Bridging the Gaps Between Software Engineering and Human-Computer Interaction

ICSE'03

International Conference on Software Engineering

Portland, Oregon

May 3-11, 2003

Contents

Model-based Simulation of Web Applications for Usability Assessment
R. Chatley, J. Kramer, J. Magee, S. Uchitel……………………..……………………..….5
Improving software usability through architectural patterns
N. Juristo, M. Lopez, A. Moreno, M. Sanchez……………….……………………...…12
Software Engineering and HCI Techniques Joined to Develop Virtual Environments
M-I Sanchez-Segura, A. de Amescua Seco, J-J. Cuadrado, A. de Antonio…………….20
Integration of Usability Techniques into the Software Development Process
X. Ferre…………………………………………………………………………….....28
Usability Engineering integrated with Requirements Engineering
B. Paech, K. Kohler……………………………… ……………………………….....36
RUPi – A Unified Process that Integrates Human-Computer Interaction and Software Engineering
K. Soares Sousa, E. Furtado ……………………………….…………………..….41
From HCI to Software Engineering and Back
J. Creissac Campos, M. Harrison…...…………………………………………..……49
Where SE and HCI Meet: A Position Paper
M. Willshire……………………. ………………………………………………….. 57
Scenario-based Assessment of Software Architecture Usability
E. Folmer, J. van Gurp, J. Bosch…..…………………….……………………….... 61
Software Engineering Overlaps with Human-Computer Interaction: A Natural Evolution
A. Milewski……………………………………………………………….………… 69
Undergraduate Software Engineering Curriculum Enhancement via Human-Computer Interaction
S. Ludi……………………………………………………………………..………72
Improving UML Support for User Interface Design: A Metric Assessment of UMLi
P. Pinheiro da Silva, N. Paton……………………………………………………..……76
Interaction Modeling as a Binding Thread in the Software Development Process
S. Barbosa, M. de Paula………………………………………………………...………84
Finding Boundary Objects in SE and HCI: An Approach Through Engineering-oriented Design Theories
A. Walenstein……………………………………………………..………………..92
Separation in Theory – Coordination in Practice
T. Clemmensen, J. Norbjerg…………………………………………………......100
Usage-Centered Design and Software Engineering: Models for Integration
L. Constantine, R. Biddle, J. Noble..…………………………………….………106
A Software Prototyping Framework and Methods for Supporting Human's Software Development Activities
J. Guan, Luqi………………………………………………………………..………114

Model-based Simulation of Web Applications for Usability Assessment

Robert Chatley, Jeff Kramer, Jeff Magee, Sebastian Uchitel
Dept of Computing, Imperial College London
{rbc,jk,jnm,su2}@doc.ic.ac.uk

Abstract

In this paper we discuss an approach for simulating the behaviour of interactive software systems, before starting on any of the actual implementation, based on a model of the system at the architectural level. By providing a mock-up of the final user interface for controlling the simulation, it is possible to carry out usability assessments of the system much earlier in the design process than is usually the case. This means that design changes informed by this usability assessment can be made at this early stage. This is much less expensive than having to wait until an implementation of the system is completed before discovering flaws and having to make major changes to already implemented components. The approach is supported by a suite of cooperating tools for specification, formal modelling and animation of the system.

1. Introduction

In recent years, there has been increasing regard for usability as a quality attribute for software. Techniques have been developed by which the usability of systems can be assessed [1][2], however these techniques often involve activities such as interviewing users, or recording their use of the system with a video camera. Using them therefore requires a working implementation of the system and a representative user. This means that usability assessment and subsequent improvement may only be carried out late in the development process. At this stage it is very expensive to go back and make major changes to the design[3].

We present a technique for modelling the system at the architectural level, including its interaction with the user, and connecting this to a realistic mock-up of the user interface. This allows some traditional usability assessment techniques to be applied much earlier in the design process, once the architecture has been determined, but before detailed design and implementation have started. It also allows for checking properties of the system, for instance finding possible deadlocks. In particular we have concentrated on developing techniques that allow the realistic simulation of web and e-commerce applications, as usability is a key factor in the success of these types of application[11]. The remainder of this paper describes the approach in detail, the supporting tools and examples of their use.

The input to our modelling and simulation technique is a set of scenarios describing interactions between different components (including the user) of the system. In previous work on scenarios we have developed techniques for analysing formal models built from sets of scenarios described in the form of Message Sequence Charts (MSCs) [4]. Here we use a set of MSCs to specify behaviours of the user and system. From these we synthesise a behaviour model which drives the simulation.

2. Background

The Labelled Transition System Analyser (LTSA) tool allows the construction and checking of models of finite state processes. This tool, which is described fully in [5], allows us to build models of the behaviour of complex systems which are amenable to formal analysis in the form of labelled transition systems (LTS), and to check properties of these models mechanically. We have extended this tool to allow system behaviour to be specified by means of sets of scenarios described in the form of MSCs. This work is described in [6].

Using these techniques, we are able to build models of the behaviour of systems, which include the user, built up from scenarios gathered during the requirements elicitation process. In this way we can model and investigate the interactions between the system and the user.

In order to make it possible to test the quality of the user's experience of the system based on the model, some sort of animation needs to be provided to allow the user to interact with the model through an interface which at least approximates that which they would use in the final system. Animation has previously been used to make it easier to interpret the meaning of traces returned from model checking[7]. A trace to deadlock for example may be illustrated in the problem domain by replaying the relevant sequence of actions as a graphical animation.

In order to simulate web applications, and to give a more accurate representation of the experience of using a web application, we have developed a new animation technology allowing users to interact with the model through the familiar interface of web pages displayed in a standard web browser. This technology is described in more detail in Section 4.

3. Animating Models

The behaviour of a model can be interactively explored using the LTSA tool. The output of such an execution is essentially a trace of action names. Each action is the abstract representation in the model of an input or output of the proposed system [5]. One of the features provided by the unextended LTSA is the ability to run or to step through a trace of possible actions, and to see the resulting state changes reflected in the state machines. LTSA can display graphically state machines reflecting the LTS for each separate component or for a composed system. The current state and the last transition made are highlighted on the display. The user can trigger any of the currently available actions by selecting them from a dialog box, causing a transition to occur.

A difficulty arises in interpreting the meaning of traces in relation to the original problem domain. Even when the meaning is clear to the model designer, the problem of communicating model to non-technical stakeholders of a system remains. Because we want to explore and assess the usability of the system based on these behaviour models, we need to develop techniques that enable us to convey the meaning of the model, in terms of the actual system that it is intended to represent, to an end user. As discussed in [7], there is a lot to be gained from using graphic animations to communicate the results of analysing formal models of systems.

One example of an animation technique which can be used with LTSA is SceneBeans[8]. SceneBeans is a Java framework for building and controlling animated graphics. It removes the drudgery of programming animated graphics, allowing programmers to concentrate on *what* is being animated, rather than on *how* that animation is played back to the user. SceneBeans is based upon JavaBeans and XML. Its component-based architecture allows application developers to easily extend the framework with domain-specific visual and behavioural components.

4. Simulating Web Applications

Scenebeans has been used to present a graphical representation of how components interact in models of complex concurrent systems, for example switching between channels on a modern television set.

However, using this particular technology to animate the model, it is difficult to produce an interface that accurately represents the type of interface users are accustomed to for a web or e-commerce application, namely that of web pages in a web browser. Therefore, it is difficult to use such a simulation for usability testing and gain an accurate idea of users' responses to the actual system. In the television example, there are sufficient differences between the way that a user would interact with the on-screen representation of a remote control and the way that they would use a physical handheld remote control to render any usability measurements taken using the simulation fairly meaningless. To give an accurate idea of the usability of the system, a mock-up of the eventual user interface needs to be provided that is much closer to what the user will actually experience.

To attack this problem in the arena of web and e-commerce applications, we have developed a new animation technology, which allows the user to interact with the model of the system by means of clicking on links and buttons in a web browser. The LTSA tool was extended so that it can provide an interface to the model through a set of web pages which can be viewed in a standard web browser. This extra functionality was provided by writing a plugin to be used with LTSA's extension mechanism, as with the Message Sequence Chart extensions.

The benefits of the approach to simulation given here as compared to, for instance, that taken in [12] are that our simulation tools work with our existing behaviour modelling tools without having to change the representation in any way, and that the appearance of the interface to the simulation can easily be made to reflect a designer's proposal for the look of the final system.

The web animator plugin allows us to associate fragments of HTML with different possible actions. These can be hyperlinks, buttons or any other interactive element commonly found on web pages. The plugin will dynamically compose a web page from these fragments and serve it to a web browser to display. The user can then click on any of the buttons or links in the browser to trigger a transition in the LTS.

The basic architecture of the Web Animator is shown in Figure 1. The plugin adds a mini webserver to the LTSA so that it can communicate with a standard web browser by means of the HTTP protocol.

The LTSA produces an XML document describing the available transitions each time that a new state is reached. An XSLT[9] transformation is applied to this XML document based on an XSL stylesheet. This stylesheet describes a transformation from XML to HTML which defines the visual appearance of the web pages. This HTML is then sent over the network via HTTP to the browser where it is rendered.

When the user is presented with such a webpage, they can click on any of the links or buttons on the page, which will cause the browser to send an HTTP request back to the server. The server analyses this request to detect what action the user has requested and triggers an appropriate transition in the LTS.

Extra decision logic has been added so that it is possible to make a distinction between actions that are carried out by different parties. This allows us to distinguish between actions performed by users and those that are carried out by components of the system without any user intervention. We call these respectively “user actions” and “system actions”. An external XML file is used to configure which actions are to be classed as system actions and which as user actions.

If in any state there are no actions available to the user, only system actions, the tool will pick a system action to perform and continue to execute system actions until a state is reached where there is a user action available. On reaching such a state control is returned to the user and the user can choose which of the actions available to them to perform next. It is possible to control the way in which the tool selects an action from the available system actions in any state by including extra information in the XML configuration file. Boolean expressions can be encoded in the XML, which can be used to make decisions on which system action to perform. These expressions can also test data which may have been input by the user through fields on the web page interface. For example, a typical scenario might be that of logging in to a website with a username and password. Depending on whether the username and password are entered correctly, the next page that the user sees will be different. This can be modelled by having a choice of two system actions *authenticate* and *reject*. If the username and password match the expected values, the system should perform the *authenticate* action, otherwise it should perform the *reject* action and ask the user to try again.

If the simulator has no extra information to guide its choice, it simply makes a random selection from the available system actions.

The visual appearance of the web pages is described in an XSL stylesheet. This is a standard way of expressing a transformation from XML to another data representation, in this case HTML. This technique is itself commonly used in web and e-commerce applications. Because the output is standard HTML, we can achieve an interface which is very close to that that might be used in the final system.

The separation of concerns, separating the definition of the visual representation and the extra decision logic from the scenarios and the specification of the behaviour model, means that we can achieve a better simulation as the different parts of the model can be worked on by different people, for instance a graphic designer could produce the visual representation without having to learn about MSCs or behaviour models. We can also change the visual representation and which actions are system or user actions independently of the behaviour model, and so

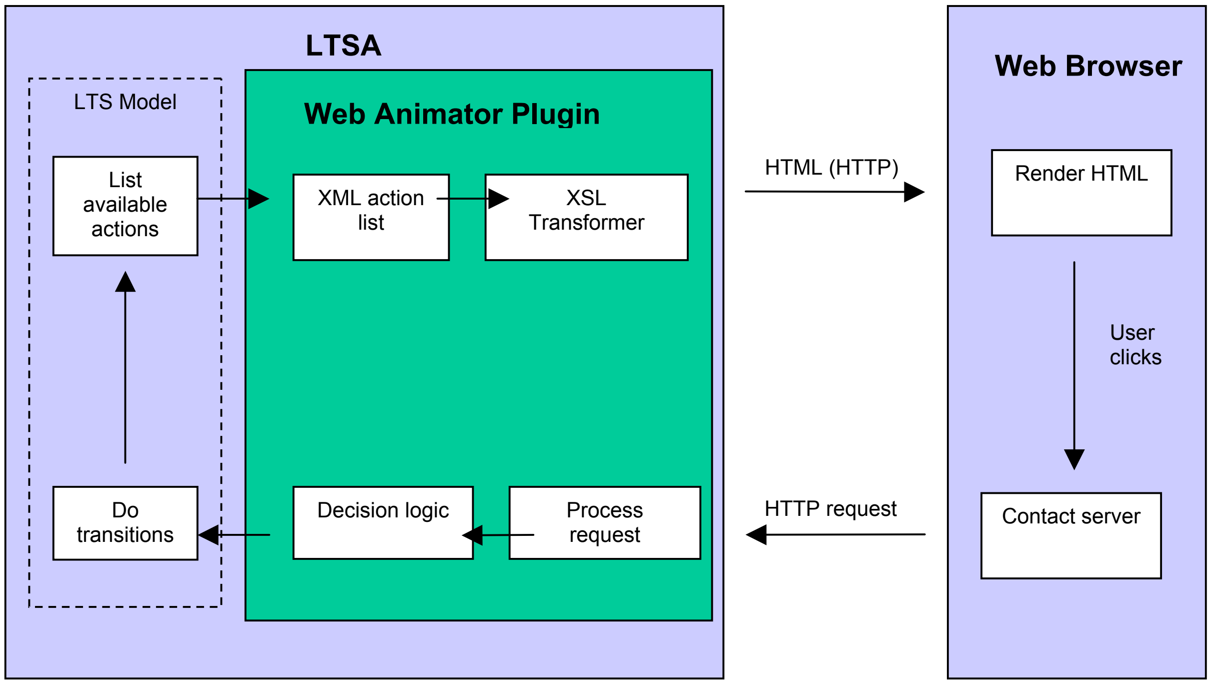


Figure 1 : Basic architecture of the web animator plugin.

do not have to recompile the model to make these changes.

The fact that the web animator serves web pages by means of the standard HTTP protocol means that we can run the system over a network, and so can observe the model on one computer whilst running the browser with the user interface on another. This greatly increases the exibility of the simulation environment.

5. Case study: LogicDIS eSuite

The eSuite product developed by LogicDIS (a Greek company who is one of the commerical partners in the STATUS[1] project) is a system that allows access to an ERP (Enterprise Resource Planning) system, through a web interface. The system employs a tiered architecture commonly found in web applications. The user interfaces with the system through a web browser. A web server runs a Java servlet and some business logic components, which communicate with the ERP.

In this case study we first give examples of scenarios describing possible uses of the system, and the interactions between system components.

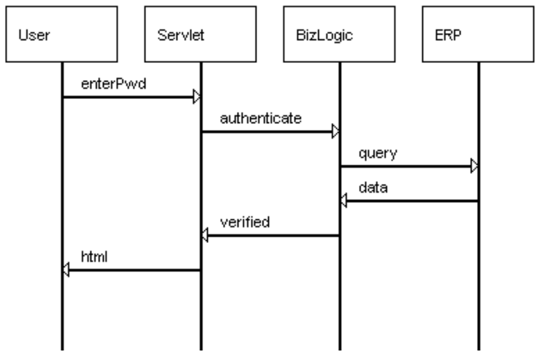


Figure 2 : An example scenario "Login" from the eSuite model.

Figure 2 gives an example of a scenario from the eSuite model. The four components that take part in the scenario are shown as *instances*. They are the user (which corresponds to a human with a web browser, messages originating from the user are sent when the user clicks a button in the browser), the servlet (software running on the web server which receives requests from the browser), the business logic (BizLogic) at the heart of the eSuite application, and the ERP, which is effectively treated as an external database.

The sequence of messages in Figure 2 show what happens when a user successfully logs in to the system. A chain of messages cascades through the tiers of the architecture and information comes back, finally resulting in an HTML page being displayed to the user informing them that they are now logged in to the site. This is just one of a large number of possible scenarios that can occur. For each of the scenarios that we are interested in, we construct a basic MSC like this. We can then describe the sequence in which these scenarios can occur by using a high level message sequence chart (hMSC). Figure 3 shows part of the hMSC for the eSuite model. It shows how scenarios can occur in turn (the completion of one enabling another to be entered), or in iteration as is the case with FailedLogin (a user can repeatedly attempt to log in, get their password wrong and try again).

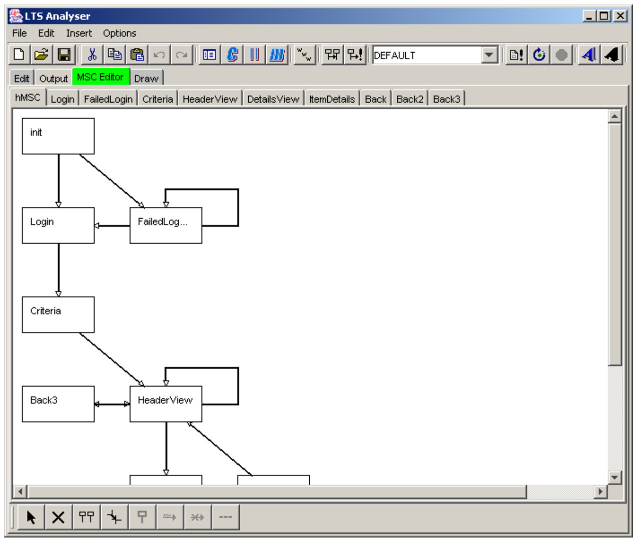

LTS Analyser
File Edit Insert Options
DEFAULT
Edit Output MSC Editor Draw
hMSC Login FailedLogin Criteria HeaderView DetailsView ItemDetails Back Back2 Back3
init
Login
FailedLog...
Criteria
Back3
HeaderView

Figure 3 : Part of the hMSC for the sSuite model displayed in the LTSA tool.

The specification consists of one hMSC and a number of bMSCs. In the tool, each bMSC has its own tab, as does the hMSC. Double-clicking on a scenario in the hMSC will drill down, opening a detailed view of the scenario as a bMSC.

From this set of message sequence charts, we use the tool to generate a textual description of the system in the FSP process calculus[5]. This can then be compiled into a set of labelled transition systems, which we can again view graphically. Figure 5 shows a graphical representation of the LTSs corresponding to the different components of the system. The top diagram in the tool represents the user. It contains only the states and actions

[1] STATUS is an ESPRIT project (IST-2001-32298) financed by the European Commission in its Information Society Technologies Programs

that pertain to the user. The highlighted states show the current position in each component's state machine at a point in the middle of a simulation.

To determine which actions should be controlled by the user during the simulation, and which were internal system actions, an XML configuration file was written detailing user and system roles. A fragment of the configuration file is shown in Figure 4. It defines the actions that are available to the user, and the conditions under which *verified* (a system action) can be performed, in this case when the login name and password match the expected values.

```
<role name="user">
   <possibleaction>enterPwd</possibleaction>
   <possibleaction>search</possibleaction>
   <possibleaction>orderHeader</possibleaction>
   <possibleaction>orderDetails</possibleaction>
   <possibleaction>itemDetails</possibleaction>
   <possibleaction>back</possibleaction>
</role>

<action name="verified">
    <conditions>
      <and>
        <equal key="login"       value="DEMO" />
        <equal key="password"    value="DEMO" />
      </and>
    </conditions>
</action>
```

Figure 4: A fragment of the XML configuration file.

An XSL stylesheet with templates corresponding to the various possible user actions was also written. Images were supplied by LogicDIS which reflect the graphical appearance of their application. These are easily incorporated into the interface using standard HTML, and our tool allows us to provide standard headers and footers for each genrated page. It is also possible to use cascading style sheets to apply a custom style to all generated pages. These are commonly used in the development of web applications to achieve a consistent look across all pages, and provide us with an easy way of reflecting the envisaged design of the finished application in the simulation.

Figures 6, 7 and 8 show mock-up interface screens from the simulation as the user walks through logging in to the eSuite application and searching for an order. As can be seen from the pictures, we have managed to replicate the look of a web application interface in the simulation very closely by using standard web page features like text-boxes, buttons and hyperlinks. The inclusion of graphics and stylesheets from LogicDIS help to reflect the look of a finished application, and show how the visual aspect of the simulation is separated from the behavioural part. Using these images, produced by a specialist designer, is a simple matter of including a couple of lines of standard HTML in the XSL stylesheet.

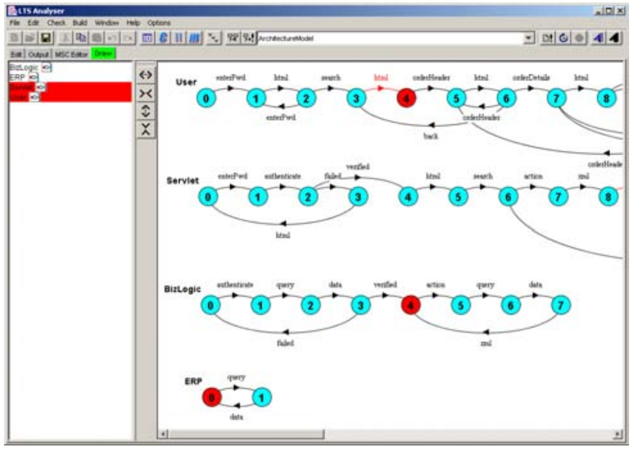


Figure 5: LTSs corresponding to the different system components, displayed in LTSA.

The user can interact with the simulation of the system in exactly the same way as they would with a real web based system, by clicking on links and buttons in the web browser. Transitions occur in the underlying LTS model, unseen by the user, and another web page is returned to them.

In this way we can allow the user to experience interacting with the system, and find out whether the series of interactions they go through to perform tasks, or find things on a website, is easy to learn, efficient to use, consistent, predictable and so on, using traditional usability assessment techniques, but using the simulation rather than the finished system.

