Università degli Studi di Milano–Bicocca

SCUOLA DI DOTTORATO DI RICERCA IN SCIENZE
Dottorato di Ricerca in Informatica
XXV Ciclo
Coordinatore: Ch.ma Prof.ssa Stefania Bandini

Making End-Users Autonomous in the Design of their Active Documents

Iade Gesso
Ph.D Dissertation

Tutor: Ch.mo Prof. Giorgio De Michelis
Supervisor: Ch.ma Prof.ssa Carla Simone

Anno Accademico 2011 – 2012

Ai miei genitori e ai nonni,
ai miei fraterni amici,
e al nostro grande mentore Galileo

To my parents and grandparents,
to my brotherly friends,
and to our great mentor Galileo

Io stimo più il trovar un vero, benché di cosa leggiera, che 'l disputar lungamente delle massime questioni senza conseguir verità nissuna.

Galileo Galilei

I greatly esteem to find a truth, even though of a light thing, than a lengthy discussion on the maximum issues without achieving any truth.

Galileo Galilei

Ci sono solo due modi di vivere la propria vita: uno come se niente fosse un miracolo; l'altro come se tutto fosse un miracolo.

Albert Einstein

There are only two ways to live your life. One is as though nothing is a miracle. The other is as though everything is.

Albert Einstein

Un uomo fa quello che è suo dovere fare — quali che siano le conseguenze personali, quali che siano gli ostacoli, i pericoli o le pressioni — e questa è la base di tutta la moralità umana.

J. F. Kennedy - Spesso citata da Giovanni Falcone

A man does what he must — in spite of personal consequences, in spite of obstacles and dangers, and pressures — and that is the basis of all human morality.

J. F. Kennedy - Often mentioned by Giovanni Falcone

Acknowledgements

In primo luogo, vorrei ringraziare la Prof.ssa Carla Simone, che ha reso possibile il lavoro di ricerca che viene presentato in questa tesi. Sono determinato a far tesoro di suoi consigli e l'esperienza che ho acquisito nel corso degli ultimi tre anni.

Devo la mia gratitudine anche al Prof. Giorgio De Michelis per essere stato il primo che mi ha proposto di intrapprendere gli studi di dottorato e per avermi sostenuto negli ultimi tre anni con i suoi discorsi. Sono inoltre grato alla Prof.ssa Alessandra Agostini e al Prof. Huu Le Van per avermi incoraggiato prima di iniziare i miei studi di dottorato e durante il loro svolgimento.

Ultimo ma non meno importante, voglio ringraziare il Dott. Ing. Federico Cabitza per avermi presentato alla Prof.ssa Carla Simone e per i suoi utili consigli su studi di dottorato, la lingua inglese e la vita accademica.

Infine, voglio ringraziare tutte le persone che hanno accettato di sottoporsi alle

First and foremost, I would to thank Prof. Carla Simone, who made possible the research work that is presented in this thesis. I'm determined to treasure her's advices and the experience that I gained in the last three years.

I owe my gratitude also to Prof. Giorgio De Michelis for being the first one who proposed me to attend the doctoral studies and for having supported me in the last three years with his talks. I am also grateful to Prof. Alessandra Agostini and Prof. Huu Le Van for having encouraged me before starting my doctoral studies and during them.

Last but not least, I want to thank Dr. Eng. Federico Cabitza for having introduced me to Prof. Carla Simone and for its useful tips about doctorate studies, English language and accademic life.

Finally, I want to thank all people that accepted to undergo to the test sessions to

sessioni di test per validare il mio lavoro.

Ed ora, è giunto il momento di passare al lato personale della mia vita professionale!

Sono estremamente grato a Julia Weekes per la sua dolcezza e per il suo prezioso aiuto con la lingua inglese.

Vorrei ringraziare le persone che ho conosciuto presso lo Xerox Research Centre Europe (XRCE) per i tre mesi felici che ho trascorso con loro. Sentiti ringraziamenti ad Antonietta, Gabriela e Mario, così come a Patrick, Christophe, Yves e Juan Antonio. Sentiti ringraziamenti anche a Celine, Christine, Marie, Pajolma, Karine, Zeinep, Douglas e tutti i miei colleghi in XRCE.

Un grande ringraziamento ai miei colleghi del personale tecnico e amministrativo dell'Università. In particolare, vorrei ringraziare Franca, Rossella, Carmela, Eleonora, Rita e Teresa così come Claudio, Daniele, Davide, Luca, Marco, Maria Grazia, Michele e Kaname. I miei più sentiti ringraziamenti alla mitica Tina.

Ultimo ma non meno importante, vorrei ringraziare i miei colleghi di dottorato Elisa, Elisabetta, Emanuele e Giorgio.

Infine, è giunto il momento di ringraziare chi fa parte della mia vita privata.

Prima di tutto, vorrei ringraziare i miei genitori, Dilva e Vito, e tutti i miei nonni, Dina e Paolo, così come Anna, Iolanda e Giuseppe (anche se questi ultimi non sono più con me). Un enorme grazie anche ai miei zii, Carla e Paolo.

validate my work.

And now, it is time to turn to the personal side of my professional life!

I am extremely grateful to Julia Weekes for her sweetness and for her invaluable help with English language.

I would to thank people that I have known at the Xerox Research Centre Europe (XRCE) for the happy three months that I spent with them. Heart-felt thanks to Antonietta, Gabriela and Mario, as well as Patrick, Christophe, Yves and Juan Antonio. Heart-felt thanks also to Celine, Christine, Marie, Pajolma, Karine, Zeinep, Douglas and all my colleagues at XRCE.

A big thanks to my colleagues in the technical and administrative staff of the University. In particular, I would to thank Franca, Rossella, Carmela, Eleonora, Rita and Teresa as well as Claudio, Daniele, Davide, Luca, Marco, Maria Grazia, Michele e Kaname. My heartfelt thanks to the mythical Tina.

Last but not least, I would to thank my doctoral colleagues Elisa, Elisabetta, Emanuele and Giorgio.

Finally, the time to thank who is part of my private life has come.

First of all, I wish to thank my parents, Dilva and Vito, and all my grandparents, Dina and Paolo, as well as Anna, Iolanda and Giuseppe (even if the latters are no longer with me). An enormous thanks also to my aunt and uncle, Carla and Paolo.

Un immenso grazie a tutti i miei amici, in particolare Antonella, Chiara, Chreselene, Elena, Mariangela, Martina, Stefania, Ushas e Veronica, Fabione e Fabietto, Gianluca, Luca, Marco, Paolo e Salvatore. Lo stesso vale per Jennifer, Rosaria e Fabio così come Francesca, Stefania e Massimo. Un caro grazie a te che sei lassù, Michael...

I miei più sentiti ringraziamenti ai miei tesisti, Euro, Mario, Matteo, Micael e Stefano. Sentiti ringraziamenti anche a Anisa, Elena, Enrica, Andrea e Marco così come ad Andrea e Gabriele.

Infine, grande grazie a te, Giulia, che mi fai sempre rilassante quando ti guido.

A huge thanks to all my friends, especially Antonella, Chiara, Chreselene, Elena, Mariangela, Martina, Stefania, Ushas e Veronica, Fabione e Fabietto, Gianluca, Luca, Marco, Paolo e Salvatore. The same goes for Jennifer, Rosaria and Fabio as well as Francesca, Stefania and Massimo. A beloved thanks to you that you're up above, Michael...

Warmest thanks to my graduating students, Euro, Mario, Matteo, Micael and Stefano. Warmest thanks also to Anisa, Elena, Enrica, Andrea and Marco as well as Andrea and Gabriele.

Finally, big thanks to you, Giulia, who always make me relaxing when I am driving you.

Wednesday 31th October, 2012

Contents

Introduction

Documents are a fundamental player in professional practices. Since the introduction of computer-based systems and the related digitization of documents, an ever increasing number of organizations considered the full transition to digital documents as the way to successfully solve the problem of managing the exponential flow of information that they generate while performing their activities (see Berry and Goulde [1994]). However, despite the promise of a more efficient management of information flows, this "imposed" transition to digital documents has not been painless. Usually, in organizations the employees work together to reach a common goal, i. e., the goal of the organization to which they belong, and during time they develop and constantly maintain some unwritten conventions that support them in performing their collaborative activities. For instance, particular signs or annotations, with commonly-agreed meanings, can evoke specific conventions in the mind of the employees, helping them to behave accordingly. Several field studies proved that in these work settings documents are a fundamental part of these conventions and, at the same time, these studies highlighted the problems that arose after the transition to a document-based system (e. g., see [Braa and Sandahl, 1998, 2000]). Actually, the extreme flexibility of paper-based documents is lost with the transitions to their digital counterparts, since employees can not autonomously adapt both the informative content and the arrangement of their digital documents (e. g., see [Morrison and Blackwell, 2009; Chen and Akay, 2011]). As a consequence, the limits of document-based systems hinder their acceptance among employees, with the risk of nullifying the advantages that the organization management expected with the transition to digital documents.

In order to avoid this kind of problems, collaborative systems should be conceived to not force users to change their work habits. In particular, to be really effective, these

systems should provide users with (i) a flexible support for their work practices, taking into account the existence of local and informal conventions and proactively promoting *collaboration awareness* among users, and (ii) a customizable environment that they can autonomously adapt to their actual and constantly evolving needs. These two requirements are at the basis of the work that is presented in this thesis, which focuses on document-based collaborative systems.

In order to meet such requirements in the design of this kind of applications, this work puts its roots in both the *Computer Supported Collaborative Work (CSCW)* and *End-User Development (EUD)* research fields. These research fields are both characterized by the fact of being strongly multidisciplinary. Moreover, these two fields of research are not uncorrelated with each other, since the EUD research field draws from the tenets of the CSCW research field. The CSCW[1] was born in the early 80s of the past century (see [Grudin, 1994]) as a community of multidisciplinary researchers that were interested in understanding the nature of cooperative work settings and how to support them with effective computer based systems [Schmidt and Bannon, 1992]; in other words, «how collaborative activities and their coordination can be supported by means of computer systems [Carstensen and Schmidt, 1999]». The multidisciplinarity of the CSCW research field has a positive influence on the design of collaborative systems, since the principles that inform the involved research fields (e. g., computer science, human-computer interaction and sociology) allow the design of systems focused on how to effectively and unobtrusively support those employees that need to cooperate in order to perform their work activities (see [Schmidt, 1991]). To this aim, CSCW researchers focus on *understaning* the real work practices of employees and, consequently, on applying the results of their empirical studies to conceive and design computer-based systems aimed at providing employees with a *better support* for their collaborative work activities without disrupting their work habits and conventions (see [Bannon and Schmidt, 1991]). On the other hand, the EUD research field is younger than CSCW. Like the case of the CSCW, even the EUD research is strongly based on a multidisciplinary approach. Drawing on concepts, notions and findings coming from an heterogeneous set of research traditions, such as *Human-Computer Interaction (HCI)*, cognitive science, requirements engineering and, as mentioned, CSCW (see [Lieberman et al., 2006a; Klann et al., 2006]), the EUD research activities focus on the main goal of empowering end-users to autonomously customize the software system that they use in order to meet their

[1] The term *Computer Supported Collaborative Work (CSCW)* was firstly introduced in 1984 by Irene Greif and Paul Cashman in a workshop that they organized, in which participants shared the interest in how employees work and what role should have the technology within work environments [Grudin, 1994].

actual needs. Spreadsheets, with their formulas, and CAD environments, with their scripting capabilities, can be considered the earliest attempts to give users the possibility to tailor software artifacts to their own and specific needs (see [Nardi, 1993]); moreover, also recording macros in word processors and defining sets of filters in email clients have to be considered as EUD activities (see [Lieberman et al., 2006a]). However, it is only with the advent of the latest technologies, like *Web 2.0* and *visual programming*, that the EUD research field has grown and allows software designers to provide end-users with a greater freedom of customization of the software environments that they daily use to perform their work activities.

In particular, the aim of the presented research work is to give a contribution to the design and implementation of technologies supportive of cooperative work that allow users to autonomously tailor their document-based systems to their needs with the flexibility of paper-based documents. More specifically, tailoring activities must not only involve the flexible definition of the informative content of documents, even if this possibility does not have to be considered trivial; rather, users must be autonomous in performing those tailoring activities that allow them to customize their systems to support the local conventions of the group in which they work. Putting in the hands of users the ability to customize systems they use every day strongly contributes in making the transition from paper-based documents to digital documents less painful, with positive effects on work practices of users. Moreover, leveraging the possibility to autonomously customize systems to support local conventions, users can also be supported in the improvement of their work habits, through a collaborative learning process.

Thesis Outline

Chapter 1 will discuss the role that documents play in supporting the collaborative work practices within groups of employees and the problems that arise with the transition to a traditional document-based collaborative system. Subsequently, the chapter focuses on the need to leverage document-related conventions to improve collaboration and to promote awareness among employees. Moreover, the chapter will describe the reference domain of this thesis, i. e., the healthcare domain. Chapter 2 will deal with the discussion of the approaches that, during time, have been developed in order to allow system designers to design effective collaborative systems with the aim to provide support for the actual needs of employees. Subsequently, the chapter describes the main tenets of the EUD as the way to design tailorable collaborative systems that end-users can autonomously adapt to meet their constantly evolving needs. Chapter 3 will introduce

the *Web of Active Documents (WOAD)* framework, a framework that has been conceived on the basis of observed practices with the aim to enable the design of document-based collaborative systems that provide end-users with a support for their needs of awareness and flexibility. Chapter 4 will deal with the description of the relevant problems that affect the WOAD framework and that actually limited its applicability: i. e., the need of users to be autonomous in creating and modifying their document templates and the rules through which they augment documents in order to proactively convey conventional awareness information to support their collaborative work activities. Moreover, the chapter will discuss the solutions that this thesis proposes to make the WOAD framework compliant with the tenets of the EUD field. Chapter 5 will provide an overview of the state of the art in the areas of the visual composition of documents and rules respectively, and in the area of the visual languages, exploring research projects as well as commercial software. Chapter 6 will describe the two visual editor prototypes that have been developed to validate the solutions that have been proposed in Chapter 4. Chapter 7 will deal with the description of the implementation of the two prototypes described in Chapter 6. The remaining part of the thesis will focus on the validation of the proposed solutions, describing two user-studies that have been conducted in the healthcare domain. Chapter 8 will discuss a qualitative field study that has been conceived to validate the expressiveness of the visual language that have been proposed to address problem to make users autonomous in defining their own WOAD mechanisms. On the other hand, Chapter 9 will deal with the discussion of a quantitative study that was aimed at validating the usability of the visual editor prototype that has been developed to support users in autonomously use the proposed visual language. Finally, Chapter 10 will summarize the findings of the described work and outline the possible future directions to further improve the WOAD framework.

1 Background

Contents

Most working activities in the tertiary sector involve a certain number of different documents, and one of their main aims is to store the information that employees need to perform their work activities [Hertzum, 1999]. The amount of information usually grows proportionally to the complexity of the work activities and, consequently, employees need to create and exchange an increasing number of documents. The advent of computer-based systems, which has developed remarkably since the last decades, has led an ever increasing number of companies to take into account the transition from traditional paper-based documents to their digital counterparts for a better management of the exponential amount of information [Berry and Goulde, 1994] with a tangible reduction of costs.

During the last decades, paper-based documents overcame their role as a mere storage medium and became a fundamental part in the work practices that employees developed to perform their work (e. g., see [Malone, 1983]). In particular, companies usually organize employees in relatively small work groups in which they cooperate to reach a common goal: in these work settings, documents assume the key role of tools that help employees to coordinate themselves and to articulate their work.

Unfortunately, as shown by several studies (e. g., see [Sellen and Harper, 2003]), this transition to digital documents might be a highly problematic process, which can affect the set of consolidated work practices and, consequently, their efficiency and effectiveness.

This chapter will start with a simple example in order to better clarify the problems that can arise within a collaborative work setting when the organization forces employees to make the transition from paper-based documents to a computer-based collaboration system. On the basis of this example, the next section will focus on the role of conventions in supporting collaborative work and on the scarce attention that software developers pay to conventions during the design of collaborative systems. Subsequently, the chapter will discuss the *Electronic Patient Record (EPR)* as a paradigmatic case of paper-based digitalization in the healthcare setting which in this thesis has been adopted as the reference applicative domain. The last section will focus on the proactive provision of conventional awareness information as a way to support and improve collaboration within groups of employees.

1.1. A First, Simple Example: The News Agency

The authors of the study presented in [Braa and Sandahl, 2000] observed a real-life case of a news agency, in which employees cooperated to create the weekly television schedules that were subsequently published by the press (e. g., newspapers and magazines). Despite its simplicity, the described scenario is very useful for better understanding the role played by paper-based documents in coordinating employees and the issues involved by the transition to their digital counterparts.

Employees involved in this activity cooperated in drawing up the whole weekly television schedule, starting from the single weekly schedules that the various television channels sent every week to the news agency by fax (see Figure 1.1 for a quick reference). The employees of the news agency reorganized and retyped information that they found on each fax, often integrating it with additional material (e. g., the editorial information); after retyping information, employees stored it into a central system that subsequently they would use to create the whole weekly television schedule. During their activities, employees put faxes on three distinct shelves. The first shelf (i. e., the *in-shelf*) was used to temporarily store the new faxes as the news agency received them; at the same time, the *in-shelf* was used both to group and sort faxes according to the television channel that sent them and to the week to which they pertained. In this way, employees could quickly have an overview of the channels that were still missing or incomplete, and consequently they became aware of which channels were still outstanding. The second

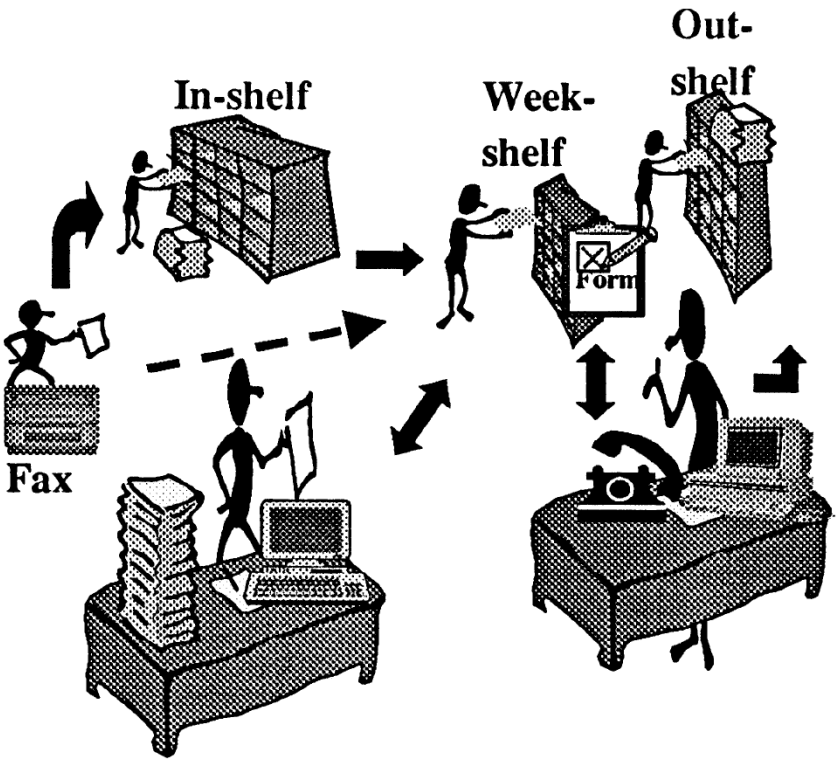


Figure 1.1.: The news agency scenario, before the transition to digital documents (from [Braa and Sandahl, 2000])

shelf (i. e., the *week-shelf*) was used when all television channels had sent their whole weekly schedules to the news agency. At that point, one of the news agency employees moved the whole set of faxes, which pertained to a specific week, from the *in-shelf* to the *week-shelf*. This had the consequence of making all the group aware that it was possible to start working on information that was written in these faxes (i. e., retyping them into the central system). Finally, the last shelf (i. e., the *out-shelf*) was used when all the week's information had been typed into the central system: faxes were moved again, from the *week-shelf* to the *out-shelf*, and this second shift indicated to employees that all the information was ready to be used in the final typesetting phase.

Even if faxes represented the most important type of document used by news agency employees, the example shows how another document, which was called *coordination form*, was adopted by employees with the explicit aim to support coordination inside their group (e. g., employees used this document to indicate either which days were missing in the schedules they received or that it was necessary to make some corrections).

When the news agency replaced faxes and the other paper-based documents that

employees used in their daily work with their digital counterparts, the need to use shelves to organize faxes disappeared, and this resulted in breakdowns in the work practices that employees had developed over time. The new system stored digital documents into common files, which were organized using a hierarchical structure, i. e., `/channel/week/day/file-name`. The purpose of system designers was to arrange files, and consequently digital documents, with the aim to allow future system improvements that would have to provide other applications the access to documents, in order to automatize some operations (e. g., making pages ready to be printed). Files were shared adopting commonly adopted techniques, i. e., email messages (to exchange documents with external subjects, like the television channels that provide the news agency with "raw" weekly schedules) and operating system shared folders (to exchange documents among news agency employees).
This approach immediately showed its limits. Using common files to store and organize digital documents had drastic consequences on employee collaborative work practices. Even if standard file systems allow users to enact a basic form of file arrangement, substituting shelves had the effect of disrupting the above depicted ability of employees to easily get a complete overview about the status of their work, simply by glancing at the shelves. This ability was particularly helpful for coordinating employees in their work activities. In fact, the news agency employees managed a huge amount of faxes (i. e., hundreds of faxes per week) and, when faxes had been replaced with digital documents, they were forced to continuously browse a similar amount of files. Despite document status could be gathered from file names, browsing files still made it more difficult to get a complete overview of what had been done, increasing the possibility of concurrent edit operations on the same documents.

This simple scenario makes it evident how documents (i. e., faxes) are not isolated entities, rather they are part of a strictly intertwined web of artifacts, people and places. Documents are not only instrumental to the goal that employees have to reach, rather they are integrated into their «social activities, and cannot be separated from the practice in which they are incorporated» [Braa and Sandahl, 2000]. In this light, it is easy to understand how faxes played an implicit, but very important role in the news agency work practices and in coordinating employees, making them aware of the context in which they were acting [Dourish and Bellotti, 1992]. The *coordination form*, which is an example of what in [Schmidt and Simone, 1996] is called "coordination mechanism", makes news agency employees aware of the status of their work activities in an *explicit* way (e. g., see [Rogers, 1993]). On the other hand, *shelves* provide employees with the same awareness, but in an implicit way (e. g., see [Heath and Luff, 1992; Hughes et al.,

1992]). In the latter case, employees can obtain the awareness information according to some conventional meanings, which employees collaboratively defined and bound to documents and to the related statuses.

1.2. Cooperative Work and Conventions

The group of employees described in the news agency example adopts some document-related *coordinative conventions* (e. g., the conventional meaning that is evoked in their minds by the position of faxes on the different shelves), which have a strong influence on their work practices. Before going forth, it is important to clearly define the concept of conventions (which includes coordinative conventions) in relation to collaborative work practices. In [Lewis, 1969], conventions are defined as a social phenomenon originated by the need to solve recurrent coordination problems. In [Mark et al., 1997], conventions are defined as «rules or arrangements established in a group, common and accessible to its members, that users need in order to cooperate effectively». Another definition of convention is reported in [Cabitza et al., 2009a] and is obtained by combining «the common-sense meaning of 'shared agreements and related practice that is either established or consolidated by usage' with the emphasis on the modalities by which practitioners articulate their activities in their mutual cooperative effort». All these definitions emphasize the role of conventions to support groups of employees in performing their collaborative work activities.

In fact, cooperating employees are fully-fledged communities and, in particular, they can be seen as *Communities of Practice*. A *Community of Practice* can be defined as a circumscribed but open community of cooperating employees that are characterized by common interests and a common ground of knowledge and experience, but their autonomous behaviors and objectives are limited by the organization in which they work (for a more detailed description, see [Wenger, 1998, 2006]). Members of a community cooperate and coordinate among each other through a well defined and domain-related set of artifacts, which can be either local to the community or imposed by the organization, and conventions contribute in intertwining these artifacts in a network of more or less explicit relationships that can be seen as a *web of coordinative artifacts* [Bardram and Bossen, 2005]. Document-based conventions are only a specific case of the more general concept of *artifact-mediated* conventions: taking again into account the case of the news agency, also shelves that are used to store and sort faxes can be considered as artifacts. Since employees autonomously and socially develop conventions to cope with the collaborative and coordinative needs of the community they belong to, conventions

are characterized by a highly *local* nature. Moreover, conventions are often strictly bound to the contingency, extemporaneous and constantly evolving needs of the community of employees within which they have been developed. For these reasons, conventions are more *informal* than the organization's policies that define and constraint the goals of the employees' work activities. Due to the informal nature of conventions and thanks to their extensive work experience, employees can continuously refine and internalize conventions, and in so doing they make a significant contribution in increasing the effectiveness of their collaborative work practices. In particular, as described in [Mark, 2002], conventions are the foundation of a continuous process of learning that concerns both the tasks that employees must perform and the social interactions within their group.

Unfortunately, as shown in [Braa and Sandahl, 2000], most document-based collaborative systems provide employees with poor support (if any) for the conventions that they developed during their collaborative activities. This lack is mainly imputable to the very little attention that IT professionals (i. e., software analysts and designers) dedicate to work practices during the collection and the analysis of requirements (in particular, functional requirements): they focus their design activities only on the storage function of documents (e. g., the content and how it is structured) and on how documents have to be presented to users. Thus, the transition to a computer-based system in a collaborative work environment could have the side effect of compromising the convention-based work practices [Ash et al., 2004] which characterizes the members of the host community. Some paper-based conventions could "survive" in traditional document-based systems thanks to the unconventional use of some parts of these systems, which in [Cabitza et al., 2009a] are called "gray zones", e. g., when employees use a long-text field in a form to annotate some acronym that is well-known and often used inside their group but is not defined at the organization level; however, this is only a workaround enacted by employees and does not represent a true form of support for conventions. In this light, in order to avoid the occurrence of the depicted work practice breakdowns, the main issue in the design of digital document systems that effectively support cooperative work settings, is to spend the right amount of time, during the requirement collection and the design phases, to identify and provide support to those conventions that «keep the practice together» [Braa and Sandahl, 2000], keeping in mind that, due to their local, informal and often extemporaneous nature, conventions can be different from one community to another. Chapter 2 provides a description of the problems related to the design of collaborative systems that provide employees with an effective support to their conventions, analyzes the different approaches that have been proposed to reach this goal, and finally focuses on the approach that has been adopted in the work described in

this thesis. The next section focuses on a scenario in the healthcare domain and on how awareness promotion plays a role in consolidating work practices.

1.3. A More Complex Scenario: The Healthcare Domain

The news agency example is very useful to give a first, but quite exhaustive overview about the role played by documents in coordinating employees, and demonstrates how documents must become a central point of interest [Lortal et al., 2005] in companies and, more generally, in organizations, which act in a heterogeneous set of domains.

The healthcare domain represents a very complex work setting in which doctors and nurses need careful coordination of their activities [Reiser, 1984; Strauss et al., 1985; Atkinson, 1995; Timmermans and Berg, 2003]. In this work setting, a very heterogeneous set of professional stakeholders [Morrison et al., 2011] needs to cooperate to provide patients with increasingly better medical care. During their work, doctors and nurses widely use well defined sets of paper-based documents with the twofold aim of better helping their patients through the process of healing (e. g., prescribing specific medical treatment and following the evolution of the illness) and managing the flow of information with the hospital management. Usually, these sets of documents, which often are called *patient records* [Dick et al., 1997], encompass a heterogeneous number of documental artifacts that differ both in structure and function, but that are highly interconnected and cross-referenced [Cabitza et al., 2005]. For these reasons, computer-based health systems should be designed to support collaboration within this work environment. Frequently, these systems do not provide users with the results they expect (in terms of supporting their local work practices) [Kuziemsky and Varpio, 2011]. This happens not for purely technical reasons, rather because those systems have not been designed paying the proper consideration to the complex web of relationships that characterize the healthcare work settings. In fact, the healthcare setting, for its criticality, has been chosen as the reference domain of this thesis.

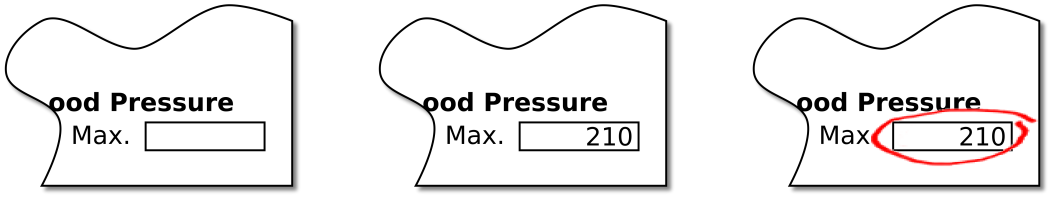
Clinical work practices involve a rich set of conventional annotations (e. g., acronyms and signs) that doctors and nurses inscribe on paper-based documents in order to enrich them with additional, contextual data with the aim to better support the coordination of the whole group of colleagues [Bringay et al., 2006]. For instance, a doctor can detect an anomalous value in the maximum blood pressure of a patient and, accordingly, she can draw a red circle around this value (see Figure 1.2), in order to make all her colleagues (i. e., both other ward doctors and nurses) aware that the patient is in a potentially critical situation; in this way, when another doctor or a nurse reads the document, the presence

of the red circle immediately makes her aware of the potential criticality, and she can enact the most suitable clinical practices to cope with that situation. The use of these conventional annotations can be either local for a single hospital ward or, otherwise, shared within different wards, but often with different conventional meanings. For instance, in both the *Internal Medicine* and the *Neonatal Intensive Care Unit* that were investigated in [Cabitza et al., 2009a], an exclamation mark drawn close to the request for a blood culture examination means "urgent". While in the *Internal Medicine* ward "urgent" means "within a day", in the *Neonatal Intensive Care Unit* "urgent" is interpreted as "within ten minutes". Local conventional annotations contribute to enhancing clinicians' efficiency and effectiveness, providing a better quality of care and the appropriate services to the patients. On the other hand, this shows how documents with their data and annotations play the role of "boundary object" [Star, 1988] when they cross different organizational units.

In fact, there is constant need for the hospital management to reach a higher level of efficiency and effectiveness in healthcare activities, with the consequent reduction of operating costs. This need leads the hospital management to adopt computer-based solution, i. e., the *Electronic Patient Record (EPR)*, in order to replace traditional paper-based records (e. g., see [Chen and Akay, 2011]). Moreover, this transition to digitalized patient records is strongly promoted by public institutions, which have the same need of efficiency and effectiveness, in order to improve the healthcare services for the citizens, reducing at the same time the costs of the public health: e. g., in 2009, U.S. President, Barack Obama, «pledged that by 2014, all American health records would exist in an electronic format» [Jacques, 2011]. Thus, public health institutions are increasingly adopting complex ICT solutions, in order to build big "data banks"[1] with the aim to monitor and coordinate different healthcare service providers (e. g., both public and private hospitals). In order to promote interoperability, public health institutions impose the adoption of standards (e. g., *HL7* or *ISO 13606*) to define the digital representation of clinical documents. As standards focus on giving documents a well-defined structure, they do not however provide any support to the local and informal conventions of doctors and nurses.

Consequently, traditional EPRs usually follow an "imposed" structure enforced by the adoption of standardized documents that have been conceived by ICT professionals at design time to provide hospitals with an unified, global solution. As in other domains, this

[1] For instance, in Italy, and in particular the Regione Lombardia, institutions promoted and funded the building of the DENALI data warehouse. See `http://www.pfizer.it/cont/pop_camuni.asp`. Accessed: 2012-06-20. (Archived by WebCite® at `http://www.webcitation.org/68Z4BHcul`).



(a) The "blood pressure" field in the paper-based "Vital Signs" document

(b) Some clinician filled in the "Vital Signs" document with the patient's blood pressure value

(c) Some other clinician highlights the critical situation drawing a red circle around the "blood pressure" field

Figure 1.2.: A simple example of how clinicians can conventionally highlight a critical situation on paper-based documents (the patient's blood pressure example)

could have negative effects on collaborative clinical activities. Indeed, the heterogeneity of the needs, work practices and conventions of each hospital ward, would require greater flexibility in digital document definition. For instance, taking again into account the *Internal Medicine* and the *Neonatal Intensive Care Unit*, these two wards have radically different purposes, and they adopt different working practices and conventions, despite they belong to the same hospital. As a consequence, the *Neonatal Intensive Care Unit* can adopt some peculiar (paper-based) documents that are meaningful only for its clinicians, and that do not exist in the *Internal Medicine* ward. In addition, these two wards can share some documents that, for the kind of patients they manage, need to be structured adopting different arrangements, in order to give more or less visibility to different pieces of relevant information or to incorporate even different kinds of contents.

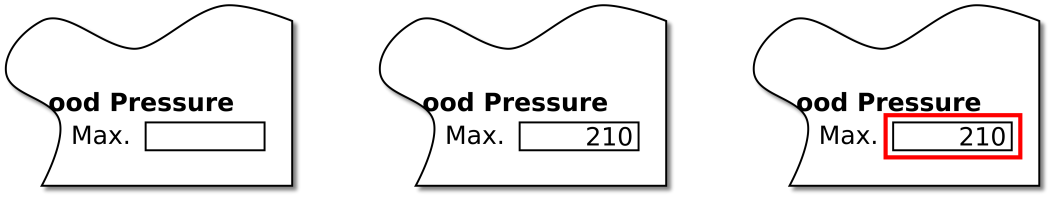
When the hospital management imposes the transition to a "paper-less" system, clinicians of a ward are forced to use a well defined set of rigid digital interfaces, which allow them to interact with the system. This rigidity is reflected in two distinct facts: (i) the absence of any possibility to customize the contained information (both in its structure and in its appearance), and (ii) users are forced to follow fixed fill-in paths, which have been conceived at design-time. The system rigidity leads to a breakdown of all those conventions that doctors and nurses usually follow in their daily working practices. For instance, digital documents do not allow clinicians to inscribe any kind of annotation (instead of what is shown in Figure 1.2 for paper-based documents), and this results in the invalidation of all those conventions that are based on this kind of contextual information. This is a major drawback, as this kind of additional information contributes to promote awareness in clinicians, enhancing the effectiveness of their activities.

1.4. Promoting Employees' Awareness

The above depicted scenarios show how any form of computer-based support for cooperative work settings, and the related practices, should necessarily take into account the existence of conventions in order to be really effective. More generally, these systems must support employees in their daily activities in a way that is as much aligned as possible with how they really work and also prevent the imposition of unnecessary and undue constraints on their work practices. Thus, as extensively discussed in [Mark, 2002; Cabitza et al., 2009a], conventions can be leveraged to improve coordination and to promote *collaboration awareness* [Lauwers and Lantz, 1990] in a *proactive* way, making employees aware of contextual information, but only when this information can be considered relevant with respect to specific conventions or situations.

The idea to proactively provide contextual, convention-based information to support employees who act in cooperative settings was emphasized in [Mark, 2002], which conceptualized awareness as an 'active learning device'. Mark described conventions as a framework for awareness, since conventions allow to understand and anticipate the behaviors of the various groups of employees, in addition to their ability to involve employees in a continuous learning process (see Section 1.2). In order to design a system that supports the proactive provision of contextual information in an effective way, software analysts and designers have to consider an important aspect that can be summarized with the question "what is the right amount of information to be provided?". This problem is related to the concept of information overload [Mark, 2002; Ash et al., 2004]. In fact, providing too much information can lead to the opposite result with respect to the aimed goal of supporting employees: providing employees with a huge amount of information can lead them to consider contextual information as a nuisance, with the risk of neglecting it totally.

In order to avoid this risk, during the design phase of the system, it is necessary to identify the best way to provide different types of awareness information that will cohabit over the same artifact (e. g., a document). Due to the informal nature of the conventions, it is necessary to find the right level of "visibility" and formality of contextual information that the system will convey to employees. On the other hand, the local nature of conventions requires the identification of the best way to provide users with this awareness information, ensuring that the identified methods can be easily tailored to the needs of each community, with respect to the specificities of different domains. In this light, during the design phase it can be useful to detect common features and recurrent provision patterns. This can help to extract the necessary requirements to



(a) The "blood pressure" field in the "Vital Signs" document

(b) Some clinician filled in the "Vital Signs" digital document with the patient's blood pressure value

(c) The filled in value triggers the computer-based system that mimics the draw of a red circle around the "blood pressure" field

Figure 1.3.: An example of "graphical clue" to proactively convey awareness information over a digital document, in a way that mimics what employees usually do with paper-based documents (see the patient's blood pressure example in Figure 1.2 for a comparison)

conceive an effective supportive technology and mechanisms of information provision that are able to evoke conventions without constraining or imposing them within employee daily practices.

Thus, taking into account the role of conventions and the related contextual information in supporting employee collaboration, a computer-based system conceived to support cooperative work settings could proactively provide awareness information through contextual indications that can be conveyed over the artifacts as "graphical clues" [Cabitza et al., 2009a], in a way that reflects the conventional annotations that employees used to inscribe on paper-based documents (e. g., as shown in Figure 1.3, the indication in the patient record of an anomalous situation by the highlighting of some data with a red border). Moreover, it is essential to provide users with graphical indications in a seamless and unobtrusive way, leaving to them the task of discerning if the conventions that have been evoked are relevant or not, and if they need to behave accordingly.

1.4.1. Awareness Promoting Information

The goal to reach in the design of document-based systems that proactively promote *collaboration awareness* among actors is to use documents as a way to convey additional contextualized information (see Figure 1.4). To this aim, the notion of *Awareness Promoting Information (API)* [Cabitza et al., 2009a] can be usefully applied.

During their observational studies in the healthcare domain, the authors identified nine main types of APIs (the whole list of the identified APIs, with the description of the related meaning, is reported in Table 1.1). Even if these nine APIs can cover most of the

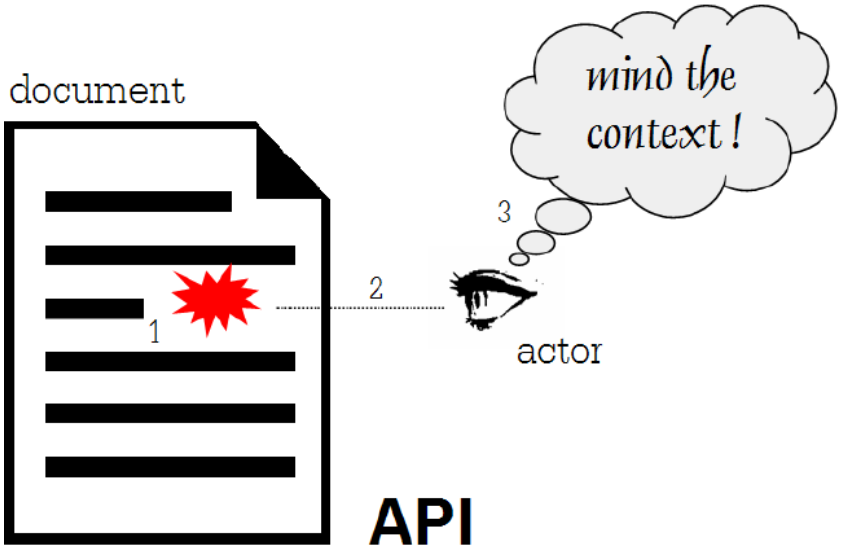


Figure 1.4.: The concept of *Awareness Promoting Information (API)* (extracted from [Cabitza and Simone, 2009a])

awareness information that clinicians consider crucial to support their cooperative work practices, this set of APIs does not constitute an exhaustive solution that cover the needs of all kinds of work settings. In fact, due to the *local* nature of conventions (see Section 1.2), the domain in which work activities take place has a strong influence on the work practices that employees develop to better cooperate, and consequently they can need different types of awareness information to effectively support their collaborative work. Thus, to give better support to collaborative work in other applicative domains, in the future, the set of APIs could be enlarged and refined to include new types of awareness information.

APIs can be grouped into three different categories, which represent the different aspects of work that they aim to support. The first category groups those APIs with the aim to support the *articulation of work activities* and the related cooperative work. At present, only the *Appropriateness API* belongs to this category, which conveys information about the activities that should be accomplished taking into account the current context[2]. The second category is related to the *interpretation of the context*. The *Inquiry API* conveys information that makes the employees aware about the existence of additional

[2] The context in which an activity can be considered appropriate or not can pertain to both the content of the artifact and other contextual conditions.

Type of API	Description
Appropriateness	Any information that provides indications on what could be appropriate to do or not to do regarding the artifact's data
Criticality	Any information that indicates the need to consider the situation reported in the artifact as critical
Deviation	Any information that indicates the need to consider some data of the artifact's content as either unusual or produced by a deviation with respect to some expectation
Inconsistency	Any information that indicates the need to check whether some data of the artifact is inconsistent with some other data or the current context
Inquiry	Any information that indicates the opportunity to get further information about the data reported in the artifact
Provisionality	Any information that indicates the need to consider some data of the artifact's content still provisional or unofficial
Quality & Safety	Any information that indicates the need to consider precise quality/safety requirements and expectations in managing the data
Revision	Any information that indicates the need to correct some data of the artifact

Table 1.1.: A taxonomy of API and their conventional meanings (extracted from Cabitza and Simone [2012])

data that could be helpful to better understand the data that is inscribed on the documental artifacts. Quite obviously, the *Revision API* and the *Inconsistency API* respectively provide to employees information about possible errors and inconsistencies in the document content. The *Provisionality API* makes employees aware that a document content is to be considered as provisional or unofficial. Finally, the *Criticality API* conveys indications about a situation that employees consider critical (e. g., the case of low blood pressure in a non-anemic patient).
Finally, under the third category are grouped those APIs that convey information about the compliance with some expectations about either the normal work practice or the quality. The *Deviation API* provides information to make the employees aware about some deviations with respect to an expected outcome (e. g., a drug prescription with drug dosages that are significantly different from those that normally have to be prescribed). Similarly, the *Quality API* provides indications about a possible content "deviation" with respect to some precise quality requirements and targets. Finally, the *Safety API* conveys to employees some indications to inform them that either the activity they are performing or that they are about to perform can lead to the occurrence of an adverse event (e. g., some adverse drug reactions, due to a patient's allergy).

As already mentioned, awareness information should be unobtrusively conveyed to the actors. This can be done through graphical cues, adopting four different provision methods:

Message displaying. Messages can display their textual content either as notes, remarks or comments. The message content can be either a default message, which is related to a particular condition, or a comment that has been annoted by an employee. Moreover, each message can be shown either in the case of particular interaction events (e. g., the insertion of particular data) or in the case of contextual events (e. g., a specific date or time).

Highlighting data values. A specific data value can be highlighted in order to make it more evident, or just different, with respect to the others in the same portion of a document (e. g., in a clinical examination report, values that are out of the standard ranges could be displayed using the red color).

Highlighting data structures. This method is similar to the previous one, but it differs in the target of the highlight action. In this case, a specific data structure (e. g., fields, sections or the whole document) can be highlighted to convey the contextual information that is related to the document structure rather than to its content.

Displaying icons, pictures or other graphical items. In this case, either data values or data structures can be coupled with graphical indications (e. g., marks, signs, lines, icons and pictures) that are conventionally associated to a well defined set of meanings (e. g., alerts, reminders, warnings or helps).

These provision methods will be mentioned in the document-based framework described in Chapter 3. Before that, the next chapter will consider issues related to the digitization process.

2

End-User Development

Contents

Since the first decade of the CSCW research field, it became immediately clear that, to be really effective, CSCW systems «should be sensitive to the work context in which they will be used [Bentley and Dourish, 1995]» and, consequently, they «should reflect the understanding of work practices [Agostini et al., 1997]», in particular obtaining «a better understanding of the way in which work is carried out in groups [Bentley and Dourish, 1995]». Moreover, as widely recognized in the CSCW literature, employees «are the ultimate custodians of and experts in their own practices [Shapiro, 1994]». However, in traditional software engineering techniques too little «attention has been paid to modes of use [De Michelis, 2003]», focussing system design on the features to provide to users (i. e., what in [Bentley and Dourish, 1995] goes under the concept of "*system-as-mechanism*").

In fact, a sharp distinction between designer and users exists in traditional software design techniques. Consequently, during the elicitation of system requirements, designers focus on the functional needs of clients (i. e., organizations) and completely neglect the

existence of users' needs (e. g., the social organization of existing work activities, their constant evolution and their flexibility with respect to the context). Traditional techniques are «developer-centric [Harker et al., 1993]», i. e., they represent system requirements in a format that is suitable to be used by software engineers and developers to design and develop the system, rather than to represent the real users' needs. As a direct consequence, adopting traditional design techniques results in systems that do not meet the real needs of users, with negative effects on the efficiency of their work practices. Moreover, users' requirements constantly change and evolve. Systems designed in this manner are prone to become outdated or insufficient fairly quickly. On the other hand, conventional design and development cycles would be too slow and time consuming with respect to dynamic changes in system requirements (e. g., an extemporaneous need to provide support for a specific and unforeseen situation).

Thus, traditional design techniques are to be considered unuseful for developing systems that effectively support collaborative work. This chapter will describe the various approaches that have been developed over time in order to improve this situation. First of all, a set of approaches that try to extend and refine traditional software engineering techniques will be described. The next sections will deal with the approaches that encompass end-users as systems' co-designers: from *participatory design* up to *End-User Development (EUD)*. Finally, the end of the chapter will focus on *meta-design* as an approach to design effective EUD environments to support work practices.

2.1. Models of Work to Improve the Design of Collaborative Systems

A first step towards the design of more effective collaborative systems is represented by the attempts to address this need through the definition of increasingly refined analysis techniques, with the aim to define models of cooperative work (e. g., see the *model of awareness* presented in [Rodden, 1996] or the *DIPA*[1] *model* [Lewkowicz and Zacklad, 2000]) that allow software engineers to design collaborative systems, which give systems end-users the ability to interact with them in a way that is closer to their actual work practices. Even if models of collaborative work can be helpful to allow software engineers to better elicit system requirements, similarly to traditional software engineering techniques, this approach shows its limits soon, especially when users ask the system to provide support in a way that software engineers did not envision during

[1] DIPA is the acronym of the French words *Données*, *Interprétations*, *Propositions*, *Accord*, which mean Data, Interpretations, Propositions, Agreement.

the design phase. As discussed in [Sommerville et al., 1994], the limits of this approach can be mainly imputed to the gap that still exists between the information needed by software engineers and those obtained with analysis methods and models that are aimed at representing the actual work context.

A lot of research efforts have been made to try to fill this gap. The interdisciplinary nature of the CSCW research area (e. g., see [Schmidt and Bannon, 1992]) suggested opening the design process to analysis techniques coming from other disciplines and research fields (e. g., from social sciences). For instance, the social nature of collaborative work (see Chapter 1) gave a significant boost to the adoption of ethnographic techniques to try to adapt analysis methods, with the aim to better elicit system requirements and to uncover the 'real world' character of work [Hughes et al., 1994]. The adoption of ethnographic techniques in the design process of systems was not free from criticisms: Anderson, for instance, argued that the interest of designers in ethnography is motivated by a fundamental misunderstanding of its role in the process of requirements elicitation, since ethnography has often been used as a method of data collection rather than focusing on its analytic aspects [Anderson, 1994]. Despite any criticisms, ethnography gained an ever increasing consensus in the design of collaborative systems (e. g., see [Greenhalgh and Swinglehurst, 2011]). In fact, it introduces a social dimension within software engineering techniques and models. More specifically, ethnographers, through qualitative techniques (i. e., surveys, field studies, observations and interviews), collect useful information and requirements that, once analyzed, give software engineers the ability to better understand the social relations and the organization of work settings in which the designed system will be used. Consequently, the expected result is a system designed to more suitably support the actual work practices.

Despite these attempts to introduce increasingly better analysis techniques and models, the previously mentioned design techniques (from traditional approaches to the more multidisciplinary ones) still are limited and consider systems in terms of "mechanisms" that provide functions to support users in performing their tasks. Moreover, all of them still do not consider users as a fundamental and active part in the system design process; at most, with the use of ethnographic techniques users become part of the design process of the system, but actually they remain a totally passive part of this process, since they are only an 'object' for the techniques of analysis and requirement elicitation.

A diametrically opposite approach to system design, which aims to overcome these limits, is focused on the idea that, due to its flexible nature, work «is better supported when systems provide a 'medium' which can be tailored to suit each participant's needs and organised around the detail of their work [Bentley and Dourish, 1995]». This shift of

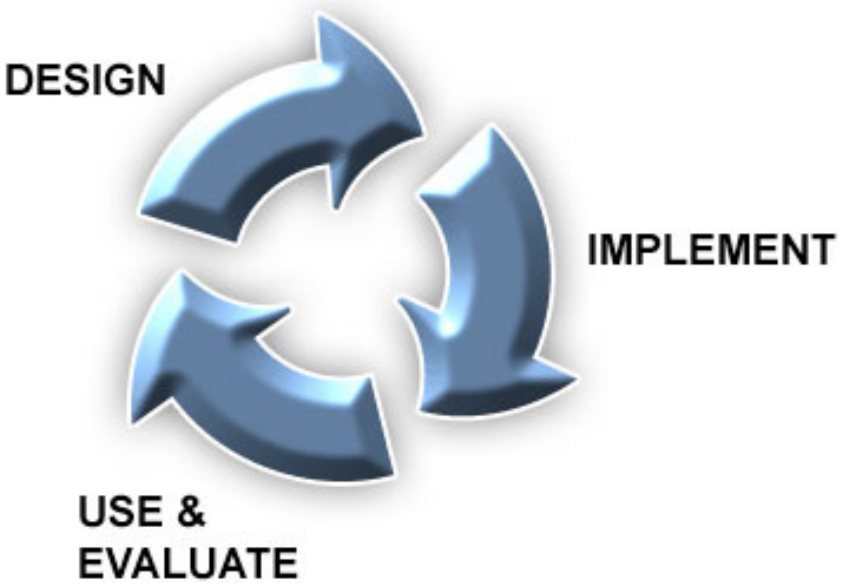
the point of view on the role played by the system consequently highlights the need for a greater involvement of users (who, it is remembered, are the end-users of the system) within the design process.

