a record in the domain; both actions will be enabled or disabled on the basis of the corresponding process and domain ACLs; this actions will be treated as predefined micro workflow use cases;
    - *Process blocks* are used to define process-level operands presentation blocks; as already introduced in the same section of chapter 5

this type of blocks has been specialized in two types of blocks: *field* and *list blocks*; both are interpreted as abstract container but the associated actions depend on the specific micro workflow declared for the block: when the block represents the start of a client-side workflow chain an action is added to the block;

  – *Execution blocks* are used to define activity-level executable code blocks but, even though this block contents are not destined to be presented to the user, the associated code must be part of the micro workflow; this block is then interpreted as a *code container* and add an action to a block when activated after an user interaction;

  – *Signaling blocks* are used to create blocks that are configured to send messages to other blocks in the same activity or to an activity as explained in 5.1.3); this blocks are interpreted as actions which could interact with the micro or macro workflow;

- entities are used inside specific blocks to create structured operands; entities, and concepts in general, have a place in the hierarchy of the domain and this information must be preserved in order to be exploited during the following transformation phases and produce layout data.

## 6.2 Abstract UI Model elements

This section describes the elements constituting the generated abstract model. As already introduced and explained, this model represents an abstract description of a specific UI family: a group of concrete user interfaces all corresponding to the same process but tailored for different modalities and devices. Without bringing concepts described in other chapters forward, the abstract model must contain all the information needed to build the actual concrete interfaces expressed in terms of data, actions, formats, available interaction modes and so on in order to create an autonomous summary of the user interface.

All nodes in the abstract user interface model graph are either Abstract Interaction Objects (AIO) or Abstract Execution Objects (AEO); the former are used to describe abstract objects conveying information to and from the user while the latter are used to describe abstract objects especially designed to enable the delineation of a workflow at UI level, the client-side workflow described in 3.2 first and in 5.1.4 then. AEOs are characterized by a trigger event, selected among the ones defined in 5.1.3, or by a set of conditions.

AIOs are divided into three main categories: Data AIOs (DAIO), Action AIO (AAIO) and Layout AIOs (LAIO). DAIOs are used to represent abstract elements with a strong data connotation: they are used to reference a set of any size of domain records, through the corresponding query, or a process operand, entity or type based. AAIOs are used to represent action elements the user can activate to start an event chain inside the user interface or to perform an operation on a field/record; these elements can also be connected to AEOs to trigger the client-side workflow. LAIOs are used to combine several DAIOs and AAIOs enforcing a relationship among them; this tool can be used to group several DAIOs (e.g., when a certain set of fields need to be treated as a whole) or to group DAIOs with the corresponding AAIOs.

The arcs in the abstract user interface model graph describe the relationships among the different types of nodes. The connections that existed among the elements of the domain and process models are translated into the connections defined among the AIOs and the AEOs; once the model is compiled, the application is represented as a graph of abstract objects. The various types of elements are distributed according to the hierarchical and logical relationships among the abstract elements of the user interface; the patterns that form in the abstract model can be exploited in order to produce the concrete user interface model.

# Chapter 7

# Concrete User Interface Model

This model describes a user interface in terms of the UI components employed to shape the application represented by the process and domain models precedently used to automatically generate the abstract model; the concrete model is the next step in the framework methodology and it is obtained through automatic generation from the abstract model once the specific target UI technology has been set. From a single abstract model one or more concrete models can be derived, one for each supported target technology; it is possible to refer to this group of concrete interfaces as a UI family since all the members share the same abstract "seed" but each one is customized by the generation process to tackle different presentation and interaction modalities.

Being specific to a particular UI paradigm, this model has to describe the actual components the final application UI is built of; these components are the nodes of the graph constituting the model itself; the concrete model must incorporate all the information needed to instantiate and configure each UI component of the final application following the requirements expressed by the domain and process models.

In order to define the Concrete User Interface Model itself, which is actually an instance of all possible concrete UI models, a Concrete User Interface Meta-model is required; this chapter is dedicated to the description of this meta-model.

## 7.1 Model generation

This model describes the user interface as a graph of concrete elements. Since it is used to assemble the actual interface, this model should convey all the information needed to select the appropriate UI components and configure them in such a way as to allow the implementation of the application requirements in regards of both the front-end workflow and the interaction with the back-end workflow. In summary, the type of information enclosed in this model is similar to what is enclosed in the abstract model described in chapter 6. The main difference consists in the representation of the UI elements; while the abstract model adopts a description centered on the intent each model unit has in the context of the interaction between the user and the application, the concrete model aggregates such elements to identify the required UI components.

In fact, the generation process translates specific patterns found in the abstract UI model into a description of the UI components required to implement them, a set of configuration parameters needed by each component category and a description of the exchanged messages used to implement the client-side workflow. The aforementioned UI components are described with a loose-coupling approach: instead of pinpointing the exact concrete component to be used for the concrete user interface, this model declares a list of the required features the actual component should conform to, *de facto* identifying a component class instead of a component instance; this design choice leaves the various UI developers to implement the same requirement set with different UI components even for different target technologies. Moreover, this approach completely decouples the application requirements represented by the four models and the concrete UI itself which needs only an abstract description of its features to fully interact with the rest of the application.

Since the UI components are usually called widgets, these model elements will be called *abstract widgets*. The freedom provided by this additional abstraction level is exploited during the actual UI composition phase.

## 7.2 Basic mapping

The concrete UI model is generated from the abstract model and the applied transformation has to translate the knowledge already extracted from the domain and process models into layout knowledge; must be kept in mind that the concept of layout is to be intended in a broad way given the multimodal general approach of the framework.