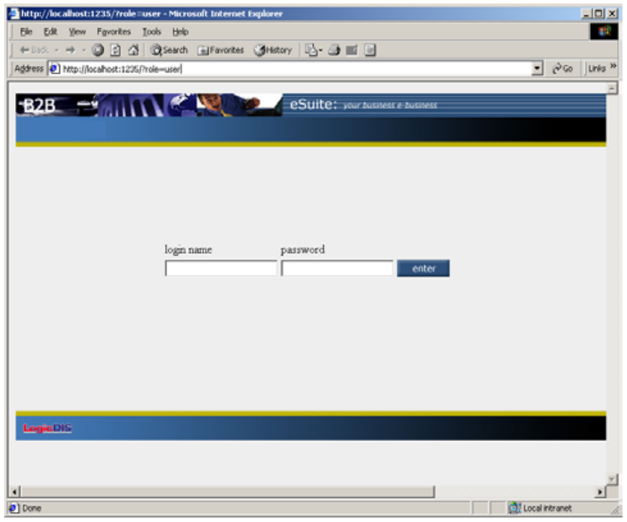
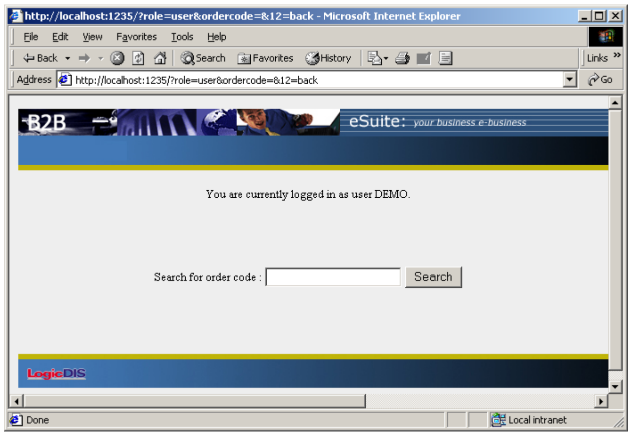

http://localhost:1235/?role=user - Microsoft Internet Explorer
Address http://localhost:1235/?role=user
B2B
eSuite: your business e-business
login name
password
enter
LogicDIS
Done
Local intranet

Figure 6: A simulation interface screen displayed in a web browser, showing the login screen.

If it is felt that the usability of the system could and should be improved, suitable changes can then be made to the design of the way that the user interacts with the system simply by making changes to the MSC specification and recompiling the model, after which the simulation can be run again. At this early stage in the development process it is still cost effective to make these changes, as development effort has not yet been spent on implementing a detailed design.


http://localhost:1235/?role=user&ordercode=&12=back - Microsoft Internet Explorer

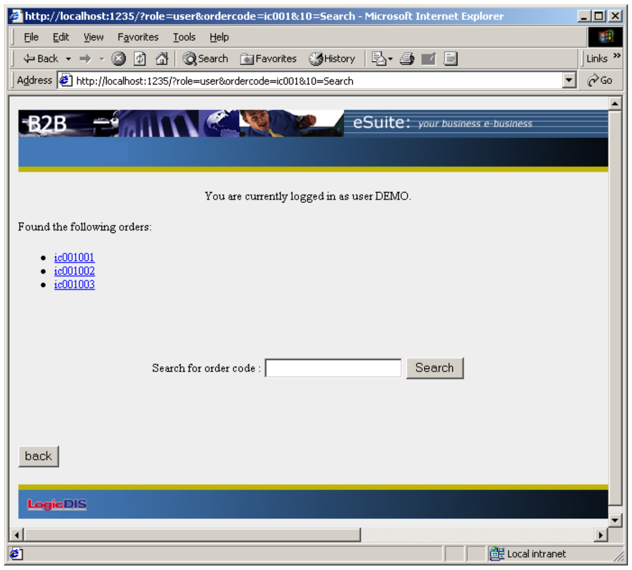
Address http://localhost:1235/?role=user&ordercode=&12=back

You are currently logged in as user DEMO.

Search for order code :

File Edit View Favorites Tools Help
Back Search Favorites History
Links
Go
B2B
eSuite: your business e-business
Search
LogicDIS
Done
Local intranet

Figure 7: When the user has logged in they can perform a search.


http://localhost:1235/?role=user&ordercode=ic001&10=Search - Microsoft Internet Explorer

Address http://localhost:1235/?role=user&ordercode=ic001&10=Search

You are currently logged in as user DEMO.

Found the following orders:

- ic001001
- ic001002
- ic001003

Search for order code :

File Edit View Favorites Tools Help
Back Search Favorites History
Links
Go
B2B
eSuite: your business e-business
Search
back
LogicDIS
Local intranet

Figure 8: The results of a search operation are displayed.

6. Conclusion

We have shown that by building a behaviour model for a system from a set of scenarios, and linking it to a suitable animation, a user can interact with a simulation of a system with a similar experience to using the real application. Simulations can be built which reflect interaction with a real system to the extent that they are suitable for performing usability assessments. Scenarios in the form of MSCs provide an easy method for describing individual system behaviours from which we can synthesise a formal model. This model is then used to constrain the behaviour exhibited in the simulation.

We presented a new animation tool which allows an interface in the form of a set of web pages to be attached to a behaviour model in the form of a labelled transition system. By harnessing standard web protocols, the tool allows a user to interact with a simulation of a web application using a standard web browser interface. This allows us to provide a realistic simulation. By separating the visual aspects of the simulation from the behaviour model we allow each to be developed separately by specialists and then easily combined.

We presented a case study in which we created a simulation of an existing e-commerce application, starting with a set of scenarios. With relatively little effort we were able to recreate the look and feel of the original application's interface in our simulation.

Using this technique affords us the possibility of performing usability tests early in the development process. Identifying usability weaknesses at this stage allows for significant changes to be made to the design of a piece of software without incurring great expense. Traditionally the results of usability tests have lead only to fairly cosmetic changes to the interface of systems, concerning the display and layout of data. Changes to the way that the user interacts with the system may require much greater change to a system, possibly at the architectural level. For instance, the ability to undo actions cannot be added as a last minute feature, it must be factored into the architecture at an early stage [10]. Early detection of usability problems and possible improvements using the simulation techniques described here can help us to engineer for usability from the start of the development process.

7. Future Work

A feature of web applications not explored here is the possibility that more than one user is using the system at the same time. In future work we hope to simulate the effect on the system of multiple concurrent users, to see whether the action of one user may affect another user's experience of the system, and whether this may be the cause of unexpected behaviour.

By combining this simulation approach with some work on stochastic modelling, it would be possible to introduce effects such as non-deterministic time delays in certain parts of the system (for instance a delay during server processing before a page is returned to the client). Introducing these effects could lead to even more realistic simulations.

If there are properties of systems which we can categorise as being undesirable from a usability perspective, it would be interesting to try to detect these using a model checking algorithm.

Another extension of this work might be to use the simulation interface to elicit further scenarios and elaborate a partial model of the system as part of a user-centred design process.

8. Acknowledgements

We gratefully acknowledge the support of the European Union under grant STATUS (IST-2001-32298), and the partners working on the STATUS project.

9. References

[1] Preece, J., Y. Rogers, H. Sharp, D. Benyon, S. Holland, T. Carey. *Human-Computer Interaction*. Addison Wesley, 1994.

[2] Constantine, L.L., L.A.D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Addison-Wesley, New York, NY, 1999.

[3] Brooks, Jr., F.P., 1995: *The Mythical Man-Month: Essays on Software Engineering, Twentieth Anniversary Edition,* Reading, MA: Addison-Wesley

[4] S. Uchitel, J. Kramer and J. Magee. "Negative Scenarios for Implied Scenario Elicitation", Proceedings of 10th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'02)

[5] Magee J., and J. Kramer, *Concurrency – State Models and Java Programs*. John Wiley & Sons, March 1999

[6] S. Uchitel, R. Chatley, J. Kramer and J. Magee. "LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios", Proceedings of TACAS 2003

[7] J. Magee, N. Pryce, D. Giannakopoulou and J. Kramer, "Graphical Animation of Behavior Models"

[8] N. Pryce and J. Magee, "SceneBeans: A Component-Based Animation Framework for Java"

[9] J. Clark, "XSL Transformations (XSLT) Version 1.0", http://www.w3.org/TR/xslt

[10] L. Bass, B. John, J. Kates. "Achieving Usability Through Software Architecture", CMU/SEI Technical report 2001

[11] Nielsen J., *Designing Web Usability*, New Riders Publishing, Indianapolis, 2000

[12] A. Egyed and D. Wyle, "Statechart Simulator for Modelling Architectural Dynamics", Proceedings of the 2nd International Working Conference on Software Architecture (WICSA), Amsterdam, 2001.

Improving software usability through architectural patterns

Natalia Juristo
School of Computing - Universidad Politécnica de Madrid, Spain

Marta Lopez
School of Computing - Universidad Complutense de Madrid, Spain

Ana M. Moreno
School of Computing - Universidad Politécnica de Madrid, Spain

M. Isabel Sánchez
School of Computing - Universidad Carlos III de Madrid, Spain

Abstract

This paper presents an approach for improving final software system usability by designing for usability, in particular by addressing usability issues in the software architecture. This approach differs from the traditional idea of measuring and improving usability once the system is complete. The work presented in this paper is part of the research conducted within the European Union - IST STATUS related to the development of techniques and procedures for supporting a forward-engineering approach to improve usability in software systems at the architectural level. In particular, we present the ongoing research about usability improvement by including architectural patterns that provide solutions for specific usability mechanisms.

1. Introduction

One reason why software architecture research is attracting growing interest is the direct relationship between architectural decisions and the fulfilment of certain quality requirements [1]. The goal is to assess software architecture for specific quality attributes and make decisions that improve these attributes. In short, a software architecture needs to be explicitly designed to satisfy specific quality attributes.

Moreover, usability is considered as just another quality attribute [2] and, therefore, we should also be able to design software architectures for usability as we do for other quality attributes.

The work presented in this paper is part of the insights and techniques developed in the STATUS project (SofTware Architectures That support USability)[1]. The goal of this project is to develop techniques and procedures to support a forward-engineering perspective to usability in software architectures, as opposed to the conventional backward-engineering alternative of measuring usability on a finished system and improving it once the system is practically complete.

In this paper, we will focus on presenting and discussing the ongoing STATUS research about architectural-level usability improvements.

For this purpose, section 2 shows the approach taken to decompose usability into levels of abstraction that are progressively closer to software architecture. These progressive levels are represented by the concepts of usability attributes, usability properties and usability patterns.

Then, section 3 shows how to incorporate the usability characteristics represented by the usability patterns into a generic software architecture. For this purpose, we will use the concept of architectural pattern, which specifies, in terms of components and their interrelationships, definite solutions for incorporating aspects that will improve final system usability into an architectural design.

Finally, section 4 presents future work to be done to complete and validate the approach taken in this paper.

2. Usability Decomposition: Attributes, Properties and Patterns

Software systems usability is usually evaluated on the finished system trying to assign values to the classical *usability attributes* [3] [4] [5]

- Learnability – how quickly and easily users can begin to do productive work with a system that is new to them, combined with the ease of remembering the way a system must be operated.
- Efficiency of use – the number of tasks per unit time that the user can perform using the system.

[1] STATUS project: EU funded project IST–2001–32298.

- Reliability – sometimes called "reliability in use", this refers to the error rate in using the system and the time it takes to recover from errors.
- Satisfaction – the subjective opinions that users form in using the system.

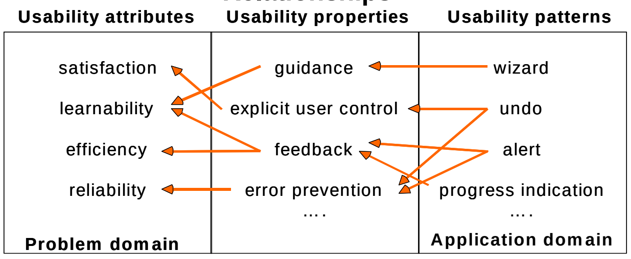
However, the level of these usability attributes is too high for us to be able to examine what mechanisms should be applied to a software architecture to improve these attributes. Therefore, the philosophy followed in STATUS was to decompose these attributes into two intermediate levels of concepts closer to the software solution: usability properties and usability patterns.

The first level involves relating the above-mentioned usability attributes to specific *usability properties* that determine the usability characteristics to be improved in a system. Usability properties can also be seen as the requirements of a software system for it to be usable (for example, provide feedback to the user, provide explicit user control, provide guidance to the user, etc). The second level was envisaged to identify specific mechanisms that might be incorporated into a software architecture to improve the usability of the final system. These mechanisms have been called *usability patterns* and they address some need specified by a usability property. Note that usability patterns do not provide any specific software solution to be incorporated into a software architecture, they just suggest some abstract mechanism that might be used to improve usability (for example, undos, alerts, command aggregations, wizards, etc.).

The procedure followed to identify the relationship between usability attributes, properties and patterns is detailed in Usability Attributes Affected by Software Architecture [6]. We took a top-down approach from usability attributes (identified in the literature), through usability properties (derived from heuristics and guidelines given in the literature to developers for improving usability), to finally identify usability patterns. Accordingly, usability patterns are the final links in the chain, and they provide examples of how to achieve some usability requirements. Nevertheless, they are not the central axis of our approach, which provides the users of the research results with a procedure for developing new usability patterns according to the context of the applications to which the results are applied.

A subset of the above-mentioned relationship is outlined in Table 1. It shows how usability properties relate patterns to usability attributes in a qualitative sense (an arrow indicates that a property positively affects an attribute, that is, improves that attribute). For example, the "wizard" pattern improves learnability: the wizard pattern uses the concept of "guidance" to take the user through a complex task one step at a time; "guidance" improves the learnability usability attribute. Usability patterns may address one or more of the usability properties and usability properties may improve one or more usability attributes.

Table 1. Attribute, Property & Pattern Relationships



The concept of usability pattern has already been used in the literature. This concept can be generally defined as "a description of solutions that improve usability attributes" [7]. The usability aspects dealt with by these patterns refer basically to user interfaces, which is why these patterns are also called user interface patterns. [8] or interaction design patterns [9]. As indicated by authors like Welie and Troetteberg [10], although several pattern collections exist, an accepted set of such patterns has not emerged. There appears to be a lack of consensus about the format and focus of user interface patterns.

Possible examples of some user interface patterns are:

- Feedback
- Wizard
- Provide the user with all information needed in the same window
- Mark required fields when filling a form
- You are here
- Grid Layout.

The differences between the usability patterns proposed in our work and the classic usability or interface patterns existing in the literature lie basically in that the classic patterns of usability are based on the improvement of the application interface, which means that these patterns are implemented mainly during the interface design phase and generally affect low-level components like pseudo-code. On the other hand, the usability patterns in our work relate the mechanisms to be considered in a software architecture, addressing usability aspects in the early stages of the development process. For example, the solution proposed by Welie [10] for the feedback pattern is based on "provide a valid indication of progress. Progress is typically the time remaining until completion, the number of units processed or the percentage of work done. Progress can be shown using a widget such as a progress bar. The progress bar must have a label stating the relative progress or the unit in which is measured". Whereas, as we will see later, we consider a progress indication pattern and provide a solution based on the components to be added to a software architecture and the relationships among these components in order to provide this mechanism.

The second column in Table 2 shows the list of usability patterns that we propose. The first column of the table shows the usability properties related to each pattern.

Table 2. List of usability patterns

Usability Property	**Usability Patterns**
NATURAL MAPPING	
CONSISTENCY (functional, interface, evolutionary)	
ACCESSIBILITY (internationalisation)	Different languages
CONSISTENCY, ACCSESIBILITY (multichannel, disabilities)	Different access methods
FEEDBACK	Alert
ERROR MANAGEMENT, FEEDBACK	Status indication
EXPLICIT USER CONTROL, ADAPTABILITY (user expertise)	Shortcuts (key and tasks)
ERROR MANAGEMENT (error prevention)	Form/field validation
ERROR MANAGEMENT (error correction),	Undo
GUIDANCE, ERROR MANAGEMENT	Context-sensitive help
GUIDANCE, ERROR MANAGEMENT	Wizard
GUIDANCE, ERROR MANAGEMENT	Standard help
GUIDANCE, ERROR MANAGEMENT	Tour
MINIMISE COGNITIVE LOAD, ADAPTABILITY, ERROR MANAGEMENT (error prevention)	Workflow model
ERROR MANAGEMENT (error correction)	History l ogging
GUIDANCE, ERROR MANAGEMENT (error prevention)	Provision of views
ADAPTABILITY (user preferences)	User profile
ERROR MANAGEMENT, EXPLICIT USER CONTROL	Cancel
EXPLICIT USER CONTROL	Multi-tasking
MINIMISE COGNITIVE LOAD ERROR MANAGEMENT (error prevention)	Commands aggregation
EXPLICIT USER CONTROL	Action f or multiple objects
MINIMISE COGNITIVE LOAD, ERROR MANAGEMENT (error prevention)	Reuse information

It should be noted that the properties of Natural Mapping and Consistency cannot be arranged around specific usability patterns. The reason is that these properties require the performance of different tasks and activities throughout the entire development process rather than the application of particular solutions at the architectural level. For example, the provision of natural mapping between the user tasks and the tasks to be implemented in the system calls for software requirements to be elicited during the analysis process bearing in mind this objective and they must be designed according to these requirements. The same goes for consistency, which involves different activities throughout the lengthy development process of the original system or new versions.

At this point, we should refer to the work of Bass, John and Kates [11], who use the concept of usability scenario, where "a scenario describes an interaction that some stakeholder (e.g. user, developer, system administrator) has with the system under consideration from a usability viewpoint". These scenarios are related to some properties and usability patterns considered in our approach. Table 3 shows a comparison of their and our approaches, through the relationships between our patterns and their scenarios. These relationships are:

- Content: achieving a particular usability pattern implies achieving a particular scenario. For example, properly provide the *Provision of views* mechanism implies "Make views accessible".
- Instantiation: a usability pattern is a special case of a scenario. For example, *Standard Help* is a case of "Help".
- Similarity: a pattern and a scenario are considered similarly in both approaches, for example, *Cancel*.
- Generality: a scenario is a special case of a pattern. For example, "Novice interfaces for users in unfamiliar contexts" is a special case of "provide a *Workflow model*".

Some of the scenarios have not been considered in our approach:

- "Account human needs and capabilities when interacting, keep coherence through multiple views, define upgrades similar to previous ones, provide easily modifiable test points for evaluation and design interfaces" are the results of specific actions to be taken during the development process and are not in keeping with the definition of usability pattern considered in our work. So, the issues referred to by these scenarios need to be dealt with within the whole development process, not specifically in terms of architecture. The STATUS project has a workpackage that deals with modifications in the development process to improve final system usability.
- "Minimize user recovery work due to system errors" refers to errors made by the software system and not by the users. Traditionally [3][4], usability efficiency deals with the prevention of and recovery from user, not system, errors.
- "Allow searching by different criteria" and "Provide alternative secure mechanisms" are specific requirements and not really usability patterns as they are considered in our work.

Table 3. Usability patterns / scenario relationship

Usability Patterns	Relationship	Scenarios
Different languages	similarity	Support international use
Different access methods	generality	Maintain device independence
Alert	generality	Verify resources before beginning an operation
Status indication	similarity generality	Present system state Predicting task duration
Shortcuts (key and tasks)		
Form/field validation	similarity	Checking for errors
Undo	similarity	Undo
Context-sensitive help	instantiation	Provide good help
Wizard	instantiation	Provide good help
Standard help	instantiation	Provide good help
Tour	instantiation	Provide good help
Workflow model	generality	Novice interfaces for user in unfamiliar contexts
History logging		
Provision of views	content similarity content	Make views accessible Provide reasonable set of views Quick navigation into a view
User profile		
Cancel	similarity	Cancel
Multi-tasking	content content	Use applications concurrently Allow to quick switch back and forth between different tasks
Commands aggregation	similarity	Aggregate commands
Action for multiple objects	similarity	Aggregate data
Reuse information	similarity	Reusing information

It might be interesting to note the similarities and differences between the two approaches, for example, the generality or specificity levels of the usability mechanisms employed by the two approaches. In this respect, it would be worthwhile discussing at the workshop the strengths and weaknesses of using usability mechanisms with differing detail levels from the viewpoint of practitioners. Note, however, that both approaches will agree with the idea of relating the different aspects of usability to the architecture through architectural patterns. These patterns will show how the scenario (Bass et al.'s approach) or the usability pattern (STATUS approach) can be represented at an architectural level.

The following section shows how we have developed design solutions that can be used to incorporate the usability mechanisms specified by the usability patterns into a software system. These design solutions are the architectural patterns mentioned above.

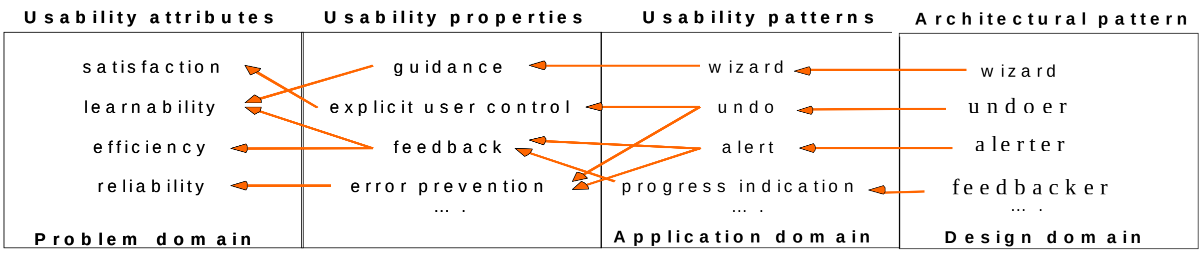
3. Architectural Patterns for Usability Support

The most widely used concept of pattern in software development is the design pattern, and it is used particularly in the object-oriented paradigm. In this context, a design pattern is a description of classes and objects that work together to solve a particular problem [11]. These patterns show a solution to a problem, which has been obtained from its use in different applications. Note, nevertheless, that a design pattern can be seen as a unique or original solution.