2.2. Involving Users in Design Activities: Participatory Design

The idea to give users (e. g., employees of organizations) a more prominent place has been considered and acknowledged to design many types of systems. As reported in [Telier, 2011, Chapter 8], examples of this kind of approach range from (i) *user-centered design* (e. g., see [Norman and Draper, 1986; Vredenburg et al., 2002]), in which the focus is on system use and usability, (ii) *contextual design* (e. g., see [Beyer and Holtzblatt, 1998; Reffat and Gero, 2000]), which is focused on the situatedness of use, to (iii) *experience design* (e. g., see [Sanders and Dandavate, 1999; Sanders, 2001]), in which designers focus on the quality of the user experience. Notwithstanding these approaches explicitly pose the emphasis on users, they are still relegated to a passive role in the design process, such as what happens in the already described approaches to system design.

In order to adopt the metaphor of "system-as-medium" (see [Bentley and Dourish, 1995]), users need to be promoted to the role of co-designers. This is the main tenet of the design approach that is called *participatory design* (for an overview, see [Greenbaum and Kyng, 1991; Schuler and Namioka, 1993; Kensing and Blomberg, 1998; Bødker et al., 2004]). The adoption of this approach to system design means that software engineers recognize the need of users «to express themselves and to participate directly and proactively in the design development process [Sanders, 2002]». In this light, in *participatory design* users, and more generally all potential system stakeholders (including, for instance, managers of organizations), are actively involved in each phase of the design process of the system with the aim «to meet the unattainable design challenge of fully anticipating or envisioning use before actual use takes place [Telier, 2011, Chapter 8]»: it is for this reason that *participatory design* can be summarized as "design for use before use" [Redström, 2008]. To reach this ambitious goal, *participatory design* techniques require to identify and consider relevant human factors in the system design phases. As reported in [Sharma et al., 2008], the typical methods that system designers use to elicit and evaluate ideas include mock-ups and simulations [Clemensen et al., 2007], as well as the examination of similar products, scenarios, sketches, models and prototypes (see Figure 2.1).

Notwithstanding the fact that *participatory design* is a significant step forward to allow IT professionals to design systems that rely on a better understanding of actual



From `http://www.personal.psu.edu/cwc5/blogs/coursedesign/lesson-04-interface-design.html`. *Accessed: 2012-08-09. (Archived by WebCite® at* `http://www.webcitation.org/69nmAT7RG`*).*

Figure 2.1.: The *participatory design* cycle

collaborative work practices, it is an approach that is still too close to traditional software engineering design practices: in fact, even if users actively contribute to the design of their system, after the completion of the design phases, they return to a passive role, i. e., the role of users of the system, with scarce chances to modify the system in case of unexpected and unforeseen situations (which can still exist, even though in smaller amounts due to the contribution of users during the design phases). Consequently, even if participatory design contributes in narrowing the misalignment between the needs of software engineers and work practice models and analysis techniques, this gap continues to exist and it is still far from being filled.

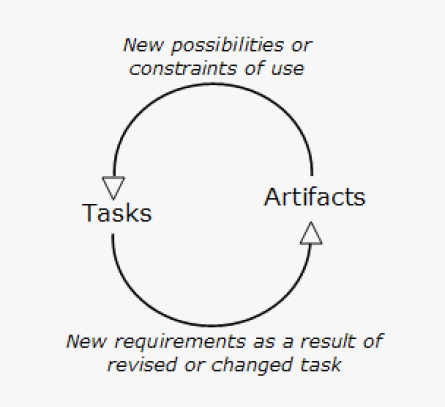
2.3. When Software Engineering is not Enough: End-User Development

Providing software engineers, and more generally IT professionals, with a set of knowledge, languages and tools, which allows them to design and build collaborative systems, might not be sufficient to fulfil the expected results. Moreover, even if participatory design techniques allow the active participation of users in the design process, this approach

still presents some limits, since it does not allow a timely management of unexpected (and often extemporaneous) changes in system requirements. In fact, during design and development phases of (complex) software systems, and more generally any kind of software artifact, designers and developers can anticipate only a part of the whole future uses and problems sets that those systems can support and address respectively. Thus, it is common that, at use time, users have the possibility to discover some mismatches between their real needs and the support that the system they are using can provide to them (see Figure 2.3). Following traditional software engineering approaches, existing software systems must be (partially or fully) re-designed, and consequently (partially or fully) re-developed, in order to adapt them to cope with these new needs and requirements. As a direct implication, this requires users to wait a certain amount of time between the detection of either new needs or unexpected problems and a newer release of their system that addresses them. But, in most common situations, new requirements and problems are related to local and extemporaneous needs of restricted groups of end-users, and this leads to the frequent case in which end-users autonomously address these needs in some unconventional ways[2] (i. e., performing some workarounds) before IT professionals delivers a new system releases. However, users are the only stakeholders that could act with a certain timeliness to adapt the system making it compliant to these new and unforeseen requirements.

Thus, as proposed by Dourish, collaborative systems should be conceived and designed to provide users with a *tailorable* "medium". In this way, users can constantly and autonomously adapt to the ever changing needs both in the applicative domain itself and in the execution context (see [Costabile et al., 2003a]). The direct involvement of users in system customization activities reinforces their role of active players in the design of collaborative systems that effectively support them in doing their *tasks*. Adopting this approach results in systems that can be continuously adapted and refined (coherently with the "task-artifact cycle" [Carroll et al., 1991] shown in Figure 2.2), with positive influences on work effectiveness and on the degree of confidence with which users interact with them.

[2] In this case, the 'unconventional ways' meaning indicates all those behaviors and uses autonomously performed by users to cope with their needs that have not been "foreseen" during system design, e. g., exploiting some system design weaknesses that generate what can be called information system "gray zones" [Cabitza et al., 2009a].



From `http://www.interaction-design.org/encyclopedia/task_artifact_cycle.html`. *Accessed: 2012-08-11. (Archived by WebCite® at* `http://www.webcitation.org/69qlRIkzy`*).*

Figure 2.2.: The task-artifact cycle

2.3.1. End-User Development

The ever increasing availability of new technologies and applications that are specifically designed to be easily understood, used and customized by end-users contributes to increase the number of users who start to approach the systems they use as active actors (e. g., [Boehm et al., 2000; Scaffidi et al., 2005] show how this phenomenon is known since the past century and researcher estimated that it would have increasingly gained a relevant position in various types of software systems). The *End-User Development (EUD)*[3] approach was born to meet these users' needs. With respect to participatory design (see Section2.2), in which users play an active role only in the design phases of their systems, in EUD the «tasks that are traditionally performed by professional software developers at design time are transferred to end users at use time [Costabile et al., 2008a]», and consequently the sharp distinction between design time and use time, which characterizes all the approaches described until now, becomes vague [Pipek and Wulf, 2009].

Within the EUD scope, end-users can be defined as «people who have certain software development skills but are not interested in software per se. They do not develop software for other people; rather they are developing software to solve specific problems that they own [Ardito et al., 2009]». As discussed in [Costabile et al., 2003b], end-users are interested in EUD mainly to reach goals that range from *changing some user interface behaviours and aspects* to *fully customize* the user interface of the system they use to meet

[3] See [Lieberman et al., 2006b; Pipek et al., 2009a; Costabile et al., 2011] for a complete overview about *End-User Development (EUD)* and the evolution of the related research field.

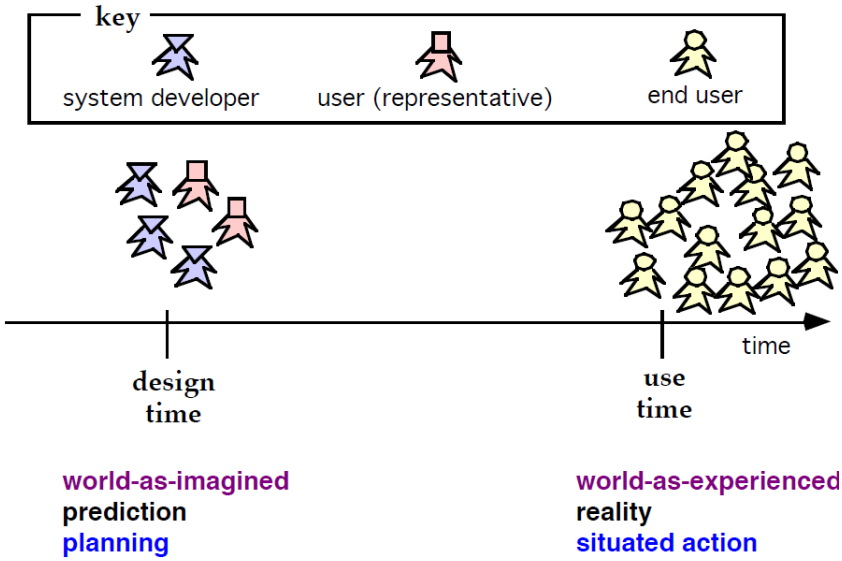


Figure 2.3.: Design and Use time (from [Fischer and Scharff, 2000])

their needs, from *customizing* existing functionalities to *adding new ones* to the system (e. g., to implement some aspects of a business process or to automate some tasks that rely on the user's domain expertise). Thus, an EUD-compliant software system must be designed to be easily tailorable at runtime by end-users, who can be able to autonomously adapt the system to their work situation, preferences and habits (see [Costabile et al., 2008a]). In this light, EUD can be defined «as a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software artefact [Lieberman et al., 2006b]».

To reach this ambitious goal, i. e., «empowering end-users to develop and adapt systems themselves [Lieberman et al., 2006a]», EUD must also take into account the tenets of other disciplines and research fields [Klann et al., 2006]. In order to be really effective, an EUD-compliant system must take into account the tenets coming from the *Human-Computer Interaction* research field, which can play a key role in making the designed system *easy* to be *understood*, to be *learned* and to be *used*. EUD-compliant systems should *adapt* to the changes that occur in contexts and in users' needs (e. g., according to users' *skills* or to the tasks they must accomplish), and EUD functionalities should be presented to users as *unobtrusively* as possible and only when they are needed. Moreover, EUD-compliant systems must take into account the social and collaborative dimensions that are related to end-users' customization activities (coherently to CSCW tenets). Providing support to EUD techniques requires software engineers to design systems in which end-users play an active role in their evolution, giving life to a long-term collaboration among system

stakeholders (e. g., system end-users and managers) and software engineers [Mørch and Mehandjiev, 2000]. As described in [Klann et al., 2006], «cooperation is an essential part of end-user development». As a matter of fact, end-users collaborate and group into communities that allow them to discuss their adaptation problems and needs, negotiate optimal solutions, and share both the expertise they acquire and the new artifacts they create with their EUD activities [Fischer, 2009], e. g., influencing the ways through which end-users concur in creating organizational expertise and the related knowledge [De Paula, 2004]. In this light, EUD can be seen as «a socio-cultural activity, depending on place, time, and people involved [Lieberman et al., 2006a]», i.e., in other words, EUD is a participative process [Fischer, 2009] that is strongly influenced by the *context* in which it takes place. This results in software systems that give end-users the possibility «to improve their work practice and determine an increase in their productivity and performance [Costabile et al., 2008a]».

The operations through which end-users can customize their EUD-compliant systems cover a wide range of possibilities (see Figure 2.4). The simplest operations belong to the parameterization category: in this case, end-users have very limited possibilities to customize their systems, since they can only change the behaviors of the system functions by specifying the values of the related parameters (e. g., the possibility of doctors to select the desired data field within their EPR to generate customized medical reports, as described in [Morrison and Blackwell, 2009, Section 3.1]). The diametrically opposite approach, i. e., *tailoring*, allows end-users to customize their systems with deeper interventions, which often require them to write some kind of executable code (e. g., word processor macros) that can have different levels of complexity. Due to the absence of a background in programming techniques, which contributes to characterize end-users as *unwitting developers* [Costabile et al., 2008b] (see Figure 2.5), system tailorability can result in a hard task for end-users. Providing users with modular software environments, through the adoption of component based approaches (e. g., see [Wulf et al., 2008]), can be helpful to ease systems' tailorability. With respect to tailoring systems through the creation of executable code from scratch, component-based approaches allow to create highly customizable systems that do not require end-users undue efforts to learn complex programming techniques. Rather, end-users need only to know how to compose (and eventually parameterize and extend) existing components, as though they were doing "bricolage" activities (see [Hovorka and Germonprez, 2009; Teoh et al., 2012]), without any concern about details of component implementation. However, end-users should at least have or acquire basic programming skills in order to connect the components they need. This significantly reduces the efforts required to users to customize their

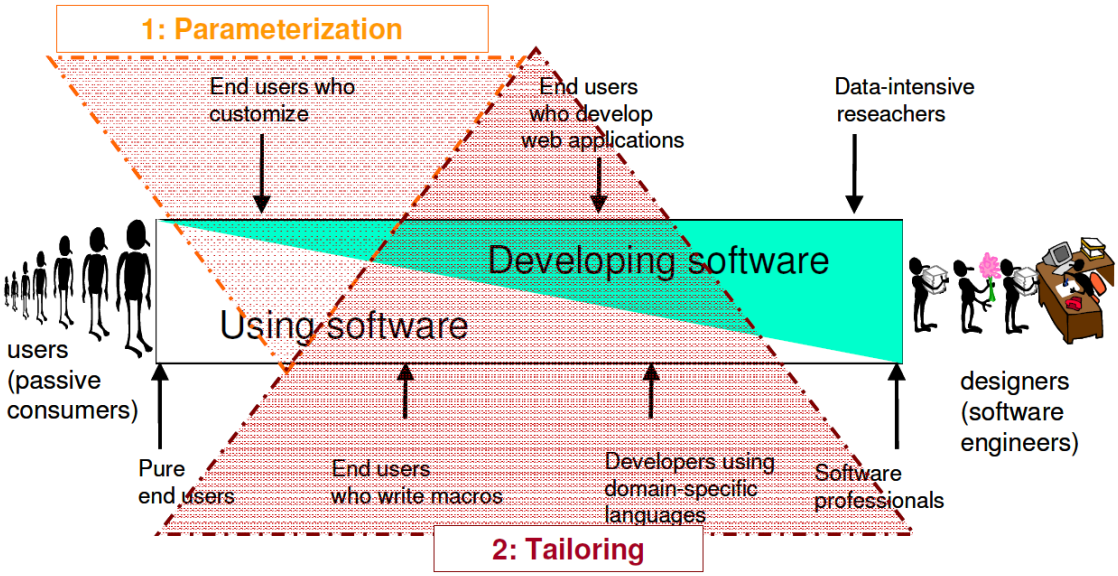


Figure 2.4.: The different approaches to system customization (from [Ardito et al., 2009])

systems, as component-based approaches constitute a middle layer between the pure parameterization of systems and programming (see [Mørch et al., 2004]).

2.3.2. End-User Development to Leverage System Appropriation

Even if, in some cases, supporting EUD could expose software systems to some security issues (e. g., see [Harrison, 2004]), giving end-users the ability to tailor systems they use to meet their local needs could result in an increasing acceptance degree of the system in their daily work practices. In fact, end-users who perform EUD activities constantly gain an increasing level of confidence with the software systems they use in their daily work; this has the positive effect of making «users feel confident or not afraid of technology [Cabitza and Loregian, 2008]». This results in a virtuous circle in which end-users increase their level of appropriation of the system (in which they can tailor to meet their real needs) and, as a consequence, this produces positive repercussions on the efficiency of the work practices that the system is aimed to support.

As described in [Dourish, 2003], appropriation is a process through which users adopt software systems, performing adaptations, to fit them into their already consolidated work practices (which can eventually evolve as a consequence of the introduction of such systems within them [Stevens et al., 2009]). To be more precise, the term 'appropriation' represents the «collaborative effort of end users, who perform "appropriation activities" to make sense of the software in their work context [Pipek, 2005]». Being appropriation

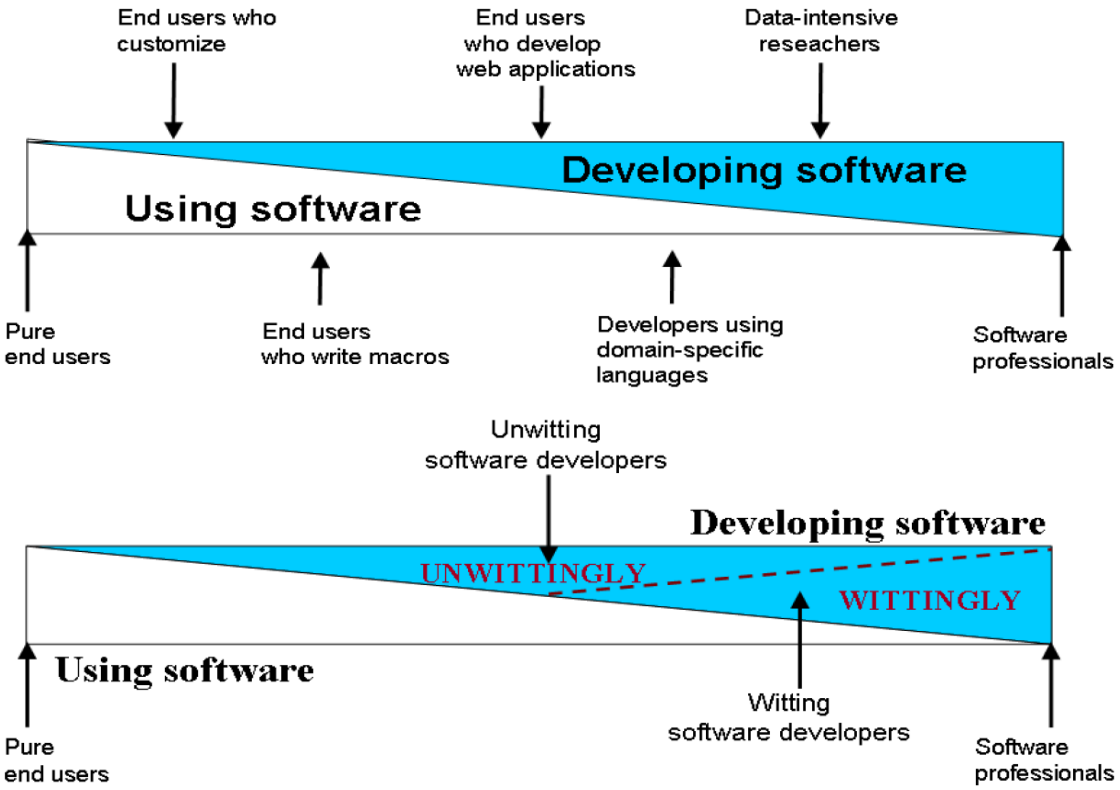


Figure 2.5.: The wide range of systems' end-users (from [Costabile et al., 2008b])

strictly related to the tailoring activities of end-users, these activities do not only concern the mere understanding of the functionalities that the system provides to its users. Rather, due to the collaborative nature of EUD (see Section 2.4) and tailoring activities (e. g., see [Pipek and Kahler, 2006]), appropriation is a set of social activities (see [Pipek et al., 2009b]) aimed at integrating and shaping the appropriate usage of the system within work practices. As described in [Wulf et al., 2008], especially dealing with collaborative work practices, it is necessary to take into account the fact that system tailoring activities, and consequently the related appropriation activities, involve a heterogeneous set of end-users who are characterized by different levels of qualification and interest, and which can change over the time.

In this light, supporting EUD is a way to provide end-users with software systems that can be considered as an infrastructure with the native support for appropriation activities. In particular, as depicted in [Stevens et al., 2009], an EUD-compliant system provides end-users with a tool that is characterized by an adequate level of flexibility, with the result of encouraging its end-users to undertake appropriation activities.

2.4. Meta-design for Practice-oriented Customizable Environments

Since EUD gives end-users the possibility to customize their systems, this requires software engineers to conceive, design and develop software environments that provide end-users with a set of tools that empower them to autonomously adapt their own collaborative systems. *Meta-design* [Fischer, 2003; Fischer et al., 2004] provides software engineers with set of useful principles to design software environments that users can autonomously customize to their specific needs. As described in [Giaccardi, 2005], «the notion of meta-design has been applied in many fields, including graphic design, industrial design, information architecture, and system design». Focusing on the system design field, *meta-design* can be defined as "*design for designers*" [Fischer and Scharff, 2000]. As described in [Ardito et al., 2012], this concept can be represented as a two-phase design process: while the first phase concerns the design and development of the environment that end-users can customize, the second phase consists of the operations that end-users perform to customize this environment to cope with their local needs.

Like what happens in *participatory design* (see Section 2.2), but more deeply, in *Meta-design* techniques end-users are active contributors in the design of their systems, relaxing the rigid boundaries between the different stakeholders involved in the design processes and promoting them to the role of co-designers (Table 2.1 summarizes *Meta-design* features, comparing them with those of both traditional and participatory design approaches). *Meta-design* adopts a *bottom-up* approach that allows end-users to create spaces of solutions rather than specific solutions to address specific problems. In fact, *meta-design* allows to create new solutions that are the result of a co-creation process in which context and exceptions constitute one of the most relevant aspects, and this requires the involvement of all the system stakeholders (i. e., all kinds of end-users, like employees and managers) in order to negotiate the optimal solution by choosing it within a range of possible ones.

Thus, adopting a *meta-design* approach asks software engineers to provide users with a "new generation" of *open*, *evolvable*, *domain-oriented* environments (see Table 2.2). However, the construction of such environments is not easy since the proposed functionalities could introduce the rigidity that the approach aims to eliminate: for example, implicitly imposing a domain model. A way to overcome this risk is to adopt ethnographic techniques to uncover the work practices (in terms of collaborative actions and conventions) and to distil on their basis the requirements of the environment that would support the construction of applications coherent with the observed practices. This approach has

Traditional Design	Meta-design
Guidelines and Rules	Exceptions and Negotiations
Content	Context
Certainty	Contingency
Resolution	Emergence
Top-Down	Bottom-Up
Creation	Co-Creation
Specific Solutions	Solutions Spaces

Table 2.1.: Traditional design versus meta-design (extracted from [Fischer and Giaccardi, 2006])

been followed in the definition of the WOAD framework that is presented in the next chapter.

Concept	Implications
Convivial Tools	Allow users to invest the world with their meaning and to use tools for the accomplishment of a purpose they have chosen
Domain-Orientation	Bring task to the forefront; provide time on task
Open, Evolvable Systems	Put owners of problems in charge; in open systems, extension is an essential part of use
Underdesigned Systems	Create seeds and constructs for design elaboration at use time
Collaborative Work Practices	Support design communities and the emergence of power users

Table 2.2.: *Meta-design* core concepts and the related implications (from [Fischer and Scharff, 2000])

3 The WOAD Framework

Contents

Documents are fundamental in supporting collaborative work practices, which are strictly related to the conventions that each group of employees has developed over time. Consequently, to be really effective, any kind of document-based collaborative system should take into account the existence of local conventions related to documents, and it should also provide end-users with the ability to use its functions without needing to change their work habits (see Chapter 1). Moreover, during the design of the system, it is crucial to take into account that, in collaborative work settings, each document is not an isolated artifact, rather it is intertwined in a set of relationships that constitute a web of coordinative artifacts (see Section 1.2). Therefore, to provide an effective support for cooperative work practices, document-based collaborative systems should support users to perform the same operations that they are accustomed to doing with paper-based documents in order to promote *collaboration awareness* inside their work group. To reach this goal, collaborative systems can convey additional, commonly agreed *Awareness Promoting Information (API)* on top of documents (see Section 1.4.1). However, due to the heterogeneity of work domains and to the local nature of collaborative work conventions,

the current document-based systems provide a limited support to the wide range of collaborative work practices. As discussed in Chapter 2, allowing end-users to autonomously tailor their document systems to their local (and, in some cases, extemporaneous) needs can result in a better support to their cooperative activities.

In light of these requirements the *Web of Active Documents (WOAD)* framework [Cabitza and Simone, 2010] has been conceived. WOAD is a *design-oriented* framework that encompasses both a *conceptual model* and a *reference software architecture*, and combines the concept of *web of documental artifacts* with the notion of *active document*.

First of all, this chapter will describe the notion of *active document* and in which way it has been adopted within the WOAD framework. Subsequently, the chapter focuses on the description of the documents in the WOAD framework. The third section will describe the first WOAD-compliant proof of concept, i. e., *Process-oriented Documentation (ProDoc)*, that has been conceived as a prototypical EPR. The next two sections go through the description of two attempts to make WOAD EUD-compliant, in order to allow end-users to be autonomous in the definition of WOAD mechanisms, i. e., the proactive rules that allow WOAD documents to be active. Finally, the last section focuses on the description of the WOAD reference architecture that has been conceived to fully support the features of the WOAD framework described in the initial sections of the chapter.

3.1. Active Documents

The concept of *active document* is straightforward: an active document is a digital document that is augmented with a set of active behaviors. The active document notion was born at the *Xerox PARC*[1] from the work of Robert Spinrad [Spinrad, 1988]. This concept led to a rapid spread of different, but in some case intertwined variants (e. g., see [Zellweger, 1988, 1989; Bier and Goodisman, 1990; Bier and Pier, 1991; Bier, 1992; Terry and Baker, 1990]). Scripted Documents [Zellweger, 1989] augmented digital documents with active behaviors with the aim to support the creation of paths among multimedia documents (e. g., to couple a text written in a foreign language with a sequence of audio documents that contain the related pronunciation). Bier proposed EmbeddedButtons [Bier and Pier, 1991; Bier, 1992], a prototypical architecture in which documents were conceived as user interfaces, and each one of their elements could act as buttons that trigger some active behavior: for instance, elements of a document could be configured to show a menu with a list of entries that allow, when a user selects them, to execute some commands, e. g., to change both the font face and font size of the selected

[1] `http://www.parc.com/`

text or to draw a box around it. Active Tioga [Terry and Baker, 1990] made digital documents able to be active in two distinct ways: dynamically computing parts of the content of a document (e. g., to insert a *Table of Contents*) and reacting to changes in a document with some activities (e. g., checking constraints on some parts of the document, for instance, checking if *Table of Contents*' entries have the correct text in the headings of each section of the document).
Nevertheless, active documents have not been an exclusive interest of *Xerox* researchers. Rather, as described in [English and Tenneti, 1994], active documents aroused research interests of a quite assorted group of companies and academic researchers. Also in these cases, active documents covered a wide range of specific behaviors: from the *Apple CADoc*™ [Towner, 1988], which allows the definition of active behaviors to perform automatic data updates, to the *IBM Quill* document editor [Chamberlin et al., 1988], which allows documents to perform active changes in the text formatting. In [English et al., 1990], *Interleaf* researchers proposed active documents as structured documents in which the objects constituting documents can interact among themselves, for instance with the aim to change the language of the user interface or to manage multimedia objects (e. g., pictures). Köppen [Köppen and Neumann, 1998] proposed the platform-independent *Active Hyperlinked Documents* (AHDs), in which active behaviors can be defined to support a set of heterogeneous situations: from the control of contents and layouts of documents to the support of collaboration within groups. Werle [Werle and Jansson, 2001] presented active documents as a way to support collaborative work: documents were considered as an active participant in work activities that can perform some active behavior according to both their content and the context in which they are used (e. g., when a document detects the presence of more than one users in a meeting room, it autonomously starts to present itself through the projector in the room). Finally, Nam proposed the concept of active documents as documents that include both data and business rules [Nam and Bae, 2002; Nam et al., 2003], with the aim to support business constraints over content in documents (in a purchase order document, for instance, required fields must be filled in and, if the customer is a premium one, the document will automatically apply a discount).

The WOAD framework active documents has been inspired by *Placeless Documents* [Dourish et al., 2000b]. Placeless Documents were conceived adopting a property-based approach, in order to overcome many of the limits of traditional hierarchical techniques that are usually adopted to organize documents (e. g., traditional file systems): while hierarchical approaches force users to define a rigid organization of documents that poorly reflects the true organization needed by users, document properties contribute to make

the organization of documents more flexible and meaningful to support users in doing their tasks. Moreover, Dourish introduced the concept of *active property*, i. e., a document property that carries some user-defined executable code to specify a specific document behavior [Dourish et al., 2000a]. For instance, users can augment a document with an active property that makes the document itself able to automatically create a backup copy on a secure storage medium. Moreover, active properties can be defined to monitor changes in documents and, accordingly, either to send notifications to some user or to manage document versioning.

However, despite the flexibility level that active properties allow to reach, they require system end-users to have a certain degree of skill in formalized languages and their syntax contraints. As described in Chapter 2, system end-users are not interested in programming, and this could result in the nullification of the advantage of being able to augment documents with user-defined active behaviors. The WOAD framework overcomes this limit by adopting an approach similar to that proposed by Nam (see [Nam and Bae, 2002]): documents are augmented with a set of user-defined proactive rules, which can be defined through symbolic and declarative expressions. The proactivity of rules has been inspired by the result of some observational studies, which showed that the most simple and, at the same time, powerful concept that the unskilled end-user can understand is *reactive behavior* [Cabitza and Simone, 2009b]. The concept of reactivity as a way to define computable sequences is not new: it has already been proposed in the *Chemical Reaction Model* [Banâtre et al., 2001] and the related *Gamma formalism*. *Gamma formalism* allows users to define a program as a simple pair composed of a *reaction condition* and an *action*. Moreover, such formalism has the advantage to allow users to express executable code in a very abstract way, without forcing them to comply with the constraints of traditional programming languages (e. g., sequentiality) and to focus on the definition of the logic of the active behavior.

3.2. WOAD Documents in Detail

WOAD documents are composed of two distinct but strictly intertwined parts: (i) the *passive* part, which is responsible for displaying and storing the document contents that users inscribe in the document as in any kind of traditional digital document, and (ii) the *active* part, which is called *mechanism* and contains the set of document-related rules. These rules do not change document contents; rather, the aim of the rules is to proactively convey the needed APIs, according to local work practices and conventions

(see Section 1.4.1). In particular, APIs change document's affordances[2], modifying how documents appear.

To define the passive part of a document, WOAD adopts an approach that has become quite common in many applications (e. g., see [Morrison and Blackwell, 2009]): in short, WOAD tries to mimic the paper-based documents look-and-feel. The structure of the passive part of a document is defined by a *document template* that describes only the topological arrangement of the basic document components, which are called *Documental widgets (Didgets)* [Cabitza et al., 2009b], i. e., coherent sets of (one or more) data fields (e. g., simple text fields, multi-line text areas, or check boxes). For instance, with respect to the healthcare scenario (see Section 1.3), a clinical document template could contain the *patient didget*, which reports the usual patient personal data (e. g., name, surname and date of birth), and the *vital parameter Didget*, which summarizes the values of the inpatient's vital parameters (e. g., body temperature, minimum and maximum blood pressure, and the heart rate). Didgets are reusable document components that can be used to define different document templates. A Didget can be defined as *multiple*, to handle those data that are needed to be organized in a tabular format (e. g., the set of vital parameters of a newborn within few moments after delivery or the data of all the previous inpatient's clinical examinations). Moreover, a Didget can be defined to be *local* or *global*, and accordingly it can replicate or not its data among the instances of a single document template. This approach results in a high level of flexibility with respect to the need to change both document layout and the pieces of information managed through the document: in fact, defining document templates only as a spatial arrangement of *Didgets* strongly increases the possibility of users to customize the document templates like what they were accustomed to doing with paper-based documents, since with respect to traditional document-based systems the underlying data model can be continuously adapted according to the Didgets that will be further added or removed.

On the other hand, defining the active part of documents (i. e., *WOAD mechanisms*) corresponds to the definition of sets of document-related rules, i. e., *if-then* conditional statements composed of two complementary parts: (i) the *if-part* contains a set of conditions that could be defined on both documents' contents (e. g., to check that the value of the inpatient's maximum blood pressure exceeded a certain threshold) and contextual information (e. g., the current date and time, or the expertise level of the clinician who is managing the document), and (ii) the *then-part* contains a set of one

[2] The concept of *affordance* was first introduced by the psychologist James J. Gibson in [Gibson, 1977], but it was thanks to Donald Norman [Norman, 1988] that this concept was introduced in the Human-Computer Interaction (HCI) research field: affordances provide users with additional information that can be useful to understand how they can interact with an "object".

or more actions with the aim «to support the access to the content, its use, the ways it is displayed to make it more meaningful, as well as towards how to make a connection between the content and some other documents either explicit or tacitly meaningful [Cabitza and Simone, 2010]» by conveying the suitable APIs, when all conditions are met. The mechanism conditions are evaluated using a congiuntive logic (i. e., the `AND` logic operator): the whole set of mechanism conditions must be verified in order to trigger the related set of actions. On the other hand, to define a disjiunctive set of conditions (i. e., in which some conditions requires to apply the `OR` logic operator), it is sufficient to split the original mechanism into a set of mechanisms whose *if-parts* are not valid at the same time.

3.3. A First WOAD Proof of Concept: ProDoc

The WOAD framework sees its first prototypical implementation in *ProDoc*, which has been designed as an innovative *Electronic Patient Record (EPR) (EPR).* The development of ProDoc has given the possibility to undertake some preliminary validation sessions of the WOAD framework within the healthcare reference domain [Cabitza et al., 2009b], in which clinicians cooperate intensely and their care activities are mediated by the documents they use.

ProDoc is a prototypical document-based application that has been tailored to meet the requirements of the healthcare domain. ProDoc is as a highly flexible EPR that acknowledges three particular requirements of clinicians [Cabitza et al., 2009b]. With *"let me keep my folders!"* clinicians express the need to have an abstraction of the concept of folder, in which they can enact a non-static arrangement of documents they need to perform the care tasks for their inpatients, e. g., allowing an easy re-arrangement of the documents that pertain to an inpatient according to the extemporaneous needs of the doctor who is examining the inpatient. *"Let me do what I do on paper"* is related to the need to keep alive all the interactions that clinicians perform with paper-based documents, e. g., the possibility to annotate some information on documents to promote the awareness of colleagues for relevant issues (see Section 1.4). Finally, *"integrate data and processes but don't mix'em up"* expresses the need to keep a non-mandatory process map with the aim to promote group awareness and coordination, and to provide an alternative way of organizing documents in terms of when they need to be created and for what aim.

ProDoc allows clinicians to use a graphical user interface, with the same look and feel of paper-based documents they usually adopt in their daily work practices, both for data

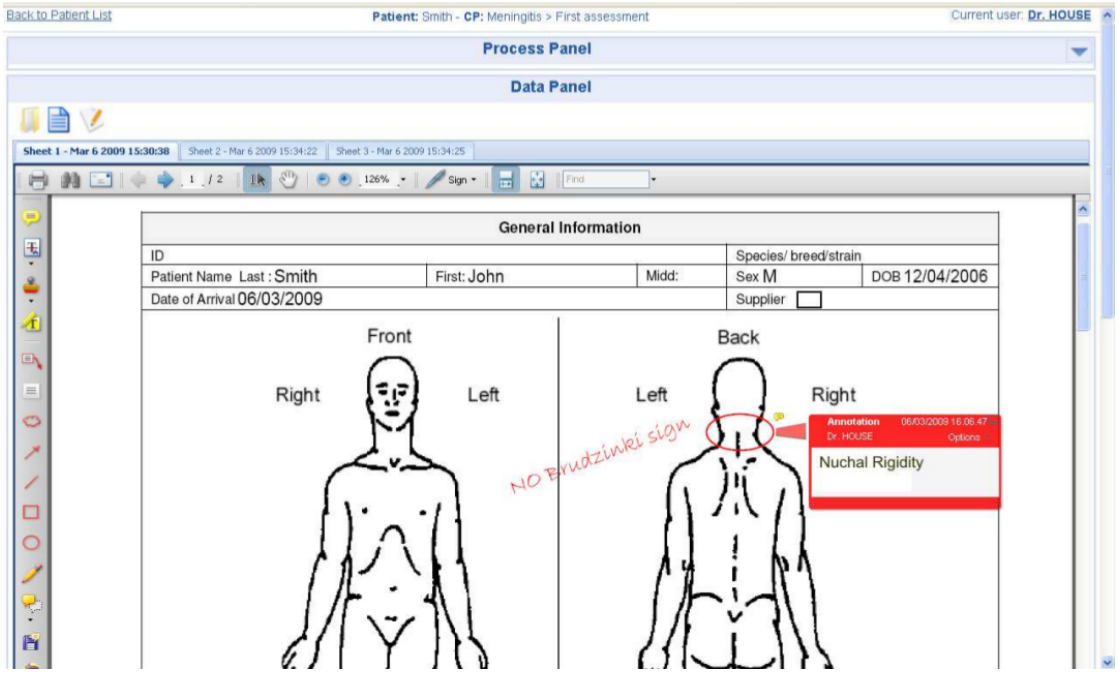


Figure 3.1.: The ProDoc *Data Panel* (from [Cabitza et al., 2009b])

entry and document retrieval operations. Moreover, documents are grouped according to the inpatient to which they refer to (the *"let me keep my folders!"* requirement).

ProDoc digital documents rely on the *Portable Document Format (PDF)* technology that provides clinicians with digital documents with the ability to reproduce some of the typical interactions of the paper-based documents (the *"let me do what I do on paper"* requirement). For instance, ProDoc provides clinicians with the possibility to annotate documents (see Figure 3.1). Moreover, ProDoc allows clinicians to create and fill in documents without being forced to follow a rigid sequence, which usually in traditional EPRs, is "imposed" at design-time. ProDoc leaves clinicians free to progressively create the documents they need according to their local conventions. For instance, with respect to the other hospital wards, in the Emergency Ward the creation of the patient's Personal Data document has a lower priority than the creation of the Anamnesis document.

With respect to the *"integrate data and processes but don't mix'em up"* requirement, ProDoc keeps a particular kind of document always visible: an active process map that clinicians have locally defined on the basis of their common agreements and that depicts the care process according to which inpatient care activities should be carried out. Through this particular type of document, which in the healthcare work setting is called *Clinical Pathway* (e. g., see [Bleser et al., 2006]), ProDoc allows clinicians to get

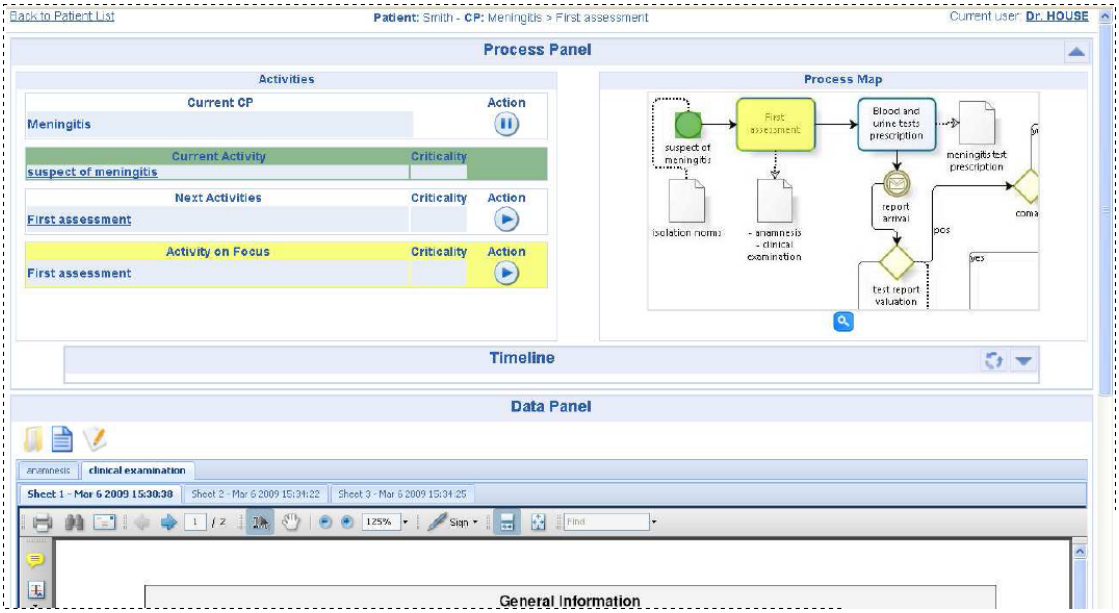


Figure 3.2.: The ProDoc process map (from [Cabitza et al., 2009b])

access to any part of the whole set of documents without being constrained by any kind of rigid workflow. At the same time, this process map promotes clinicians' awareness of the intended flow of care activities: it is possible with a glance to get an overview about the current activity, activities that have already been performed and those that have not yet been performed (see Figure 3.2).

Finally, ProDoc user interface embeds a *timelined view* that provides a browsable view of the sequence of relevant events, which are arranged according to the moment in which they occurred and are augmented with the reference to the activity in which they have been generated (see Figure 3.3). In fact, the timeline allows clinicians to be aware of both the process history and any relevant event that occurred during their care activities (e. g., the creation of a new document or the update of an existing one).

The healthcare domain provides a very good work setting to validate WOAD framework support for collaborative practices. Nevertheless, the practice to couple documents with a process map, which allows to manage documents according to well-defined work practices and conventions, is not a peculiarity of the healthcare setting; rather it is a common practice in a wide range of domains. This makes it possible to use ProDoc in other document-based cooperative work domains. in this light, for instance, ProDoc has been also tested and validated in the archaeological domain [Locatelli et al., 2010]. In this case, ProDoc has been validated in some user sessions involving archaeology students during their practical study activities in the excavations with archaeological finds: these

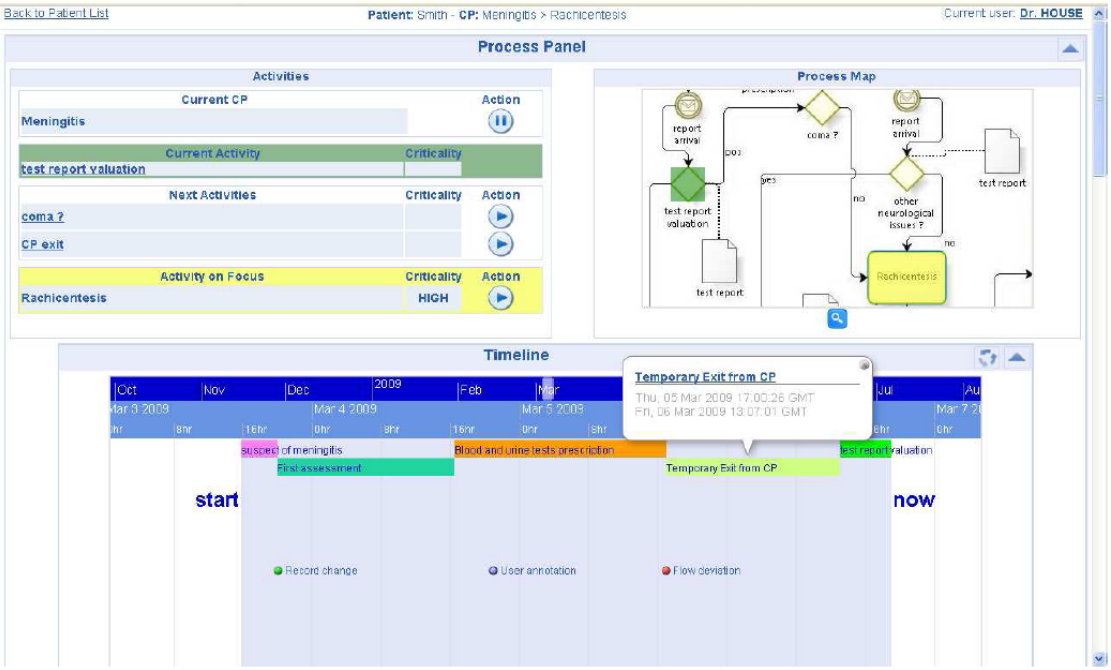


Figure 3.3.: The ProDoc events' timeline (from [Cabitza et al., 2009b])

activities require archaeologists to cooperate and to continuously produce and update a huge amount of documents.

ProDoc was conceived to check the main tenets of the WOAD framework concerning the possibility to collaboratively annotate documents and to express the relationships between documents and the process that uses them. As such, it offers limited capability for the promotion of awareness among the collaborative actors. However, it was helpful for checking the overall approach and defining an initial architecture of the WOAD framework.

The next effort was dedicated to the design of the functionality supporting the definition of the WOAD mechanisms supporting awareness promotion. The next two sections will go through the description of two attempts to overcome this limit. The former is on the LWOAD denotational language (see next section). The latter is on a form-based visual mechanism editor, which has also been the first attempt to make the WOAD framework compliant to the EUD tenets (see Section 3.6).

3.4. LWOAD

In order to define WOAD mechanisms, the WOAD framework was endowed with a denotational language that is called LWOAD [Cabitza and Simone, 2009b]. The aim of LWOAD is to make domain-expert end-users autonomous in defining their own WOAD mechanisms, and this contributes in making the WOAD framework consistent with the *End-User Development (EUD)* tenets (see Chapter 2). The idea behind LWOAD is to let users define their mechanisms using a semi-structured natural language that can be easily translated into machine readable formalisms (LWOAD) and then executed by a rule engine.

The LWOAD syntax reflects the WOAD mechanism structure (see Section 3.2). The *antecedent* represents the mechanism's *if-part* and contains the set of conditions that the *Mechanism Interpreter* has to check (e. g., with reference to Figure 3.4a, to check if the newborn's blood pressure is lower than `70 mmHg`). In the *antecedent*, end-users must define a list containing entities of two different types: (i) *facts* (e. g., the `document-fact`), which are evaluated to check their truthfulness and to provide access to the related data (e. g., the `?bp` placeholder gives access to the value of the `current-blood-pressure` document field), and (ii) *tests*, which allow to define comparisons of data values (e. g., `test (< ?bp 70mmHg)`). In particular, facts can be characterized by specifying their attribute values, or placeholders (e. g., `?bp`) that allow to refer to these values "outside" the fact in which they are declared, using a prefix notation[3]. The values are managed through a *key-value* data structure (e. g., `(document-fact (record-id ?pr) (name SS) (latest-weight ?lw))`, as shown in Figure 3.4a). For instance, LWOAD provides the *relation-fact* (see Figure 3.4b) that is useful for checking if someone has previously defined any kind of relationship between two different documents, e. g., in the case of the so called "positive redundancy" [Cabitza and Simone, 2008]. This fact is characterized to allow users to specify five distinct parameters: (i) the name of the relation, (ii) its description, (iii) the granularity level of the relation (i. e., class, e. g., clinicians and patients, or instance, e. g., Dr. Red and Mrs. White), (iv) the reference to its source, and (v) the reference to its target.

On the other hand, the *consequent*, which represents the mechanism's *then-part* (see Section 3.2), defines the sequence of actions that the Mechanism Interpreter must execute to convey the right API, when all the conditions in the *antecedent* are met (e. g., taking again into account Figure 3.4a, to convey the *Criticality API* to make clinicians aware of

[3] The *prefix notation* is also known as *Polish notation* and consists in declaring the operator followed by the list of its operands, e. g., `(operator operand_1 operand_2 ... operand_n)`. Some high-level programming languages, like the LISP family, use the prefix notation.

the existence of a newborn's critical situation).

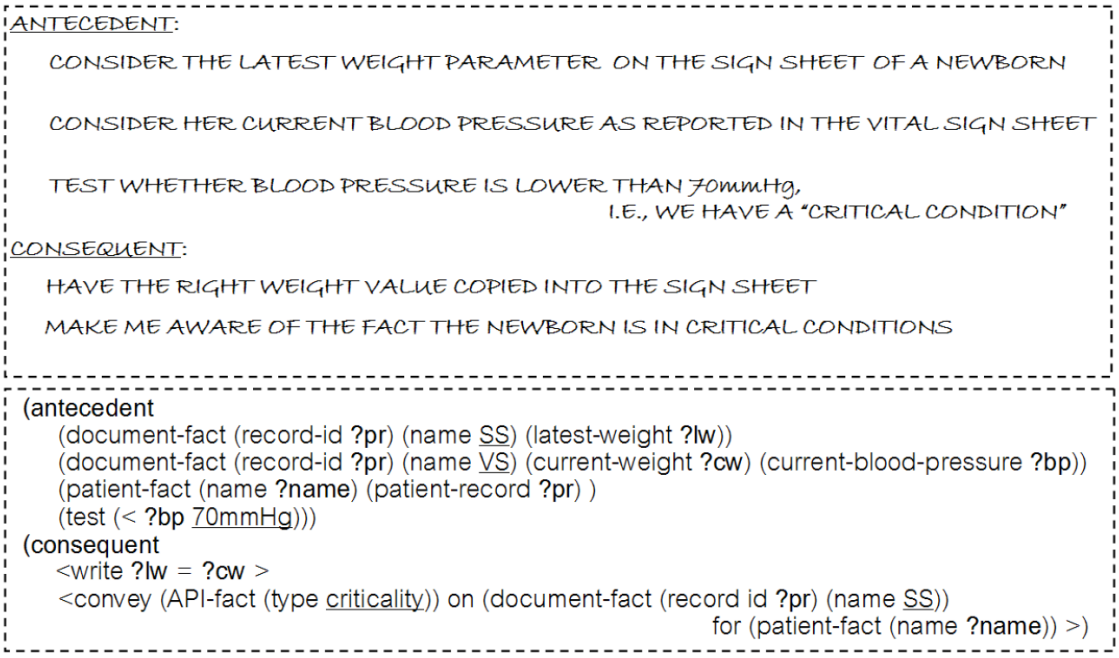
3.5. The First WOAD Mechanism Editor

LWOAD was useful for checking the overall acceptance of a rule-based approach by the end-users, but its usage would require the presence of a designer, since its syntax is too close to one of the classic programming languages, although the translation from the natural language formulation was understood by the end-users.