The main idea on which the transformation is based consists in the translation of the hierarchical relationships set among the elements of the domain and process models by the corresponding developers into layout relationships. This information has already been used to create an abstract model of the UI through the automatic generation of the abstract user interface model. In particular, referring to the three levels structure introduced in 6.1 and focusing on the macro workflow and the data layer, it is possible to observe the following:

- as explained in 5.1.2, each activity is modeled as a graph of blocks and this structure defines a hierarchy among the blocks; these relationships represent the logical model the process designer defined for the single activity; an appropriate layout must be derived in order to preserve that vision; in this particular case, structural relationships among the blocks are used to infer a layout and identify sections in the presentation activity;

- blocks are of different types, some are invisible to the final user since are part of the client-side workflow; the remaining types are designed to enable data presentation and manipulation and are therefore defined by a reference domain data type. In particular, entity blocks are defined by a tree of domain entities; the structural relationships defined among the entities by the domain designer represents a hierarchy based on the “has a” relationships; process blocks are populated by records of a single domain entity and, even though no structural information among the concepts is used, can still leverage the information contained in the domain model;

- each entity is constituted by properties and attributes are used to characterize those properties; all this information can be exploited to derive an appropriate layout; in this particular case, focusing on a language for graphical UIs, the properties represent the actual data fields, the attributes contribute in defining a priority among the properties and the relationship between the entities are used to infer a layout for the group of fields.

In summary two main mapping strategies are taken into account, one for each of the authored models in the framework: at domain level the relationships defined among the concepts are used to derive a structure for the fields inside each widget while at process level the relationships among the blocks are used to derive a structure for the widgets inside each activity.

To generate the concrete user interface model a mapping between the aforementioned abstract model elements and the appropriate abstract widgets is required. The available abstract widgets are different for each target technology since each of them uses different interaction paradigms. The mapping function selects the abstract widgets required to cover each detected abstract model pattern and, when needed, compose them in special *widget containers* that are used to define the layout of the activity in accordance with the aforementioned logical/hierarchical relationships.

For the same group of elements found in the abstract model, one or more widgets could be required: the mapping function is not necessarily a one-to-one correspondence; moreover the number could change for different target technologies. The way each abstract widget or widget container is treated is decided during the actual UI composition phase. Moreover, each abstract interaction element should have at least one corresponding abstract widget to properly work: otherwise no resulting UI could be described; if an abstract element is mapped to only one abstract widget, the process is straightforward but if the mapping is one to many, it is possible to adopt a rule-based approach to select the most appropriate candidate or leave the decision to a specific user.

# Chapter 8

# Concrete User Interface

As explained in chapter 3 the framework aims at the production of multiple user interfaces from a declarative description of the data and the processes fulfilling the application requirements. To achieve this result, said models are used to automatically generate an abstract UI model and a concrete UI model. The concrete model is forwarded to the device the final user selected to access the application and is used as a blueprint for the actual UI composition leveraging the widgets available on the device at runtime. As explained in 3.2 the whole system can be divided in two main elements: the back-end system is devoted to all functionalities related to the actual generation process and to the services providing support for application business process execution while the front-end system, the client, is used to assemble the UI and enable user-interaction with the application.

This chapter is dedicated to the analysis of the approach to widgets composition and user interface creation executed on the aforementioned client device.

## 8.1 The client device

The client device represents the actual interface between the user and the process. To enable this interaction the client builds the UI using just the received concrete model as a blueprint; as explained in chapter 7, the concrete model contains a description of the UI in terms of abstract widgets. From this description the client composes the available *concrete widgets*, or simply *widgets*, and shapes the application. Different clients could support diverse UI and implementation technologies to serve the same process to different users.

The step that leads from the abstract widgets to the concrete widgets is really important for this framework: the concrete model is generated from the abstract model once the target technology is set; from the family of possible concrete models one is produced and forwarded to the client that requested it. This model defines the actual widgets to use to shape the interface; leveraging the loose-coupling approach described in 7.1, the client is able to comply with the requirements and select the most appropriate among a set of feature-equivalent widgets. The set of widgets the client has access to is obviously dependent on the specific target technology but could vary also on the basis of the particular device class: taking as example the world of the graphical user interfaces it is easy to understand that a web-based application should use different UI components if run on a desktop, on a tablet or on a smartphone.

This aspect can be seen as a first layer of context-awareness, from the point of view of the concrete model itself, offered by the framework. In this particular case, this context-awareness trait is used to support plasticity of the interface defined in the concrete model. Obviously there is a trade off between two possible solutions: using the same concrete model to target different devices or using different concrete models altogether; this kind of choice should be made on a case by case basis and taking into account the "distance" between the devices features.

## 8.2 Widgets

Final user interfaces are the result of the composition of UI units called widgets; this components are hosted on the client itself and are designed to accommodate different application needs; using the received concrete model as a blueprint, the client is able to select the more appropriate concrete widgets for each particular abstract widget.

The independence between the system producing the UI specification, the concrete model, and the system producing the final UI, the client, requires that the latter is ready to shape UIs with the same expressive level the concrete model has in defining them: an agreement among the two parts is required. For this reason the notion for a “complete” set of widgets called *theme* is required; a theme is a set of widgets able to cover all possible abstract widgets defined in any concrete model for the specific technology the client refers to. For this reason each client should host at least one theme for each supported target technology: even though client are usually tailored for a specific interaction paradigm, the same device could support different approaches to be accessible fr different user categories. While, for example, one theme could be enough for any voice-based client, several could be necessary for devices designed for graphical interfaces in order to comply with different users needs; each theme could be different simply in style or in the implementation of different accessibility policies or just in offering alternate versions of the same widgets to meet different users tastes.

As will be more apparent in the following sections, the employed widgets need to be especially developed components in order to integrate with the others framework elements.

