# 3 HCI, USABILITY AND SOFTWARE ENGINEERING INTEGRATION: PRESENT AND FUTURE

Ahmed Seffah*, Michel C. Desmarais**, and Eduard Metzker***

* Human-Centered Software Engineering Group Computer Science Department,
Concordia University,
** Computer Engineering Department,
École Polytechnique de Montreal, Quebec, Canada
*** Software Technology Lab, Daimler Chrysler Research and Technology Centre,
Ulm, Germany

**Abstract**

In the last five years, several studies and workshops have highlighted the gap between software design approaches in HCI (Human Computer Interaction) and software engineering. Although the fields are complementary, these studies emphasize that they are not well integrated with each other. Several frameworks have been proposed for integrating HCI and usability techniques into the software development lifecycle. This chapter reviews some of the most relevant frameworks. It assesses their strengths and weaknesses as well as how far the objective of integrating HCI methods and principles within different software engineering methods has been reached. Finally, it draws conclusions about research directions towards the development of a generic framework that can: (1) facilitate the integration of usability engineering methods in software

development practices and, (2) foster the cross-pollination of the HCI and software engineering disciplines.

## 3.1 INTRODUCTION

It is no coincidence that the titles of some chapters in this book evoke the terms "solitudes" and "competition" to characterize the relation between the fields of HCI (Human-Computer Interaction) and SE (Software Engineering) (cf. Jerome and Kazman, chapter 4; Sutcliffe, chapter 5). This uneasy cohabitation dates back to the early days of HCI when human-centered design has been presented as the opposite, and sometimes as a replacement, to the system-driven philosophy generally used in software engineering (Norman and Draper, 1986). Although numerous HCI researchers and practitioners view User Centered Design (UCD) as a process and as a set of specific methodologies to design and develop interactive software applications, HCI is by no means considered a central topic in SE. For example, the SWEBOK, an IEEE initiative for the definition of SE knowledge and practices, defines HCI as a "related discipline", termed "software ergonomics" (Abran et al., 2004). Usability is considered one of many non functional requirements and quality attributes. No reference is made to specific UCD methods such as those found in the ISO 13407 standard, "Human-centred design processes for interactive systems" (ISO/IEC, 1999).

The reality is that UCD and software engineering techniques do have overlapping objectives of defining which methods to use in the software development process, what kind of artefacts (documents and deliverables) to use, and what quality attributes to prioritize. However, we argue that they have different perspectives on the software development process itself, as depicted in figure 3.1. The SE community focuses on the "system 1" perspective in Figure 3.1: software development is driven by specifications provided for defining the application, including the interface. The user interface has to meet the functional and usability requirements, but the requirements are tied to the system, that corresponds to the application itself. The focus is on the software application and the interface is one of many components that has to meet certain requirements.

In contrast, the proponents of UCD, and more specifically the "quality in use" approach (Bevan, 1999), focus on the "system 2" perspective: The priority is to ensure that each class of users can perform a given set of tasks with the application. The ultimate requirements are tied to what the user can perform with the application. Consequently, the software development process is driven by the need to define and validate these requirements and closely depends on the tasks defined and the users' capabilities and characteristics.

The two perspectives do not differ solely from their philosophical stance. The can have significant impacts on the how the software development process will be defined and planned, which activities will be conducted, which tools will be used, and the expertise of the team members and its leader. The impacts are particularly important with regards to the requirements management and quality control activities.

While both perspectives are valid, the SE approach is always necessary, since there necessarily is a "system 1" perspective. It either stands alone in the absence of a significant user interface component, or it is embedded in the "system 2" perspective

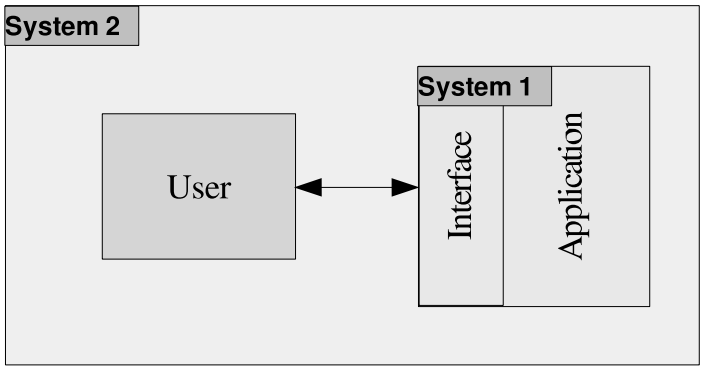


**Figure 3.1** The two system perspectives.

otherwise. But when the application's degree of interactivity and interface complexity is high, the "system 2" perspective, we argue, should prevail. The whole development process must then put the emphasis on defining, validating, and measuring what the use can do with the application. Thus the need to integrate UCD approaches to the SE development process.

Bringing together the SE and UCD approaches was the major goal of several workshops organized during the last decade, as well as the goal of a number of research efforts. These workshops highlighted the UCD and SE (Software Engineering gaps and the importance of addressing them (van Harmelen et al., 1997; Artim et al., 1998; Seffah and Hayne, 1999; Nunes and e Cunha, 2000; Gulliksen et al., 1998; Gulliksen et al., 2001; Harning and Vanderdonckt, 2003; Kazman et al., 2003; John et al., 2004b). The starting point of all these workshops was the two workshops organized by van Harmelen et al at CHI'97 and CHI'98 conferences on Object-Oriented Models in User Interface Design and on incorporating Task Analysis Into Commercial And Industrial Object-Oriented Systems Development (van Harmelen and Wilson, 1997; Artim et al., 1998).