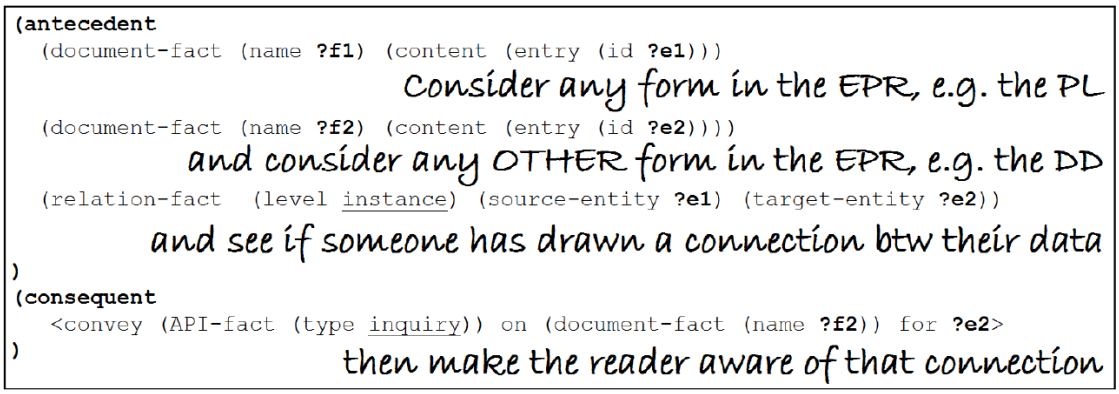
Thus, in order to overcome this LWOAD limit, the visual composition of mechanisms is the approach that has been considered to be more intuitive to give to a larger number of end-users the possibility to autonomously create the mechanisms they need. The first attempt to create a visual editor to allow the visual composition of the WOAD mechanisms resulted in a form-based editor by which users can create their mechanisms following a predefined and step-like process (an extensive description of this visual mechanism editor is reported in [Cabitza et al., 2011a,b]).

As shown in Figure 3.5, the user interface is arranged in a three-column layout. The smaller column, positioned on the left side of the user interface, contains three distinct panels: (a) the list of existing document templates, (b) the trash area, and (c) the list of already created mechanisms. The remaining part of the user interface is composed of two columns with the same width: while the most central one represents the mechanism's *antecedent* (i. e., the *if-part*), in which users define the set of medchanism conditions, the right one represents the mechanism's *consequent* (i. e., the then-part), in which users define the set of APIs that the mechanism will convey when conditions are met (see Section 3.2).

Most of the mechanism composition process is based on drag'n'drop operations and selections from short closed-option menus. The user who wants to compose a mechanism simply picks up the document template she needs in the list on the top of the left column and drags it over one of the other two columns. When the user drops the selected document template in one of these columns, a new panel representing the document template is added to the column in which the drop took place. According to the column in which the document template is dropped, the related panel allows to define a set of conditions or a set of APIs to be conveyed respectively. Each one of these panels contains a set of three drop-down menus that are followed by an input text field. The drop-down menus allow the user to select one of the Didgets in the document template (e. g., the *Personal Data* Didget), one of the selected Didget data fields (e. g., the inpatient's name), and the conditional operator (e. g., either the "`equals`" or the "`contains`" operator) or the API to be conveyed respectively. On the other hand, the input text field allows



(a) A WOAD mechanism to convey an API related to a newborn's critical situation (*Criticality API*).



(b) A WOAD mechanism to convey an API (*Inquiry API*) to make clinicians aware of the existence of a relation between two documents.

Figure 3.4.: Two examples of the LWOAD mechanisms, compared with the same specifications in natural language (from [Cabitza and Simone, 2009b])

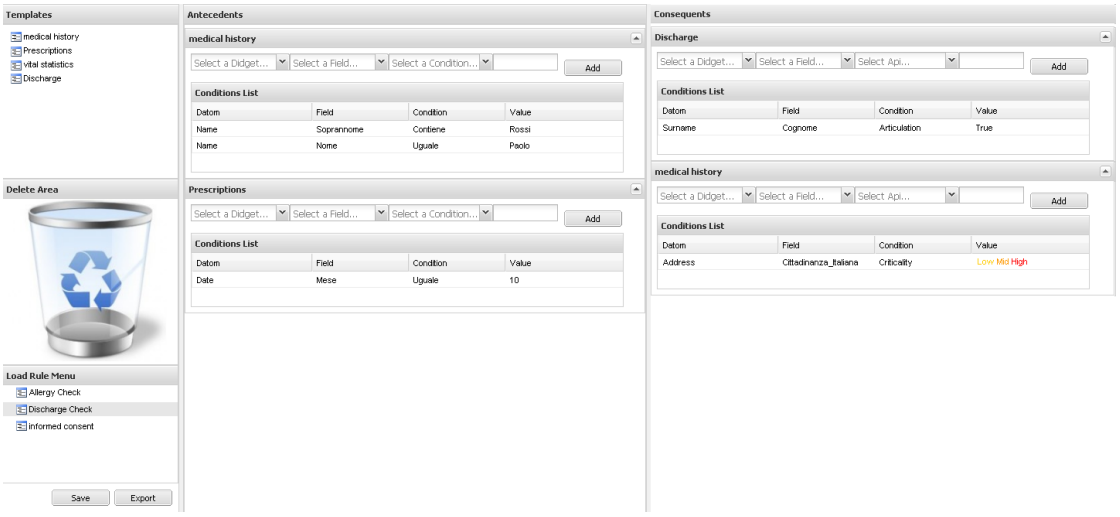


Figure 3.5.: The first WOAD Mechanism Editor (from [Cabitza et al., 2011a])

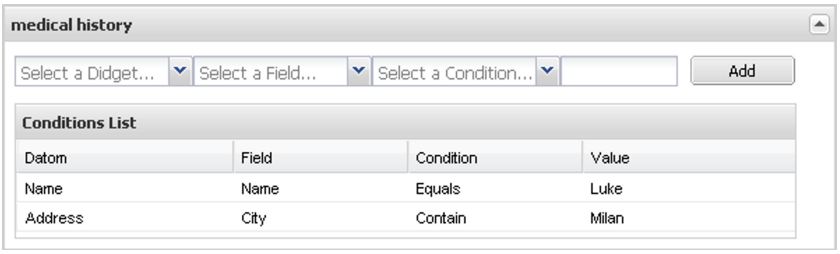


Figure 3.6.: The mechanism's *if-part* form (from [Cabitza et al., 2011b]).

to specify a value that parameterizes either the condition the user is defining or the API the user wants to convey. These four input fields are followed by a button that allows to add either the new condition or the new API to the mechanism. The remaining part of the panel below the set of input fields is taken up by a table that summarizes the already defined conditions or APIs related to the document template to which the panel is pertaining (Figure 3.6 shows a document template's panel to specify the set of conditions). Unlike what happens in the panels pertaining conditions, in which the users can only specify a single value to parameterize the condition they are defining, in some cases (e. g., to react in different ways, according to the user's expertise level), when users add a new API in the panel pertaining a mechanism's *consequent*, they may specify more than a single value for the same API parameter: to make this operation as simple as possible, the visual editor shows to users some specialized dialog boxes, one for each type of APIs (e. g., Figure 3.7 shows the dialog to parameterize the *Criticality API*), through which they can easily specify all the API's parameter values.

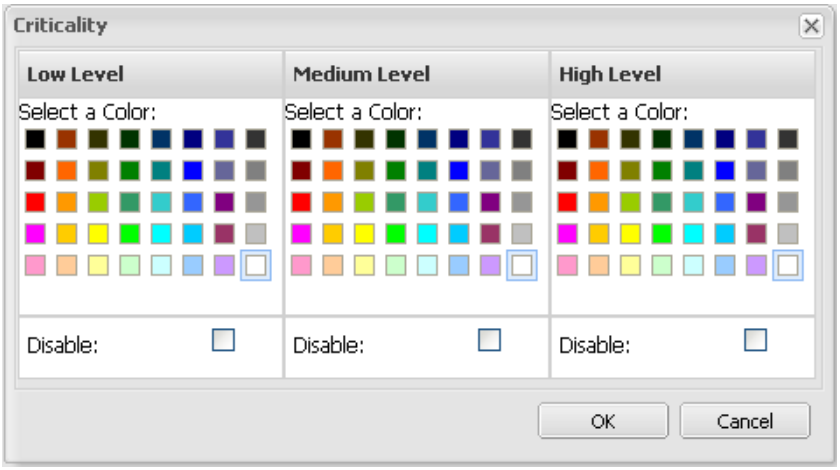


Figure 3.7.: The dialog box through which users can specify the *Criticality API* parameter values (from [Cabitza et al., 2011b])

3.6. The WOAD Reference Architecture

To support the concepts that have been outlined in the previous sections, the WOAD framework encompasses a reference architecture (see Figure 3.8), which is described in [Cabitza and Zorzato, 2010, Section 3].

The *Layout Engine* is the component that receives users' requests for documents and renders the retrieved active documents[4]. As shown in Figure 3.9, a request for a document is forwarded to the *Document Manager*, which is the component responsible for building the passive part of the document. The *Document Manager* interacts with two distinct components to complete its task: the *Document Data Repository* and the *Template Manager*. While the role of *Document Data Repository* is to manage data persistence, the *Template Manager* is the component that is responsible for storing and providing access to document templates. As a matter of fact, the *Document Manager* simply couples the document template with the contents that are related to the requested document. In order to do this task, the *Document Manager* is divided into two cooperating subcomponents: the *Didget Manager* and the *Document Builder*: while the former manages Didgets and maintains them synchronized with the *Document Data Repository*, the latter is responsible for retrieving the needed document template from the *Template Manager* and to fill in it with the needed Didgets, which are retrieved from the *Didget Manager*.

On the other hand, the *Document Manager* also interacts with the *Mechanism In-*

[4] Since the initial implementation of the WOAD framework, the *Layout Engine* is any Internet browser with the full support for the *World Wide Web* standards, i. e., HTML, CSS and JavaScript.

terpreter. The *Mechanism Interpreter* is the component responsible for asynchronously checking the WOAD mechanism conditions and to activate the mechanisms whose conditions are met. The *Mechanism Interpreter* is the rule engine that constitutes the heart of WOAD active documents: currently, it is based on the *RETE algorithm* [Forgy, 1982] and in particular on its Java implementation, i. e., *JBoss Drools*[5]. This algorithm executes rules, i. e., WOAD mechanisms, adopting a resolution strategy that is based on both specificity and currentness. *RETE* is an efficient *pattern matching* algorithm aimed at implementing rule-based systems in which that rules are quickly and efficiently[6] executed.

The execution of a WOAD mechanism requires the *Mechanism Interpreter* to interact with the *Markup Tagger* component that is responsible for coupling the passive part of the requested document, coming from the *Document Builder*, with the metadata that have been produced by the *Mechanism Interpreter*, in order to convey to users the APIs specified by the WOAD mechanisms. The *Markup Tagger* translates WOAD mechanism metadata into a format that the *Layout Engine* can manage (e. g., HTML tags' attributes), and, in so doing, the *Markup Tagger* makes the requested document active.

Moreover, the *Mechanism Interpreter* constantly monitors Didgets interacting with the *Document Manager*, and in particular with the *Didget Manager*. In this way, when the content of a WOAD active document is modified, the *Mechanism Interpreter* automatically re-processes WOAD mechanisms to best fit the updated context.

This architecture was the starting point for the implementation effort described in this theses. The next chapter describes how it has been modified and extended.

[5] `http://www.jboss.org/drools/`

[6] The *RETE* algorithm is designed to increase rule execution speed, even if this implies a higher memory usage (see [Forgy, 1982] for details). The algorithm efficiency is mainly due to the following features: (a) *node sharing* reduces, or eliminates, certain types of redundancy, (b) *storing partial matches* allows to avoid complete re-evaluation of all facts each time the knowledge base changes, and (c) an *efficient removal of memory elements* is enacted when facts are retracted from working memory.

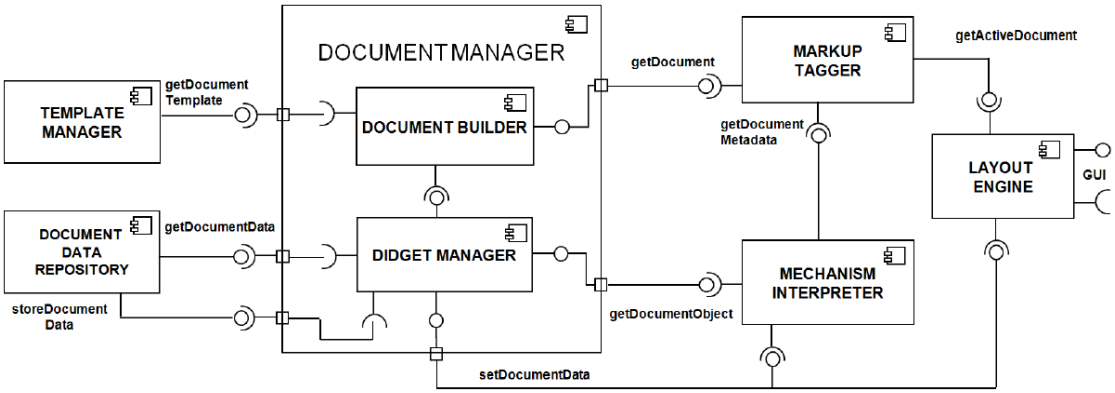


Figure 3.8.: The UML Composite Structure diagram of the original WOAD reference architecture (from [Cabitza and Zorzato, 2010])

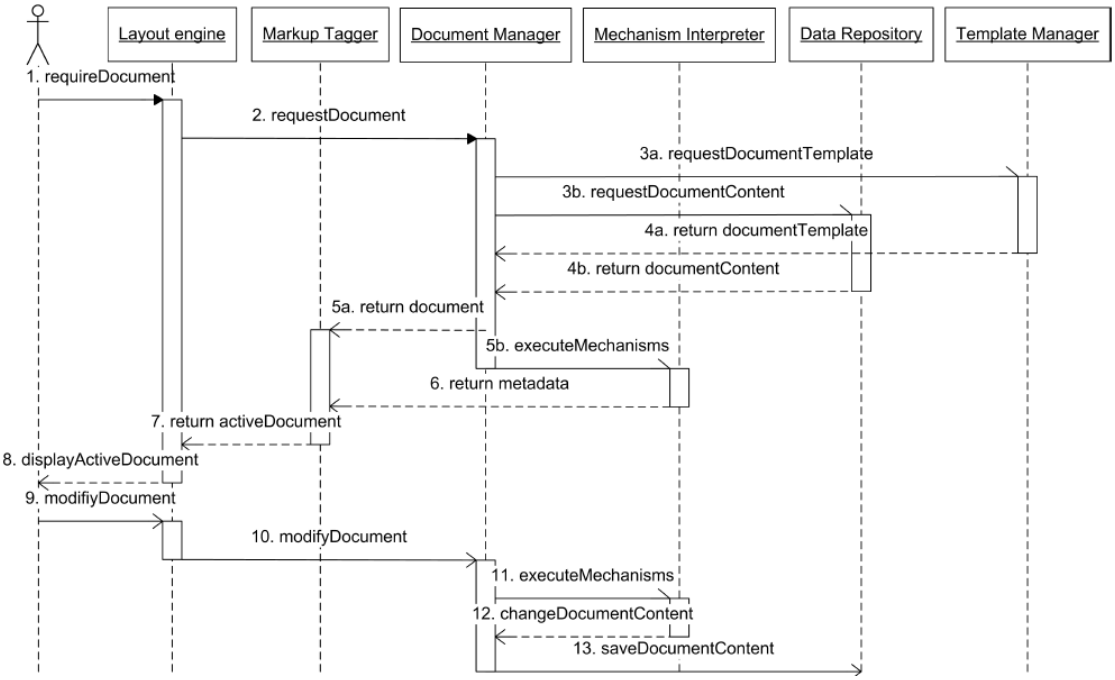


Figure 3.9.: The UML Sequence diagram of the original WOAD reference architecture (from [Cabitza and Zorzato, 2010])

4

Open Problems and Proposed Solutions

Contents

Even if the WOAD framework (see Chapter 3) represented a significant improvement toward the design of effective document-based collaborative systems that can be easily customized to meet cooperative local needs and work practices, the framework presented some relevant problems that actually limited its applicability.

In particular, two aspects have the effect to hinder the effectiveness of a WOAD-compliant collaborative system: (i) usually, employees are accustomed to be autonomous in defining and managing both the structure and the informative content of the paper-based documents they use in their daily work: they perform this task simply using a common word processor to create and modify their document templates, which they can print when it is needed; and (ii) in most cases employees are domain-experts without (or, at most, with very poor) competences and interests in any kind of computer programming techniques: this feel prevents them from being autonomous in the composition of the WOAD mechanisms they need.

In order to avoid to fall back into the same problems and limits of traditional collaborative systems, which have been described in Chapter 2, the challenge is to make the WOAD framework compliant to the EUD tenets. In this light, the idea is to provide

end-users with a customizable environment that allows them to be autonomous in both creating and adapting the document templates they need and in defining the related WOAD mechanisms.

First of all, this chapter will deal with the description of the main requirements of the two solutions that have been conceived to address the above open issues of the WOAD framework, and which contributed to make it compliant to the EUD tenets. Subsequently, the chapter focuses on the description of the architectural choices that allowed to improve the flexibility of the WOAD framework and the tailorability of WOAD-compliant collaborative systems.

4.1. Customizing Documents as Easily as Using a Word Processor

Paper-based documents require employees a very little effort either to be created or to change their structure, both in terms of their arrangement and their informative content, i. e., the data they can manage. Indeed, users can be autonomous in performing these operations thanks to the use of traditional word processors as design environments for their paper-based document templates. In fact, this kind of office automation software unwittingly provides end-users with a *What You See Is What You Get (WYSIWYG)* environment in which they can easily build the document templates they need through simple drag'n'drop interactions: users pick up data fields they need, drag them over documents and drop them in documents at the desired positions. When document template editing is completed, users can store or print it like they are already accustomed to do with any other document they manage with the same word processor.

End-users strongly expressed their need to have document-based systems that allow them manage digital document as they did with their paper-based documents, i. e., the *"let me do what I do on paper"* requirement that have been described in Section 3.3. Even if *Didgets* allow the flexible composition of the WOAD documents following a component-based approach (see [Won et al., 2006; Wulf et al., 2008]), the WOAD framework lacks in supporting end-users to autonomously design the documents they need. Thus, the idea is to support them in doing this task through a visual editing environment (namely, the *WOAD Template Editor*). The visual editor must provide users with a user interface similar to those provided by traditional word processors, without the need to force them to learn to use a completely new and unknown user interface. In particular, the user interface must provide users with (i) an editing area that mimics the look and feel of a traditional paper-based document (as in most part of word processors); (ii) the possibility

to compose document templates through a drag'n'drop-based interaction; and (iii) the flexible management of the informative content of document templates, i. e., by adding or removing Didgets, allowing users to adapt them to their constantly evolving needs, without the constraints imposed by a rigid data model. This last feature implies that the editor should be able to constantly adapt the data structures underlying the documents, acting in a transparent manner according to the changes that users make to the document templates.

In this way, the efforts required to employees to move from a traditional word processor to a document-based system would be reduced as the interface looks familiar to them. This contributes to increase the acceptance degree of such systems when they replace traditional paper documents.

The need to cope with the requirement of flexibility of data structures revealed a relevant lack in the initial formulation of the concept of Didget. In particular, this lack is related to the Didgets' ability to share data among different document instances. As Didgets have been defined, they support only a binary level of data sharing: while in *local* Didgets data pertains only to the document instance in which it has been entered, *global* Didgets can only share data among all document instances. In fact, limiting data sharing in this way results in a feature that does not meet real-life situations. For instance, from the analysis of the documents adopted in the healthcare domain emerged the need of clinicians to share some pieces of data only among all the documents pertaining to a single inpatients (e. g., the inpatient's personal data). This required to conceive a more flexible Didgets' data sharing capability (see [Cabitza et al., 2011a,b]). In order to be more consistent with respect to real-life situations, this Didgets' capability has been extended to have more granularity with respect to its initial binary definition (see Table 4.1). This resulted in three distinct levels of data sharing, in addition to the case in which data is not shared and remains *local* to the single document template instance (namely, `G0`): for instance, the value of the daily measurement of the inpatient's body temperature. The other three levels have been conceived to have an increasing degree of data sharing among documents. With the `G1` level the content of a Didget is shared among all the instances of a specific document template that are related to a specific resource (e. g., the same patient). With the `G2` level, a Didget can share pieces of data among the documents that are based on different document templates, but this data is still related to the same resource (e. g., some pieces of patient's personal data, like the patient `id`, her name and surname). Finally, the `G3` level allows a Didget to share its content among all documents, without any constraint both on document template and resource.

	Data Shared Between		
	Instances	Templates	Resources
G0	✗	✗	✗
G1	✓	✗	✗
G2	✓	✓	✗
G3	✓	✓	✓

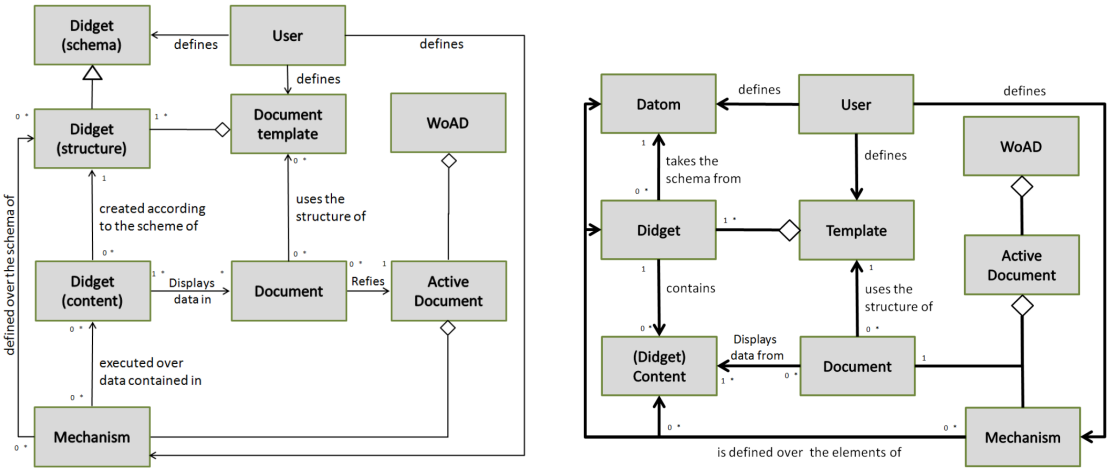
Table 4.1.: The redefined levels of the Didget's data sharing capability (from [Cabitza et al., 2011a])

Users should be able to autonomously configure the level of data sharing of a Didget. Since this operation affects the configuration of the underlying data structures of documents, the visual editor must allow users to set this Didget feature in a visual manner. Similarly, since users can need to manage repeatable data, Didgets have been conceived to support users in the management of this kind of data (see Section 3.2). Thus, also in this last case, the visual editor must allow users to configure this feature of Didgets in a visual manner.

A final issue emerged during the identification of the requirements of the WOAD Template Editor: the initial definition of the three entities that concur in defining the concept of Didget generated a confusion in the interaction between users and WOAD designers. In order to give a more clear definition of this fundamental concept of the WOAD framework, these three entities have been renamed (see Figure 4.1 for a comparison): (i) the *Didget schema*, which defines the group of data fields and their specific features (e. g., format and data type), became *Documental atom (Datom)*; (ii) the *Didget structure*, i. e., the instance of a Didget within a document template (e. g., in the Clinical Examinations document template), simply became Didget; and (iii) the *Didget content*, which represents the data users filled in (e. g., some vital parameters that clinicians filled in one instance of the Clinical Examinations document template), simply became *Content*.

4.2. WOAD Mechanisms for Non-Programmer End-Users

The WOAD framework has been conceived to design document-based collaborative systems that proactively support users in their work practices, providing them with conventional awareness information, i. e., the *Awareness Promoting Information (API)* that have been described in Section 1.4.1. Nevertheless, since users are the only ones who have the full knowledge of the conventions of the group in which they act, in order to provide a really effective support to their collaborative work practices, users must be



(a) The original entities in the WOAD framework (from [Cabitza and Zorzato, 2010])

(b) The more clearly defined WOAD framework's entities (from [Cabitza and Gesso, 2011])

Figure 4.1.: The relationships among WOAD concepts

able to autonomously define their own WOAD mechanisms, i. e., proactive rules, even if they do not have any computer programming skill.

Despite the WOAD framework encompasses the LWOAD denotational language for the definition of the WOAD mechanisms in a flexible and powerful manner, this language is still far to be widely usable by domain-expert end-users, since, like any other formalized language, it must comply with strict syntactic constraints, requiring users to have some skill with programming languages and techniques. In order to make users autonomous in the definition of their WOAD mechanisms, the visual approach has been considered to be more intuitive with respect to LWOAD. Nevertheless, the first attempt to make users autonomous in defining WOAD mechanisms resulted in a dead end. Since this first visual mechanism editor adopted a form-based user interface, which forced users to follow a predefined and step-like process (see Section 3.5), the WOAD mechanisms created using this visual editor were extremely simple and not suitable to represent the most part of real-life conventional rules.

However, the idea to adopt a visual approach to make users able to autonomously define WOAD mechanism has not been discarded. Rather, wanting to adopt a visual approach to design the new *WOAD Mechanism Editor*, the main requirement is to provide users with an intuitive visual metaphor and a usable user interface, which minimize the users' efforts to both learn and use them. In the literature, a heterogeneous set of metaphors have been proposed to visually represent rules, and at present it does not exist a universally recognized or a predominant approach to perform this task (see

Section 5.2.2). However, the WOAD Mechanism Editor should adopt a visual metaphor that is sufficiently simple to be used without requiring to train users prior using it.

Moreover, once a user completed the creation of a new document template, through the WOAD Template Editor, the WOAD Mechanism Editor should immediately give her the possibility to seamlessly use the Didgets related to the new document template within the WOAD mechanisms she wants to create or to modify. On the other hand, since users can also remove Didgets from their document templates, the WOAD Mechanism Editor should be able to make the user aware that the rule she is modifying refers to some Didgets that do not still exist in a particular document template. Consequently, a fundamental requirement is that the WOAD Mechanism Editor must be strictly integrated with the WOAD Template Editor.

4.3. A Platform-independent MVC Model for Digital Documents: XForms

A useful approach that allows to meet the need of a modular user interface (i. e., documents) is the *Model-View-Controller (MVC)* design pattern (e. g., see [Reenskaug et al., 1996]), which allow to separate the data model (namely, the *Model*) from the presentation layer (i. e., the *View*) and the applicative logic (i. e., the *Controller*). Thus, this approach allows to easily design and implement the user interface (i. e., the *View*) without the need of a deep knowledge of applicative logic: it is only needed know the data model. In this way it is easier to design platform-independent user interfaces. The WOAD relies on the MVC design pattern since its initial design, but its first prototypical implementation (i. e., ProDoc) lacks in the full platform independence, even if it has been conceived as a web application. This is due to the fact that document templates have been mimicked using standard PDF forms. In order to overcome this limit, the idea is to adopt a platform-independent solution to define form-based digital documents. In particular, the need is to apply this approach in order to define Datoms.

XForms[1], which is a *World Wide Web Consortium (W3C)* standard that have been conceived to replace the standard *HyperText Markup Language (HTML)* forms, can be useful to combine the MVC design pattern with the need to design and implement a fully platform-independent web application. Through its XML-based syntax, XForms provide a flexible way to define both the user interface and the underlying data model. Thus, XForms is one of the best candidates to simplify the design and the development of platform-independent web application (for instance, see [Cardone et al., 2005]).

[1] `http://www.w3.org/MarkUp/Forms/`

In this light, the idea is to adopt a subset of the XForms syntax to define Datoms, which end-users (e. g., clinicians) would find in the WOAD Template Editor in order to use them to autonomously compose their own document templates (e. g., the *Vital Signs* document template). As a direct consequence, due to the fact that each Datom describes both a piece of user interface and the related piece of the underlying data model, when end-users either compose or modify their own document templates, at the same time they implicitly and increasingly define the underlying data structure. In this way it is possible to easily overcome the limitations of traditional approaches, in which data is stored in pre-defined data structures (e. g., tables) that software engineers define at design-time and that are difficult to adapt to the ever changing users' requirements.

4.4. The Renewed WOAD Architecture

The introduction of the two visual editing component in the WOAD framework (see Section 4.1 and Section 4.2) and the other improvements of the WOAD framework required a (partial) refactoring of the original WOAD reference architecture (see Figure 3.8). The resulting WOAD reference architecture (see Figure 4.2) mainly differs from the original one for the presence of the two visual editors. Moreover, the already existing WOAD components have undergone a substantial reorganization, with the aim to better define the conceptual role played by each WOAD component, with an improvement of the overall maintainability of each WOAD component. On the other hand, the adoption of the XForms standard to define the structure of documents required the redefinition of the Document Builder component. In particular, the Document Builder is composed of two sub-components: the *Document Composer* and the *Markup Manager*. The Document Composer interacts with both the Template Manager and the Didget Manager with the aim to retrieve the document template and all the needed Didgets respectively, in order to compose an empty instance of the requested document (see Figure 4.3). On the other hand, the Markup Manager gets the empty document instance, i. e., an empty XForms document, and it fills in this document instance with the related Didgets' content; subsequently, the Markup Manager translates the resulting document from the XForms syntax to the HTML syntax, which can be managed by the Layout Engine. With regard to both the other phases of creating instances of the documents and those pertaining to the execution mechanisms, the reorganization of the architecture had no effect on the interactions among the components, which actually are the same as those described in Section 3.6.

On the other hand, the introduction of the two visual editors required to support

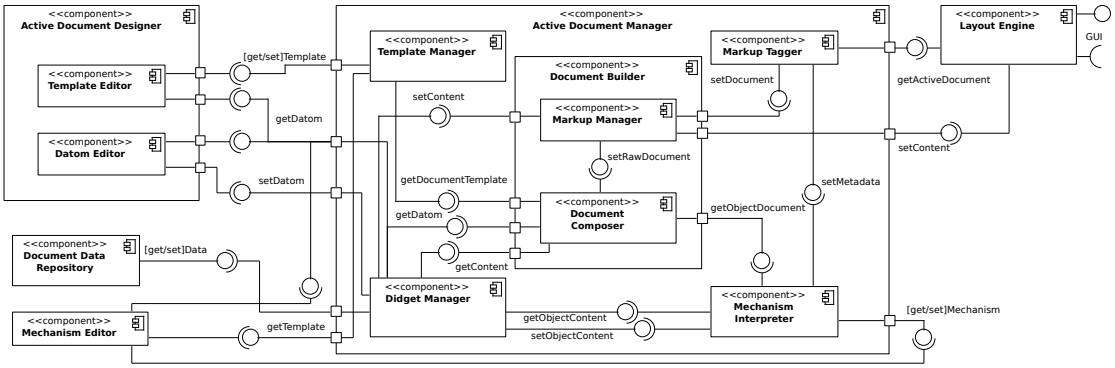


Figure 4.2.: The UML Composite Structure diagram of the renewed WOAD reference architecture (from [Cabitza et al., 2011a])

the communication among these new components and the already existing ones. The WOAD Template Editor needs to communicate with both the Template Manager and the Didget Manager. When a user starts to modify an existing document template, the WOAD Template Editor queries the Template Manager to retrieve the data representing the document template; similarly, WOAD Template Editor queries the Didget Manager with the aim to update the list of existing Datoms and Didgets that the user can add to the document template. When the user drops a Datom, as described in Section 6.1, the WOAD Template Editor creates the related new Didget. This requires the WOAD Template Editor to interact with the Didget Manager in order to make persistent the new Didget. Finally, when the user stores the document template, the WOAD Template Editor interacts again with the Template Manager, with the aim to make persistent the document template. Similarly, the WOAD Mechanism Editor needs to communicate with both the Template Manager and the Didget Manager, but in addition it also needs to communicate with the Mechanism Interpreter. When a user starts to create or modify a WOAD mechanism, the WOAD Mechanism Editor queries both the Template Manager and the Didget Manager with the aim to collect all the existing document templates and the related Didgets, in order to allow the user to use the latter to compose the her WOAD mechanism (see Section 6.2). On the other hand, once the WOAD mechanism is completed and the user chooses to make it available to the system to be executed, the WOAD Mechanism Editor interacts with the Mechanism Interpreter, with the aim to put the mechanism in the set of rules that the rule engine of the Mechanism Interpreter can execute.

Even if the work that has been done also concerned the redesign and the development of most part of the components of the WOAD reference architecture (see Figure 4.2), in

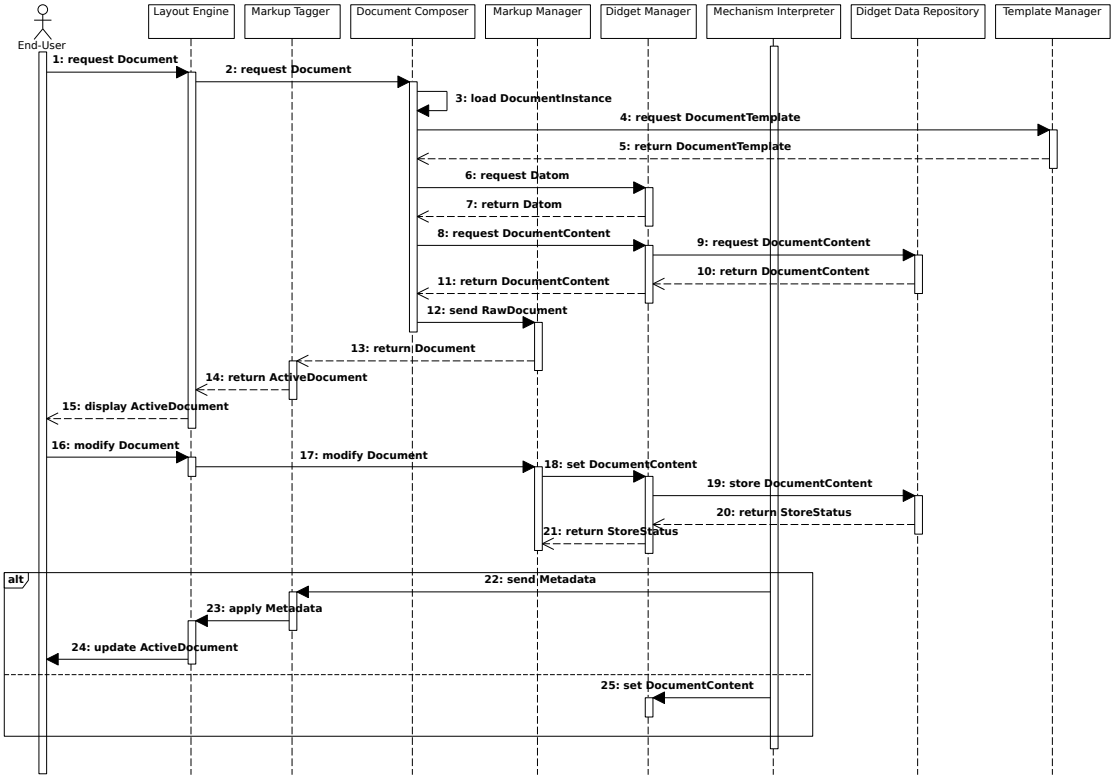


Figure 4.3.: The UML Sequence diagram of the renewed WOAD reference architecture (from [Cabitza et al., 2011a])

the following, next chapters will focus on the description of both the WOAD Template Editor and the WOAD Mechanism Editor, since these two components of the WOAD framework architecture are those that users can interact with and therefore have been validated, as it will be show in Chapter 8 and Chapter 9.

5 Related Works

Contents

The previous chapters introduced both the concept of awareness promotion, with the aim to support and enhance convention-based collaborative work practices, and a rule-based approach to proactively convey conventional awareness information to users on top of their digital documents, i. e., the WOAD mechanisms. The number of solutions to convey awareness informations to users has grown over time. Consequently, these solutions have been adopted to support the needs of awareness of different applicative domains, such as the healthcare domain (e. g., see [Ray et al., 2008; Bardram and Hansen, 2010]). Awareness information are conveyed adopting different approaches that range from notification systems up to complex 3D representations of a virtual environment where actors interact. For instance, Prinz proposed NESSIE [Prinz, 1999], an application independent environment aimed at supporting collaboration by conveying awareness information through an event-based notification system. Benford [Benford and Fahlén, 1993] proposed a spatial model to manage group interaction in virtual environments, in which the objects within the virtual space (including members of the group) interact on the basis of a mutually defined level of awareness. A similar approach has been proposed in [Sandor et al., 1997], which discusses the AETHER spatial model and its prototypical

implementation. Nevertheless, in most cases these awareness systems do not give users the possibility to autonomously define the awareness information they want to convey as the awareness functionality usually flanks other collaborative applications as an application that can be at most customized. In [Simone and Bandini, 2002] a language to allow users to define awareness promoting mechanisms is proposed: however, its implementation does not offer a user-friendly way to construct those mechanisms as required by the EUD approach. On the other hand, also rule-based systems have been widely adopted in an heterogeneous set of applicative domains, but with radically different purposes than the proactive provision of awareness information: the rule-based approach is adopted in expert systems to perform inferences on data in order to derive useful information (e. g., see [Waterman et al., 1986]). For instance, the rule-based approach is widely adopted in the healthcare domain to support clinician in a wide range of activities: from the detection of diseases outbreaks [Wong et al., 2002] to the telemonitoring of inpatients [Seto et al., 2012], from data validation [Supekar et al., 2002] and data analysis [Sun et al., 2010] up to automatic adaptation of the care workflow [Müller and Rahm, 1999]. Nevertheless, adding new rules in rule-based systems is not a task that end-users can autonomously perform, since these operations still require a certain degree of IT skills to be performed. On the other hand, as discussed in Chapter 2, software engineers have high level IT skills, but they lack in having the full understanding of the jargon through which domain-expert end-users express their needs and requirements. Thus, the task to add new rules, as well as to modify or delete existing ones, requires to be performed by another kind of stakeholder of rule-based systems, i. e., the knowledge engineer, which gathers the new requirements from end-users and updates the set of rules accordingly. In this way, the adaptation process requires a certain amount of time, since, once the knowledge engineer gathered end-users' requirements and developed the related set of rules, the outcome of her work must be validated by end-users before being applied to the rule-based system [Bultman et al., 2000].

The first section of this chapter will deal with a concise review of some relevant approaches that have been proposed to cope with the need of tailorability of the EPRs. Subsequently, according to the purposes of the work presented in this thesis, the focus of this chapter will be restricted to those solutions that are related to the open issues of the WOAD framework that have been described in Chapter 4, i. e., the visual composition of documents and rules respectively. To this aim, the second section of the chapter is composed of three subsections. The first subsection will deal with a set of existing approaches that allow the visual composition of documents. The second subsection will focus on the description of a set of solutions that allow the visual composition

of rules, although these solutions are aimed at supporting knowledge engineers rather than end-users. Nevertheless, since one of the goals of this thesis is to make end-users autonomous in defining the rules that they need through the use of a visual metaphor, the last subsection of the chapter will go through the description of a set of purely visual programming languages.

5.1. The Tailorability of the Electronic Patient Records

Within the healthcare domain the need to make the definition of the user interfaces of an EPR more flexible, typically more or less structured forms, so that these can be tailored in order to better meet the local needs of each single group of clinicians has been recognized long ago and confirmed in a number of recent field studies (e. g., see [Mamlin et al., 2006; Morrison and Blackwell, 2009; Chen and Akay, 2011]). Nevertheless, despite this recognition, the tailoring activities that are necessary to reach this goal still require that IT professionals work together with clinicians in order to be able to perform the due customizations in a timely and aptly manner: in other words, clinicians can not autonomously tailor their EPRs.

Mamlin [Mamlin et al., 2006], for instance, proposed *OpenMRS*, an open-source, modular solution to allow the implementation of EPRs that are flexible with respect to the fact that they can be tailored to meet the specific needs of different healthcare institutions. OpenMRS relies on a fixed patient-centric data model that is compliant with the *HL7 standard* [Dolin et al., 2006, p. 7]. In this project, clinical records can be defined using the template engine of the Velocity project[1]. Moreover, OpenMRS encompasses both a decision support module employing the Arden syntax[2] and a compliant rule builder module. On the other hand, OpenMRS adopts a data-driven approach in order to perform data entry operations: a form is a collection of data pointers; this allows for the creation of flexible data entry forms, which do not require any programming activity to be created.

In [Morrison and Blackwell, 2009] two commercially available EPRs that provide clinicians with advanced customization features are described: i. e., *Centricity Electronic Medical Record*[3] and *MetaVision*[4]. The customizability of Centricity concerns the possibility of clinicians to generate their unstructured textual reports by selecting the contents that they need through menus and checkboxes. Moreover, Centricity allows

[1] `http://velocity.apache.org/` [2] `http://www.hl7.org/special/Committees/arden/index.cmf` [3] `http://www3.gehealthcare.com/` [4] `http://www.idm-soft.com/`

clinicians to customize both terminology and boilerplate sections of their reports through a "medical specification language"; yet this approach requires them to learn general purpose programming skills. MetaVision allows for the definition of customized forms and operations, by means of an ad-hoc scripting language. In this light, performing MetaVision customizations requires that clinicians work in collaboration with IT professionals.

A more lightweight approach to EPRs customizability is proposed in [Chen and Akay, 2011]. The proposed approach is to adopt FileMaker[5] to develop flexible EPRs for small- and medium-size clinical settings. Such an approach leverages the FileMaker's capability of generating databases on the basis of the composition of intuitive form-like *Graphical User Interfaces (GUIs)* through simple drag'n'drop interactions. In this project, EPRs can be dynamically updated to meet the constantly evolving needs of clinicians with little effort as modifications performed at interface level are seamlessly reflected in the underlying data structures. Moreover, the authors corroborate their idea by arguing that FileMaker has already been extensively adopted in a number of Japanese hospitals to develop flexible frontends for their institutional EPRs. Nevertheless, even if FileMaker provides its users with a user-friendly graphical interface, deep customizations, like those related to active behaviors to attach to the defined interfaces, still require the involvement of IT professionals to be accomplished and deployed safely.

5.2. A Review of State of the Art of Visual Editors

The idea to switch from complex, text-based editing interfaces to the ones that follow a *What You See Is What You Get (WYSIWYG)* approach dates back to the last quarter of the past century (e. g., see [Lampson, 1978]). Given the benefits that this approach leads to end-users in the use of their software systems, in less than a decade, researcher started to conceive a similar idea to make easier tools to write software source code, i. e., switching from text-based programming languages to new ones that adopt a visual approach (e. g., see Chang [1987]; Pau and Olason [1991]). Even if WYSIWYG solutions was born with the aim to display digital documents (e. g., letters and articles that usually users typeset using word processors) as they would have been at printing time, during the years visual editing solutions have been progressively applied to an increasingly wider and heterogeneous set of applicative domains, including rule programming.

[5] `http://www.filemaker.com/`

5.2.1. Document Editors

In recent years, visual document editing solutions gained a constantly increasing attention by both commercial software producers and IT researchers. This interest has led to the proliferation of heterogeneous solutions, ranging from general-purpose solutions to the most strictly domain-related ones (a concise, cursory comparison can be seen in Table 5.1).

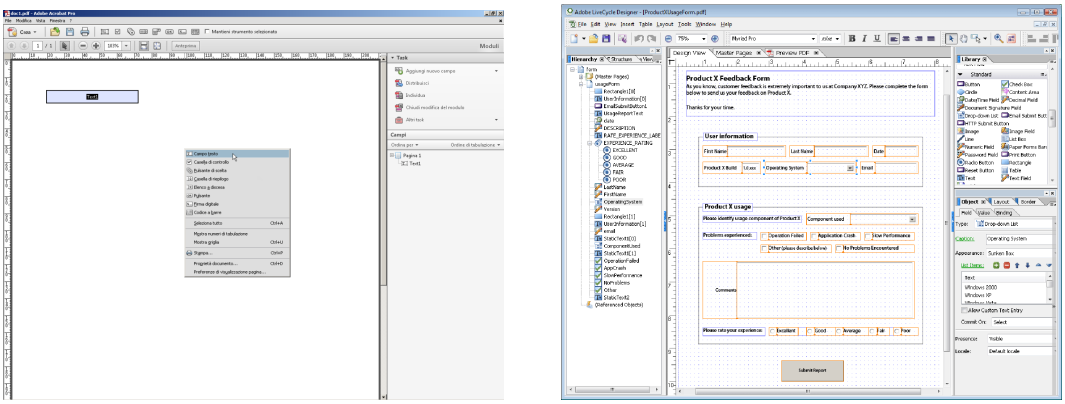
Walking through the commercial solutions landscape, it is possible to see how the set of tools that allow to edit documents in a visual manner has grown over time, and how user-friendliness and richness of features of these tools have reached an impressive level, establishing the *standard de facto* for this kind of tools. Even performing a cursory analysis, it is possible to identify some features that characterize the approaches adopted by these software solutions: (i) the adoption of a strong paper-based document metaphor rather than organizing documents using "simple" forms, (ii) the use of desktop technologies and standards rather than those belonging to the world wide web (even if technology and market evolution are making this separation increasingly blurred), and (iii) the need to interact with external data services rather than to be self-contained entities.

Adobe Systems Inc. provides a set of distinct commercial products, for different usage areas, which allow end-users to create and edit their form-based documents. Acrobat is the professional, market reference solution that allows desktop users to fully manage *PDF forms*[6]. Users can create their forms directly inside the digital representation of a paper-based document (see Figure 5.1a) simply picking the desired form field control and dragging it to the preferred position. Entered data can be either sent to an external data service or stored directly inside the PDF file. On the other hand, LiveCycle is the Adobe enterprise business oriented platform that includes LiveCycle Designer[7] a tool that allows users, through a WYSIWYG environment (see Figure 5.1b), to create and modify their own form document templates[8], which can be converted at runtime in a variety of formats (e. g. PDF forms, HTML forms or Flash applications). LiveCycle Designer allows to define reusable *form fragments*[9], with the aim to avoid the need to create every form from scratch. This LiveCycle Designer feature is quite similar to the component-based approach adopted to define WOAD document templates (see Section 3.2). Finally, the pair Dreamweaver[10] and ColdFusion[11] allows users to build their

[6] According to their standard revision, PDFs can handle two mutually incompatible forms format: AcroForms (the original PDF form standard introduced in the 1.2 format revision) and XFA (since PDF 1.5 format revision; `http://partners.adobe.com/public/developer/xml/index_arch.html`). [7] `http://www.adobe.com/products/livecycle/designer/`. [8] `http://www.adobe.com/products/livecycle/forms/` [9] `http://www.adobe.com/products/livecycle/designer/capabilities/` [10] `http://www.adobe.com/products/dreamweaver.html` [11] `http://www.adobe.com/products/coldfusion/`

interactive web pages, containing the needed forms (either in HTML or PDF format), using a graphic environment and without the need to code anything, but this goal is mainly achieved through the massive use of option dialog boxes.
OpenOffice.org[12], and consequently the family of office automation tools which rely on the *Open Document Format (ODF)*[13], provide end-users with the support for the *XForms* technology[14]. Users can easily and quickly create and edit their document-centric form-based applications, just putting form elements directly inside word processor documents, picking them up from a palette and positioning them through drag'n'drop actions. Document created in this way can be filled in with data using the word processor application, but the data that users enter in the form fields has to be stored into an external data structure (e. g., into a XML document or sending it to a remote data storage service). Similarly, *IBM Lotus Symphony Documents*[15] provides support for paper-like documents coupled with XForms fields, but in addition it provides an easy way to use the *IBM Lotus Forms* technology as storage and data manager backend.
Microsoft Sharepoint Designer[16] (also known as *Microsoft Office SharePoint Designer*) provides users with a WYSIWYG environment that allows them to build and maintain their SharePoint web pages, including those that contain SharePoint-related forms. SharePoint Designer is part of the Microsoft SharePoint infrastructure and actually provides a way to build end-user front-ends for those kind of web applications. During editing activities, SharePoint Designer inserts *Extensible Stylesheet Language (XSL)*[17] code directly within the HTML code, without requiring users to learn it.
FileMaker Pro[18] is a user-friendly, easy to use relational database application with a database engine that is strictly integrated with its GUI. This allows end-users to create and modify their database schemas just designing the related forms (e. g., screen, layouts and reports), which are subsequently used to access and display data. The design phase consists in dragging elements that the user desires into forms, and in so doing the database engine will automatically handle them in the right way.
Finally, *SAP NetWeaver Visual Composer*[19] is a web-based visual modelling tool, which

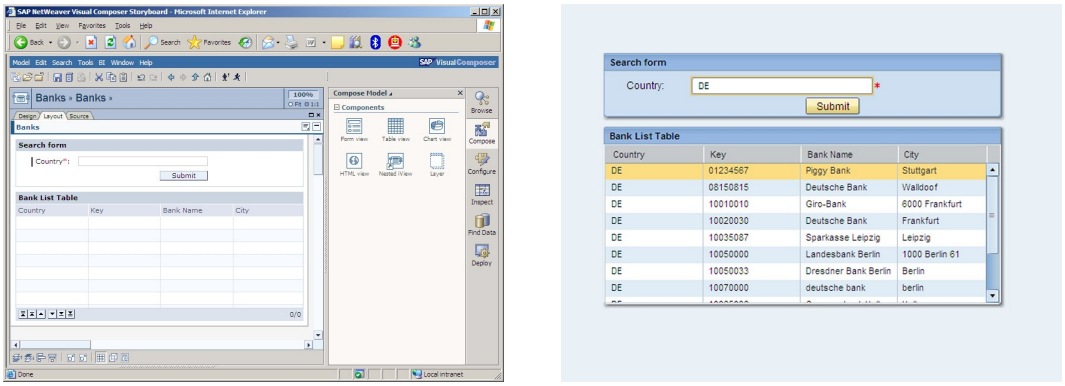
[12] `http://www.openoffice.org/`
[13] `http://www.odfalliance.org/`
[14] See `http://www.w3.org/TR/xforms11/` and `http://opendocument.xml.org/node/192`
[15] *IBM Lotus Symphony* (`http://symphony.lotus.com/`) is an office automation suite distributed by IBM's Lotus Software division that supports *Open Document Format (ODF)* an integrates some pieces of OpenOffice.org source code.
[16] `http://sharepoint.microsoft.com/en-us/product/related-technologies/pages/sharepoint-designer.aspx` [17] `http://www.w3.org/Style/XSL/` [18] `http://www.filemaker.com/` [19] SAP AG (`http://www.sap.com/`).



