INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Improving and Mediating Usability-to-Software Engineering Communication

Helder Manuel Antunes

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

August 2001

ABSTRACT

Improving and Mediating Usability-to-Software Engineering Communication

Helder Manuel Antunes

Our research investigates how to integrate usability concerns into the software development lifecycle and in particular how to improve the communication between usability and software engineers. In the last few years, many software development teams have tried to integrate user-centered design techniques into their software engineering lifecycle. However, because of lack of understanding and communication between two diverse teams, they often run into problems. One problem arises from the fact that the software engineering teams have their own techniques and tools for managing the whole development lifecycle including usability issues, and it is not clear where exactly in this usability engineering techniques should be placed and integrated with existing software engineering methods to maximize benefits gained from both. Several research attempts have been done to reconcile the user-centered design approach and traditional software engineering methods. The optimal methodology is therefore a tailored one, catering to the needs of each project and team. Our research, rather than investigating, how to alleviate the usability concerns of adapted software development lifecycle, focuses on the communication between usability engineers and developers. To motivate our

investigations, we first examine two popular requirement-engineering processes: RESPECT and the use-case driven process, as defined in the Rational Unified Process framework. This discussion allows us to identify milestones of a relevant communication between members of the software development team and usability specialists. Finally, we describe a tool which aims to assist both software and usability engineers to mediate their communication.

Keywords: Usability Engineering, Software Development Lifecycle, Use-Case, Human-to-Human Communication.

Acknowledgement

At this moment, I would like to thank many people who helped me to get through tough times and who made this thesis possible.

First and foremost, I would like to thank my supervisor Dr Ahmed Seffah, for both his supervision and patience.

I also want to give a special thanks to Celia, Panagiota and Cedric that helped me to proof read my thesis, as well as my family, and my friends that supported me actively during my master, amoung others Ana-Paula, David, Jorge-Filipe, Delphine, Rony, Mahtab.

I would also like to mention Julien, Jean-Marc, Laetitia, Elodie, Nicolas, Leesa, Seb, Laurian, Yves, Vero, Hassan and Bahia, with whom I had fun during the good and the bad days.

Contents

List of Tables

List of Figures

Chapter 1

Needs and Issues for Integrating Usability in Software Engineering Lifecycle

A vast chasm separates the requirements of the people on the early side of a product life from those on the late side. The first, the early adopters, want technological superiority, and they will suffer any cost, whether initial purchase price or cost of maintenance and usage, for the benefits. The others, the conservative late adopters, want reliability and simplicity: Their creed is "turn it on, use it, and forget it." Products have to be developed, marketed, and sold very differently for these two groups of people. (User-Centered Development, Norman, 1987)

1.1 The Problem: User-centered Versus Software System-Oriented Development Approaches

In the early stages of a technology, the consumers are typically *technically sophisticated.* As a technology matures, customers seek rather convenience, high-quality experience, low cost, and reliable technology than small technological advantages. The same product that satisfied the early adopters now creates confusion. They require assistance, handholding. Companies must build up elaborate service organizations to handle the customer needs. The computer industry has responded to these pressures by doing more of the same, only more forcefully. After all, their most successful strategies in the past were to increase the number of functions and features of each product. This is only

natural human behavior. But it is the wrong behavior in these circumstances. Changing times require changing behavior. The entire product development process must change. Now, for the first time, not only must the company take marketing seriously, it must entertain yet a third partner to the development process: user experience. Moreover, the entire development process has to be turned around so that it starts with user needs and ends with engineering.

Why is everything so difficult to use? The real problem lies in product development, in the emphasis on the technology rather than on the user, the person for whom the device is intended. To improve products, companies need a development philosophy that targets the human user, not the technology. Companies need a user-centered development approach.

1.1.1 User-Centered Development: Focus on User's Experience

The design phase of User-Centric Development focuses on the user during the development process of the software. In software development, the user's needs are often badly captured, and expressed in the initial problem specification. The representative of the user not knowing exactly what he wants, we have to interact with him often to refine the problem specification.

In UCD, the methodology maintains a set of models representing the user and the system, depending on the methodology used the form and organization of these models is different. Common representations use UML's use-cases, or refined version of it, to capture user's needs formulated as tasks or goals. The initial information captured about the user and system is also organized in various ways, depending on the methodology, and usually refined iteratively to match to a certain level of accuracy of the user's needs (problem specification) in one hand and the system possibilities (solution specification) in the other. The difficulty is that this approach reduces costs by taking away degrees of design freedom from the system designer.

Alas, a full-fledged Usability Engineering Group is rare in today's technology companies. Most companies have a few people scattered here and there who work on the user interface (variously called *human interface groups* or *human factors groups*). They are technical writers and industrial designers, perhaps a few graphical designers. But these people are seldom in one organization, seldom given much power. They are usually relegated to the minor ranks, called upon at the tail end of product development to "make it easy to use," "make it pretty," "explain how to use it." This is not the way to deliver quality user experience.

1.1.1.1 Design Guidelines

Guidelines can help establishing rules for coordinating individual design contributions, to make design decisions just once rather than leaving them to be made over and over again by individual designers, defining detailed design requirements and evaluating user interface software in comparison with those requirements. But it's difficult sometimes to make the tradeoffs among these principles when they come into conflict; we often have to figure out the best solution by guessing, or by resorting to other means. If you're a novice designer, it's hard even to remember all these principles, let alone use them effectively! Indeed the [ESD/MITRE 1986] compilation of user interface design guidelines already gathered 944 guidelines.

1.1.1.2 Discount Usability Engineering

Nielsen [Nielsen 93] pointed out many methods to evaluate the usability of software, including techniques like proactive field study, co-discovery learning, heuristics evaluations, thinking aloud protocol, coaching methods, etc. It was enhanced later with pluralistic walkthroughs [Nielsen & Mack 94], cognitive walkthroughs [Nielsen and Mack 94], teaching methods [Vora & Helander 95], remote testing [Hartson et. Al. 96], feature inspection, questionnaire, scenario-based checklist, etc. All these were later organized in Mayhew's [Mayhew 99] Software Development Life-cycle.

The goal of these Evaluation Methods is to produce a formal report with problems identified or recommendations for changes. Human factor experts usually perform these evaluations, but software developers can perform some of them having pertinent results. Heuristics evaluation constitutes the archetype of usability evaluation, used from the design phases to the deployment phases of the usability engineering lifecycle. They cover what the interface should have. Even if software developers can perform it, familiarity with the rules is very important, in the quality of the review. In a cognitive walkthrough, analyst simulates a user's problem solving process at each step in carrying out a task scenario on a given user interface design to analyze it for usability success and failures. They have been already promoted in traditional software engineering [Yourdon 89], and do not require usability experts.

1.1.1.3 Limitations of these Techniques

These techniques deal with simple usability malfunction, or surface problems (like inconsistencies in the interfaces, or problems of use); they do not try to "stick to the user" by implicating him in the development process. Implying that engineers may be working on features that do not correspond to the user's needs, they are still too much system-centric.

In the design side, software engineers are still performing tasks in system-centered ways (like writing the specifications, organizing the interface). There is a need for a usability team that will not be influenced by system-centric concerns, and will defend user's needs.

1.1.2 System-Oriented Development: Focus on Functionality

On system-centric modeling the initial problem specification guides the creation of a first approximate solution specification. Then the engineering staff tunes this initial solution specification to reflect their best understanding of an optimal design. From then on the goal is to iterate between problem specification and solution specification, refining each. Every iteration pulls both problem specification and solution specification closer to some

intermediate description. This approach does not incorporate the problem statement and it's refinements (user needs) in each iteration, as would UCD.

System-centric modeling also provides its savings by combining the object model representing the problem specification with the object model representing the system design. In principle the object model ought to be constrained by both the content of the problem specification and the constraints of the system's technology. Unfortunately, the reason for going this route usually is that the system designer needed more degrees of design freedom to arrive at an understandable and well-performing design, than actual guidance on the problem to solve. In my experience, the object model nearly always ends up much more closely reflecting technology constraints than it does the problem specification.

1.1.2.1 Limitations of GUI Builders, RAD and CASE Tools

Style guides, as well as GUI standards, by settling many details of the interface in a certain context (usually a platform) help designers to deliver applications that are consistent with the other application of the same platform. These are implanted in many GUI builders, which makes the new application more usable for a user used to this specific look-and-feel. Typically, technology oriented users, such as programmers, appreciate this inter-application consistency, because this feature facilitates the learning of new applications. On the other hand, current tools require the specification of a lot of details (programmers must give the exact widget, font, alignment, color), so designers are led to focus on unimportant details, evaluators' focus on wrong issues. Also, these tools have a poor support for iterative design. Doing changes to get a new prototype takes too much time, which is a key point of iterative development. In early design we could use tools like J. Landay's Sketching Interfaces Like Krazy ([Landay 96]), a sketching tool that allows the designers to easily change the prototypes, but it does not generate any code and there is no tool in the common market that has good reengineering properties.

1.1.2.2 Use-Case Driven Approach, Traditional Lifecycle, UML, etc.

Traditionally Software Engineering tried to unify its different processes and methodologies, to reach a certain common ground or standard. Key moments of these unifying process were the creation of a Software Development Life-Cycle (a.k.a. SDLC) [ISO/IEC 12207:1995] sequencing all the techniques used to develop software, and the coming of UML [Booch & al. 97] standardizing the notation in the different documents, later gathered in the Rational Unified Process.

Software engineering practices and research have led to several object-oriented development methodologies that are highly adequate for the development of systems with little or no user interaction. However, for interactive systems with a significant user interface, these methods have a major gap. Most of them do not propose, at least explicitly, any mechanisms (or modeling) for: (1) explicitly and empirically identifying and specifying user needs and requirements, (2) testing and validating requirements and early user interface prototypes with end-users before, and during. As a result of this major weakness, interactive systems developed using such methods can meet all functional requirements, and yet be unusable. This problem explains a large part of the frequently observed phenomenon whereby large numbers of change requests to modify the services of an application are made after its deployment. The UCD (user-centered design) is a proven and popular approach for developing more usable interactive systems. However, the lack of UCD integration in traditional software engineering compromises its effective use in software engineering lifecycle.

1.1.3 Reconciling User-centered versus Traditional Software Engineering Approaches – The Why?

Norman showed that software engineering that is in used in the industry does not provide software fitting to a market that requires more and more services, quality and usability. A pure User-Centered engineering cannot supplant traditional software engineering, for cost-effectiveness reasons. Firms don't want to afford a migration from one more-or-less

well-adapted process to a new-born process that has not proven reliable yet. UCD has to be seen as a methodology that possibly can come to enhance the existing SDLCs.

While usability evaluation can be effectively staffed on a contract basis, user interface analysis and design work can be more difficult to coordinate especially with contractors or consultants, thus forcing big projects to find other ways to incorporate the usability in their development process. What seems to be the more straightforward method is to incorporate usability features in the already-existing software development lifecycles of the firms. As we will see in the next section, many researchers are exploring the method, coming up with integration methods at different levels of the development cycle. However, some concerns are growing concerning the need to keep the two different processes as two different entities in the development process. Indeed the experience has shown that bringing Human-Factor experts in a software engineering team will denature their work, influenced by their environment they will tend to focus on technical constraints to the depends of user's concerns. In the same way, software engineers tend to describe a user task as a use-case, which can be justified by the analysis that these two different methodologies have a different philosophy (characterized by different standards) as shown in Figure 1.

In the separation of user- and system-modeling approaches, the model specifying the user's needs and the object model specifying the provided solution are separately maintained throughout the project's life cycle. The difficulty in this case is to engineer a development process that minimizes the cost of maintaining synchrony between these two independent but linked methodologies. The problem is not stated as "How to integrate usability features in a Use-case driven process?" but "How to synchronize this process with a user-centered process?".

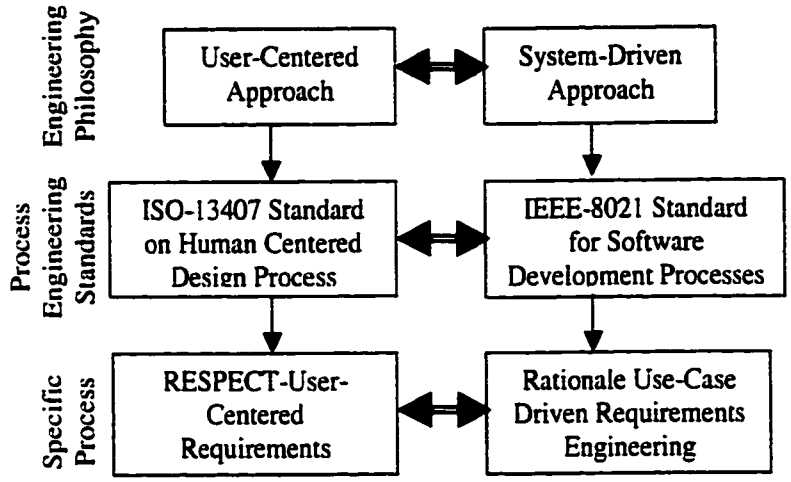


Figure 1 A View of our Research Case Study and Framework

1.2 Background and Related Work - A Brief Literature Review

The following investigations show that the philosophy of the use case-driven software development approach is highly compatible with the user-centered techniques. Most of them suggest specific, yet powerful, enhancements to the use case-driven software development approach, particularly in the user requirements and usability specification chapters. We will also point out the problems that arise from the different aims of the different philosophies.

1.2.1 Rosson– Integrating Development of Tasks and Object Models

[Rosson 99] suggests a methodology, supported by a tool that enhance Object Oriented Analysis & Design (OOAD) with a scenario-based approach, and favors an object-by-object analysis. For this we first extract from the usage scenarios, potential computational objects that we organize as a network of collaborating objects; then we focus on a

specific object trying to assign functionality to it (this Object-by-Object analysis is help by the Point-Of-View Browser that keeps user-relative descriptions of the each object).

The communication approach is middle-out, since she iteratively elaborate a set of user tasks (described in user interaction scenarios) in two directions: toward networks of collaborating computational objects on the one hand, and toward detailed user-interaction episodes on the other. Which is the opposite from prototyping tools like Visual Basic, which are outside in, because the focus is on screen design.

These techniques guarantee a good object model as well as the taking into account of the user's point of view. It satisfies our main concern: the incorporation of the user's opinion in the software development process. However, in this technique the user interface is designed relying only on the user's description of their tasks, and usability claims. Rosson already determined that this would cause mismatches with the user's view, which she says to be minor compared to the need of structure in the task model (needed for evolutiveness). She somehow defines an intermediate philosophy. The aim is not the user and his needs, or a good structure of the software any more, the aim is to have a good midpoint construct that helps having a good interface as well as a good structure of the program. This solution did not seem to develop in the industry market, may be being too different from the methods in use.

1.2.2 Artim – Integrating User Interface Design And Object-Oriented Development Through Task Analysis And Use Cases

[Artim 98] tried to augment use case based methodologies to support interface design and usability engineering. This integration is architectured on the synchronization of the problem specification and the solution specification (figure 2); which are updated at each iteration through an assessment of impact of the changes in the models.