### 8.2.1 Widgets architecture

To design these fundamental components with coherence and uniformity, all widgets share a common architecture. Consequently, this standardization also produces more easy to design and implement widgets. The chosen architectural style is the Model-View-Controller (MVC) pattern, always keeping in mind the multimodality context. Given the particular nature of the framework, the description in these terms of the widgets layer should be accompanied by the analysis of the full framework. Regardless of the specific architecture used to implement it, it is possible to describe the whole resulting system as an MVC organized as follows (as shown in figure 8.1):

- the Model is represented by the data layer containing the domains records and, in general, the system status; to enable interaction this layer offers the query and write services introduced in 4.3.2;
- the Controller corresponds to the workflow engine which executes the process and communicates with the other components and with other processes through a messaging system; interaction with the data layer is performed by the aforementioned write and query services;

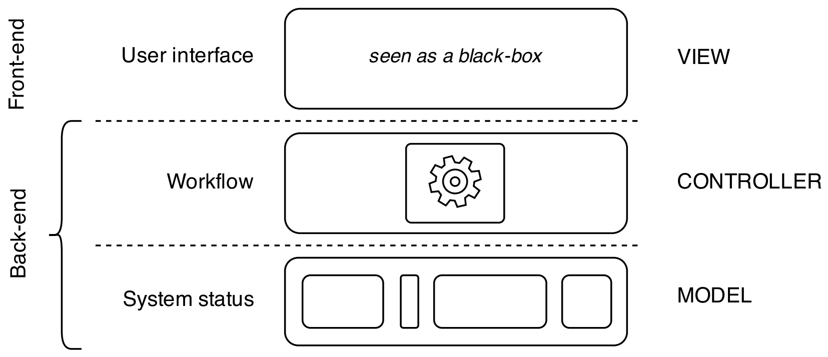


Figure 8.1: *Back-end Model-View-Controller*

- the client and the widget-based UI covers the role of the View thus completing the MVC architectural pattern; could be useful to note that this third component is actually optional from the point of view of the process itself: as already explained in 3.2, even though the main aim of the framework is to deliver a methodology for the production of applications with a UI, it is still possible to define processes without any presentation activity.

In the previous paragraph, the client has been represented as a black box; in more details, also this component adopts the MVC style; it is possible to map the three elements of this architectural pattern as follows (as shown in figure 8.2):

- the Model is represented by the back-end system storing the data produced by both the server-side workflow and the client-side workflow and providing the tools needed to read and write records and, in general, the system status;

- the Controller widget layer provides all the mechanisms required to implement the inter-widget communication, data exchange and view control; it offers a unified and coherent interface towards the view layer thus providing a tool to easily interact with a potentially extremely variegated world;

- the View widget layer is completely dedicated to providing the tools needed to enable the user interaction with data and with the application in general.

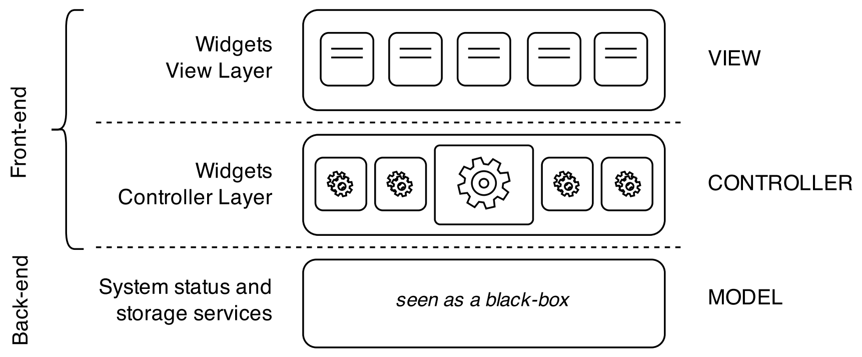


Figure 8.2: *Front-end Model-View-Controller*

The adoption of this pattern for the widgets layer is required by the multimodal nature of the framework: given the same abstract user interface model, a new concrete model must be generated for each target technology; the concrete model defines which abstract widget must be used; even though the abstract widget are coupled with a particular UI paradigm and thus possess specific parameters and configurations, it is easy to understand that different abstract widgets mapped on the same abstract pattern in the abstract model, share the same underlying "intent". This intent is tightly coupled with the way each widget behaves and therefore with the controller layer implementing that behavior each concrete widget is constituted by. This implies that it is possible to create widgets with the same controller but different look, or "skin". For example, a GUI component displaying a set of elements could be implemented as a vertical list of elements taking a limited amount of horizontal space or as a grid taking a large amount of both vertical and horizontal space and also capable of including more information for each element.

The Controller is specialized for each use case and, offering a similar interface towards the workflow and data levels, contributes in decoupling the specific target technology from the communication protocol, agnostic regarding the specific UI type, working as the sole endpoint for communications.

With this approach it is easier to develop and manage many sets of UI components in a multimodal scenario since many share a common controller.

Taking the two parts together, it is possible to describe the whole as a distributed MVC system in which the Model is represented by the data model, the client covers the role of the View while the Controller is distributed

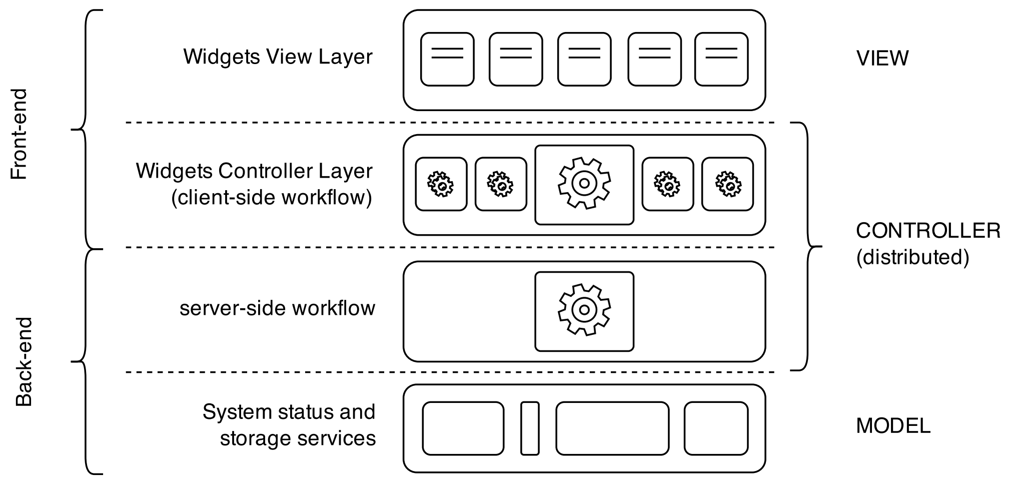


Figure 8.3: *Combined Model-View-Controller*

among the workflow engine and the controller layer of the composed widgets (as shown in figure 8.3); the server-side workflow and the client-side workflow, combined, work as an abstraction layer for the records contained in the data layer and, in general, the system status beyond the process execution and, ultimately, the UI.