(a) The Acrobat form designer

(b) The LiveCycle Designer

Figure 5.1.: Adobe form design solutions



(a) Designing form view layout

(b) The designed form view at runtime

Figure 5.2.: The *SAP NetWeaver Visual Composer Layout Board* user interface

is oriented towards business processes. This visual modeler enables users to quickly build their own business application (reusable) components, without the need to code anything and in a graphical manner (see Figure 5.2). Users can select the views to be used to display data requests and results (e. g., forms, tables or charts), and connect them with the required data sources, just drawing connection links.

Enlarging the view to look towards the scientific research landscape, it is possible to identify a set of relevant contributions related to the visual composition of document templates (which often are conceived as user interfaces [Boyer, 2008]).

Visual TDL Document Editor (VTDE) [Yamazaki et al., 2000] is a document editor that is oriented towards the healthcare domain and its aim is to create and edit *Electronic Patient Record (EPR)* templates. In particular, VTDE manages only a specific *eXtensible*

Markup Language (XML) dialect, i. e., the *Template Definition Language* (TDL)[20], which has been conceived and developed to reach the goal of promoting EPR templates sharing among clinicians and institutions. The VTDE user interface is composed of two distinct windows: a text editor and a graphic preview. Users can define their EPR templates by entering TDL constructs, i. e., tags, in the text editor window and, at the same time, they can immediately see the results of their editing operations, which are directly rendered within the graphic preview window. Despite the presence of the 'visual' word in its name, VTDE is not really a visual editing solution, and this is due to the fact that the only graphical help that it provides to users comes from the preview window: the editing operations still imply that users should define document templates directly typing the TDL code. Moreover, authors report that VTDE completely lacks in supporting users in the definition of any kind of rule to dynamically change templates. The only way they propose to achieve this goal is to describe rules using TDL code comments: starting from these comments, other users or software developers could implement the described rules. Finally, even if it has been conceived with sharing purposes, with respect to WOAD documents (see Section 3.2), VTDE seems to not support any kind of content sharing among documents.

AgentFlow System [Chen and Wang, 2001], which is the basis of the commercial Internet workflow environment provided by the FlowRing company[21], provides users with its *FormDesigner* (see Figure 5.3), which is a graphic development tool that allows to build and modify the application forms. Users, even without programming skills, can edit their forms simply using drag'n'drop to position the desired basic UI components. Moreover, users with some programming skills may autonomously build and plug into the system their own components. In order to do this, FormDesigner provides at least two different methods. The former is based on the idea that users may build components using some user-specified properties with the aim to represent a complex application logic. The latter requires the user to describe the application logic using an intermediate scripting language, simply implementing some programming interfaces and using system objects and functions (e. g., Agenda and Form objects). If the scripting language is still considered an unsatisfying way to reach the desired goals, users can also implement their own native code[22]. The main difference with respect to WOAD relies on the work environment to which these two solutions are addressed. While WOAD is a framework conceived to develop flexible document-based and collaborative applications, FormDesigner is part of

[20] The *Template Definition Language* is used to define both EPR templates content and structure. A TDL description is presented and discussed in [Yamazaki and Satomura, 2000]. [21] `http://flowring.com/` [22] The integration with user-defined native code relies on the *Java Native Interface (JNI)* technology.

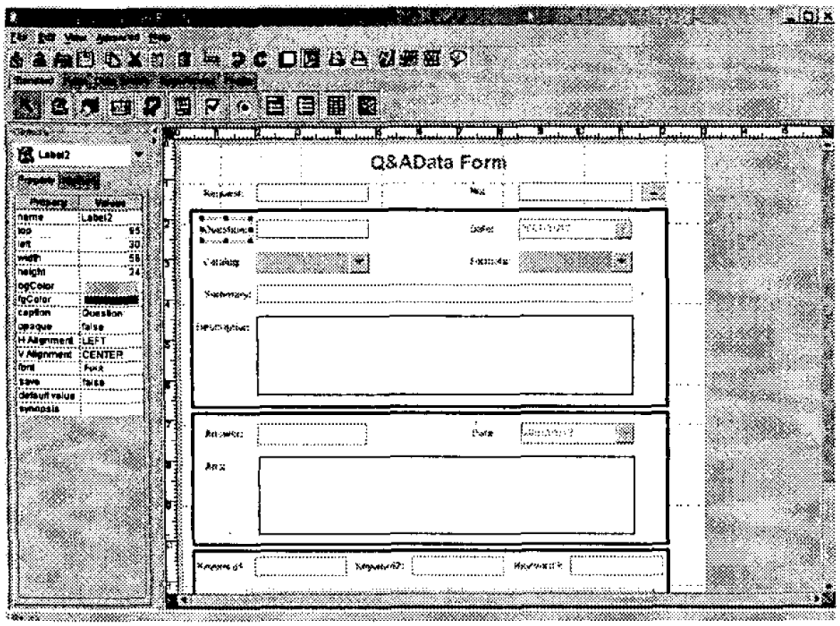


Figure 5.3.: The FormDesigner of AgentFlow System (from [Chen and Wang, 2001])

a WYSIWYG environment that aims to support business application development, and this makes it to all intents and purposes a tool only for developers. Moreover, even if FormDesigner allows to extend its set of components using its visual environment, it does not completely avoid the need for using a traditional programming language in order to define complex application logic.

Despite a lot of WYSIWYG tools populate the landscape of web design, most of them provide users with some graphical helps with the only aim to create static web pages (i. e., documents). *WebSheets* [Wolber et al., 2002] is a solution to allow end-users (e. g., web designers) to create dynamic web pages with the capability to access and modify an underlying database, without imposing the need to enter any kind of programming statements. WebSheets provides a standard WYSIWYG HTML editing environment coupled with a specific *Programming by Example* (PBE) technique, which is called *Query by Example* (QBE): users can enter either some sample tabular data (see Figure 5.4), telling the system to automatically create and map the related database table, or some expressions directly into table cells, setting up a set of conditions to filter or delete some table rows. Moreover, WebSheets allows end-users to insert spreadsheet formulas directly inside the editing environment, in order to perform computations on data. WebSheets allows users to define, through database standard operation and spreadsheet formulas, some interactive behaviors that are triggered by events and depend on document data,

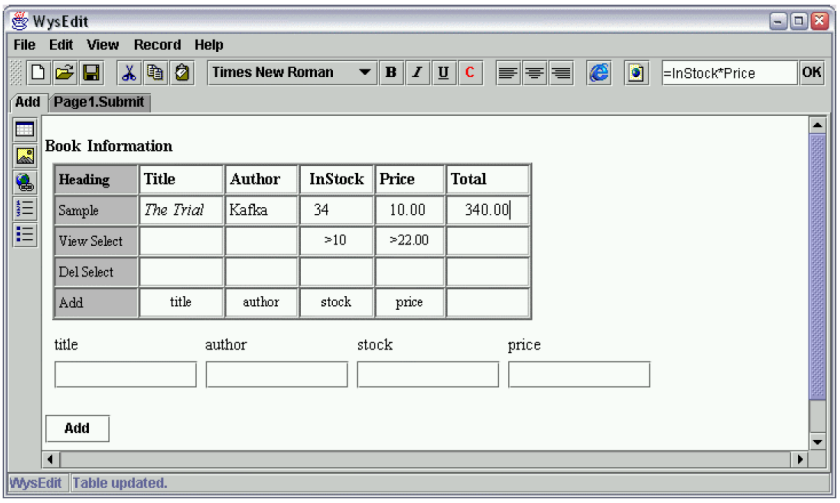
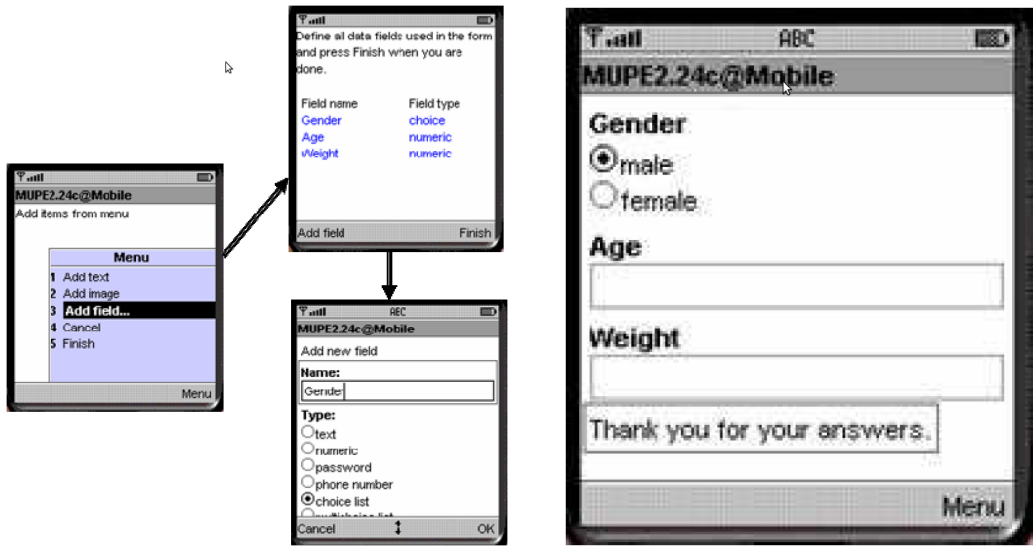


Figure 5.4.: WebSheets Development of a BookStore (from [Wolber et al., 2002])

which can be both entered directly in the page or collected from other pages. To directly enter data through the page, users have just to specify in a specific table row what are the names of input boxes, which at runtime are rendered in the related table. Similarly to what happens in WOAD document templates, WebSheets automatically creates and maps the underlying data structure. On the other hand, WebSheets is constrained to use a database and consequently it is limited in handling data using only a tabular organization. Moreover, like in WOAD, WebSheets allows for data sharing between different pages, although with a very limited granularity degree: it is possible to refer to external pieces of data from the other pages inside the development environment just qualifying the field name with the page name (e. g., `page_name.field_name`).

Mobile Form-Editor [Chande and Koivisto, 2006] is a WYSIWYG solution that allows end-users to easily create their Mobile Forms. This kind of forms is conceived to be conveyed to users through mobile devices (i. e., mobile phones), and consequently the *Mobile Form-Editor* environment has been conceived to be executed on this particular kind of devices. Mobile Form-Editor is a simple graphic wizard with the aim to assist end-users in doing three different, but strictly connected, operations: (i) to define the set of form fields (i. e., field names and types) that the user has to fill in (see Figure 5.5a), (ii) to specify form field attributes (e. g., label and position) and form look-and-feel (see Figure 5.5b), and (iii) to define how to present responses to users once data has been received. Due to the particular kind of device to which Mobile Forms are targeted, Mobile Form-Editor turns out to be radically different with respect to any WOAD framework feature.



(a) Data type definition for From Fields **(b)** Form user interface design

Figure 5.5.: The Mobile Form-Editor wizard (from [Chande and Koivisto, 2006])

Oryx Editor [Decker et al., 2008a] is an web-based visual design environment that has been initially conceived to model business process. Oryx Editor is an open source solution with a plug-in architecture, which makes easier to augment its visual design capabilities (e. g., to include the support for the UML and XForms). Focusing on the design of XForms documents, the editor allows end-users to drag XForms component from a palette and place them inside a drawing area, which mimics the final aspect of the form. Moreover, with the same interaction mode, users can add to these components the desired XForms interactive features (e. g., alert messages and actions). On the other hand, users can define the underlying XML data structure, but this operation needs to be done through a property panel, which allows users to type the related *XPath*[23] syntax into a simple textual input field. Moreover, Oryx Editor encompasses only the visual editing support for XForms, leaving the task to use them through the software solution that users prefer. In order to do this, once users complete the design of their forms, they have to export their models, translating them from the Oryx Editor internal representation to the XForms standardized syntax (see Section 7.1.1 for further details).

Data-Interactive DOcument (Dido) Karger et al. [2009] has been conceived to be a self-contained single web page application, which also embeds an application development environment. Dido is an *active document* (see Section 3.1) and allows end-users to

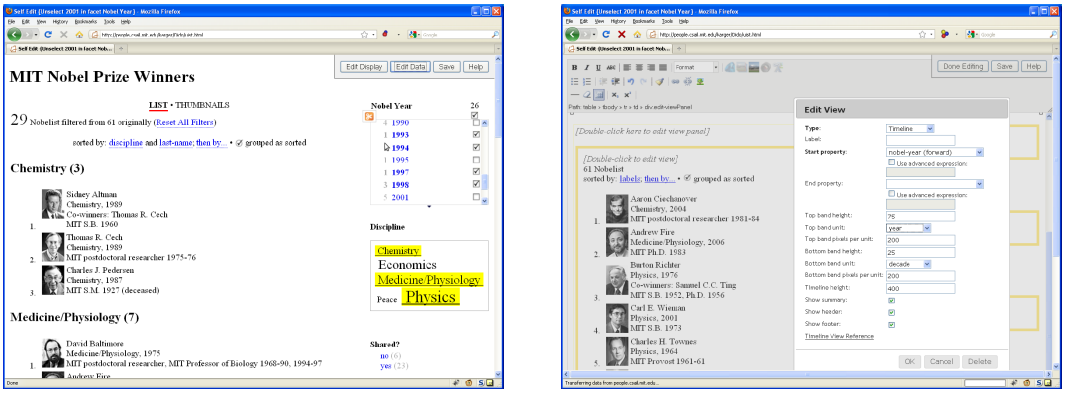
[23] `http://www.w3.org/TR/xpath/`

author (not program) applications that meet their specific information management needs. Dido's target are CRUD (i. e., Create, Read/visualize, Update, and Delete) applications[24]. Moreover, Dido implements an in-document data store system that makes unnecessary the use of storage servers, like database servers. In this light, Dido is aimed at focusing end-users on editing (not coding) operations and, in order to reach this goal, it encompasses (i) a data editor, which allows users to define their own rich and structured data items, (ii) an interactive tool to visualize (see Figure 5.6a) and edit data, directly within the page, and (iii) a "metaeditor" that allows users to modify the visualizer/editor (see Figure 5.6b) in a WYSIWYG manner. Dido uses three different structures to build the page: (a) the *lens*[25], which is a template that is used to define how a data item has to be rendered (i. e., a HTML fragment that specifies arrangement and appearance of "data slots" to be filled with contents), (b) the *facet*, namely a widget with the aim to filter data items according to the value of some specific property, and (c) the *view* that shows (filtered) data in some way according to both lens and facet. When users load a Dido page, the page is presented in visualization mode, and users can enter data just clicking the "Edit Data" or the "New Item" buttons, in order to modify existing data or create new data items respectively. Using the "Display Editor" button users switch Dido to the meta-editing mode, which invokes a WYSIWYG editor (i. e., an extended version of the TinyMCE[26] editor) on the entire web page, and they can customize the page by adding static contents, formatting some page portions, specifying the layout and configuring views and facets (e. g., using the appropriate configuration dialog box the view can be switched from a list to a timeline). Finally, "Lens Editor" allows users to customize lens, using the above mentioned WYSIWYG editor to shape them. As mentioned, similarly to WOAD, Dido is based on the concept of active document. In both approaches, documents can be augmented with active behaviors by end-users using a WYSIWYG environment and are triggered by data changes and users' interactions. On the other hand, unlike WOAD, Dido documents seem to be not able to share any piece of data among each others.

5.2.2. Rule Editors

In recent years rule-based system earned increasing popularity in business applications, and consequently the number of visual rule editors is increasing. Nevertheless, unlike what

[24] The CRUD approach is the basis of most *Content Management Systems (CMS)*, e. g., Drupal, Microsoft Sharepoint, and Semantic Mediawiki. [25] Lens-based architecture is the direct consequence of using the *Exhibit* framework [Huynh et al., 2007]. [26] `http://tinymce.moxiecode.com/`



(a) Visualizing data in Dido **(b)** Editing the Visualization to add a timeline

Figure 5.6.: The Dido UI (from Karger et al. [2009])

happens in the WOAD framework, usually these rule-based systems are not associated to the user interfaces through which users fill in and inquire data (i. e., documents); rather, these visual editors are aimed at supporting users in the definition of traditional business rules for their applications. Similarly to what regards document visual editors, undertaking a cursory analysis of the commercial landscape, it is possible to identify those software solutions that established the *standard de facto* for editing rules in a visual manner. On the other hand, both commercial and scientific research landscapes include visual rule editors that ranges from the more general-purpose to the more domain-related ones (a concise, cursory comparison can be seen in Table 5.2).

SAP NetWeaver Business Rules Management[27] encompasses the *Rules Composer*. Rules Composer is part of *SAP NetWeaver Developer Studio*, which is an Eclipse-based visual editing environment (see Figure 5.7), and allows users to build and maintain rules through a rich set of user-friendly rule representation formats (e. g., decision tables).
JBoss provides a visual rule editor bundled in the *Drools Guvnor*[28] suite, which is a centralized, web-based repository for Drools knowledge bases. The provided rule editor allows users to define their own Drools rules, through an user interface that is pretty like to a wizard: users can enqueue rule conditions and actions simply acting on a button (see the plus green icon in Figure 5.8).
Bosch proposes the *Visual Rules Suite*[29] that encompasses two different rule editors. The former is called *Visual Rules Modeler* and is an Eclipse-based solution that allows users to pick up rule components from a palette and drop them into the rule flow (see

[27] SAP AG (`http://www.sap.com/`). [28] `http://www.jboss.org/drools/drools-guvnor.html` [29] `http://www.visual-rules.com/business-rules-management-software-rules-engine.html`

	Type			Aged [a]	Features					
	Commercial	Open Source	Research		Std. Technologies [b]	Drag'n'Drop	Paper Metaphor	Autom. Mapping [c]	Formulas	Fragments [d]
Adobe Acrobat	×	-	-	n/a	×	×	×	-	-	-
Adobe LiveCycle	×	-	-	n/a	×	×	×	-	-	×
Adobe Dreamweaver (with ColdFusion)	×	-	-	n/a	×	×	-	-	-	-
AgentFlow FormDesigner [Chen and Wang, 2001]	× [e]	-	×	×	×	×	-	-	-	-
Data-Interactive DOcument Karger et al. [2009]	-	×	×	-	×	-	-	×	-	-
FileMaker Pro	×	-	-	n/a	-	×	-	×	-	-
IBM Lotus Symphony Documents	- [f]	-	-	n/a	×	×	×	-	-	-
Microsoft SharePoint Designer	-	-	-	n/a	×	×	-	-	-	-
Mobile Form-Editor [Chande and Koivisto, 2006]	-	-	×	-	×	-	-	-	-	-
OpenOffice.org	-	×	-	n/a	×	×	×	-	-	-
Oryx Editor [Decker et al., 2008a]	-	×	×	-	×	×	-	-	-	-
SAP NetWeaver Visual Composer	×	-	-	n/a	-	×	-	-	-	-
Visual TDL Document Editor [Yamazaki et al., 2000]	-	-	×	×	-	-	-	×	-	-
WebSheets [Wolber et al., 2002]	-	-	×	×	×	-	-	×	×	-

[a] Research project older than five years. [b] Based either on open or proprietary standardized technology.
[c] Underlying data structures are automatically mapped with the related visual component.
[d] Reusable components that can be shared between different documents.
[e] AgentFlow FormDesigner is the basis of a commercial software produced by FlowRing. [f] Freeware license.

Table 5.1.: Summary of the analyzed documents visual editing solutions

Figure 5.9). The latter is the web-based counterpart of the desktop editor and is called *Visual Rules WebModeler*. Despite the adoption of a visual user interface, all these rule editing solutions are still strongly tied to be used by IT professionals (i. e., software analysts or developers).

The *Oryx Knowledge Base Editor*[30] has been conceived and developed within *Mandarax Project*[31]. It encompasses a visual rule editor (see Figure 5.10), which allows end-users to define their own set of derivation rules. In this project, rules are composed of a set of *prerequisites* (i. e., *conditions*) and a *conclusion* (e. g., either logic expressions or actions), which is a rule structure similar to how WOAD mechanisms are structured.

[30] *Oryx Knowledge Base Editor* is not part of the same project in which the *Oryx Editor* (see Section 5.2.1) has been developed: this is only a case of homonymy. See `http://oryx.sourceforge.net/`
[31] Mandarax is a project with the aim to integrate derivation rules into Java (see `http://mandarax.sourceforge.net/`).

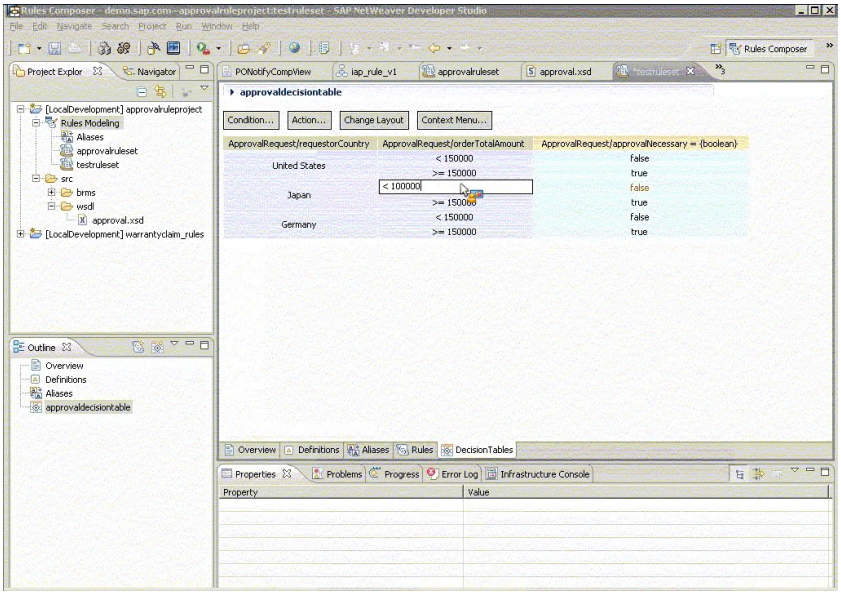


Figure 5.7.: SAP NetWeaver BRM Rules Composer

End-users add rules to the knowledge base using a simple wizard that allows them to specify both prerequisites and conclusions in the same window. Rules can contain both simple logic predicates, complex functions and expression in a formal natural language, which end-users can define according to the needs related to their specific application domain.

Finally, a very niche solution have been conceived at the *N.A.S.A. Work Systems Design & Evaluation Group* of the *Intelligent Systems Division*, which developed the *OCAMS Rule Editor*[32] on the basis of OCA officers work practices. OCAMS Rule Editor has been conceived as an engineering system with the aim to be easily adapted to the changing requirements while minimizing maintenance costs. OCAMS Rule Editor allows *Orbital Communications Adapter Monitoring System* (OCAMS) officers to manage decision-making rules that specify which files (e. g., schedules, procedures, commands, e-mail, photographs and news) to mirror and archive: officers can easily view deployed rules, seeing which rule is related to a specific mirroring or archiving decision, and manage rules as requirements change. In compliance to one of the basic EUD tenets, this tool is aimed at supporting OCAMS officers to adapt the system without requiring any intervention by the OCAMS developers to change source code.

Like in the case of the visual editors of documents, the scientific research landscape, as well as the commercial one, is populated by a set of relevant contributions concerning

[32] `http://ti.arc.nasa.gov/news/ocams-rule-editor/`

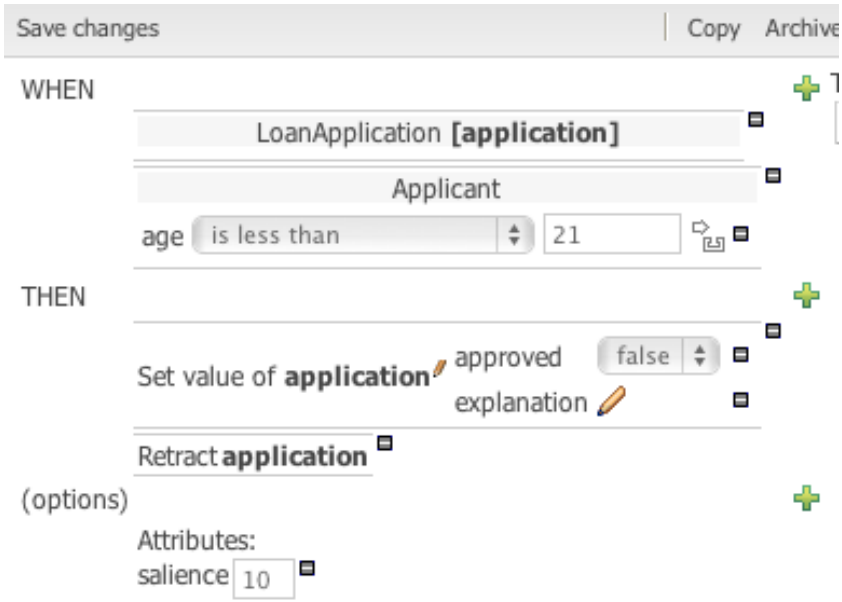


Figure 5.8.: JBoss Drools Guvnor visual rule editor

the visual composition of rules.

Andersen Consulting[33] developed *Eagle* [Davidowitz, 1996], which is a set of integrated tools, architectures and reusable components with the aim to allow end-users to externalize business object behaviors. This goal is achieved through the definition of rules that rely on the Smalltalk language. End-users are supported in creating their rules through a "point-and-click" user interface, which allows them to visually check the correctness of their rules by building and displaying the related abstract syntax tree. On the other hand, similarly to what happens in Visual TDL Document Editor (see Section 5.2.1), end-users have to type the rule executable code into a textual editor, like in any traditional development environment, using the traditional Smalltalk language constructs.

Restricting the point of view to the healthcare domain, [Chen et al., 2002] describes a rule-based real-time alerting system, which has been tailored to meet the needs of an *Intensive Care Unit* (ICU). This alerting system has been designed to rely on the database of a clinical information system and to convey notices to clinicians through mobile phones or pagers. The conditions of rules involve some physiological parameters and must comply with a simple, intuitive, and easy to use grammar. This grammar allows clinicians, i. e., system end-users, to define two kinds of conditions that are called *basic rules*, i. e., some simple logic expressions, and *advanced rules*, i. e., time bounds, parameter

[33] `http://www.ac.com/`

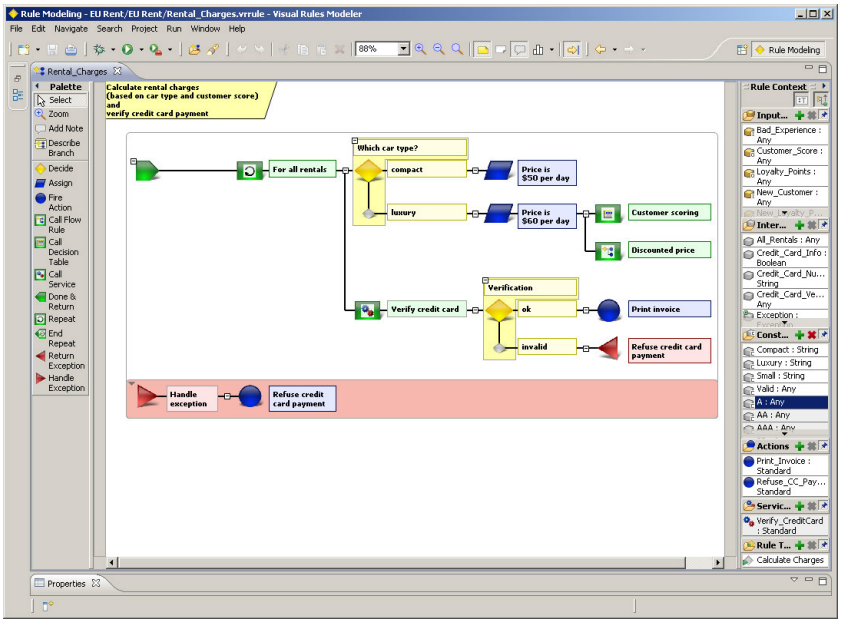


Figure 5.9.: Bosch Visual Rule Modeler

changes, or any valid combination of any kind of rule. Rules can be defined using the graphic editor, which lists physiological parameters and provides a set of buttons that specify the allowed conditional constructs (see Figure 5.11). On the other hand, the text message to be conveyed can be only entered through an input dialog box, which requires to specify the clinician who will be alerted, a priority value and, obviously, the text of the alert message. Similarly to Eagle Rule Editor, the alerting rules editor displays to users the code of rule conditions in its textual representation. The user interface does not provide any kind of drag'n'drop interaction; thus, users interact with the editor in a way that is more similar to a wizard.

MARBLS [Krebs et al., 2012], i. e., *Medical Alert Rule BuiLding System*, is an end-user programming environment that allow to design and test clinical alert rules in a visual manner. MARBLS user interface encompasses two relevant components (see Figure 5.12), i. e., the rule workspace and the query explorer. MARBLS allows end-users to define rules in two ways: by (i) composing visual blocks in the rule workspace; and by (ii) acting directly on the charts in the query explorer (to update rules in the rule workspace). The MARBLS visual language relies on the MIT OpenBlocks library [Roque, 2007] that will be described in the Section 7.2, since it has been adopted to develop the WOAD Mechanism Editor.

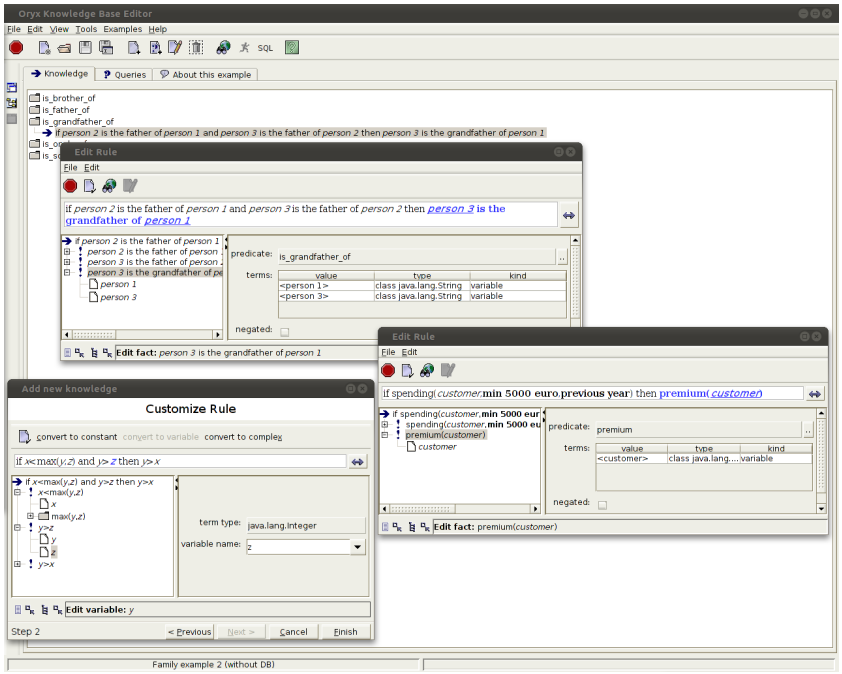


Figure 5.10.: The Oryx Knowledge Base Editor

The *DRREd Rule Editor* [Bassiliades et al., 2005] is a Java-based RuleML[34] visual editor, which is part of a visual development environment that allows end-users to define defeasible logic[35] rule bases over ontologies. The goal of this editing solution is to provide end-users with a tool that enhances user-friendliness and efficiency in the development of this kind of rules. Being RuleML a XML dialect, the DRREd Rule Editor has been conceived to follow the hierarchical structure of this kind of meta-languages: the left area of the user interface displays the entire rule organized in a tree, while in the right area users can manage the attributes related to the selected node in the left tree. Moreover, users can add new rule elements interacting directly with the tree representation in the left area, according to the constraints imposed by either the *Document Type Definition (DTD)* or the *XML Schema Definition (XSD)*[36] that defines the RuleML grammar. Visual editing is achieved only through the use of menus and dialog boxes, which allow end-users to interact with the XML syntax tree; the editor does not support any kind of drag'n'drop interaction.

[34] `http://ruleml.org/`

[35] For more details about defeasible logic, see [Nute, 1994; Antoniou et al., 2001] and `http://defeasible.org/`.

[36] During the project evolution, RuleML migrates from *DTD* to *XSD* for a better specification of language properties.

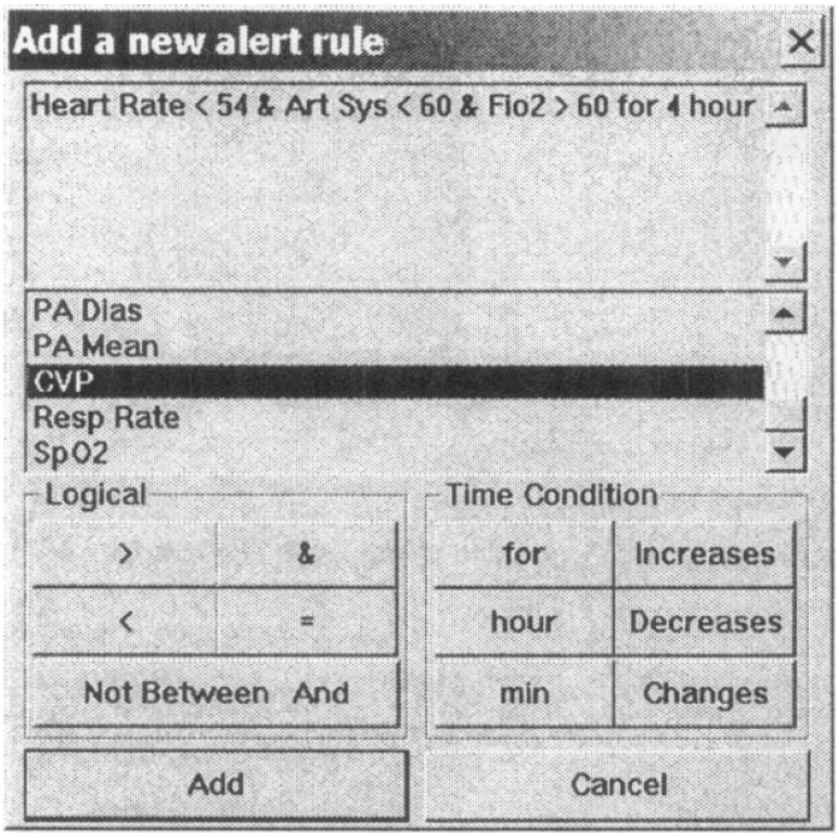


Figure 5.11.: The Real-Time Clinical Alerting System rules visual editor (from [Chen et al., 2002])

In [Li et al., 2010], authors present a visual editor that allows to create rules with the aim of optimizing airplane load plannings. The system allows users to define rule conditions expressing them using the formalized *Object Constraint Language* (OCL), which is a standard object relationship description language. The editor uses graphic elements (e. g., condition, action and flow) to build rules. Users have just to pick up those elements from a palette and dropping them into the model, and they compose the rule flow creating connections between the elements that have been previously placed in the model. This rule editor adopts a different approach to define rules. While most part of rule editors keep conditions separated from actions, grouping them into two distinct and homogeneous sets, this editor allows users to build rules as a flow, which contains an intertwined set of both conditions and actions.

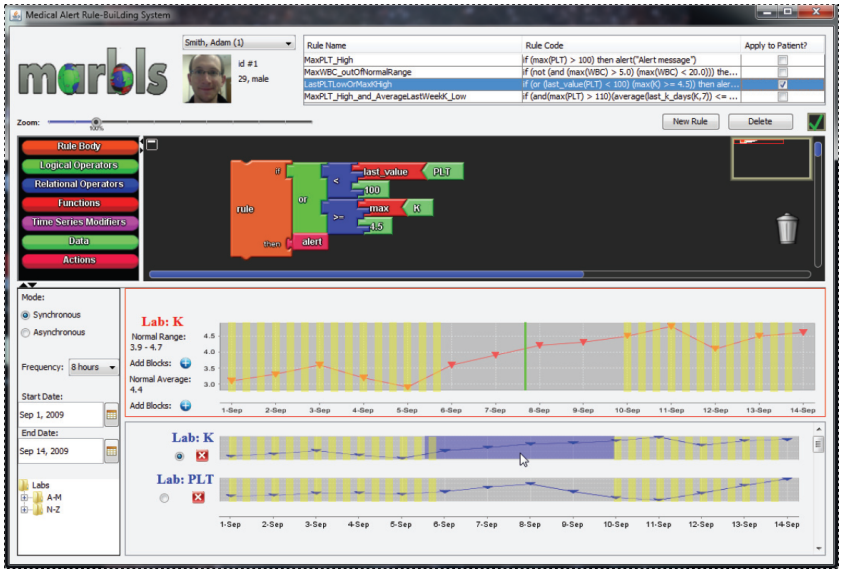


Figure 5.12.: The MARBLS visual rule editor (from [Krebs et al., 2012])

5.2.3. Visual Languages

To the increase of adoption of visual editing (e. g., see the two previous sections) solutions corresponds a similar increase in the proliferation and widespread use of visual programming languages. Even performing a cursory analysis, it is possible to see how this kind of languages can be adopted in a wide set of heterogeneous domains, ranging from general-purpose software development (e. g., Fabrik [Ingalls et al., 1988] and its "successor", i. e., Lively Fabrik) to multimedia compositions (e. g., the case of *vvvv*[37] and *Buzz Machines*[38]) and embedded software (e. g., Figure 5.15 shows a Minibloq[39], a visual language to visually generate Arduino[40] sketches). Nevertheless, most visual languages have been conceived and developed by researchers and companies in the educational field (e. g., see [Maloney et al., 2010; Repenning et al., 2011; Stolee and Fristoe, 2011]).

AgentSheets [Repenning et al., 2011] is an educational visual language with the aim to teach young students IT concepts and skills, in particular computer programming, allowing them to create their own web-based games. Basically, AgentSheets provides a drag'n'drop programming interface that allows users to fill in a computational grid, which looks like a spreadsheet, with numbers and strings, but also with programmable agents. Those agents are represented through iconic pictures, and they can be customized

[37] `http://vvvv.org/`
[38] `http://www.buzzmachines.com/`
[39] `http://blog.minibloq.org/`
[40] http://www.arduino.cc/

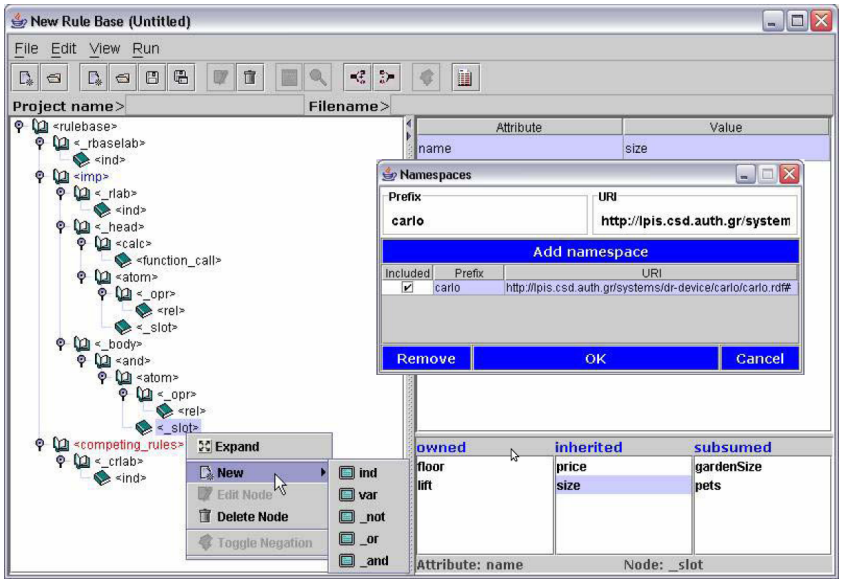


Figure 5.13.: The DDREd Rule Editor (from [Bassiliades et al., 2005])

through a block-based visual language, e. g., to react to mouse and keyboard interactions, to play an animation or to produce sounds.

Another relevant visual language in the education field is *Scratch* [Maloney et al., 2010]. Scratch is an open-source solution that is born at the *MIT Media Lab* with the aim to allow kids, but in general any kind of user, to learn computer programming by creating their own simple games or interactive (and animated) stories. Scratch is based on the *Squeak*[41] programming language and maps its formal constructs (e. g., conditional or loop constructs) with a set of graphic building block, which the user can arbitrarily compose to define the game or the story application logic. *Microsoft Kodu* [Stolee and Fristoe, 2011] represents a third example of visual language that is aimed at helping its users (i. e., kids) to learn computer programming concepts and skills. Kodu allows its users to create their own *Microsoft Xbox* video games. Through the Kodu developing environment, users can define both the game world and the game characters. More in deep, Kodu is an interpreted, rule-based language in which rules are simple "when-do" constructs. Rules can be grouped by pages. Pages are constructs that can be evaluated atomically, and they represent the "place" where users define the application logic that controls their video game characters.

In recent times, due to the diffusion of increasingly powerful smart mobile devices, end-users felt the need to autonomously develop their own mobile applications. This need

[41] *Squeak* is a *Smalltalk* implementation (`http://www.squeak.org/`).

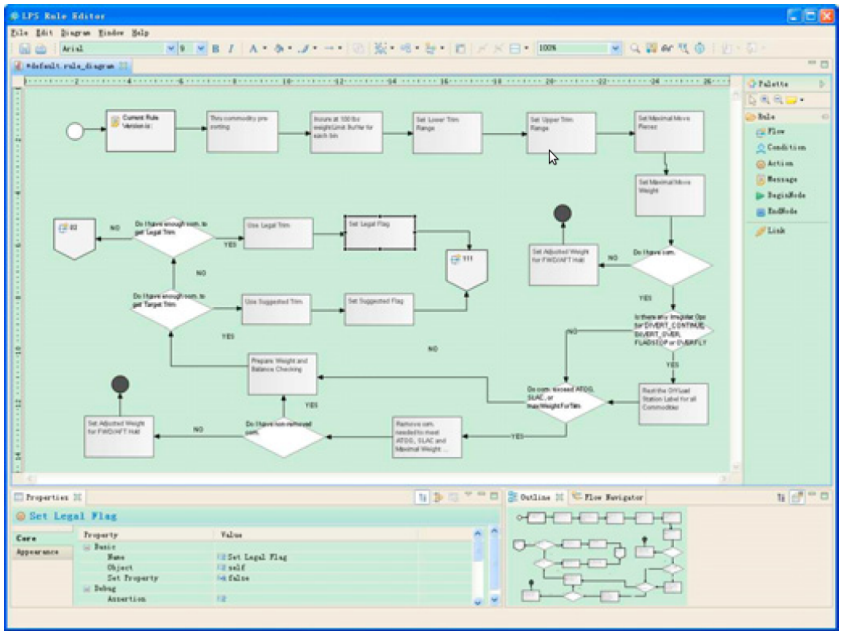


Figure 5.14.: The Airline Load Planning System rules visual editor (from [Li et al., 2010])

has aroused the interest of researchers for the adoption of visual languages with the aim of developing mobile applications directly through such kind of devices. Danado e Paternò, for instance, proposed Puzzle [Danado and Paternò, 2012], which is a visual language that relies on the "jigsaw" metaphor, i. e., the language constructs are represented through composable "jigsaw pieces". Puzzle allows the users to autonomously create their own applications directly on their mobile devices.

Enlarging the view to the general-purpose languages, it is possible to see how visual languages had been adopted in a heterogeneous set of application domains. *Yahoo Pipes*[42], for instance, is a free of charge web-based solution that is aimed at empowering end-users in creating their own views of some raw data (*mashups*), using a metaphor that is inspired by UNIX pipes. On the other hand, *LabVIEW*[43] is a proprietary, commercial graphic development and execution environment for the language that is called "G", which can be used to create applications, for instance, in the fields of industrial automation and data acquisition.

SourceBinder [Conradi et al., 2010] is an example of general-purpose visual language that empowers end-users to develop their own *Adobe Flash* applications. SourceBinder is based on web standard technologies, and it maps most part of standard *ActionScript*

[42] `http://pipes.yahoo.com/pipes/`

[43] `http://www.ni.com/labview/`

	Type			Aged [a]	Features		
	Commercial	Open Source	Research		Drag'n'Drop	Document-related	Need Coding
Airline Load Planning System [Li et al., 2010]	-	-	×	-	×	-	-
Bosch Visual Rules Modeler	×	n/a	-	n/a	×	-	-
Bosch Visual Rules WebModeler	×	n/a	-	n/a	×	-	-
DRREd [Bassiliades et al., 2005]	-	-	×	×	n/a	-	-
Eagle Rule Editor [Davidowitz, 1996]	-	-	×	×	-	-	×
JBoss Drools Guvnor	-	×	-	n/a	-	-	-
MARBLS [Krebs et al., 2012]	-	-	×	-	×	-	-
OCAMS Rule Editor	-	-	-	n/a	-	-	n/a
Oryx Knowledge Base Editor	-	-	-	n/a	-	-	-
Real-Time Clinical Alerting System [Chen et al., 2002]	-	-	×	×	-	-	×
SAP NetWeaver Business Rules Management	×	-	-	n/a	n/a	-	-

[a] Research project older than five years.

Table 5.2.: Summary of the analyzed rules visual editing solutions

classes with the concept of nodes. Nodes can be "wired" to define the control "flow" (see Figure 5.17). Moreover, in order to support different levels of complexity, each node can be managed accordingly: from the simple ability of being only composed with other nodes to the possibility of being customized by editing the node source code.

On the other hand, *Pure Data* [Puckette, 1996] is an open source, visual programming environment with the aim to support end-users in building multimedia applications, focussing on the data-flow rather than on the textual code to reach the same goal. Pure Data applications rely on three basic types of text entities: *atoms*, which are the basic units of data (e. g., float number or symbols), *objects*, which are programming language functions (i. e., both basic operators, like mathematical and logical functions, and both general and specialized functions, e. g., the the Fast Fourier transform), and *messages*, which are composed of atoms and can be used as parameter for objects (e. g., the classic "Hello, world!" string). Users can visually connect entities, and in so doing they creates the "patches"[44] through which users defines the application logic.

Finally, *Lively Fabrik* [Lincke et al., 2009] is a web-based end-user environment, which

[44] The concept of "patch" is a metaphor that comes from world of analog audio synthesizers, which created sound effects according to how their pins were connected with patch cables.