Besides the idea of usability pattern, we also used the concept of architectural pattern. Given that we have defined a usability pattern as a mechanism to be applied to the design of a system architecture in order to address a particular usability property, an architectural pattern will determine how this usability pattern is converted into software architecture. In other words, what effect the consideration of a usability pattern will have on the components of the software architecture. Abstracting the definition of design pattern, an architectural pattern can be defined as a description of the components of a design and the communication between these components to provide a solution for a usability pattern. Like design patterns, architectural patterns will reflect a possible solution to a problem, the implementation of a usability pattern, although this will be a unique solution in each case.

Therefore, the architectural pattern is the last chain in the usability attribute, property and pattern chain that connects software system usability with software system architecture. Accordingly, another column can be added to Table 1, as shown in Table 4.

Table 4. Usability attributes/properties/pattern and architectural pattern relationships



3.1. Procedure for outputting architectural patterns for usability

In the following, we describe the procedure followed to identify the architectural patterns that design the proposed usability patterns. This procedure is composed of two parts:

1. Application of a process of induction to abstract the architectural patterns from particular designs for several projects developed by both researchers and practitioners. For this purpose, we took the following steps:
   1.1. We asked designers to build the design models for several systems without including usability patterns.
   1.2. For each usability pattern, we asked designers to modify their earlier designs to include the functionality corresponding to the pattern under consideration.
   1.3. For each usability pattern, we abstracted the respective architectural pattern from the modifications made by the developers to the design.

   This process was carried out on two applications: restaurant orders and tables management and ride control and maintenance at an amusement park.
2. Application of the architectural patterns resulting from the previous step to several developments to validate their feasibility.

To illustrate this process of induction, below we show part of the induction of one of the architectural patterns on the restaurant orders and table management application, specifically the pattern related to the usability pattern *Progress Indication*.

The sequence diagram shown in Figure 1 and the class diagram shown in Figure 2 show part of the design of this application, specifically the part related to the entry of the menu requested by the restaurant customer. This part was designed without taking into account the usability property on feedback. As we can see from the diagrams, the system user is not receiving any information on what the software system is doing. Figure 3 and Figure 4 show the sequence and class diagrams, respectively, now considering the inclusion of the usability pattern for *Status Indication* on this same functionality. We can see how the inclusion of an object of the Feedbacker class provides the user with information on system operation.

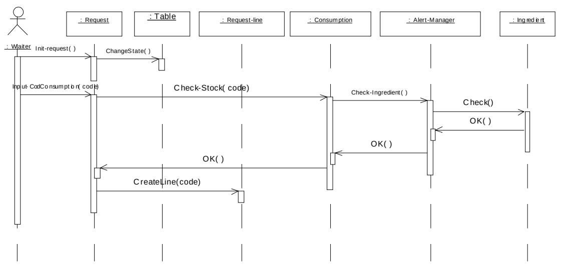


Figure 1. Interaction diagram without usability pattern

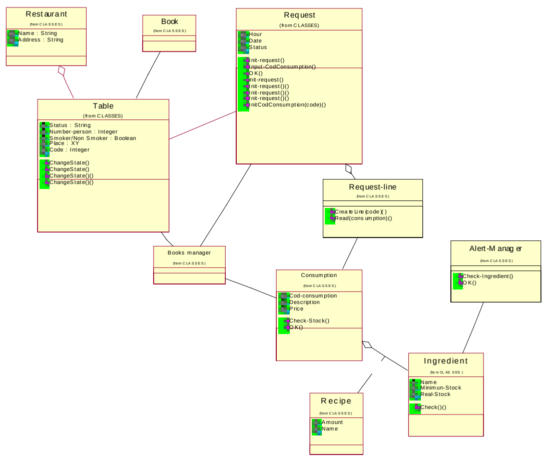


Figure 2. Class diagram without usability pattern

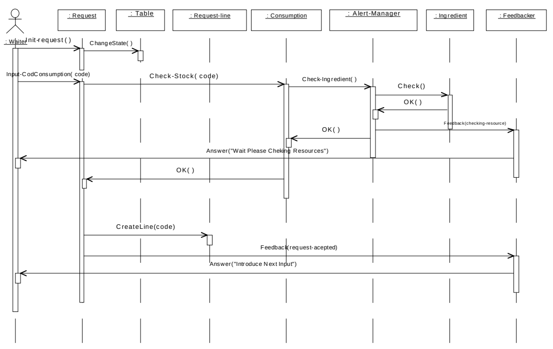


Figure 3. Interaction diagram with usability pattern

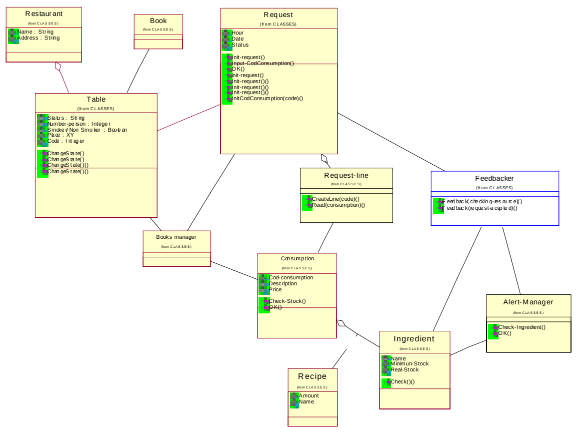


Figure 4. Class diagram with usability pattern

From this design and others for the same system and the other application created by other developers, we have abstracted a general design solution as shown in Figure 5.

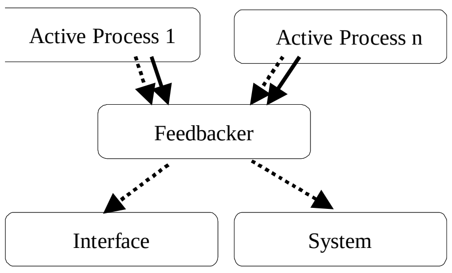


Figure 5. Generic solution for the Feedbacker pattern

Likewise, we have applied this inductive process to the other usability patterns to develop the respective architectural patterns. For details of this process, see Techniques and Patterns for Architecture-Level Usability Improvements [13]. Table 5 summarises the architectural patterns defined together with their underlying usability patterns.

Table 5. List of usability and architectural patterns

Usability Patterns	**Architectural Pattern**
Different languages	Language-recogniser
Different access methods	Device- recogniser
Alert	Alerter
Status Indication	Feedbacker
Shortcuts (key and tasks)	Shortcutter
Form/field validation	Checker
Undo	Undoer
Context-sensitive help	Sensitive-helper
Standard help	Standard helper
Tour	Guided helper
Workflow model	Filter
History logging	Logger
Provision of views	Viewer
User profile	Profiler
Cancel	Canceler
Multi-tasking	Dispatcher
Commands aggregation	Aggregator
Action for multiple objects	Multi-executer
Reuse information	Reuser

3.2.Description of the Architectural Patterns

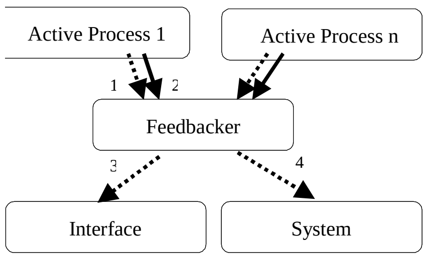
Since the ultimate aim of this work is to provide a set of architectural recommendations to improve the usability of the software systems, these recommendations will be described in an architectural pattern catalogue. Each pattern in this catalogue has to be described according to following elements:

- **Pattern Name** - Patterns must have suggestive names that give an idea of the problem they address and the solution in a word or two.
- **Problem** – This describes when to apply the pattern and in which context. In the case of architectural patterns, the problem will refer to a specific usability pattern to be materialised.
- **Solution** – This describes the elements that make up the architecture, their relationships, responsibilities, etc. The solution does not describe a definite design, as a pattern can be seen as a template that can be applied in many different situations. Particularly, the solution for a specific pattern will be specified from :
  - **Graphical representation** - A figure that represents the components of the architecture and their iterations. Numbered arrows between the different components will represent the iterations. The arrows with solid lines specify the data flow, while the dotted lines represent the control flow between the components.
  - **Participants** – A description of the components that take part in the proposed solution and the iterations (represented by arrows) to determine how they are to assume their responsibilities.
- **Usability benefits** - Description of which usability aspects (usability properties) can be improved by including the right pattern.
- **Usability rationale** - A reasonable argumentation for the impact of pattern application on usability, that is, what usability attributes have been improved, and which ones may get worse. Initially, this feature will be completed with information coming from others authors or from the experience of the consortium members. However, once the patterns have been applied to real applications, this field will be filled in with empirical experience.
- **Consequences** - Impact of the pattern on other quality attributes, like flexibility, portability, maintainability, etc. As for the above feature, this one will be filled in with the results of empirical experience.

- **Related patterns** - Which architectural patterns are closely related to this one, and what differences there are.
- **Implementation of the pattern in OO** - The architectural patterns provided are patterns that can be applied in any development paradigm. However, as these patterns have been obtained and refined for OO applications, we will provide guides tending to address pattern application in this field. Basically, we will describe the classes deriving from the pattern's main components. These guides are illustrated in the example shown in the following section.
- **Example** of the application of the pattern in question.

In the following, we show how the architectural pattern "Feedbacker" is described:

- Pattern Name: Feedbacker
- Problem: The user should be provided with information pertaining to the current state of the system.
- Solution:
  - Graphical representation:



  - Participants:
    - Active-process i: this module has been represented more than once, because there may be several processes running simultaneously that request feedback (1), and it will be each active process that sends the information that it wants to be fed back (1) to the Feedbacker.
    - Feedbacker: this module is responsible for receiving the request and data (1) (2) that indicate the type of feedback requested and the data to be fed back from each active process. Additionally, it must know the recipient of this feedback and will send this feedback either to another part of the system (4) and/or to the interface (3) to inform the user. [10] specifies some guidelines on how to display this feedback on the user interface, for example, how often it should be refreshed or where the particular information should be placed. These details should be taken into account during low-level design.
    - Interface: the interface is responsible for receiving the feedback and displaying it to the user (3).
    - System: this component is optional and represents other parts of the system that should be informed of the feedback (4).
- Usability benefits: giving an indication of the system's status provides feedback to the user about what the system is currently doing, and what the result of any action they take will be.
- Usability rationale: providing feedback gives the user information about what the system is working on and whether the application is still processing or has died. So, this pattern raises *satisfaction*.
- Consequences:
  - This pattern prevents additional *system load* by avoiding retries from users [10].
  - This pattern improves system *maintainability* because it channels the feedback information better as compared with when the feedback exists but is indiscriminately emitted by any other system module.
- Related patterns:
- OO implementation: This architectural pattern will give rise to a Feedbacker class specialised in informing the user and the system of what is going on. This means that all the classes that want to report something to the system must report to the feedback manager, Feedbacker, so that this manager can properly distribute this information either inside or outside the system.
- Example: This section would detail one of the examples used to get this pattern, for example, the example shown in Figure 4 and Figure 5.

4. Future Work

Now that we have satisfactory solutions for the architectural patterns, the next step is to apply these to different designs. The aim is to check the feasibility of the solution provided by each pattern and derive and refine recommendations for its application by practitioners. This task is part two of the procedure for outputting architectural patterns described in section 3.2.

After generating the architectural patterns we propose to present a set of practical guides that provide practitioners with information on:

- How to select an architectural pattern, for example, from the usability attributes that are to be enhanced in each design and the impact on the other quality attributes.

- How to use an architectural pattern for inclusion in a given design.

The effort to improve software architecture with regard to usability presented in this paper is related to another important part of STATUS, which is the assessment of this architecture with respect to usability. This assessment is being conducted in two ways in the project: a scenario-based architectural assessment and a simulation-based architectural assessment (two papers addressing this research have also been submitted to this ICSE workshop). This evaluation will yield the set of shortcomings that a given software architecture has with respect to certain usability attributes or parameters. The architectural patterns could, therefore, be used to implement usability improvement solutions for the detected shortcomings.

However, the idea of architectural patterns can also be used independently of the architecture evaluation, as they provide design solutions for certain usability requirements (any *usability properties* included in the requirements specification). The consideration of these usability requirements at the start of development and later in design, by means of architectural patterns, is expected to provide improvements in final system usability.

We have to take into account that the final software system usability has to be validated and measured when the system in question has been built and is operational. Therefore, we will have to wait until these results have been applied to real projects to get empirical data to properly verify the as yet intuitive benefits that the use of architectural patterns can provide for software systems usability. Half of the STATUS project time has been allocated to validating the ideas of this paper and the remainder of the research with the industrial partners. This validation will kick off in March 2003 and run until June 2004.

5. References

[1]. J Bosch. Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach, Pearson Education, Addison-Wesley, 2000.

[2] X. Ferré, N. Juristo, H. Windl, L. Constantine. Usability Basics for Software Developers. *IEEE Software*, vol 18 (11), p. 22-30

[3] L. L. Constantine, L. A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Addison-Wesley, New York, NY, 1999

[4] J. Nielsen. *Usability Engineering*. AP Professional, 1993.

[5] B. Shackel. "Usability – context, framework, design and evaluation". In *Human Factors for Informatics Usability*. pp 21-38. Ed. by B. Shackel and S. Richardson. Cambridge University Press, 1991.

[6] A. Andrés, J. Bosch, A Charalampos, R. Chatley, X. Ferre, E. Forlmer, N. Juristo, J. Magee, S. Menegos, A. Moreno. *Usability attributes affected by software architecture*. Deliverable 2. STATUS project, June 2002. Http://www.ls.fi.upm.es/status

[7] Perzel,, D Kane D. (1999) *Usability Patterns for Applications o the World Wide Web.* PloP'99

[8] G Cascade. *Notes on a Pattern Language for Interactive Usability,* Proceedings of the Computer Human Interface Conference of the ACM, Atlanta, Georgia, 1997.

[9] J Tidwell. *Interaction Design Patterns.* Pattern Languages of Programming 1998, Washington University Technical Report TR 98-25.

[10] M. Welie, H Troetteberg. *Interaction Patterns in User Interfaces*. PloP'00.

[11] E Gamma, R Helm, R Johnson, J Glissades. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1998.

[12] L Bass, E Bonie, J Kates. *Achieving Usability Through Software Architecture.* Technical Report. CMU/SEI-2001-TR-005, March 2001.

[13] N. Juristo, M. López, A. Moreno, M Sánchez. *Techniques and Patterns for Architecture-Level Usability Improvements*. Deliverable 3.4. STATUS project. Http://www.ls.fi.upm.es/status.

Software Engineering and HCI Techniques Joined to Develop Virtual Environments

María-Isabel Sánchez-Segura
Antonio de Amescua Seco
Juan-José Cuadrado
Escuela Politécnica Superior
Universidad Carlos III de Madrid
Avda. Universidad, 30, Leganes 28911,
Madrid, Spain
(+34) 91 624 91 04
{misanche; amescua; cuadrado}@inf.uc3m.es
Angélica de Antonio
Facultad de Informática
Universidad Politécnica de Madrid
Campus de Montegancedo s/n
28660 Boadilla del Monte, Madrid, Spain
(+34) 91 336 69 25
angelica@fi.upm.es

1. Abstract

This paper focuses on a specific kind of software applications called Virtual Environments (VEs) or Virtual Worlds, Virtual Communities, etc., inhabited by avatars, which are the representation either of real humans users of the VE or autonomous agents. Virtual Environments (VEs), or Virtual World ,are growing very quickly and people demand easier and more believable ways to interact in these new sites [4],[10], [12].

From the point of view of software engineering, VEs can be seen as a special kind of software systems, in this sense, they must be analyzed, designed, implemented, etc., as any other software system. However, VEs development require special tasks and techniques, which are not provided by traditional software engineering methodologies. The question we would like to address is: define the products to be developed during the construction of an VE taking into account other disciplines like Human Computer Interaction, Artificial Intelligence, etc., to enhance the quality of the development process.

1.1. Keywords

Virtual Environment, Software Engineering, Development Process, HCI techniques.

2. Introduction

Today, virtual environments are being used in many fields: Social Worlds, Finance, Commerce, Banking, Information System Sciences, Communication, CSCW (Computer Supported Collaborative Worlds), Education, Entertainment and Leisure, Medicine, Architecture, Geography, etc.[3]. This kind of application also seems to be the future of interactive programming [1] and can be used especially to demonstrate risky situations.

We are going to focus on the most recent VEs, based on 3D graphics and inhabited by Avatars[1] and autonomous agents.

Nowadays, the implementation process of VEs is well known but informal. In fact, good and useful results can sometimes be achieved with a modest outlay of hardware and resources. The problem comes from the very expensive construction [14], derived from following an informal process.

Therefore, the need for a more formal process is evident. From the Software Engineering perspective, a set of processes to formalize the development of this kind of application can be defined.

The Software Engineering Research Community is not the only one interested in the development of VEs. The need to define new techniques inspired by the Software Engineering discipline, is widely-known by scientists body related with HCI (Human Computer Interaction). [2].

Current software engineering process models [6] [7], must be enhanced allowing the use of different techniques coming from disciplines such as HCI or Artificial Intelligence to develop systems like VEs where the core of the development process will be provided by software engineering discipline. Specific techniques for the interface design will be provided by HCI discipline and Artificial Intelligence provides the appropriate techniques to design and implement the "knowledge" of the system because in this kind of applications the representation of the users into the VE must be as credible as possible.

3. Processes to be enhanced

In this section, we will present general and specific features of VEs-based applications with the aim of finding out why some processes must be used and what features cause the processes selected to be redefined.

Taking as software engineering pillars the process models [6] [7], Figure 1 shows the set of processes selected to be redefined or just enhanced. Those not selected could be used without any change because they do not depend greatly on VEs development

[1] Avatar comes from Sanskrit and means incarnation.

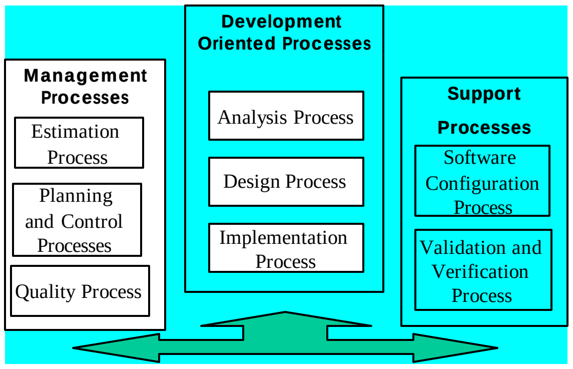

Management Processes
Estimation Process
Planning and Control Processes
Quality Process
Development Oriented Processes
Analysis Process
Design Process
Implementation Process
Support Processes
Software Configuration Process
Validation and Verification Process

Figure 1 Processes Selected

In this paper we are just going to focus on Analysis and Design processes, the methodology which covers all the development processes was defined and was coined SENDA, the methodology details can be found in [13].

Up to now we have been talking about Process Models. The next question could be: *What about the development methodology?* To date, there are many methodologies whose goals are the disciplined development of software systems supported by Software Engineering paradigms. Those best-suited to VEs development are Object Oriented methodologies. L.Casey [11] said that the use of Object Oriented Techniques could be the key to a real advance in VEs development based on Software Engineering development.

In our experience, however, traditional Object Oriented methodologies are not enough for VEs development so, traditional methodologies must be enriched with a set of new tasks mainly in the analysis and design processes.

In this paper, we are going to focus on Analysis and Design processes due to the extension of this paper, with a short description of each process and their interrelationship, as well as the tasks where HCI and Artificial intelligence techniques could be useful. The symbol notation used can be found in [9], and the next notation has been used to name tasks "*Process Acronym plus order-number*". Possible process acronyms are:

- ❖ A: Analysis Process
- ❖ 3DD: 3D Design Process
- ❖ SD: System Design Process
- ❖ AD: Actions Design Process

Order-Number, in tasks name, does not mean the order in which the tasks must be completed. It is only used to name the tasks, without any significant notation, except for the process acronym.

4. VE Analysis Process

IEEE Requirements Standard [5] will be used for some tasks in this process. Object Oriented techniques [8] will be used for the Static and Dynamic Modeling. Conceptualization task need special treatment because there are a set of requirements such as auto-triggered behaviors, collision detection, etc., which cannot be defined with either Structured or Object Oriented Techniques. We have defined specific techniques to deal with this task.

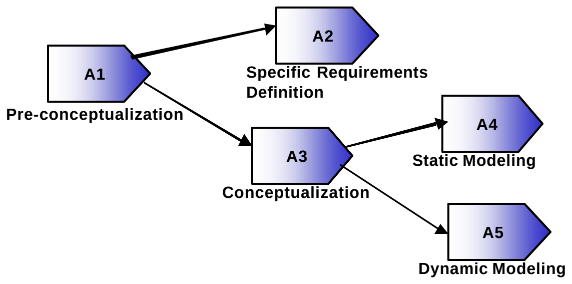


Figure 2 Analysis Process Tasks

The relationship between identified tasks, Figure 2, is defined through the output and input artifacts flow.