Finally, regarding the completed application, each activity is perceived as a whole: the individual widgets that shape the interface form a graph of components and the result works as an interconnected structure with single controller and view layers.

With this approach it is possible to define different widgets sharing the same controller layer and consequently the same functionalities but with different view layers.

### 8.2.2 Low-tech clients

In this chapter the client was treated as an advanced device, able to parse the received concrete model, perform choices, select widgets among those stored on the client itself and finally compose the final UI. The rendered interface is run on device which should be able to execute the client-side workflow and support the interaction with the user.

All these operations require a minimum level of computational power which is delivered by most of the devices we typically use but that could be unavailable on more simple and lightweight hardware (e.g., wearable devices). Could be useful implementing a method capable of supporting this class of devices; thanks to the distributed nature of the framework, this goal could be easily achieved introducing an *ad hoc* component that works as a proxy for the actual device. This component could take on the responsibility of executing all the computationally onerous tasks and leaving just the essential functions to the device. In fact, the functionalities of the client, kept together in the framework description, are split among the proxy, which makes all the "hard" work, and the device which simply acts as an I/O module.

For example, a GUI-oriented low-capabilities device could be implemented with a raster UI whose only purpose is the display of the UI and the forwarding of the user-inputs to the proxy which, on the other hand, would be in charge of the actual rendering, the implementation of the client-side workflow and the interaction with the back-end.

## 8.3 Widgets composition

Should be already clear that the widgets described in this framework are fairly complex components: instead of being simple UI elements, they are designed as small applications encapsulating all the high-level UI features so important for delivering an effective and engaging user interface (e.g., usability, accessibility and context-awareness). This implies that said widgets, that it is possible to describe as *complex widgets*, require a composition technique in order to be shaped into an applications complying with specific functional and non-functional requirements.

These complex widgets must be written following the framework communication and interaction constrains in order to enable integration with other widgets and other components in general. As described in 8.2.1, widgets adopt a layered architecture: the View is completely customizable while the Controller is inherited from the Software Development Kit (SDK). Each target platform has a different SDK tightly coupled with the specific technology employed by the client and providing the tools needed to implement basic and advanced features.

With this approach, the widget developer is completely in control of all the aspects specifically related to the final user interface, e.g. being free to implement the most recent usability and accessibility practices, to adopt new engaging UI elements and paradigms and to follow the updated guidelines for the platform or technology in use periodically released by the platform vendors. Moreover, this freedom is achieved in the context of the automatic generation and composition framework drastically reducing maintenance and update costs thanks to its declarative approach in domain and process definition. This aspects are clearly really important in a field so susceptible to technological advancements and users tastes.

# Part IV

# Implementation

# Chapter 9

# Demonstrator

To test and validate the proposed framework a Java demonstrator was developed. For obvious reasons, it can't be considered a production system but can be used to highlight and show the main concepts introduced and described in the previous chapters. In particular the focus was placed on the generation and composition phases which constitute the core of the framework.

In summary, this demonstrator enables a user to *i*) define a domain with its entities and enumerations; *ii*) create records for the domain concepts; *iii*) design a process leveraging the domain elements composed of block of different types; and *iv*) use the generated interface to accomplish the tasks the process was designed for. All this operations are done programmatically and every piece of information is exchanged as a Java object without, for example, resorting to a serialized format. Moreover, this is an "in memory" system and every computation is volatile; configurations such as domains, processes and so on can be loaded from a file at system start.

## 9.1 Demonstrator modules

Several modules were developed to implement the demonstrator; a description of each of this modules is given in this section.

### 9.1.1 Domain repository

The Domain Repository module implements the tools needed to declare a domain through the definition of the corresponding model: leveraging a specific API, *de facto* reifying the domain meta-model described in chapter 4, it is possible to declare entities and enumerations; for each concept it is possible to define properties and configure said properties with attributes. In particular TerminalProperties are configured with type factories; from those the actual values will be generated. Each element is represented as a node in the domain graph while arcs correspond to structural or semantic relationships. Each element, domain included, is identified by a specific string handle. The repository allows the definition of multiple domains. In Listing 9.1 a simple domain definition is reported: entities, properties and entity functions are used.

Another API in the same module provides interfaces for the creation and management of buckets and records according to the instance model introduced in 4.3.1. In this module this aspect is implemented only with interfaces and abstract classes in order to delegate the concrete implementation to specific services and to keep separated the domain specification from the actual instance model. With this API, it is possible to define buckets, associated to specific entities or enumerations, and create records, identified by a RecordKey, to populate them. When a record is added to a Bucket, it is checked against the domain model for validation.

Finally, a third API provides the tools needed to define filters; these are used to select specific subsets of relevant records in a specific bucket. Composite filters can be defined through the combination, via the AND, OR and NOT logical operators, of the two basic EqualsFilter and GreaterThanFilter. The arguments of these basic filters are the path to the property, also nested, to be evaluated and the value used for comparison. This API provides a *fluent interface* to enhance code readability as apparent from the example reported in Listing 9.2.

### 9.1.2 Domain services

The Domain Services module implements the services needed to interact with records. In particular two interfaces were developed: the QueryService interface and the WriteService interface which are respectively used to read and write records grouped in buckets. From the generic Bucket interface

```java
Entity asset = new Entity("asset");
Property assetName = new TerminalProperty(
  "name", new TextTypeFactory()
);
asset.addProperty(assetName);

Entity person = new Entity("person");
Property authorName = new TerminalProperty(
  "name", new TextTypeFactory()
);
Property authorAsset = new EntityProperty(
  "asset", asset, new Cardinality(1, 1)
);
person.addProperty(authorName, authorAsset);

Entity event = new Entity("event");
Property eventDate = new TerminalProperty(
  "date", new DateTypeFactory()
);
Property eventValue = new TerminalProperty(
  "value", new TextTypeFactory()
);
Property eventAuthor = new EntityProperty(
  "author", person, new Cardinality(1, 1)
);
event.addProperty(eventDate, eventValue, eventAuthor);

TextFunction textFunction = new TextFunction(
  "label",
  "[<date>] Value is <value>, sampled by <author.surname>",
  String.class
);
event.addPropertyFunction(textFunction);

Domain log = new Domain("log");
log.addEntity(event, person);
```