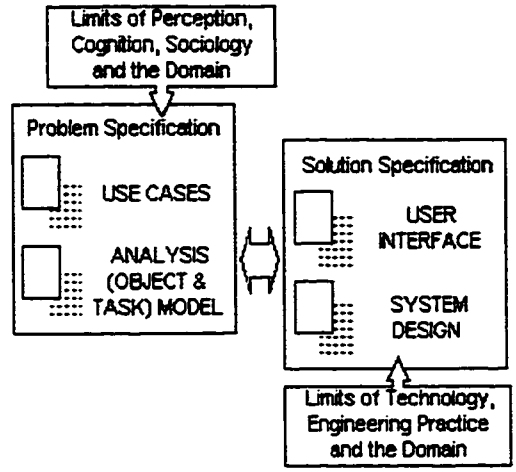


Figure 2 A Development Process Composed of a Problem Specification and a Solution Specification

The amount of work it generates justifies the use of a CASE tool to manage use-case descriptions along with additional information pertinent to user interface analysis and design. This tool also manages many views of the same model, and is able to generate five types of reports on a model corresponding to the needs of different actors and steps of the development process:

- A "user's view" of the use cases,
- A "developer's view" of the use cases,
- A glossary of terms,
- A more focused user interface analyst's report,
- An actor-system interaction report

Theoretically, having a consistent model that provides simple views for any actor and automatically includes the user's concern, should be enough to enable the software engineers to keep track of the user's needs during their design process. However, as Artim pointed out in his case study, the culture of the software engineers does not include collaborating with the user in the process of building a better system. These sociological forces within development teams will limit even discard any impacts of the user in the development of the system, thus providing a system that fits to static user's

specifications, rather than fitting the best to the user's need. We can deduce from this case study that even though the process of development determines directly the product being created, it is not the only factor.

1.2.3 Mayhew – Usability versus Use Case-Driven Lifecycle

Since the apparition of the concept of software lifecycle, it's content has evolved. Nowadays the standard content appearing to be Use-case driven processes like the Rational Unified Process. The generic term of Software Development Life-Cycle (SDLC) usually design this more traditional Use-Case Driven Life-Cycle.

[Mayhew 1999]'s main contribution to software engineering is the Usability Engineering Lifecycle. She developed this methodology over many years of experience, aimed at achieving usability in software engineering. This work is a backbone to usability engineering in the way that it references and organizes under the form of a process many engineering techniques that could participate in having more usable software. The keystones of this lifecycle are:

- Structured usability requirements analysis tasks
- Explicit usability goal setting task, driven directly by requirements analysis data
- Tasks supporting a structured, tops-down approach to user interface design that is driven directly from usability goals and other requirements data
- Objective usability evaluation tasks for iterating towards usability goals

The general organization of the process is described in the figure 3, and further details are given in [Mayhew 99].

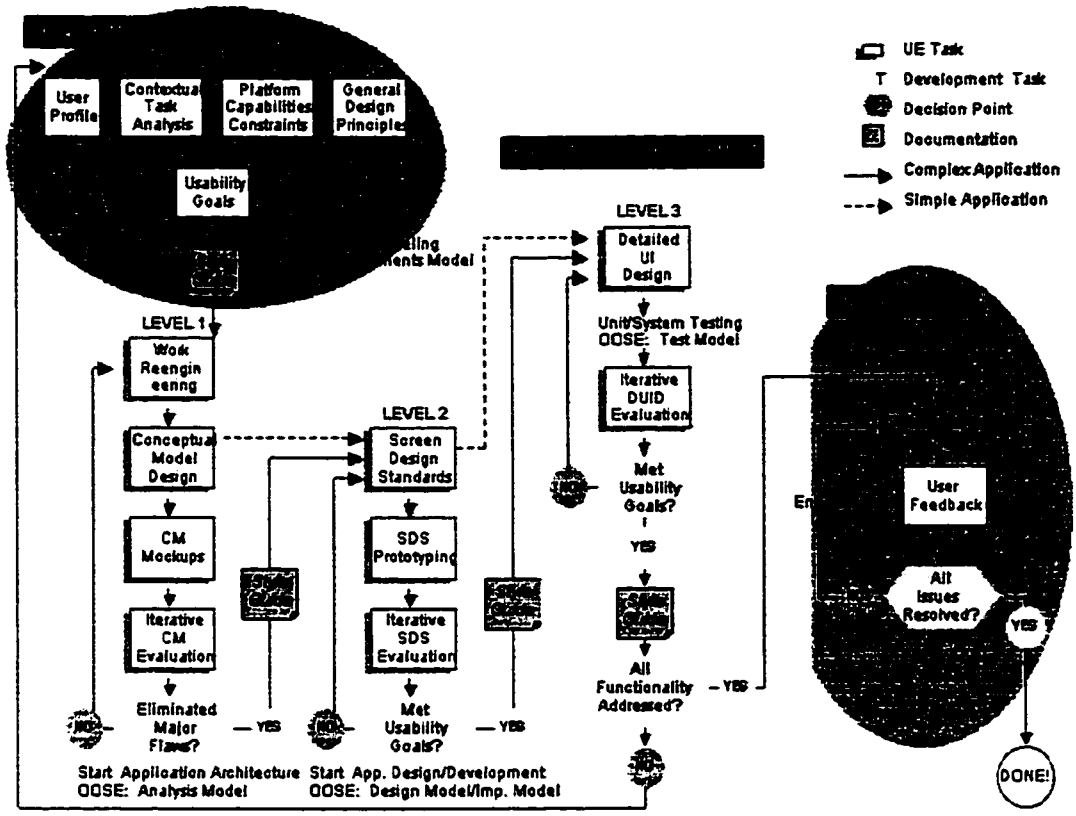


Figure 3 Usability Engineering Lifecycle

The advent of a state-of-the-art usability engineering lifecycle, rather than a usability-enhanced software engineering lifecycle supports the opinion that usability engineering corresponds to a new and different optic of software engineering. A complete merging into a single streamed lifecycle seems possible, but complicate. In the same ways as "one cannot serve two masters", a methodology like a software lifecycle or development process cannot comply with two different philosophies.

1.2.4 Jarke - Scenarios as Intermediate Design Artifacts

[Jarke 99] tries to clarify the purpose and manner to use scenarios in the modeling process, since this concept can be used in very different manners. He defines the scenarios as constructs that describe possible set of events that might reasonably take

place; they offer "middle-ground abstraction between models and reality". They are typically used in four approaches (Figure 4):

- Capture a sequence of work activities
- View a sequence of representations or interfaces
- View the purpose of users in the usage of the software
- View the lifecycle of the product.

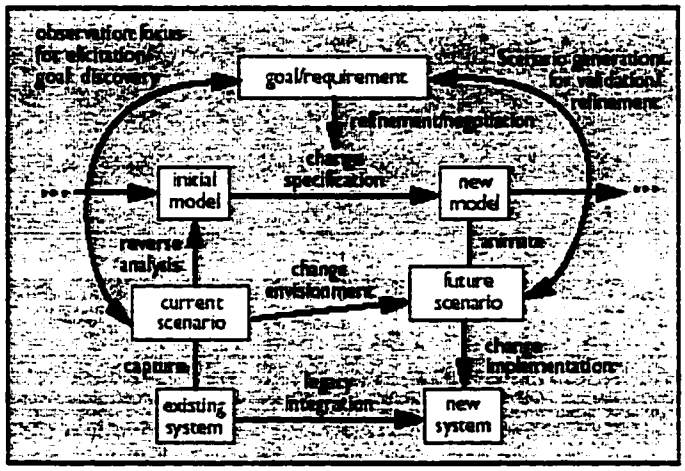


Figure 4 Change Process with Goals and Scenarios

As is described in Figure 4, a simple artifact can be used in many ways, misleading sometimes. So, it is not because a construct is powerful (or lenient in its definition) that it adequately describes a requirement.

1.2.5 Krutchen – Use Case Storyboards

[Krutchen 99], who works at Rational, adds a new artifact to the Rational Unified Process, the Use-Case Storyboard, which provides a high level view of dynamic window relationships such as window navigation paths and other navigation paths between

objects in the user interface. These Use-Case Storyboards have to be written at analysis-time, as all the use-cases. It brings many convenient constructs like:

1. Flows of events, also called storyboards –text user-centered description of interactions
2. Class Diagrams – classes that participate to the use-cases
3. Interaction Diagrams – describe the collaboration of the objects
4. Usability Requirements – text version of usability requirements
5. References to the User-Interface Prototype – text description the user-interface prototype
6. Trace dependency – sort of map of the use cases

Krutchen also gives guidelines on how to use this new construct. He recommends among other to have a Human-Factors expert write these documents. This is based on the claim that traditional software engineers will not use this artifact in a proper way, not being used to its philosophy. A big concern about this new technique comes from specifying the interface and the interactions at the beginning, rather that deciding of them as a result of design, thus limiting the possibilities of the interface; by "putting cart before horse" [Constantine and Lockwood 2000]. This also illustrates somehow, that use-case can adapt to usability engineering, but do not force the traditional Use-Case-Driven-Process designers to use them adequately.

1.2.6 Nunes and Cunha – WISDOM, Use Cases Annotation

[Nunes and Cunha 99] developed the Whitewater Interactive System Development with Objects Models (WISDOM), a lightweight software engineering methodology corresponding to a software engineering need of Small and Medium Enterprises. Whitewater approach describes the development process as moving quickly as a whole towards its goal in a messy way. It uses UML to support Human-Computer interaction techniques; and is evolutionary in the sense that the project evolves incrementally through an iterative process. One piece of originality in this work is to try to harness the

development, even-if it is seen as an unorganized set of contributions moving the software towards completion.

Later on, in a broader approach [Nunes and Cunha 2000] decided to add extensions to UML, in order for it to accommodate task analysis. The modeling constructs have to accommodate:

- Describing user and their relevant characteristics
- Describing user behavior/intentions while performing the envisioned or supported task
- Specify abstract and concrete user interface

For that they define a framework composed of the two models (problem-centered interaction model, and the solution-centered analysis model) sharing information in an unspecified manner (Figure 5). There are many changes and adds to UML to support this: change of class stereotype boundary, control and entity; add of task, interaction space, class stereotype, add-ons of the associations communicate, subscribe, refine task, navigate, contains, etc.

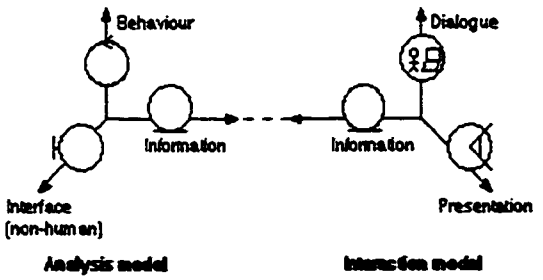


Figure 5 New Analysis Framework for Interactive System

The communication scheme here seems to be a 'co-evolutionary development of interactive systems', having both models and works evolving in concurrence. But concerns arise about the frequent communication misadventures between HCI and

Software Engineering people, as well as the tendency of misinterpretation of modeling of constructs like use-cases, this originating from people having different cultures having a different understanding of a polyvalent language like UML.

1.2.7 Constantine and Lockwood – Usage-Centered Design and Essential Use Cases

Use cases that are very powerful but often misused because of their lenient definition. [Constantine & Lockwood 99] try to harness the potential of the use-cases, so they can be used as task models. What they call usage-centered design step into the field of usability, by providing a greater focus on interface design and user tasks. For that, he first recapitulates previous works by structuring the models into five kinds of interrelated models, organizing the three center ones by a map (fig 1), so we respectively have the:

- User Role Map structuring the user roles -which hold the user information-,
- Navigation Map structuring the content models -which hold the interface views-,
- Use Case Map structuring the use cases -which hold the task descriptions -,
- Domain Model –which holds glossary, data and class models-,
- Operational Model –which holds environmental and contextual factors-.

The communication scheme is outside in, since the interface and the user needs are concerns presents since the beginning of the development. These three sets of models can be developed/enhanced concurrently, which cuts from more traditional (even-if iterative) sequential approaches.

In the attempt to completely specify the design methodology, they define the notion of essential use-cases. These essential use-cases try to enhance usability by focusing on intention rather than interaction, and simplification rather than elaboration. The use-cases repertory user intentions and system responsibilities focusing only information considered essential and occulting unneeded and redundant information, which make use-cases more subjective to eventual technological or environmental changes in the technology, or in the environment. Constantine is giving a structure to the essential use-

cases (Figure 7), at the same time as defining the syntax of the narratives. He also admits the restrictions that bring essential use-cases in the domain of software engineering, and the design of internal software architecture, that's why he advocated the use of essential use-cases only in the core process, which is vital for having good usability properties.

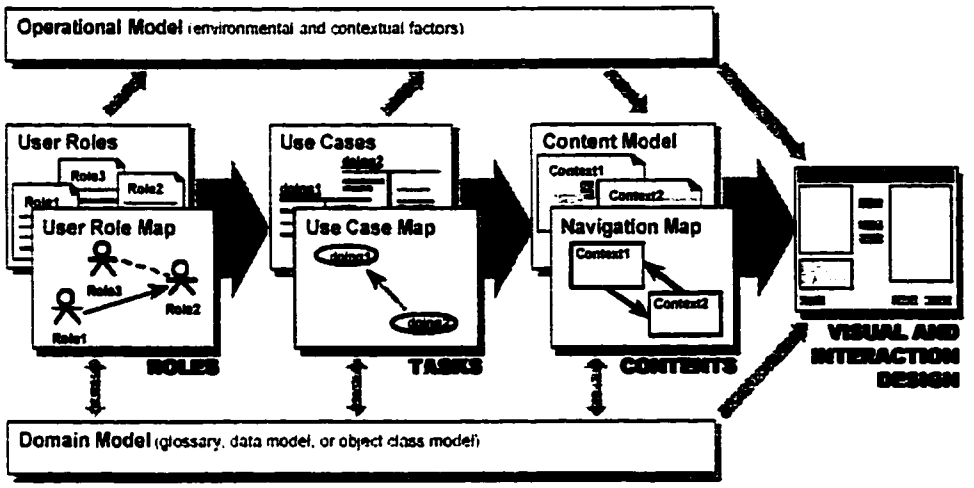


Figure 6 Logical Relationships among Primary Models in Usage-Centered Design

ID	Name
Contextual Purpose	Supported Roles
Specializes	Extends
Resembles	Equivalents
Preconditions	
User Intentions	System Responsibilities
Asynchronous Extensions	Asynchronous Extensions
(steps)	(steps)
Post-conditions	
Business Rules	

Figure 7 Schematic Framework for Structured Essential Use-Cases

Constantine references in his work many top researches in the field, and highlights many interesting points. Among them I'd like to focus on:

- The need to structure the weakly specified use-cases (allowing good designer to do powerful things as the same time as to mislay inexperienced designers). As we said previously: the use-cases as a construct is compatible with usable engineering, but it does not force it to be usable. We have to develop a framework that facilitates and forces it in some way to take care of user concerns.

- The trade off between usability, which requires abstraction and essence, and implementation that rather requires technical details in the models. This points out the philosophy differences that we highlighted before: usability does not have the same concerns as software engineering. Melting it in a unique process followed by a unique team will be inadequate. We need to look at all the dimensions for integration.

- The innovation of a more flexible and efficient process than traditional sequential processes, by introducing concurrency in the design process. The models are just considered as holding places for the "designer's fragmentary but evolving understanding". The views of these three models as ever evolving and interrelated but independent enough to be considered different entities corresponds for me to a good architecture of the models, since not being very constraining it organizes pretty well the knowledge of the model. We however fear problems of synchronization during the eventual update of the models. The communication between these parts, and the synchrony between the models, may be hard to achieve; and is for me a real concern.