As will be detailed in this chapter, the conclusions of these investigations brought to light some of the major integration issues, including:

- Extending software engineering artefacts for user interface specification, such as annotating use cases with task descriptions (Constantine and Lockwood, 1999; Rosson, 1999; Cockburn, 1997; Dayton et al., 1996),

- Enhancing object-oriented software engineering notations and models (Nunes and e Cunha, 2000; Artim et al., 1998; Kruchten, 1999; da Silva and Paton, 2001).

- Possible extensions of UCD methods for requirements gathering through field observations and interviews, deriving a conceptual design model from scenario, task models and use cases (Rosson, 1999; Paternò, 2001; Benyon and Macaulay,

2002) and using personae (Cooper and Reimann, 2000) as a way to understand and model end-users.

- New methodologies for interactive systems design such as those introduced by Nielsen (1995, 1993), Mayhew (1999), and Roberts (1998), as well as approaches complementing existing methodologies (Constantine and Lockwood, 1999; Kruchten, 1999).

We will review the these different issues and the frameworks proposed for integrating UCD and SE in the following sections.

## 3.2 DEVELOPMENT PROCESSES

HCI practitioners and researchers have proposed a number of development processes that take into account the particular problems encountered in the engineering of highly interactive systems. Examples of the large number of methodologies are the Star Lifecycle (Hix and Hartson, 1993), LUCID ("Logical User-Centered Interface Design" method of Smith and Dunckly, 1998), the Usability Engineering Lifecycle (Mayhew, 1999), Usage-Centered Design (Constantine, 1999), SANE Toolkit for cognitive modeling and user-centered design (Bosser et al., 1992), SEP (for user-centered requirements using scenarios) (McGraw, 1997) and IBM-OVID (Object, View, Interaction and Design) (Roberts, 1998; see also Roberts, chapter 11 in this book).

Reviewing of all these approaches would go beyond the scope of this chapter. Some of these methods, and in particular those aiming to bridge object-oriented models, scenarios, and task models are detailed in Van Harmelen (1997). In the following sections, we focus some of the main approaches, namely scenario-based approach (Carroll, 2000), contextual design (Beyer and Holtzblatt, 1998), the star lifecycle (Hix and Hartson, 1993), the usability engineering lifecycle (Mayhew, 1999), and usage-centered design (Constantine and Lockwood, 1999). We also refer the reader to Roberts' recent coverage of the OVID and IBM's approaches (see Roberts, chapter 11 in this book) and the Cognetic's corporation's LUCID framework[1].

### *3.2.1 Scenario-Based Design*

Carroll and Rosson (Carroll, 2000; Rosson and Carroll, 2002) developed a usability engineering approach centered around different stages of scenario definition (see Figure 3.2). Scenarios are not a novel concept. They are known by both the human factors community, for conducting task analysis, and by the software engineering community, as instances of use-cases. However, the scenario-based usability engineering process places a much greater emphasis, and provides greater details on their role during the early phases of the software lifecycle.

Scenarios are used for clarifying usability requirements and for driving the iterative design process. Scenarios describe an existing or envisioned system from the perspective of one or more real or realistic users. They are used to capture the goals,

[1]See `http://www.cognetics.com/lucid/index.html`.

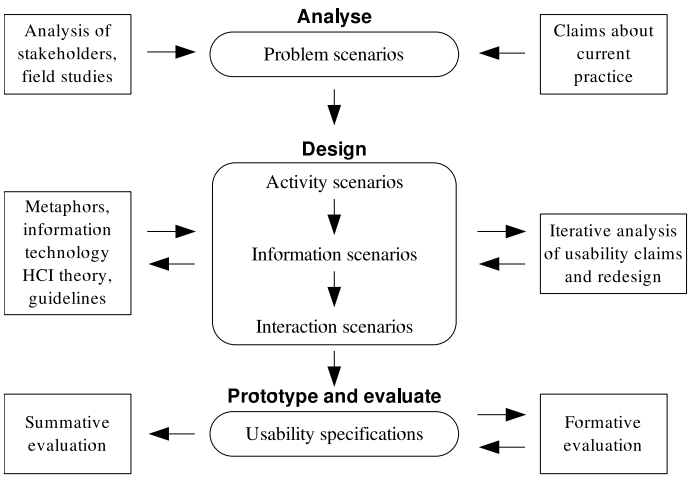


**Figure 3.2** Carroll and Rosson's scenario-based framework (adapted from Carroll and Rosson, 2002).

intentions, and reactions of the user. Carroll attributes several merits to scenarios, in particular improving the communication between users, designers and stakeholders. As previously mentioned, communication with different groups involved in the design process is one of the major integration obstacles for UCD methods. Because scenarios are formulated in plain natural language, they have the advantage of being understood both by users and designers. This enables all project participants to share in the design process and discuss potential solutions.

Claims are the second core concept of scenario-based design (see section 3.4.4). Claims are developed in parallel to scenarios. The core elements that are addressed in a scenario are listed with their potential advantages and drawbacks. This clarifies which effects each element has on the usability of the system. If the drawbacks of an element outweigh the advantages, it is usually discarded.

### *3.2.2 Contextual Design*

Contextual design, developed by Beyer and Holtzblatt (Beyer, 1998), stresses the behavioral aspects of the system design process. In their view, almost all of the system design process is driven by these aspects. Software requirements engineering, as a subsequent phase, is viewed as a response to the systems design process. In other words, the complete system design process should be an input to the software requirements engineering process. The key activities of contextual design are: Observe and interview the customer, construct and consolidate work models, redesign work models, and design the system.

Beyer and Holtzblatt (Beyer, 1998) emphasize that "the ability to see, manipulate, and design a process for delivering systems is a fundamental skill when it comes to

establishing their techniques in software development processes". However, they have provided only very generic recommendations for adapting and tailoring the overall approach to different project configurations. They recommend for example 'to recognize which parts are critical and which are less necessary in each case', 'what works for a two-person team won't work for a fifteen person team', 'what works to design a strategy for a new market venture won't work for the next iteration of a 10-year old system', 'tailor things you pick up to your problem, team and organization' and finally 'what you do with it is up to you' (Beyer, 1998).