Listing 9.1: Example domain definition

```java
Filter authorFilter = new EqualsFilter(
  "author.name", new TextType("John Doe")
);

Filter dateFilter = new GreaterThanFilter(
  "date", new DateType(new Date())
);

Filter filter = new FilterBuilder()
  .addFilter(authorFilter).addFilter(dateFilter)
  .notJoin().andJoin()
  .getFilter();
```

Listing 9.2: Example filter definition

defined in the domain module, a concrete ListBucket class has been implemented; the service ListBucketService implements the two service interfaces and uses the ListBucket as records container; in this particular case, the RecordKey has been implemented as the index of any record in the *in memory* ArrayList constituting the base structure of the ListBucket. In Listing 9.3 an example of records definition is reported. Moreover, an entity function defined in listing 9.1 is executed to generate a description of the selected record (line 18).

This module is also in charge of applying filters defined with the elements provided by the domain module. An example of usage of this feature is reported in Listing 9.4; the entity is inferred from the bucket and the specific service implementation, in this specific case the ListBucketService, applies the filter and returns a filtered bucket in response.

### 9.1.3 Process repository and workflow engine

This module implements the process repository and a basic workflow engine. Reifying the process meta-model described in chapter 5, this module provides the tools required to define processes as graphs of activities; aside from the required Start and End activities, it is possible to define Presentation activities and Custom activities; the former are used to define the abstract user interface model elements for the process while the latter are used to execute custom Java code. Each presentation activity is in fact modeled as a graph of blocks of different types: an EntityBlock is configured

```
ListBucketService service = new ListBucketService();

Record assetRecord = service.buildRecord(asset)
  .setField("name", new TextType("ASSET-001"));
service.addRecord(assetRecord);

Record personRecord = service.buildRecord(person)
  .setField("name", new TextType("John Doe"))
  .setField(authorAsset, assetRecord);
service.addRecord(personRecord);

Record eventRecord = service.buildRecord(event)
  .setField("date", new DateType(new Date()))
  .setField("value", new TextType("42"))
  .setField("author", personRecord)
service.addRecord(eventRecord);

String label = service.execute("label", eventRecord);
```

Listing 9.3: Example buckets and records creation

with a domain entity and is used to display records retrieved, with a filter if needed, from the corresponding buckets; a FieldBlock is configured to contain a value generated from a particular type factory; finally, a ListBlock can be used to display records of a given entity. An example activity definition is reported in listing 9.5;

It is also possible to define the client-side workflow as a set of event/action couples; four trigger events were implemented: ADD, REMOVE, SELECT and DESELECT; when a record is added or removed from a FieldBlock or ListBlock or when a record is selected or deselected in an EntityBlock, the corresponding action, called script, is triggered; two scripts were implemented: AddScript and RemoveScript. An example is reported in listing 9.6; In this particular case, when events are selected or deselected from the events block, the corresponding records are added to or removed from the shelf block; as a consequence, the value contained in those records is added

```
Entity event = log.getEntity("event");
Bucket filteredBucket = service.getBucket(event, filter);
```

Listing 9.4: Example filter usage

```java
Entity event = log.getEntity("event");

EntityBlock eventsBlock = new EntityBlock(
  new Id("Events"), log, service, service
);
eventsBlock.addEntity(event);

Block shelfBlock = new ListBlock(new Id("Shelf"), event);

Block sumBlock = new FieldBlock(
  new Id("Total value"),
  event.getTerminalProperty("value").getTypeFactory()
);
eventsBlock.addBlock(shelfBlock, sumBlock);
```

Listing 9.5: Example presentation activity definition

to or removed from the value contained in the sum block.

Moreover this module implements basic mechanisms required to start, execute and stop processes and to enable interaction between the active processes and the client controlling them. In fact, this module implements a basic process manager which works also as a repository.

### 9.1.4 Text-based user interface client

This module represents a text-based client; its main objective is to generate the user interface from the abstract model and enable the final user to interact with the process. In order to do so, this module implements a simple wrapper that connects to the manager to get the list of available processes and controls them. This particular implementation is based on the System.out Java interface and relies on a numeric input as a selection from a menu of actions displayed to the user after each update of the process state or the UI itself. A set of widgets implement the functionalities required to present the records to the user and the menu of available actions: for this particular case there is a one-to-one mapping between the blocks in the abstract model and the UI widgets (i.e., an EntityBlockWidget, a FieldBlockWidget and a ListBlockWidget were developed).

In summary, the UI wrapper talks with the process manager; the user selects the process and the managers starts its execution; when a presentation activity becomes active, the manager forwards the abstract activity model to

```
eventsBlock.addBlockAction(new BlockAction(
  TRIGGER_EVENT.SELECT, new AddScript(shelfBlock, null)
));
eventsBlock.addBlockAction(new BlockAction(
  TRIGGER_EVENT.DESELECT, new RemoveScript(shelfBlock, null)
));

shelfBlock.addBlockAction(new BlockAction(
  TRIGGER_EVENT.ADD, new AddScript(sumBlock, "value")
));
shelfBlock.addBlockAction(new BlockAction(
  TRIGGER_EVENT.REMOVE, new RemoveScript(sumBlock, "value")
));
```

Listing 9.6: Example presentation activity definition

the manager which maps the abstract elements on the available widgets and assembles the actual UI and a menu that can be used to interact with the UI. In particular, the menu offers commands for process navigation and for UI navigation; the former are used to signal the workflow engine and therefore to trigger the activity level transitions (described in 5.1.3); the latter are required since, for the particular nature of the UI paradigm adopted, only one block at the same time is shown to the user. In listing 9.7 a sample of a generated UI is reported; each block of text in the listing correspond to one UI widget for a total of 4 distinct widgets: the first block (lines 3-7) corresponds to the user authentication widget, the second and the third blocks (lines 9-15 and 17-23) to the main menu widget, the fifth block (lines 31-41) to the process menu and the fourth block (lines 25-29) to an EntityBlockWidget. Only this last widget is not from the wrapper required to manage the interaction; the EntityBlockWidget is derived from the abstract description of the presentation activity partly defined in listing 9.5 and populated with records defined in listing 9.3 and automatically loaded from the domain repository module.