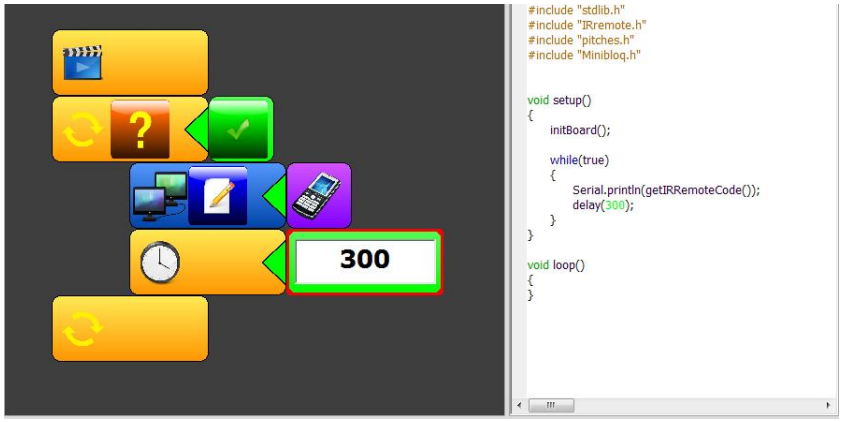


Figure 5.15.: A Minibloq "code" snippet, with the related `C` sketch

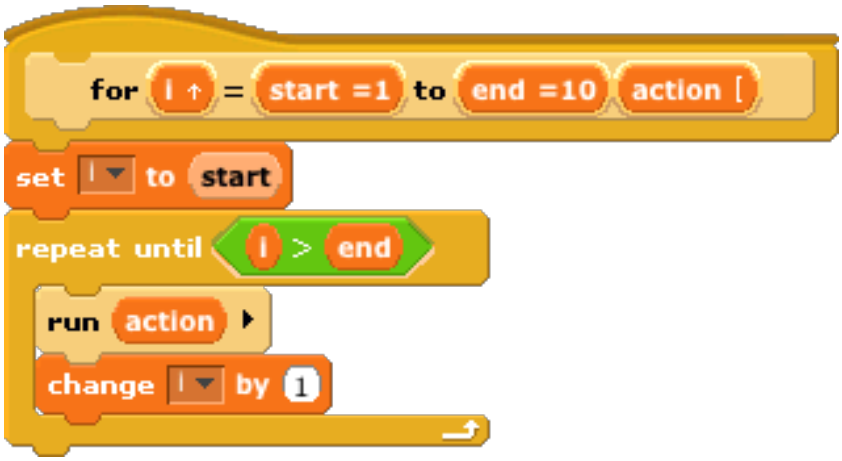


Figure 5.16.: A Scratch "code" snippet

extends the original Fabrik system through the integration with the *Lively Kernel*[45], in order to allow end-users to create their own web applications and mashups directly within their preferred web browser. The integrated development environment provides users with a general purpose visual programming language. Like most visual languages, also Lively Fabrik adopts a graphic building blocks metaphor. Users can build their own applications composing different language blocks, which at this abstraction level can be seen as "black boxes", in particular connecting the input and output *pins* that are exposed by each block (see Figure 5.18). Moreover, similarly as what happens in AgentSheets (described above), users can define or customize the application logic that is endowed in blocks: the application logic of a block is defined through a scripting language, i. e., the

[45] `http://www.lively-kernel.org/lively/index.html`

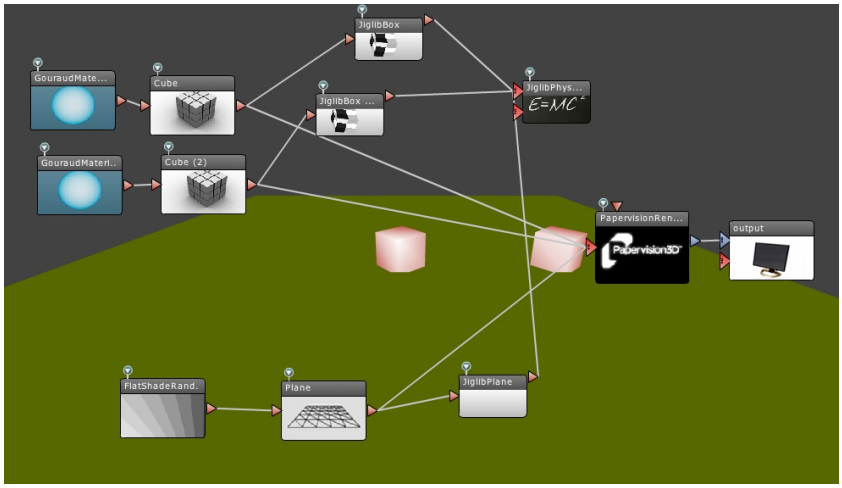


Figure 5.17.: A SourceBinder visual language example

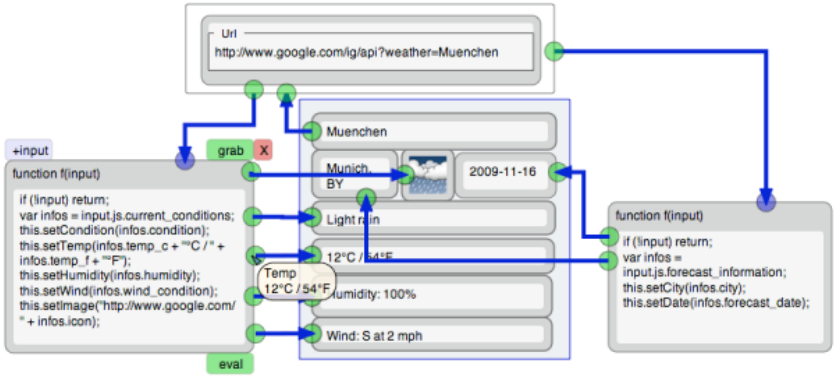


Figure 5.18.: A weather monitor built with Lively Fabrik

standard JavaScript (see the two blocks at the left and the right sides in Figure 5.18).

Summarizing (see Table 5.3), despite the applicative domain to which they are addressed, it is possible to notice how visual languages adopt recurrent types of metaphor and concepts. Most visual languages rely on the "building block" metaphor (e. g., Minibloq and Scratch), others implement application logic using pipes and connectors (e. g., Yahoo Pipes and Pure Data), and some of them combine these two approaches (e. g., Lively Fabrik and SourceBinder). In some cases, like Microsoft Kodu, visual languages are coupled with the notion of rule-based programming.

	Type			Aged [a]	Domain				Features		
	Commercial	Open Source	Research		General-purpose	Education	Multimedia	Other [b]	Building blocks	Connectors	Rule-based
AgentSheets [Repenning et al., 2011]	-	-	×	n/a	-	×	-	-	-	-	-
Buzz Machines	-	-	-	n/a	-	-	×	-	×	×	-
Fabrik [Ingalls et al., 1988]	-	-	×	×	×	-	-	-	×	×	-
LabVIEW	×	-	-	n/a	-	-	-	×	-	×	-
Lively Fabrik [Lincke et al., 2009]	-	-	×	-	×	-	-	-	×	×	-
Microsoft Kodu [Stolee and Fristoe, 2011]	-	n/a	×	-	-	×	-	-	-	-	×
Minibloq	-	-	-	n/a	-	-	-	×	×	-	-
Pure Data [Puckette, 1996]	-	×	×	×	-	-	×	-	×	×	-
Puzzle [Danado and Paternò, 2012]	-	-	×	-	-	-	-	×	×	-	-
Scratch [Maloney et al., 2010]	-	×	×	n/a	-	×	-	-	×	-	-
SourceBinder [Conradi et al., 2010]	-	×	×	n/a	×	-	-	-	×	×	-
Yahoo Pipes	-	-	-	n/a	-	-	-	×	×	×	-
vvvv	-	-	-	n/a	-	-	×	-	×	×	-

[a] Research project older than five years. [b] E. g., embedded software, industrial automation and mashups.

Table 5.3.: Summary of the analyzed visual languages

6 The WOAD Visual Editors

Contents

Drawing upon the solutions to the open problems described in Chapter 4, two distinct, but integrated prototypical visual editing solutions have been designed and developed: the WOAD Template Editor and the WOAD Mechanism Editor respectively. This chapter describes the user interface of the two visual editors and how users can perform their tasks with them. On the other hand, implementation details will be described in Chapter 7.

6.1. The WOAD Template Editor

The WOAD Template Editor prototype was conceived to provide users with an editing environment with a familiar user interface (see Section 4.1). Moreover, a key aspect concerned the need to reduce any kind of user interaction that requires users to perform operations in a non-visual manner, in order to reduce their efforts to learn how to use the WOAD Template Editor.

The user interface of the WOAD Template Editor presents a three-column layout (see Figure 6.1) [Cabitza et al., 2011b]. In particular, both left and right columns are resizable panels, which can be minimized to leave more space for document templates editing purposes. The left column (which is referred as A in Figure 6.1) contains the

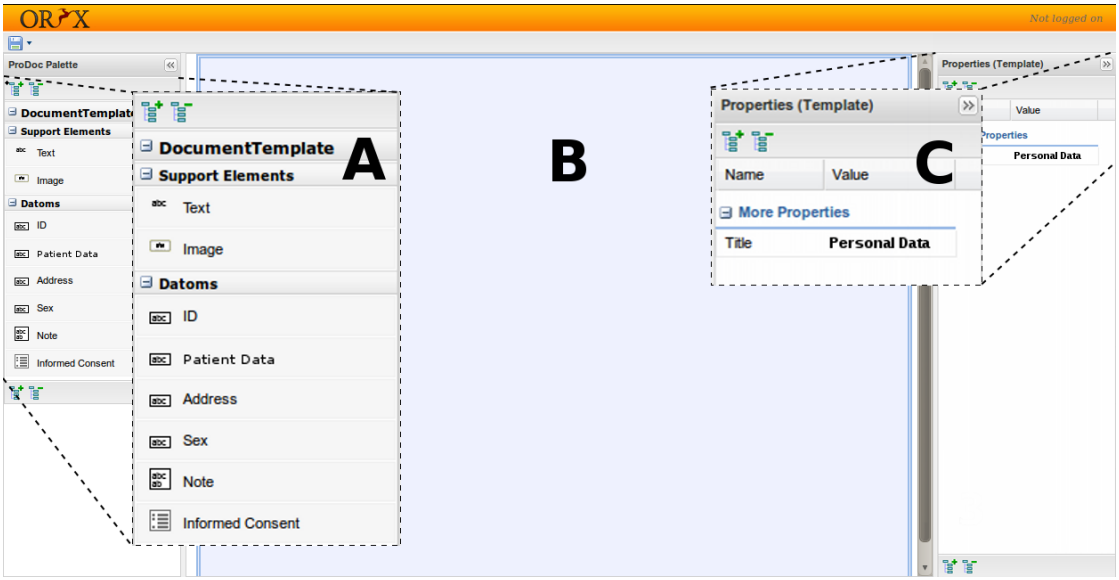


Figure 6.1.: An overview of the WOAD Template Editor user interface

palette that lists all the available document components that users can use to build their own document templates, grouping them into three homogeneous categories. Since documents are not only composed of data fields, the first group in the palette (namely "Support Elements") contains a set of document components that allow users to enrich their document templates with pieces of text and images (e. g., to create the heading of a document or to add some descriptive texts). Below this first group of document components, the palette contains the groups pertaining both to the existing Datoms and (if any) the already created Didgets (i. e., the instances of Datoms that have already been placed either in the document template that the user is currently editing or within another one that the user has previously edited). The central area (B in Figure 6.1) contains the WYSIWYG representation of a document (with the same proportions of a standard A4 paper sheet) in which users can put and manage the Didgets and the other document components they need to add to their document templates. Finally, the right column (C in Figure 6.1) contains the list of properties that allow to simply customize the value of some parameters (if any) pertaining to the selected element in the editing area: for instance, assuming that the user added to the document template an image, with the aim to add the hospital logo to the heading of the document, she can reach this goal just specifying the value of the picture *Uniform Resource Identifier (URI)* in the properties exposed by the image document component (see Figure 6.2).

When users want either to create or edit a document template, they simply have to

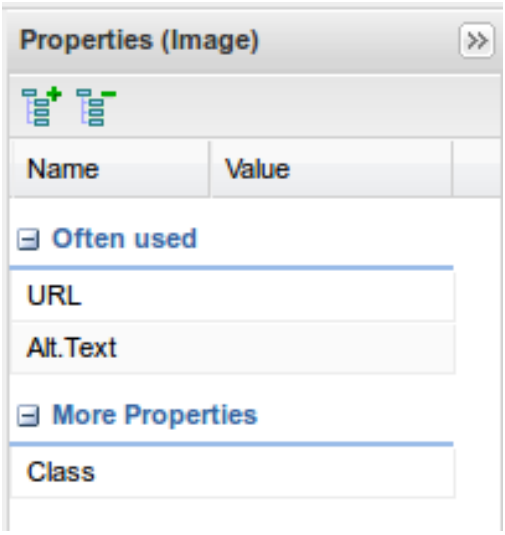


Figure 6.2.: The Properties panel of the WOAD Template Editor (showing the image component's properties)

select the Datoms they needs from the palette in the left column (A in Figure 6.1), drag them over the WYSIWYG representation of the document template in the central area (B in Figure 6.1) and drop them at the desired position. Once a Datom has been dropped, the related new Didget is created and added in the left palette (see Figure 6.3). As described in Section 3.2, already existing Didgets can be reused within different document templates. Using an already existing Didget in a document template requires users to perform the same drag'n'drop operations they have to perform when they place a Datom. However, in this case, when the existing Didget is dropped within the document template, nothing new is added to the left palette.

For instance, clinicians of an hospital ward could need to define their own customized document template of the *Patient Data Sheet*. Using the WOAD Template Editor, a clinician can autonomously complete this task: using drag and drop, she picks up the *Patient Data*, the *Sex* and the *Address* Datoms in the left palette and she drops them in the empty document template, creating their respective Didgets. Similarly, the clinicians can complete the document template by adding a simple piece of text, in order to create a heading for the document (the final result can be seen in Figure 6.4).

In some cases, which are strictly dependent on the applicative domain (typically the healthcare domain), users can have the need to share pieces of data among different documents. Since Didgets can be reused and they have been conceived to meet the users' need to share their content among different documents (see Section 3.2 and Section 4.1),

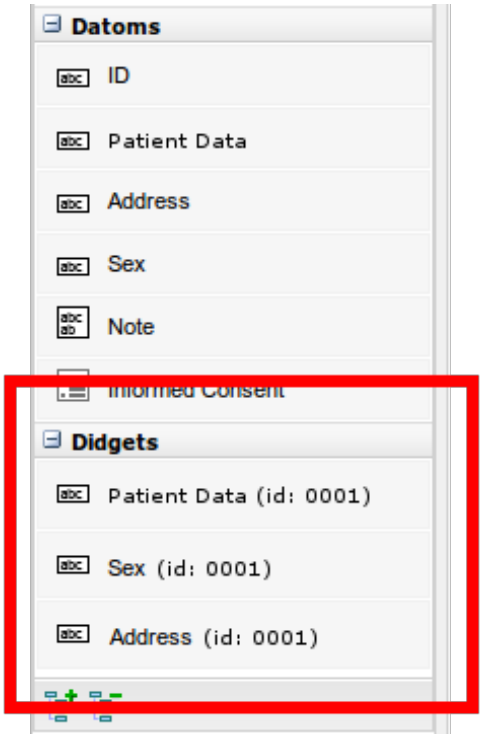


Figure 6.3.: The WOAD Template Editor's palette, after the creation of some new Didgets

the user interface of the WOAD Template Editor makes users able to specify the level of data sharing of Didgets in a fully visual manner. When users drop a Datom and create the related new Didget, by default the latter will hold only local data (see Table 4.1). Subsequently, users can simply set the level of data sharing of a Didget through the first four icons of the contextual menu that appears below the Didget (see the icons labelled with numbers from 0 to 3 in Figure 6.5).

Moreover, users often need to manage data that, in their paper-based documents, is organized in tabular form, and accordingly they have the need to create similar tables in their document templates. Didgets meet this users' need since they have been conceived to be configured either as "single" or as "multiple" (as described in Section 3.2). The user interface of the WOAD Template Editor allows users to set this Didgets' feature in a visual manner: using the last icon on the right side of the Didget's contextual menu (see Figure 6.5), users can select the "multiple" option. Once a Didget has been configured as "multiple", users must set the number of times that Didget's fields will be displayed. This operation can be simply performed through the list of properties in the right column: users need just to specify the value of the `Table Rows` property.

Figure 6.4.: The *Patient Data Sheet* document template

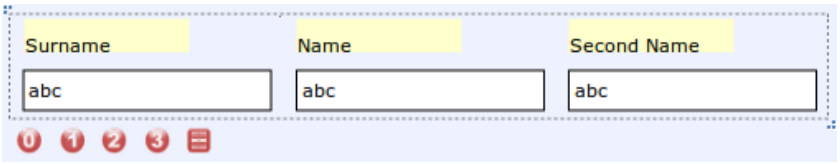


Figure 6.5.: A detailed view of the Didget's contextual menu

Once the user completes all the editing operations she is performing, in order to make the WOAD document template persistent, she just has to act on the Save button in the WOAD Template Editor toolbar, which can be easily recognized by the familiar "floppy disk" icon as in traditional word processors (see Figure 6.1). This simple operation has the effect to store the document template within both the Repository (this function will be described in Section 7.1.1), with the aim to make it available for further changes (also by other users), and the WOAD Template Manager, making it available to the other WOAD framework components (see Section 7.1.2). In this way, when users switch back to the main user interface of their WOAD-compliant systems, they can immediately create the documents as instances of the new document template and fill them in with data.

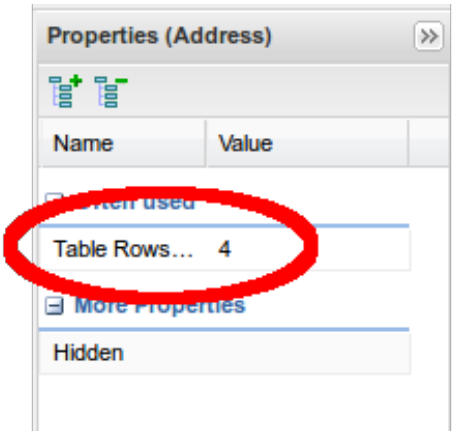


Figure 6.6.: Setting the number of repetitions for a "multiple" Didget

6.2. The WOAD Mechanism Editor

Similarly to the WOAD Template Editor prototype, even the WOAD Mechanism Editor prototype was conceived to be as simple as possible. In this case, the choice to create the WOAD mechanisms through a visual language (see Section 4.2) had a strong influence on how the need of intuitiveness and usability has been addressed: (i) the visual language should contain the constructs users need to compose their own WOAD mechanisms, and (ii) the editor's user interface should be conceived to reduce the interactions performed in a non-visual manner (i. e., typing some textual data).

For this reason, this section will first go through the description of the WOAD Visual Language and, subsequently, of the user interface of the WOAD Mechanism Editor.

6.2.1. The WOAD Visual Language

The WOAD Visual Language is based on the "building block" metaphor (see Section 5.2.3). According to this metaphor, the language constructs are represented through a set of composable blocks. In particular, the WOAD Visual Language relies on the visual language defined within the MIT OpenBlocks framework [Roque, 2007] (further details about the choice of this particular visual language will be provided in Section 7.2).

The WOAD Visual Language encompasses a set of simple constructs, which have been defined according to the findings of previous field studies in the healthcare domain Cabitza et al. [2009a]; Cabitza and Simone [2012] and validated as it will be described in Chapter 8. These constructs allow users to define mechanisms that reach a discrete level of complexity: the complete list of language constructs is reported in Table 6.1. The `mechanism` construct provides users with the visual representation of a WOAD

mechanism, with the distinction between *when-* and *then-part* (see Figure 6.7). In order to define their WOAD mechanisms, users need to make reference to the Didgets' data fields both within mechanisms' conditions and actions. To this aim, the visual language provides a dynamically populated set of data field constructs, which represent Didgets that have already been created within the WOAD Template Editor. These constructs are labelled following the `template_id.didget_id.field_id` notation. Furthermore, users often need to specify some constant value (e. g., a positive number, a string of characters or a boolean value). To this aim, the visual language encompasses a set of four constructs: while two of them represent the boolean values (i. e., `true` and `false`), the other two constructs allow users to specify the textual or the numeric values they need. This is the only type of construct that requires users to insert textual values. Similarly, also the value of Didgets' data fields can be numeric, boolean or textual. For this reason, the shape of both constant value blocks and Didgets' data fields blocks changes according to the data type of their values (e. g., numeric blocks have a triangular shape on their left side). Consequently, also the other language constructs (e. g., comparators) are defined to be connected with blocks representing different types of data.

In the mechanism's *when-part*, users can define the set of conditions picking the language constructs they need within a predefined range of choices: comparators, arithmetical operators and aggregators. The first two are very simple and intuitive, while aggregators deserve some more attention. In some cases, users may need to define conditions on sets of values: for instance, a clinician may need to define a condition that checks if a particular allergy is reported in the list of patient's allergies. Another example may involve the need to check if a patient has more than a certain number of allergies (e. g., more than three allergies). Through aggregators, users can define this kind of complex conditions. Aggregators encompass two constructs that are similar to some spreadsheets' functions (i. e., `count` and `avg`) and the inclusion operator (i. e., `in`), which allow users to define conditions over sets of fields[1]. In the WOAD Visual Language, a basic assumption is that all conditions in the mechanism's *when-part* are evaluated as in strictly logical conjunction (i. e., in `AND`); if users need to define a disjunctive set of conditions (i. e., `OR`), they simply must define distinct mechanisms, each one containing the portions of conditions to be evaluated. For instance, if a clinician wants to check that an inpatient is in a critical situation of hypothermia or high fever, the related condition is "body temperature $<$ 35 *OR* body temperature $>$ 40". In this case, the clinician must define a mechanism to check if "body temperature $<$ 35" (i. e., hypothermia)

[1] The sets of fields are defined using the `*` wildcard character and are referenced through the `template_id.didget_id.*` notation.

and another one to check if "body temperature $>$ 40" (i. e., high fever). This choice guarantees on the one hand a simpler formulation of the *when-part* of the mechanisms and on the other hand a greater independence between rules in case of their modifications.

In the *then-part* of a mechanism, the only available constructs are the `set` actions, which allow users to specify the APIs they want to convey. For instance, when a clinician defines a mechanism to check if a patient is in a critical situation (e. g., the patient's body temperature is greater than 40 Celisus degrees), in order to proactively convey this awareness information among her colleagues, she just need to pick the `set criticality` construct and to specify the data field construct that refers to the Didget's data field close to which she wants to convey the *Criticality API* (e. g., the `body_temperature` field in the Didget called `vital_parameters`). Even if APIs are conveyed as graphical clues (see Section 1.4.1), the `set` constructs do not allow users to specify the graphic features of APIs, since the definition of the graphic style of APIs is not part of the WOAD Visual Language. This is motivated by the aim to decouple the definition of WOAD mechanisms from the definition of the affordances that are shown to users when APIs are conveyed. In this way, such affordances can be defined according to the well-known conventions of the group to which users belong (see Chapter 7 for further details). Nevertheless, in some cases users might want to partially modulate the APIs they want to convey, in order to give more specificity to the conveyed information. This goal can be simply reached by specifying the value of some optional, API-related parameters, the value of which can be specified using the constructs to define constant values. For instance, the `set safety` action accepts a parameter, called `risk level`, which allows users to convey different affordances (e. g., different icons), giving more granularity to the conveyed awareness information. A further example is represented by the `set appropriateness` action that accepts a parameter aimed at specifying a message to be conveyed through the *Appropriateness API*.

6.2.2. The WOAD Mechanism Editor User Interface

The WOAD Mechanism Editor is a window-based application that provides users with a visual editing environment within which they can use the WOAD Visual Language to autonomously define their own WOAD mechanisms.

The WOAD Mechanism Editor main window presents a two-column layout (see Figure 6.7). While in the left column there is the palette that makes accessible to users the WOAD Visual Language constructs (i. e., blocks), the central part of the window is the editing area in which users can drop and connect the blocks they need to compose the mechanism. Moreover, the top region of the user interface contains the toolbar that

	Constructs	When-part	Then-part	Recursive [a]
Action	`set API` [b]		×	
Aggregation	`average`, `count`, `in`	×		
Arithmetical	$+$, $-$, $*$, $/$, %	×		×
Comparison	$=$, $\neq$, $<$ [c], $\leq$ [c], $>$ [c], $\geq$ [c], `like` [d]	×		
Constant Values	`true`, `false`, «constant value»	×	×	
Field Accessors	`template_id.didget_id.[field_id, *]`	×	×	
Logic	`not`	×		×
Rule	`mechanism`			

[a] The constructs can be recursively composed.
[b] `API` is a placemark for APIs' names (e.g., `set criticality`). [c] Only for numbers.
[d] Only for strings.

Table 6.1.: The constructs of the WOAD Visual Language

provide users with familiar buttons, through which they can perform operations, for instance, to store WOAD mechanisms for further modifications. The bottom part of the editing area contains the trash icon to delete the currently selected mechanism.

The editing area allows users to drop and connect the visual language blocks that they pick up and drag from the palette, in order to visually define their own mechanisms. The palette gives access to the language blocks grouping them in homogeneous sets, which are presented to the users as a list of tabs. The first eight tabs pertain the constructs of the WOAD Visual Language different from the Didgets' data fields; the other tabs give access to the blocks related to Didgets' data fields, grouping them according to the document template in which users inserted them using the WOAD Template Editor. Tabs are characterized by a color and a label, which allow users to quickly identify the kind of blocks they give access: e. g., with reference to Figure 6.7, white tabs group blocks that provide access to the data fields in the document templates' Didgets. Moreover, the tabs pertaining to blocks with similar meanings and purposes share the same color (e. g., the three tabs that group comparator blocks are coloured in blue). Each tab in the palette gives access to its set of WOAD Visual Language blocks by showing a scrollable panel. Inside these panels, blocks are presented to users directly in their graphic look. Moreover, on the top of each panel there is an icon (see Figure 6.8) that suggests at a glance the category of language blocks that are visualized. In particular, in the panels that group blocks related to the Didgets' data fields of a specific document template, the icon is a small thumbnail that represents how the document template looks like. This allows users to view the related document template in a zoomable preview window (see Figure 6.9).

For instance, clinicians of a *Neonatal Intensive Care Unit* (NICU) of a hospital could

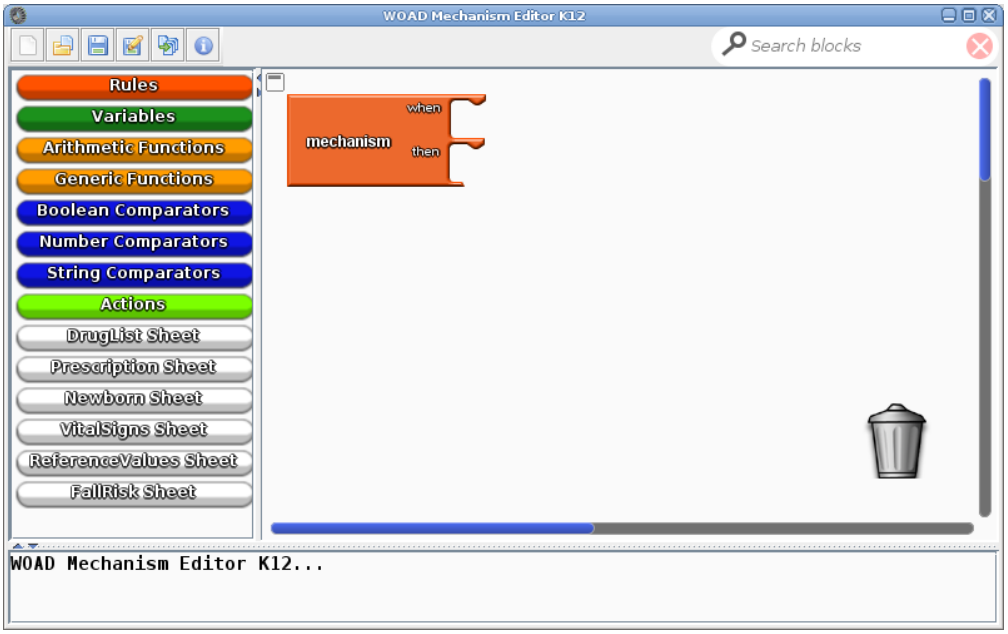


Figure 6.7.: An overview of the WOAD Mechanism Editor user interface

have the need to create a mechanism that conveys a *Criticality API* if the APGAR score of a premature newborn is lower than 4, within five minutes after the delivery. This mechanism only requires data fields from the document template that clinicians call *Newborn Sheet*. Using the WOAD Mechanism Editor, a clinician can autonomously create this mechanism. When she starts to create the new mechanism, the WOAD Mechanism Editor automatically places a "`mechanism`" block in the editing area. At this stage, to define conditions in the mechanism's *when-part*, the clinician would perform a simple sequence of operations: picking up the "less than" operator block (i. e., "<") in the palette; dragging and dropping the selected block in the editing area; connecting it to the "`mechanism`" block. Then, the clinician would repeat these operations to put in connection with the "<" block both the block related to the APGAR field in the *Newborn Sheet* document template and the block in which she will specify a numeric constant value, i. e., 4 (see Figure 6.10a). By repeating this sort of drag'n'drop operations, the clinician would define the second condition of the mechanism, in order to check if the premature newborn is in her first five minutes of life. In regard to the *then-part*, the clinician would complete the mechanism by adding a specific action block (i. e., the `set criticality` block) to convey a *Criticality API* close to the APGAR field in the *Newborn Sheet*. This requires the user to perform the same drag'n'drop operation that she performed to connect blocks in the *when-part* of the mechanism (the final result can

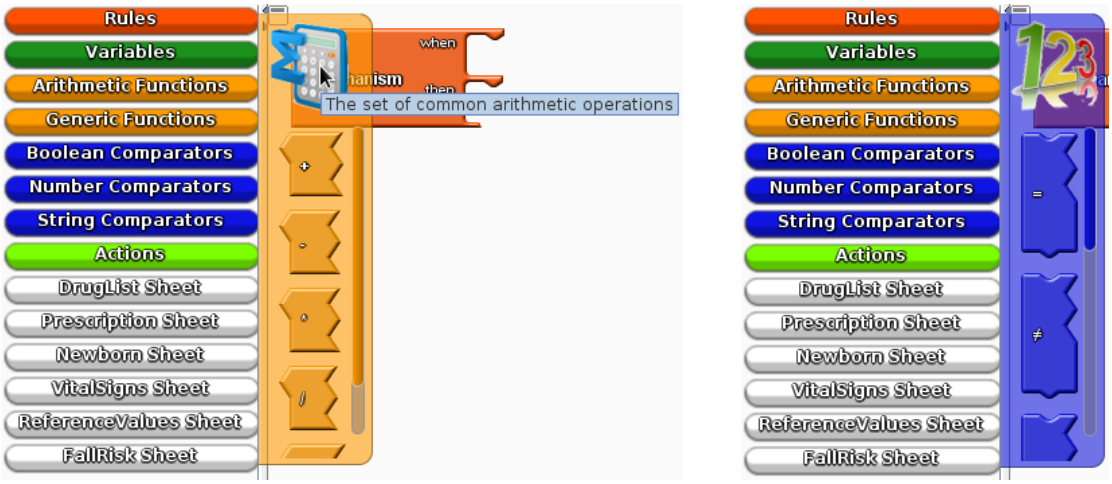


Figure 6.8.: A detailed view of the WOAD Mechanism Editor palette

be seen in Figure 6.10b).

Once the user completes all the editing operations she is performing, in order to make persistent the WOAD mechanism, she just has to act on the 'Save' button in the toolbar (i. e., the button labelled with the "floppy disk" icon, as in the WOAD Template Editor). In this way, users can make further changes to the WOAD mechanism. Nevertheless, this operation does not make the mechanism available to the Mechanism Interpreter (see Section 3.6) in order to be executed. To reach this goal, the user has to act on the 'Export' button (i. e., the fifth button from the left in the toolbar) in order to translate the WOAD mechanism into the format that the rule engine of Mechanism Interpreter is able to execute (further details about this translation process will be described in Section 7.2.3).

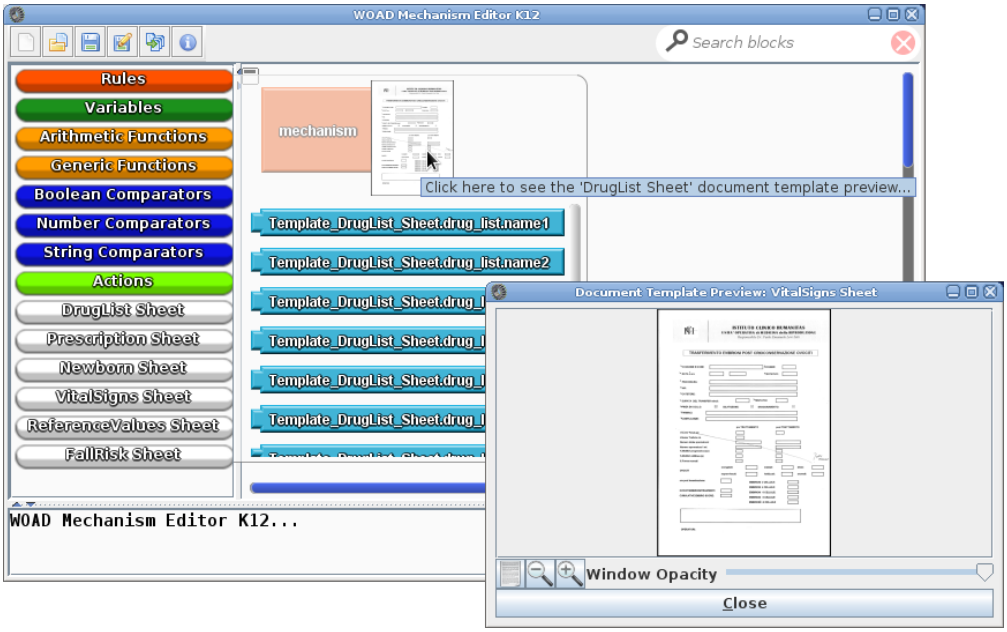
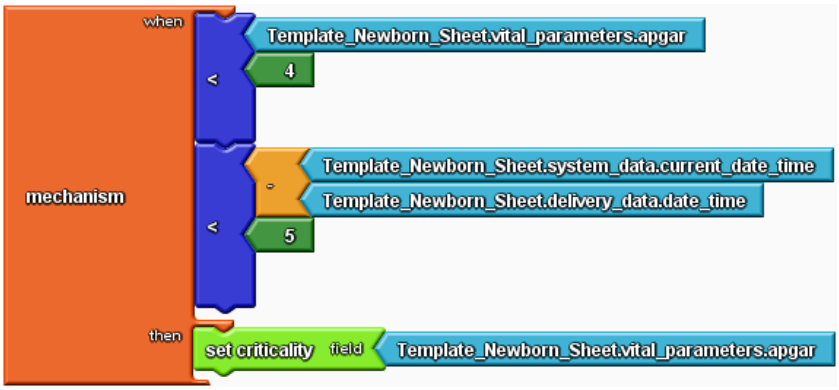


Figure 6.9.: The blocks to access Didgets's data fields and the document template preview window

(a) Checking the APGAR score field.



(b) The completed "APGAR" mechanism.

Figure 6.10.: Building the mechanism to check premature newborns' APGAR score

7

The Implementation of the WOAD Visual Editors

Contents

The development of articulated, even if prototypical, software systems involves software functions (e. g., to manage drag'n'drop interactions) that have already been developed by different teams of developers and made available to software designers according to the open-software approach. This triggered a virtuous circle that led these software artifacts to be increasingly improved and purged of errors and bugs, with the result to produce a set of roughly equivalent software solutions.

In this light, in order to avoid to re-implement all functionalities from scratch, to develop the two prototypes of WOAD Visual Editors, respectively to compose the WOAD Document Templates and the WOAD Mechanisms, the approach that has been adopted involves the reuse of existing software solutions (e. g., see [Lenz et al., 1987; Krueger, 1992; Basili et al., 1996]). This approach allows to focus on the development of new and innovative features, rather than to lose time and efforts to re-develop already existing basic

functionalities (e. g., components' palettes and drag'n'drop management). However, one has to notice that sometimes the poor documentation associated with the open-software components hinders this virtuous process.

This chapter describes the most important implementation details of the two WOAD visual editors that have been described in Chapter 6. For the sake of uniformity with Chapter 6, the first part of this chapter describes the implementation of the WOAD Template Editor, followed by the description of the implementation of the WOAD Mechanism Editor.

7.1. The WOAD Template Editor

In the light of software reuse, before starting to build the WOAD Template Editor prototype, an analysis of the existing XForms editing solutions has been undertaken (see Table 7.1), since to meet the need of flexibility in the document structure it has been chosen to adopt the XForms standard to define Datoms (see Section 4.3), and consequently the generated documents. Unlike the survey of the existing document visual editors that has been presented in Section 5.2.1, which was aimed at describing the most relevant feature of this kind of solutions without any technology-related constraint, the aim of this second, and more focused analysis was to find a suitable XForms editing solution that allows to be extended and customized in order to build the WOAD Template Editor.

The initial idea was to customize an existing word processor with XForms editing capabilities, in order to provide end-users with a well-known user interface. Due to its features and its extendability, *OpenOffice.org Writer* (see Section 5.2.1), and consequently its "brothers" (e. g., *LibreOffice Writer* and *IBM Lotus Symphony Documents*), was the perfect candidate to be customized: it provides end-users with (i) a user interface that mimics a paper-based document, within which they can drag and drop the desired XForms components, and (ii) an easy way to define the underlying XForms data model. Nevertheless, despite its apparent simplicity, a more detailed analysis has immediately shown the limits of *OpenOffice.org Writer*. In particular, the XForms support is only partial: XForms documents are stored directly into binary *Open Document Format (ODF)* files, like what happens with any other OpenOffice.org document. This results in the limit to be unable to communicate with an external, centralized data storage service, i. e., the WOAD Document Data Repository component (see Figure 3.8 and Figure 4.2). Moreover, storing XForms documents in the binary ODF format would require to adopt *OpenOffice.org Writer* as the WOAD Layout Engine component, rather than a simple web browser.

The alternative solutions to edit XForms documents that are available can be grouped in two families: (a) the editors that relies on some *Integrated Development Environment (IDE)*[1], and (b) the web-based ones. While the editors belonging the first group must be discarded, because they are too much far from end-users skills, web-based editors resulted the best choice to be used to build the WOAD Template Editor. With the advent of the *Web 2.0*, web applications have been subjected to a remarkable development, which has led their usability to compete and, in some cases, to overtake the one of desktop applications.

The most notable web-based XForms editors that emerged from the analysis of existing solutions were *Orbeon Forms* and the already mentioned *Oryx XForms Editor* (see Section 5.2.1). Orbeon Forms encompasses a XForms processor, a XForms repository and the visual editor. The editor vaguely mimics the appearance of paper-based documents and does not allow the free arrangement of XForms components (i. e., users can only split up XForms documents in different sections, within which they can arrange XForms components following a grid layout). On the other hand, Oryx XForms Editor provides users with an environment that is composed of a repository and the visual editor. In particular, the visual editor provides users with the ability to define the arrangement of XForms documents without any constraint. Moreover, the editor can be easily extended through the creation of plug-ins to support any kind of document (e. g., *Business Process Model and Notation (BPMN)* diagrams). This flexibility has been the key feature that promoted Oryx XForms Editor as the foundation upon which to build the WOAD Template Editor.

7.1.1. Oryx: An Extendable Editing Environment

Oryx is an open source[2], web-based visual environment that was born in the academic landscape (see [Decker et al., 2008a,b]). To be precise, Oryx is a suite of two distinct, but intertwined web applications: the Oryx Editor, which provides users with a visual editing environment, and its complementary application, i. e., the Repository (see Figure 7.1), which is responsible store the XForms documents that have been created using the editor. Moreover, Oryx provides some basic social and collaborative features, which make it compliant with some of the EUD tenets (see Section 2.3.1). In fact, the Repository application is responsible to manage both the sharing of XForms documents within the working environment (through the integration of simple social features, like comments and ratings) and their collaborative improvement (using a simple lock strategy based

[1] Most of the IDE-based XForms visual editors rely on Eclipse (`http://www.eclipse.org/`). [2] Oryx is distributed under the MIT License (`http://opensource.org/licenses/mit-license.php`).

	XForms		Features							
	UI	Model	License	Type	Export	Extendable	XSD Import	Drag'n'Drop	Free Layout	CSS Support
Oryx XForms Editor	×	×	Open source (MIT)	Web-based Application	×	× (Plug-ins)	-	×	×	×
Orbeon Forms	×	-	Open source (LGPL) [a]	Web-based Application	-	× (Source code)	-	-	-	-
OpenOffice.org Writer	- [b]	×	Open source (LGPL)	Standalone Application	-	× (Extensions)	-	×	×	-
IBM Visual XForms Designer	×	×	Free [c]	Plug-in (Eclipse)	×	-	×	×	×	×
JBoss VPE XForms Plug-in	×	×	Open source (LGPL)	Plug-in (JBoss Tools [d])	×	× (Source code [e])	-	×	×	×

[a] The *Orbeon Forms* commercial version provides more functions, e. g., Oracle and Alfresco integration, PDF and *XML Schema Definition (XSD)* support, and resources versioning.
[b] *OpenOffice.org Writer* uses its own standard form controls to render XForms and ONLY within its GUI.
[c] For non-commercial use. The commercial counterpart is called *IBM Lotus Forms Designer*.
[d] *JBoss Tools* is a set of Eclipse plug-ins for JBoss and the related technologies
[e] `http://anonsvn.jboss.org/repos/jbosstools/workspace/kukeltje/xforms/nl.fortythree.jbosstools.vpe.xforms/`

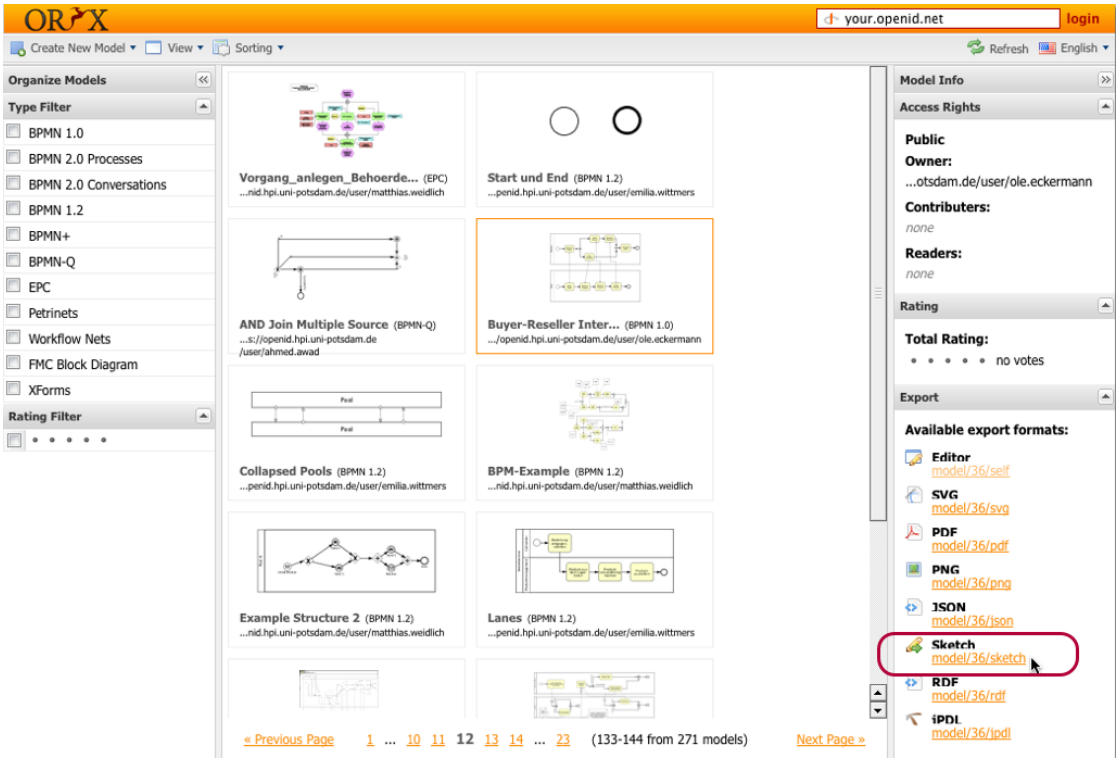
Table 7.1.: Summary of existing XForms visual editors

on the mutual exclusion technique). Leveraging these collaborative and social features, employees are encouraged to autonomously create and continuously refine the XForms documents they need, in order to better cope with the actual needs of the group in which they perform their activities (see Chapter 1).

Both the Oryx Editor and the Repository are based on a platform-independent client-server architecture that heavily uses AJAX techniques: while the server-side part is implemented using the standard Java programming language, the client-side part relies on the W3C standard languages, i. e., the *eXtensible HyperText Markup Language (XHTML)*, the *Scalable Vector Graphics (SVG)* and the JavaScript language (in particular the whole environment is based on the ExtJS[3] framework).

Within the Oryx XForms Editor visual environment, all editing operations are managed by its client-side component. As a direct consequence, the Oryx XForms Editor is only able to produce a SVG-based graphic representation of the XForms documents that users are editing, and accordingly, it stores them within the Repository using a JSON-based representation. In this light, XForms documents need to be translated into the XForms syntax, in order to make them available to other applications (e. g., a XForms processor). This translation process is executed on the server-side part of the Oryx XForms Editor.

[3] `http://www.sencha.com/products/extjs`



From `http://bpt.hpi.uni-potsdam.de/pub/Oryx/DeveloperNetwork/repository-sketchy-models.png`. *Accessed: 2012-08-28. (Archived by WebCite® at* `http://www.webcitation.org/6AFtXJd7t`*).*

Figure 7.1.: The Oryx Repository user interface

The main components of the Oryx XForms Editor user interface have already been described in Section 6.1. Summarizing, with reference to Figure 7.2, the central area of the page[4], i. e., the editing area (namely, the *canvas*) is devoted to receive the various XForms components that users drop in the document they are editing. On the left side, Oryx XForms Editor presents to users the palette of XForms components (which are grouped by category) they can pick up, drag and drop within the *canvas* area. On the other hand, on the right side, the Oryx XForms Editor provides users with the possibility to specify the values of some properties of XForms components (e. g., a XForms input field) with the aim to customize it (e. g., specifying the id or background color).

As mentioned, one of the most peculiar feature of the Oryx XForms Editor is its plug-in based architecture, which significantly contributes to make it easily adaptable and extendable. In fact, the Oryx Editor core only implements the basic parts of the user

[4] The ExtJS framework allows to arrange web pages to simulate the window-based layout that users usually find in the traditional desktop applications.

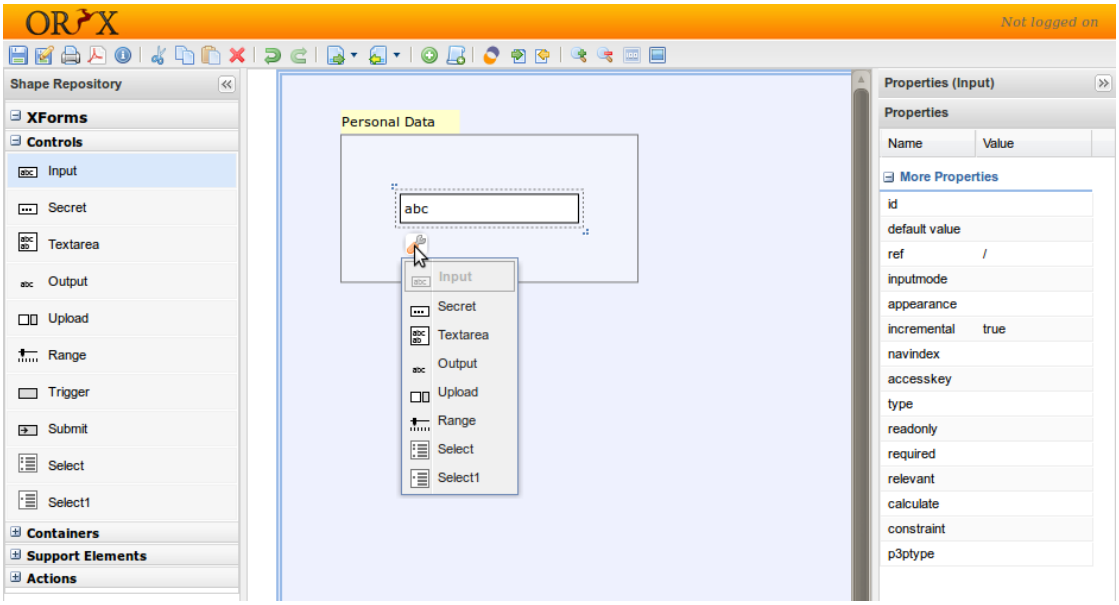


Figure 7.2.: The Oryx XForms Editor user interface

interface (i. e., the empty toolbar, the empty lateral panels and the canvas) and exposes the plug-ins' *Application Programming Interface* to interact with them, constituting the "backbone" of its visual editing capabilities. On the other hand, XForms editing capabilities are obtained through a pre-installed set of plug-ins.

Each plug-in is subdivided into client-side part and server-side part. The latter is not mandatory and it is needed only when the same goal cannot be achieved by the client-side counterpart[5]. On the other hand, the client-side part of a plug-in is mandatory and it implements the JavaScript code to manage the user interface (e. g., to add buttons to the toolbar or additional user interface components to the two panels) and users' interactions with model components, their properties and the *canvas* area (e. g., when a user selects a XForms component to drag it to a new position). One of the most relevant part of a plug-in is constituted by *StencilSets*[6]. At a glance, a *StencilSet* can be seen as a *JavaScript Object Notation (JSON)*-based list of palette components; however, this description of *StencilSets* is too much reductive. The *StencilSet* main aim is to map the XForms components with their SVG-based representation, to define the relations

[5] Since using the JavaScript programming language within a standard web browser makes impossible to directly interact with the objects and the other data structures that are stored on the web application server, in order to reach this goal it is needed to delegate this task to a custom server-side Java code (e. g., to interacting with a database or executing complex operation to convert a XForms document into the XForms syntax).