1.2.8 Bevan – Barriers to User-Centered Development

[Bevan 97] while investigating the barriers to a usable software, points out among other:

- The abundance of different standards (ISO 13407: for UCD Process; ISO 14598-1 and ISO 9126-1: for quality in use; ISO 9241-11 for the definition of usability).
- Incomplete requirements causes defects (RESPECT tries to solve that)
- The absence of focus on usability in the firms (the are few norms outside of UCD).

These problems come from a gap between research and industry; from an overly delayed culture change, on the consumers' side as well as the firms' side. The standards do exist but are not implemented, firms do not want to invest the money, designers do not take training in usability, purchasers do not take usability criteria into account. That's why Bevan advocates to concern more about user in different levels of industry, at the same time as he provides supporting material on the web, to facilitate the culture change.

1.2.9 Cockburn – Enhancing Use Cases Form

Engelberg proposes a hierarchized task analysis similar to Cockburn work. [Cockburn 97] wants to reorganize the use-cases: they are not well defined and many different uses coexist (different in purpose, content, plurality and structure). He proposed to structure them with respect to goals, which limits the scenario explosions, but is rather unusual as a structure for requirements (Figure 8). The goals are structured as a tree containing "Summary goals" as high-level goal, and "User goal" as atomic goal (perform goal A is performing goal A1 then A2). Sub-functions will be user to achieve these End-branch User goals.

The approach is relevant and interesting. It belongs to the set of work that tries to use or adapt use-cases to capture user-centered (non-functional) requirements. User concerns (problem specification) are captured in the hierarchy of goal.

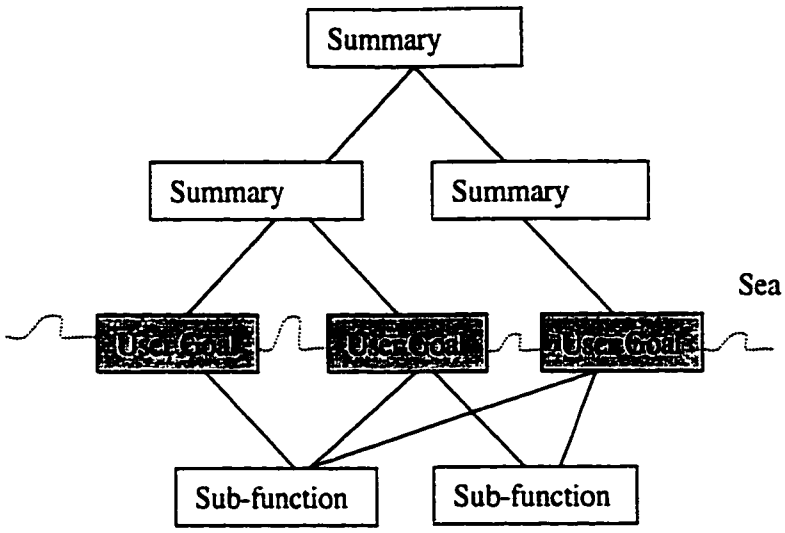


Figure 8 Structure of Use-Cases

Difficulties arise from the multitude of levels in the goals, and the fact that simple features appearing in many different interfaces can cross-multiply the number of scenarios. But I believe that an organization of the use-case cannot be optimal for all the problem types, and we may alter the original organization by goals to fit better to these cases.

The success of these techniques will reside in its ability to adapt to different cases of use, which was the feature that made traditional use-cases powerful. However I am somehow skeptical in our ability to have programmers implement programs following a set of goals, since it may not correspond to their education (e.g. a software engineer/technical may have troubles to design/implement a module that has to meet qualitative usability goals).

1.2.10 Forbrig and Seffah – Comparing Use-Cases versus Task Analysis

[Forbrig 99] & [Forbrig & Dittmar 99] introduce a framework of task models, user models, and object models (Figure 9). These multiple dimensions require tools enabling the manipulation of all this models along the different phases of software development.

This should contain any information model that can be used for an integration of human-factors in a software engineering process.

Related to this work, [Seffah & Hayne 99] also investigated the similarities of the task analysis and use case techniques via an attempt to recast the results of a task analysis into the format of use cases. They concern also about the importance of human-to-human communication, which does not appear in use cases.

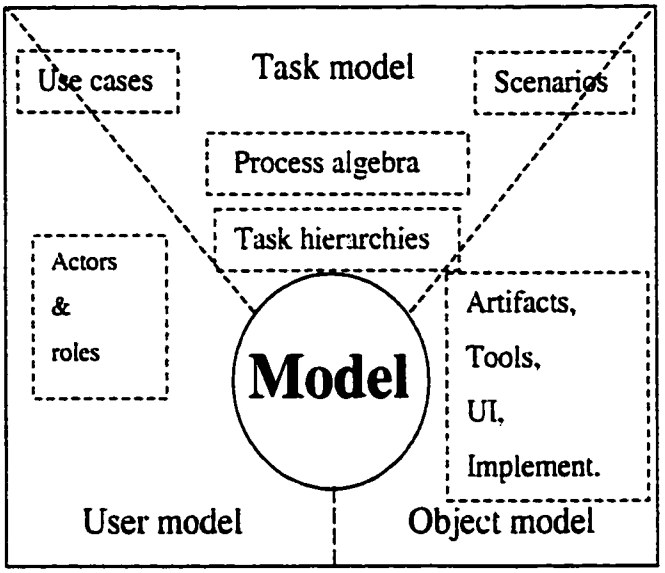


Figure 9 Views on the Requirements and Analysis Model

1.2.11 Elissa Darnell – A Framework for Applying Usability Techniques

[Darnell 2001] defines a framework for applying usability research methods throughout the overall product development lifecycle. She focuses overall on techniques to be used before the design or after the coding, like ethnographic interviews, retrospective work evaluation, competitive evaluation, natural use observation, and focus group (Figure 10).

In her work she basically reveals the fact that even if usability techniques are slowly emerging in the design, they are not at all present in the marketing department and we do not use the previous version release to increase the usability of the current software. So people's opinion needs to mature to usability.

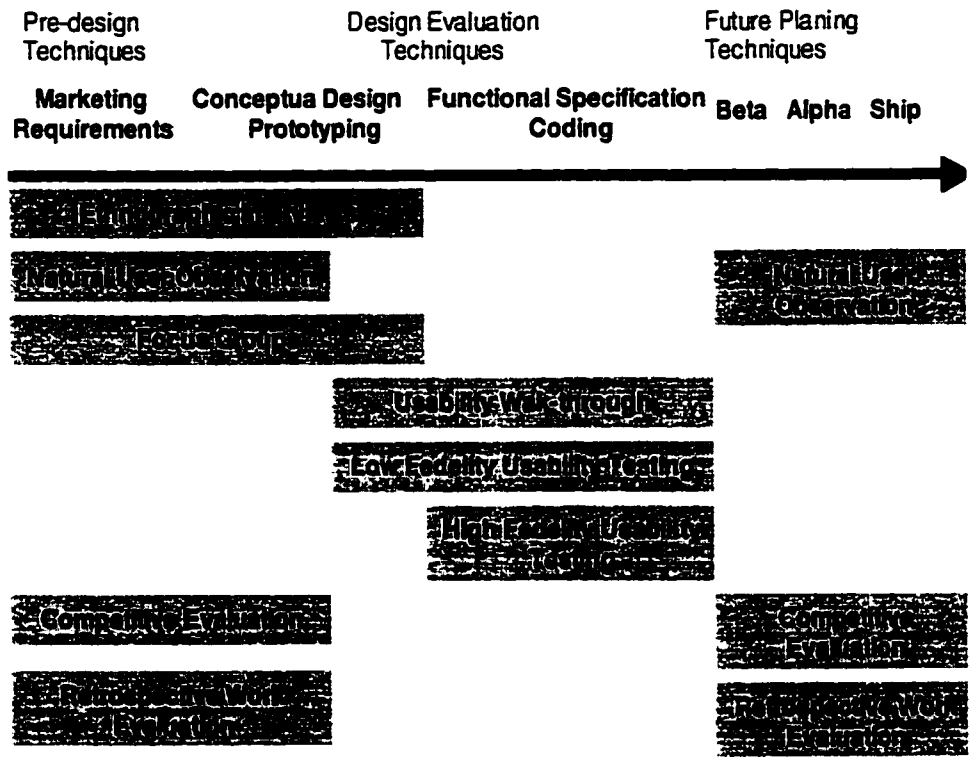


Figure 10 Usability Research Methods in the Development Cycle

1.2.12 Survey Summary

All these investigations are significant improvements to a specific software development process in a certain organizational context. However these restrictions on the process make the improvements hard to re-use and extend to other SDLC. For example some of these techniques focus on the models/artifacts by postulating traditional software

engineering processes and teams polyvalent. They will have a much more limited impact on teams that do not have any experience in usability engineering (designer defining goals like system-requirements).

Also, even though these works are very different, many solutions and concerns are repeating:

- The influence of the minding (culture + environment) of the designer on his work, what we can call the human dimension of development.
- The need for an integrated structure in the engineering process, also called the activity or process dimension of the development.
- More specifically the need for appropriate modeling constructs (refine the use of use-cases), the artifacts aspect of the development.

We need to find/create a structure to sort these techniques or concerns, in order to appreciate better their contributions; and to eventually integrate them into a framework.

1.3 Towards a Formal Framework for Integrating Usability Cost-Effectively in Software Development Lifecycle

Our investigations highlighted the need for a formal framework for defying, studying and validating the integration of usability concerns and human factors in general in the overall software development lifecycle. This framework would enforce the absolute requirements for a proper integration of usability concerns in any software engineering process. The framework should also consider the integration at the human/team, the activity/process, as well as the artifacts/models levels. The following are the four dimensions that should be considered:

- Activity dimension including synchronization between activities
- Actor dimension including team organizations and communication schemes

- Artifact dimension including the formats and structure of the artifacts
- Tools factors dimension including the use of tools to facilitate the communication.

1.3.1 Integration by Artifacts

Jarke points out that scenarios are used in software engineering as intermediate design artifacts in an expanded goal-driven change process. They provide a task-oriented design decomposition that can be used from many perspectives, including usability trade-off, iterative development and manageable software design object models. Constantine suggests that use case specifiers first prepare lightweight use case model descriptions (*essential use cases*) that do not contain any implicit user interface decisions. Later on, the user interface designer can use these essential use cases as input to create the user interface without being bound by any implicit decisions. Krutchen introduces the concept of *use case storyboard* as a logical and conceptual description of how a use case is provided by the user interface, including the interaction required between the actor(s) and the system.

Nunes proposes to annotate use cases using non-functional requirements at the level of abstraction at which they should to be considered. Artim for example, emphasizes the role of task analysis by providing a user-centric view of a suite of applications, and then emphasizes use cases by providing each application with a method of exploring user-system interaction and describing system behavior. Ralyte in the CREWS project develops a framework for integrating different kinds of scenarios into requirement engineering methods. Rosson proposes combining the development of tasks and object-oriented models, which are viewed as a refinement of rapid prototyping and an extension of scenario-based analysis. Later on Nunes in Framework for interactive system also takes to integrate the two processes by doing a consistency check of the artifacts leaving the processes to be loosely/indirectly coupled.

These Enhancements demonstrate the need to have two models (one for the user information, one for the system information) that are evolving in concurrence. The need

of two different models is justified by the fact that because the information is of a complete different type.

1.3.2 Integration by Activities

Rosson changes the design activity by basing it on usage scenarios, and then iteratively elaborating user interaction scenarios and collaborating computational objects. The changes may occur at the Object-by-Object analysis, for which we may refine the user interaction scenarios and computational objects. Artim bases the design process on two independent but interrelated models (a problem specification and a solution specification) that are iteratively updated during the end of cycle assessment of the impact. Nunes has also two independent models: the analysis model and interaction model, that are evolving in concurrence, he does not specify more explicitly the refinement process leaving it up to the process manager.

These structures demonstrate the need to have two models (on for the user information, one for the system information) that are evolving in concurrence, because we need to refine the models to converge to the solution that corresponds to the best trade-off.

1.3.3 Integration by Actors

Culled from our day-to-day experience, four different ways, for involving usability expert in the software development teams, are possible:

(1) Resort to third part companies specialized in usability engineering,
(2) Involve a consultant expert in usability,
(3) Form/create a usability team, and finally
(4) Provide training to some members of the development team that can act as the champions of the usability.

In small projects, the resort to a third-part company or to a consultant plays the trick, but departments doing regular software development would rather have usability experts either state-of-the-art human factor experts or reconverted software engineers. This implies also that these usability/human-factor experts have to be somehow integrated in the global software development process of the companies. Artim pointed out in his case study, the problems linked to the culture of the software engineers that does not include collaborating with the user. Bevan pointes out the absence of focus on usability from the certain levels of industry. Well know difficulties comes from educational gap, use of different notations, languages and tools, as well as the perception of the role and importance of the design artifacts.

For big projects we need to have an on site team, dealing with the usability problems. Whatever the approach chosen for involving usability engineers in the software development lifecycle we can characterize the functioning or some dysfunction in the work pattern. The difficulties of communication between the software development team and the usability specialists could seriously compromise the integration of the usability expertise in software development lifecycle.

1.3.4 Summary of the Aspects to be addressed

We can distinguish different dimensions of the development process that have to be considered for a cost-effective integration of usability in software development lifecycle. Most of the works we examined just focus are had doc solutions. However, they highlighted four levels for integrating usability in the software development lifecycle:

- The documents/artifact
- The process/activity
- The person/actors
- The support resources/tools

1.4 Complexity of the Integration Methods

We are at an early stage of our investigations and so the existing methods may be simpler than eventual upcoming ones. However we can already differentiate in integration methods simple modifications from more complex interaction schemes. The importance (in size and relevance) of the amelioration/change is not related to its complexity. We will discard the methods that change completely the SDLC.

1.4.1 Simple Modifications

These are modifications that do not require big changes (structural changes) in the existing SDLC.

What we called **elementary modifications** are modifications that affect only one element of the SDLC. For example we can change a very specific artifact, and so indirectly the way to perform the corresponding activity. No change in the main communication line is needed. A typical example is what Krutchen called 'use-case storyboards'. Other atomic methods can be the adding of usability inspection in certain activities. Per se, it is very often not sufficient for a full proper integration, and comes with more important changes.

Complementary modifications consist in adding new elements in the SDLC without changing the existing communication line. Darnell suggested an example of these complementary modifications. She proposed a framework for using usability research methods (usability walkthrough, user testing) in the SDLC. Her complements to the existing SDLC and does not force a change in the structure.

1.4.2 Structural Modifications

Structural modifications affect the communication line or the structure of the SDLC. They add a set of activities in the SDLC that can change the sequence of the activities. Artim initiated a new communication line based on the interaction between the problem

specification and the solution specification. Constantine sets the communication scheme to outside in with his usage centered-design. Nunes adopts a loose communication scheme that is only constraint by the need to support the dual model (analysis model, interaction model).