### *3.2.3 Star Lifecycle*

The star lifecycle proposed by Hix and Hartson (1993), focuses on usability evaluation as the hub process activity. Placed around this central task are the following activities: (1) system, task, functionality, user analysis, requirements and usability specifications, (2) design, design representation and rapid prototyping, (3) software production and deployment. The results of each activity are subjected to an evaluation before moving on to the next process activity. It is possible to start with almost any development activity. The bi-directional links between the central usability evaluation task and all other process activities cause the graphical representation of the model to assume a star shape.

One of the drawbacks of this approach is outlined by Hix and Hartson (1993). Project managers tend to have problems with the highly iterative nature of the model. They find it difficult to decide when a specific iteration is completed, thus complicating the management of resources and limiting their ability to control the overall progress of the development process. An obvious solution to this problem is to establish control mechanisms such as quantitative usability goals that serve as stopping rules.

Hix and Hartson give some basic advice on tailoring the overall approach to a specific development context. They suggest a top down approach if the development team has some experience with and prior knowledge of the target system structure. Otherwise they favour a more experimental bottom-up approach. They suggest that the overall approach should be configured to accommodate the size of the project, the number of people involved, and the management style. They explicitly emphasize the necessity to view usability engineering as a process, but they agree that the design phase is one of the least understood development activities. They provide special methods and notations to support the process. For example, the user action notation (UAN) specifies user interaction in a way that is easily readable and yet unambiguous for implementing the actual interface. Usability specification tables are employed for defining and tracing quantitative usability goals.

### *3.2.4 Usability Engineering Lifecycle*

Proposed by Deborah Mayhew, the usability engineering lifecycle is an attempt to redesign the complete software development process around usability engineering knowledge, methods, and activities (Mayhew, 1999). This process starts with a structured requirements analysis concerning usability issues. The data gathered from the requirements analysis is used to define explicit, measurable usability goals for the pro-

posed system. The usability engineering lifecycle accomplishes the defined usability goals via an iteration of usability engineering methods such as conceptual model design, user interface mock-ups, prototyping and usability testing. The iterative process terminates if the usability goals have been met or the resources allocated for the task have been consumed.

As outlined by Mayhew (1999), the usability engineering lifecycle has been successfully applied in various projects. However, some general drawbacks were discovered by Mayhew during these case studies. One key concern is that redesigning the overall development process around usability issues often poses a problem to the organizational culture of software engineering organizations. The well-established development processes of an organization cannot easily be turned into human-centered processes during a single project. Furthermore, development teams often have insufficient knowledge to perform the UCD activities, which hampers the establishment of UCD activities in the engineering processes. Precisely how the UCD activities proposed in the usability engineering lifecycle should be integrated smoothly into engineering processes practiced by software development organizations was declared by Mayhew to be an open research issue.

Mayhew names several success factors for practicing UCD. First, all project team members should carry out UCD process steps. Mayhew stresses the importance of ultimately formalizing UCD within a development organization and methodology. Project team participation is necessary and having a design guru on board is not enough.

### *3.2.5 Usage-Centered Design*

Usage-centered design, developed by Constantine and Lockwood (1999), is based on a process model called the activity model for usage-centered design (see figure3.3). The activity model describes a concurrent UCD process that starts with the activities of collaborative requirements modeling, task modeling, and domain modeling in order to elicit basic requirements of the planned software system. The requirements analysis phase is followed by the design activities of interface content modeling and implementation modeling. These activities are continuously repeated until the system passes the usability inspections carried out after each iteration. The design and test activities are paralleled by help system and documentation development and standards definition for the proposed system. This general framework of activities is supplemented by special methods like essential use case models or user role maps.

Constantine and Lockwood provide many case studies where usage centered design was successfully applied, yet they encountered many of the same organizational obstacles as Mayhew (Mayhew, 1999) when integrating their UCD approach into the software engineering processes in practice. They propose that ‘new practices, processes, and tools have to be introduced into the organization and then spread beyond the point of introduction’. A straightforward solution to these problems is the establishment of training courses for all participants of UCD activities offered by external consultants. However, this solution is regarded as time consuming and cost intensive in the long run. Also, it tends to have only a temporary effect and thus does not promote organizational learning in UCD design methods. Constantine and Lockwood conclude that it is necessary to build up an internal body of knowledge concerning

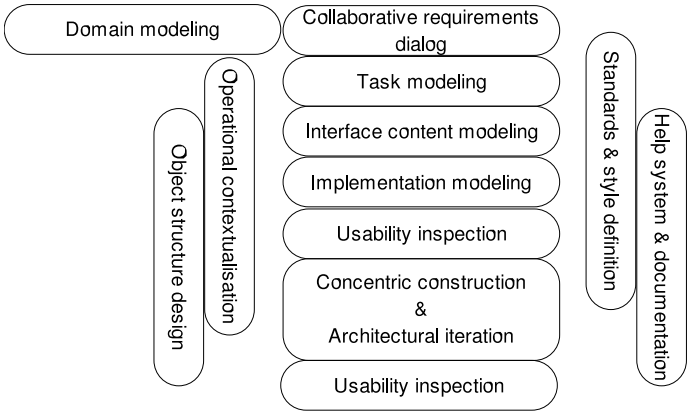


**Figure 3.3** Constantine and Lockwood's usage-centered design (adapted from Constantine and Lockwood, 1999).

UCD methods, best practices and tools, in a way that is tailored to the needs of the development organization.