## 9.2 Implemented framework features

Although the developed UI module is rather rudimentary and the architecture is not at production level, the demonstrator implements the core framework features needed for generation and UI composition. In particular

```
-------------------------------------------------------------
----------------------- WELCOME ----------------------------

USER AUTHENTICATION
Login: user
Password: ********

-------------------------------------------------------------

MAIN MENU
1) Start process: "Events manager"
2) Quit

user > 1

-------------------------------------------------------------

MAIN MENU:
1) View process: "Events manager"
2) Quit

user > 1

-------------------------------------------------------------

== EVENTS ==

1) [ ] [2014-12-11 10:02] Value is 42, sampled by Doe

-------------------------------------------------------------

PROCESS MENU
1) Send signal: "View selected events details"
2) Send signal: "Close"
3) Show block: "Shelf"
4) Show block: "Total value"
5) Select element: "1"
6) Back

user > _

```

Listing 9.7: Text-based UI sample

models described in chapters 4, 5, 6 and 7 a the respective main features were implemented and employed.

Every model works as a decoupling and data exchange point between the architecture levels; in particular, the information reaching the UI client is abstract and represents all the structural data needed to choose and configure the appropriate widgets and to assemble them into the UI; each widget is then capable of connecting to the domain repository to fetch the contextually relevant data even adopting a continuous query approach for live-update UI elements.

Widgets are effectively in charge of data presentation and interaction and an *ad hoc* development of advanced UI components is also correctly decoupled from domain and process models. This means that the generation process resolves in the widgets that are assembled on the basis of hierarchical relations and a UI developer can concentrate on the user-central themes and technologies with a minimal "distraction" from communication and data access problems.

# Conclusions

As stated by Meixner in [MPV11] the realization of an holistic Model-driven development process is a step of primary importance for the effective and efficient design and development of complex interactive systems in the future. This means that the UI development process has to be integrated in the development of the entire software system it belongs to. As a matter of fact, the research has focused its attention on the UI life cycle giving birth to methodologies and tools specific for the UI creation independently of the rest of interactive system. Model-driven engineering constitutes one of the main approaches used in the development of complex systems and even though this approach has been successfully adopted in the UI development there is still no integration of the two fields in a unique methodology.

This thesis addresses this research challenge defining a new methodology in the form of a framework for the automatic generation of user interfaces in the context of business processes development; the generated UIs are produced with a multimodal approach and with a methodology aiming at overcoming the limitations of the currently available solutions in delivering usable, adaptive and appealing user interfaces. This approach follows the model-driven paradigm and aims at defining the different steps that have to be performed in the process for the creation of an application and of the associated UI. Specifically, are defined the different users involved in the different steps and the models produced at the end of each step of the process.

The context for this thesis was selected after focusing on the available enterprise platforms possessing, among their features, the capability to render a user interface for their business processes; in particular, the attention was placed on the NEGENTIS Enterprise Software Platform [NEG]; the NE-

GENTIS Platform has been effectively adopted in various production and research projects such as the SIMOB Project in which it was employed to shape an InfoMobility Integrated Platform [GPCC13] and, more recently, the SITMar Project in which it has been used to provide innovative real-time services for goods monitoring in multimodal transport [ZCA$^{+}$14]. Even though the NEGENTIS Platform has been considered the reference environment for this thesis, it is important to emphasize that the present work constitues a general approach and it is also appliable in other contexts.

The proposed methodology in fact foresees the definition of a set of meta-models for the design of the various aspects of both the UI and the application and said methodology is synthesized in a specific framework covering both the design and the runtime phases of the multimodal and adaptive UI and global application life cycle. This framework describes a set of levels and each level addresses different aspects of the development process leveraging suitable representation models.

The framework is divided into four different abstraction levels and the instances handled by each level belongs to a specific model envisioned in the meta-model depicted in the present work. In particular were designed: a Domain Model, to represent all the concepts characterizing each application domain; a Process Model, to represent the tasks fulfilling the application requirements; an Abstract User Interface Model, automatically generated from the two previously introduced models and describing any possible user interface generated for the specific use case; a Concrete User Interface Model, automatically generated from the abstract model and describing the family of concrete user interfaces for a specific use case once a particular target technology has been chosen.

The envisioned production system is made of two elements: the back-end subsystem and the front-end subsystem. The design phase involves the creation of two artifacts, namely the Domain and the Process models, corresponding to different steps in the application definition process; these two models are created by two corresponding users: the domains designer and the processes designer; these two models are hosted on the back-end subsystem in two corresponding repositories. Once the design phase is concluded, the end user can access the desired application by means of a compliant client device, the front-end subsystem, and a set of services provided by the back-end subsystem; at this point the back-end "prepares" the business process execution and, knowing the characteristics of the client, is able to automatically generate the concrete user interface model and forward it to the client; the client uses it as a blueprint to assemble the final user interface from the set of complex widgets, the UI building blocks, available at runtime on the device itself; these complex widgets are components designed by a specific UI developer in accordance to the framework guidelines and loaded on the client.

To test and validate the proposed framework a Java demonstrator has been developed. This component has proven useful to show that was possible to create a user interface from the declarative description of the application domain and the required business process without any type of coupling with the framework models or with any particular technology.

Future works comprehend the production from the present work of a set of specifications and requirements for the design and implementation of a reference architecture. In particular, it is foreseen the implementation of the aforementioned specifications and requirements as an extension to the functionalities of the NEGENTIS Enterprise Software Platform.