During the *"Pre-conceptualization"* task, the analyst, with the client's assistance, must elaborate a "List of Initial Requirements" and the "Problem Definition". The *"Conceptualization"* task takes these products in order to obtain the "Conceptualization document" that contains "Use Cases" and "Use Concepts". We propose a new term **"Use Concept"**, not yet defined. Each Use Concept is defined by a brief description of the functionality, which will not be demanded directly by the user, and the following three fields:

- ***Purpose***: Use Concept main goal.
- ***Working Mode***: the way the Use Concept is going to be used.
- ***Dynamic***: the Use frequency.

Conceptualization can be easily enhanced using HCI techniques, methods such as GOMS (Goals, Operators, Methods ans Selection Rules) which is useful to model an activity as it is understood by the user.

5. VE Design processes

The traditionally known "Design Process", has been subdivided into three processes, due to the features of VEs developments.

- ***3D Design Process***: includes the graphic design of scenarios, decorative objects, avatars, etc. The products obtained from this process can be seen as requirements to explain "how" the environment and its elements are. This is why they are defined under *3D Design Process.*
- ***Actions Design Process***: in VEs development special attention must be dedicated to give credibility to avatars and the rest of the elements. So a special set of tasks must be defined in this sense.
- ***System Design Process***: this process corresponds with the traditional and well known Design process in software engineering methodologies.

5.1. Design Processes Roles Involved

In design processes, two kinds of roles are involved.

System Designer: typically assigned to define "how" is the application. By this, we mean the person who defines the control of the system following the System Analyst's definition of "What". In VEs, the System Designer is also the person who guides the Graphic Designer because of his/her knowledge of the application to be developed. The System Designer must also have basic knowledge in graphic design.

Graphic Designer: his job in the Design Process is feedback, View Maps, Environment Modeling Forms and Avatar Modeling [2]Forms for the System Designer. After the feedback stage, Graphic Designers can begin the task of implementing 3D objects in the Graphic Design Tool selected, although this task is included in *Implementation Process*. It is important to remember that Graphic Designers are artists and do not need to have any computer knowledge.

5.2. 3D Design Process

We have fully defined this process. Many times, the problem Graphic Designer has during the implementation process, is the absence of guidelines on how to develop his task.

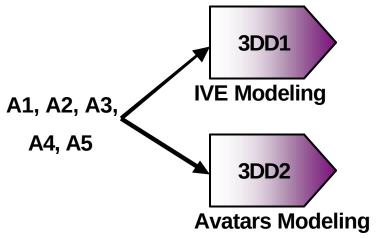


Figure 3. 3D Design Process Tasks

2 See 5.2.

The *System Designer*, which is the person in charge of these tasks, must have a good knowledge of the physical IVE features and must also have a basic knowledge of graphic design. The result of these tasks will be a set of *forms and View Maps*, for the Graphic Designer to be used during the Implementation Process, to build three-dimensional models with the selected tools.

Both, "IVE Modeling" and *"Avatars Modeling"* tasks, take output product from Analysis Process Tasks. The former includes the detailed definition of a set of virtual spaces and the objects to be included, and the latter, the detailed definition of the virtual inhabitants, their appearance and physical structure. The products of these tasks are:

- ***View Maps***
- ***Environment Modeling Forms.***
- ***Avatar Modeling Forms.***
- ***Avatars Hierarchy Model.***

In a larger version of this paper, more detailed information is given on these tasks and techniques.

5.3. Actions Design Process

In VEs development, as in any kind of interactive development, "behaviour definition" is very important. VEs must be used to make interaction among users, through avatars, easier and believable.

The aim of this process is the definition of the actions that can take place within the VE. Many people will be involved in this process: psychologists, sociologists, etc., because a multi-disciplinary work is necessary to provide this kind of application with sufficiently interesting interactive features to give credibility to avatars and the rest of VE elements.

It is very important to emphasize the relation between "Actions Design" and "3D Design" processes. They must be coherent.

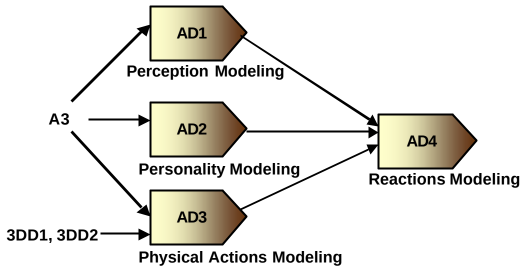


Figure 4. Actions Design Process Tasks

"Conceptualization" task A3, returns every action to be done by avatars and the rest of the elements into the VE. These actions include the detection of the events, which occurs in the environments, how avatars feel these events, and how these feelings are shown through the physical representation of elements.

These are all defined by the set of tasks included in this process. Techniques, like rules, etc, taken from Artificial Intelligent Field, must be used in this process. Also Genetic Algorithms are very useful for the avatars to be able to learn into the VE along its execution.

5.4. System Design Process

As we have mentioned before, the tasks included in this process are well defined in most Object Oriented methodologies.

"Expanded Static Modeling" (SD1) and *"Expanded Dynamic Modeling" (SD2)* tasks take:

- Class Diagram from *"Static Modeling" (A4)* and Transition Diagrams and Event Traces from *"Dynamic Modeling" (A5)* task (both from Analysis Process).
- *"Physical Actions Modeling"* task (AD3) to create new classes and methods derived from the exact definition of movements, etc., which is the output of AD3.

"System Architecture Design" (SD4) task and *"Data Persistence Design" (SD5)* tasks take the "Specific Requirements Document", returned from "A2", where restrictions or details to be used in SD4 and SD5 can be found. *"Interface Design"* task can be widely enhanced using HCI techniques because the interface use to be a very important part of the VE.

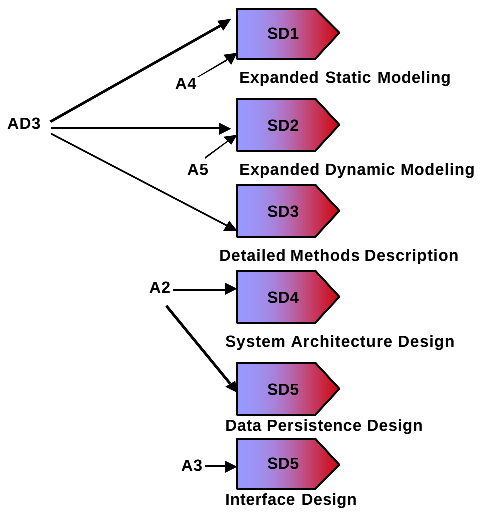


Figure 5 System Design Process Tasks

6. Experimental Results

The process model presented has been used to develop the Run&Freeze environment, funded by the Amusement ESPRIT IV project 25197.

Run&Freeze is a children´s game. We have selected it because the rules are very simple but rich in features such as social interaction, mobility of the participants, immersion, spatial sense, etc.

The VE shown in Figure 6 and Figure 7 was developed from the process model SENDA described above.

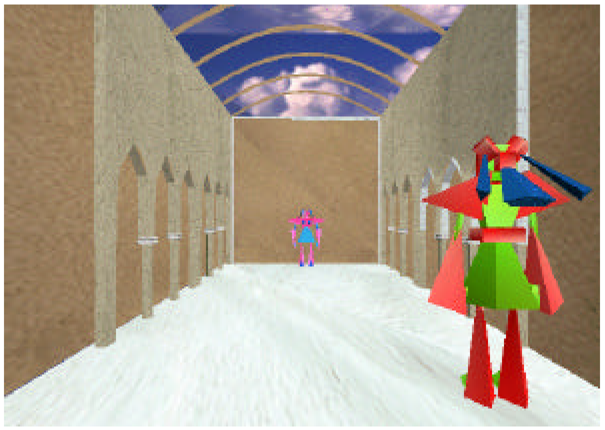


Figure 6 Run&Freeze environment with two avatars

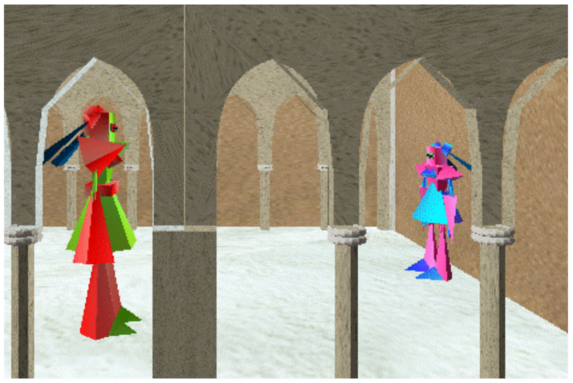


Figure 7 Run&Freeze environment lateral view

7. Conclusions and future trends

From the results obtained, it must be noted that the processes and techniques proposed as well as the combination among HCI and Artificial Intelligence techniques are powerful and flexible enough to allow for the creation of different VEs respecting the constraints of the application (to run in real time, etc.)

HCI techniques has been very useful in the design process to a better understanding of the problems the users could have with the use of the VE.

The proposed techniques have also useful to improve the communication among people with different backgrounds.

As we have already mentioned, there are many different processes to be enriched; the improvement of management processes, refining the estimation and the planning processes also using HCI knowledge to enhance estimation methods, because this discipline can provide information about the cost and effort in the interface design which can be used to refine the drivers in real estimation methods.

8. Acknowledgements

This research is partially supported by the Amusement Project (ESPRIT IV LTR project 25197)

9. References

1. Berenguer, X., (1997). Writing Interactive Programs. Magazine Formats.
2. Brown, J. Exploring human-computer interaction, and software engineering methodologies for the creation of interactive software. SIGCHI Bulletin (ACM), Vol. 29, Num 1, 32-35. 1997

3. CALT: The Center For Advanced Learning Technologies. INSEAD -Bd de Constance - F-77305 Fontainebleau Cedex, France. (2000). Available at: http://www.insead.fr/CALT/Encyclopedia/Computer Sciences/VR/vr.htm
4. Damer, B. Interacting and Designing in Virtual Worlds on the Internet. Tutorial for CHI97.
5. IEEE Std 830-1998. (1998) IEEE Recommended Practice for Software Requirements Specification Institute of Electrical and Electronics Engineers
6. IEEE Std. 1074-1991. (1991). IEEE, Standard for Developing Software Life Cy cle Processes. New York (EE.UU.), IEEE Computer Society.
7. ISO/IEC Standard 12207:1995. (1995). Software Life Cycle Processes. Ginebra (Suiza), International Organization for Standarization.
8. Jacobson, I., et al. (1992) Object Oriented Software Engineering: A Use Case Driven Approach. Reading, MA: Addison-Wesley.
9. Kruchten, P. (1999). The Rational Unified Process. An Introduction. Addison-Wesley Object Technology Series.
10. Landauer C., Bellman K. Integration and Modelling in MUVEs. Proceedings of the Virtual Worlds and Simulation Conference. VMSIM' 98. Society for Computer Simulation International. San Diego, USA, pp 187-192. 1998
11. Larijani, L.C. Realidad Virtual. McGrawHill, 1994.
12. McKay, D.P. Wath' s in a virtual world?. Proceedings of the virtual worlds and simulation conference. VSIM' 98. Society for computer simulation, San Diego, USA. P-9. 1998
13. Sánchez-Segura, M.I. Aproximación Metodológica al Desarrollo de Entornos Virtuales Ph. D. Thesis. Technical University of Madrid. Spain. (2001)
14. Venus: Virtual Environments For You. (1999) Available at: http://leonardo.ucs.ed.ac.uk/venus/foryou/foryou.html

Integration of Usability Techniques into the Software Development Process

Xavier Ferre
Universidad Politecnica de Madrid
xavier@fi.upm.es

Abstract

Software development organisations are paying more and more attention to the usability of their software products. To raise the usability level of the software product, it is necessary to employ usability techniques, but their use is far from straightforward since they are not, in most cases, integrated with the software engineering development processes. Offering average software developers a way to integrate usability activities and techniques into their existing software development process can bridge the gap between usability and software engineering practice. The only requirement is for the existing process to be based on iterative refinement. We present a handy grouping of usability techniques as increments that developers can introduce into their software development process. We have arrived at this result by surveying the usability literature, adapting usability concepts to software engineering terminology, and examining the development time constraints on the application of usability activities and techniques.

1. Introduction

Usability is not addressed in software development as often as would be necessary to output highly usable software. It is properly addressed only in projects where there is an explicit interest in usability, and the quality of the system-user interaction is perceived as critical by the software development organisation. In this kind of projects, usability experts drive the development, using mostly usability-related techniques in the phases previous to coding. The processes followed and the techniques applied come from the HCI (Human-Computer Interaction) field, and they are not defined so as to be understandable in software engineering, so they cannot be employed "as they are" by average developers.

Larman states that there is probably no other kind of software development technique with a greater mismatch between its importance for the success of software development and the lack of rigorous attention and formal education than usability engineering and the design of the user interface [1]. Due to the increasing perception of usability as strategic for software development businesses, an increasing number of software development organisations are pursuing the aim of integrating usability practices into their software engineering processes, but it is not an easy endeavour [2]. Some proposals for integration ([3], [4]) present ad-hoc solutions that have been created for particular software development organisations. However, their approach is not general enough for them to be applied by other organisations.

One of the virtues of the HCI field lies in its multidisciplinary essence. This characteristic is, at the same time, the main obstacle to its integration with software engineering: while the HCI foundations come from the disciplines of psychology, sociology, industrial design, graphic design, and so forth; software engineers take a very different view, a typical engineering approach. Both fields speak a different language and they deal with software development from a different perspective [5]. We have tried to approach the integration of usability activities and techniques in a general software development process by employing the concepts and terminology that average software developers use, that is, software engineering terminology and concepts.

For this purpose, we first surveyed the usability literature to identify the characteristics that define a user-centred process and choose the usability techniques and activities best suited for inclusion in a software process. Then, we mapped all the findings in the usability field to the respective development activities, as expressed in software engineering. And, finally, we distilled the findings as process increments or deltas, which group similar kinds of usability techniques that are meant to be applied close together in terms of development time.

Each organisation can evaluate whether the type of process it has in place meets the minimum requirements for the incorporation of usability-oriented techniques and activities and, if the requirements are met, can add all or some of the process increments containing usability techniques to the existing software development process.

2. Characteristics of a User-Centred Software Development Process

We have studied the HCI literature to identify the characteristics that a software development process should have for it to be considered user-centred. [6], [7], [8], [9], [10] and [11] agree on considering iterative development as a must for a user-centred development process. The complexity of the human side in human-computer interaction makes it almost impossible to create a correct design at the first go. Cognitive, sociological, educational, physical and emotional issues may play an important role in any user-system interaction, and an iterative approach is the most sensible way to deal with these issues.

The other two characteristics that are mentioned by several sources are: active user involvement; and a proper understanding of user and task requirements. These two conditions can be met by introducing usability techniques that can help software developers with the integration of users into the design process and with the enhancement of requirements activities with specific usability aspects. On the contrary, the first condition (that is, to be based on iterative refinement) is an intrinsic characteristic of the software process, and it will be the only requirement for an existing development process to be a candidate for the introduction of usability techniques and activities.

When trying to output the activities that form part of a user-centred process, we found that the HCI field offers a heterogeneous landscape of methods and philosophies, like, for instance, usability engineering, usage-centred design, contextual inquiry, and participatory design. Each author attaches importance to a few techniques, and the terminology may vary from one author to another. For this reason, we first surveyed the HCI literature to identify the most agreed upon usability activities that should be part of the software development process. We have listed the usability activities in Figure 1, grouped according to the kind of development activity to which they belong.

We then surveyed usability techniques, which we allocated to the set of activities obtained previously. There are a host of usability techniques (we identified 82 techniques in our survey), some of which are just small variants of another technique. We have weighted the utility of each particular technique, trying to remove techniques whose objective can be attained by another technique from our selection. Where there were several candidate techniques, we considered the following criteria for our choice: how alien the technique is to software engineering, its general applicability, the cost of application, and its general acceptance in the HCI field. And we preferred to choose techniques that were less alien to software engineering, less costly to apply, and most generally accepted.

We have output a set of candidate usability techniques for inclusion in the software development process, where there is at least one technique to cover each usability activity and subactivity that can lead to an improvement in the usability level of the final software product. The set of candidate usability techniques selected amounts to a total of 51 techniques, which we are not detailing here for reasons of space.

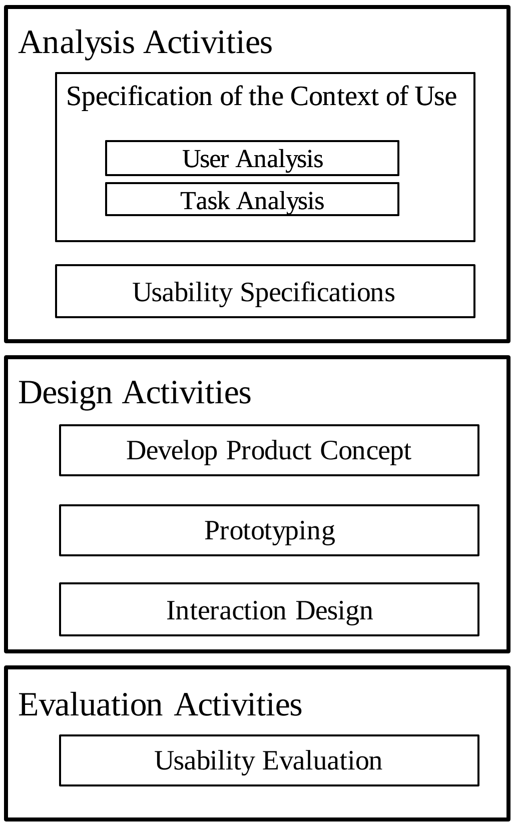


Figure 1. Usability activities grouped according to the generic type of development activity

3. Adaptation of Usability Activities to Software Engineering Development Process Concepts and Terminology

For the set of usability techniques we have chosen to be used by software developers, they need to be matched to the development process. Therefore, we need to adapt the results of our usability techniques and activities survey to software engineering concepts and terminology. Wherever possible, the SWEBOK [12] has been used as a basis for defining the activities in a traditional software development process.

Figure 2 shows the mapping between the usability activities from the usability literature (on the left-hand side) and a generic development process (on the right-hand side of the figure).

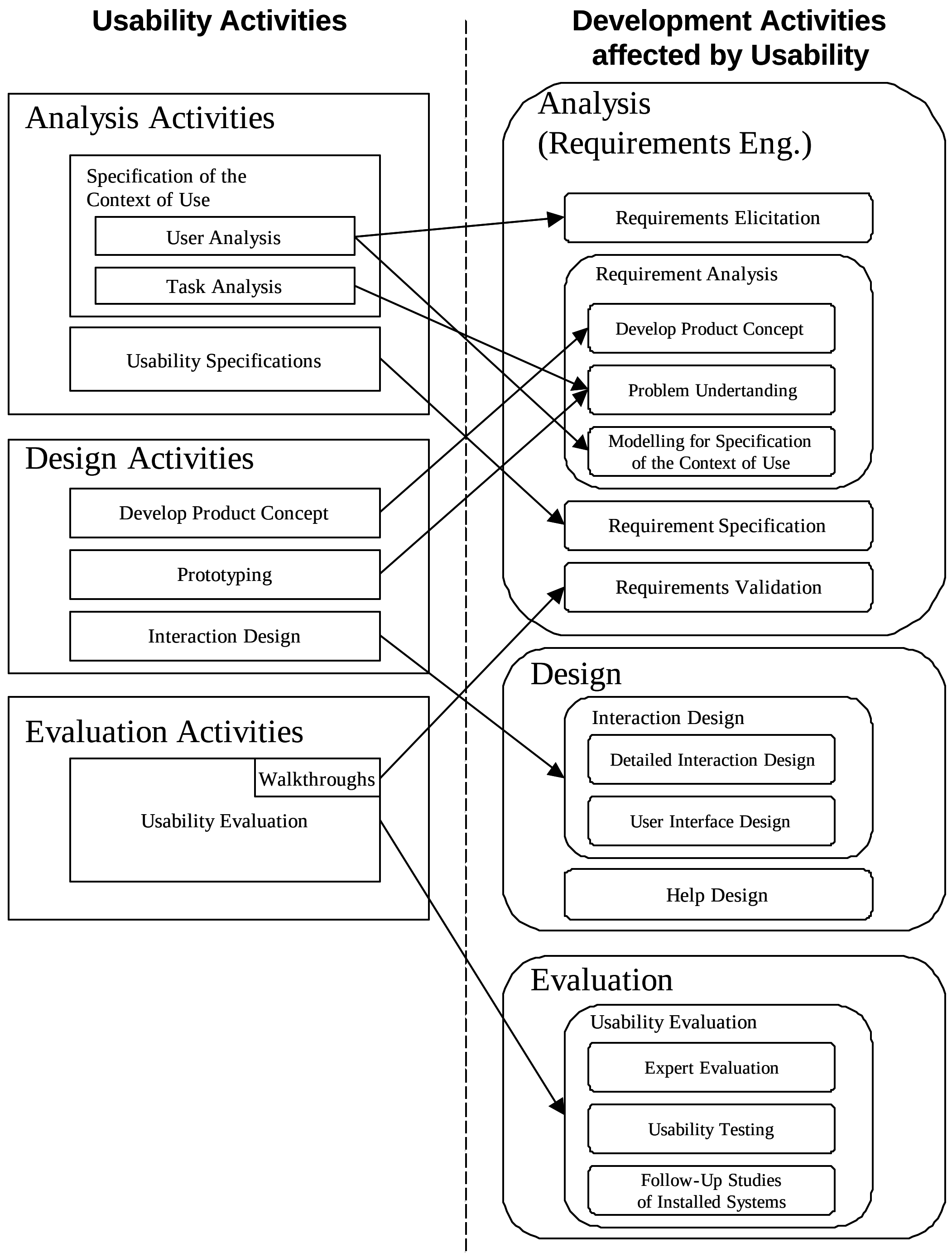


Figure 2. Mapping of usability activities to general development activities

Regarding analysis, we have taken the SWEBOK as a source for detailing the analysis activities. The relevant requirements engineering activities for our purpose are: requirements elicitation, requirement analysis, requirement specification and requirements validation. By relevant we mean that there are usability activities that interlink with development activities, so they can be mapped to certain software engineering analysis efforts. We have allocated two usability design activities to analysis, because of their close connection to requirements activities. They were considered as design activities in our survey, because they appear as such in usability literature. However, Prototyping is considered in software engineering as a technique that can be used for problem understanding, while Develop the Product Concept is the kind of design known as innovation design, and it is usually performed as part of requirements engineering efforts. The SWEBOK considers innovation design not as part of the software design activity, but as part of the requirements analysis activity. Therefore, we have included Prototyping and Develop the Product Concept as part of Requirement Analysis. Walkthroughs are a kind of evaluation activity that can be performed on analysis products, so it has been highlighted within Usability Evaluation in Figure 2 to be able to show its link with Requirements Validation.