1.4.3 Advanced Modifications

Advanced modifications cannot be described by only a change in the communication line. Usually, they come on the top of a structural change. They can correspond to many different things:

- Patterns on how teams should be organized (organizational patterns of [Coplien 95] and [Cockburn 96])
- Recommendation on how certain people should organize their work (Cockburn's communication patterns)
- Warnings about common error/mistakes in the process, and how to avoid them (e.g. software engineers tend to mix-up doing task analysis)
- Techniques, actions to solve for a specific software process problem taking into account the forces factor [Coplien 95]
- Etc.

These modifications commonly correspond to the manager's savoir-faire, or his instinct on how to do things.

Chapter 2

Integrating Use-Case-Driven and User-Centered Requirements Engineering Processes: A Case Study

An early version of this chapter was submitted and accepted to the conference IHM-HCI 2001.

2.1 A brief Description of the Processes Investigated in Our Case Study

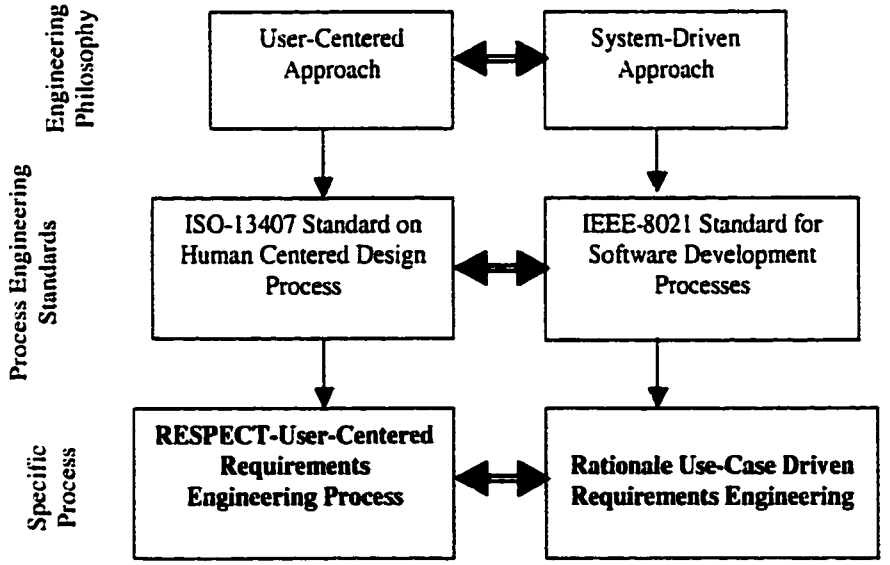


Figure 1bis View of our Research Case Study and Framework

As starting point of our investigations and a research case study, we are considering the following two requirements processes (Figure 1):

- The use case driven requirements workflow as defined in the Unified software engineering Process (UP) proposed by Rational Software Inc [Booch, 1999].
- The RESPECT framework (REquirements SPECification in Tematics), which is concerned with the capture and specification of end-user requirements (Maguire, 1998).

2.1.1 Capturing User Requirements as Use Cases in the Unified Process

The goal of the requirements process, as defined in the unified process (UP), is to describe what the system should do in terms of functionality, and allow the developers and the customer to agree on this description. Use cases are the most important requirements artifact. They are used by (1) the customer to validate that the system will be what is expected in terms of functionalities, and (2) by the developers to achieve a better understanding of the requirements and a starting point for technical design. A Use case storyboarding, which is a logical and conceptual description of how use cases are provided by the user interface, includes the required interaction between the user(s) and the system. Storyboards represent a high-level understanding of the user interface, and are much faster to develop than the user interface itself. The use case storyboards can thus be used to create and evaluate several versions of the user interface before it is prototyped designed and implemented [Krutchen 99].

One of the weaknesses of use case driven requirements workflow is that the use cases attempt to describe representative ways in which the user will interact with the software but is not comprehensive. Another weakness of this process is that the main people involved in this process are stakeholders and technical persons including use case specifier and user interface developer. End-Users are not directly involved. Use case specifier details the specification for a part of the system's functionalities by describing the requirements aspect of one or several use cases.

2.1.2 User Requirements Engineering in the RESPECT Framework

RESPECT is a user-centered requirements engineering framework developed by European Usability Support Centers. The RESPECT process is a concrete implementation of the iterative user-centered design process for interactive software suggested by the ISO-13407 Standard [ISO 13407:1999]. The RESPECT process starts from the point where project is summarized from the end user point of view. By the end of the process, it is produced different text-based forms that detail the user interface, user support and help, the physical and organizational context, equipment and hardware constraints, usability goals that must be achieved, as well as the system installation procedure.

Although RESPECT is a highly detailed process for capturing and validating context of use and usability requirements with the active involvement of end-users and stakeholders, the text-based forms produced are not easily understandable by software development teams. They are also a source of ambiguity and inconsistency, especially when they are compared to the use cases.

2.2 Principles for Integrating the Two Processes

The framework postulates that we keep both process separated because the subtle functioning of the teams have to be preserved as much as possible. So **the framework describes the essential set of communication points between the two systems/teams.** We could try to integrate the two engineering philosophies more but it would cause many malfunctioning due to the antagonism of the two philosophies (e.g. actors mislead by different wording, discard of some details of the models) as explained in Figure 1. We don't have now the tools to solve this malfunctioning.

Our first step for improving and mediating software-to-usability integration involved identifying complementarities between the use-case requirements and RESPECT processes. We lead the use case following the three supposed dimensions, and we isolated four principles outlined below summarizing these complementarities.

2.2.1 The Artifact Dimension

Firstly, RESPECT captures a complete description of the context of use including user characteristics, task analysis, as well as the physical, technical and organizational environments in which the system will be used. Although in theory use cases have the potential to gather the non-functional requirements that are a simplified description of the context of use, in practice, use cases have been used for gathering the system functionalities and features including technical capabilities and constraints. Therefore:

> ***Principle 1:*** Context of use and functional requirements should be considered as two views of the requirement picture. The software view on this picture is a set of artifacts describing the functionalities and the technical requirements of the system. The usability view is a set of artifacts describing the context of use and the usability goals/factors in which the functionalities will be used.

To a certain extent, this principle means that both the software and the usability views are important. Table 1 indicates the software and usability views for each of the processes that we considered in our case study. Such classification of the artifact can facilitate the identification of potential relationships between artifacts.

Table 1 Relationship between RESPECT and UP Requirements Artifacts

	RESPECT	UP Requirements Workflow
Software View	General system characteristics System functions and features User Interface	Use Case Diagram Requirements attributes Boundary class Use case storyboard User interface prototype
Usability View	Organizational structure Task scenario and interaction steps Technical environment User support Physical environment Social and Organizational environment	Stakeholder and Users needs Additional Requirements
Other artifacts that cannot be classified	Standards and style guides to apply Test plan Implementation plan	Vision document Glossary

Secondly, in RESPECT, the context of use is described using a non-formal notation, which is easy to understand by end-users and stakeholders. However, these forms are a cause for inconsistency and ambiguity when used by software developers. The artifacts that are produced and the semi-formal notation used in use case approach are more understandable by software developers. Use cases as a notation can also support, in a certain extent, automatic generation of code [Krutchen, 1999, Booch, 1999].

Principle 2: As [Artim 98] discussed about "one model, but many views and notations." We strongly share his belief that different notations for the same concept may foster communication between persons. This means that we can use different notations to describe the artifacts related to the functional and context of use including text-based forms and use cases. However, this requires maintaining the correspondence between multiple views at an abstract level using a high level notation.

2.2.2 The Activity Dimension

In RESPECT as in other similar approaches, usability specialists use the *context of use* as an important input for usability testing. Software developers use the functional requirement artifacts as a starting point for technical design and implementation.

Principle 3: A common step to the two processes should include activities for reviewing and validating the integrity and consistency of all requirements artifacts from both the usability and software views. After validation, we should generate a usability testing and implementation portfolios.

For example, the usability-testing portfolio should include the entire usability requirement artifacts that will be used during usability testing. The implementation plan should include the artifacts that required for implementing the system.

2.2.3 The Actor Dimension

Fourthly, it is important for usability-to-software engineering collaboration and for consistency and coherence of requirement artifacts to gain a high-level understanding of the system, this from the beginning. Therefore:

> ***Principle 4:*** The requirements should start when a representative set of users and/or stakeholders are invited to summarize the system from the future user's perspective. They are mainly asked to answer different questions that we organized in a system summary form. Users and stakeholders, the main contributors during this step, are invited to give brief answers to these questions. All completed forms are then analyzed and compiled in a unique system summary form by usability engineers. This compiled form is approved by software developers, stakeholders and users. It is used as a roadmap during the requirement process and represents a general consensus on the system.

Table 2 is an example of the system summary form that we developed. User-centered requirements frameworks such RESPECT and use case-driven approach supporters [Constantine 99] suggested similar questions.

Table 2 An Example of the System Summary Form

Questions	Assumptions
What is the purpose of the system?	ISO 9000-based quality system over an Intranet
Why is this system necessary?	Supporting the development of the company outside the country (new clients, remote offices.)
Who will use the system?	Employees and some of the company's clients
What will the users accomplish with the system?	Access to quality procedures and associated forms Learn the quality system and the ISO 9000 standard
Where will the system be used?	Standalone workstations and personal digital assistants
How will users learn to use the system?	Introductory course and online assistant
How will the system be installed?	By a Webmaster for the server version, and by employees on their PDA (download from the server)
How will the system be maintained?	By a Webmaster and a quality control manager

2.3 A Framework for User-Centered and Use-Case Driven Requirements Engineering

We will describe first the schematic process view of the framework or architecture, and the results obtained during the study to, then draw conclusions.

2.3.1 Process-View of the Integration Framework.

Based on these principles, we iteratively defined, used and validated a framework for improving software-to-usability engineering integration (Figure 11). This framework clarifies how usability expert activities can be incorporated in the software development lifecycles. It also clarifies the relationships between activities done by software engineers and usability experts.

Mainly the framework has been used in 10 projects conducted at CRIM (Computer Research Institute of Montreal) between 1997 and 2000. All the projects are related to Web-based interactive systems including, for example, an environment for managing ISO 9001 documentation, a tool for sharing resources as well as a Web-based training system. RESPECT and use case-driven approaches were used simultaneously by software and usability experts. At the end of each project, a series of ethnographic interviews where all participants were interviewed was conducted. We asked them to describe their activities during the projects and to highlight the difficulties in term of communication. We also reviewed the framework with all participants and asked them about potential improvements.

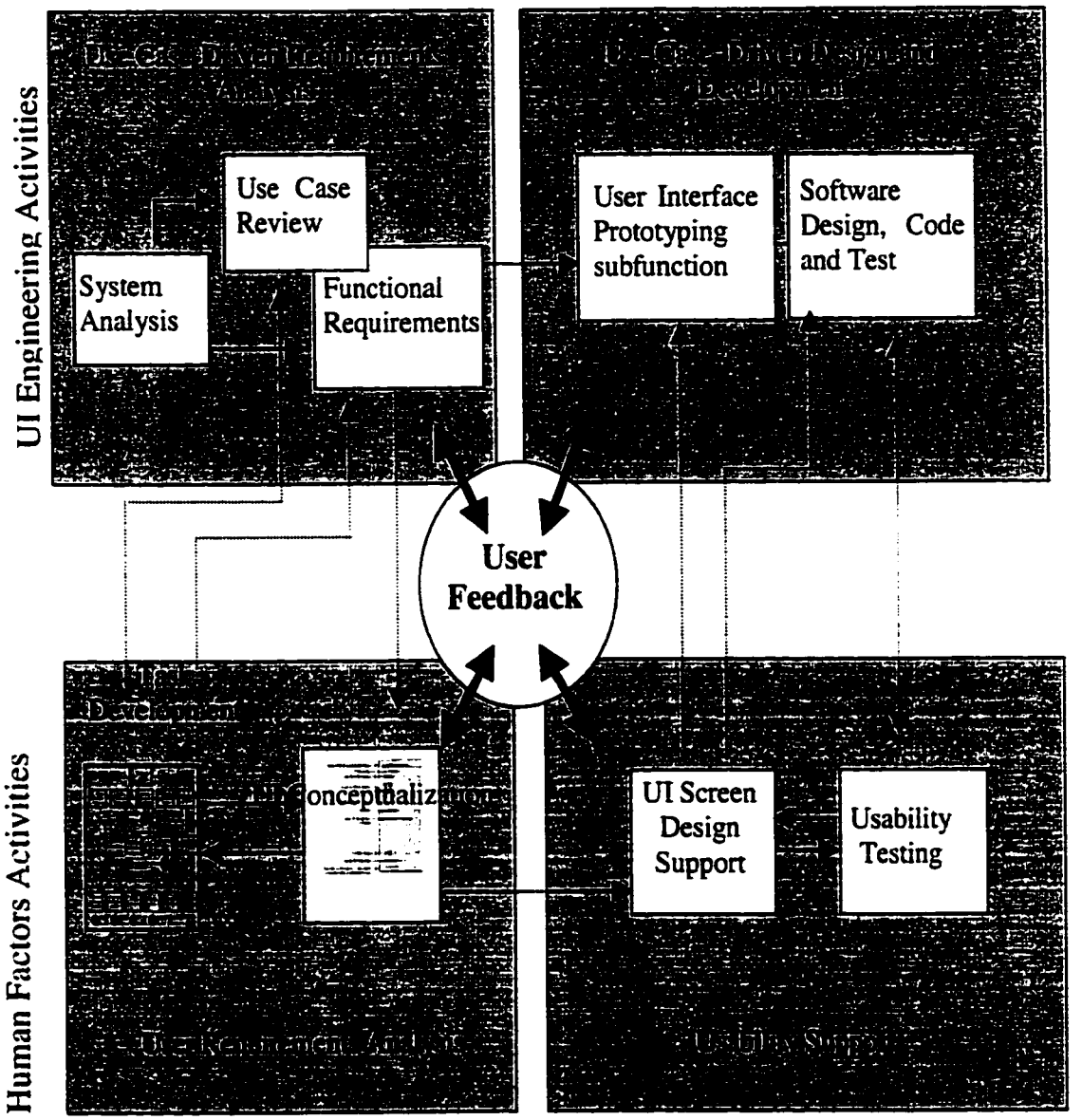


Figure 11 A Process-View of the Framework for User-Driven and Use Case-Based for User Interfaces Engineering.

2.3.2 Summary of the Case Study

In this part, we presented our investigations on how to improve and mediate the communication between usability expert and software development teams. With respect to experimentation, two specific processes constitute the focus of our interests: use case-driven and the user-centered requirements engineering processes. Further to the framework for improving software-to-usability engineering integration we defined, we identified the following principles that we consider as critical issues.

First, the requirements of an interactive system must be defined on two levels, but not independent of one another, as it is today. The first level is concerned with the specification of the context of use, and the second focuses on functional requirements. Different specification notations may be used for the two levels, but they should exploit an integrated representation of all the requirements artifacts. In our case, we adopted the text-based forms as used in RESPECT and the graphical representation of use cases as defined in Unified Method Language.

Secondly, the list of artifacts describing the context of use ensures a good usability specification. Better still, this list can assist with generating functional requirements, at least to a limited extent. This result is fundamental because it can minimize requirements artifacts inconsistency and improve communication between software and usability engineers.

Thirdly, the integration process should add activities to support the two levels of requirements stipulated in (principle 1). These activities will implements the reviewing and validating of the integrity and consistency of all requirement artifacts. The automation of the activity can be done depending on the form and syntax of the requirements artifacts.