# List of Figures

1.1 *The simplified Cameleon Reference Framework* . . . . . . . . . 9

3.1 *Logical levels and models* . . . . . . . . . . . . . . . . . . . . . . 35
3.2 *Architecture overview* . . . . . . . . . . . . . . . . . . . . . . . . 37

4.1 *Structural relationship* . . . . . . . . . . . . . . . . . . . . . . . 45
4.2 *Instance model, buckets and records* . . . . . . . . . . . . . . . 50

5.1 *Workflow engine life cycle* . . . . . . . . . . . . . . . . . . . . . 54
5.2 *Signaling for transition triggering* . . . . . . . . . . . . . . . . 58
5.3 *Block to block signaling* . . . . . . . . . . . . . . . . . . . . . . 59
5.4 *Distributed workflow* . . . . . . . . . . . . . . . . . . . . . . . . 60

6.1 *Three levels structure* . . . . . . . . . . . . . . . . . . . . . . . . 65

8.1 *Back-end Model-View-Controller* . . . . . . . . . . . . . . . . . 75
8.2 *Front-end Model-View-Controller* . . . . . . . . . . . . . . . . . 76
8.3 *Combined Model-View-Controller* . . . . . . . . . . . . . . . . . 77

# Listings

4.1 Domain formal representation . . . . . . . . . . . . . . . . . . . 47
5.1 Process model formal representation . . . . . . . . . . . . . . . 62
9.1 Example domain definition . . . . . . . . . . . . . . . . . . . . . 83
9.2 Example filter definition . . . . . . . . . . . . . . . . . . . . . . 84
9.3 Example buckets and records creation . . . . . . . . . . . . . . . 85
9.4 Example filter usage . . . . . . . . . . . . . . . . . . . . . . . . 85
9.5 Example presentation activity definition . . . . . . . . . . . . . 86
9.6 Example presentation activity definition . . . . . . . . . . . . . 87
9.7 Text-based UI sample . . . . . . . . . . . . . . . . . . . . . . . . 88

# Bibliography

[ABY14] Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. Adaptive Model-Driven User Interface Development Systems. *ACM Computing Surveys (CSUR)*, 47(1):9:1–9:33, May 2014.

[APB$^+$99] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: An Appliance-independent XML User Interface Language. *Comput. Netw.*, 31(11-16):1695–1708, May 1999.

[BBF$^+$87] Bill Betts, David Burlingame, Gerhard Fischer, Jim Foley, Mark Green, David Kasik, Stephen T. Kerr, Dan Olsen, and James Thomas. Goals and Objectives for User Interface Software. *SIGGRAPH Computer Graphics*, 21(2):73–78, April 1987.

[BCPS04] Silvia Berti, Francesco Correani, Fabio Paternò, and Carmen Santoro. The TERESA XML Language for the Description of Interactive Systems at Multiple Abstraction. In *Leveles, Proceedings Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages*, pages 103–110, 2004.

[Bel10] M. Bell. *SOA Modeling Patterns for Service Oriented Discovery and Analysis.* Wiley, 2010.

[BF14] Marco Brambilla and Piero Fraternali. Large-scale Model-Driven Engineering of web user interaction: The WebML and WebRatio experience. *Science of Computer Programming*, 89, Part B(0):71 – 87, 2014. Special issue on Success Stories in Model Driven Engineering.

[BMvD07] E. Bozdag, A. Mesbah, and A. van Deursen. A Comparison of Push and Pull Techniques for AJAX. In *Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on*, pages 15–22, October 2007.

[CCRR02] James L. Crowley, Joëlle Coutaz, Gaeten Rey, and Patrick Reignier. Perceptual Components for Context Aware Computing. In Gaetano Borriello and LarsErik Holmquist, editors, *UbiComp 2002: Ubiquitous*

*Computing*, volume 2498 of *Lecture Notes in Computer Science*, pages 117–134. Springer Berlin Heidelberg, 2002.

[CCT$^+$02] Gaelle Calvary, Joelle Coutaz, David Thevenin, Quentin Limbourg, Nathalie Souchon, Laurent Bouillon, Murielle Florins, and Jean Vanderdonckt. Plasticity of User Interfaces: A Revisited Reference Framework. In *In Task Models and Diagrams for User Interface Design*, pages 127–134. Publishing House, 2002.

[CCT$^+$03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A Unifying Reference Framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289 – 308, 2003. Computer-Aided Design of User Interface.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. Citeseer, 2003.

[CH06] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[CO08] D. Crocker and P. Overell. Augmented BNF for Syntax Specifications: ABNF. RFC 5234 (INTERNET STANDARD), January 2008.

[Cor06] James R Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[CR02] Joëlle Coutaz and Gaëtan Rey. Foundations for a Theory of Contextors. In *Computer-Aided Design of User Interfaces III*, pages 13–33. Springer Netherlands, 2002.

[Dey00] Anind Kumar Dey. *Providing Architectural Support for Building Context-aware Applications*. PhD thesis, Atlanta, GA, USA, 2000. AAI9994400.

[Erl05] T. Erl. *Service-oriented architecture: concepts, technology, and design*. The Prentice Hall Service-Oriented Computing Series from Thomas Erl Series. Prentice Hall Professional Technical Reference, 2005.

[EVP01] Jacob Eisenstein, Jean Vanderdonckt, and Angel Puerta. Applying Model-based Techniques to the Development of UIs for Mobile Computers. In *Proceedings of the 6th International Conference on Intelligent User Interfaces*, IUI '01, pages 69–76, New York, NY, USA, 2001. ACM.

[Flo06] Murielle Florins. *Graceful Degradation: a Method for Designing Multi-platform Graphical User Interfaces*. PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, July 2006.

[Fon10] José Manuel Cantera Fonseca. Model-Based UI XG Final Report, 2010.

[FV10] David Faure and Jean Vanderdonckt. User Interface Extensible Markup Language. In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '10, pages 361–362, New York, NY, USA, 2010. ACM.

[GGGCVMA09] J. Guerrero-Garcia, J.M. Gonzalez-Calleros, J. Vanderdonckt, and J. Muoz-Arteaga. A Theoretical Survey of User Interface Description Languages: Preliminary Results. In *Web Congress, 2009. LA-WEB '09. Latin American*, pages 36–43, November 2009.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GLCV08] Josefina Guerrero García, Christophe Lemaigre, Juan Manuel González Calleros, and Jean Vanderdonckt. Model-Driven Approach to Design User Interfaces for Workflow Information Systems. 14(19):3160–3173, November 2008.