On the other hand, we have not used the SWEBOK as a basis for design and evaluation activities. Unlike Analysis, we have usability activities in Design and Evaluation that are relatively independent of other design or evaluation development activities, and the structure for design and evaluation activities described in the SWEBOK is not suitable for usability activities.

We have divided design into Interaction Design and Help Design. The activity of Interaction Design is decomposed into Detailed Interaction Design and User Interface Design for the sake of clarity. Help Design was previously considered just a technique, since this was how it was presented in the usability literature. However, we have upgraded it to activity, since we have realised that the design of the help subsystem is an activity that is independent of the other design activities.

Usability Evaluation is also a separate activity from other development activities, but it is very complex. So we have subdivided it into the three main families of usability evaluation activities: Expert Evaluation, Usability Testing, and Follow-Up Studies of Installed Systems.

Having matched usability activities to software engineering activities, we could then map usability techniques to activities.

4. Allocation of Usability Techniques to Development Activities

Our primary basis for the allocation of usability techniques to development activities was the mapping of usability activities to general development activities.

We allocated all the usability techniques chosen in our survey to the development activities in Figure 2. For reasons of space, we cannot detail the full allocation here. However, Figure 3 shows the allocation for analysis

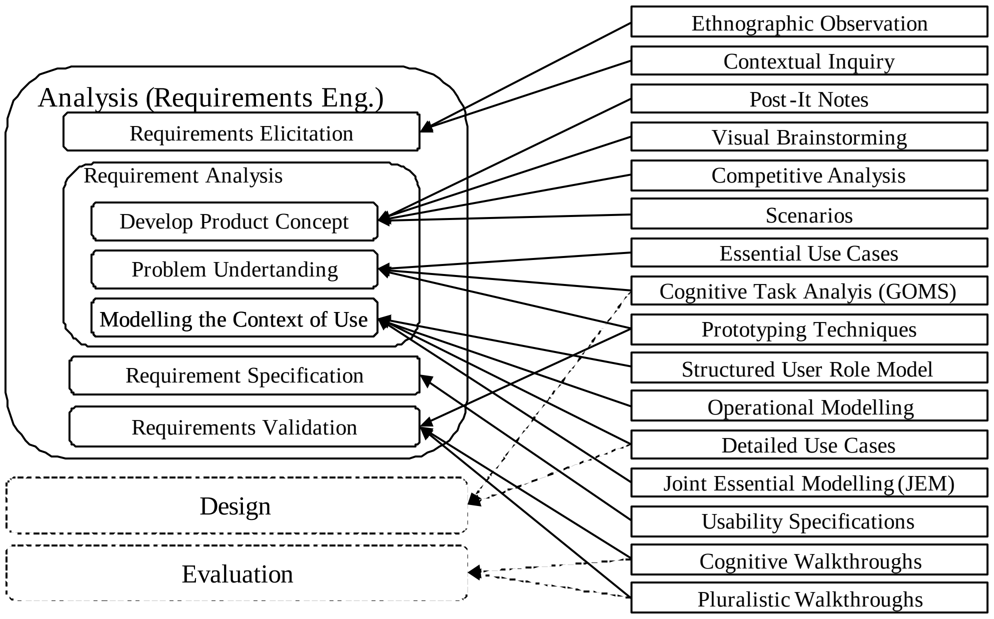


Figure 3. Allocation of usability techniques that apply in analysis

activities. The techniques are linked to the analysis activity to which they are allocated by an arrow. Some techniques may be applied in more than one activity, like Prototyping, which is used for Problem Understanding and also for Requirements Validation. Dashed lines mean that the technique is also applied during design or evaluation, like, for example, Detailed Use Cases, which are applied for Modelling the Context of Use, but also in design.

For the allocation of usability techniques to analysis activities, we have used their allocation to usability activities obtained from the literature survey, but we have also compared the objective of each technique and its products with the definition of analysis activities given in the SWEBOK.

5. Time Constraints for the Application of Usability Activities and Techniques

It is not enough just to allocate usability techniques to development activities, since not all usability techniques are applicable at any time in an iterative development. Any iterative process is divided into stages and, while not all iterative processes are the same, they usually follow a similar pattern regarding development time. We have defined a generic set of development stages that is applicable to most iterative processes. This model of process stages is shown in Figure 4. Prior to the iterative cycles, there is an initial exploration stage, which we have called Elaboration.

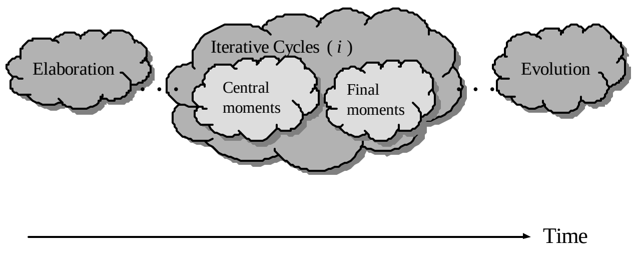


Figure 4. Stages in the development process

Afterwards, within the iterative cycles, we make a distinction between the main part of each cycle (Central moments) and the last part of each cycle (Final moments), where certain activities are performed, typically evaluation activities. Finally, when the system has been installed and is operational at the customer’ s site, the cycles are called Evolution.

We have studied the time of application of the different usability techniques in each activity to output a distribution of work across the different kinds of activities, related to the time in the development process when each effort is performed, as shown in Figure 5. The X-axis

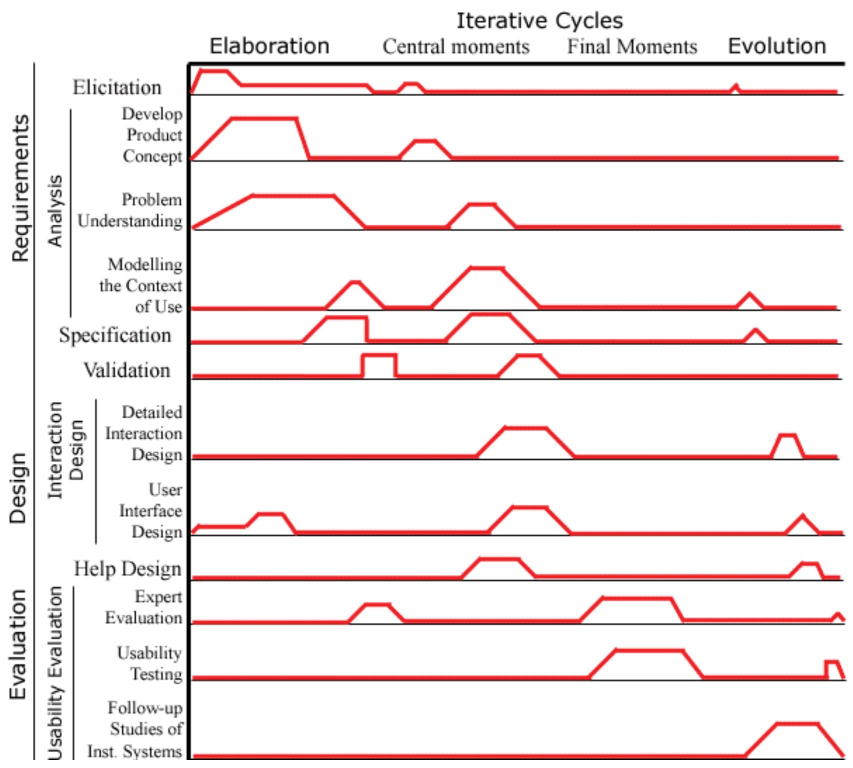


Figure 5. Amount of work on each activity at development stages

represents time. Therefore, the slopes in the lines denote some precedence between the different kinds of activities, like, for example, between the different requirements activities: first, there is some elicitation, followed by some development of the product concept (overlapping with the previous task), and then some problem understanding activities, and so on. Note that the amount of work on each activity is approximate, it should not be taken literally.

We focused first on usability techniques to allocate activities to moments in development time. We allocated each usability technique to a development stage according to its time of application in a user-centred development process. For example, techniques for developing the product concept are aimed at the very first development effort, where the needs are identified and the general system scheme is established, that is, the Elaboration stage.

Considering the time constraints for the techniques applied in each usability activity, we have then identified time constraints for usability activities. For example, elicitation is mostly performed in the Elaboration cycles (with more emphasis on the early stages), while some elicitation activities are performed at the beginning of the central moments within the Iterative Cycles, and a small amount of work may be done in Evolution cycles.

6. Definition of Process Increments

As a result of the work described above, we have classed the usability activities and techniques to be applied in the development process as increments, which we have called deltas, grouping techniques that are meant to be applied together according to the nature of the activities to which they belong (analysis, design or evaluation), and to the moment in development time when they can more effectively improve the usability of the software product.

We have defined seven deltas in order to get a better match with the general stages of an iterative software development process:

- **Δ1**: Early Analysis
- **Δ2**: Usability Specifications
- **Δ3**: Early Usability Evaluation
- **Δ4**: Regular Analysis
- **Δ5**: Interaction Design
- **Δ6**: Regular Usability Evaluation
- **Δ7**: Usability Evaluation of Installed Systems

Analysis activities are the ones that allow for a greater subdivision and call for careful integration with software engineering activities. Therefore, we have three deltas for analysis activities (Δ1, Δ2 and Δ4) plus Δ3, which, although formed by evaluation techniques, is applied in analysis activities. Traditional usability design activities are quite uniform and can be integrated in just one delta, Δ5. Usability evaluation activities, apart from the above-mentioned Δ3, have been divided into the activities to be performed during the main iterative cycles (Δ6), and the activities to be performed once we have an operational system working in the customer organisation (Δ7).

Figure 6 shows how the deltas group similar kinds of activities to be performed close together in development time. Each triangle represents one of the deltas, and they are placed over the distribution of work represented in Figure 5. The location along the X-axis represents the moment in development time when the delta should be applied, and the location on the Y-axis represents the kind of activities the delta groups. Note that the size of the deltas is not meaningful, as its only purpose is to cover the activities each increment contains.

Each process increment or delta is described according to the following structure:

- **Purpose**: The reasons why the delta should be added to an existing development process in order to improve the usability level of the resulting software product.
- **Phase**: Main type of activity: analysis / design / evaluation
- **Stage**: Development process stage where it is applicable.
- **Participants**: Members of the development team and other stakeholders who are meant to participate in the application of the techniques.
- **Activities/Techniques/Products**: List of the usability techniques that the delta groups, along with the documents or models produced by each technique. The techniques are grouped by the activity required to produce each product.

To make deltas handy for developers, each delta is summarised on just one page. Again, for reasons of space, we cannot detail all the deltas here. By way of an example, Table 1 gives a description of Δ1: Early Analysis. The set of techniques that form the delta are organised according to the activity to which they are allocated. This means that developers can add them more easily to their development process.

The products to be produced or refined by the application of each technique are detailed as well. For example, Ethnographic Observation and Contextual Inquiry are elicitation techniques and they help to build the Structured User Role Model, the Operational Model and the Use Case Diagram. Note that the same product may be related to several techniques, like, for example, the User Structured Model. This means that this product is defined by applying a conjunction of the usability techniques that produce or refine it.

Along with the description of the deltas, we offer developers a catalogue of usability techniques. The catalogue contains a brief description of the usability techniques mentioned in the increments (it includes 31 usability techniques). Deltas make reference to usability techniques that are mostly unknown to average developers, so the catalogue aims to provide them with an idea of what each technique is about. The catalogue refers developers to HCI literature and usability training for a deeper understanding of the techniques.

7. Conclusions

One of the reasons why usability techniques are not regularly used in software development, despite how critical software usability is for the overall quality of the software product, is the lack of integration with software engineering concepts, terminology and process.

We have presented a way of integrating usability activities and techniques into the software development process. The integration of usability techniques and activities is packaged in the form of increments that are to be incorporated at different places in a software development process. The appeal for a software development organisation lies in the fact that it does not have to abandon the in-house process to adopt some improvements, as it is enough to just modify the existing process by adding some or all of the proposed increments.

Offering average software developers an organised and manageable set of usability increments for their software development process, then, supports the current demand for flexible software processes.

We are currently applying the results we have presented here to three real projects at two companies. Through this validation, we intend to not only check and improve our proposal, but also to study the minimum support and training necessary for developers who are new to usability, for them to routinely apply the proposed increments and their associated techniques.

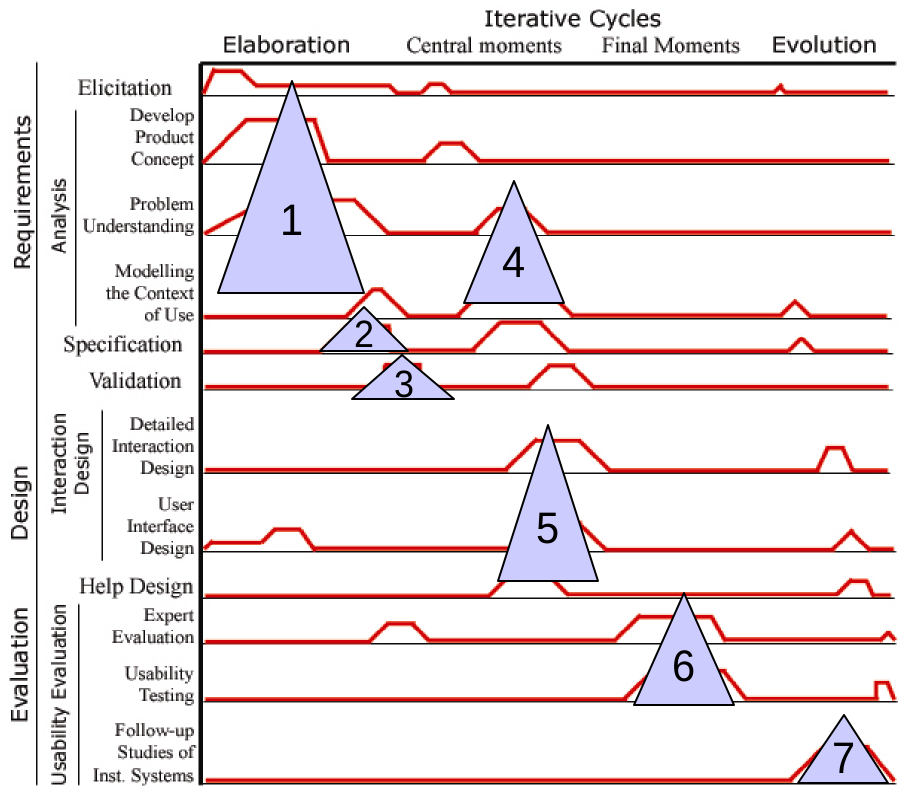


Figure 6. Grouping of usability activities in deltas according to the moment of application in development

Acknowledgments

This research has been funded by the "Ministerio de Ciencia y Tecnología" of Spain, project number TIC2002-00320.

Table 1. Δ1: Early Analysis

PURPOSE	Usability offers several techniques for analysis at the early stages of the project. These activities can give the tasks of requirements elicitation and analysis the user-centred flavour that ensures that usability is sufficiently catered for in later development activities.	
PHASE	Analysis	
STAGE	Elaboration	
PARTICIPANTS	Customer, users, developers	
ACTIVITIES	**TECHNIQUES**	**PRODUCTS**
ELICITATION	Ethnographic Observation	*-Structured User Role Model* *-Operational Model* *-Use Case Diagram*
	Contextual Inquiry	
REQ. ANALYSIS - MODELLING THE CONTEXT OF USE	Structured User Role Model	*-Structured User Role Model*
	JEM	*-Structured User Role Model* *-Essential Use Cases* *-Use Case Diagram*
	Operational Modelling	*-Operational Model*
REQ. ANALYSIS – DEVELOP PRODUCT CONCEPT	Post-It Notes	*-Product Concept*
	Visual Brainstorming	
	Competitive Analysis	*-Product Concept* *-List of needs and key/differentiating features*
	Scenarios	*-Scenarios*
REQ. ANALYSIS – PROBLEM UNDERSTANDING	Essential Use Cases	*-Essential Use Cases*
	Prototypes (paper and chauffeured)	*-Paper prototype*

8. References

[1] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (2nd Edition)*, Prentice Hall PTR, Upple Saddle River, NJ, 2002.
[2] IEEE, Special Issue on "Usability Engineering in Software Development", *IEEE Software*, Guest Editors: N. Juristo, H. Windl, and L. Constantine, Vol. 18, no. 1, January/February 2001.
[3] J. Anderson, F.Fleek, K. Garrity, and F. Drake. "IntegratingUsability Techniques into Software Development". *IEEE Software*, vol.18, no.1, January/February 2001, pp. 46-53.
[4] K. Radle, and S. Young. "Partnering Usability with Development: How Three Organizations Succeeded". *IEEE Software*, vol.18, no.1, January/February 2001, pp. 38-45.
[5] X. Ferre, N. Juristo, H. Windl, and L. Constantine. "Usability Basics for Software Developers", *IEEE Software*, Vol. 18, no. 1, January/February 2001, pp. 22-29.
[6] J. Preece, Y. Rogers, H. Sharp, D. Benyon, S. Holland, and T. Carey, *Human-Computer Interaction*. Addison Wesley, Harlow, England, 1994.
[7] ISO, *International Standard ISO 13407. Human-Centred Design Processes for Interactive Systems*, ISO, Geneva, Switzerland, 1999.
[8] L. L. Constantine, and L. A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centred Design*. Addison-Wesley, New York, NY, 1999.
[9] D. Hix, and H.R. Hartson. *Developing User Interfaces: Ensuring Usability Through Product and Process*. John Wiley and Sons, New York, NY, 1993.
[10] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, Reading, MA, 1998.
[11] J. Nielsen. *Usability Engineering*. AP Professional, Boston, MA, 1993.
[12] IEEE Software Engineering Coordinating Committee. *Guide to the Software Engineering Body of Knowledge - Trial Version 1.00*. IEEE Computer Society, Los Alamitos, California, May 2001.

Usability Engineering integrated with Requirements Engineering

B. Paech, K. Kohler
Fraunhofer Institute Experimental Software Engineering
Sauerwiesen 6, 67663 Kaiserslautern, Germany
{ paech,kohler}@iese.fhg.de

Abstract

In this paper we argue that the gap between Software Engineering and Human-Computer Interaction should be closed through the integration of usability engineering and requirements engineering. In particular, we present the elements such an integrated process has to cover.

1. Introduction

Requirements Engineering (*RE*) is the *systematic process of developing requirements through an iterative cooperative process of analyzing a problem, documenting the resulting observations in a variety of representation formats, and checking the accuracy of the understanding gained [1].* The essential tasks of RE during software engineering (*SWE*) are the elicitation and negotiation of requirements, their specification and validation as well as the their management over time.

The field of human-computer interaction (*HCI*) is concerned with the joined performance of tasks by humans and computational machines [2]. From the computer science perspective one essential contribution of HCI is the design, evaluation and implementation of interactive computing systems for human use [2]. The process that guides these tasks by specifying, measuring and improving the usability of a product is commonly called *usability engineering [3].*

The requirements process is typically characterized as an *analysis* process, where user needs and constraints must be elicited and analyzed. In contrast, most software engineers view HCI-activities as *design* (or test) activities.

In practice, this often leads to the misunderstanding that HCI-considerations can be brought in after the requirements are elicited and that requirements can be elicited without the consideration of the user interface.

In our view, a fundamental prerequisite for bridging the gap between SWE and HCI is that RE is understood as a *design* activity that includes the design of the user interface. During RE the *support for the user through the software system is designed*. There are many design decisions to be made such as the to-be-activities of the user tasks supported by the software, the system functions which perform parts of these activities, or the interaction between system and user when the system functions are executed.

To substantiate our claim we explain in the following the *requirements design decision types* (*REDT*) we have identified. This list covers typical functional requirements as well as typical HCI-issues such as screen structure.

Based on the REDT it is easy to argue that HCI and RE must be closely integrated to enable informed requirements decision making.

The paper is structured as follows: First we present the REDT, then we discuss the implications for an integrated RE and HCI process.

2. Requirements Design Decisions

In the following we concentrate on functional requirements and on *user interface- and information-intensive systems* (*UIS* for short).
By an extensive study of the HCI- and SWE-literature we identified 16 requirements decisions to be made for UIS. These are shown in Figure 1 at the end of the paper.