Fourthly, the approval of the documents by the users and the stakeholders is necessary to guarantee the usability of the product. However trivial, it remains a keystone to the

integration. To keep the good properties of the requirements, the users and stakeholders have to be omnipresent in the software engineering process as well as the usability process.

2.4 Deductions from the Case-Study

These conclusions will allow us to study the veracity of the assessments about the dimensions of the integration made in part one. We will try to "relativize" the notion of dimension. Later we will study the implication of these statements.

2.4.1 Validation of the Model

The four principles or milestones for integration that we found can be organized following the three dimensions, which consolidate our postulate:

1. Two levels of the requirements is mainly in the Artifact dimension
2. Different notations for the same concepts is also in the Artifact dimension
3. Reviewing and validating requirements is in the Activity dimension
4. Involving users and stakeholders in the requirement process is in the Actor dimension

As we can see, even if any integration has to exist on all of the three dimensions, we seem to be able to characterize the integration quite independently on each dimension. For example, involving the users (and stakeholders) in the requirement process is an integration that implies changes in all of the three dimensions. We need to

- create/change the *artifacts* to support user validation,
- create/change an *activity* to involve the users in the process, and ultimately
- bring this new *actors*, and have them interact with the already-involved ones.

This integration is more present in the actor dimension. It illustrates the need to a communication line in the process.

The case study was supposed to extract standard milestones for a cost-effective integration, and it came-up with milestones on all different aspects extracted in the first part of the thesis. **So the a priori decomposition of the integration following three dimensions seems to be relevant and useful to characterize any integration in an eventual framework.** The actual relevance and usefulness of this approach can only be determined by a thorough study, and is not part of this research.

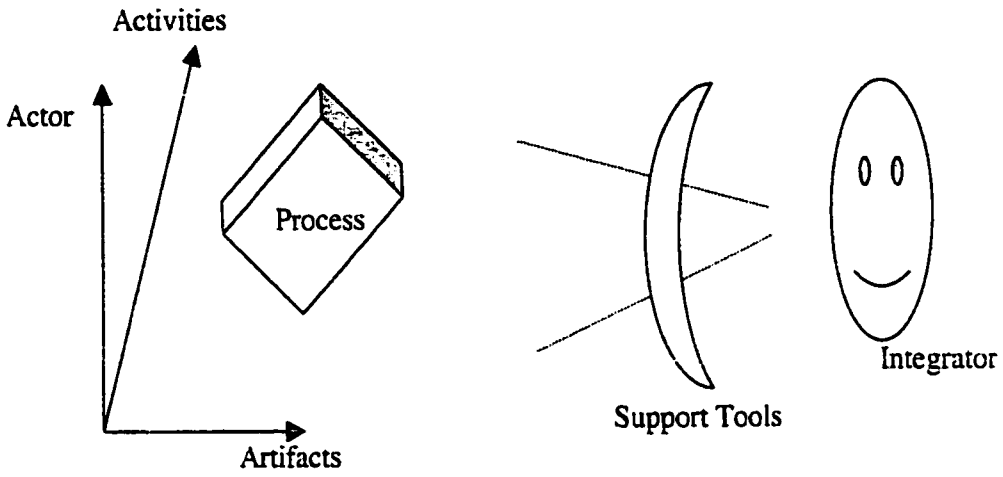


Figure 12 A Schematization of the Three A's and the Support Tools Model

2.4.2 Use of the Dimensions in Cooperation

We just showed that the integration is describable in three different dimensions or aspects. The simple communication lines just need a description in one dimension and more sophisticated could hypothetically use many. Previously integration methods just focused on a specific aspect of the integration to avoid the complexity in adjusting the organizations of each dimension. Very often the integration was done on the artifacts, but the secondary implied changes to be done one the activities and actors dimensions were just briefly mentioned. It is normal at an early stage of the research, but is rather incomplete at the implementation stage. This can be explained by the fact that the SDLC

has a general architecture (typically the 'process' description of the SDLC) and this architecture is somehow present in all dimensions. But when we need more granularity and accuracy in the description, the study of a single aspect/dimension of the integration is not sufficient.

In fact, more than studying the existing integration methods, we need to provide a framework that would help people willing to integrate usability in their SDLC (ultimately any feature in their SDLC). So we could provide a framework (and eventually constructs) to describe the organization on the three dimensions and then the interaction between these dimensions.

2.4.3 Complexity of Integration

As is pointed out in [Cockburn 96] and [Cockburn 2000], software engineering is reaching projects where the communication and human-factors are taking more and more importance. [DeMarco & Lister 87] was already talking about the subject. The case study came up with mainly principles that deal with the need for changing the structure of workflow/communication line. The postulate that we need to keep both processes separated not to loose subtle organizations in the process, depict the concerns that we had, not to be able to manage these organizations. These underlying structures of the processes are harder to describe, due to the complexity or fuzziness of the elements. We need for this to go through schemes or patterns.

This has to be included in our framework in order to formalize eventual changes in these structures.

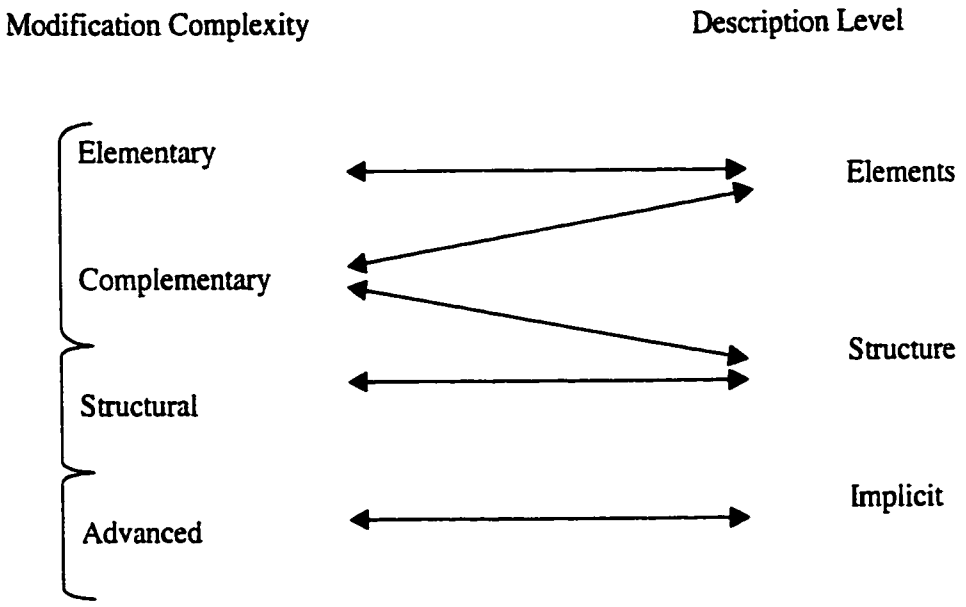


Figure 13 Complexity of Modifications and the Corresponding Level of Description

- In the **elementary level of description**, we describe each element, so the basic level of description.
- In the **structural level of description**, we describe the communication line or structure of the process.
- In the **Implicit level of description**, we will need to describe implicit organization of many forms (some times organizations of organizations), an a priori choose of patterns as form of description seems interesting. It will be required to solve the detected communication/organizational problems between teams, as well as for any detail specific to the process (complex modifications), it needs to contain what we called the manager's savoir-faire.
- A complementary modification needs to change both the communication line/structure (in a very light manner) and to create a new element by describing it.

An eventual framework has to cover all this levels of description, in order to do any of the levels of modification.

2.4.4 Framework for Integration

The framework is usually needed when a researcher or a software engineer wants to integrate usability (or other features) in his theoretical or already existing SDLC. This framework has to help him to envisage all aspects of his integration, and to formalize it. We can divide it into several steps:

- Visualize and understand the communication line (as a sequence of steps) and the global organization on three dimensions plus the eventual helping factors (as simple relations between elements).
- Describe the implicit organization of the existing SDLC, change or add new patterns and link them to the elements it is related to.

Trial and errors can be a useful approach, the framework should favor it.

Chapter 3

Tools for Supporting Software-to-Usability Communication and Integration

In this chapter, we describe an XML-based tool that helps to identify, study, and mediate software-to-usability communication milestones while supporting the integration of usability features (and eventually others) in an existing SDLC. This chapter also introduces the concept of patterns as a means for gathering and disseminating the best usability integration solutions in existing SDLC.

3.1 XML-Based Tool

The XML-based tool, required to fit the needs mentioned above, must describe an existing SDLC in such a manner that the general communication milestones can easily be identified. The details can be found in the following subsections.

3.1.1 The Proposed Process Description Language

As described in [Booch 97], a process is a set of who (actor), what (artifact) and how (activity). More clearly, *Actors* write *Artifacts* during *Activities*. For representation purposes, we will use the concept of *steps* as a construct to organize activities. Also as part of the description of a process, we will describe in a formal way the relationship between these different components of a process.

3.1.1.1 Actor

An actor is basically a person that performs a task in the current process, usually an activity or part of an activity, either alone or within a team. Any actor is identified by his name (or a codename like "usability engineer 2"). We can associate certain constraints with each actor by including a short text in the description of the actor (e.g. "needs to be a Human-Factor expert", "needs to be critical", "needs to have strong management skills", "cannot also be implied in the design phase", etc.).

The description above is sufficient for most of the modifications. However for more advanced modifications, certain descriptions may mention the actor's position, knowledge competencies, communication skills and social aptitudes among others, which are useful to describe the organization of actors within the three kinds of networks:

- The **formal hierarchical/management network** aims to set goals and to act such that these goals are achieved. The nodes of this network are the managers and the leaves their subalterns. These hierarchical positions and job titles loosely determine the actor's domain of competence and thus, the tasks an actor will have to perform.
- The **informal information network** organizes the knowledge and competence. This network is needed to perform complicated tasks that one person alone cannot perform. A node is called "nerd" or "expert in". The knowledge, experience, competence or savoir-faire influences an actor's performance in a specific task.
- The **informal social/communication network**, organized by character type or interest field, regulates the team's work to avoid malfunctioning such as communication problems or conflict. The important nodes in this network are 'influence guys', secondary nodes are 'nice guys' and 'comprehensive guys', and nodes with small weight are 'nerds', 'new comer', etc. An actor's communication skills and social aptitudes determine how he can handle a communicational or organizational problem between people.

3.1.1.2 Artifact

An artifact is a document written during one or many activities by the corresponding actor(s). One actor is responsible for an artifact, so that this actor may be contacted in case others require explanations about it during the process. An artifact usually has a format or language that standardizes its form, to avoid confusions or misinterpretations (e.g. UML). Templates may be provided to simplify the creation of the artifact, as well as other resources like FAQ or previous projects.

The artifacts of a process may or may not be organized, depending on the exact process. However, it is strongly recommended to organize them with any document (e.g. use-case map) that summarizes the different artifacts and provides access to them as well. [Constantine & Lockwood 99] provides a good example by organizing the documents in five sets, namely user-role, content, use-case, domain, and operational models, and giving three maps to structure them, specifically user role, navigation, and use-case map. However, [Cockburn 97] provides a very different structure that also has very good properties and fits in only one map (cf. Figure 8). There is no common number nor common format for maps. Thus, we must postulate that the maps in the models must be defined as normal artifacts and attached to activities and actors, as would a normal artifact.

3.1.1.3 Activity

An activity is the description of how a set of actors gets the necessary information to write one or many artifacts. We can have many resources to support the process, such as guidelines on how to perform the activity, previous records of the same activity, etc. An activity has a current status that determines whether it is performed or not (more formally: unassigned, assigned, completed or approved). Comments describing the events that have taken place so far can be added to the activity, thus helping the 'users' to perform the activity.

The activities are organized using causality relationships (e.g. successor, predecessor) that partially order the set of activities, thus determining possible sequences of activities. Bad flow charts of activities can lead to impossible completion of the activities. Commonly used processes are iterated to ensure the quality and quantity of the results. Very often, people define a sequential or total ordering of the activities to simplify their management. However, others view the aforementioned approach as too restrictive and rather prefer slack approaches [Nunes & Cunha 99] that merely synchronize the activities at specific consistency checking points. In our Use-Case (Part II), we defined a process composed of two parallel sub-processes, linked together at milestones. The syntax must provide facilities to create this type of link. This link will indirectly determine activities that are critical bottlenecks to the workflow.

3.1.1.4 Global Structure of the Process

For clarity, we provide the abstraction construct *steps*, which allows to structure the activities into an ordered tree. The steps are a totally ordered set, so they are easier to represent and comprehend; the order depicting the communication line or general information flow in the process. However, the user should not misinterpret the sequence of steps as an organization between activities, artifacts or actors; rather, it is just a classification: actors and artifacts can appear in different steps, and activities can be related or linked to activities in different steps.

Complications may occur when changes are made during the integration of usability features in a process. In our case study, a dual information flow is needed (Figure 11). If we begin the simplification into a unique information flow, it will influence organizational structures and spoil implicit structure in the two already validated models. So the construct has to allow for a complicated information flow (such as the use of two parallel information flow, or a very loose one).

3.1.1.5 Standard of the Process Structure: XML

The process description specified above is the core of the Framework. We need the following features in upcoming version of the tools:

- To Open/Save descriptions of processes (so process engineers can smoothly move back and forth in the entire process reengineering lifecycle).
- To Import/Export descriptions to other workflow systems (other SPEG tools)
- To allow for Scalability on any dimension (in case of increase of the process' size)
- To allow for Portability of the files in the Internet (implies self containment of a process and that's a good asset to have)
- To allow for speed in loading the process (another good asset to have)

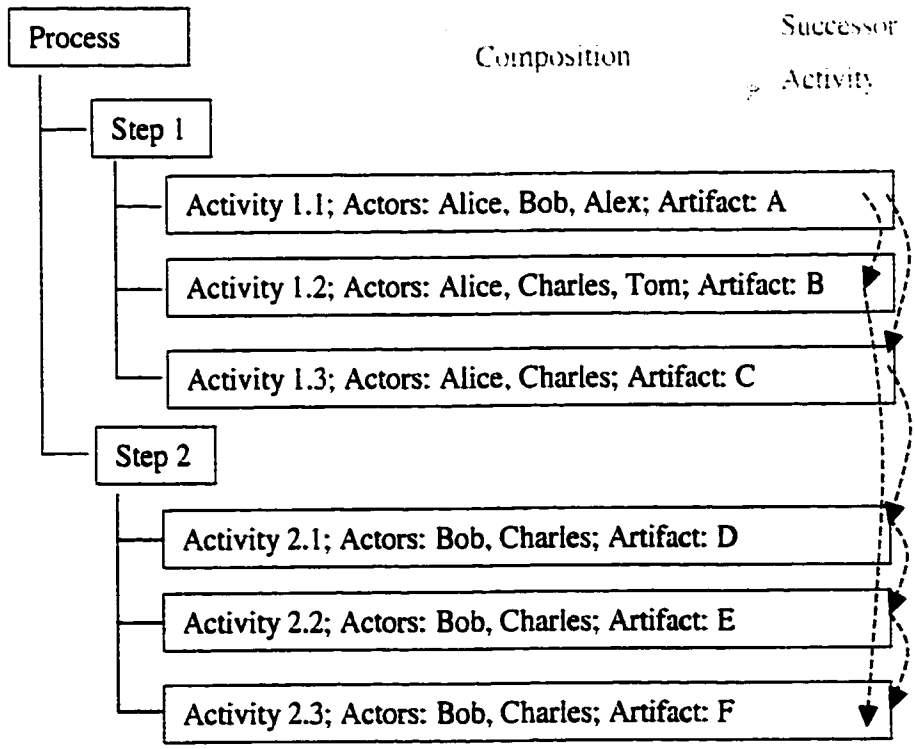


Figure 14 Organization of the Process Structure

A language like XML can achieve the features described above. We can easily save and open text-based files from any support. It can translate to HTML and other common data types by using existing or upcoming conversion tools. An optimized specialized library can do the loading, thus limiting the slowness of the Java/XML couple.