### *3.2.6 ISO Standards 13407*

This review of process oriented integration of UCD and SE would be incomplete without mentioning the ISO 13407 standard (ISO/IEC, 1999). This standard defines the characteristics of what a user-centered development project must hold. It does not define a user-centered process by itself, only its characteristics. Neither is it meant to be a replacement, but instead it is intended to suggest a number of complementary means of addressing usability engineering issues.

A total of 36 activities are identified for the "human-centered design" framework (Maguire, 2001). They are classified into five categories that represent different phases of the development lifecycle: (1) planning, (2) context of use analysis, (3) requirements analysis (4) design, and (5) evaluation. Example of such activities for each respective phases are *usability planning and scoping*, *task analysis*, *scenarios of use*, *paper prototyping*, and *participatory evaluation*. ISO 13407 can also be used in conjunction with ISO TR 18529 for assessing the development process according to the "usability maturity model" (Earthy et al, 2001).

### *3.2.7 The Status UCD and SE Processes in Reality*

The UCD processes we briefly surveyed provide frameworks for planning the software development processes and activities. How widespread they are used, and how deeply they affect the software development and management processes remains an open question.

Jerome and Kazman's chapter in this book (4) suggest that their penetration in the real world of software engineering is relatively low. Roberts (chapter 11) and Vreden-

burg (2003) point out the importance of UCD and the OVID methodology at IBM, but there is no quantitative figures to assess the impact in reality. There are compelling accounts that some UCD processes are adopted and that they drive the whole development process. Some examples can be found in Rosson and Carroll's (2002) or in Landauer's book (1995). However, such examples do not constitute evidence of a trend itself. Our own experience is that the application of UCD processes remain an exception in general.

Our own assessment is that what is likely a mainstream practice is the adoption of some HCI practices, such as usability testing. Such fundamental UCD activities are probably widespread in many projects and corporations, but the whole software development process does not necessarily qualify as a UCD process.

While this section covered processes and activities, we now turn to another approach of integrating UCD and SE that relies on artefacts and documents.

## 3.3 ARTEFACTS

Besides activities, artefacts—the intermediate and final deliverables of software development project—are the elements that characterize an SE methodology. A number of researchers have attempted to identify artefacts that can help bridge UCD and SE approaches and techniques. They have looked for similarities and complementarity that could help merging typical UCD and SE approaches. We will focus on artefacts around scenarios (Carroll, 2000, Benyon, 2002), use cases (Cockburn, 1997; Seffah, 1999; Constantine, 1999), object-oriented notations such as UML (Paterno, 2001; Krutchen, 1999; da Silva, 2001; Markopoulos, 2000) and task models (Artim, 1998; Dayton, 1996; Rosson, 1999).

These investigations demonstrate that in use case-driven software development, human-centered design processes in general and task analysis approaches in particular are highly compatible. This may be considered as a starting point for cross-pollinating functional and user requirements engineering techniques and tools. For example, user-centered requirement artefacts such as task and user models can substantially improve certain basic weaknesses of the functional requirements approach. One weakness addressed by the user-centered approach is in identifying the context of use.

### *3.3.1 Scenarios as a Vehicle for Bridging Object Oriented Analysis and Design*

As an artefact for capturing user requirements, scenarios have been promoted both in HCI (Carroll, 2000) and software engineering (Jarke, 1999). However the concept of scenarios has not been consistently defined. Jarke (1999) proposed to clarify the purpose and manner of using scenarios in the modeling process, since scenarios can be used in very different manners. Jarke points out that scenarios are used in software engineering as intermediate design artefacts while Carroll argued that scenarios could be used as a driving force in the entire design process.

Rosson (1999) suggests enhancing the object-oriented analysis and designing approach with a scenario-based approach. Once scenarios are completed, she proposes first extracting elements that are potential computational objects, and then organizing

them as a network of collaborating objects. The next step is to focus on a specific object and try to assign functionality to it. This object-by-object analysis is supported by the Point-Of-View Browser that maintains user-relative descriptions of each object. The communication approach is middle-out, since it iteratively elaborates 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. This is the opposite of prototyping tools such as Visual Basic, which are outside in, because the focus is on screen design.

Such an approach guarantees a good object model as well as satisfying the need to take into account the user's point of view. It also addresses our main concern: The incorporation of the user's needs in the software development process. However, in this technique the user interface design relies 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 evocativeness). She defines an intermediate philosophy. The aim is not the user and their needs, or a good structure of the software; the aim is to have a good midpoint construct that helps establish a good interface as well as a good program structure. This solution did not seem to develop in the industry market, perhaps because it is too different from the methods currently in use.

Similar to Rosson's work, Jarke (1999) also proposed to clarify the purpose and manner in which to use scenarios in the modeling process. He defines scenarios as constructs that describe a possible set of events that might reasonably take place; they offer "middle-ground abstraction between models and reality". Scenarios are typically used in four approaches:

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

One of the major weaknesses of scenarios as an integration artefact is the fact that informal representations of scenarios, generally statements in natural language, are often insufficient for overcoming the difficulty of communication between users, developers, usability expert and stakeholders with differing backgrounds. Scenarios in natural languages suffer from ambiguity and imprecision. Formal representations of scenarios provide a solution to the ambiguity problem and facilitate formal proof and analysis of properties of requirements. However these formal specifications are often difficult to understand and develop for newcomers to this area. A trade-off is needed between the precision of formal representations and the ease of communication of scenarios in the context of accomplishing a task. Designers and users need to able to develop and reason about scenario descriptions throughout the development lifecycle in a variety of media, purposes, and views, either to discuss existing options or to stimulate imagination.