- **(T1) Decisions about the user tasks:** The decisions determine the user roles and the tasks of these roles to be supported by the system. Business processes determine these tasks.
  Example: A customer of a book-store has the tasks "search book" and "buy book".

- **(D1) Decisions about the as-is activities:** The user tasks consist of several activities. As-is activities are the steps user currently perform as part of their work without the new system. Decisions must be made what the as-is-activities of a task are (as these are rarely explicit) and whether they are relevant for the system. These decisions shape the understanding of the purpose and the responsibilities of the new system.
  Example: The activities of the "buy book"-task in an conventional book store are "select book", "carry book to the till", "pay book".

- **(D2) Decisions about the to-be activities:** It has to be decided how the as-is-activities will change as a consequence of the new system. As-is-activities always have potential for improvement. New technologies like the internet or handheld can result in radically new to-be activities. To-be-activities constitute the steps of the user tasks in the future.

Example: The to-be activities for the "buy book" task change in comparison to (D1). For the Web-book-store they are "select book", "provide payment information" and "order book". To complete the buying task the system additionally has to support the "delivery of the book" task of the bookseller.

- **(D3) Decisions about the system responsibilities:** Typically, the system does not support all to-be-activities, but only a subset. These are the system responsibilities. These decisions clarify the key-contribution of the system.

  Example: The booksellers' activity to "inform customer about shipping of the order" is supported by the software, and thus is a system responsibility. In contrast, the to-be-activity "package book" is not supported by software.

- **(D4) Decisions about the domain data relevant for a task:** System responsibilities of UIS manipulate data. Decisions have to be made what domain data is relevant for the system responsibilities.

  Example: "Book", "order" and "customer" are examples of domain data.

- **(I1) Decisions about the system functions:** System responsibilities are realized by system functions. The decision about the system functions determines the border between user and system.

  Example: The system responsibility "select book" is supported by the system functions "search book" and "shopping bag".

- **(I2) Decision about user-system interaction:** It has to be decided how the user can use the system function to achieve the system responsibilities. This determines the interaction between user and system.

  Example: The following interaction defines the system responsibility "select book":
  - User calls the "search book" function by specifying search criteria.
  - System displays a list of books.
  - User marks one or more books and calls the "shopping bag" function.
  - System stores the marked books in the shopping bag.

- **(I3) Decisions about interaction data:** For each system function the input data provided by the user as well as the output data provided by the system has to be defined.

  Example: Interaction data of the "search criteria" example in (I2) is for example "book title", "author name", "ISBN", "key-word".

- **(I4) Decision about the structure of the user interface (UI-structure):** Decisions about the grouping of data and system functions in different *workspaces* have to be made. System functions and data grouped in one workspace will be close together in the graphical user interface (*GUI*). This means that users need less navigation effort in the interface to invoke system functions and view data within the same workspace. By the UI-structure the rough architecture of the user interface is defined. This structure has a big influence on the usability of the system.

  Example: The shopping system has three workspaces a "select book" workspace, the "place order" workspace and the "provide customer data" workspace (see Figure 2 at the end of the paper).

- **(C1) Decision about the application architecture:** The code realizing the system functions is modularised into different components. In the decision about the component architecture existing components and physical constraints as well as quality constraints such as performance have to be taken into account. During requirements only a preliminary decision concerning the architecture is made. This is refined during design and implementation.

  Example: The software follows the model-view-controller paradigm consisting of three subsystems: the core, the GUI and the database.

- **(C2) Decisions about the internal system actions:** Decisions have to be made regarding the internal system actions that realize the system functions. The system actions define the effects of the system function on the data. These decisions also define an order between the system actions as far as is necessary to understand the behaviour of the system function.

  Example: The "place-order" function internally checks whether the customer paid bills of previous orders.

- **(C3) Decisions about internal system data:.** The internal system data refines the interaction data to the granularity of the system actions. The decisions about the internal system data reflect all system actions.

  Example: To check whether the customer paid bills of previous orders, a "payment behaviour" record has to be added to the customer data.

- (**G1) Decisions about navigation and support functions:**
  It has to be decided how the user can navigate between different screens during the execution of system functions. This determines the navigation functions. In addition support functions that facilitate the system functions have to be defined. These functions realize parts of system functions that are visible to the user, for example by processing chunks of data given by system functions in a way that can be represented in the user interface. Another example are support functions that make the system more tolerant against user mistakes.
  Example: A support function is the function "check address" that checks for the completeness of the customer address before the complete order is submitted. This avoids incomplete order information.

- **(G2) Decision about dialog interaction:** For each interaction the detailed control of the user has to be decided. This determines the dialog. It consists of a sequence of support and navigation functions executions. These decisions also have a strong influence on the usability of the system.
  Example:
  - User presses "send order" function.
  - Systems checks for completeness of order information.
  If e.g. the "shipping address" is missing, the system asks the user to specify the"shipping address". It opens the "customer account" screen containing the address fields.
  - User types "name", "street" and "city". User selects "country" from a list. User presses "send order" function again.
  - System shows "Thank you" screen and sends confirmation mail.
  -
- **(G3) Decisions about detailed UI-data:** For each navigation- and support-function the input data provided by the user as well as the output data provided by the system has to be defined. These decisions determine the UI-data visible in each screen.
  Example: To specify the country of the shipping address a "choice box" lists all European countries.

- **(G4) Decisions about screen-structure:** The separation of workspaces as defined in (I4) into different screens that support the detailed dialog interaction as described in (G2) has to be decided. The screen-structure groups navigation and support functions as well as UI-data. The decisions to separate the workspaces in different screens are influenced by the platform of the system.
  Example: The "select book" workspace is realized by two HTML-Pages. One to search for books ("search book screen") and one to view details of selected books ("book detail screen").

3. Implications for an integrated process

There is no approach so far in the literature which covers all REDT presented in the last section. SWE approaches such as the RUP [1], typically focus on D2-D4, I1,I3 and C1-C3. HCI-approaches focus on task modelling (T1) and user interface concepts (T1, I2-I4, G1-G4), e.g. [5][6][7][8]. With the advent of use cases, I2 and sometimes also G2 are nowadays also designed as part of the RE process, e.g. [9]. In practice, typically D1-D4, I1-I4 and C1-C3 are fixed separately from G1-G4. Often HCI-models such as user interface prototypes are used to stimulate elicitation of requirements, but the decisions wrt. the user interface are not seen as part of RE. One notable exception is [10] which uses the user interface design to drive the requirements specification.

In the following we argue that there are inherent dependencies between these REDTs which imply that HCI and RE activities must be closely intertwined.
The REDTs are aligned on 4 abstraction levels:

- **Task level**: The motivation for users to use a UIS is their work. UIS support the tasks users do as part of their work in a specific role. Decisions about the roles and tasks to be supported by the UIS are made on this level.

- **Domain level**: Looking at the tasks in more detail, reveals the activities users have to perform as part of their work. These activities are influenced by organizational and environmental constraints. At this level, it is determined how the work process changes as a result of the new system. This includes in particular the decision what activities will be supported by the system and which domain data is relevant for these activities.

- **Interaction level:** On this level decisions about the partition of activities between human and computer are made. They define how the user can use the system functions to achieve the system responsibilities. This decision has to be aligned with the decision about the UI-structure, which the user can use to invoke the system functions.

- **System level**: Decisions about the internals of the application core and the graphical user interface *(GUI*) are on the system level. They determine details about the visual and internal representation of the system to be developed.

Each level corresponds to a specific view on the system and its context on a specific level of detail. Furthermore, the decisions on one level depend on the decisions of the previous levels. Decisions of one level have to be made after all decisions of the previous level have been determined. If decisions of lower levels are made without taking into account the higher level decisions, the system will not support the users adequately in their tasks.

So the first major observation is that the decision about the tasks is an indispensable prerequisite for starting the RE process. As for example advocated in [11], RE approaches often start with goals. However, there is little guidance on how to identify these goals. Task support is the most important goal, since a system will only be accepted by the users, if their tasks are adequately supported. SWE can learn a lot from HCI for the identification of tasks.

The second major observation is that – in contrast to typical RE approaches - the decision about the UI-structure (I4) is an essential ingredient of the interaction level. Use Cases have shifted the focus from system functions (I1) to the interaction between system and user (I3), but without a preliminary UI-structure, it is not possible to make adequate decisions about the interaction (see [7]and [10] for forceful arguments why this is necessary). It is an interesting observation that such an integrating structure is also part of the application core and the GUI, namely the architecture (C3) and the screen structure (G4). At all levels there are decisions concerning behaviour chunks like activities, functions or actions as well as decisions concerning data. Interaction and dialog put these chunks into a sequence. UI-structure, architecture and screen-structure group data and behaviour chunks together.

A third major observation is that the design of the application core and the GUI are heavily interdependent. The details of the system functions depend on the way these functions are presented to the user. Navigation and support functions must be designed to ease the control of the user on the execution of the system functions.

Thus, altogether these dependencies imply that RE and HCI must be intertwined and thus RE and HCI experts must collaborate closely.

4. Conclusion

We have presented the fundamental decisions to be made during RE and argued that they need to include the usability engineering decisions.

The REDT and their dependencies have been identified from conceptual considerations as well as our experience. We have validated them by looking at different RE and HCI approaches. We checked whether these decisions are covered by these approaches and whether we miss issues covered in the approaches.

Of course, further application to industry-scale projects is necessary to evaluate them wrt. completeness and necessity. In [12] we sketch a specification method covering all the REDT. We believe that in practice there is rarely time to specify all of them explicitly, but we are convinced that depending on the project context, different subsets of the decisions for different subsystem parts should be specified explicitly.

In our view an agreement about the decision types and their dependencies is an important prerequisite for the development of joint RE and HCI curriculae and joint RE and HCI processes and tools. The curriculae should cover all the REDT such that RE and HCI experts are aware of the decisions to be made by the other experts and their dependencies. Processes and tools should in particular support the intertwining of the decisions e.g. with giving detailed guidance to use decisions from the higher-levels to come up with the lower-levels decisions.

5. Acknowledgement

We thank Soren Lauesen for fruitfull discussions on the intertwining of SWE and HCI decisions. The work has been funded by the BMBF in the project EQF under the label: e.-Qualification Framework - VFG0008A.

6. References

[1] Loucopoulos, P., Karakostas, V., *System requirements engineering*, McGraw-Hill, 1995

[2] Hewett, T., Baecker, R., Card, S., Carey, T., Gasen, J., Mantei, M., Perlman, G., Strong, G., and Verplank, W., ACM SIGCHI Curricula for Human-Computer Interaction, 1996, htttp://www.acm.org/sigchi

[3] Good ,M, Spine, T. M., Whiteside, J. , George, P., User-derived impact analysis as a tool for usability engineering, Conference proceedings on Human factors in computing systems, April 1986

[4] Kruchten, P. B., *The Rational Unified Process: An Introduction*, Addison-Wesley, 2000

[5] Diaper, D., *Task analysis for human-computer interaction.* Ellies Horwood, 1989

[6] Hackos, J.T., Redish, J.C., *User and Task Analysis for Interface Design*, John Wiley & Sons, 1998

[7] Beyer, H., Holtzblatt, K., *Contextual Design: Defining Customer Centered Systems*, Morgan Kaufmann Publishers,1998

[8] Constantine, L., Lockwood, L., *Software For Use*, Addison Wesley, 1999

[9] Armour, F., Miller, G., *Advanced Use Case Modeling*, Addison-Wesley, 2000

[10] Lauesen, S., Harning, S., "Virtual Windows: Linking User Tasks, Data Models and Interface Design*", IEEE Software*, pp. 67-75, July/August 2001

[11] Cockburn, A.,*Writing Effective Use Cases*, Addison Wesley 2001

[12] Paech, B., Kohler, K., Task driven requirements in object-oriented development, in Leite, J., Doorn, J.,(eds.) *Perspectives on Requirements Engineering*, Kluwer Academic Publishers, to appear

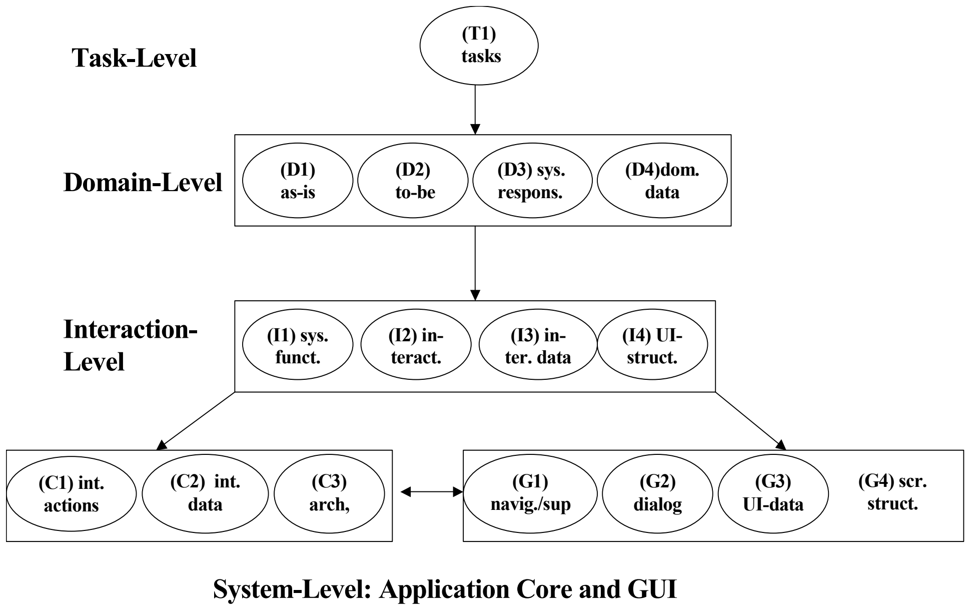
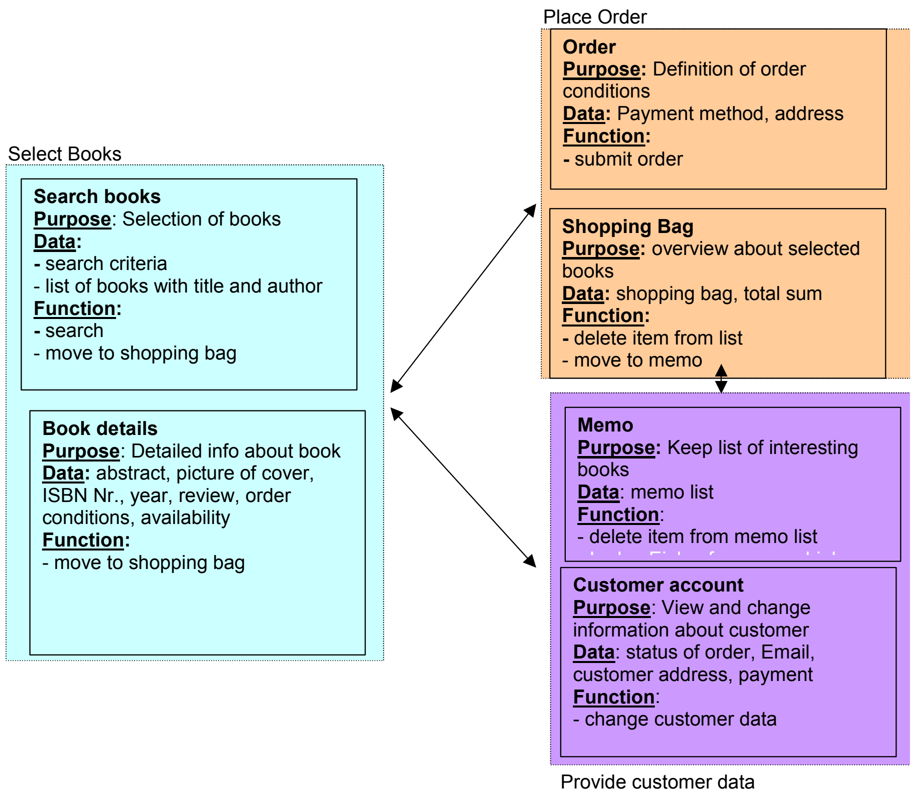


Figure 1: Requirements Design Decision Types



Select Books

Search books
Purpose: Selection of books
Data:
- search criteria
- list of books with title and author
Function:
- search
- move to shopping bag

Book details
Purpose: Detailed info about book
Data: abstract, picture of cover, ISBN Nr., year, review, order conditions, availability
Function:
- move to shopping bag

Place Order

Order
Purpose: Definition of order conditions
Data: Payment method, address
Function:
- submit order

Shopping Bag
Purpose: overview about selected books
Data: shopping bag, total sum
Function:
- delete item from list
- move to memo

Memo
Purpose: Keep list of interesting books
Data: memo list
Function:
- delete item from memo list

Customer account
Purpose: View and change information about customer
Data: status of order, Email, customer address, payment
Function:
- change customer data

Provide customer data

Figure 2: Workspace example

RUPi – A Unified Process that Integrates Human-Computer Interaction and Software Engineering

Kênia Soares Sousa, Elizabeth Furtado
Universidade de Fortaleza
Av. Washington Soares, 1321 – Edson Queiroz – Fortaleza, CE, Brasil
kenia@unifor.br, elizabet@unifor.br

Abstract

This research has the main objective of presenting a study on the areas of Human-Computer Interaction (HCI) and Software Engineering (SE), focusing on the importance of integrating HCI in SE, more precisely in the Software Development Process (SDP). More specifically, it concerns the adaptation of the Rational Unified Process (RUP) towards having HCI aspects integrated into its main workflows, generating the Rational Unified Process for Interactive Systems, called RUPi. The RUP is a well-established SDP that intends to guarantee quality by controlling the project schedule, budget, communication, productivity, and trustworthiness. Meanwhile, the RUPi intends to guarantee usability, accessibility, and acceptability by focusing on the users and on their context of use, modeling users' tasks, considering guidelines during the SDP, and defining mechanisms to design the User Interfaces (UIs) and to test their usability.

1. Introduction

Many SDP that exist nowadays guarantee quality for the developed system, focusing on schedule, budget, communication, and productivity. But on the other hand, few of them guarantee usability, that is, ease of learn and use. This way, they forget to focus on the users, in their tasks, in their contexts of use, and on the application of guidelines, which are recommendations concerning the usability of the system.

Many works as [1], [10], and [19] have shown that the integration of aspects related to HCI in the SDP brings improvements to the developed software, such as, higher user satisfaction derived from a user-centered SDP. Such integration aims to promote usability, accessibility, and acceptability for interactive systems.

In order to achieve these goals, this work presents a new SDP (RUPi[1]) [23] based on the existing one (RUP), but it has HCI concepts inserted into it. The RUP is a SDP that has many qualities that made it be chosen for this research, such as, being well-defined, applied worldwide, and flexible. The research on this SDP has the goal of acquiring knowledge on its workflows, as well as on the activities, artifacts and workers involved in each workflow. A *worker* performs activities and creates, modifies, and controls the artifacts. An *activity* defines the work that the workers perform in order to produce a valuable result to the project. An *artifact* is a tangible product that can be produced, modified, or used by a process. A *workflow* describes a group of activities that produce a result to the project and shows interactions among workers.

The HCI concepts studied in this research are able to contribute to the generation of usable systems when applied in a SDP. These HCI concepts are: human factors, guidelines, user-interfaces (UIs) for all, and the generation of model-based UIs. These concepts were chosen based on established literature that consider these aspects fundamental to be applied during the SDP.

Our strategy to demonstrate the importance of HCI concepts integration is the explanation of artifacts generated with the RUP, followed by the explanation and presentation of artifacts generated with the RUPi. The comparison between these artifacts will make it possible for us to envision the benefits of applying a SDP that includes HCI aspects.

In the RUPi, we created new workflow details, composed of a set of activities that will be included in the existing workflows. We inserted new workers and new artifacts to be produced by them while performing new activities, part of the newly inserted workflow details.

[1] The RUPi name was elaborated based on the UMLi[20], which stands for Unified Modeling Language for Interactive Systems, an extension of UML. UMLi addresses the modeling of complete interactive applications, specially their UIs.

The RUP will have its characteristics presented in the second section. Some aspects of HCI were selected and will be presented in the third section. Finally, these aspects will be included in the RUP, generating the RUPi, which will have its specific characteristics presented in the fourth section.

2. The Rational Unified Process

The RUP is a software development process concerned with the production of software that meets the needs of its end users within the predefined budget, schedule and with high quality. RUP is able to aggregate the best software development practices in a suitable way for a great number of organizations and projects. The six best practices are so-called because they are commonly used by successful organizations and for being able to mitigate the main causes of software development problems. The best practices applied in a SDP are the following: i) develop software iteratively; ii) manage requirements; iii) use component-based architectures; iv) visually model software; v) continuously verify software quality; and vi) control changes to software. [8]

The RUP architecture is divided in two dimensions. The first one represents the dynamic aspect of the process, and it is expressed in terms of phases: Inception, Elaboration, Construction, and Transaction. The second one represents the static aspect of the process, and it is expressed in terms of workers, artifacts, activities, and workflows.

Concerning its dynamic aspect, each phase executes some activities of related workflows in order to have a product in the end of each iteration. In *the inception phase*, the focus is on understanding general requirements, and on the definition of the project scope. In the *elaboration phase*, the focus is on requirements, but some efforts are devoted to the production of an architecture prototype in order to minimize technical risks by trying to adopt new solutions and learning new tools and techniques. In the *construction phase*, the focus is on the design and implementation, when the initial prototype evolves into the first operational product. In the *transition phase*, the focus is on ensuring that the system has the correct level of quality to reach its goals, when the defects are corrected, users are trained, features are adjusted, and elements are added in order to deliver the final product.