[6] Each plug-in can encompass more than one *StencilSet*, in order to support different types of document components.

among them (e. g., which XForms components can be positioned within each other and which cannot) and the groups to which they belongs (which allow to create collapsible subcategories in the palette of components placed on the left panel).

7.1.2. Implementing the WOAD Template Editor Prototype

The preliminary activity that has been undertaken to move a step towards the implementation of the WOAD Template Editor (which has been described in Section 6.1) was to configure the Oryx Editor to provide users with an environment that mimics the ones that they can find in traditional word processors (see Section 4.1).

The development of WOAD Template Editor's features made necessary to define a new *StencilSet* and to create a set of plug-ins, just like what happens in adding to the Oryx Editor the ability to support any other kind of model.

First of all this required to map Datoms within a suitable *StencilSet*[7], in order to dynamically populate the palette in the left panel of the user interface, as required when a user drops a Datom in the *canvas*, this would cause the creation of a new Didget that would appear in the palette to make it reusable within the other document templates. To reach this goal the JSON object corresponding to a *StencilSet* should be suitably structured and a specific plug-in, which encompasses both client-side and server-side code, has to be created: while the server-side part is responsible to generate the new Didget and to accordingly update the *StencilSet* data structure, the client-side code manages drop operations of the Datoms in the *canvas* and updates the palette when the server-side part communicates that the new Didgets have been successfully created.

Moreover, since in the WOAD Template Editor the Didget's contextual menu allows to visually manage the Didgets' ability to both share their content among different instances of document templates and their ability to manage multiple data in a tabular arrangement, the development of a second plug-in was required. The aim of this plug-in is to manage users' interactions with the contextual menu, and accordingly to update the Didgets' status in the *StencilSet*.

Finally, the development of a third plug-in was needed to both manage storage operations and the communication among the Oryx Editor core and the other two

[7] The operation of automatically filling in the *StencilSet* of the WOAD Template Editor is a task of another WOAD framework component, i. e., the Datom Editor (see Figure 4.2). Taking into account that the detailed description of the Datom Editor is out of the purposes of this thesis, it is sufficient to know that this component has been obtained by customizing the Oryx Editor plug-in for XForms in two ways: (i) reducing the set of available XForms elements (leaving only those aimed at filling in data), and (ii) adding a custom plug-in to manage the integration with the other WOAD components, i. e., the WOAD Template Editor and the Didget Manager.

WOAD Template Editor plug-ins. In particular, to store a document template, this plug-in needs to perform two distinct operations: (i) to store the internal Oryx Editor data structure that represents the document template in the Repository, to allow further changes and their sharing among users, and (ii) to serialize the same document template, using a XML-based representation, to make it available to the other components of the WOAD framework, i. e., to both the Template Manager and the Didget Manager (see Figure 4.2).

7.2. The WOAD Mechanism Editor

Similarly to what happened for the development of the WOAD Template Editor, also the development of the WOAD Mechanism Editor prototype was based on a software reuse approach. As proposed in Section 4.2, a fundamental guideline that has been taken into account during the design and the development of the WOAD Mechanism Editor is intuitiveness. Since the survey of visual rule editors (see Section 5.2.2) encompasses solutions that resulted to be scarcely customizable to meet the WOAD framework concepts (i. e., Datoms, Didgets and document templates), the choice was to use a visual language to make them autonomous in the definition of their own WOAD mechanisms. As highlighted in Section 5.2.3, most of the existing visual languages rely on the "building block" metaphor, and some of them couples this metaphor with the concept of rule (i. e., Microsoft Kodu).

However, like in the case of visual rule editors, most visual language are very specialized and are not easy to customize. Thus, the primary need was to find a general-purpose visual language solution that would allow to be easily customized, in order to be integrated in the WOAD framework in compliance with its modularity and extensibility. The solution which has proved to be best suited to meet these needs was the *OpenBlocks* library [Roque, 2007]. OpenBlocks is an open-source Java library that was born at the *Massachusetts Institute of Technology (MIT)* and has become quite popular even among big IT vendors[8]. In particular, to develop the WOAD Mechanism Editor prototype, the choice has been to customize the demonstrative Java-based application provided with the *OpenBlocks Sample project*[9], which encompasses a basic implementation of a visual editing environment (further details will be provided in Section 7.2.2 and Section 7.2.3).

[8] For instance, *MIT OpenBlocks* had been also used by *Google Inc.* within the *App Inventor for Android* project (now *App Inventor Edu* at *MIT Media Lab*). See `http://appinventoredu.mit.edu/`. [9] `http://education.mit.edu/openblocks`

7.2.1. The OpenBlocks Language

The MIT OpenBlocks library encompasses a block-based visual language in which visual blocks can be defined in a declarative manner, through an XML-based syntax (see Appendix A for details). This makes easy to map the constructs (e. g., the `if-then-else` and the `atan` consturucts) of the most part of traditional programming languages with their OpenBlocks-based visual representation (e. g., the OpenBlocks sample application encompasses the mapping with the language underlying the MIT StarLogo TNG[10]).

Each visual block is displayed through a specific set of visual features: a shape, a color and a label. Within their labels, which is mandatory, blocks display to users the name of the related the language constructs. Blocks' shapes and colors contribute in making users able to better recognize blocks at a glance (for instance, if a block represents a boolean function rather than a procedure or an arithmetical function). Visual blocks can expose an arbitrary number of different connectors (see Figure 7.3), which allow to compose them to visually define sequences of code. In particular, connectors can be grouped by two complementary macro-types, i. e., *sockets* and *plugs*. Connectors can be configured to represent different types of data: for instance, a socket connector can be defined to accept connections with numeric data type plugs only. Connector's data type determines the shape that the related connector will expose (e. g., numeric data type can be related to an angular connector shape). Moreover, also connectors can be coupled with a label, which in this case is optional, to provide users with a very short but intuitive description of the role played by the labelled connector. For instance, as shown in Figure 7.3, the `ifelse` block, which is a *function* block that represents an *if-then-else* construct, exposes three connectors: (i) the `test` *socket* connector, which represent a set of conditions and accepts to be connected only with blocks that exposes a boolean *plug* connector, and (ii) both the `then` and the `else` *socket* connectors that accept other *function* blocks. All these blocks' visual features contributes to make the composition of complex sequences of blocks more intuitive and less prone to errors.

7.2.2. The OpenBlocks Visual Editor User Interface

The *OpenBlocks Sample project* encompasses a minimal visual editing environment with a two-column layout (see Figure 7.4), which allows users to focus on OpenBlocks features and visual editing operations.

Since the OpenBlocks visual editor constitutes the "backbone" of the WOAD Mechanism

[10] StarLogo TNG is an educational simulation environment that has been developed at *MIT Media Lab* (see `http://education.mit.edu/projects/starlogo-tng` for more details).

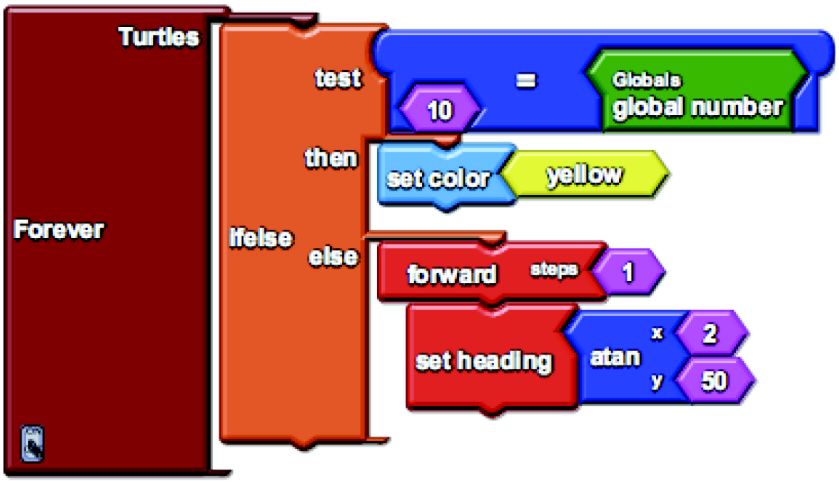


Figure 7.3.: An example of the OpenBlocks visual language

Editor, the most relevant parts of the user interface have already been described in Section 6.2.2. On the other hand, with respect to the WOAD Mechanism Editor, the *editing area* is organized in collapsible pages. Each page represents an independent entity of the application (e. g., the *turtles* of the sample environment). Moreover, the editing area provides users with two useful widgets, i. e., the *MiniMap*, in the upper right corner, and the *Trash* in the lower right corner. The *MiniMap* shows a thumbnail of the whole editing area, which gives users the ability to understand at a glance the status of their coding activities. On the other hand, the *Trash* icon provides users with a meaningful and well-known metaphor, which meaning is trivial.

Similarly, a user who wants to code using the OpenBlocks visual editor must perform the same drag'n'drop operations that have been already described to use the WOAD Mechanism Editor.

7.2.3. Implementing the WOAD Mechanism Editor prototype

The first activity that has been undertaken to develop the WOAD Mechanism Editor was to map the WOAD Visual Language constructs to the OpenBlocks language definition. To reach this goal, it has been necessary to customize the language definition file (i. e., the `lang_def.xml`[11] file), in order to declare the blocks pertaining to the WOAD

[11] The `lang_def.xml` is responsible to store the definition of the visual blocks (i. e., block's label, type and color, its connectors and the related data type). Moreover, the `lang_def.xml` file stores the whole configuration of the visual editing environment. It allow to define the configuration of the editing area (i. e., the set of pages in which to split the editing area, and the features of both the *MiniMap* and the *Trash* widgets), the tabs in the palette and which blocks pertain to a specific tab. Further details about the `lang_def.xml` file and its grammar definition (i. e., the `lang_def.dtd`) will be provided in Appendix A.

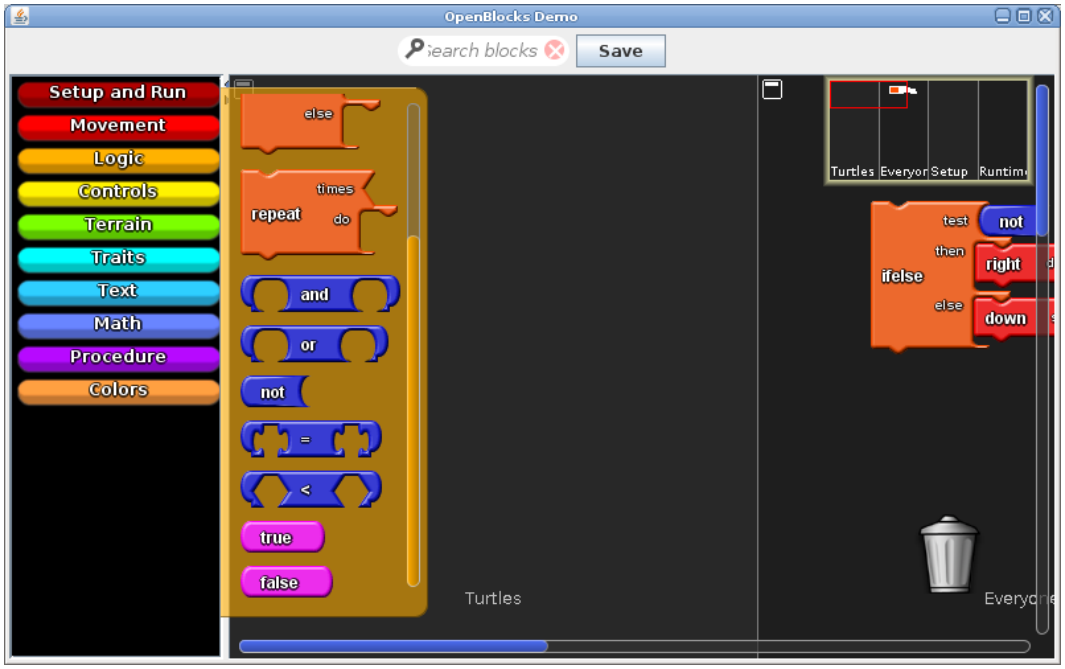


Figure 7.4.: The OpenBlocks Sample user interface

Visual Language. Moreover, a fundamental aspect to be considered was the need to strictly integrate the WOAD Mechanism Editor with the WOAD Template Editor, in order to automatically populate the palette with the tabs corresponding to the existing document templates and, consequently, with the corresponding visual blocks related to their Didgets' data fields (see Section 6.2.2). To this aim, a translation module is responsible to query the WOAD Template Manager and to update the definition of the blocks pertaining to the Didgets' data field within the `lang_def.xml` file.

The reuse of the OpenBlocks demonstration editor user interface required a simplification of its features, in order to give users the possibility to focus on the composition of their mechanisms, minimizing at the same time any possible risk to confuse them with unnecessary graphical elements. The first step towards the simplification mainly concerned the editing area, and it has been achieved by customizing its basic configuration (within the `lang_def.xml` file): the number of pages in which users can drop the visual language blocks (see the description of the editing area in Section 7.2.2) has been reduced to a single page and the *MiniMap* widget has been removed.

After this preliminary simplification, the OpenBlocks demonstration editor has been customized to automatically place a "`mechanism`" block in the editing area when the user starts to create a new mechanism and to make the palette more intuitive. In order to

```
<Block id="12" genus-name="not-equals">
  <Location>
    <X>151</X>
    <Y>25</Y>
  </Location>
  <BeforeBlockId>9</BeforeBlockId>
  <Sockets num-sockets="3">
    <BlockConnector
        connector-kind="socket"
        connector-type="number"
        ...
        con-block-id="17">
    </BlockConnector>
    ...
  </Sockets>
</Block>
```

Listing 7.1: An example of the OpenBlocks serialization format

provide users with an intuitive description of visual blocks, meaningful icons and short *pop-up* messages have been added on top of the scrollable panels that are associated to the tabs in the palette. In particular, the icons of the scrollable panels that pertain to document templates are thumbnails that are automatically generated and show how the related document templates look like. These particular panels' icons provide users with another feature that can be helpful to make them able to better contextualize the blocks they are using with respect to the document template those blocks refer to. In fact, each document template icon acts as a clickable button that opens a *preview window* in which is displayed the zoomable preview of the related document template.

7.2.3.1. The WOAD Intermediate Language

Since the OpenBlocks library heavily relies on XML to define visual blocks' features, as a direct consequence, also its serialization format (see Listing 7.1) relies on the same meta-language (the grammar of the serialization format is defined in the `save_format.dtd` file[12]). On the other hand, as described in Section 3.6, the actual implementation of the Mechanism Interpreter is based on JBoss Drools, which requires to express rules using its native Drools Rule Language (DRL) syntax. Thus, in order to obtain the full integration of the WOAD Mechanism Editor with the other component of the WOAD framework, it was needed to make the visually defined mechanisms readable by the JBoss Drools engine in the Mechanism Interpreter.

This goal has been reached with the definition of an intermediate, XML-based language, which preserves the mechanism application logic and discards all the topological data of

[12] A more detailed description of the OpenBlocks serialization grammar will be provided in Appendix A.

the OpenBlocks serialization format. First of all, the need was to formalize the WOAD Intermediate Language grammar[13]. To this aim, the *XML Schema Definition (XSD)* language allows to define XML-based grammars and provides a level of flexibility and expressiveness greater than that of the more traditional *Document Type Definition (DTD)* language. Moreover, the adoption of the *Java Architecture for XML Binding* (JAXB) framework has been helpful to easily integrate the translation stack within the Java-based source code the OpenBlocks demonstrative editor, since the XSD grammar has been transparently mapped into the corresponding set of Java classes. In addition, the joint use of the JAXB framework and XPath expressions has led to the development of lightweight translation components.

The introduction of the WOAD Intermediate Language required to develop two distinct translators: while the former translates the OpenBlocks language into the WOAD Intermediate Language (e. g., see Listing 7.2), the latter translates the WOAD Intermediate Language into the DRL syntax. Each generated DRL rule is stored into a file within a shared directory, which is readable by the WOAD Mechanism Interpreter. Even if this solution requires a two-pass translation process (see Figure 7.5), at the same time, it introduces the substantial advantage to have a visual editing solution that is integrated within the WOAD reference architecture, but it is also independent of the underlying execution engine, and vice versa. This results in the possibility to change the execution engine without affecting the features of the WOAD Mechanism Editor.

Nevertheless, the execution of this process of translation is not in itself sufficient to maintain constantly updated the set of mechanisms that the WOAD Mechanism Interpreter can execute. In fact, even if the visually defined mechanisms are translated into the DRL syntax, this does not imply that they are automatically loaded in the working memory of the rule engine of the WOAD Mechanism Interpreter. This is due to the fact that the JBoss Drools rule engine requires to statically load rules during its initialization phase, and it does not encompass any process of automatic update of the already loaded rules. In order to overcome this limit, the WOAD Mechanism Interpreter has been endowed with a module, i. e., the *Rule Watcher*, that asynchronously monitors