### *3.3.2 Bridging Task Analysis and Object-Oriented Models*

In model-based task analysis as practiced in HCI, the objective is normally to achieve a generic and thus abstract model of the user tasks, typically in a hierarchical form of goals, sub-goals and methods for achieving the hierarchy of goals. In object-oriented development, use cases are often employed in gathering functional requirements. Can task analysis models be improved by use case techniques? Can use cases be improved by the incorporation or consideration of formal task models? Are there ways of integrating the two approaches? Such questions have been widely discussed (Dayton, 1996, Artim, 1998; Seffah and Hayne, 1999; Forbrig, 1999; Engelberg, 2001).

Cockburn (1997), for one, recognizes that use-cases are not well defined and many different uses coexist, with differences in purpose, content, plurality and structure. He proposes to structure them with respect to goals or tasks. Although this approach may appear unusual as a structure for requirements, it follows a natural hierarchical organization typical of task analysis techniques (Dayton, 1996). The goals are structured as a tree containing "Summary goals" as high-level goals, and "User goals" as atomic goals (e.g. performing summary goal A involves performing user goal A1 then A2).

### *3.3.3 Extending UML Notation for User Interface Modeling*

Several research investigations have been conducted with a view to improving the unified modeling language (UML) for user interfaces and interactive systems engineering. Nunes and Cunha (2000) proposed the Whitewater Interactive System Development with Objects Models (WISDOM), as a lightweight software engineering methodology that uses UML to support Human-Computer interaction techniques. WISDOM is evolutionary in the sense that the project evolves incrementally through an iterative process. A novel aspect of this work is the addition of extensions to UML to accommodate task analysis. The modeling constructs have to accommodate:

- A description of users and their relevant characteristics
- A description of user behavior/intentions in performing the envisioned or supported task
- A specification of the abstract and concrete user interface.

WISDOM applies many changes and additions 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. But concerns arise about the frequent communication misadventures between HCI and Software Engineering specialists, as well as the tendency to misinterpret constructs such as use-cases, caused by different cultures having a different understanding of a versatile language like UML.

In the same vein as this work, Markopoulos (2000) and da Silva (2001) also proposed extensions to UML for interactive systems. In contrast, a task is represented as classes in WISDOM and by activities in the UMLi framework proposed by (da Silva, 2001). Mori, Paterno et al (2002) also suggested an extension of their task modeling notation, CTT (Concurrent Task Tree).

The above research shows that UML suffers from a lack of support for UI modeling. For example, class diagrams are not entirely suitable for modeling interaction, which is a major component in HCI. The IBM-OVID methodology is an attempt to provide an iterative process for developing an object-oriented model by refining and transforming a task model (Roberts, 1998).

### *3.3.4 Augmenting Use Cases for User Interface Prototyping*

Artim (1998), Constantine and Lockwood (1999), and Kruchten (1999) all tried to augment use cases to support interface design and prototyping. This integration is based on the synchronization of the problem specification and the solution specification; these two specifications are updated at each iteration through an assessment of impact of the changes in the models.

Theoretically, having a consistent use cases model that provides simple views for any actor and automatically includes the user's concerns should be enough to enable the software engineers to keep track of the user's needs during their design process. However, as Artim and the participants in his workshop (Artim, 1998) pointed out, the culture of software engineering does not include collaborating with the user in the process of building a better system. These sociological forces within development teams severely limit the user's impact in the development of the system, thus providing a system that fits to user interface specifications, rather than optimizing the fit to the user's needs. Thus, even though the development method directly determines the product being created, it is not the only factor.

Constantine and Lockwood (1999) try to harness the potential of use-cases with the goal of replacing task models and scenarios, which are generally proposed as a starting point for UI prototyping. They structure their method into five kinds of artefacts, organizing the three center ones by a map, so we respectively have the following:

- Maps:
  - 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.

These maps and models can be developed or enhanced concurrently, which departs from more traditional (albeit iterative) sequential approaches. In the attempt to completely specify the design methodology, they define the notion of essential use-cases. These essential use-cases aim to enhance usability by focusing on intention rather than interaction, and simplification rather than elaboration. The use-cases provide an inventory of user intentions and system responsibilities, focusing only on information considered essential and hiding unneeded information; this approach helps use-cases

adapt to eventual technological or environmental changes. Constantine gives a structure to essential use-cases, at the same time defining the syntax of the narratives. He also acknowledges the limitations of essential use-cases in the domain of software engineering; for this reason he advocates the use of essential use-cases only in the core process, where usability characteristics are essential.

Krutchen (1999) proposes to add a new artefact to the Rational Unified Process: the use-case storyboard. This artefact provides a high-level view of dynamic window relationships such as window navigation paths and other navigation paths between objects in the user interface. Use-case storyboards have to be written at analysis time, at the same time as the use-cases. They include many useful constructs such as:

1. Flows of events, also called storyboards. These are textual user-centered descriptions of interactions.
2. Class Diagrams. These are classes that participate in the use-cases.
3. Interaction Diagrams. These describe the collaboration between objects.
4. Usability Requirements. These are textual version of usability requirements.
5. References to the User-Interface Prototype. This is a text description of the user-interface prototype.
6. Trace dependency. This is a type of map of the use cases.

Krutchen (1999) also proposed guidelines on how to use this new construct. He recommends that a human factors expert should write these documents, because traditional software engineers will not design or use this artefact correctly, not being used to its philosophy. A big concern about this new technique comes from its practice of specifying the interface and the interactions at the beginning, rather that deriving them from the UI design, thus "putting the cart before the horse" and limiting the possibilities of the interface (Constantine, 1999). This also illustrates that use-cases can adapt to usability engineering, but there is no assurance that designers will use them adequately.