Concerning its static aspect, a process describes who (workers) is doing what (artifacts), how (activities), and when (workflows).

Figure 1 depicts the relationship among the four of the RUP workflows and their specific artifacts and workers. Only these workflows were chosen because they are the most commonly applied phases in any SDP, especially in interactive systems design, such as in [12] and in [15].

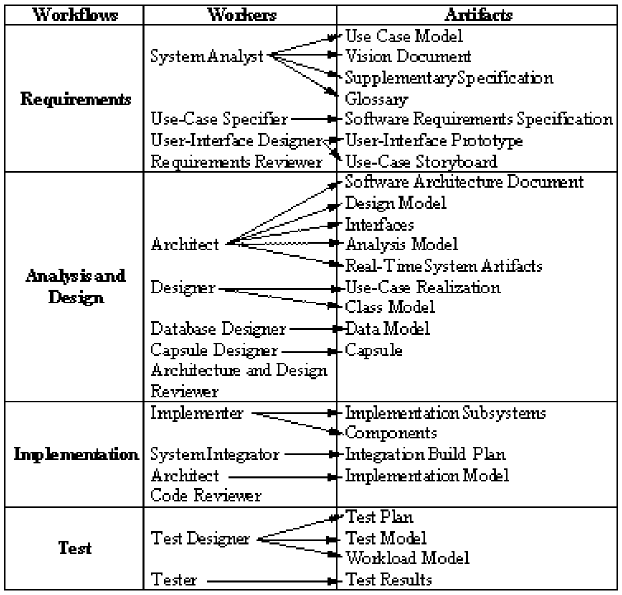


Figure 1 - Relationship among workflows, workers and artifacts in the RUP

The requirements workflow of the Rational Unified Process presents an approach to identify and control the system requirements. The analysis and design workflow of the Rational Unified Process shows an approach to translate the requirements into a specification that describes how to implement the system. The implementation workflow of the Rational Unified Process has an approach to implement systems by developing components, and incrementally integrating them, and producing prototypes to reduce risks. The test workflow of the Rational Unified Process presents an approach to evaluate the quality of the product, beginning early in the life cycle throughout the process until the evaluation of the final product delivered to users.

We verified some difficulties can be identified in the use of the RUP for developing interactive applications, such as: i) no consideration of human factors during the SDP; ii) no consideration of international and intercultural aspects during the UI design; iii) no consideration of user-centered models during the UI design and; iv) no consideration of UI customization based on the interaction technology and on the users. In order to attend these aspects, we propose the integration of HCI aspects that address the considerations mentioned above, resulting in the RUPi.

3. Human-Computer Interaction

HCI is becoming critical in the emerging information society because of the proliferation of interactive systems as tools for communication, collaboration, and social interaction among groups of people, and because human activities are increasingly

becoming mediated by computers, so computers are viewed as tools for productivity enhancement.
HCI concepts are becoming more important in the SDP because they focus the design process of an interactive system on the user. Besides, according to [11], "Almost half of the software in systems being developed today and thirty-seven to fifty percent (depending on life-cycle phase) of the efforts throughout the life cycle are related to the system's user interface."

The HCI concepts studied and presented in this work are: human factors, guidelines, UI for all, and model-based UI generation.

3.1. Human Factors

Human factors study the way people perform their activities. It is important to include human factors since the beginning of the software development in order to reach human factors goals. These goals are: time to learn how to use the software; speed of performance of tasks using the software; rate of errors by users while performing their tasks; retention of knowledge on the software over time; and subjective satisfaction of users about the software. [17]

Designers have to be sensitive to human capacities and needs related to technology use. This attention is extremely useful in order to avoid users feeling frustration, fear, and failure when facing a complex system with incomprehensible terminology, or chaotic layout. In order to develop an interactive system that provides users the possibility of performing their tasks effectively, designers must consider the diversity of human physical and intellectual abilities, personalities, cultural backgrounds, and age constraints. In addition, they can consider some cognitive models to help identify the task of the system to be developed [4].

3.2. Guidelines

Guidelines are a set of advices that guide designers towards developing usable UIs. They are useful to provide solutions to design problems or to serve as a source of information necessary for unfamiliar design situations.

There are some principles that can be applied in any interactive system design in order to achieve users' satisfaction. This satisfaction can be achieved by "providing simplified data-entry procedures, comprehensible displays, and rapid informative feedback that increase feelings of competence, mastery, and control over the system" [17].

Some organizations adapt guidelines to suit the reality of their cultural needs and characteristics or to suit a single project requirements, producing a document composed of a collection of principles and rules that can be used to develop consistent UIs, called style guide. Furtado [6] describes a collaborative process to define a style guide for an organization. This approach is implemented through a tool, which supports the participatory and evolutionary process of creation of a style guide. This approach for elaborating a style guide is structured with questions and answers and it can be used as a help if a developer encounters a problem when using a development tool or as documentation for new developers.

3.3. User Interfaces For All

In the information era, it is important to develop high-quality user interfaces, accessible and usable by a diverse user population with different abilities, skills, requirements, and preferences in a variety of contexts of use, and through a variety of different technologies, that is, user interfaces for all [18]. It is becoming compelling to design for the greatest possible user population.

One of the basic guidelines for HCI design is to study the user. Designers increasingly have to provide information to be used by diverse user groups, including people with different cultural, educational, training, and employment background, novice and experienced computer users, the very young and the elderly, and people with different types of disabilities.

The development of User interfaces for all can be achieved by applying international and intercultural, usability, accessibility, and acceptability methods.

- International and intercultural UIs provide availability of, and easy access to interactive systems among people for countries worldwide. In order to achieve this, user-centered design, globalization and intercultural issues are considered;
- In order to achieve usability, the interactive system should be easy to learn; efficient to use; easy to remember; have a low error rate; and pleasant to use;
- Information is accessible by all kinds of people, if careful attention is provided to their abilities, needs, and preferences during interactive systems development and;
- Acceptability is the possibility that users have to access, understand, and use interactive systems easily.

3.4. Model-Based User Interface Generation

The process of generating a UI based on models means that UI's characteristics and its functionalities are generated from specifications represented in different models.

These models can be the following: conceptual interface model, task model, user model, and context of use model. The conceptual interface model defines the data that a user can view, access, and manipulate through a UI. The visualization can be displayed for the user in different media, e.g., text, sound, graphics, etc. The task model consists of a list of tasks that allows designers to envision what tasks users perform. The user model includes descriptions of users, and the way they perform their tasks. Users perform their tasks in a specific environment, which is represented by the context of use model [4]. Graphical scenarios can help designers understand different contexts of use where multiple types of users may carry out multiple tasks.

4. The Integration of Human-Computer Interaction in the Rational Unified Process

4.1. Works with Human-Computer Interaction in Software Engineering

Although many software engineers have not yet noticed the importance of HCI in the SDP, some HCI experts have been trying to demonstrate that the integration between these two areas can bring improvements, such as, higher user satisfaction derived from a user-centered SDP [7].

Benner [1] presents the integration of scenarios in the software development process in order to better describe properties of the domain, define system requirements, and evaluate design alternatives. The intention to integrate scenarios in the software development process is devoted to better represent situations of users performing tasks while interacting with an interactive system. Having scenarios as artifacts, generated by specialized tools, can be helpful to guide designers during the UI generation and to evaluate usability.

Tavares [19] presents the integration of UI design mechanisms in the software development process by using UML diagrams and scenarios. The intention is to apply modeling techniques during requirements analysis in order to design the UI. The modeling techniques proposed are: scenario generation; task modeling; UML diagrams generation, such as, use case model, activity diagram, interaction diagram, class diagram; and a formalism related to the designer message specification language, called LEMD, which specifies the UI according to the semiotics engineering approach. These modeling techniques are applied in order to generate prototypes to be validated by users.

Long [10] presents the integration of human factors with SE by applying guidelines in a design method based on Denley's work [2]. It is also presented the application of a design method, called a Method for Usability Engineering (MUSE), based on Lim and Long's work [9]. The first phase involves: collecting information about users and their tasks, and producing a task model of the system to be designed. The second phase involves: eliciting users' needs and defining the application domain, preparing the conceptual design of the system considering users' needs and the application domain as a task model. The third phase includes: preparing the interaction design that represents human behaviors and considers the style guide, the users' needs and the application domain.

None of the works mentioned above considers the main HCI concepts during the entire development process, such as, human factors, guidelines, and user interfaces for all.

4.2. RUPi Characteristics

The six best practices adopted by the RUPi are:

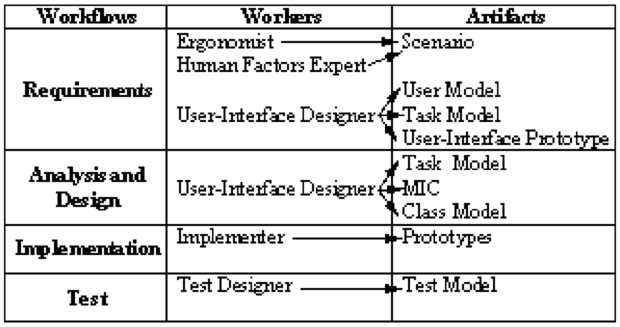
- Develop software iteratively – HCI aspects can be applied since early in the lifecycle until the product is ready for release, what raises the quality of the final product. Prototypes can be generated by applying the results of modeling techniques starting in the requirements workflow.
- Manage requirements - Analysis of the users' tasks, the environment where they are performing their tasks, and the devices that they are using, guaranteeing that the software accommodates users' reality.
- Use component-based architecture - Modular architecture definition, but, more specifically, it suggests the adoption of the n-tier architecture. They provide the software to be divided in independent layers, what provides higher reusability and flexibility.
- Visually model software – The use of user, task, context, and domain models. These models can be directly aided by graphical scenarios.
- Continously verify software quality - Quality assessment also starts early in the lifecycle and focuses on areas of highest risk. RUPi focuses on usability, accessibility, and acceptability.
- Control changes to software - RUPi proposes maintainability efforts towards the models defined during requirements elicitation.

The RUPi process structure is divided in two dimensions: the dynamic and the static dimension. The phases in the dynamic dimensions are the same ones defined in the RUP. What changes is the application of HCI concepts in every phase and their evaluation in every milestone. The elements in the static dimension, which are workers, activities, artifacts, and workflows, have the same definitions and purposes as the ones defined in the RUP. What changes is the content of each workflow, that is, which activities will be performed and which artifacts will be produced to accommodate HCI aspects and who will be the workers.

The RUPi has four workflows, which are based on the RUP workflows and were adapted by the inclusion of HCI concepts, described in section 3. The four chosen workflows for RUPi are: requirements, analysis and design, implementation, and test. This addition of HCI concepts is done in order to perform activities that produce artifacts that consider:

- human factors, considered during the development of an interactive system to help users perform their tasks effectively;
- usability requirements, used to decide which UI option is most suitable to the performance of a specific task by a particular user;
- accessibility and acceptability issues, applied during the definition of guidelines in order to accommodate characteristics of all different kinds of users;
- usability evaluation, used to test the usability of an interactive system, considering users´ characteristics, tasks performed, environment of work, and technology used and;
- graphical scenarios, generated to graphically represent specific situations experienced by users while interacting with the system.

Figure 2 depicts the relationship among the four of the RUPi workflows and their specific artifacts and workers.



Workflows	Workers	Artifacts
Requirements	Ergonomist Human Factors Expert User-Interface Designer	Scenario User Model Task Model User-Interface Prototype
Analysis and Design	User-Interface Designer	Task Model MIC Class Model
Implementation	Implementer	Prototypes
Test	Test Designer	Test Model

Figure 2 - Relationship among workflows, workers and artifacts in the RUPi

Following, the RUPi workflows are presented.

The Requirements workflow of the RUPi contains the workflow details of the RUP requirements workflow, and adds some more:

- user modeling, specifies the users who will use the system. This specification details information about: general knowledge, level of experience performing their tasks, using a computer, or using a similar system, preferences related to how data is displayed on the screen, physical and psychological abilities, role in the organization, working method, etc. Figure 3 depicts the user model for a project management software, in which we present four kinds of users with their specific characteristics;

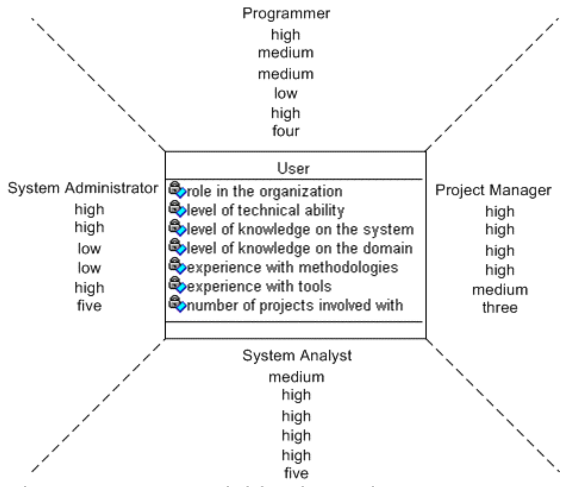


Figure 3 – User model for the project management software

- task modeling, defines the tasks performed by users and the cognitive characteristics necessary for users to perform them. In this work, we used the MAD formalism [16], in which a task is divided in two or more subtasks that are performed in order to fulfill a certain objective. This model is organized in hierarchical levels that link the tasks and organize their performance in: sequential, parallel, simultaneous, alternative, optional, or recurrent;
- context of use modeling, includes definition of the environment where the users are located in order to perform their tasks. This model can be enhanced with the generation of scenarios that are helpful to anticipate risks, prepare prototypes, and verify the accessibility and acceptability issues. Figure 4 depicts a scenario showing the library of a software development organization, where a developer is searching for a book [4];

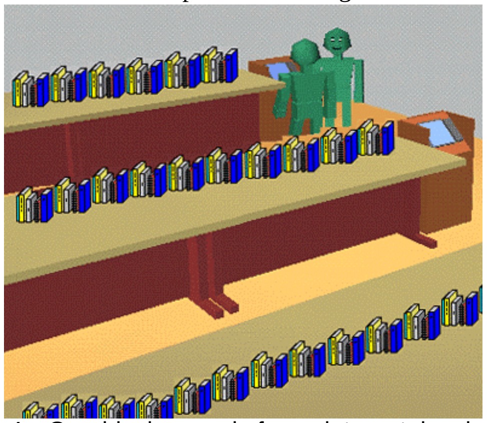


Figure 4 – Graphical scenario for an intranet development project

- domain modeling, specifies the aspects of the work to be performed with or without the system;
- definition of guidelines based on the users, their tasks, their working environment, and the system domain and;
- definition of usability requirements, related to users' satisfaction and the performance of the system. These requirements can be gathered by analyzing the user, the task, and the context of use models.

The Analysis and Design workflow focuses on transforming requirements into a design specification useful for the implementation workflow. This is achieved by defining the software architecture, designing components and the database. This workflow can be incremented with the following workflow details:

- Refinement of the task model performed in the previous workflow by considering characteristics of different kinds of users and different interaction styles;
- UI conceptual design, represents the interactive part of the system by graphically representing the navigational structure of the system. To choose the best option, the designer should associate usability requirements to the design options, as shown in figure 5, and discuss them with the users to come to a decis ion consensus and;

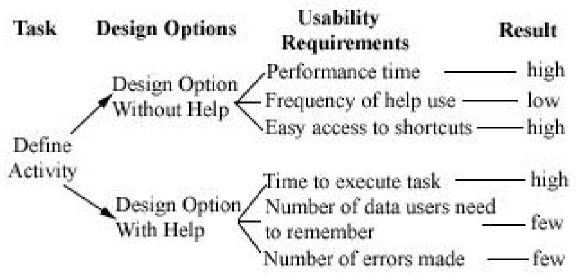


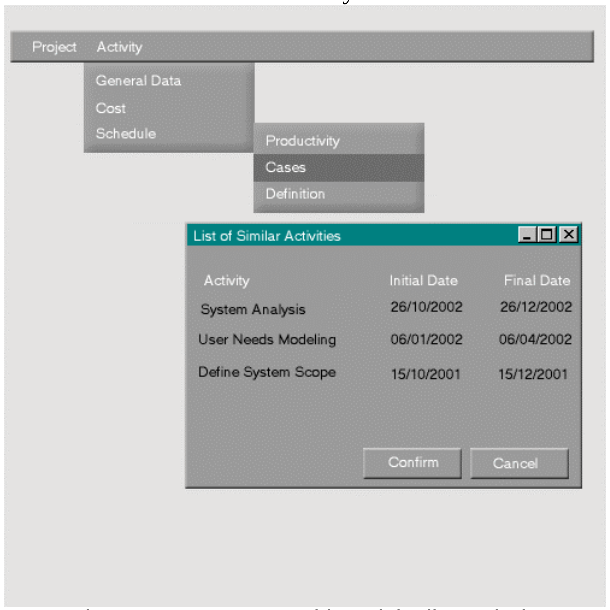
Figure 5 – Usability requirements related to design options

- class model, which is the foundation for the implementation of the project. Considering the UI conceptual design and domain modeling, we can define the class model in a n-tier architecture because the information is already separated. In such a class model, the first layer represents the interface, the second one is the control layer, the third is the business logic layer, and the fourth one is the data layer.

The Implementation workflow focuses on the codification and integration of components. The generation of the physical UI can be enhanced by the application of guidelines. Guidelines principles are interpreted considering the users, the tasks they perform, and the environment where they interact in order to select and apply them correctly. A software tool supports designers and developers during the development process, among other goals, to achieve usability. There are some tools that intend to help in the application of guidelines during the generation of UIs, such as SEGUIA [22] and SIERRA [21]. These tools are useful because of the absence of consistency among guidelines and the lack of a uniform structure among guideline documents. These shortcomings make it hard to organize guidelines and inter-relate guideline documents.

As a consequence of correctly applying guidelines during the implementation workflow, the physical UI is generated in a way that accommodates users' expectations towards usability, acceptability, and accessibility, thus, their satisfaction.

Figure 6 shows an adaptation in the UI towards the steps taken by users to achieve their goal while performing a set of tasks. The options in the menu offer a list of workers' productivity and a list of similar activities related to prior projects in order to help the user define the schedule for a specific activity. This adaptation is helpful for novice users, since it does not expect users to have enough experience in project management, nor to directly define the schedule for an activity.

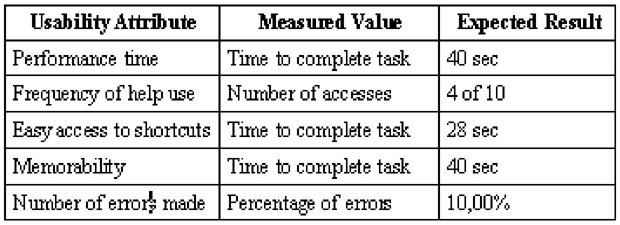


Activity	Initial Date	Final Date
System Analysis	26/10/2002	26/12/2002
User Needs Modeling	06/01/2002	06/04/2002
Define System Scope	15/10/2001	15/12/2001

Figure 6 – Prototype with activity list as help

Although some UI adaptation aspects demonstrate the differences in applying one of the two SDP, there are some aspects, related to the advantages of applying HCI in a SDP, which are not visually noticed. Such advantages are related to user satisfaction, user participation during the SDP that leads to more precision in the accommodation of users' requirements, risks anticipation derived from task modeling, among other benefits related to the cognitive level.

The Test workflow focuses on the quality of the product since early in the SDP. Its main activities are verifying the integration of components, correcting the implementation of requirements, and correcting detected defects. The main workflow detail to be included in this workflow is primarily concerned with usability aspects of the system developed, which is usability evaluation. Usability evaluation [14] should be a concern during this workflow since it involves analysis of a sy stem in the users' point of view. That is, this evaluation takes into consideration: users' characteristics, the tasks they perform, the environment in which they work, and the technology they use. Despite of the evaluation method applied, these aspects are important to predict the usability of a system. Evaluation tests can be composed of test cases that describe what aspects to be tested in the system. In the RUPi, these aspects are closely related to usability requirements defined in the requirements workflow. An example of a RUPi test case contains a list of usability attributes defined according to the usability requirements, the specification of the measured value, and the expected value for each tested attribute, as depicted in figure 7. The test case prepared using the RUP concerns functionality attributes.



Usability Attribute	Measured Value	Expected Result
Performance time	Time to complete task	40 sec
Frequency of help use	Number of accesses	4 of 10
Easy access to shortcuts	Time to complete task	28 sec
Memorability	Time to complete task	40 sec
Number of errors made	Percentage of errors	10,00%

Figure 7 – Test case related to usability attributes

4.4. RUPi Workflow Analysis

Analyzing the application of HCI aspects in these four workflows, we can notice the difference in the artifacts created by RUP and by RUPi. In the latter, there are more artifacts that consider the domain of work, the users, the tasks they perform, the devices they use, and the environment where they work. In addition to that, guidelines and usability requirements are considered in order to accommodate any specificity derived from the HCI aspects analyzed by the development team.

The main intention of this section is to differentiate the artifacts produced by RUPi from the ones from RUP, and especially to show the advantages achieved by the SDP from the RUPi artifacts.

In the RUP requirements workflow, the main artifact is the use case model. In the RUPi, the main artifacts are the user model and the task model oriented by the use case model. These models contribute to the generation of a UI that is suitable to the users' reality, opposed to being too generic to an extent that does not support users with their tasks.

In the RUP analysis and design workflow, the primary artifact is the design model that contains the class model. In the RUPi, the class model is designed according to the n-tier architecture, which allows this model to be reused by other systems in a more flexible manner. Besides that, there is an analysis of UI conceptual design options based on usability requirements. This analysis makes the decision process more concrete because it is based on HCI aspects and not on personal opinions.