Figure 7.5.: The translation process from OpenBlocks serialization format to DRL

[13] A detailed description of the WOAD Intermediate Language grammar is provided in Appendix B.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<ns2:rule xmlns:ns2="http://www.maclab.disco.unimib.it/WOADMechanism">
  <when>
    <condition type="&lt;">
      <left_side>
        <field template="Template_Newborn_Sheet" didget="vital_parameters"
            name="apgar" />
      </left_side>
      <right_side>
        <constant_value type="number">4</constant_value>
      </right_side>
    </condition>
    <condition type="&lt;">
      <left_side>
        <operation type="-">
          <first_operand>
            <field template="Template_Newborn_Sheet" didget="system_data"
                name="current_date_time" />
          </first_operand>
          <second_operand>
            <field template="Template_Newborn_Sheet" didget="delivery_data"
                name="date_time" />
          </second_operand>
        </operation>
      </left_side>
      <right_side>
        <constant_value type="number">5</constant_value>
      </right_side>
    </condition>
  </when>
  <then>
    <action type="API" name="Criticality">
      <field template="Template_Newborn_Sheet" didget="vital_parameters"
          name="apgar" />
    </action>
  </then>
</ns2:rule>
```

Listing 7.2: The APGAR mechanism translated in the WOAD Intermediate Language

the directory that contains translated rules: if any change is detected (e. g., a new rule has been added, or an existing one has been modified or deleted), the *Rule Watcher* accordingly updates the rule engine working memory. In particular, the *Rule Watcher* relies on the Java *WatchService* class[14], which allows to constantly monitor a registered resource (i. e., in this case, the directory that stores DRL files) without the need to execute polling operations.

[14] The *WatchService* class is part of the New IO package that has been introduced in Java 7.0. For further details, see `http://docs.oracle.com/javase/7/docs/technotes/guides/io/index.html`.

7.2.3.2. The Definition of APIs' Affordances

Since WOAD Visual Language does not encompass any construct that allow users to define the affordance that will be conveyed by the APIs (see Section 6.2.1), this required to conceive a solution to allow users to unambiguously specify the conventional graphic features, i. e., the conventional *style*, of the API-related affordances.

To this aim, the approach that has been chosen is to group all API-related styles within an unique *style sheet*. In particular, since in the current implementation the *Layout Engine* component is a web browser and WOAD documents relies on the XForms standard, the choice has been to adopt the *Cascading Style Sheet (CSS)* standard (which is widely implemented in standard web browsers).

The deployment of the WOAD-compliant system requires users to interact with some experienced people (e. g., power users or designers) to translate their conventional API-related affordances into the related set of CSS styles (e. g., a *Criticality API* will be configured to change the color of the text to red). In particular, each API is related with at least one CSS class, which defines its graphic style. When a WOAD mechanism is triggered, the Mechanism Interpreter send the API to be conveyed to the Markup Tagger, which translates the API in the related CSS class and apply it to the related Didget's data field. Once the Markup Tagger has applied the class to the Didget's data field, the Layout Engine automatically updates the document, showing the API-related affordance to users.

Nevertheless, this approach has not to be considered a definitive solution. Even if CSS editors have become a widespread software, most of them are tools that have been conceived to be used by developers and web designers, who have a good knowledge of the features of this language. Since the definition of the API-related affordances has to be mainly performed on behalf of users, this approach does not fully meet EUD tenets (see Section 2.3). To overcome this limit, further efforts will be made to allow users to be autonomous in the definition of API-related affordances (see Section 10.1).

8

Validating the WOAD Visual Language

Contents

Once the WOAD Mechanism Editor has been developed (see Section 6.2 and Section 7.2), the next step that has been undertaken concerned the validation of the expressiveness of the WOAD Visual Language. In order to perform this validation, it has been chosen to undertake a qualitative user study in a real-life scenario, and in particular within the reference working environment of this thesis, i. e., the healthcare domain (see Section 1.3). This qualitative user study took place in a Northern Italy teaching hospital, i. e., the *Presidio Ospedaliero di Vimercate*[1].

First of all, this chapter will briefly describe the organizational setting within which the qualitative user study took place. Subsequently, the chapter describes the qualitative interview technique, which has been adopted to perform the described user study. The subsequent section will go through the description of the results of the interview sessions that have been performed with clinicians, in order to elicit the rules governing the promotion of awareness about specific case of interact, namely the adverse events concerning the inpatients' fall risk. The chapter ends with the description of the activities that allowed clinicians to define the WOAD mechanisms related to the elicited rules, and the

[1] `http://www.aodesiovimercate.it/`

discussion that originated the definition of a practice-based solution to the problem of conflicting rules.

8.1. Organizational Setting

The organization of inpatient care activities within the *Presidio Ospedaliero di Vimercate* focuses on the care workflow rather than on the single care tasks. In this light, inpatients are considered as a fundamental and active part of the care process. Moreover, within the hospital there was the need of an horizontal integration of the activities performed by all the stakeholders of the care process. To this aim, the hospital adopted an approach to care activities that is based on the intensity of care. In this way it is possible to evaluate inpatients even from a nursing point of view, rather than only from medical point of view: nursing activities take into account the needs of the inpatient as well as her disease. To better cope with this care approach, the Management Unit of the hospital chose to deploy a mobile and paperless EPR solution[2], which have been integrated with the legacy information system. A relevant feature of this EPR is the possibility to be easily adapted to the needs of the various wards, even if this still requires the intervention of IT staff of the hospital. Currently, this EPR is used daily by both doctors and nurses who cooperate in the management of the medical record of an inpatient throughout all her hospital stay, thanks to the advantages of the implementation of a mobile solution.

In the last decades, the theme of inpatient safety gained a prominent role in the interests of the healthcare professionals as well as of both hospital management and public health institutions (see [Kohn et al., 2000; Wachter et al., 2004; Wachter, 2010]). It is on the basis of this consciousness that the management of the *Presidio Ospedaliero di Vimercate* felt the need of an ICT-based systematic approach that allows clinicians to minimize the risks for the inpatients. In particular, the management focuses on the possibility that some adverse events can occur to the inpatients during their hospital stay and on the need to make clinicians constantly aware about the possibility that an inpatient might be in a risky condition. This focus is also motivated by the introduction of the EPR as this event could influence the capability of clinicians to recognize and deal with adverse events.

The user study described in the following took place to identify the clinicians' needs to cope with the problem of inpatients' safety. One of the preliminary activities that have been undertaken has been the identification of the risk factors. To this aim ethnographic

[2] The *Electronic Patient Record (EPR)* adopted within the *Presidio Ospedaliero di Vimercate* is based on the *Tabula Clinica* (see `http://www.tabulaclinica.com/`) framework and has been developed by Dedalus (see `http://www.dedalus.eu/`).

techniques has been adopted. Ethnographic techniques encompass a heterogeneous set of methods and strategies for collecting and analyzing data, and require researchers to spend some time within the social context that they wish to study, even interacting with people who belong to it (see [Pope, 2005]). In particular, the qualitative interview has been chosen as the tool to perform this qualitative user study. Even if it is a much more interpersonal method with respect to other techniques of data collection (e. g., surveys), thanks to the dialogue that takes place between interviewer and interviewee, the qualitative interview allowed to better elicit the actual clinicians needs, especially uncovering the social aspects of the clinicians care activities.

8.2. The Qualitative Interview

The interview is a qualitative research method that is particularly useful to uncover particular aspects, e. g., a relevant event, related to the personal world of the interviewees (i. e., the users), which otherwise would remain hidden behind the experience of the interviewees themselves (see [Rubin and Rubin, 2005]).

Qualitative interviews can be grouped in three different categories (see [Denzin and Lincoln, 2003; Myers and Newman, 2007]):

Structured Interview This kind of interview is often adopted when the interviewer is the researcher but a third subject. Structured interviews are conceived to be composed of a pre-defined set of questions, which are presented to the interviewees following a pre-defined and rigid order, without any possibility of improvisation.

Semi-structured or Unstructured Interview In this case, not all questions have been defined before the beginning of the interview sessions: during the interview, the interviewer can improvise by changing the order of questions, introducing new ones and and omitting other ones. Usually these kind of interviews are held directly by researchers.

Group Interview This kind of interview contemporaneously involves more interviewees, who can be interviewed by one or more interviewers. Moreover, the type of the interviews can range from structured interviews to unstructured ones.

Since semi-structured interview provides interviewers with a certain degree of freedom, usually researchers adopt this kind of interview to undertake their qualitative research activities. In this way, researchers can be able to continuously adapt the interviews to the various situations they meet during their interview sessions.

Moreover, in order to better support researchers to undertake effective interview sessions, Myers and Newman [Myers and Newman, 2007] developed a set of guidelines[3] with the aim to avoid that the typical problems[4] of qualitative interviews can arise, influencing the interviews' results (see Figure 8.1):

1. the researcher must situate both itself and the interviewee with preliminary questions (i. e., questions about, for instance, name, role, experience, age, nation of origin and gender);

2. the interviewee must be made to feel comfortable, in order to improve the overall quality of the interview;

3. since interview sessions involve different kind of interviewee, with different roles, experiences and responsibilities, it is crucial that the researcher avoid the arise of the *elite bias* (see Footnote 4);

4. the researcher must take into account that each interviewee is an active player in the interview as well as the researcher itself;

5. the researcher must be able to adapt the jargon that she use to make questions according to the one used by the interviewee to express its answers (the *mirroring* technique);

6. the researcher must be able to adapt the interview according to its interpretation of the interviewee's attitudes;

7. the interview must be conceived to comply with well-defined ethical standards.

In order to conduct the qualitative study that is described in next sections, it has been chosen to involve clinicians in some interview sessions based on the technique of semi-structured interviews.

[3] Myers and Newman developed their guidelines on the basis of the *dramaturgical model*, in which both interviewers and interviewees are seen as actors of a theatrical performance, i. e., the qualitative interview. Each actor influences the quality of the interview, helping the interviewer to elicit relevant information.
[4] As described in [Myers and Newman, 2007], some of the most relevant problems that can arise during an interview are (a) the artificiality of the interview, (b) the lack of trust and time, (c) the elite bias, i. e., the interviewer may change the importance attributed to the responses according to the role played by the interviewee, (d) the Hawthorne effects, i. e., the interviewer could interfere with interviewees' behaviors, (e) the language ambiguity, (f) the failure of the interview, and (g) the building of knowledge .

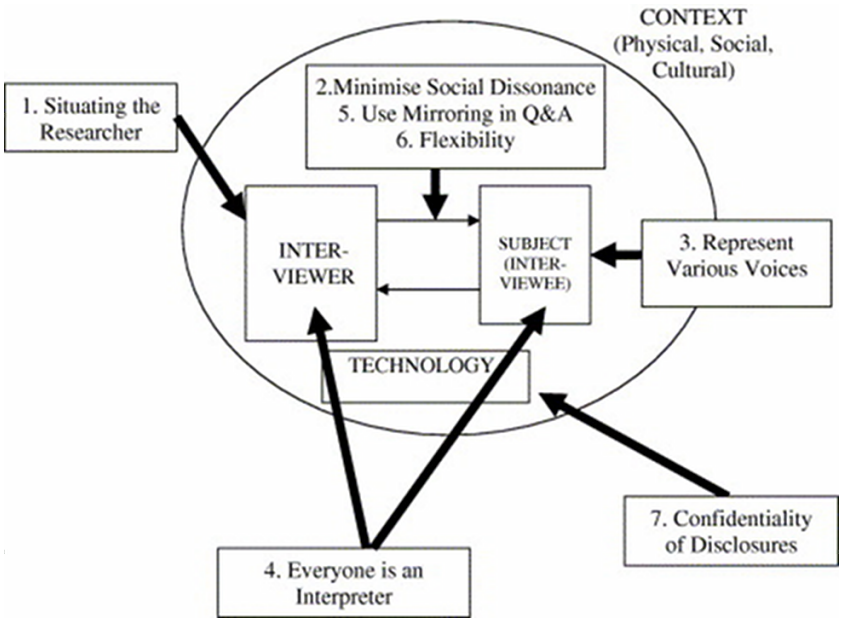


Figure 8.1.: The Myers and Newman guidelines for the qualitative research interview (from [Myers and Newman, 2007])

8.3. Identifying the Inpatient's Adverse Events

The interview sessions have been undertook within the hospital from March 2012 to May 2012 and involved four employees of the hospital, which are characterized by a high level of experience in their work activities and in the use of the EPR adopted within the hospital. In particular, the first two interviewees are the director of the unit for the management of information systems and the head of nursing. Subsequently, they suggested to interview two senior nurses. In particular, one of the interviewed nurses is a member of the group for prevention and management of falls within the hospital. Figure 8.2 shows some of the questions that have been administered to the interviewees.

The four interviewees were highly interested and willing to undergo to the interview, since they were conscious of the benefits obtainable from the results of this study for both patients and clinicians. The first outcome of the interviews was that the interviewed clinicians are generally satisfied of the EPR they use in their daily activities. In particular, clinicians expressed their favour with respect to the possibility to have access to the whole inpatient's clinical history, which can be kept constantly updated and made available every time it is needed (through the use of mobile devices). Moreover, clinicians expressed their appreciation for the better management of their care activities, reducing the risk of misunderstanding with both their colleagues and the doctors.

Focusing on the identification of the possible adverse events that can occur to the inpatients during their hospital stay, clinicians outlined some relevant scenarios, which range from the risk of malnutrition, infection or decubitus ulcers up to the occurrence of catastrophic events. Notwithstanding the wide range of possible adverse events, most part of the interviewed clinicians focused on the inpatients' fall risk. This is partially motivated by the fact that a statistical study conducted within the *Presidio Ospedaliero di Vimercate* has shown that falls involve more than the 1% of the inpatients (value to be considered extremely significant, considering the size of this hospital). Moreover, the Italian Health Ministry included the fall risk in the set of sentinel events[5], which is called "death or serious damage for the inpatient's fall"[6], showing that this topic is considered relevant also at the national level.

In this light, since clinicians consider crucial to be able to avoid the occurrence of this adverse event, the study has been focused on the elicitation of the pre-conditions that lead to the fall of an inpatient.

8.3.1. The Inpatient's Fall Risk

The analysis of the answers given by clinicians during the interview sessions has shown that their EPR does not encompass any kind of tool to help them in preventing the inpatients' fall. Moreover, from this analysis it has been possible to understand that determining the possibility of an inpatient's fall requires clinicians a lot of efforts to consider a wide range of factors, which are characterized by a heterogeneous nature (e. g., subjective perceptions of inpatients and objective measurements and observations performed by clinicians):

Conley scale The Conley is an ordinal scale, which is composed of six factors to which is assigned a score and indicates the presence of the fall risk, if its overall score is greater than the threshold value of two points out of ten (see [Conley et al., 1999]). In particular the factors influencing the Conley score are: (1) other fall events in the past three months, (2) vertigo or dizziness in the past three months, (3) incontinence or loss of faeces in the past three months, (4) impairment of the

[5] A sentinel event represents a particular type of serious adverse event that is sufficient if occurs even once to make necessary to conduct an investigation with the aim to make explicit the triggering factors, in order to implement the appropriate corrective actions.

[6] The third report on the Monitoring Protocol of Sentinel Events shows that, in the period between the years 2005 and 2010, the inpatient' fall is the second type of sentinel events that hospitals signalled to the Italian Health Ministry (16.8% of the signalled events). See `http://www.salute.gov.it/imgs/C_17_pubblicazioni_1642_allegato.pdf`. Accessed: 2012-10-15. (Archived by WebCite® at `http://www.webcitation.org/6BQvrQOUb`) - in Italian.

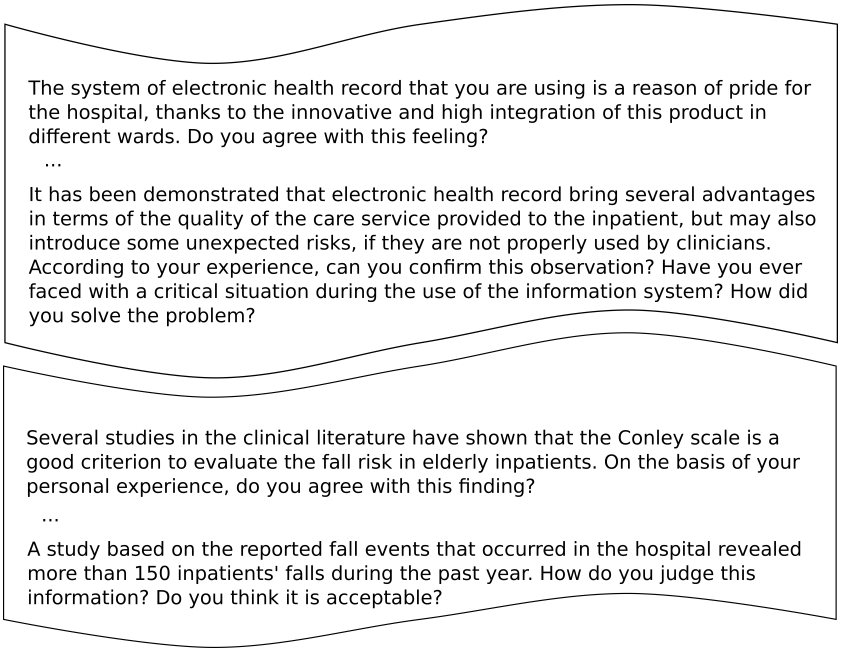


Figure 8.2.: Some questions that have been administered to clinicians during the semi-structured interviews

walk capabilities, (5) anxiety, and (6) loss of wisdom. Even if clinicians trust this scale, they consider suitable the evaluation of other risk factors, in the case the Conley score of the inpatient is less than two.

Vital parameters In some cases, clinicians reported experiences of inpatients' falls even if they were not considered in a risky conditions, but they presented significant deviations in some of their vital parameters with respect to the reference values. For this reason, clinicians indicated the possibility that vital parameters can have some influence on the inpatient's fall risk (Table 8.1 summarizes the set of vital parameters that clinicians indicate as the crucial ones). In particular, the *Braden* scale evaluates the influence of decubitus ulcers on the fall risk (see [Bergstrom et al., 1987]); on the other hand, the *VAS* scale provide a similar evaluation with respect to the subjective perception of pain by the inpatient (see [Hayes and Patterson, 1921; Aitken, 1969]).

Pathologies Clinicians consider some pathologies (e. g., heart and neurological diseases, disorders of the musculoskeletal apparatus and gastrointestinal) an additional indicator of the fall risk.

Pharmacotherapy The number and the type of drugs (e. g., sedatives, laxatives and

Vital Parameter	Reference Values	Threshold
Body temperature	32 – 42	> 38
Braden scale	0 – 20	< 14
DTX-glucose	20 – 800	`avg`[a] + 100 OR `avg`[a] − 50
Minimum arterial pressure	40 – 165	`avg`[a] + 20 OR `avg`[a] − 20
VAS scale	0 – 10	$>= 4$

[a] Average reference values.

Table 8.1.: The set of inpatients' vital parameters

cardiovascular drugs) administered to the patient may have repercussions on the possibility that she can fall. Some of the interviewed clinicians indicate a value between three and four for the number of drugs administered at the same time in order to consider the inpatient in a risky situation.

Other factors Clinicians indicate also *old age* (i. e., a more than eighty years old inpatient), the *use of walking aids* or the *postoperative conditions* as pre-conditions that can affect the inpatient's fall risk level.

The interviewed clinicians indicated all these risk factors with reference to more than sixty-five years old inpatients. However, in the case of over eighty years old inpatients, clinicians indicated this condition as a high risk factor of fall.

Starting from the elicited risk factors and working closely with clinicians, a set of pre-conditions that can lead the inpatient to a fall event has been defined. This set of pre-conditions allowed to define a set of rules, with the aim to promote among clinicians the awareness that a fall event can disrupt the safety of an inpatient. In particular, according to the severity of the risk that is expressed by the pre-conditions, clinicians commonly agreed in defining three fall risk levels, i. e., 'high', 'medium' and 'low'. Table 8.2 summarizes these rules, grouping them according to the related risk level. This set of rules has been used to validate the expressiveness of the WOAD Visual Language (see Section 6.2.1), as described in the next section.

8.4. WOAD Mechanisms and the Inpatient's Fall Risk

Since one of the purposes of the described qualitative user study was the validation of WOAD Visual Language expressiveness, it was necessary to give clinicians the possibility to focus on the task of creating the set of rules to convey a suitable API, i. e., the *Safety API*, to make clinicians constantly aware of a risky condition of an inpatient

Rule Name	LHS	RHS
Conley Result	Conley >= 2 Age >= 65	Safety API (High Risk)
Old Age	Age >= 80	
Polytherapy	Number of Drugs >= 4 Age >= 65	
Critical Diseases	∃ Critical Disease [a] Age >= 65	Safety API (Medium Risk)
Critical Drugs	∃ Critical Drug [b] Age >= 65	
Deambulation Aid	∃ Deambulation Aid	
Surgeries	∃ Recent Surgery Age >= 65	
Body Temperature	Temperature > 38 Age >= 65	Safety API (Low Risk)
DTX-glucose	DTX >= `avg` [c] + 100 OR DTX <= `avg` [c] − 50 *Age* >= 65	
Minimal Arterial Blood Pressure	Pressure >= `avg` [c] + 20 OR Pressure <= `avg` [c] − 20 Age >= 65	
Scales	VAS >= 4 Braden < 14 Age >= 65	

[a] E.g., cardiovascular, cerebral and musculoskeletal diseases. [b] E.g., sedative and cardiovascular drugs.
[c] Average reference values.

Table 8.2.: The set of rules to check the inpatients' fall risk factors

(see Section 1.4.1 for more details about APIs). In the particular case, the *Safety API* has been configured to convey its awareness information through an exclamation mark icon, the color of which ranges from green (low risk), through orange (medium risk) to blue (high risk).

To this aim, clinicians have been provided with a simple dashboard (see Figure 8.3) that summarizes all the identified fall risk indicators (that represent the elicited risk factors). The choice to build a dashboard is motivated by the need of clinicians to be able to have a complete overview of the inpatient situation at a glance that emerged from the interview sessions. This dashboard has been built like any other document of the WOAD framework, i. e., creating a document template using the WOAD Template Editor (see Section 6.1). To reach this goal, the first activity has been to create a set of Datoms representing and grouping the various fall risk indicators (e. g., the Conley score or the set of the vital parameters). Moreover, the related Didgets was populated with data extracted from the official EPR of the hospital. Moreover, the choice to do not ask clinicians to create the dashboard was made with the aim to avoid to overload

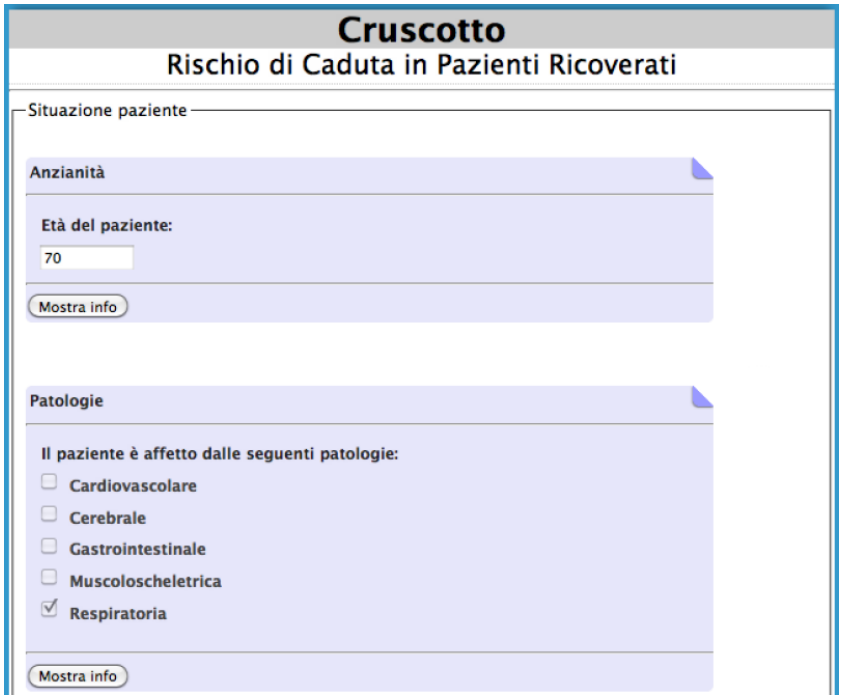
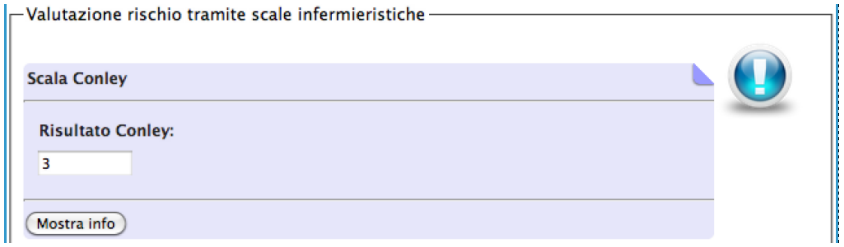


Figure 8.3.: The inpatient's fall risk dashboard WOAD document (in Italian)

clinicians with the additional, even though minimal, effort that is needed to learn to use the WOAD Template Editor.

Once clinicians have been provided with the dashboard, some observation sessions have been undertaken while they performed the task of creating the WOAD mechanisms to convey the *Safety API* near to the dashboard indicators that verify the elicited fall risk pre-conditions. In order to reach this goal, a preliminary training session has been administered to clinicians in order to briefly illustrate them the basic concepts concerning the user interface of the WOAD Mechanism Editor and the composition of the blocks of the WOAD Visual Language. Subsequently, clinicians were asked to create the WOAD mechanisms they need using the WOAD Visual Language. The feedbacks obtained during these observation sessions were for the most part positive, since clinicians did not present particular problems in building the WOAD mechanisms. This result allowed to assert that the WOAD Visual Language can be considered sufficiently expressive and intuitive to support clinicians in the task of defining the WOAD mechanisms they need to support the provision of suitable awareness information to their colleagues. In particular, according to the rules summarized in Table 8.2, clinicians autonomously composed some of the related WOAD mechanisms. Figure 8.4 shows the WOAD mechanisms that clinicians created to monitor the Conley score of a over sixty-five years old inpatient (see Section 8.3.1), with

(a)



(b)

Figure 8.4.: The Conley WOAD mechanism and the *Safety API* conveyed through the Fall Risk Dashboard (in Italian)

the aim to convey a *Safety API* for a high fall risk level (i. e., according to the *Safety API* configuration, a blue exclamation mark icon), if the value of the Conley score is greater than 2 (i. e., Conley score $\geq$ 2). On the other hand, Figure 8.5 shows the set of two WOAD mechanisms pertaining the DTX-glucose rule: in this particular case, since the rule encompasses a disjunctive condition (i. e., `OR`), clinicians created two distinct mechanisms to check if the inpatient's glucose exceeded an upper or a lower threshold and, if one of these conditions is verified, the related WOAD mechanism will be triggered to convey a *Safety API* for a low fall risk level.

The collaborative rule definition raised an interacting discussion about the possibility that multiple rules can be active at the same time, and on what might be the best strategy to convey awareness in these cases. Within rule-based systems, this kind of situations can be solved through the adoption of the *conflict resolution* techniques[7], changing the system

[7] Rule-based systems can perform different strategies to solve rules conflicts: (a) executing the first rule in which all conditions are verified (with the risk of an infinite loop on this rule), (b) executing a random choice, (c) executing the rule with the highest number of conditions, (d) executing the rule that has been used more recently, and (e) executing the rule with the highest salience.

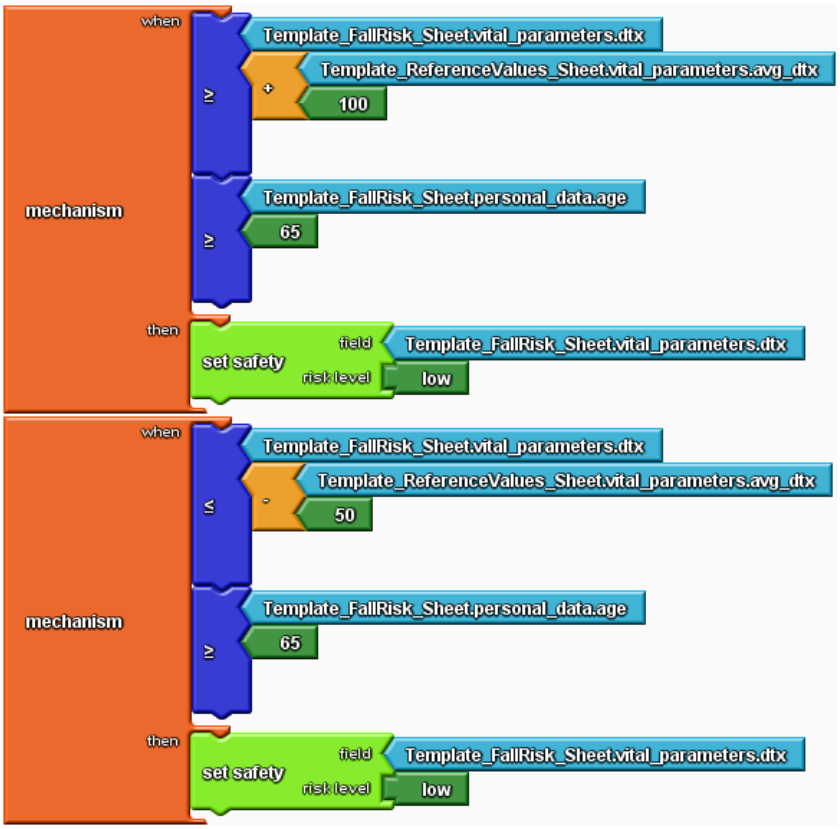


Figure 8.5.: The DTX-glucose WOAD mechanism

behavior by managing the activation of the rules. On the other hand, since WOAD mechanisms have been conceived only to convey suitable APIs without performing any kind of inference, the default behavior of the WOAD framework is to make evident this situation showing all the suitable cues associated to each *right-hand side* in order to make the different interpretations visible and let the users solve the conflict on the basis of their experience and competences. This default behavior was discussed with the clinicians and they preferred to look for a solution more focused on the case at hand. First of all, clinicians proposed to select only the rule that presents the higher risk level. However, this solution provides only a first approximation to the resolution of this kind of situations, since, the validation of this strategy revealed a borderline situation, which can be summarized with the question "if all the active rules pertain either to the 'medium' and to the 'low' risk, which is the overall risk level? The overall risk level is 'medium' or higher?". This borderline situation is more evident if active rules pertain to the same risk level (i. e., "if all the active rules pertain to the 'medium risk' category, the overall risk is still 'medium' or it becomes 'high'?"). The solution to this type of problem was found in

agreement with the clinicians, who proposed to define a new rule that groups in its *left hand side* (i. e., the rule's *when-part*) all the conditions of the rules that are active at the same time; moreover, the *right hand side* of this rule (i. e., the *then-part*) would convey to clinicians an awareness information concerning a risk level that is greater than the highest risk of conflicting rules (e. g., if the active rules pertain either to the 'medium' or to the 'low' risk, the new rule would pertain to the 'high' risk). In the head of clinicians, this particular solution had the advantage to consider the cumulative effects of several risk symptoms with respect to take into account one by one rules that are active at the same time, thus making clinicians aware about the true fall risk level of the monitored inpatient.

9 Validating the WOAD Mechanism Editor

Contents

In the light of the positive results of the qualitative validation of the WOAD Visual Language, which have been described in the previous chapter, a second user study has been performed with the aim to validate the usability and the user-friendliness of the WOAD Mechanism Editor, in particular focusing on the intuitiveness of the user interface. To this aim, a quantitative user study[1] has been performed within the same hospital where the user study described in the previous chapter took place, with the aim to understand if end-users with no particular programming skills could be able to autonomously use the editor, without any particular training on how to use it.

To this aim, a set of students and post-graduate residents that work within the hospital have been enrolled in this user study. They were used as "proxy users" [Friedman and Wyatt, 2006] for the WOAD Mechanism Editor. Proxy users are users that sufficiently represent the typical users of the system being tested. Enrolling proxy users is useful in those situations in which it is difficult to involve the actual users directly, in this case doctors and nurses actually working in a hospital setting full time. In particular, this quantitative user study required clinicians to autonomously use the WOAD Mechanism Editor to compose three WOAD mechanisms each one corresponding to a real-life

[1] The user study will be discussed in a chapter of the book "Emerging Research and Trends in Interactivity and the Human-Computer Interface" edited by Katherine Blashki and Pedro Isaías that will be published by IGI Global in 2013.

conventional rule, which has been discussed with clinicians during other field studies in the healthcare domain (e. g., see [Cabitza et al., 2009a; Cabitza and Simone, 2012]).

This chapter will deal with the description of this quantitative user study and will discuss the obtained results. First of all, the chapter will go through the description of the activities that were aimed at designing the user study. Subsequently, the chapter will describe the procedure that has been adopted to execute the test. Finally, the last section will summarize and discuss the obtained results.

9.1. Designing the User Study

Drawing upon the general structure of empirical studies that has been proposed in [Perry et al., 2000], the first activity that has been undertaken was the design of the user study, and the result has been a four-step structure: (i) formulation and refinement of the hypothesis to be tested; (ii) gathering of participants (i. e., clinicians) who claimed to be interested in attending the study, random separation of them in two groups and, finally, execution of the testing sessions; (iii) systematic analysis of the collected data through the calculation of some statistics; and, finally, (iv) on the basis of the results of statistics and the related analysis, drawing of some remarks about the possibility to reject or not the initial hypothesis.

The hypothesis under test was that «the participants that received the training session (i. e., the U_1 group) and those that did not receive such a training (i. e., the U_2 group) would take an equal time, on average, to perform each task» (i. e., $H_0 : \mu T_{U_1} = \mu T_{U_2}$). In this light, the aim of the study was to collect sufficient evidence to determine whether H_0 could be either rejected or, conversely, confirmed. The latter case would prove that untrained users can use the WOAD Mechanism Editor as much skilfully as users that received a specific and effective training on its usage.

To this aim, the study has been designed to ask users to perform three tasks with an increasing level of complexity. Nevertheless, tasks were structured to involve no more than two documents:

Task 1 This task required users to check if the weight of the newborn is included within the range of reference values (i. e., $2500g \leq$ Newborn Weight $\leq 4500g$), and otherwise to convey the awareness information of a critical situation. The task involved only one document that is called 'Objective Examination' and contains the Didget pertaining to the '`newborn_summary`' Datom). Since the condition can be expressed as a disjunctive boolean expression, i. e., `newborn_summary.weight` $< 2500g$ OR `newborn_summary.weight` $> 4500g$, users were asked to compose two distinct

WOAD mechanisms: the former to check if `newborn_summary.weight` $< 2500g$ and the latter to check if `newborn_summary.weight` $> 4500g$; both mechanisms, if activated, would have to convey a *Criticality API*, which has been defined to show a red border around the `weight` data field in the `newborn_summary` Didget.

Task 2 This second task required users to check if the newborn has at least a malformation, and in this case to convey awareness information to make evident that there is a medium risk for the safety of the newborn. Even in this case, users were asked to compose a WOAD mechanism that only involved a single document: the 'Familiar Medical History'. The related document template contains a Didget that pertains to the '`malformations`' Datom, which represents a check list of the most common malformations. Moreover, this task required users to use the '`count`' aggregator. Users were asked to compose a WOAD mechanism that counts the number of checked malformations in the `malformations` Didget and, if the resulting value is greater than zero (i. e., `count(malformations.*)` > 0), to convey a *Safety API* near to the first data field of this Didget. In order to convey the *Safety API* for a medium risk, users were asked to specify the 'medium' value for the 'risk level' parameter of the *Safety API*.

Task 3 The last task required users to check if a drug that they prescribed to an inpatient is in the official list of drugs of the hospital pharmacy, and otherwise to convey the awareness information to make their colleagues aware of the need of reviewing the name of the prescribed drug. This task involved two distinct documents: the 'Prescription Sheet', which contains the drugs that doctors prescribed to an inpatient, and the 'Drug List', which contains the list of drugs of the hospital pharmacy. For the sake of simplicity, the 'Prescription Sheet' has been defined to contain four distinct Didget pertaining to the '`drug`' Datom, while the 'Prescription Sheet' is composed of a single Didget pertaining to the '`drug_list`' Datom, which has been configured as "multiple" (see Section 3.2). The task required users to compose a WOAD mechanism that uses the '`in`' aggregator to check that the name of the first drug is not in the list of drugs, and accordingly to convey a *Revision API* near to wrong drug name in the 'Prescription Sheet'.

Moreover, the choice to ask clinicians to perform three distinct tasks allows to have a first understanding on the existence of a self-teaching process triggered by the user of the WOAD Mechanism Editor. In other words, if users can easily acquire skills while using the editor.

9.2. Performing the User Study

After the initial phase concerning the design of the study, the next phase has been the enrollment of the participants (i. e., clinicians). Once the number of participants that have been gathered was considered sufficient to have a significant sample, testing sessions were started. This stage of the study has been articulated in four further steps. First of all, a preliminary questionnaire has been administered to participants with the aim to get some information about them, like their education level, their work experience and their self-assessed IT skills. Subsequently, all participants underwent a short talk that provided them with a brief outline of the scope and the aim of the test session.

After completing this initial phase, in which all participants received the same information, they have been randomly divided in two evenly-matched groups. Only one of the two groups of participants received a comprehensive training session (i. e., the U_1 group). Conversely, the other group did not receive any training, as it was supposed to be the "control group" (i. e., the U_2 group). The training session consisted in showing to participants a short movie that illustrates how to use the WOAD Mechanism Editor and how to solve simple tasks by creating a couple of pertinent WOAD mechanisms. At the end of this training session, participants were asked to fill in a second questionnaire to assess the perceived satisfaction with the training that they received in terms of efficacy.

Once all the preliminary phases were completed, it has been possible to start the actual test session, where participants were asked to perform the three tasks that have been described in Section 9.1. Each participant was timed while she was performing each one of three tasks. In order to keep the maximum objectiveness of the measurements, during the execution of tasks the participants were not influenced in any way.

Finally, at the end of the test session, participants were asked to fill in a short final questionnaire with the aim to collect an informal feedback "on the spot", in terms of the evaluation of their overall satisfaction in using WOAD Mechanism Editor, the usability of its user interface and the intuitiveness of the WOAD Visual Language, as well as comments and suggestions.

9.3. Discussing the Results

At the end of the test session, the collected data has been organized and analyzed in order to try to validate the hypothesis under test. A total of 34 participants were enrolled in this study. The enrolled participants were students of both medicine and nursing as well as post-graduate residents. The enrollment criteria required participants to have

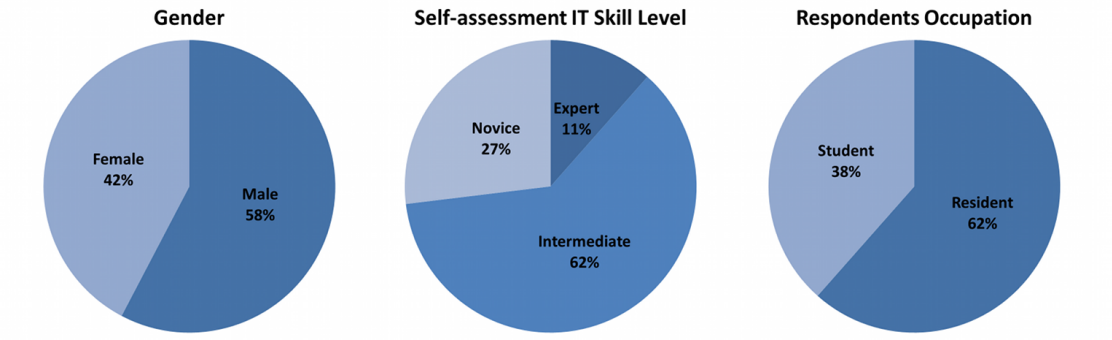


Figure 9.1.: The main characteristics of the enrolled clinicians

some experience with computers, but no particular skill in either programming or visual programming editors. For this reason, four potential subjects, who declared to have very good programming skills, were discarded from the initial sample of 38 participants. Participants were between 25 and 32 year old, and only 14 of them (42%) were female (see Figure 9.1); 21 of them (62%) were already graduated. Most of the participants defined themselves as intermediate computer users, while 9 of the other defined themselves as beginners (27%) and only 4 as experts (11%); moreover, 8 of them (24%) declared to have some experiences in computer programming, but only 2 of them (6%) declared to have had previous experiences with visual editing environments. Only a small set of 6 participants were workers (19%) with an average job experience of 2.5 years. Since the participants have been splitted into two groups it was fundamental that these groups were homogeneous with respect to the characteristics of the participants. To this aim, a Chi-squared test has been performed and allowed to state that the two groups were equal with respect to gender, IT skills and occupation.

Of the 34 participants, only three of them dropped out from the not trained group before completion. This results in a final sample of 31 participants. Moreover, at the satisfaction questionnaire on the effectiveness of the training session, 12 out of 17 participants found the training session at least effective ("effective" or "very effective"): notably, only these 12 participants were considered for the execution step so as to consider only those that thought to have been benefitted from receiving the training and therefore maximize the differences between the two groups. This skimming was aimed at discarding participants who could perform as if they did not receive the training just because they could not exploit the training as an advantage for their performance. Moreover, only those participants that could accomplish the tasks correctly or, at least, with minor mistakes have been considered. For this reason, those participants that, irrespective of the training received, did a bad performance of the tasks have not been considered.

	Groups	N	Mean (seconds)	Standard Deviation	Mean Std. Error	p-Value [a]
Task 1	Trained – T_{U_1}	12	391.00	173.41	48.10	0.077
	Not trained – T_{U_2}	14	510.92	157.28	43.62	
Task 2	Trained – T_{U_1}	12	280.31	128.62	35.67	0.658
	Not trained – T_{U_2}	14	260.15	99.03	27.47	
Task 3	Trained – T_{U_1}	12	295.08	87.66	24.31	0.380
	Not trained – T_{U_2}	14	257.69	122.75	35.05	

[a] Student's T-test for the equality of means of independent samples.

Table 9.1.: Statistical comparisons between the "trained" group (i. e., the U_1 group) and the "not trained" group (i. e., the U_2 group)

Figure 9.2 shows the chart of the mean time of the two groups for each task. Except for what concerns *Task 1*, in which the U_1 group has a better mean time than the U_2 group, in *Task 2* and *Task 3*, the U_2 group has a slightly better mean time than the U_1 group. Moreover, Table 9.1 reports the final results of the test.

In order to understand if the initial hypothesis should be rejected or not, the collected data have been used to compute some statistics. In particular, collected data has been used to perform the Student's T-test for the equality of means of independent samples. The preliminary activity has been to set the significance level of the test at the conventional threshold of the probability to discard the initial hypothesis (i. e., H_0) if this is actually true (i. e., $alpha = 0.05$). The idea was to confirm the initial hypothesis to be true (as initially assumed) if and only if the probability of observing data at least as extreme as that observed is below 5%. According to the time measures that have been collected during the execution of the test, and hence to the so called "mean time to performance" (in seconds) for each task, it is not possible to reject either $H_{0_{Task\ 1}}$, $H_{0_{Task\ 2}}$ or $H_{0_{Task\ 3}}$ (see $p - Value > 0.05$ for all three tasks in Table 9.1). Therefore, H_0 can be considered true for all three tasks, allowing to claim that the WOAD Mechanism Editor does not require specific skills to support realistic tasks of computational augmentation of regular clinical documents.

In regard to the post-test evaluation in the final questionnaire, the results have been analized focusing on the subjective assessment of the overall value of the user experience, of the user-friendliness of the graphical user interface, and on the extent the visual language was found intuitive, i. e., easy to comprehend and apply to the requested cognitive tasks. These assessments were solicited asking the participants to choose one possible value in an ordinal scale from 1 to 4, with explicit anchors being "very low", and "very high". Since the middle option has been purposely avoided to limit

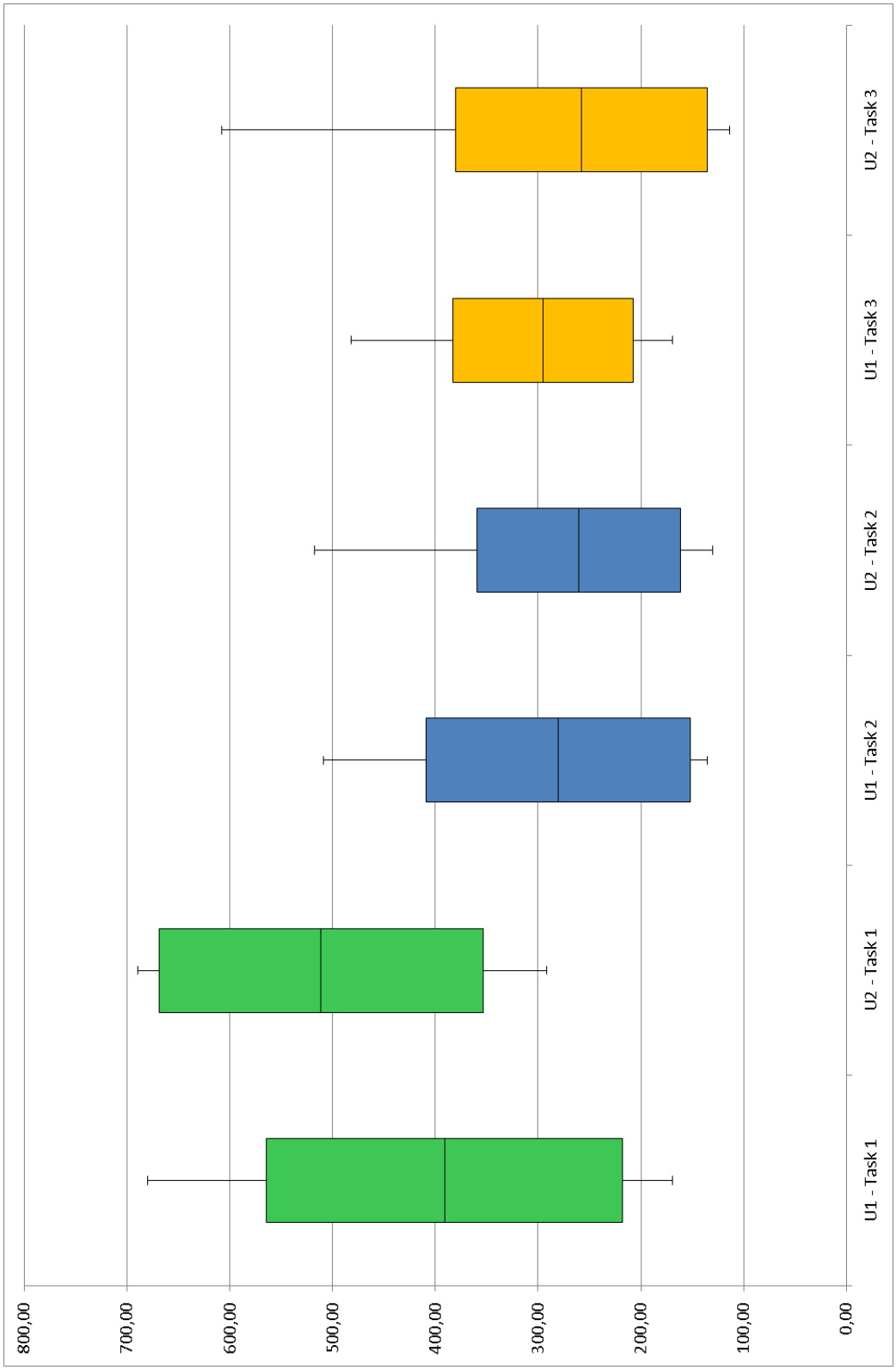


Figure 9.2.: The boxplot chart of the mean times of performance of the two groups of participants: the "trained" group (i. e., U_1) and the "not trained" group (i. e., U_2)

central tendency bias, participants have been provided with the opportunity to select a "don't know option". In Figure 9.3, the corresponding results are shown. Calculated means, which have been reported for descriptive (i. e., not inference) purposes only, are respectively: Overall value: 2.78 (SEM[2] 0.13); GUI user-friendliness: 2.57 (SEM 0.14); visual language intuitiveness: 2.63 (SEM 0.17). This can be taken as an informal indication that respondents found the editor usable and the visual language intuitive enough for their purposes (since the mean was always greater than 2.5). A two sample Kolmogorov-Smirnov Test[3] provided insufficient evidence to reject the assumption that the median assessments from the two distinct groups were not uniform (Overall value: $Z = 0.53, p - Value = 0.94$; GUI user-friendliness: $Z = 1.1, p - Value = 0.18$; visual language intuitiveness: $Z = 1.24, p - Value = 0.09$), that is it is possible to claim that user satisfaction did not depend on having received the training or not (which was nevertheless considered effective by whom received it).

Summarizing, the statistical results that have been discussed above gave a preliminary positive contribution to suffrage the initial hypothesis that the WOAD Mechanism Editor can be profitably used also without a specific (and effective) training session. In other words, the WOAD Mechanism Editor does not require specific IT skills to be used by domain-expert users.

[2] Standard Error of the Mean.[3] The Kolmogorov-Smirnov Test has been performed with SPSS v. 17.0.

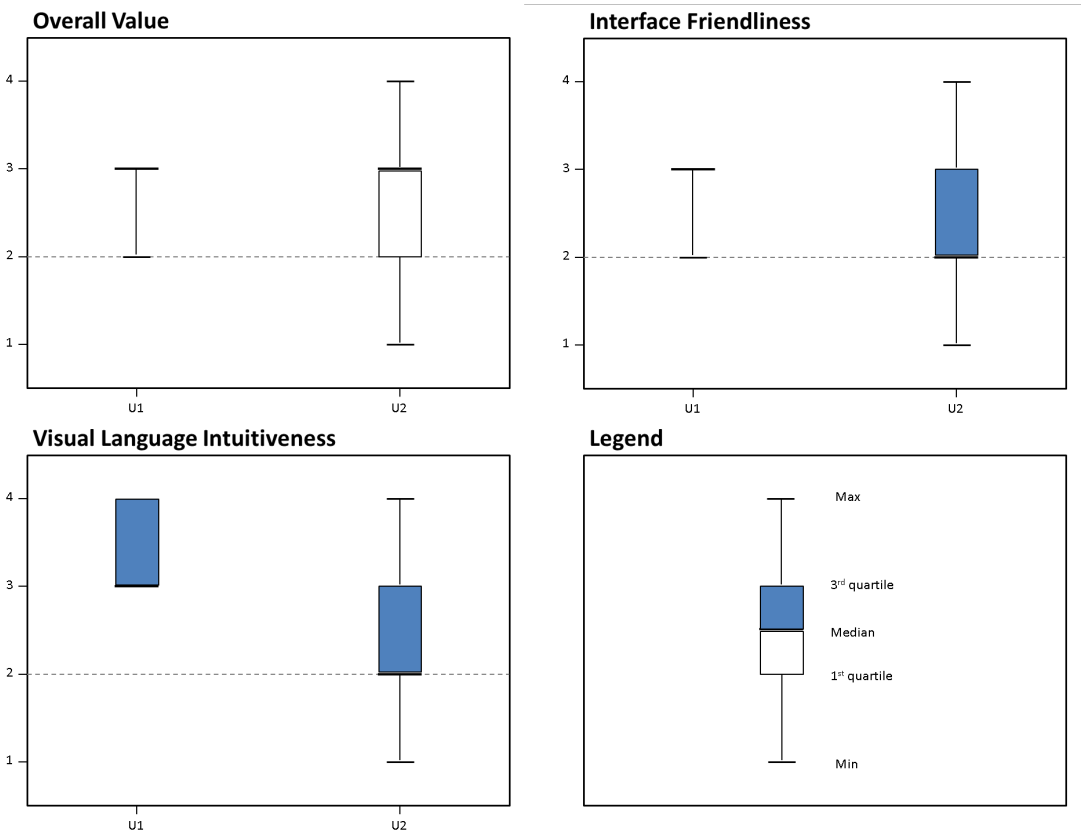


Figure 9.3.: The boxplot charts of the final evaluations of the two groups of participants (i. e., the "trained" group, U_1, and the "not trained" group, U_2)

10 Conclusions

This thesis focused on the need of end-users to be autonomous in tailoring the document-based systems that they use to perform their collaborative work activities. In particular, the leitmotif of this work is to allow users of WOAD-compliant systems to perform their tailoring activities through a visual approach that does not impose them to learn technical skills specific of IT professionals.

The starting point of this work was that the systems compliant with the earlier definition of the WOAD framework presented some weaknesses that risked to strongly limit its applicability (see Chapter 3 and Chapter 4). First of all, WOAD-compliant systems still required the presence of IT professionals to perform the task of defining the WOAD mechanisms, i. e., rules, that users need to couple to their digital documents with the aim to convey suitable *Awareness Promoting Information (API)*, in order to support and promote *collaboration awareness* among their colleagues on the basis of their local conventions. Since users are the only ones who have the full knowledge of these conventions and their continuous evolution, it is crucial to support them in autonomously performing the task of composing the WOAD mechanisms that they need. Moreover, several studies in the healthcare domain (e. g., [Cabitza et al., 2009b; Morrison and Blackwell, 2009; Chen and Akay, 2011]) showed that the users need to autonomously customize the documents that they utilize to perform their daily work activities, as their are accustomed to do with traditional paper-based documents. In particular, paper-based documents allow users not only to change the appearance of documents, but also to modify the informative content of documents according to their current needs. On the other hand, the systems based on digital documents, e. g., traditional EPRs, force users to switch to rigid user interfaces with structure and informative content that differ from those of the paper-based counterparts, without any possibility to be adapted. Even if

the WOAD framework yet copes with this need of flexibility, since WOAD documents have been conceived to rely on a modular approach, it lacked in supporting users in autonomously composing their document templates in a simple manner.

The two weaknesses of the WOAD framework described above and the findings of a field study in the healthcare domain (see [Cabitza et al., 2009a]) suggested to adopt the tenets of the EUD research field to provide a suitable answer to the users' need of autonomy. In this light, a bottom-up approach has been adopted to abstract and generalize the outcomes of this analysis, allowing the elicitation of a set of requirements to suitably improve the WOAD framework and to make it EUD-compliant. To this aim, we proposed to adopt a visual approach to allow users to be autonomous in both defining and structuring the informative content of their digital documents as well as in creating their WOAD mechanisms. In order to validate this approach, two prototypical visual editors have been developed, i. e., the WOAD Template Editor and the WOAD Mechanism Editor. Moreover, the latter has been conceived to adopt a block-based visual language. This required to conceive a visual language, i. e., the WOAD Visual Language, whose set of constructs have been defined on the basis of the findings of another field study in the healthcare domain (see [Cabitza and Simone, 2012]).

Since the WOAD Template Editor mimics the WYSIWYG approach of traditional word processors, which it is assumed that users are familiar with, it has not undergone a deep validation. On the other hand, the visual approach of the WOAD Mechanism Editor required to be deeply validated, in order to understand if the building block metaphor was sufficiently user-friendly to make users autonomous in the task of defining WOAD mechanisms. To this aim, two user studies have been conducted in the healthcare domain (see Chapter 8 and Chapter 9). The first user study was focused on validating the expressiveness of the WOAD Visual Language. The involved clinicians did not present relevant problems in composing the WOAD mechanisms that allow them to be aware of an inpatient's risky condition. This made us confident that the WOAD Visual Language can be considered sufficiently expressive and intuitive to support clinicians in the definition of the WOAD mechanisms. The second user study was aimed at validating the usability of the user interface of the WOAD Mechanism Editor: users should be able to use the WOAD Mechanism Editor without any kind of training on its use. Even in this case the obtained results have been positive, supporting the claim that the WOAD Mechanism Editor can be used without requiring users neither to have specific IT skills nor to undergo specific training.

The work presented in this thesis gives a contribution to achieve the goal of designing and implementing technologies supportive of cooperative work that can be easily tailored

by users to support their local conventions and work practices. Empowering users in this way allows them to become an active player in the process of customization and constant adaptation of their collaborative systems, contributing in having timely solutions to extemporaneous problems and needs. Moreover, this involves users in a process of appropriation that concerns the use of the system as well as the internalization of conventions. In this way, the transition from paper-based documents to digital documents can also have positive effects on the improvement of the effectiveness of the daily users' work practices.

Despite the improvements this work made to the WOAD framework towards flexibility and the support for local conventions, the WOAD framework can be further improved to provide end-users with a document-based collaborative system that fulfil their actual needs: these future improvements will be discussed in the next section.

10.1. Future Works

Although the two preliminary validation sessions to which the WOAD Mechanism Editor has undergone gave positive results, the collection of further evidences is needed, in order to corroborate the idea that the adopted visual approach is an effective way to empower end-users in being autonomous in editing their WOAD mechanisms. To this aim, longer and more exhaustive validation sessions are needed. Due to the collaborative nature and the heterogeneity of care activities, the healthcare domain represents an excellent candidate to perform these further validation sessions, even though other collaborative domains could be taken into account.

In the light of providing end-users with a technology supportive of cooperative work that preserves the flexibility of the paper medium, it is possible to follow three different directions towards the further improvement of the WOAD framework: (a) supporting users in collaboratively annotating their documents; (b) improving the support for collaborative tailoring activities; and (c) supporting a wider set of document types.

Supporting users' annotations. The role that annotations play in supporting cooperation is well recognized within the CSCW literature. For instance, [Bringay et al., 2006] described annotations as a fundamental feature to promote collaboration among the users of document-based EPRs. The field studies that have been conducted in the healthcare domain [Cabitza et al., 2009a] further confirmed that clinicians strongly appreciate the possibility to annotate their digital documents as they are already accustomed to do with the paper-based counterparts. Thus, the WOAD framework should be endowed with collaborative annotation capabilities that make

users able to annotate any part of a document (e. g., texts, data fields and images). Annotations could be a free text object as well as a label, i. e., what it is commonly denoted with the term 'tag'. To be more precise, a tag could be, for instance, either a system-defined or a user-defined keyword as well as a date (in a standard format). Moreover, the field studies discussed in [Cabitza et al., 2012] showed that users appreciate the possibility of annotating existing annotations, that is making annotations recursive. This leverages a distributed and asynchronous communication process among users that further improves collaborative practices and promotes knowledge creation and sharing among users. For instance, doctors and nurses can use recursive annotations to discuss the trajectory of an inpatient's illness or the response of the inpatient to a particular drug, contributing in the continuous improvement of their understanding of the inpatient's health problems. Moreover, the results of other field studies in the healthcare domain [Cabitza et al., 2005; Cabitza and Simone, 2008] showed that clinicians often define relationships among documents, contributing in building a *web of artifacts* (see Section 1.2). To this aim, annotations could be leveraged to promote this conventional practice. The idea is to allow users to share annotations among different documents, in order to support them in defining the relationships that involve these documents. For instance, clinicians could use this functionality to make explicit that a piece of information about a treatment is causally related to a specific value of some vital parameters in order to help their colleagues, at a later time, in reconstructing the whole trajectory of the inpatient's illness. This is specifically useful when this relationship is not part of the usual caring process. Finally, since annotations could contain conventional information, the WOAD mechanisms should be able to use these pieces of information in both the *when-part* and the *then-part*. In the first case, users could define WOAD mechanisms to convey APIs whenever specific conditions involving either the content or the existence of annotations are verified. On the other hand, using the information contained in the annotations within the *then-part* allows WOAD mechanisms to change the affordances of the involved annotations, according to the conveyed APIs.

Improving the support for collaborative tailoring. The continuous process of development and refinement of conventions is a collaborative process that involves all the members of the group in which this process takes place. Moreover, as argued in [Rode, 2005], the need to consider the collaborative dimension of the tailoring activities of end-users has been recognized long ago (see [Nardi, 1993]). In this light, since users perform tailoring activities that are related to their local conventions, the

WOAD framework should comply with the fundamental requirement of supporting users in performing tailoring activities in a collaborative manner. With respect to the document templates, this requirement has been partially addressed with the adoption of the Repository application, which is part of the Oryx XForms Editor. The Repository provides users with a shared space where to store document templates; its adoption allowed to endow the WOAD framework with basic collaborative features, like comments and ratings as well as a simple lock strategy to avoid concurrent changes on the same document template (see Section 7.1.1). Nevertheless, the WOAD framework does not support cooperation in the definition of WOAD mechanisms. This situation could be improved in two ways. First of all, the WOAD Mechanism Editor could be fully integrated with the Repository, in order to provide users with an unique storage space for both document templates and the related WOAD mechanisms. On the other hand, the WOAD Mechanism Editor could be further improved allowing users to collaboratively annotate the WOAD mechanisms, in order to promote the communication among users who progressively modify a WOAD mechanism. To complete the set of tools of the WOAD framework supporting tailorability, we notice that users are not autonomous in defining the configuration of the affordances conveying APIs (see Section 7.2.3), requiring them to delegate this task to more ICT experienced people. Thus, users should be empowered in autonomously and collaboratively defining APIs' affordances through a user-friendly editor.

Supporting different kinds of documents. Usually, the term 'document' denotes a textual content. Nevertheless, the role of document can be also assumed by other 'objects' (e. g., audio and video contents). In the healthcare domain, for instance, a radiographic image is considered as a document. Moreover, to perform their care activities clinicians also adopt charts, which summarize data in a graphic manner, and process maps, i. e., the *Clinical Pathways* (e. g., see [Bleser et al., 2006]), which support them in performing care activities by following a commonly agreed set of steps. Within the WOAD framework, documents are forms that users fill in with data during their work activities. In this light, the WOAD framework should be improved enlarging the range of the supported types of documents in order to include images, charts and process maps: users should be able to add annotations to these kinds of documents and to use the related information within their WOAD mechanisms. Moreover, the WOAD framework should allow users to autonomously create their own process maps. To this aim, it can be useful to take into account the possibility of customizing an existing plug-in for the Oryx XForms Editor in

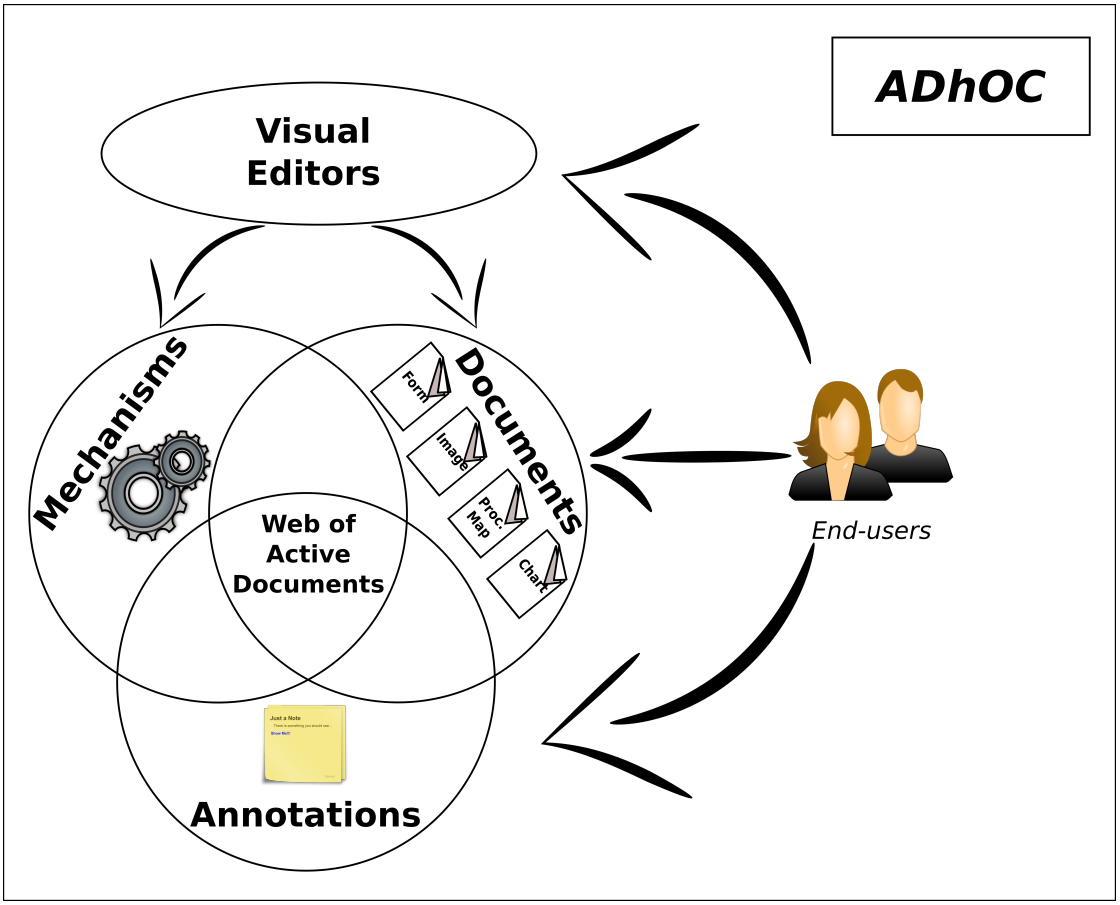


Figure 10.1.: ADhOC, the future WOAD-compliant prototypical environment with the support for annotations and multimedia documents

order to allow end-users to edit process maps that they need in a visual manner.

These further improvements will lead the WOAD framework towards providing end-users with an even more effective, flexible and comprehensive support for their document-based cooperative work. The idea of integrating the envisioned new features within the WOAD framework concurs in tracing the guidelines towards ADhOC, i. e., the acronym for *Active Document with Human inside*, which is a new WOAD-compliant prototypical environment (see Figure 10.1) that is currently under development and that this thesis contributed to construct.

Appendices

A

The MIT OpenBlocks Grammars

This appendix presents the two DTD grammars through which the MIT OpenBlocks library validates its configuration file, which encompasses the configuration of whole set of language constructs and their configuration as well as the configuration of the visual editing environment, and the serialized code fragments respectively.

A.1. The grammar of the OpenBlocks Language Definition

The most important construct that is defined in this grammar is `BlockGenus` that allows to specify the configuration of a visual language block, including the unique name, the color and the label, as well as the set of connectors. The `BlockConnector` construct allows to specify the features of a block connector (e. g., type and label).

Another important construct is `BlockDrawer` that is responsible to group a set of visual language blocks and to populate the palette of blocks.

The following is the complete code for this grammar:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!--
    Document   : lang_def.dtd
    Description:
        This defines the language and initial workspace setup.
-->


<!ELEMENT BlockLangDef ( BlockConnectorShapes, BlockGenuses, BlockFamilies?,
    BlockDrawerSets?, Pages?, TrashCan?, MiniMap?)>

<!--This defines a mapping between block connector shape type to number-->
<!ELEMENT BlockConnectorShapes (BlockConnectorShape*)>
```

```
<!ELEMENT BlockConnectorShape EMPTY>
<!ATTLIST BlockConnectorShape shape-type CDATA #REQUIRED>
<!ATTLIST BlockConnectorShape shape-number CDATA #REQUIRED>

<!ELEMENT BlockGenuses (BlockGenus*)>
<!--This defines a single block genus-->
<!ELEMENT BlockGenus (description?, BlockConnectors?,Stubs?, Images?,
    LangSpecProperties?)>
<!ATTLIST BlockGenus name CDATA #REQUIRED>
<!ATTLIST BlockGenus initlabel CDATA #REQUIRED>
<!-- the kind of a genus can affect the rendering of a block.  relevant kinds are:
    - command: performs an operation and may take in more than one input
    - data: returns primitive values such as number, string, boolean
    - function: takes in an input and performs an operation to produce an ouput
-->
<!ATTLIST BlockGenus kind CDATA #REQUIRED>
<!ATTLIST BlockGenus color CDATA #REQUIRED>
<!ATTLIST BlockGenus editable-label (yes|no) "no">
<!ATTLIST BlockGenus label-unique (yes|no) "no">
<!ATTLIST BlockGenus is-label-value (yes|no) "no">
<!ATTLIST BlockGenus label-prefix CDATA #IMPLIED>
<!ATTLIST BlockGenus label-suffix CDATA #IMPLIED>
<!ATTLIST BlockGenus page-label-enabled (yes|no) "no">
<!--is-starter and is-terminator only apply to blocks of kind: command -->
<!ATTLIST BlockGenus is-starter (yes|no) "no">
<!ATTLIST BlockGenus is-terminator (yes|no) "no">


<!--This defines a block description and the description of its block arguments-->
<!ELEMENT arg EMPTY>
<!ATTLIST arg n CDATA #REQUIRED name CDATA #IMPLIED>

<!ELEMENT description (text, arg-description*)>
<!ELEMENT text (#PCDATA|note|em|i|br|arg)*>
<!ELEMENT arg-description (#PCDATA)>
<!ATTLIST arg-description n CDATA #REQUIRED name CDATA #REQUIRED>
<!ELEMENT note (#PCDATA|arg|i)*>
<!ELEMENT em (#PCDATA)>
<!ELEMENT i (#PCDATA)>
<!ELEMENT br (#PCDATA)>

<!--BlockConnectors are where blocks get connected-->
<!ELEMENT BlockConnectors (BlockConnector*)>
<!ELEMENT BlockConnector (DefaultArg?)>
<!ATTLIST BlockConnector label CDATA #IMPLIED>
<!ATTLIST BlockConnector label-editable (yes|no) "no">
<!-- Order matters with socket connectors and at most one plug is allowed (no
    multiple return types) -->
<!ATTLIST BlockConnector connector-kind (plug|socket) #REQUIRED>
<!-- for connector-type use the shape-type values specified in block connectors-->
<!ATTLIST BlockConnector connector-type CDATA #REQUIRED>
```

```
<!ATTLIST BlockConnector position-type (single|mirror|bottom) "single">
<!ATTLIST BlockConnector is-expandable (yes|no) "no">

<!ELEMENT DefaultArg EMPTY>
<!ATTLIST DefaultArg genus-name CDATA #REQUIRED>
<!ATTLIST DefaultArg label CDATA #IMPLIED>

<!ELEMENT Stubs (Stub*)>
<!--This defines a stub of a block, so that the block can exist as a single
    entity and have mini-references to it-->
<!ELEMENT Stub (LangSpecProperties)>
<!ATTLIST Stub scope CDATA #IMPLIED>
<!ATTLIST Stub stub-genus (getter|setter|caller|agent|inc) #REQUIRED>

<!-- Defines the images that are drawn on the block itself.
     Note: For now, only one image is enabled and wrap-text and image-editable
         have no effect.
     Note: make sure FileLocation specified is relative to workspace directory -->
<!ELEMENT Images (Image)>
<!ELEMENT Image (FileLocation)>
<!ATTLIST Image wrap-text (yes|no) "no">
<!ATTLIST Image image-editable (yes|no) "no">
<!ATTLIST Image block-location
    (center|east|west|north|south|southeast|southwest|northeast|northwest)
    "center">
<!ATTLIST Image width CDATA #IMPLIED>
<!ATTLIST Image height CDATA #IMPLIED>
<!ELEMENT FileLocation (#PCDATA)>

<!ELEMENT LangSpecProperties (LangSpecProperty*)>
<!ELEMENT LangSpecProperty (#PCDATA)>
<!ATTLIST LangSpecProperty key CDATA #REQUIRED>
<!ATTLIST LangSpecProperty value CDATA #REQUIRED>

<!--This defines a BlockGenus Family-->
<!ELEMENT BlockFamilies (BlockFamily*)>
<!ELEMENT BlockFamily (FamilyMember*)>
<!ELEMENT FamilyMember (#PCDATA)>

<!-- Defines BlockDrawerSets and their Block Drawer content-->
<!ELEMENT BlockDrawerSets (BlockDrawerSet*)>
<!ELEMENT BlockDrawerSet (BlockDrawer*)>
<!ATTLIST BlockDrawerSet type (bar|stack) "bar">
<!ATTLIST BlockDrawerSet name CDATA #REQUIRED>
<!ATTLIST BlockDrawerSet location
    (east|west|north|south|northeast|southeast|southwest|northwest) "west">
<!-- window-per-drawer specifies if each drawer should be its own draggable
    window. otherwise, all the drawers
        are contained within one draggable window and only one drawer can be
            opened at once.
```

```
107         Whether or not the window is draggable depends if drawer-draggable is set
                to "yes." -->
    <!ATTLIST BlockDrawerSet window-per-drawer (yes|no) "yes">
    <!ATTLIST BlockDrawerSet drawer-draggable (yes|no) "yes">
    <!-- the width of all the drawers within this set -->
    <!ATTLIST BlockDrawerSet width CDATA #IMPLIED>
112
    <!--This defines BlockDrawers and their content-->
    <!ELEMENT BlockDrawer ( (BlockGenusMember | Separator | NextLine)* )>
    <!ATTLIST BlockDrawer name CDATA #REQUIRED>
    <!ATTLIST BlockDrawer type (default|factory|page|custom) "default">
117 <!ATTLIST BlockDrawer is-open (yes|no) "no">
    <!ATTLIST BlockDrawer button-color CDATA #REQUIRED>
    <!ELEMENT BlockGenusMember (#PCDATA)>
    <!ELEMENT Separator EMPTY>
    <!ELEMENT NextLine EMPTY>
122
    <!-- Defines Pages dividing the Block Canvas and the optional PageDrawers
        associated with them
         Each Page can have only one PageDrawer.
         For now, every page must have a drawer or no pages can have drawers.
         The block canvas need not contain any pages.  You may choose to have
127      a blank canvas instead of a canvas of pages.
    -->
    <!ELEMENT Pages (Page*)>
    <!--
            drawer-with-page auto generates a new drawer for each new page created
                by a user
132     and creates an empty drawer for each page that does not specify a page drawer
    -->
    <!ATTLIST Pages drawer-with-page (yes|no) "no">
    <!ELEMENT Page (PageDrawer?)>
    <!ATTLIST Page page-name CDATA #REQUIRED>
137 <!ATTLIST Page page-width CDATA #REQUIRED>
    <!ATTLIST Page page-drawer CDATA #IMPLIED>
    <!ATTLIST Page page-color CDATA #IMPLIED>
    <!ATTLIST Page page-shape CDATA #IMPLIED>

142 <!ELEMENT PageDrawer (BlockGenusMember*)>

    <!-- If specified a trash can will appear on the workspace.
            For both of its child elements, a location for the images should be
                specified relative to the working directory.
            The open trash image appears when a user drags a block over the trashcan.
                 The closed trash image is the default
147         image during steady state.
        -->
    <!ELEMENT TrashCan (OpenTrashImage, ClosedTrashImage)>
    <!ELEMENT OpenTrashImage (#PCDATA)>
    <!ELEMENT ClosedTrashImage (#PCDATA)>
152
```

```
  <!-- By default, a minimap will always appear in the upper right corner
          of the block canvas, unless enabled is set to "no."
          -->
  <!ELEMENT MiniMap EMPTY>
157 <!ATTLIST MiniMap enabled (yes|no) "yes">

  <!-- By default, typeblocking will be enabled, such that when the user types onto
      the canvas
          blocks will fly out that match the entered text.
          -->
162 <!ELEMENT Typeblocking EMPTY>
  <!ATTLIST Typeblocking enabled (yes|no) "yes">
```

A.2. The grammar of the OpenBlocks Serialization Format

This grammar specifies how to serialize the configuration of the OpenBlocks canvas area, in order to store the visually defined code fragments. In particular, it defines how to store blocks, their connection among each other and their topological arrangement.

The following is the complete code for this grammar:

```
  <!-- Definition of save XML format for Codeblocks -->
 2
  <?xml version="1.0" encoding="UTF-16"?>

  <!-- Root element -->
  <!ELEMENT CodeBlocks (Pages?, BlockDrawerSets?)>
 7
  <!ELEMENT Pages (Page+)>
  <!-- NOTE: Exclusive saving (where only changed information or information that
            differs from the language definition file is saved) is not
            enabled for pages YET -->
12 <!ATTLIST Pages is-blank-page (yes|no) "no">
  <!ELEMENT Page (PageBlocks?)>
  <!ATTLIST Page page-name CDATA #REQUIRED>
  <!ATTLIST Page page-width CDATA #REQUIRED>
  <!ATTLIST Page page-color CDATA #REQUIRED>
17 <!ATTLIST Page page-drawer CDATA #IMPLIED>

  <!ELEMENT PageBlocks (Blocks*, BlockStub*)>

  <!-- If the Block is a BlockStub, then specify the following -->
22 <!ELEMENT BlockStub (StubParentName, StubParentGenus, Block)>
  <!ELEMENT StubParentName (#PCDATA)>
  <!ELEMENT StubParentGenus (#PCDATA)>

  <!ELEMENT Block (Label?, PageLabel?, Location?, BoxSize?, Collapsed?, Comment?,
      BeforeBlockId?, AfterBlockId?, CompilerErrorMsg?, Plug?, Sockets*,
      BlockStubInfo?, LangSpecProperties?)>
```

```
<!ATTLIST Block id CDATA #REQUIRED>
<!ATTLIST Block genus-name CDATA #REQUIRED>
<!ATTLIST Block has-focus (yes|no) "no">

<!ELEMENT Label (#CDATA)>
<!ELEMENT PageLabel (#CDATA)>
<!ELEMENT CompilerErrorMsg (#CDATA)>

<!-- x, y location within the block's page-->
<!ELEMENT Location (X, Y)>
<!ELEMENT X (#CDATA)>
<!ELEMENT Y (#CDATA)>

<!-- width, height box size -->
<!ELEMENT BoxSize (Width, Height)>
<!ELEMENT Width (#CDATA)>
<!ELEMENT Height (#CDATA)>

<!-- existence of this element implies the block is collapsed -->
<!ELEMENT Collapsed>

<!-- Comment widget associated with a Block -->
<!ELEMENT Comment (Text, Location, Collapsed?)>
<!ELEMENT Text (#PCDATA)>

<!-- Only the Block ID of the block connected at these connectors are
     specified. -->
<!ELEMENT BeforeBlockId (#CDATA)>
<!ELEMENT AfterBlockId (#CDATA)>

<!ELEMENT Plug (BlockConnector)>
<!ELEMENT Sockets (BlockConnector*)>

<!-- NOTE: Exclusive saving is not enabled for Block Connectors YET -->
<!ELEMENT BlockConnector (EMPTY)>
<!ATTLIST BlockConnector label CDATA #IMPLIED>
<!ATTLIST BlockConnector connector-kind (plug|socket) "socket">
<!ATTLIST BlockConnector init-type CDATA #IMPLIED>
<!ATTLIST BlockConnector connector-type CDATA #IMPLIED>
<!ATTLIST BlockConnector con-block-id CDATA #IMPLIED>
<!ATTLIST BlockConnector position-type (single|mirror|bottom) #IMPLIED>
<!ATTLIST BlockConnector is-expandable (yes|no) #IMPLIED>

<!ELEMENT LangSpecProperties (LangSpecProperty*)>
<!ELEMENT LangSpecProperty (#PCDATA)>
<!ATTLIST LangSpecProperty key CDATA #REQUIRED>
<!ATTLIST LangSpecProperty value CDATA #REQUIRED>

<!-- To dynamically add drawers and blocks within them to the workspace
     If a drawer set or drawer already exists with the name specified for
     these widgets within the lang def file, then the loader will
```

```
        simply insert the new drawers or        blocks into the these already
            loaded widgets.   -->

  <!-- Defines BlockDrawerSets and their Block Drawer content-->
  <!ELEMENT BlockDrawerSets (BlockDrawerSet*)>
82 <!ELEMENT BlockDrawerSet (BlockDrawer*)>
  <!ATTLIST BlockDrawerSet type (bar|stack) "bar">
  <!ATTLIST BlockDrawerSet name CDATA #REQUIRED>
  <!ATTLIST BlockDrawerSet location
      (east|west|north|south|northeast|southeast|southwest|northwest) "west">
  <!-- window-per-drawer specifies if each drawer should be its own draggable
      window.  otherwise, all the drawers
87        are contained within one draggable window and only one drawer can be
            opened at once.
          Whether or not the window is draggable depends if drawer-draggable is set
            to "yes." -->
  <!ATTLIST BlockDrawerSet window-per-drawer (yes|no) "yes">
  <!ATTLIST BlockDrawerSet drawer-draggable (yes|no) "yes">

92 <!--This defines BlockDrawers and their content-->
  <!ELEMENT BlockDrawer ( (BlockGenusMember | Separator | NextLine)* )>
  <!ATTLIST BlockDrawer name CDATA #REQUIRED>
  <!ATTLIST BlockDrawer type (default|factory|page|custom) "default">
  <!ATTLIST BlockDrawer is-open (yes|no) "no">
97 <!ATTLIST BlockDrawer button-color CDATA #REQUIRED>
  <!ELEMENT BlockGenusMember (#PCDATA)>
  <!ELEMENT Separator EMPTY>
  <!ELEMENT NextLine EMPTY>
```

The WOAD Intermediate Language XML Schema

This appendix presents the definition of the grammar of the WOAD Intermediate Language. The grammar of the WOAD Intermediate Language has been defined using the *XML Schema Definition (XSD)* syntax. Figure B.1 gives a graphic representation of the constructs of the WOAD Intermediate Language.