[GPCC13] D. Giuli, F. Paganelli, S. Cuomo, and P. Cianchi. Toward a Cooperative Approach for Continuous Innovation of Mobility Information Services. *Systems Journal, IEEE*, 7(4):669–680, Dec 2013.

[Hix90] Deborah Hix. Generations of User-Interface Management Systems. *IEEE Softw.*, 7(5):77–87, September 1990.

[HPBL00] Mare Hassenzahl, Axel Platz, Michael Burmester, and Katrin Lehner. Hedonic and Ergonomic Quality Aspects Determine a Software's Appeal. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '00, pages 201–208, New York, NY, USA, 2000. ACM.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Sci. Comput. Program.*, 72(1-2):31–39, June 2008.

[JK06] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*, MoDELS'05, pages 128–138, Berlin, Heidelberg, 2006. Springer-Verlag.

[Kay07] Michael Kay. XSL transformations (XSLT) Version 2.0. W3C recommendation, W3C, jan 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[LVM$^+$05] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. USIXML: A Language Supporting Multi-path Development of User Interfaces. In Rémi Bastide, Philippe Palanque, and Jörg Roth, editors, *Engineering Human Computer Interaction and Interactive Systems*, volume 3425 of *Lecture Notes in Computer Science*, pages 200–220. Springer Berlin Heidelberg, 2005.

[MDZ13] Jovanović Mlađan, Starčević Dušan, and Jovanović Zoran. Languages for model-driven development of user interfaces: Review of the state of the art. *Yugoslav Journal of Operations Research*, 23(3):327–341, 2013.

[MHP00] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction - Special issue on human-computer interaction in the new millennium, Part 1*, 7(1):3–28, March 2000.

[MPV11] Gerrit Meixner, Fabio Paternò, and Jean Vanderdonckt. Past, Present, and Future of Model-Based User Interface Development. *i-com*, 10(3):2–11, 2011.

[MSB11] Gerrit Meixner, Marc Seissler, and Kai Breiner. Model-Driven Useware Engineering. In *Model-Driven Development of Advanced User Interfaces*, pages 1–26. Springer, 2011.

[MSUW04] S. J. Mellor, K. Scott, A. Uhl, and D. Weise. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley, New York, 2004.

[Mye87] B. A. Myers. Gaining General Acceptance for UIMSs. *SIGGRAPH Comput. Graph.*, 21(2):130–134, April 1987.

[NEG] NEGENTIS s.r.l. Official website: http://www.negentis.com/.

[Pat99] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, London, UK, UK, 1st edition, 1999.

[Pat05] Fabio Paternò. Model-based tools for pervasive usability. *Interacting with Computers*, 17(3):291–315, 2005.

[PE02] Angel Puerta and Jacob Eisenstein. XIML: A Common Representation for Interaction Data. In *Proceedings of the 7th International Conference on Intelligent User Interfaces*, IUI '02, pages 214–215, New York, NY, USA, 2002. ACM.

[PFRK12] Roman Popp, Jürgen Falb, David Raneburger, and Hermann Kaindl. A Transformation Engine for Model-driven UI Generation. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 281–286, New York, NY, USA, 2012. ACM.

[Pic00] R.W. Picard. *Affective Computing*. MIT Press, 2000.

[PMM97] Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, INTERACT '97, pages 362–369, London, UK, UK, 1997. Chapman & Hall, Ltd.

[PRK13] Roman Popp, David Raneburger, and Hermann Kaindl. Tool Support for Automated Multi-device GUI Generation from Discourse-based Communication Models. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 145–150, New York, NY, USA, 2013. ACM.

[PSM+08] Fabio Paternò, Carmen Santoro, Jani Mantyjarvi, Giulio Mori, and Sandro Sansone. Authoring Pervasive Multimodal User Interfaces. *Int. J. Web Eng. Technol.*, 4(2):235–261, May 2008.

[PSS09] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 16(4):19, 2009.

[Pue96] Angel R. Puerta. The MECANO Project: Comprehensive and Integrated Support for Model-Based Interface Development. In *Computer-Aided Design of User Interfaces I, Proceedings of the Second International Workshop on Computer-Aided Design of User Interfaces, June 5-7, 1996, Namur, Belgium*, pages 19–36, 1996.

[RPV12] David Raneburger, Roman Popp, and Jean Vanderdonckt. An Automated Layout Approach for Model-driven WIMP-UI Generation. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 91–100, New York, NY, USA, 2012. ACM.

[SBM07] Robbie Schaefer, Steffen Bleul, and Wolfgang Mueller. Dialog Modeling for Multiple Devices and Multiple Interaction Modalities. In *Proceedings of the 5th International Conference on Task Models and Diagrams for Users Interface Design*, TAMODIA'06, pages 39–53, Berlin, Heidelberg, 2007. Springer-Verlag.

[Sch96] Egbert Schlungbaum. Model-based User Interface Software Tools Current state of declarative models. Technical report, Graphics Visualization and Usability Centre, Georgia Institute of Technology, GVU Tech Report, 1996.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[SVM08] A. Stanciulescu, J. Vanderdonckt, and T. Mens. Colored graph transformation rules for model-driven engineering of multi-target systems. In *Proceedings of the third ACM international workshop on Graph and model transformations GRaMoT 08*, pages 37–44. ACM, 2008.

[Tha84] Alan L. Tharp. The Impact of Fourth Generation Programming Languages. *SIGCSE Bull.*, 16(2):37–44, June 1984.

[Van05] Jean Vanderdonckt. A MDA-compliant Environment for Developing User Interfaces of Information Systems. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, CAiSE'05, pages 16–31, Berlin, Heidelberg, 2005. Springer-Verlag.

[VB93] Jean Vanderdonckt and François Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 424–429. ACM, 1993.

[ZCA$^+$14] I. Zappia, P. Cianchi, G. Adembri, M. Gherardelli, D. Giuli, and F. Paganelli. SITMar project: An integrated platform for goods monitoring in multimodal transport. In *Euro Med Telco Conference (EMTC), 2014*, pages 1–6, Nov 2014.