XML technology favors the presentation of the data as a tree, so we will first organize the data upon the global structure of the process: the organization following processes, steps, and activities (Figure 14).

For more details about the Process Description Language, see the DTD presented in annex 1.

3.1.2 Our SUCRE Framework

Software-Usability Concurrent Requirements Engineering (SUCRE) aims at supporting the Integration Framework. We will first study its requirements, and then focus on the tool itself.

3.1.2.1 Brief Description of the User

In the versions of the program that we will use for our research (called early versions or early phases of development), we will consider the integrator as the only user.

The integrator can be a researcher in the field of software/usability engineering or an actual member of a SEPG (Software Engineering Process Group). He is literate, knows how to use a computer, and excels with the notions of process, activity, artifact, actor, etc. He also has knowledge of software engineering and usability engineering processes.

His goal is to improve an existing process by adding new features to it. He will follow a methodology, similar to the one described in 3.1.3.

3.1.2.2 User Requirements: Prioritization of the Needs

As seen previously, the user will have to perform many tasks with the tool. An ordering of these is needed to structure the development process. Our priorities for this research framework can be summed up by the three sets below:

The core features:

- Concurrently Visualize the structure of one or two processes in three dimensions
- Modify the communication line by changing the predecessor/successor links between the activities
- Provide flags/status to support a walkthrough of the process by the integrator
- Open a process, Save the modifications under a different name

The secondary features:

- Define processes via new activities, actors, artifacts, resources, etc.
- Provide alternate ways of displaying the process structure and links (toggle between views)

The peripheral features:

- Provide printing support
- Provide a well-furnished library and help facility
- Support a walkthrough of the process by different actors

Defining processes is set as a secondary feature, since we can hardcode a process directly in XML via any word processor. The program structure is designed to support all the features, but some functionality will not be fully developed before ulterior versions of the tool.

3.1.2.3 Requirements for Visualizing a Process

The aim of the tool is to best assist the user in the understanding of a process: the interface needs to be intuitive, with a good conceptual model supporting both the representation of the process as well as the ease of changing it. This implies a graphical user-interface, with mouse-based interaction.

The interface has to allow for the comparison of two processes (e.g. concurrent scan through). The conceptual model has to facilitate this by simultaneously displaying both processes, in an identical manner with identical interaction schemes.

Since the user knows how to use a computer, look-and-feel constructs from existing platforms can be used to reinforce the intuitiveness of the tool, like standard menus, toolbars, and directory-like tree descriptions.

3.1.2.4 Technical Requirements: Java Swing

The implemented tools will remotely access an XML description file. The portability of the tool would be an asset but not essential. Rather, we will stress the technical requirements on the need for an evolving structure. The term "evolving" is used for the need to add new elements on the process description, the need to have tools that display as many significant aspects of the process as possible, and the need to determine the look and feel of the interface (will take long to settle on).

As we noticed here the changes in upcoming versions will change overall the interface and the database (XML file), so at design-time we could consider an MVC architecture for the program, thus separating the two unknowns.

3.1.2.5 Description of the Tool Prototype

As proposed in the requirements we divided the interface into two equal zones for two eventual processes (Figure 15). The interactions in the top half and the bottom half are the same.

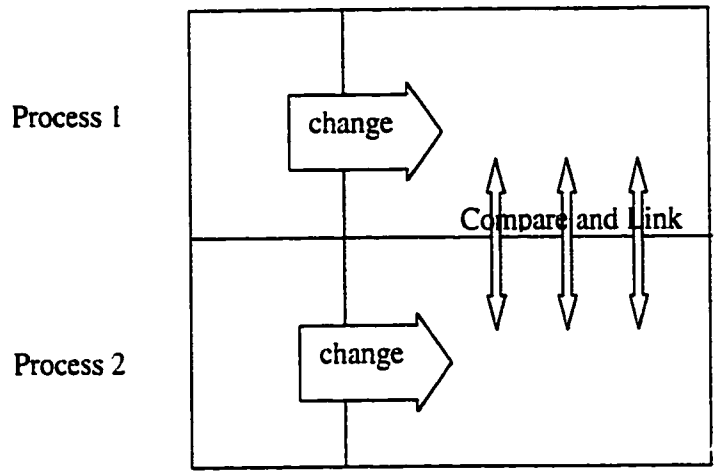


Figure 15 Conceptual Model

The toolbar and the menu provide the traditional open/save/print features, as well as the cut/copy/paste edition tools. Also, other tools are provided to toggle views and browse the process structure.

The left area contains a directory-like tree description of the process structure (cf. figure 14 and annexes) with the process as the root node, which opens into a list of steps (as subdirectories) and the steps open into activities. When we click on a step or an activity, the details of the selected item is displayed on the right area of the sub-window in the currently selected visualization mode.

The right area contains detailed views of the processes or steps, in the current visualization mode. Details specific to one object can be obtained by right clicking on the object. Two display modes are available, specifically matrix display or graph display.

The matrix representation was the only one fully developed. The matrix representation displays the relations between the entities in a condensed or expanded manner. The following views were found to be more pertinent in the process understanding.

- STEPS X ACTIVITIES CONDENSED (early version in Annex 1) displays all process steps along with the activities that belong to them
- STEPS X ACTIVITIES EXPANDED (early version in Annex 2) displays the distribution of all the activities status inside all the steps
- ACTIVITIES X ACTORS CONDENSED displays all process activities along with the actors involved into them
- ACTIVITIES X ACTORS EXPANDED displays the distribution of all the activities status for all the actors involved into them
- STEPS X ACTORS CONDENSED displays all process steps and actors along with the actors that belong to them
- STEPS X ACTORS EXPANDED displays the distribution of all the steps status for all the actors involved into them

As required, the architecture separates the model from the view (annex 4), thus easing the modification in the interface and the process description standards.

3.1.3 Matching our Requirements

We came up with a tool that still has to be tested in working conditions. However this tool does not support the description of implicit details, which is essential for a good framework. So we need to study eventual complements to this framework.

3.2 Patterns for Integration

As recognized previously, a process contains implicit structures that are not easily describable, but that we need to formalize somehow in order to provide a useful framework for further investigations and validation. This section explores how some of

the existing pattern languages can be extended and generalized so that they are used as a tool for supporting integration.

3.2.1 Definition of Patterns

The concept of pattern was defined in [Alexander 77]; it describes a way to solve a specific problem, in a precise context. It has been used in many different forms. A good Pattern should have six properties [Lea 93]:

- Encapsulation: Each pattern encapsulates a well-defined problem/solution.
- Generativity: Each entry contains a local, self-standing process prescription describing how to construct realizations.
- Equilibrium: Each pattern identifies a solution space containing an invariant that minimizes conflict among forces and constraints.
- Abstraction: Patterns represent abstractions of empirical experience and everyday knowledge.
- Openness: Patterns may be extended down to arbitrarily fine levels of detail.
- Composability: Patterns are hierarchically related. These relations include, but are not restricted to, various whole-part relations.

Patterns are numerous and diverse, and a listing of them would only be partial, since a pattern varies with respect to the specific context it is being used in. This implies that we have to stay very generic in our framework to be able to incorporate any kind of patterns. However we did a small review of interesting patterns that could complement our framework. Some of these are not patternized yet but will be in one form or another.

In what follows, we briefly discuss how patterns can support the integration at the four levels of integration we discussed, namely in including them in actor, activity, tool and artifact dimensions. We also provide some examples.

3.2.2 Actor Dimension

The people-related patterns touch many aspects of software development (hierarchical organization, information, social and communicational aspect, etc). Some patterns are to be used by individual actors for self-management purposes, such as:

- Pedagogical Patterns [Sharp, Manns & al. 1996]. The aim of pedagogical patterns is to capture and disseminate experiences of learning and teaching object technology. It can be used individually by people willing to learn fundamental concept of usability.

Other patterns focus on interaction networks between people, and their evolution. For instance:

- Project Risk Reduction Patterns [Cockburn 97b]. To reduce project risk, we tend to apply some particular strategy (often a staging strategy: incremental, iterative, spiral, eddy, or fountain). Each gives us new information early, to enable some mid-course adjustment. You can invent your own strategies, creating whatever sequence of development you need, if you keep in mind the fact that each action should reduce risk of non-delivery. In the framework we link each people-related pattern to the person it relates to.
- Organizational Patterns [Coplien 95]. Organizational Patterns describe the structure and practices of human organizations. Coplien focuses on organizations that build (or use, or administer) computer software. Organizational solutions are less widely understood or practiced in project management, whereas many of the challenges and opportunities for software quality and productivity have traditionally been attacked with technology; there is a rich body of literature on such technology and design.
- Emerging Patterns in Human Competence and Business Development [Docherty 97]. Dealing with aspects like the evolution of technical competence of the personnel, is an aspect of a development process. Docherty does not work in the software patterns field, however he provides a reference for evaluating the character, depth and stability

of the social and technical changes taking place in a firm, and the nature of the new competencies which emerge in the production personnel, which seems patternisable.

- Peopleware [DeMarco & Lister 87]. DeMarco and Lister address many aspects related to teams as a society. They look at the office environment, having the right people in a team, etc. (As well as previously seen aspects like organization). They describe usual errors or "garden path that lead managers down, usually to their regrets". A form which is similar in its intends to patterns.

3.2.3 Artifact Dimension

The artifacts-related patterns usually describe how to write an artifact, or give a model solving a problem. The artifacts being inherently static, there is few or no interaction to study. These patterns are largely used for software architecture and design.

- Architectural Patterns. An architectural pattern expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- Design Patterns [Gamma 93]. A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.
- HCI Patterns Library: Common ground [Tidwell 98]. Tidwell defined a library containing interface patterns, which is intended to be used by people who design traditional user interfaces, Web sites, on-line documentation, video games, and other such things, and also people who implement such artifacts, or test them for usability, or manage teams who design and implement them. It aims to benefit, however, not only the individual HCI designers in their routine work, but also the whole industry in developing better tools and paradigms.

3.2.4 Activity Dimension

The activities can be described by patterns in two levels. The first level is "how to perform a task", the second how to coordinate or organize the tasks.

- Process Pattern [Ambler 98]. Process patterns are a collection of general techniques, actions, and/or tasks (activities) for developing object-oriented software. An important feature of a process pattern is that it describes what you should do but not the exact details of how you should do something. It includes Task Process Patterns, Stage Process Patterns and Phase Process Patterns, Life Cycle Process Patterns, Approach Process Patterns. Ambler also describes process anti-patterns or approaches and/or series of actions for developing software that are proven to be ineffective and often detrimental to your organization.

3.2.5 Patterns that can be used at Different Integration Levels

Some patterns affect many dimensions indefinitely, and cannot be categorized as related to only one of them.

- Business Process Reengineering (BPR) [Beedle 95]. BPR is the fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in critical contemporary measures of performance, such as cost, quality, service and speed. The definition of Business Processes should lead to Jobs and Structures, which in turn require Management and Measurement Systems, that reinforce a set of Values and Beliefs (Business System Diamond). The process side can only be attacked at the same time as the people side. In fact, Beedle studies the simultaneous implementation of OO architectures and BPR, which demands many changes in the business organization, its software development organization and its

enterprise architecture. These costs are justified, since we know that implementation of OO enterprise architectures brings ample business benefits in BPR environments such as: increased "enterprise conceptual integrity", reusability, generativity, and increased business effectiveness (cost, quality, service or speed).

- Process and Organizational Patterns [Coplien 95]. After his description of organizational patterns, Coplien describes the union of process and organizational patterns. These patterns have to be applied in a specific organizational and process context, however they are similar to organizational patterns in their goals, and means.

3.2.7 Conclusion to the Tools Provided

After investigating the pattern concept and analyzing different process and organizational patterns languages, we realized that patterns could complement our framework in an efficient way. They can be used at the four levels of integration we identified. They also provide more flexibility than the XML-based tool we presented in this chapter.

Conclusions

In this thesis, we presented our investigations on the way to improve and mediate the communication between usability expert and software development teams. With respect to experimentation, two specific processes constitute the focus of our interests: use case-driven and user-centered requirements engineering processes. Furthermore, we identified the following principles that we consider as critical issues in our defined framework for improving software-to-usability engineering communication.

Firstly, the requirements of an interactive system must be defined on two interrelated levels, but not independent as it is done. The first level is concerned with the specification of the context of use, and the second focuses on functional requirements. Different specification notations may be used for the two levels, but they should exploit an integrated representation of all the requirements artifacts. In our case, we adopted the text-based forms as used in RESPECT and the graphical representation of use cases as defined in the Unified Method Language.

Secondly, the list of artifacts describing the context of use ensures a good usability specification. Better still, this list can assist with generating functional requirements, at least to a limited extent. This result is fundamental because it can minimize the inconsistency of requirements artifacts and improve communication between software and usability engineers.

4.1 Methodological Summary of the Thesis

After studying the literature, we induced a model cataloguing the integration methods. This model organizes different aspects of the integration, which are pertinent but may not be the only existing aspects.

The pertinence of the rather a priory decomposition into the three 'A's dimensions was revealed during the use-case, which proves that these aspects have to be dealt with during integration. The secondary factors to the dimensions were kept apart, since they can always be emulated by a virtually infinite set of actors performing their task, or having their knowledge and techniques. Further studies can however be done on this subject.

The significance of the categorization of the modifications (or integration elements) into four levels of complexity was proved by the use-case and other literature. The elementary and structural complexity levels being trivial decompositions and the advanced level of complexity being proved by the necessity to preserve the implicit structure during integration, and the recurrent problems of the integration not respecting this policy.

The framework defined from this model, takes into account any level of complexity of modification (since the advanced level contains any non-categorized level of modification), but may not be able to describe it for two reasons:

- Patterns for defining this aspect and detail level may not exist yet
- There exists a modification that cannot be described as a pattern.
- Also the framework may not deal with a certain particular aspect independent from our three 'A's (independent in the way that it cannot be describe in terms of 'A's).

This framework is interesting for the integration of usability features into a SDLC, since it deals with many important aspects, but does not ensure a complete integration under all aspects. The addition of new aspects is however not likely, since the framework already

includes all the aspects addressed in the literature and the modeling languages. Also adding new patterns to describe aspects of the SDLC is rather easy, and can be done for any aspect.

4.2 Discussion on the evolution of the tool

Since our goal for this tool is to best support all aspects of integrating usability concerns in SDLC, a couple of obvious evolutions of the tool come in mind:

Firstly, the enhancement of the interface to best fit the different users' needs in terms of usability. Some researches evaluated that it takes ten years from the start of a project for the achievement of a tool that fully matches its users' needs.

Secondly, guidelines for a specific integration pattern could be provided to help the integrator (researcher, designer, or SPEG) in his methodology of integration, as well as a library of template processes, and standard patterns.