The following is the complete code for this grammar:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.maclab.disco.unimib.it/WOADMechanism"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:woad="http://www.maclab.disco.unimib.it/WOADMechanism"
  xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
  xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
  jaxb:version="2.0"
  jaxb:extensionBindingPrefixes="xjc">
    <annotation>
      <appinfo>
        <jaxb:globalBindings>
          <xjc:simple />
        </jaxb:globalBindings>
      </appinfo>
    </annotation>

        <element name="rule" type="woad:rule"/>
        <complexType name="rule">
                <sequence>
                        <element maxOccurs="1" minOccurs="1" name="when"
                            type="woad:when"/>
                        <element maxOccurs="1" minOccurs="1" name="then"
                            type="woad:then"/>
                </sequence>
        </complexType>
        <complexType name="when">
                <sequence>
```

```
                        <element maxOccurs="unbounded" minOccurs="1"
                                name="condition" type="woad:condition" />
                </sequence>
        </complexType>
        <complexType name="operation">
                <sequence>
                        <element maxOccurs="1" minOccurs="1" name="first_operand"
                            type="woad:condition_component"/>
                        <element maxOccurs="1" minOccurs="0"
                            name="second_operand"
                            type="woad:condition_component"/>
                </sequence>
                <attribute name="type" type="woad:operation_type"/>
        </complexType>
        <simpleType name="operation_type">
                <restriction base="string">
                        <enumeration value="+"/>
                        <enumeration value="-"/>
                        <enumeration value="*"/>
                        <enumeration value="/"/>
                        <enumeration value="%"/>
                </restriction>
        </simpleType>
        <complexType name="constant_value">
                <simpleContent>
                        <extension base="string">
                                <attribute default="string" name="type"
                                        type="woad:data_type" />
                        </extension>
                </simpleContent>
        </complexType>
        <simpleType name="data_type">
                <restriction base="string">
                        <enumeration value="string"/>
                        <enumeration value="number"/>
                        <enumeration value="boolean"/>
                </restriction>
        </simpleType>
        <complexType name="condition">
                <sequence>
                        <element name="left_side" type="woad:condition_component"
                            maxOccurs="1" minOccurs="1"/>
                        <element name="right_side"
                            type="woad:condition_component" maxOccurs="1"
                            minOccurs="0"/>
                </sequence>
                <attribute name="type" type="woad:condition_type"/>
        </complexType>
        <simpleType name="condition_type">
                <restriction base="string">
                        <enumeration value="=="/>
```

```
                    <enumeration value="!="/>
                    <enumeration value="&gt;"/>
                    <enumeration value="&gt;="/>
                    <enumeration value="&lt;"/>
                    <enumeration value="&lt;="/>
                    <enumeration value="like"/>
                    <enumeration value="not"/>
            </restriction>
        </simpleType>
        <complexType name="field">
        <annotation>
            <documentation>This element represents a didget data field in a
                specific template. If the @template attribute is omitted or
                its value is *, then the element is referencing a didget data
                field in any template of the web of documents.</documentation>
        </annotation>
        <attribute name="template" type="string" default="*"></attribute>
            <attribute name="didget" type="string" use="required"></attribute>
            <attribute name="name" type="string" use="required"></attribute>
            <attribute name="type" type="woad:data_type"
                use="required"></attribute>
        </complexType>
        <complexType name="condition_component">
            <choice>
                    <element name="constant_value" type="woad:constant_value"
                        />
                    <element name="field" type="woad:field" />
                    <element name="operation" type="woad:operation" />
                    <element name="aggregator" type="woad:aggregator" />
                    <element name="conditions" type="woad:conditions_list" />
            </choice>
        </complexType>
        <complexType name="then">
            <sequence>
                    <element name="action" type="woad:action"
                        maxOccurs="unbounded" minOccurs="1"></element>
            </sequence>
        </complexType>
        <simpleType name="action_type">
            <restriction base="string">
                    <enumeration value="API"></enumeration>
                    <enumeration value="KEI"></enumeration>
            </restriction>
        </simpleType>

        <complexType name="action">
            <sequence>
                    <element name="param" type="woad:action_param"
                            maxOccurs="unbounded">
                    </element>
                    <element name="field" type="woad:field"></element>
```

```
            </sequence>
            <attribute name="type" type="woad:action_type"></attribute>
            <attribute name="name" type="woad:action_name"></attribute>
        </complexType>

        <simpleType name="action_name">
            <restriction base="string">
                <enumeration value="Apropriateness"></enumeration>
                <enumeration value="Criticality"></enumeration>
                <enumeration value="Revision"></enumeration>
                <enumeration value="Schedule"></enumeration>
            </restriction>
        </simpleType>

        <simpleType name="aggregator_type">
            <restriction base="string">
                <enumeration value="avg"/>
                <enumeration value="count"/>
                <enumeration value="in"/>
            </restriction>
        </simpleType>

        <complexType name="aggregator">
            <attribute name="type" type="woad:aggregator_type"></attribute>
            <attribute name="fieldset_id" type="string"></attribute>
            <attribute name="value_id" type="string"></attribute>
        </complexType>

        <simpleType name="field_set">
            <restriction base="string"></restriction>
        </simpleType>

        <complexType name="conditions_list">
            <sequence>
                <element name="condition" type="woad:condition"
                    maxOccurs="unbounded" minOccurs="1"></element>
            </sequence>
        </complexType>

        <complexType name="action_param">
            <sequence>
                <element name="value"
                    type="woad:constant_value"></element>
            </sequence>
            <attribute name="id" type="ID">
                <annotation>
                    <documentation>The unique parameter name
                        (key)</documentation>
                </annotation>
            </attribute>
        </complexType>
```

</schema>

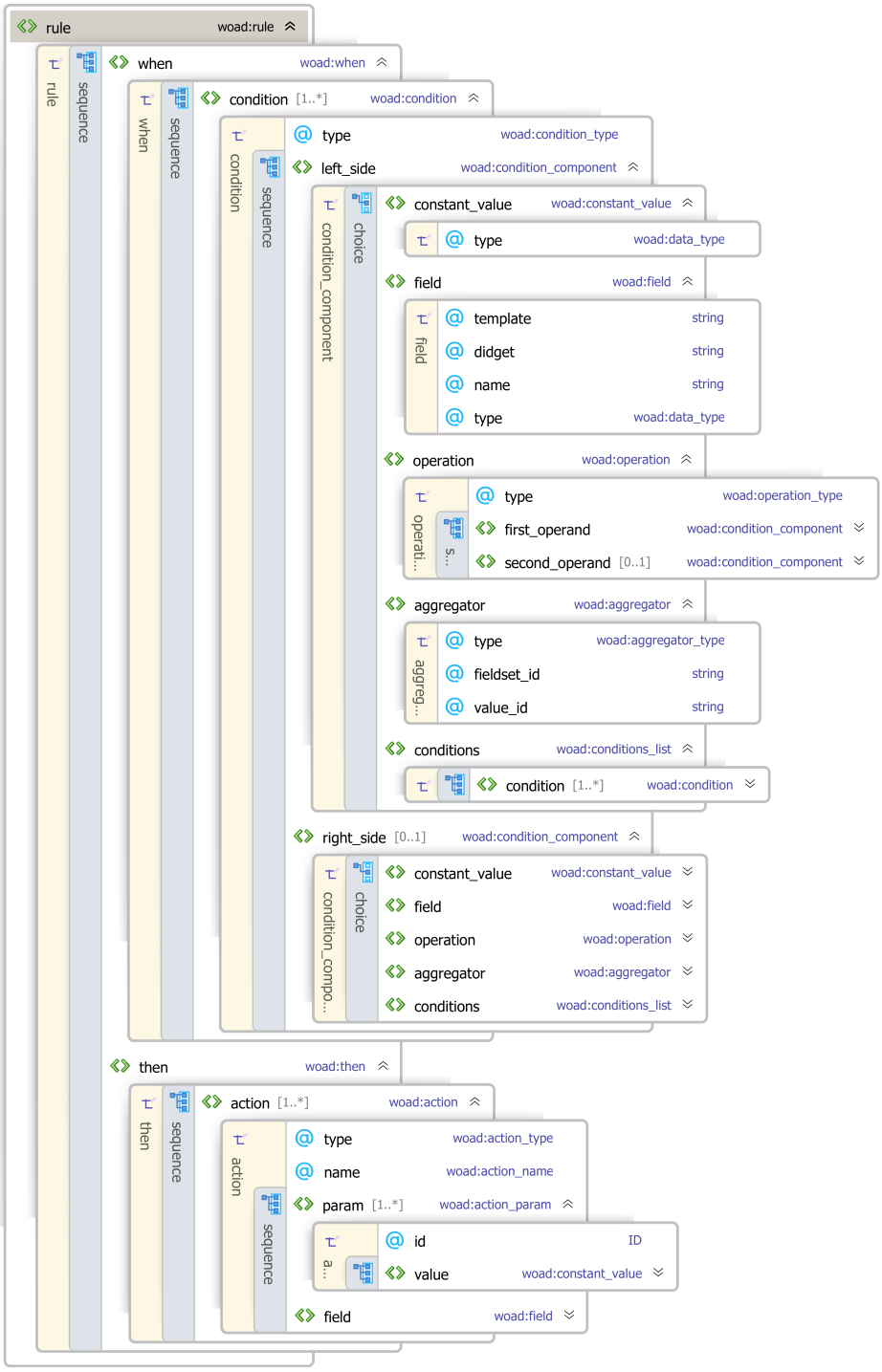


Figure B.1.: The graphic representation of the WOAD Intermediate Language XML Schema

Bibliography

Agostini, A., De Michelis, G., and Grasso, M. A. (1997). Rethinking CSCW systems: the architecture of MILANO. In *ECSCW'97: Proceedings of the fifth conference on European Conference on Computer-Supported Cooperative Work*, pages 33—48, Norwell, MA, USA. Kluwer Academic Publishers.

Aitken, R. (1969). Measurement of feelings using visual analogue scales. *Proceedings of the royal society of medicine*, 62(10):989.

Anderson, R. J. (1994). Representations and requirements: the value of ethnography in system design. *Hum.-Comput. Interact.*, 9(3):151–182.

Antoniou, G., Billington, D., Governatori, G., and Maher, M. J. (2001). Representation results for defeasible logic. *ACM Trans. Comput. Logic*, 2(2):255–287.

Ardito, C., Buono, P., Costabile, M. F., Lanzilotti, R., and Piccinno, A. (2012). End users as co-designers of their own tools and products. *Journal of Visual Languages & Computing*, 23(2):78 – 90. Special issue dedicated to Prof. Piero Mussio.

Ardito, C., Lanzilotti, R., Mussio, P., Parasiliti Provenza, L., and Piccinno, A. (2009). Redefining the roles of users and designers in interactive system lifecycle. In *Proceedings of CHItaly 2009, June 17-19, Rome, Italy.*

Ash, J. S., Berg, M., and Coiera, E. (2004). Some unintended consequences of information technology in health care: The nature of patient care information system-related errors. *Journal of the American Medical Informatics Association*, 11(2):104–112.

Atkinson, P. A. (1995). *Medical Talk and Medical Work*. Sage Publications Ltd.

Bannon, L. J. and Schmidt, K. (1991). CSCW: four characters in search of a context. *Studies in computer supported cooperative work: theory, practice and design*, pages 3—16.

Banâtre, J., Fradet, P., and Le Métayer, D. (2001). Gamma and the chemical reaction model: Fifteen years after. In Calude, C., PAun, G., Rozenberg, G., and Salomaa, A., editors, *Multiset Processing*, volume 2235 of *Lecture Notes in Computer Science*, pages 17–44. Springer Berlin / Heidelberg. 10.1007/3-540-45523-X_2.

Bardram, J. and Hansen, T. (2010). Context-Based workplace awareness. *Computer Supported Cooperative Work (CSCW)*, 19(2):105–138.

Bardram, J. E. and Bossen, C. (2005). A web of coordinative artifacts: collaborative work at a hospital ward. In *GROUP'05: Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*, page 168–176, New York, NY, USA. ACM Press.

Basili, V. R., Briand, L. C., and Melo, W. L. (1996). How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39(10).

Bassiliades, N., Kontopoulos, E., and Antoniou, G. (2005). A visual environment for developing defeasible rule bases for the semantic web. In Adi, A., Stoutenburg, S., and Tabet, S., editors, *Rules and Rule Markup Languages for the Semantic Web*, volume 3791 of *Lecture Notes in Computer Science*, page 172–186. Springer Berlin / Heidelberg.

Benford, S. and Fahlén, L. (1993). A spatial model of interaction in large virtual environments. In *CSCW'93: Proceedings of the 3rd European Conference on Computer Supported Cooperative Work*, pages 109—124, Dordrecht. Kluwer Academic Publishers.

Bentley, R. and Dourish, P. (1995). Medium versus mechanism: supporting collaboration through customization. In *ECSCW'95: Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work,*, pages 133–148, Stockholm, Sweden. Kluwer Academic Press.

Bergstrom, N. et al. (1987). The braden scale for predicting pressure sore risk. *Nurs Res*, 36(4):205–10.

Berry, M. and Goulde, M. (1994). A new view of documents. integrated information management in the '90s. *Workgroup Computing Report*, 17(8).

Beyer, H. and Holtzblatt, K. (1998). *Contextual design: defining customer-centered systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Bier, E. and Goodisman, A. (1990). Documents as user interfaces. In *EP90: proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography, Gaithersburg, Maryland, September 1990*, page 249.

Bier, E. A. (1992). EmbeddedButtons: supporting buttons in documents. *ACM Trans. Inf. Syst.*, 10(4):381–407.

Bier, E. A. and Pier, K. (1991). Documents as user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology*, CHI '91, page 443–444, New York, NY, USA. ACM.

Bleser, D., Depreitere, Waele, D., Vanhaecht, Vlayen, and Sermeus (2006). Defining pathways. *Journal Of Nursing Management*, 14:553–563.

Boehm, B. W., Clark, Horowitz, Brown, Reifer, Chulani, Madachy, R., and Steece, B. (2000). *Software Cost Estimation with Cocomo II*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

Boyer, J. M. (2008). Interactive office documents: a new face for web 2.0 applications. In *Proceeding of the eighth ACM symposium on Document engineering*, DocEng '08, page 8–17, New York, NY, USA. ACM.

Braa, K. and Sandahl, T. (1998). Information and process integration in enterprises. rethinking documents. Kluwer Academic Publishers.

Braa, K. and Sandahl, T. (2000). Introducing digital documents in work practices - challenges and perspectives. *Group Decision and Negotiation*, 9(3):189–203. Publisher Kluwer Academic Netherlands.

Bringay, S., Barry, C., and Charlet, J. (2006). Annotations: A functionality to support cooperation, coordination and awareness in the electronic medical record. In *COOP'06: Proceedings of the 7th International Conference on the Design of Cooperative Systems*, France, Provence.

Bultman, A., Kuipers, J., and van Harmelen, F. (2000). Maintenance of KBS's by domain experts. In Logananthara, R., Palm, G., and Ali, M., editors, *Intelligent Problem Solving. Methodologies and Approaches*, volume 1821 of *Lecture Notes in Computer Science*, pages 37–101. Springer Berlin / Heidelberg. 10.1007/3-540-45049-1_17.

Bødker, K., Kensing, F., and Simonsen, J. (2004). *Participatory IT Design. Designing for Business and Workplace Realities.* MIT Press.

Cabitza, F., Corna, S., Gesso, I., and Simone, C. (2011a). WOAD, a platform to deploy flexible EPRs in full control of End-Users. In Blandford, A., De Pietro, G., Gallo, L., Gimblett, A., Oladimeji, P., and Thimbleby, H., editors, *EICS4Med 2011: Proceedings of the 1st International Workshop on Engineering Interactive Computing Systems for Medicine and Health Care, co-located with the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2011) Pisa, Italy, June 13, 2011*, volume 727, pages 7—12. CEUR-WS.org.

Cabitza, F. and Gesso, I. (2011). Web of active documents: an architecture for flexible electronic patient records. In Fred, A., Filipe, J., and Gamboa, H., editors, *Biomedical Engineering Systems and Technologies. Third International Joint Conference, BIOSTEC 2010, Valencia, Spain, January 2010. Revised Selected Papers*, volume 127 of *Communications in Computer and Information Science*, pages 44—56. Springer.

Cabitza, F., Gesso, I., and Corna, S. (2011b). Tailorable flexibility: Making End-Users autonomous in the design of active interfaces. In Blashki, K., editor, *MCCSIS 2011: IADIS Multi Conference on Computer Science and Information Systems, Rome, Italy, July 20–26, 2011.* IADIS.

Cabitza, F. and Loregian, M. (2008). Much undo about nothing? Investigating why email retraction is less popular than apologizing. In *NordiCHI '08: Proceedings of the 5th Nordic conference on Human-computer interaction*, page 431–434, New York, NY, USA. ACM.

Cabitza, F., Sarini, M., Simone, C., and Telaro, M. (2005). "When once is not enough": The role of redundancy in a hospital ward setting. In Pendergast, M., Schmidt, K., Mark, G., and Ackerman, M., editors, *GROUP'05: Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work, GROUP 2005*, pages 158—167, Sanibel Island, Florida, U.S.A. ACM Press.

Cabitza, F. and Simone, C. (2008). Supporting practices of positive redundancy for seamless care. In *CBMS'08: Proceedings of the 21th IEEE International Symposium on Computer-Based Medical Systems, June 17-19 2008, Jyväskylä, Finland*, pages 470–475. IEEE Computer Society.

Cabitza, F. and Simone, C. (2009a). Active artifacts as bridges between context and community knowledge sources. In *C&T2009: Proceedings of the 4th International*

Conference on Communities and Technologies. June 2009. Penn State University, PA, USA., pages 115—124. ACM Press.

Cabitza, F. and Simone, C. (2009b). LWOAD: a specification language to enable the End-User develoment of coordinative functionalities. In Pipek, V., Rosson, M. B., de Ruyter, B. E. R., and Wulf, V., editors, *IS-EUD'09: Proceedings of the 2nd International Symposium on End-User Development, 2009, Siegen, Germany, March 2-4, 2009.*, volume 5435 of *Lecture Notes in Computer Science*, page 146–165. Springer.

Cabitza, F. and Simone, C. (2010). WOAD: a framework to enable the End-User development of coordination oriented functionalities. *Journal of Organizational and End User Computing (JOEUC)*, 22(2).

Cabitza, F. and Simone, C. (2012). Affording mechanisms: an integrated view of coordination and knowledge management. *Computer Supported Cooperative Work (CSCW)*, 21(2):227—260.

Cabitza, F., Simone, C., and Locatelli, M. P. (2012). Supporting artifact-mediated discourses through a recursive annotation tool. In *GROUP'12: Proceedings of the 17th ACM international conference on Supporting group work*, pages 253–262, New York, NY, USA. ACM.

Cabitza, F., Simone, C., and Sarini, M. (2009a). Leveraging coordinative conventions to promote collaboration awareness. *Computer Supported Cooperative Work (CSCW)*, 18(4):301—330.

Cabitza, F., Simone, C., and Zorzato, G. (2009b). ProDoc: an electronic patient record to foster Process-Oriented practices. In *ECSCW'09: Proceedings of the European Conference on Computer Supported Cooperative Work. Vienna, Austria, September 9-11, 2009.*, pages 119–138. Springer.

Cabitza, F. and Zorzato, G. (2010). Developing a flexible electronic patient record as a web of active documents. In *Healthinf2010: Proceedings of the Third International Conference on Health Informatics, 20-23 January 2010, Valencia, Spain*, pages 46—53.

Cardone, R., Soroker, D., and Tiwari, A. (2005). Using XForms to simplify web programming. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, page 215–224, New York, NY, USA. ACM.

Carroll, J. M., Kellogg, W. A., and Rosson, M. B. (1991). The task-artifact cycle. In Carroll, J. M., editor, *Designing Interaction: Psychology at the Human-Computer Interface*, pages 74—102. Cambridge University Press, New York, NY, USA.

Carstensen, P. H. and Schmidt, K. (1999). Computer supported cooperative work: New challenges to systems design. Technical report, vol. 43, CTI Working Paper. Published in Kenji Itoh (ed.). Handbook of Human Factors/Ergonomics, Asakura Publishing, Tokyo 2003, pp. 619-636.

Chamberlin, D., Hasselmeier, H., and Paris, D. (1988). Defining document styles for WYSIWYG processing. *Document Manipulation and Typography, EP88*, pages 121—137.

Chande, S. and Koivisto, A. (2006). Mobile form-editor: a push based group communication tool. In *Proceedings of the 3rd international conference on Mobile technology, applications & systems*, Mobility '06, New York, NY, USA. ACM.

Chang, S. (1987). Visual languages: A tutorial and survey. In Gorny, P. and Tauber, M., editors, *Visualization in Programming*, volume 282 of *Lecture Notes in Computer Science*, pages 1–23. Springer Berlin / Heidelberg. 10.1007/3-540-18507-0_1.

Chen, H., Ma, W., and Liou, D. (2002). Design and implementation of a real-time clinical alerting system for intensive care unit. In *Proceedings of the AMIA Symposium*, page 131.

Chen, W. and Akay, M. (2011). Developing EMRs in developing countries. *Information Technology in Biomedicine, IEEE Transactions on*, 15(1):62—65.

Chen, Y. and Wang, F. (2001). An editing system for working processes. In *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, page 332–338.

Clemensen, J., Larsen, S., Kyng, M., and Kirkevold, M. (2007). Participatory design in health sciences: Using cooperative experimental methods in developing health services and computer technology. *Qualitative Health Research*, 17:122–130.

Conley, D., Schultz, A., and Selvin, R. (1999). The challenge of predicting patients at risk for falling: development of the conley scale. *Medsurg nursing: official journal of the Academy of Medical-Surgical Nurses*, 8(6):348.

Conradi, B., Serényi, B., Kranz, M., and Hussmann, H. (2010). SourceBinder: community-based visual and physical prototyping. In Pipek, V., Rohde, M., Budweg, S., Draxler, S., Lohmann, S., Rashid, A., and Stevens, G., editors, *ODS 2010: Proceedings of the 2nd International Workshop on Open Design Spaces*, volume 7 of *International Reports on Socio-Informatics*, page 23–35, Stiftsgasse 25, 53111, Bonn, Germany. IISI - International Institute for Socio-Informatics.

Costabile, M. F., Dittrich, Y., Fischer, G., and Piccinno, A., editors (2011). *End-User Development: Third International Symposium, IS-EUD 2011, Torre Canne, Italy, June 7-10, 2011, Proceedings*, volume 6654 of *Lecture Notes in Computer Science*. Springer-Verlag New York Inc.

Costabile, M. F., Fogli, D., Lanzilotti, R., Mussio, P., Parasiliti Provenza, L., and Piccinno, A. (2008a). Advancing end user development through metadesign. In *End User Computing Challenges and Technologies: Emerging Tools and Applications*, pages 143–167. IGI Global.

Costabile, M. F., Fogli, D., Letondal, C., Mussio, P., and Piccinno, A. (2003a). Domain-Expert users and their needs of software development. In *UAHCI Conference*, volume 4, pages 232—236, Crete.

Costabile, M. F., Lanzilotti, R., and Piccinno, A. (2003b). Analysis of EUD survey questionnaire. Technical report.

Costabile, M. F., Mussio, P., Parasiliti Provenza, L., and Piccinno, A. (2008b). End users as unwitting software developers. In *WEUSE '08: Proceedings of the 4th international workshop on End-user software engineering*, page 6–10, New York, NY, USA. ACM.

Danado, J. and Paternò, F. (2012). Puzzle: A Visual-Based environment for end user development in Touch-Based mobile phones. In Winckler, M., Forbrig, P., and Bernhaupt, R., editors, *Human-Centered Software Engineering*, volume 7623 of *Lecture Notes in Computer Science*, pages 199—216.

Davidowitz, P. (1996). Externalizing Business-Object behavior: A point-and click rule editor. *The Smalltalk Report*, 6(1):4–10.

De Michelis, G. (2003). The "Swiss pattada". *interactions*, 10(3):44–53.

De Paula, R. A. (2004). *The construction of usefulness: how users and context create meaning with a social networking system*. PhD thesis, University of Colorado at Boulder, Boulder, CO, USA.

Decker, G., Overdick, H., and Weske, M. (2008a). Oryx–An open modeling platform for the BPM community. In Dumas, M., Reichert, M., and Shan, M., editors, *Business Process Management*, volume 5240 of *Lecture Notes in Computer Science*, page 382–385. Springer Berlin/Heidelberg.

Decker, G., Overdick, H., and Weske, M. (2008b). Oryx–Sharing conceptual models on the web. In Li, Q., Spaccapietra, S., Yu, E., and Olivé, A., editors, *Conceptual Modeling - ER 2008*, volume 5231 of *Lecture Notes in Computer Science*, page 536–537. Springer Berlin/Heidelberg.

Denzin, N. K. and Lincoln, Y. S. (2003). The interview: from structured questions to negotiated text. In *Collecting and interpreting Qualitative materials*. Sage, Thousand Oaks.

Dick, R. S., Steen, E. B., and Detmer, D. E., editors (1997). *The Computer-Based Patient Record: An Essential Technology for Health Care, Revised Edition*. National Academy Press.

Dolin, R. H., Alschuler, L., Boyer, S., Beebe, C., Behlen, F. M., Biron, P. V., and Shabo, A. (2006). HL7 clinical document architecture, release 2. *Journal of the American Medical Informatics Association*, 13:30–39.

Dourish, P. (2003). The appropriation of interactive technologies: Some lessons from placeless documents. *Comput. Supported Coop. Work*, 12:465–490.

Dourish, P. and Bellotti, V. (1992). Awareness and coordination in shared workspaces. In *CSCW'92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, CSCW '92, page 107–114, New York, NY, USA. ACM Press.

Dourish, P., Edwards, W. K., Howell, J., LaMarca, A., Lamping, J., Petersen, K., Salisbury, M., Terry, D., and Thornton, J. (2000a). A programming model for active documents. In *UIST2000: Proceedings of the ACM Symposium on User Interface Software and Technology*, San Diego, USA.

Dourish, P., Edwards, W. K., LaMarca, A., Lamping, J., Petersen, K., Salisbury, M., Terry, D. B., and Thornton, J. (2000b). Extending document management systems with User-Specific active properties. *ACM Transactions on Information Systems*, 18(2):140–170.

English, P. M., Jacobson, E. S., Morris, R. A., Mundy, K. B., Pelletier, S. D., Polucci, T. A., and Scarbro, H. D. (1990). An extensible, Object-Oriented system for active

documents. In *EP90: proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography, Gaithersburg, Maryland, September 1990*, pages 263—276.

English, P. M. and Tenneti, R. (1994). Interleaf active documents. *Electronic publishing*, 7(2):75–87.

Fischer, G. (2003). Meta—Design: beyond User-Centered and participatory design. *Human-computer interaction: theory and practice*, 1:88.

Fischer, G. (2009). End-User development and meta-design: Foundations for cultures of participation. In *End-User Development*, page 3–14.

Fischer, G. and Giaccardi, E. (2006). Meta-design: A framework for the future of End-User development. In Lieberman, H., editor, *End User Development – Empowering people to flexibly employ advanced information and communication technology*, pages 427–457. Kluwer Academic Publishers, Dordrecht, The Netherlands, NL.

Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A. G., and Mehandjiev, N. (2004). Meta-design: a manifesto for end-user development. *Commun. ACM*, 47(9):33–37.

Fischer, G. and Scharff, E. (2000). Meta-design: design for designers. In *DIS '00: Proceedings of the 3rd conference on Designing interactive systems*, pages 396–405, New York, NY, USA. ACM.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence*, 19(1):17–37.

Friedman, C. P. and Wyatt, J. (2006). *Evaluation Methods in Biomedical Informatics*. Health Informatics. Springer, 2nd edition.

Giaccardi, E. (2005). Metadesign as an emergent design culture. *Leonardo*, 38(4):342–349.

Gibson, J. (1977). *The theory of affordances*. Perceiving, Acting, and Knowing. Lawrence Erlbaum Associates, Hillsdale, NJ.

Greenbaum, J. and Kyng, M. (1991). *Design at work: Cooperative Design of Computer Systems*. Lawrence Erlbaum Associated, Hillsdale.

Greenhalgh, T. and Swinglehurst, D. (2011). Studying technology use as social practice: the untapped potential of ethnography. *BMC Medicine*, 9(1):45.

Grudin, J. (1994). Computer-supported cooperative work: history and focus. *Computer*, 27(5):19–26.

Harker, S., Eason, K., and Dobson, J. (1993). The change and evolution of requirements as a challenge to the practice of software engineering. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 266 –272.

Harrison, W. (2004). From the editor: The dangers of End-User programming. *Software, IEEE*, 21(4):5—7.

Hayes, M. and Patterson, D. (1921). Experimental development of the graphic rating method. *Psychol Bull*, 18:98–99.

Heath, C. and Luff, P. (1992). Collaboration and control. crisis management and multimedia technology in london underground control rooms. *Computer Supported Cooperative Work, The Journal of Collaborative Computing*, 1(2):69–94.

Hertzum, M. (1999). Six roles of documents in professionals' work. In *ECSCW'99: Proceedings of the Sixth European conference on Computer supported cooperative work*, page 41–60, Norwell, MA, USA. Kluwer Academic Publishers.

Hovorka, D. S. and Germonprez, M. (2009). Tinkering, tailoring, and bricolage: Implications for theories of design. In *AMCIS 2009 Proceedings*.

Hughes, J., King, V., Rodden, T., and Andersen, H. (1994). Moving out from the control room: ethnography in system design. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, page 429–439, New York, NY, USA. ACM Press.

Hughes, J. A., Randall, D., and Shapiro, D. (1992). Faltering from ethnography to design. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, CSCW '92, page 115–122, New York, NY, USA. ACM.

Huynh, D. F., Karger, D. R., and Miller, R. C. (2007). Exhibit: lightweight structured data publishing. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, page 737–746, New York, NY, USA. ACM.

Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., and Doyle, K. (1988). Fabrik: a visual programming environment. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, page 176–190, New York, NY, USA. ACM.

Jacques, L. (2011). Electronic health records and respect for patient privacy: A prescription for compatibility. *Vand. J. Ent. & Tech. L.*, 13:441–441.

Karger, D. R., Ostler, S., and Lee, R. (2009). The web page as a WYSIWYG end-user customizable database-backed information management application. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, UIST '09, page 257–260, New York, NY, USA. ACM.

Kensing, F. and Blomberg, J. (1998). Participatory design: Issues and concerns. *Journal of Computer Supported Cooperative Work*, 7(3–4):167–185.

Klann, M., Paternò, F., and Wulf, V. (2006). Future perspectives in End-User development. In *End User Development*, volume 9, page 475–486. Kluwer Academic Publishers.

Kohn, L. T., Corrigan, J. M., and Donaldson, M. S., editors (2000). *To Err Is Human: Building a Safer Health System.* Institute of Medicine (IOM).

Krebs, D., Conrad, A., and Wang, J. (2012). Combining visual block programming and graph manipulation for clinical alert rule building. In *Proceedings of the 2012 ACM annual conference extended abstracts on Human Factors in Computing Systems Extended Abstracts*, CHI EA '12, pages 2453–2458, New York, NY, USA. ACM.

Krueger, C. W. (1992). Software reuse. *ACM Comput. Surv.*, 24(2).

Kuziemsky, C. E. and Varpio, L. (2011). A model of awareness to enhance our understanding of interprofessional collaborative care delivery and health information system design to support it. *International journal of medical informatics*, 80(8):e150–e160.

Köppen, E. and Neumann, G. (1998). A practical approach towards active hyperlinked documents. *Computer Networks and ISDN Systems*, 30(1-7).

Lampson, B. (1978). Bravo manual. *Alto User's Handbook*, page 31–62.

Lauwers, C. J. and Lantz, K. A. (1990). Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems. In *CHI'90: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 303–311, Seattle, Washington, USA. ACM Press.

Lenz, M., Schmid, H., and Wolf, P. (1987). Software reuse through building blocks. *Software, IEEE*, 4(4):34 –42.

Lewis, D. K. (1969). *Convention: A Philosophical Study.* Harward University Press.

Lewkowicz, M. and Zacklad, M. (2000). Using Problem-Solving models to design efficient cooperative Knowledge-Management systems based on formalization and traceability of argumentation. In Dieng, R. and Corby, O., editors, *Knowledge Engineering and Knowledge Management Methods, Models, and Tools*, volume 1937 of *Lecture Notes in Computer Science*, pages 59—67. Springer Berlin / Heidelberg. 10.1007/3-540-39967-4_21.

Li, F., Tian, C., Zhang, H., and Kelley, W. (2010). Rule-based optimization approach for airline load planning system. *Procedia Computer Science*, 1(1):1455–1463.

Lieberman, H., Patern\'o, F., Klann, M., and Wulf, V. (2006a). End-User development: An emerging paradigm. In *End User Development*, volume 9, pages 1–8. Kluwer Academic Publishers.

Lieberman, H., Patern\'o, F., and Wulf, V., editors (2006b). *End User Development*, volume 9 of *Human-Computer Interaction Series*. Springer Netherlands.

Lincke, J., Krahn, R., Ingalls, D., and Hirschfeld, R. (2009). Lively fabrik - a web-based end-user programming environment. *C5 '09: Proceedings of the 2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing*, page 11–19.

Locatelli, M. P., Ardesia, V., and Cabitza, F. (2010). Supporting learning by doing in archaeology with active process maps. In *eLearning 2010, Proceedings of the IADIS International Conference on e-Learning, Freiburg, Germany. 26 - 29 July 2010.*, volume 1, pages 218–225.

Lortal, G., Lewkowicz, M., and Todirascu, A. (2005). AnT&CoW, a tool supporting collective interpretation of documents through annotation and indexation. In *Proceedings of DIENG, MATTA, IJCAI - KMOM Workshop*, Edinburgh.

Malone, T. W. (1983). How do people organize their desks?: Implications for the design of office information systems. *ACM Transactions on Information Systems*, 1(1):99–112.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., and Eastmond, E. (2010). The scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15.

Mamlin, B., Biondich, P., Wolfe, B., Fraser, H., Jazayeri, D., Allen, C., Miranda, J., and Tierney, W. (2006). Cooking up an open source EMR for developing countries: OpenMRS - a recipe for successful collaboration. In *AMIA Annual Symposium Proceedings*, volume 2006, page 529.

Mark, G. (2002). Conventions and commitments in distributed CSCW groups. *Computer Supported Cooperative Work (CSCW)*, 11(3):349–387. 10.1023/A:1021289427473.

Mark, G., Fuchs, L., and Sohlenkamp, M. (1997). Supporting groupware conventions through contextual awareness. In Prinz, W., Rodden, T., Hughes, J., and Schmidt, K., editors, *ECSCW'97: Proceedings of The Fifth European Conference on Computer Supported Cooperative*, page 253–268, Lancaster, UK. Kluwer Academic Publisher.

Morrison, C. and Blackwell, A. (2009). Observing End-User customisation of electronic patient records. In Pipek, V., Rosson, M., de Ruyter, B., and Wulf, V., editors, *End-User Development*, volume 5435 of *Lecture Notes in Computer Science*, pages 275–284. Springer Berlin / Heidelberg. 10.1007/978-3-642-00427-8_16.

Morrison, C., Fitzpatrick, G., and Blackwell, A. (2011). Multi-disciplinary collaboration during ward rounds: Embodied aspects of electronic medical record usage. *International journal of medical informatics*, 80(8):e96–e111.

Myers, M. D. and Newman, M. (2007). The qualitative interview in IS research: Examining the craft. *Information and Organization*, 17(1):2–26.

Mørch, A. I. and Mehandjiev, N. D. (2000). Tailoring as collaboration: The mediating role of multiple representations and application units. *Computer Supported Cooperative Work (CSCW)*, 9(1):75–100. 10.1023/A:1008713826637.

Mørch, A. I., Stevens, G., Won, M., Klann, M., Dittrich, Y., and Wulf, V. (2004). Component-based technologies for end-user development. *Commun. ACM*, 47(9):59—62.

Müller, R. and Rahm, E. (1999). Rule-based dynamic modification of workflows in a medical domain. In *Proceedings of BTW99*.

Nam, C. and Bae, J. H. J. (2002). A framework for processing active documents. In *KORUS-2002: Proceedings of The 6th Russian-Korean International Symposium on Science and Technology, 2002.*, page 122–125.

Nam, C., Jang, G., and Bae, J. J. (2003). An XML-based active document for intelligent web applications. *Expert Systems with Applications*, 25(2):165–176.

Nardi, B. A. (1993). *A small matter of programming: perspectives on end user computing.* MIT Press, Cambridge, MA, USA.

Norman, D. A. (1988). *The Design of Everyday Things.* Doubleday, New York, USA.

Norman, D. A. and Draper, S. W. (1986). *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA.

Nute, D. (1994). Defeasible logic. In *Nonmonotonic reasoning and uncertain reasoning*, volume 3 of *Handbook of logic in artificial intelligence and logic programming*, pages 353—395. Oxford University Press, Inc.

Pau, L. F. and Olason, H. (1991). Visual logic programming. *Journal of Visual Languages & Computing*, 2(1):3 – 15.

Perry, D. E., Porter, A. A., and Votta, L. G. (2000). Empirical studies of software engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, page 345–355, New York, NY, USA. ACM.

Pipek, V. (2005). *From tailoring to appropriation support: Negotiating groupware usage*. PhD thesis, Faculty of Science, Department of Information Processing Science, University of Oulu, Finland.

Pipek, V. and Kahler, H. (2006). Supporting collaborative tailoring. In Lieberman, H., Paternò, F., and Wulf, V., editors, *End User Development*, volume 9 of *Human–Computer Interaction Series*. Springer Netherlands.

Pipek, V., Rosson, M. B., and De Ruyter, B., editors (2009a). *End-User Development: 2nd International Symposium, IS-EUD 2009, Siegen, Germany, March 2-4, 2009, Proceedings*, volume 5435. Springer-Verlag New York Inc.

Pipek, V., Rosson, M. B., Stevens, G., and Wulf, V. (2009b). Supporting the appropriation of ICT: End-User development in civil societies. In Carroll, J. M., editor, *Learning in Communities*, Human–Computer Interaction Series, page 25–27. Springer London.

Pipek, V. and Wulf, V. (2009). Infrastructuring: Toward an integrated perspective on the design and use of information technology. *Journal of the Association for Information Systems*, 10(5):1.

Pope, C. (2005). Conducting ethnography in medical settings. *Medical Education*, 39(12):1180–1187.

Prinz, W. (1999). NESSIE: an awareness environment for cooperative settings. In *ECSCW'99: Proceedings of the Sixth European conference on Computer supported cooperative work*, page 391–410, Norwell, MA, USA. Kluwer Academic Publishers.

Puckette, M. (1996). Pure data: another integrated computer music environment. *Proceedings of the Second Intercollege Computer Music Concerts*, page 37–41.

Ray, P., Parameswaran, N., Chan, V., and Yu, W. (2008). Awareness modelling in collaborative mobile e-health. *Journal of Telemedicine and Telecare*, 14(7):381–385.

Redström, J. (2008). RE:Definitions of use. *Design Studies*, 29(4):410 – 423.

Reenskaug, T., Wold, P., Lehne, O., et al. (1996). *Working with objects: the OOram software engineering method*. Manning.

Reffat, R. and Gero, J. (2000). Situatedness: A new dimension for learning systems in design. *A. Brown, M. Knight and P., Berridge (eds), Architectural Computing: from Turing to*, page 252–261.

Reiser, S. J. (1984). Transformation and tradition in the sciences: Essays in honor of i. bernard cohen. page 303–316. Cambridge University Press, New York, USA.

Repenning, A., Ahmadi, N., Repenning, N., Ioannidou, A., Webb, D., and Marshall, K. (2011). Collective programming: Making End-User programming (More) social. In Costabile, M., Dittrich, Y., Fischer, G., and Piccinno, A., editors, *End-User Development*, volume 6654 of *Lecture Notes in Computer Science*, pages 325–330. Springer Berlin / Heidelberg. 10.1007/978-3-642-21530-8_34.

Rodden, T. (1996). Populating the application: A model of awareness for cooperative applications. In *CSCW'96: ACM Conference on Computer Supported Cooperative Work*, pages 87—96, Boston, Mass. (USA). ACM Press.

Rode, J. (2005). *Web application development by nonprogrammers: user-centered design of an end-user web development tool*. PhD, Virginia Polytechnic Institute and State University.

Rogers, Y. (1993). Coordinating computer-mediated work. *Computer Supported Cooperative Work (CSCW)*, 1(4):295–315. 10.1007/BF00754332.

Roque, R. V. (2007). *OpenBlocks: an extendable framework for graphical block programming systems*. Master thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science.

Rubin, I. and Rubin, H. (2005). *Qualitative Interviewing: The Art of Hearing Data*. Sage Publications, Incorporated, 2nd edition.

Sanders, E. (2002). From user-centered to participatory design approaches. *Design and the social sciences: Making connections*, page 1–8.

Sanders, E. and Dandavate, U. (1999). Design for experiencing: New tools. In *Proceedings of the First International Conference on Design and Emotion, Overbeeke, CJ, Hekkert, P.(Eds.), Delft University of Technology, Delft, The Netherlands*, page 87–91.

Sanders, E. B. (2001). Virtuosos of the experience domain. In *Proceedings of the 2001 IDSA Education Conference*.

Sandor, O., Bogdan, C., and Bowers, J. (1997). Aether: An awareness engine for CSCW. In *Proceedings of the Fifth European Conference on Computer Supported Cooperative Work*, page 221–236.

Scaffidi, C., Shaw, M., and Myers, B. (2005). Estimating the numbers of end users and end user programmers. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '05, page 207–214, Washington, DC, USA. IEEE Computer Society.

Schmidt, K. (1991). Riding a tiger, or computer supported cooperative work. In *ECSCW'91: Proceedings of the Second European Conference on Computer-Supported Cooperative Work*, pages 1–16, Amsterdam, NL. Kluwer.

Schmidt, K. and Bannon, L. (1992). Taking CSCW seriously: Supporting articulation work. *Computer Supported Cooperative Work (CSCW)*, 1(1-2):7–40.

Schmidt, K. and Simone, C. (1996). Coordination mechanisms: Towards a conceptual foundation for CSCW systems design. *Computer Supported Cooperative Work, The Journal of Collaborative Computing*, 5(2-3):155–200.

Schuler, D. and Namioka, A. (1993). *Participatory design: principles and practices*. CRC.

Sellen, A. J. and Harper, R. H. R. (2003). *The Myth of the Paperless Office*. MIT Press, Cambridge MA.

Seto, E., Leonard, K. J., Cafazzo, J. A., Barnsley, J., Masino, C., and Ross, H. J. (2012). Developing healthcare rule-based expert systems: Case study of a heart failure telemonitoring system. *International Journal of Medical Informatics*, 81(8):556 – 565.

Shapiro, D. (1994). The limits of ethnography: combining social sciences for CSCW. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, CSCW '94, page 417–428, New York, NY, USA. ACM.

Sharma, V., Simpson, R., LoPresti, E., Mostowy, C., Olson, J., Puhlman, J., Hayashi, S., Cooper, R., Konarski, E., and Kerley, B. (2008). Participatory design in the development of the wheelchair convoy system. *Journal of NeuroEngineering and Rehabilitation*, 5(1):1.

Simone, C. and Bandini, S. (2002). Integrating awareness in cooperative applications through the Reaction-Diffusion metaphor. *Computer Supported Cooperative Work, The Journal of Collaborative Computing*, 11((3-4)):495–530.

Sommerville, I., Bentley, R., Rodden, T., and Sawyer, P. (1994). Cooperative systems design. *The Computer Journal*, 37(5):357–366.

Spinrad, R. (1988). Dynamic documents. *Harvard University Information Technology Quarterly*, 7(1):15–18.

Star, S. L. (1988). The structure of Ill-Structured Solutions:Heterogeneous Problem-Solving, boundary objects and distributed artificial intelligence. In *Proceedings of the 8th AAAI Workshop on Distributed Artificial Intelligence, University of Southern California*.

Stevens, G., Pipek, V., and Wulf, V. (2009). Appropriation infrastructure: Supporting the design of usages. In Pipek, V., Rosson, M. B., Ruyter, B., and Wulf, V., editors, *End-User Development*, volume 5435, pages 50–69. Springer Berlin Heidelberg, Berlin, Heidelberg.

Stolee, K. T. and Fristoe, T. (2011). Expressing computer science concepts through kodu game lab. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, page 99–104, New York, NY, USA. ACM.

Strauss, A., Fagerhaugh, S., Suczek, B., and Wiener, C. (1985). *The Social Organization of Medical Work*. University of Chicago Press., New York, NY, USA.

Sun, Y., Nguyen, A., Sitbon, L., and Geva, S. (2010). Rule-based approach for identifying assertions in clinical free-text data. In F Scholer F, T. A., Turpin, A., and Trotman, A., editors, *Australasian Document Computing Symposium (15th)*, page 93–96, Melbourne, VIC. School of Computer Science and IT, RMIT University.

Supekar, K., Marwadi, A., Lee, Y., and Medhi, D. (2002). Fuzzy Rule-Based framework for medical record validation. In Yin, H., Allinson, N., Freeman, R., Keane, J., and Hubbard, S., editors, *Intelligent Data Engineering and Automated Learning — IDEAL*

2002, volume 2412 of *Lecture Notes in Computer Science*, pages 1–27. Springer Berlin / Heidelberg. 10.1007/3-540-45675-9_67.

Telier, A. (2011). *Design Things*. Design Thinking, Design Theory. The MIT Press.

Teoh, S. Y., Wickramsinghe, N., and Pan, S. L. (2012). A bricolage perspective on healthcare information systems design: an improvisation model. *SIGMIS Database*, 43(3):47—61.

Terry, D. B. and Baker, D. G. (1990). Active tioga documents: an exploration of two paradigms. *Electron. Publ. Origin. Dissem. Des.*, 3:105–122.

Timmermans, S. and Berg, M. (2003). The practice of medical technology. *Sociology of health and illness*, 25(3):97–114.

Towner, G. (1988). Auto-updating as a technical documentation tool. In *Proceedings of the ACM conference on Document processing systems*, DOCPROCS '88, page 31–36, New York, NY, USA. ACM.

Vredenburg, K., Mao, J., Smith, P. W., and Carey, T. (2002). A survey of user-centered design practice. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, CHI '02, page 471–478, New York, NY, USA. ACM.

Wachter, R. (2010). Patient safety at ten: unmistakable progress, troubling gaps. *Health Affairs*, 29(1):165–173.

Wachter, R. et al. (2004). The end of the beginning: patient safety five years after "To err is human.". *Health Affairs*, 23(1):1–12.

Waterman, D. A., Paul, J., and Peterson, M. (1986). Expert systems for legal decision making. *Expert Systems*, 3(4):212—226.

Wenger, E. (1998). Communities of practice: Learning as a social system. *The System Thinker*, 9(5).

Wenger, E. (2006). Communities of practice: A brief introduction.

Werle, P. and Jansson, C. G. (2001). Active documents supporting teamwork in a ubiquitous computing environment. In *In Proceedings of the PCC Workshop*.

Wolber, D., Su, Y., and Chiang, Y. T. (2002). Designing dynamic web pages and persistence in the WYSIWYG interface. In *Proceedings of the 7th international conference on Intelligent user interfaces*, IUI '02, page 228–229, New York, NY, USA. ACM.

Won, M., Stiemerling, O., and Wulf, V. (2006). Component-Based approaches to tailorable systems. In Lieberman, H., Patern\'o, F., and Wulf, V., editors, *End User Development*, volume 9 of *Human-Computer Interaction Series*, page 115–141. Springer Netherlands.

Wong, W., Moore, A., Cooper, G., and Wagner, M. (2002). Rule-based anomaly pattern detection for detecting disease outbreaks. In *Eighteenth national conference on Artificial intelligence*, page 217–223, Menlo Park, CA, USA. American Association for Artificial Intelligence.

Wulf, V., Pipek, V., and Won, M. (2008). Component-based tailorability: Enabling highly flexible software applications. *International Journal of Human-Computer Studies*, 66(1):1–22.

Yamazaki, S. and Satomura, Y. (2000). Standard method for describing an electronic patient record template: application of XML to share domain knowledge. *Methods of information in medicine*, 39(1):50–55.

Yamazaki, S., Takabayashi, K., Hirose, Y., Satomura, Y., Kamiyama, T., and Matsuo, H. (2000). Visual editor for template design of electronic patient record. In *Proceedings of the AMIA Symposium*, page 1160.

Zellweger, P. T. (1988). Active paths through multimedia documents. In *Proceedings of the International Conference on Electronic Publishing on Document manipulation and typography*, page 19–34, New York, NY, USA. Cambridge University Press.

Zellweger, P. T. (1989). Scripted documents: a hypermedia path mechanism. In *Proceedings of the second annual ACM conference on Hypertext*, HYPERTEXT '89, page 1–14, New York, NY, USA. ACM.

Acronyms

AJAX Asynchronous JavaScript and XML

API Awareness Promoting Information

BPMN Business Process Model and Notation

CMS Content Management Systems

CSCW Computer Supported Collaborative Work

CSS Cascading Style Sheet

Datom Documental atom

Didget Documental widget

DTD Document Type Definition

EPR Electronic Patient Record

EUD End-User Development

GUI Graphical User Interface

HTML HyperText Markup Language

IDE Integrated Development Environment

JNI Java Native Interface

JSON JavaScript Object Notation

LISP List Processor

MIT Massachusetts Institute of Technology

MVC Model-View-Controller

ODF Open Document Format, Standard ISO/IEC 26300:2006

PDF Portable Document Format

ProDoc Process-oriented Documentation

SVG Scalable Vector Graphics

UML Unified Modeling Language

URI Uniform Resource Identifier

WOAD Web of Active Documents

W3C World Wide Web Consortium

WYSIWYG What You See Is What You Get

XHTML eXtensible HyperText Markup Language

XML eXtensible Markup Language

XSD XML Schema Definition

XSL Extensible Stylesheet Language

List of Figures

List of Tables