## 3.4 DESIGN KNOWLEDGE

In addition to efforts made by the UCD and SE communities to bridge the technical aspects of their methods, communication factors are also important. Like others (Sutcliffe, 2000; Henninger, 2000), we strongly argue that methods and tools for capturing and disseminating HCI and usability design knowledge and best practices can facilitate the integration and cross-pollination of the HCI and software engineering disciplines.

HCI has a long tradition of devising ways to capture knowledge so as to guide the design and evaluation of interactive systems. Prominent examples are guidelines (Vanderdonckt, 1999), interaction patterns (Erickson, 2000; Tidwell, 1998) and claims (Sutcliffe, 2000). Here, we summarize these methods and discuss how they can be extended to support effective integration.

### *3.4.1 Guidelines*

Vanderdonckt, (1999) defines a guideline as "a design and/or evaluation principle to be observed in order to get and/or guarantee the usability of a user interface for a given interactive task to be carried out by a given user population in a given context". A prominent example of a guideline collection is Smith and Mosier's set of 944 general-purpose guidelines (Smith, 1986).

A detailed analysis of the validation, completeness and consistency of existing guideline collections has shown that there are a number of problems with guidelines (Vanderdonckt, 1999). Guidelines are often too simplistic or too abstract, they can be difficult to interpret and select, they can be conflicting and they often have authority issues concerning their validity. One of the reasons for these problems is that most guidelines suggest a context-independent validity framework but in fact, their applicability depends on a specific context.

The general utility of detailed design guidelines for augmenting development processes has also been questioned. We argue that the massive context information that is necessary to describe the context of use of a guideline together with the problem of conflicting guidelines makes guidelines virtually useless for developers who are not experts in usability. Hartson and Hix (1993) noted that applying guidelines to specific situations requires a higher level of expert knowledge and experience than most interaction designers have.

Although guidelines remain an important tool for teaching user interface design, their utility to professionals is questioned and, until we find better means to help people apply them in specific context, they cannot be considered a successful avenue for the integration of HCI practices and SE.

### *3.4.2 Style guides*

A style guide is a document that contains descriptions of the usage and style of particular interaction components such as menus, dialogue boxes and messages. Commercial style guides such Apple "Human Interface Guidelines" (Apple, 1987), the Microsoft Windows Interface Style Guide (Microsoft, 1995), or the Java Look and Feel style guide from Sun Microsystems (Sun Microsystems, 1999) are often associated with a commercially available toolkit. They can act as a basis for customized style guides that are tailored for the products of an organization.

Style guides are mainly used during development and usability inspection of user interfaces to ensure consistency of user interaction designs. The development of a style guide is an important early activity for project teams. Style guides are a useful way to capture and document design decisions and to prevent constantly revisiting these decisions.

Although more specific than guidelines, style guides suffer from many of the same problems, such as conflicts, inconsistencies, and ambiguities. Furthermore style guides are limited to a very particular type of application or computing platform. Therefore, their ability to disseminate established HCI and usability practices to a wide audience is limited.

### *3.4.3 HCI and Usability Design Patterns*

The limitations of guidelines and style guides motivated some researchers to introduce interaction patterns, also called HCI patterns (Erickson, 2000, Tidwell, 1998). An HCI pattern is described in terms of a problem, a context and a solution. The solution is assumed to be a proven one to a stated and well-known problem. Many groups have devoted themselves to the development of patterns and patterns languages for user interface design and usability. Among the heterogeneous collections of patterns, Common Ground, Experiences and Amsterdam play a major role in this field (Tidwell, 1998).

Patterns provide more useful and specific information than guidelines, by explicitly stating the context and the problem and by providing a design rationale for the solution. Patterns contain more complex design knowledge and often several guidelines are integrated in one pattern. Patterns focus on "do this" only and therefore are constructive and less abstract.

In contrast, guidelines are usually expressed in a positive or negative form; do or don't do this. Therefore guidelines are useful for evaluation purposes. They can easily be transformed into questions for evaluating a UI.

Erickson (2000) proposed to use pattern languages as a descriptive device, a lingua franca for creating a common ground among people who lack a shared discipline or theoretical framework. In contrast, both Alexander (1977) (the father of patterns) and the software pattern community tend to use patterns more prescriptively. The software pattern community focuses on using patterns to capture accepted practice and to support generalization; Alexander's central concern is using patterns to achieve the ineffable "quality without a name," which characterizes great buildings and houses. HCI design patterns are generalizations of specific design knowledge that can increase quality of design.

Certain issues remain to be addressed in patterns and current HCI patterns languages. To begin with, there are no standards for the documentation of patterns. The Human-Computer Interaction community has no uniformly accepted pattern form. Furthermore, when patterns are documented (usually in narrative text), there are no tools to formally validate them. There should be formal reasoning and methodology behind the creation of patterns, and in turn, pattern languages. A language in computer science has syntax and semantics. None of the current pattern languages follow this principle; rather they tend to resort to narrative text formats as illustrated in the Experiences example. Finally, the interrelationships described in the patterns are static and not context-oriented. This is a major drawback since the conditions underlying the use of a pattern are related to its context of use.

### *3.4.4 Claims*

Another approach to capturing HCI design knowledge is claims (Sutcliffe, 2000). Claims are psychologically motivated design rationales that express the advantages and disadvantages of a design as a usability issue, thereby encouraging designers to reason about trade-offs rather than accepting a single guideline or principle. Claims provide situated advice because they come bundled with scenarios of use and artefacts

that illustrate applications of the claim. The validity of claims has a strong grounding in theory. This is also a weakness of claims, because each claim is situated in a specific context provided by the artefact and usage scenario. This limits the scope of any one claim to similar artefacts.

## 3.5 ORGANISATIONAL APPROACHES

We now turn to organisational approaches to filling the current gap between UCD and SE.