Thirdly, the development of a web-based version could help, by realizing a less costly introduction of consultants to help with integration. The consultant could, from the web, look at the existing process and provide certain patterns to solve the current problems of a firm.

Fourthly, the definition of two levels of use and the corresponding security mechanism would enable integrators to process in real-time and modify the process as well as test the aforementioned modifications in parallel.

4.3 Discussion on the evolution of the framework

This framework opens prospects to many interesting perspectives. First, even if the framework was developed aiming at an integration of usability concerns in a traditional SDLC, it can easily be adapted to the description of any SDLC, and to the integration of any feature in this SDLC. This is somewhat speculative and, of course, further study needs to be performed in order to prove it.

Secondly, the patterns are rather informal and high level, which does not facilitate reuse of the knowledge they contain. Some of the patterns could be coded in a more formal/structured way that could be reused by the framework, without implying a human presence.

Thirdly, we could associate an evaluation system with the framework that would allow the integrators to quantify the description level they attained on each dimension. This could eventually facilitate the determination of a firm's maturity level with respect to their software process (i.e. SEI-CMM level 5, ISO 9004).

References

[Alexander 1977] Alexander, Christopher, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. A Pattern Language, Oxford University Press, New York, 1977.

[Alexander 1979] Alexander, Christopher. The Timeless Way of Building. New York: Oxford University Press, 1979.

[Antunes, Seffah, Djouab 2001] H. Antunes, A. Seffah and R. Djouab. "Reconciling Usability-Centered and Use Case-Driven Requirements Engineering Processes", Australian Conference on User Interfaces, January 2001, Brisbanne, Australia.

[Ambler 1998] Ambler, Scott W; Process Patterns: Building Large-Scale Systems Using Object Technology. New York: SIGS Books/Cambridge University Press.

[Appleton 1997] Appleton, Brad; "Patterns for Conducting Process Improvement". Proceedings of the 1997 Fourth Conference of Patterns Languages of Program Design (PloP'97).

[Artim 1998] Artim John M., VanHarmelen Mark; "Incorporating Work, Process and Task Analysis into Commercial and Industrial Object-Oriented System Development", *SIGCHI Bulletin*, 30(4), 1998.

[Bayle 1998] Bayle, E., Bellamy, R., Casaday, G., Erickson, T., Fincher, S., Grinter, B., Gross, B., Lehder, D., Marmolin, H., Potts, C., Skousen, G., & Thomas, J. (1998). Putting it all together: Towards a Pattern Language for Interaction Design. A CHI'97 Workshop. SIGCHI Bulletin, 30, 1, 17-23.

[Online: http://www.acm.org/sigchi/bulletin/1998.1/erickson.html, http://www.pliant.org/personal/Tom_Erickson/Patterns.WrkShpRep.html], 1998.

[Beedle 1995] Beedle, Michael A. "Object Based Reengineering", Object Magazine 4(2), 1995.

[Beedle 1997] Beedle, Michael A; "Pattern Based Reengineering", Object Magazine, 6(11) Jan 1997, pp. 56-70.

[Bevan 1997] Bevan N., Azuma M. Quality in Use: incorporating Human Factors into the Software Engineering Lifecycle, Proc. of the 3rd IEEE International Software Engineering Standards Symposium and Forum, 1997.

[Booch et. al. 1997] Booch, Grady; Jacobson, Ivar; and Rumbaugh, James; The Unified Modeling Language (V 1.0), Reading, Massachusetts: Addison-Wesley, 1999. [Ref: http://www.rational.com/uml/].

[Booch et. Jacobson 1999] Booch, Grady; Jacobson, Ivar; and Rumbaugh, James.; *Unified Software Development Process*. Reading, Massachusetts: Addison-Wesley, 1999.

[Cockburn 1996] The interaction of social issues and software architecture; Alistair Cockburn; Commun. ACM 39, 10 (Oct. 1996), Pages 40 – 46.

[Cockburn 1997] Cockburn Alistair; "Structuring Use Cases with Goals." Journal of Object-Oriented Programming, Sept-Oct 1997 and Nov-Dec 1997.

[Cockburn 1997b] Cockburn, Alistair; "Project Risk Reduction Patterns" publication? [Online: http://members.aol.com/acockburn/riskcata/riskbook.htm]

[Cockburn 2000] Cockburn Alistair; "Growth of Human Factors in Application Development", submitted to the IBM Systems Journal. [Online: http://hometown.aol.com/acockburn/papers/adchange.htm]

[Constantine & Lockwood 1999] Constantine Larry L., and Lockwood Lucy A.D.; *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Reading, Massachusetts: Addison-Wesley, 1999.

[Constantine & Lockwood 2000] Constantine Larry L., and Lockwood Lucy A.D.; "Structure and Style in Use Cases for User Interface Design" [Online: http://www.foruse.com/Files/Papers/structurestyle2.pdf].

[Conway 1968] Conway, M.E., "How do committees invent?", Datamation , 14 . 4 (April 1968), pp. 28-31, as cited in The Mythical Man-Month , by Frederick Brooks, Anniversary edition, 1995, Addison-Wesley, p. 111.

[Coplien 1995] Coplien, James. "*A Generative Development-Process Pattern Language*" in the first *Pattern Languages of Program Design* book (Addison-Wesley, 1995).

[Darnell 2001] Darnell Elissa. "Usability Throughout the Product Development Cycle" in design by people for people Essays on Usability, ed. Branaghan Russell J, p79-86.

[DeMarco & Lister 1987] DeMarco, Tom; and Lister Timothy. Peopleware: Productive Projects and Teams, 2nd ed., ISBN: 0-932633-43-9, 1999 (1st edition in 87).

[DeMarco 1993] DeMarco, Tom. Talk at CaseWorld, Boston, Mass., January 1993.

[Docherty 1997] Docherty, Peter and Nyhan Barry. Understanding Industry in Transition. In Human Competence and Business Development: Engineering Patterns in European Companies. Eds. Peter Docherty and Barry Nyhan. Springer.

[ESD/MITRE 1986] Report from the MITRE Corporation, to the Electronic Systems Division of the United States Air Force, "Guidelines for Designing User Interface Software" also known as ESD/MITRE Project 5220 & 5720. [Online at: http://www.shu.ac.uk/schools/cms/teaching/crr/DOCS/s_and_m.txt and http://www.cms.dmu.ac.uk/General/hci/hcibib/guidelines], 1986.

[Forbrig & Dittmar 1999] Forbrig Peter; and Dittmar, Anke (1999). Relations between Use cases and Task Analysis, in 13th European Conference on Object-Oriented Programming - Lisbon Portugal 14-18 June 1999 Workshop on Integrating Human Factors into Use Cases and OO Methods.

[Forbrig 1999] Forbrig, Peter; "Task and Object-Oriented Development of Interactive Systems – How many models are necessary?" *Design Specification and Verification of Interactive Systems Workshop DSVIS'99*, Braga, Portugal, June 1999.

[Gamma 1993] Gamma, Erich; Helm, Richard; Johnson, Ralph; and Vlissides, John. Design Patterns: Abstraction and Reuse of Object-Oriented Design. ECOOP '93 Springer.

[Hartson et. Al. 1996] Hartson, H. Rex; Castillo, J.C.; Kelso, J.; Kamler, J., et.al., (1996) "Remote evaluation: the network as an extension of the usability laboratory," Human Factors in Computing Systems. Common Ground. CHI 96 Conference Proceedings, New York, NY, USA, ACM, 1996. p. 228-35 [Online: http://info.acm.org/sigchi/chi96/proceedings/papers/Hartson/hrh_txt.htm]

[ISO/IEC 12207:1995] ISO/IEC 12207: Information technology - Software life cycle processes. [Ref: http://www.iso.ch/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=21208], 1995.

[ISO 13407:1999] ISO 13407 (1999). Human-centred design processes for interactive systems. Geneva, Switzerland: International Organization for Standardization.

[Jarke 1999] Jarke, Matthias; "Scenarios for Modeling," *Communications of the ACM* 42(1), 1999.

[Krutchen 1999] Krutchen Philippe; "Use Case Storyboards in the Rational Unified Process," Workshop on Integrating Human Factors in Use Case and OO Methods. 12th European Conference on Object-Oriented Programming. Lisbon, Portugal, June 14-20, 1999.

[Lea 1993] Dough Lea; Review of "Christopher Alexander: An Introduction for Object-Oriented Designers" [Online: http://gee.cs.oswego.edu/dl/ca/ca/ca.html]

[Landay 1996] Landay, James A.; SILK: Sketching Interfaces Like Krazy. James A. Landay, Technical Video Program of CHI '96, April 14-18, 1996 [Online:http://web.cs.cmu.edu/afs/cs.cmu.edu/user/landay/pub/www/research/publications/CHI96/video.html], 96

[Mayhew 1992] Mayhew, Deborah J.; Principles and Guidelines in Software User Interface Design, Prenctice-Hall Publishing Company.

[Mayhew 1999] Mayhew, Deborah J.; The Usability Engineering Lifecycle:A Practitionner's Handbook for User Interface design, Morgan Kaufmann, San Francisco (1999)

[Nielsen 1993] Nielsen, Jakob. Usability Engineering. Academic Press, Boston, ISBN 0-12-518405-0 [Ref: http://www.useit.com/jakob/useengbook.html], 1993.

[Nielsen & Mack 1994] Nielsen, Jakob; and Mack, Robert L. Usability Inspection Methods. John Wiley & Sons, New York, NY, ISBN 0-471-01877-5 [Ref: http://www.useit.com/jakob/inspectbook.html], 1994.

[Nunes & Cunha 1999] Nunes, Nuno Jardim; and Falcão e Cunha, João. "Detailing Use-Cases with Activity Diagrams and Object Views", in 13th European Conference on Object-

Oriented Programming – Lisbon Portugal 14-18 June 1999 Workshop on Integrating Human Factors into Use Cases and OO Methods.

[Nunes & Cunha 2000] Nunes, Nuno Jardim; and Falcão e Cunha, João. "Towards a UML profile for interaction design: the Wisdom approach" UML'2000. [Online: http://math.uma.pt/tupis00/submissions/nunesCunha/nunescunha.html]

[Riehle & Zullighoven 1996] Riehle, Dirk; and Züllighoven, Heinz. "Understanding and Using Patterns in Software Development", Theory and Practice of Object Systems, Vol 2, p3-13, 1996 [Online: citeseer.nj.nec.com/riehle96understanding.html]

[Rosson 1999] Rosson Mary Beth. Integrating Development of Task and Object Models. Communication of the ACM January 1999/Vol 42, No.1, (pp 49-56), 1999.

[Seffah & Hayne 1999] Seffah A., and Hayne C.; "Integrating Human Factors in Use Case and OO Methods," *12th European Conference on Object-Oriented Programming Workshop Reader*, Lecture Notes in Computer Science 1743, 1999.

[Sharp, Manns & al. 1996] Sharp, Helen; Manns, Mary Lynn; McLaughlin, Phil; Prieto Maximo; and Dodani Mahesh. "Pedagogical Patterns -- Successes in Teaching Object Technology" ACM SIGPLAN Notices, Vol. 31(12), December 1996, pp. 18-21.; 1996

[Tidwell 1998] Tidwell, Jennifer. "Interaction design patterns" in Patterns Languages of Programs (PLoP '98), Allerton Conference Center, Urbana Champaign, IL, [Online: http://www.mit.edu/~jtidwell/interaction_patterns.html], 1998

[Vora & Helander 1995] Vora, P. R.; and Helander, M. G. A teaching method as an alternative to the concurrent think-aloud method for usability testing. In Y. Anzai, K. Ogawa, & H. Mori (Eds.), Symbiosis of human and artifact: Future computing and design for human-computer interaction. 375-380. Proceedings of HCI International 95, Volume 2. Advances in Human Factors/Ergonomics, Volume 20B. Amsterdam, The Netherlands: Elsevier. [ref: http://www.hcirn.com/ref/refv/vorah95.html],1995.

[Yourdon 1989] Yourdon, Edward N. Modern Structured Analysis. Englewood Cliffs, N.J.: Prentice-Hall, 1989.

Glossary

Dimensions: independent and atomic aspect of a development process, all this aspects have to be considered in order to change consistently the process. Any modification interacts with many of these dimensions. We distinguished three of them: the three 'A's:

The **Artefact Dimension**: Modelling structure formats and techniques. E.g. UML defines a 'standard' for this artefact dimension, that is very often completed by a structure between the documents in the way that does [Cockburn 97] in structuring by goals, or [Constantine & Lockwood 99] with the usage-centred design framework.

The **Actor Dimension**: Human point of view of the process. This dimension has two faces: the profile of the actors (knowledge, communication skills), and their organisation, which is related to the organisational Patterns. This dimension is related to management, and we find good information about it in [DeMarco & Lister 87], [Cockburn 1996], []

The **Activity Dimension**: 'Process' point of view of the software life-cycle. This organised as a sequence of activities. Nowadays development processes are mostly iterative, and we try to bring more and more enhancements/refinements to it. Some people want to give it a more loose structure: we try in our case-study to have a process composed of two sub-processes linked via essential milestones, [Nunes & Cunha 2000] advocates a looser coupling between the two processes.

Factors: aspect of a development process that is somehow related to one or many dimensions and come to complete it. These are very various factors, but sometimes essential to the feasibility of the process, i.e. without them a task may take too much time, or be too complex.

Resources: Any element or piece of information that is independent of the process, but will help during the process, e.g. a dictionary, previous-project models, etc.

Tools: Any tool than can help an actor to perform an activity, or eventually replace him. Text Editors, Spreadsheets, schedulers, CSCW are good examples of this tools. Ultimately they can replace an actor doing for example consistency checking between the documents.

Complexity of a Modification: (cf. Fig 13) extent of the modification, it is used to sort the modifications by degree of importance. We distinguished four of them.

Elementary modifications: Modifications that affect only one element of the SDLC. For example we can change a very specific artefact, and so indirectly the way to perform the corresponding activity. No change in the main communication line is needed.

Complementary modifications: adding new elements in the SDLC without changing the existing communication line.

Structural modifications: modification that affect the communication line or global architecture by adding an activity or a set of activities in the SDLC, and changing the sequencing of the activities.

Advanced modifications: modifications cannot be described by only a change in the communication line. Usually, they come on the top of a structural change.

Level of Description: level of details needed in the description of the process to be able to perform the change in it.

In the **elementary level of description**, we describe each element, it is the basic level of description.

In the **structural level of description**, we describe the communication line or structure of the process.

In the **implicit level of description**, we will need to describe implicit organization of many forms (some times organizations of organizations), an a priori choose of patterns as form of description seems interesting. It will be required to solve the detected communication/organizational problems between teams, as well as for any detail specific to the process (complex modifications), it needs to contain what we called the manager's savoir-faire.

Patterns: A pattern is the abstraction from a concrete form, which keeps recurring, in specific non-arbitrary contexts. [Riehle & Zullighoven 96]

Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution. As an element in the world, each pattern is a relationship between a certain context, a certain system of forces, which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves. The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is a process and a thing; both a description of a thing, which is alive, and a description of the process, which will generate that thing. [Alexander 79]

Pattern formats:

Alexanderian Form: original pattern form from Christopher Alexander. Includes Title, Asterisks, Picture, Introductory paragraph, Headline, Body of the problem, Solution, Diagram, Connections to lower level patterns.