### *3.5.1 Evidence Based Usability Engineering*

Evidence-based Usability Engineering (EBUE) is an approach that addresses the problem of the integration, adoption, and improvement of UCD methods at the organizational level (Metzker, 2003). It acknowledges that a team and an organization in general has to adopt new techniques in a progressive manner, first by recognizing and assessing the strengths of certain approaches and then by a process of selecting and refining these techniques. EBUE is art of an integrative framework - a UCD meta-model - to support measurement-based integration of usability concerns in any software engineering process.

EBUE discards the philosophy of a static, one-size-fits-all UCD process model. Instead, it proposes using a configurable pool of UCD methods. Examples of UCD methods considered for the UCD method pool are heuristic evaluations, card sorting, cognitive walkthroughs and user role maps.

Based on the characteristics of the project at hand, specific methods are selected from the UCD method pool. The selected methods form a UCD process kit, which is tailored to the characteristics of the project at hand. During the course of a project, qualitative feedback such as comments on and extensions of UCD methods is gathered from the project team and used to improve the method pool. This could be done in post-mortem sessions, which are already successfully used in software development projects to reflect on software processes (Birk, 2002). Additionally, quantitative feedback on the utility and usability of UCD methods from the perspective of the project team should be collected in the form of quick assessments. The results of such assessments are a measure of the quality of UCD methods as perceived by project team members.

The quantitative feedback is accumulated and integrated across project boundaries and used to extract relationships between UCD methods, project characteristics and the perceived utility of UCD methods. These relationships can be exploited in future projects as a body of evidence to choose optimal UCD method configurations for defined project characteristics. Figure 3.4 provides an overview of the UCD meta-process as suggested by EBUE.

The first cross-organizational studies on the acceptance of the EBUE framework by practitioners show a high level of acceptance of the approach and a good compatibility with current industrial software development practices (Metzker, 2003).

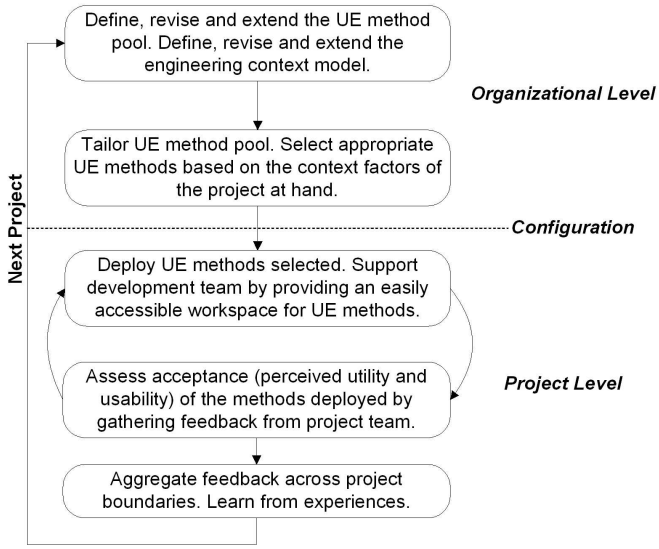


**Figure 3.4** Meta-Model for institutionalizing UCD methods

In our view, adoption-centric approaches such as EBUE have a high potential impact on current software development practice, helping to overcome the lag between usability engineering research and practice.

### *3.5.2 Concurrent Evolution of Interface and Software Components*

There is a compelling need to define a more flexible and efficient software development process that can support concurrency in the design process and requirement process. If software architecture and UI design are interdependent, as Bass and his colleagues have demonstrated (Bass and John, 2003), how should the software development process be organized? Although a better understanding of these interdependencies is emerging, it is not yet clear how to represent and coordinate them in the software development process.

### *3.5.3 Concurrent Processes and Cultures*

Throughout most of the previous sections, we have assumed that a software engineering approach is already established in a software development team, and that the task is to determine where user-centered design techniques could be incorporated. Although this is correct in many cases, the inverse can also occur. A development process may be dominated by HCI concerns and we may fail to integrate a software engineering method into it. For example, Web site design and development is often driven by an information content management paradigm or one of graphical interface design. However, Web sites are evolving toward providing advanced features such as elec-

tronic payment and intranet services. In this context, we are moving away from a paradigm of online brochure to one where Web sites are Internet-based transactional services. Web designers have to consider new design issues such as software maintenance, quality management, security, etc.

Web development is not the sole example. To some extent, the computer gaming industry is another domain where software development is driven by non-traditional methods. In this industry, the entertainment perspective is the dominant paradigm. Role playing and user experience are often at the core of the game development cycle. Although game producers do have software engineering processes in place, it is quite common to find the production department struggling with the problem of interfacing the creative team with the developers, or struggling to implement a rigorous software quality process in a culture that thrives on creativity.

It remains an integration challenge to determine how to merge traditional software engineering techniques into a culture and a process dominated by methods inspired from other fields such as content publishing, entertainment and HCI driven approaches.

### *3.5.4 Supporting Continuous Improvement*

The integration of HCI and SE will obviously not yield a one-size-fits-all solution. A better understanding will be required of the appropriateness of solutions to contexts. What are the factors that should be involved in adapting or improving an integration framework in a given context? This question needs to be addressed in the following four dimensions:

1. Activity, including synchronization between activities;
2. Actor, including team organizations and communication schemes;
3. Artefact, including the formats and structure of the artefacts;
4. Tool, including the use of tools to facilitate the communication.

In developing any integration framework, three types of improvements can be supported in the software development lifecycle:

Elementary improvements are modifications that affect only one element of the SDL (software development lifecycle). For example we can change a very specific artefact, and thereby indirectly change the way to perform the corresponding activity. A typical example is what Krutchen called use-case storyboards. Another elementary improvement can be the addition of usability inspection in certain activities.