GoF: Gang of Four Form, it contains Name, Intent, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Known uses.

Acronyms

Alexandrian Form: text format for a pattern, introduced by [Alexander 77].

CASE tool: Computer Aided Software Engineering; tool to aid to develop software.

CSCW: Computer Supported Co-operative Work

GoF format: The two most popular formats are called GoF Form (named after the format used by the "Gang of Four" in their "Design Patterns" book)

GUI: Graphical User Interface, generic term for software interface.

HCI: Human Computer Interface, part of software engineering (more precisely usability engineering) focusing on providing quality interfaces.

IAT: Improvement Action Team -- another commonly used acronym is PAT (Process Action Team)

ISO 9004:2000 International Standard Organisation std. 9004. Description of the practices to be implemented to make your quality management system increasingly effective in achieving your own business goals.

OOAD: Object Oriented Analysis and Design, Analysis and Design Methodology based on the 'objects'.

PEG: Process Engineering Group, team working on the definition of the software process. The SEI Software CMM uses the acronym SEPG (Software Engineering Process Group)

PIT: Process Improvement Team, team taking care of the SPI. Other acronyms used are PWG (Process Working Group), and sometimes even SEPG (Software Engineering Process Group), though this latter acronym is more often used in place of PEG (see below).

RAD: Rapid Application Development, tools and techniques focusing mainly on the rapid development of application.

SDLC: Software Development Life-Cycle, description of the process of developing software.

SEI-CMM: Software-Engineering-Institute's Capability Maturity Model (-Carnegie Melon University-). It helps to evaluate and organise your process, by defining five levels of capability of process (the first level being rough detail of definition and the last level a continuous improvement).

SPI: Software Process Improvement, new branch of software engineering dealing with the improvement of the Software Development Lifecycle.

UCD: User-Centred Design, design methodology guaranteeing usability to software.

UCDD: Use-Case Driven Design, traditional design methodology based on Use-Cases incorporated to the Rational Unified Process (more frequently written UCD in software engineering).

UML: Unified Modelling Language, de facto standard modelling language cf. [Booch & al. 97]

UPADE: Usability Pattern Assisted Design Environment, environment presented at Concordia University by A. Seffah

Annexes

Annex 1: XML DDT

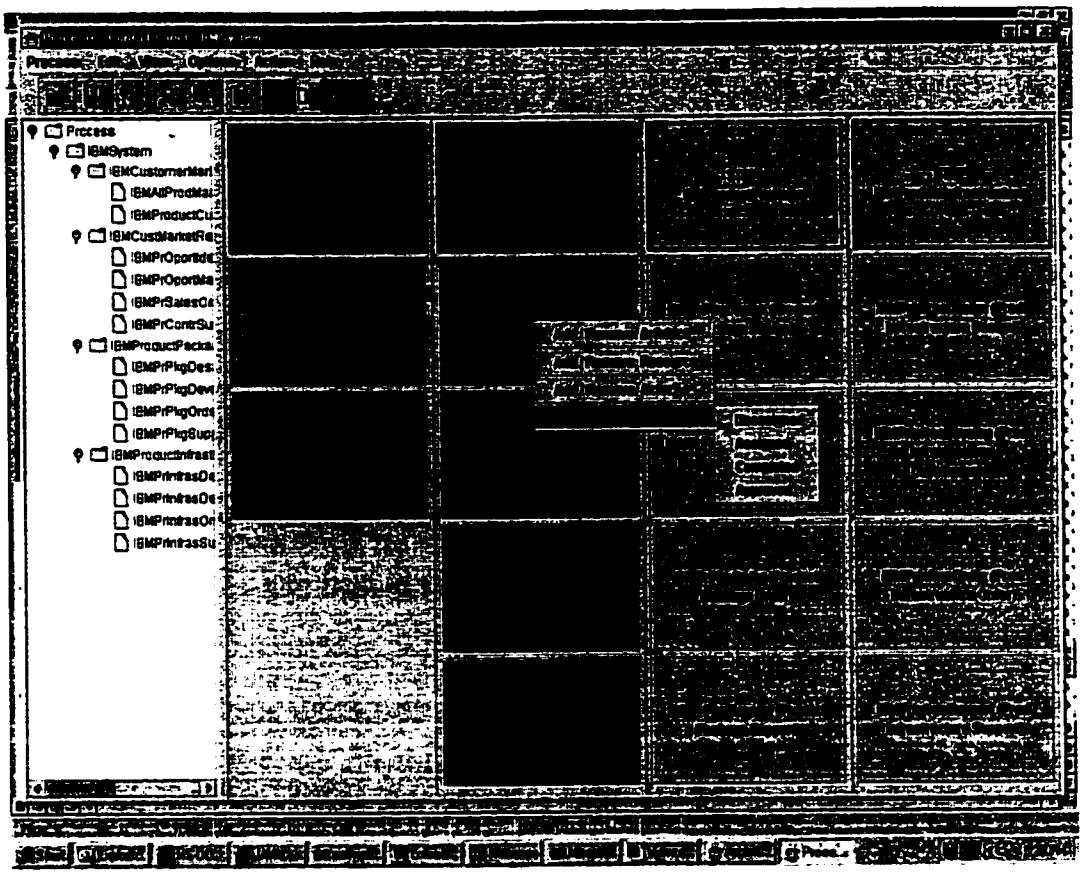
```
<?xml version="1.0" encoding="UTF-8"?>
<epss.data.Process
name="RESPECT"
url ="./descriptions/RESPECT.html">
        <stepArray>
                <epss.data.Step
                        name="USER CONTEXT AND EARLY DESIGN"
                        url="./descriptions/1.html">
                        <activityArray>
                                <epss.data.Activity
                                        name="Summarize Project"
                                        url="./descriptions/1.1.html"
                                        status="0"
                                        isLocked="false"
                                        lockingActor="">
                                <actorArray>
                                        </actorArray>
```

```
                    <artefactArray>
    <string value="./forms/1.1:Project Summary from Users'ViewPoint.ps"/>
                    </artefactArray>
                    <resourceArray>
    <string value="//resources/User Interface GuideLines.ps"/>
    <string value="//resources/Human Factors Standards.ps"/>
                    </resourceArray>
                    <predecessorActivityArray>
                    </predecessorActivityArray>
                </epss.data.Activity>
                <epss.data.Activity
                    name="Identify Users and Stakeholders"
                    url="./descriptions/1.2.html"
                    status="0"
                    isLocked="false"
                    lockingActor="">
                <actorArray> ... </actorArray>
                <artefactArray>
    <string value="./forms/1.2:List of Users and current or expected role in the
system.ps.ps"/>
                    </artefactArray>
                <resourceArray>
    <string value="//resources/User Interface GuideLines.ps"/>
```

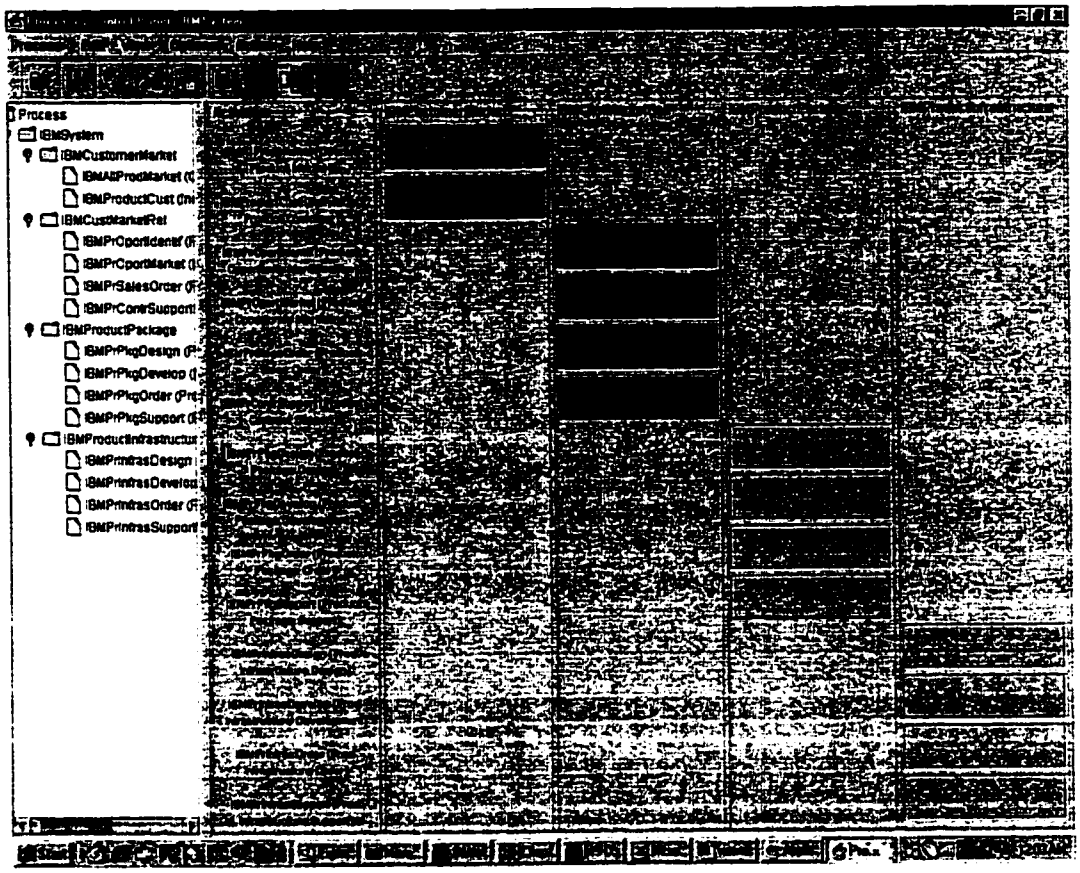
```
        <string value="//resources/Human Factors Standards.ps"/>
                                        </resourceArray>
                                    <predecessorActivityArray>
                                        <epss.data.ActivityReference      stepName="USER
CONTEXT AND EARLY DESIGN" activityName="Summarize Project"/>
                                    </predecessorActivityArray>
                                </epss.data.Activity>
                            </activityArray>
                        </epss.data.Step>

        </stepArray>
</epss.data.Process>
```

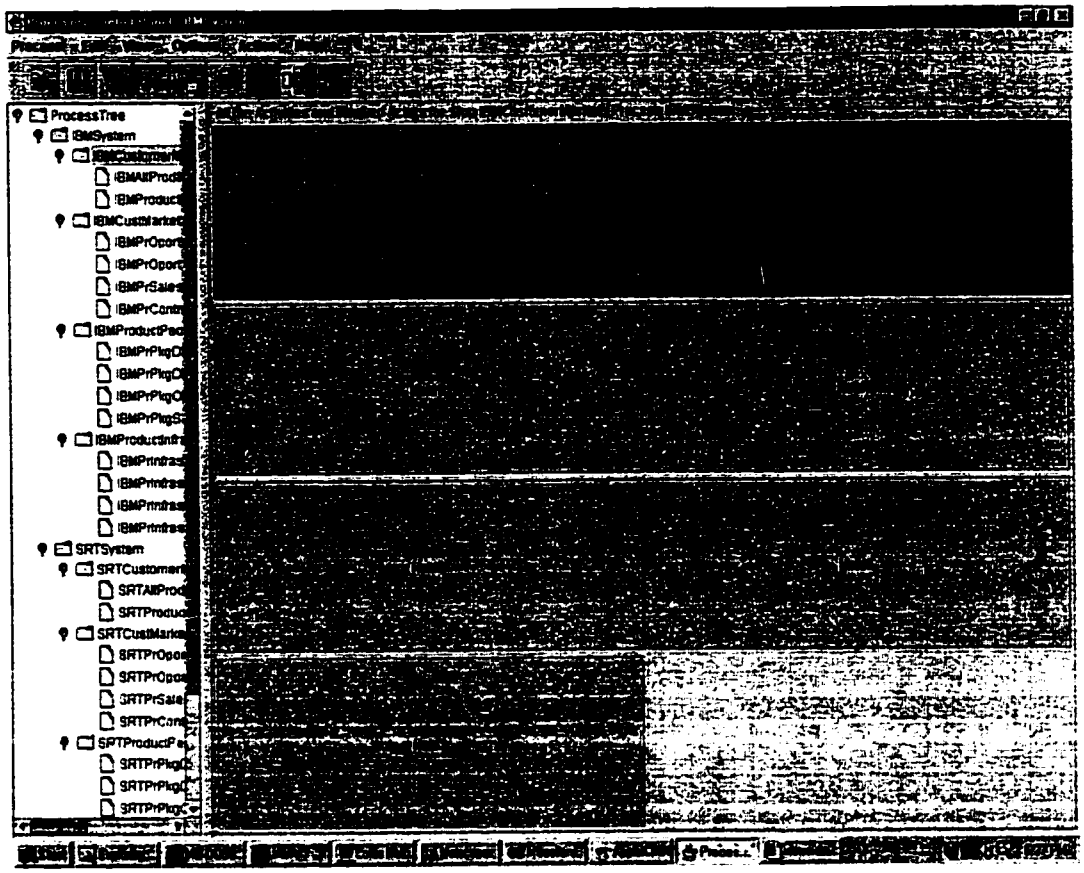
Annex 2: Steps X Activity Condensed View



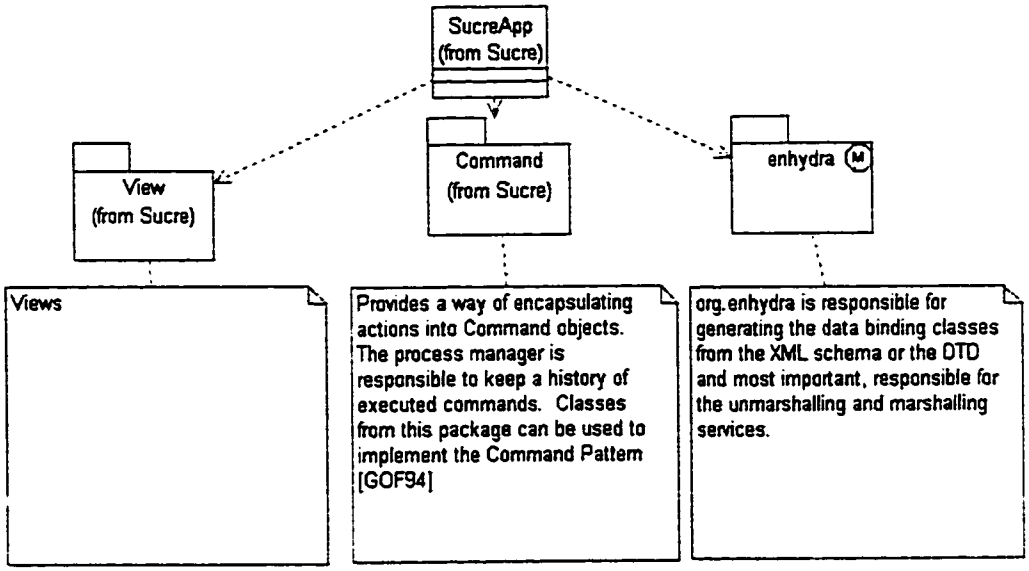
Annex 3: Steps X Activities Expanded View



Annex 4: Step Details View



Annex 5: Architecture of SUCRE



Annex 6: Data Model of SUCRE