Complementary improvements consist in adding new elements in the SDL. For example, we can propose a framework for using usability research methods (usability walkthrough, user testing) in the SDL. This approach complements the existing SDL and does not force a change in the structure.

Structural improvements affect the communication process or the structure of the SDL. They add a set of activities in the SDL that can change the sequence of activities. Constantine uses an outside-in communication scheme in his usage-centered design.

Nunes adopts a loose communication scheme that is only constrained by the need to support the dual model (analysis model, interaction model).

### *3.5.5 Agile Methods and UCD*

Agile methods are gaining significant acceptance, or at least visibility, in the software engineering field, and they are also gaining adepts in industry and even in some engineering schools. The Agile approach emerged as a response to highly organized and documentation intensive processes such as RUP(©), the Rational Unified Process that is widely in use nowadays, and other proprietary processes. The claim made by the agile community is that heavy documentation and processes create latency, inefficiency, and a tendency for the development team to adhere to inappropriate requirements instead of responding to changes in their definitions. Because user requirements are particularly prone to changes as the users uncover new needs and correct their initial requirements as the software application unfolds, this claim appears to address an important issue in HCI.

Moreover, the Agile movement also emphasizes the importance of human factors in the development process as expressed by two of their four core values: (1) individuals and interactions over processes and tools, and (2) customer collaboration over contract negotiations (see `http://www.agilemanifesto.org`). Here, again, the agile movement appears to lean towards a user-centered approach.

In spite of these principles that suggest the approach is user-oriented, agile processes have come under criticism from the UCD community, such as this quote from Constantine (2001):

> Informants in the agile process community have confirmed what numerous colleagues and clients have reported to me. XP and the other light methods are light on the user side of software. They seem to be at their best in applications that are not GUI-intensive. As Alistair Cockburn expressed it in email to me, this "is not a weak point, it is an absence." User-interface design and usability are largely overlooked by the agile processes. With the possible exception of DSDM [(http://dsdm.org)] and FDD `http://www.featuredrivendevelopment.com/`, users and user interfaces are all but ignored.

Amongst the most important shortcomings addressed to agile methods, the lack of distinction between the client and the user is probably the most critical. Because clients are often represented by people from marketing or from management, user needs are often misunderstood. Moreover, Agile methods do not provide guidance on how to validate user requirements. There is no reference to principles such as those found in ISO 13407 or Maguire (Maguire, 2001b) and, thus, the resulting application could still miss important user requirements. This problem is particularly important in software designed for teamwork where social interactions and business processes are difficult to anticipate; the problem is also common in software with a high degree of novelty.

However, recent efforts are addressing the shortcomings in agile methodologies by integrating UCD principles. Patton (2002), for one, reports a successful experience in integrating the interaction design approach in an agile approach already in place. But

more research is required to arrive at a comprehensive integration of agile approaches and UCD.

## 3.6 CONCLUSION

In this chapter, we highlighted some of the obstacles to integrating HCI and usability concerns in mainstream software development. We surveyed many of the approaches proposed for filling in the current gaps between HCI, usability and software engineering. The fundamental questions addressed in this chapter are:

1. How can the software engineering lifecycle be re-designed so that end users and usability engineers can participate actively?

2. Which usability artefacts are relevant and what are their added values and relationships to software engineering artefacts?

3. What are the usability techniques and activities for gathering and specifying these relevant usability artefacts?

4. How can these usability artefacts, techniques and activities be presented to software engineers, as well as integrated in the software development lifecycle in general? What types of notations and tool support are required?

The frameworks summarized in this chapter provide partial answers to these questions. Although the answers are not complete, they are useful for usability and software specialists who are interested in the development of methodologies and standards, who have researched or developed specific user-centered design techniques or who have worked with software development methodologies. They offer insights in how to integrate user-centered best practices and user experiences with software engineering methodologies.

Patterns and use cases are useful artefacts for bridging gaps between HCI and SE. Boundary objects serve each discipline in its own work and also act as a communication tool to coordinate work across disciplines. For example, a designer uses patterns to explore design ideas in the space of presentation and navigation; a usability expert uses them to perform an early usability test; a software engineer uses them as part of the specification of the interface code. The patterns perform different functions for each discipline, yet provide common ground for sharing knowledge.

However, we have not covered all aspects of integration. In particular, the cultural gaps between the SE and HCI is an important aspect that is not addressed here. We refer the reader to the chapters in this book by Jerome and Kazman (chapter 4), Sutcliffe (chapter 5), and also Blomkvist (chapter 12). We could also have raised the issue of academic and professional training, which are key factors to cultural and interdisciplinary team integration. It is our own teaching experience that some software engineering students are enlightened by sunddenly discovering the importance of human factors in designing interactive software. Yet, most of them will have very little training in HCI. The same could be said about software project managers who are often the ones responsible for implementing the development process, defining and

staffing the activities, etc. Our knowledge of software management processes indicates that it suffers a lack of awareness of the need to integrate UCD and SE practices. All these issues would deserve to be covered.

Finally, we believe a forum is required for promoting and improving HCI and usability engineering techniques and software engineering approaches in the two communities. An example of this type of forum would be to combine the IEEE-ICSE and ACM-SIGCHI conferences for one year where avenues for:

- Sharing ideas about potential and innovative ways to cross-pollinate the two disciplines
- Disseminating successful and unsuccessful experiences in how to integrate usability into the software engineering lifecycle, in different sizes of organization
- Building a tighter fit between HCI and software engineering practices and research.

## Acknowledgements

The authors would like to express their sincere thanks to all of the participants in the workshops that they organized over the last five years. Thanks also to the National Science and Engineering Research Council of Canada and Daimler Chrysler, Software Technology Centre, for their financial support. We are also grateful to Fran cois Aubin for sharing with us figure 3.1's perspective.