In the implementation workflow, the most important artifact is the system. The UI designed following workflow details from RUPi present adaptation in the semantic level, which is related to adapting the UI towards the steps taken by users to achieve their goal while performing a set of tasks; and adaptation in the syntactic level, which is related to adapting the UI towards visualization aspects, such as, menu option, font size, font colors, background colors, etc. The UI designed following workflow details from RUP do not present any level of adaptation. It is noticeable that the RUPi's UI is more usable than the other one because of its possibility of customization.

In the test workflow, the key artifact is the test model, which contains test cases. The RUP's test cases verify and validate the system functionality, while RUPi's test cases focus on usability.

The application of RUPi will prove to achieve the goals mentioned above along with usability goals. These usability goals have proven to bring benefits, such as ease to learn and use the system, reducing training time and the number of errors, consequently, reducing costs and time spent in the correction of defects.

5. Conclusion

This research demonstrated the need for the integration of HCI concepts into a SDP because of the importance of concentrating on the users, their tasks, their environment, and the technologies they use.

The main contribution of this work is the definition of RUPi and its first workflows by taking into account HCI concepts.

Each RUPi workflow suggests the generation of some artifacts that intend to aid the development team in developing an interactive system that considers human factors, usability requirements, accessibility and acceptability issues, usability evaluation, graphical scenarios, among other aspects.

The future of this work is to define the other five RUPi workflows. In addition to that, we intend to develop a module for a project management software. This module will guide the development team in generating a UI by applying those HCI concepts integrated into the workflows. The automation of this integration will help in making it more easily applicable in many organizations.

6. References

[1] BENNER, Kevin M. et al. **Utilizing Scenarios in the Software Development Process.** California: Elsevier Science, 1993.

[2] DENLEY apud LONG, John. **Integrating Human Factors with Software Engineering for Human-Computer Interaction.** In: Seventh journal on Engineering of Human-Computer Interaction (IHM 1995). France: Cépaduès-Éditions, 1995, p.5.

[3] DIX, Alan et al. **Human-Computer Interaction.** England: Prentice Hall, 1993.

[4] FURTADO, Elizabeth. **Um Analisador Ergonômico de Interfaces Homem Máquina.** In: II Semana Universitária da UECE. Anais do VI Encontro de Iniciação Científica. Ceará, 1997.

[5] FURTADO, Elizabeth; Sousa, Kênia. **An Ontology-Based Method for Universal Design of User Conceptual Interfaces Using Scenarios.** In: Task Models and Diagrams For User Interface Design (TAMODIA 2002). Bucharest:INFOREC, 2002.

[6] FURTADO, Elizabeth; Sousa, Kênia; CÓLERA, César. **An Environment to Support Developers in Elaborating a Participatory and Evolutionary Help on Style Guide.** In: Seminário em Fatores Humanos para Sistemas de Computação, 2002, Fortaleza. Usabilidade: um direito. Fortaleza: Banco do Nordeste, 2002. p. 84 – 92.

[7] HEFLEY, William et al. Integrating Human Factors with Software Engineering Practices. In: Human-Computer Interaction Institute Technical Reports, Pennsylvania, 1994. Available in: <http://reports-archive.adm.cs.cmu.edu/hcii1994.html>. Accessed in 15 nov. 2002.

[8] KRUCHTEN, Philippe. **The Rational Unified Process - An Introduction.** 2 ed. New Jersey: Addison-Wesley, 2000.

[9] LIM; LONG apud LONG, John. **Integrating Human Factors with Software Engineering for Human-Computer Interaction.** In: Seventh journal on Engineering of Human-Computer Interaction (IHM 1995). France: Cépaduès-Éditions, 1995, p.7.

[10] LONG, John. **Integrating Human Factors with Software Engineering for Human-Computer Interaction.** In: Seventh journal on Engineering of Human-Computer Interaction (IHM 1995). France: Cépaduès-Éditions, 1995.

[11] MYERS; ROSSON apud HEFLEY, William et al. **Integrating Human Factors with Software Engineering Practices.** In: Human-Computer Interaction Institute Technical Reports, Pennsylvania, 1994, p.4.

[12] NEWMAN, William M.; Lamming, Michael. **Interactive System Design.** England: Addison-Wesley, 1995.

[13] NIELSEN, Jakob. **Usability Engineering.** California: Morgan Kaufmann, 1993.

[14] PREECE, Jenny et al. **Human-Computer Interaction.** England: Addison-Wesley, 1994.

[15] REDMOND-PYLE, David; Moore, Alan. **Graphical User Interface Design and Evaluation -A Practical Process.** England: Prentice Hall, 1995.

[16] SEBILLOTTE, Suzanne. **Hierarchical Planning as a Method for Task Analysis: The Example of Office Task Analysis.** In: Behavior and Information Technology. v. 7, n. 3. 1988, p. 275-293.

[17] SHNEIDERMAN, Ben. **Designing the User Interface: Strategies for Effective Human-Computer Interaction.** 3 ed. England: Addison-Wesley, 1998.

[18] STEPHANIDIS, Constantine. User Interfaces for All: New Perspectives into Human-Computer Interaction. In: STEPHANIDIS, Constantine. (ed) **Universal Access in HCI - Towards an Information Society for All.** New Jersey: Lawrence Erlbaum Associated, 2001.

[19] TAVARES, Tatiana A.; Leite, Jair C. **ARFDIU -Um método para Integrar Análise de Requisitos Funcionais com o Design de Interfaces de usuário Usando UML e outros formalismos.** In: V Symposium on Human Factors in Computer Systems (IHC 2002). Fortaleza: SBC, 2002, p. 313-323.

[20] UMLi. The Unified Modeling Language for Interactive Applications. Available in: <http://www.cs.man.ac.uk/img/umli>. Accessed in 30 jan. 2003.

[21] VANDERDONCKT, Jean. **Accessing guidelines information with Sierra.** In: Proceedings of Fifth International Conference on Human-Computer Interaction (HCI International 1995). London: Chapman & Hall, 1995, p. 311-316.

[22] VANDERDONCKT, Jean. **Assisting Designers in Developing Interactive Business Oriented Applications.** In: Proceedings of the Eighth International Conference on Human-Computer Interaction (HCI International 1999). Mahwah: Lawrence Erlbaum Ass., 1999, p. 1043-1047.

[23] SOUSA, Kênia Soares; FURTADO, Elizabeth. **Integration of Human-Computer Interaction in a Software Development Process.** In: HCI INTERNATIONAL 2003, 2003, Creta. 2003.

From HCI to Software Engineering and back

José Creissac Campos
Departamento de Informática
Universidade do Minho, Campus de Gualtar
4710-057 Braga, Portugal.
Jose.Campos@di.uminho.pt

Michael D. Harrison
Department of Computer Science
The University of York, Heslington
York YO10 5DD, UK.
Michael.Harrison@cs.york.ac.uk

Abstract

Methods to assess and ensure system usability are becoming increasingly important as market edge becomes less dependent on function and more dependent on ease of use, and as recognition increases that a user's failure to understand how an automated system works may jeapordise its safety. While ultimately only deployment of a system will prove its usability, a number of approaches to early analysis have been proposed that provide some ability to predict the usability and human-error proneness of the fielded system. The majority of these approaches are designed to be used by human factors specialists, require specific expertise that does not fall within the domain of software engineering and fall outside standard software development life cycles.

However, amongst this number, some rigorous mathematical methods have been proposed as solutions to the more general problem of ensuring quality of system designs but with limited success. This paper discusses their limitations both in terms of the broader software engineering agenda and in terms of their effectiveness for usability analysis, the opportunities that they offer and discusses what might be done to make them more acceptable and effective. The paper positions those methods that have been effective against less formal usability analysis methods.

1 Introduction

Although much emphasis has been placed on engineering more reliable hardware and software systems, relatively little significance has been attached to human factors issues. In fact human factors account for a very large proportion of failures in systems and the proportion is growing as methods of software development improve. “Although valid figures are difficult to obtain there seems to be general agreement to attribute somewhere in the range of 60-90% of all system failures to erroneous human actions” [7]. A study of 1100 computer related fatalities between 1979 and 1992 estimates that 92% of these fatalities could be attributed to human computer interaction [11]. These numbers clearly indicate the need for a better integration of human-factors concerns into the software engineering life-cycle.

The use of rigorous, mathematical, methods has been proposed as a solution to the problem of improving the quality of system designs. The potential of these methods has led to their application to interactive systems development (see, for example, [6]). In recent years a focus of attention has been the possibility of performing rigorous and systematic reasoning in order to assess the usability characteristics of systems. The application of automated reasoning techniques, in particular, has been studied [15, 2, 17].

In considering the specification of a device, usability engineering is concerned with how it interacts with the context in which it is placed. The focus is not so much on the device and its structure (though this is of interest) as the way that a device is embedded in its context, that is its work environment. Therefore, as well as the specification of the device, the specification of an interactive system where the device is located is also of interest. This interactive system will involve humans and devices. It is the interplay between them that should be analysed.

Usability analysis is a difficult process, and not necessarily one where it is obvious that mathematical methods and tools can play a relevant part. Roles for the different components involved may vary and the objectives which the system is intended to achieve will also change over time. Specifying how people actually behave rather than how they should behave is extremely difficult. In this paper formal approaches to the representation and analysis of interactive systems shall be considered that aid the process of usability engineering. In particular, these formal techniques shall be compared with a class of more informal but systematic usability analysis methods.

The paper discusses the role that model based techniques can play in modelling and analysing interactive devices, assumptions about their use, and the work that they do. In the next section less formal techniques are discussed before identifying their ingredients in the context of a more formal analysis.

2 Approaches to usability analysis

Ultimately, the only reliable method of assessing systems is to observe them in a broad enough set of situations that are as close as possible to those that are envisaged for its use. The problem with empirical analysis is that of the effort involved in constructing a prototype that is sufficiently robust to be fielded. Changes are therefore more expensive to make than they would be at earlier design stages. It is also difficult to obtain situations that are truly representative of the kind of environment in which the finished system will be used and are exhaustive of these possible situations. On the whole, successful systems evolve over time, using experiments with prototypes through trial and error.

The role of analytic techniques is not to solve all the usability problems, that would be impossible, but to produce early feedback about the design of an interactive system before expensive decisions have been made. In the main these techniques are intended for use by human factors experts. They operate on some informal representation of the design, for example a storyboard or a draft of the user manual. The description is rarely sufficient to provide a specification adequate as a basis for programming the system.

Two issues are important in the use of these informal methods. Firstly, because the method is informal it will be more difficult to perform it systematically and therefore ensure coverage. Secondly, the method relies on application by those who have sufficient expertise to apply it correctly, for example interpreting any questions that are asked of the design appropriately.

Although the aim of these methods is that they should be cheap to apply, their cost of application depends on the expertise of those who do the application. They have in common the goal to be able to reason about the usability characteristics of systems during the development stages of its design. Deferring the quality assessment until a prototype of the system is available, can become too costly. The possibility of verifying the specification of the system in the early stages of the design process could reduce the number of changes in the future and thereby reduce the overall cost of development.

Informal methods vary as to whether work representations, that is descriptions of what users are intended to do, are involved in their application. Two classes of methods shall be briefly discussed. The first, *usability inspection*, assumes no representation of the user or the work that is intended to be performed by the device under analysis. Instead it involves systematic interrogation of the design representation using a series of standard questions. *Cognitive walkthrough* on the other hand requires an initial model of how the device is to be used. Again questions are asked systematically of the device but these questions are related to the task representations.

2.1 Usability Inspection

The first class of methods involves systematic inspection of the design by means of guidelines for good practice in the design. It is assumed that there are a number of general characteristics that all usable systems should exhibit. In [12], a usability inspection method (Heuristic Evaluation) is proposed based on this type of approach. Applying heuristic evaluation involves setting up a team of evaluators to analyse the design of the user interface. To avoid bias in the analysis, the team members should not have been involved in the design process. They should be human-factors experts, although studies have shown that teams of users can also provide useful results [3]. Once all evaluators have performed their analysis, results are aggregated. This provides a more comprehensive analysis of the design.

To guide analysis a set of design heuristics is used based on general purpose design guidelines. The set proposed by [12] comprises nine heuristics: *simple and natural dialogue*; *speak the user's language*; *minimise user memory load*; *be consistent*; *provide feedback*; *provide clearly marked exits*; *provide short cuts*; *good error messages*; and *prevent errors*. Using these heuristics, evaluators will inspect the proposed design in order to assess if relevant guidelines are obeyed. Obviously not all heuristics will be appropriate all of the time.

The method prescribes nothing about how analysis is performed in terms of whether or not the system follows a guideline. Typically some type of informal walkthrough approach is used (see Section 2.2). It is also assumed that the team of analysts will envisage situations in which the system shall be used. In effect the analysts imagine possible work situations implicitly, whereas in the method to be discussed next these situations are made explicit.

2.2 Cognitive Walkthroughs

Heuristic evaluation makes no *explicit* documented assumption about how the system is to be used. Hence a heuristic might invite the team of analysts to consider any situation *at all* where feedback is not given. In practice it may well happen that in certain tasks *feedback* is essential while in other tasks users may have enough understanding of what will happen next for feedback to be superfluous.

A class of informal but *structured* techniques make explicit (to differing degrees) the work or tasks that are to be done in using the device. The consideration of what happens in the context of a particular task is therefore the fundamental unit of analysis. This general class of walkthrough techniques attempts a more work-based usability analysis by *simulating* how users are expected to behave when faced with the system under analysis.

Cognitive walkthroughs [9] is one such technique. Its aim is to analyse how well the interface will guide the user in performing tasks. User tasks must first be identified, and the model of the interface (the device model) must be sufficiently detailed to cover all possible courses of action the user might take.

Analysis of how a user would execute the task is performed by asking three simple questions at each stage of the interaction: *Will the correct action be made sufficiently evident to users?*; *Will users connect the correct action's description with what they are trying to achieve?*; *Will users interpret the system's response to the chosen action correctly?*. To answer these questions, probable courses of action the users will take must be presumed and documented as so-called user models. Problems are identified whenever there is a "no" answer to one of the questions above.

There are variations of this basic idea that assume more or less expertise from the analyst. Methods span human factors and human computer interaction and those developed in one tradition are barely known in the other. The methods that exist differ in two respects: in the way that items or units from the work description are used and in the types and generality of the questions.

For example, cognitive walkthrough asks three simple questions and is designed to explore usability or learnability, the THEA technique [16] provides a larger set of eighteen questions based around Norman's model of cognition, and focusses on human performance with a particular focus on human error. HEIST on the other hand [8] includes timing and other aspects of context in its very large set of questions. TRACEr invites questions in domain terms relating specifically to air traffic control concepts [18].

3 More formal analytic methods

Informal analytic approaches pose problems for engineers of complex interactive systems. Results are largely dependent on the skills of the evaluators: human factors skills rather than software engineering skills. This is a particular problem in the case of heuristic evaluation where the underlying theory is less explicit within the method. The human evaluators will bring into the analysis process a number of assumptions about user behaviour, and about the usage of the device. The validity of the analysis is limited by these assumptions.

Walkthrough approaches are more structured but may become extremely resource intensive as systems increase in complexity and possible activities that relate to the system become more diverse.

Although formal approaches are generally more limited in their application because of the cost of producing initial models they have the potential advantage that they can: demand more clarity from the analyst about the design description as well as the assumptions that form the basis for analysis; provide a precise description that can be used as a basis for systematic mechanical analysis in a way that would not otherwise be possible; and provide an external representation that may make it possible for some of the processes, currently only within the domain of expertise of a human factors expert, to be performed by an engineer.

Formal approaches bring more rigour and automation to usability analysis at the expense of flexibility. They require a specific type of model and a specific type of verification technique which can limit the analysis to be performed.

Proving properties of interactive systems shares with any other approach to analysis the requirement that there should be a model of the key features that are to be under scrutiny in the analysis. Here a model of interactive behaviour is required which highlights the interactive behaviour of the system while discouraging inappropriate bias in the analysis. A device description is to be explored in the context in which it is used. Instead of analysing all possible behaviours it is necessary to decide how to specify those behaviours that correspond to the way that the device shall be used. This itself narrows the context to those features that the designer considers to be important. Once the device and the rules for specifying its context have been explored it is also necessary to decide under what conditions the use of the device is of concern. It may be appropriate to analyse potentially extreme situations or where certain actions might be discrepant with the expectations of the user. Each of these issues shall be explored in more detail below. Four steps as represented in Figure 1 are involved in the process.

Initially, usability requirements, perhaps based on usability guidelines or heuristics, are identified and used to aid the specification of which parts of the device are to be explored.

Once the initial requirements have been expressed, a model of the interactive device and a set of properties are formulated. It is envisaged that at each stage of this process human factors expertise will be involved in assessing the validity of these properties or models. Models capture assumptions about the user's view of the system and properties capture constraints that if broken represent potential failures in the use of the system.

The next stage is to verify whether the properties hold of the device under the constraints imposed by the environment. For example it may be the case that it is necessary to explore a property that reflects the visibility of actions but only over certain paths corresponding to the likely tasks that are to be performed.

Finally the results are explored with the human factors analyst for their likely consequences. If the answer is pos-

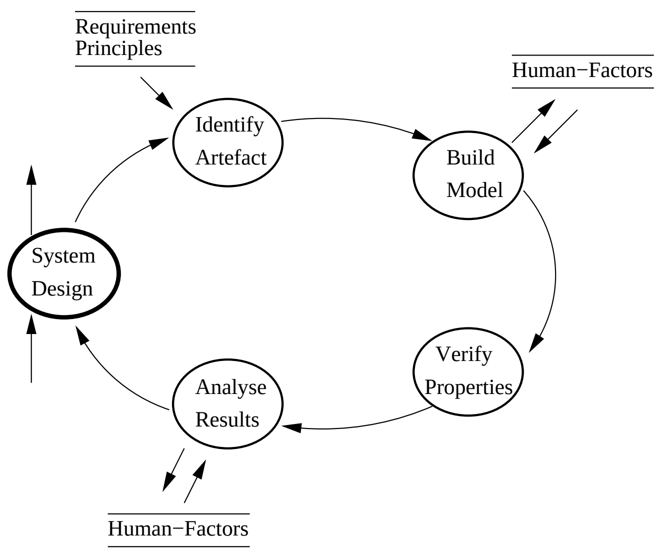


Figure 1. Integration of verification in development

interactor MCP
includes
  aircraft **via** plane
  dial(ClimbRate) **via** crDial
  dial(Velocity) **via** asDial
  dial(Altitude) **via** ALTDial
attributes
  $\boxed{\text{vis}}$ pitchMode: PitchModes
  $\boxed{\text{vis}}$ ALT: boolean
actions
  $\boxed{\text{vis}}$ enterVS, enterIAS, enterAH, toggleALT
  enterAC
axioms
  [] plane.altitude = 0
  [crDial.set(t)] $\text{pitchMode}'=\text{VERT\_SPD} \wedge \text{ALT}'=\text{ALT}$
  ...
  [enterAC] $\text{pitchMode}'=\text{ALT\_CAP} \wedge \neg\text{ALT}'$
  per(enterAC) $\rightarrow$
    $(\text{ALT} \wedge |\text{ALTDial.needle - plane.altitude}| \leq 2)$
  $(\text{ALT} \wedge |\text{ALTDial.needle - plane.altitude}| \leq 2) \rightarrow$
    obl(enterAC)
  ...

Figure 2. The MCP model

itive then it can be said that under the assumptions used in expressing the model and the property, the system satisfies the usability criteria. If the answer is negative then the reasons why must be investigated. A negative result might indicate that the device model is incomplete. It may therefore be necessary to determine whether all relevant aspects of the device have been considered. Alternatively it might point to a situation where the assumptions made about the user (or the context of usage) must be refined. For example it may be necessary to eliminate specific courses of action from the expected user behaviour. Finally it might be a genuine usability problem.

4 Analysing an interactive device

Formal analysis has been used in a variety of ways to explore interactive systems. In the next sections it will be shown how this type of approach can be useful in reasoning more systematically about usability.

A first approach is to model the device and use knowledge about the user to drive the analysis. This is the approach taken by Campos and Harrison [2]. They use a Modal Action Logic (MAL) based notation for modelling, and a tool that translates the models into SMV (a model checker). Properties can be checked using this system by writing them in Computational Tree Logic (CTL).

Modelling is performed from the point of view of the interaction between device and user and involves taking account of the specific analysis to be performed. To give a flavour of the notation, an excerpt of a model is presented in Figure 2.

The specification of the interactive behaviour of the device asserts constraints on the way the device is viewed, for example by defining user level actions and the way information about those actions may be perceived (cf. the $\boxed{\text{vis}}$ annotations). Further constraints may be imposed by considering the means by which behaviours should take account of contextual factors.

In [2] the mode control panel (MCP) of an aircraft is analysed regarding altitude acquisition. The design of the interface has been based on the plausible assumption that if the altitude capture (`ALT`) is armed the aircraft will stop at the desired altitude (selected in the altitude dial — `ALTDial`). This can be expressed as the CTL formula:

$$
\begin{aligned}
&\text{AG}((\text{plane.altitude} < \text{ALTDial.needle} \wedge \text{ALT}) \rightarrow\\
&\quad \text{AF}(\text{pitchMode=ALT\_HLD} \wedge\\
&\qquad \text{plane.altitude=ALTDial.needle}))
\end{aligned}
$$

which reads: it always (AG) happens that if the plane is below the altitude set on the MCP and the altitude capture is on then (AF) the altitude will always be reached and the pitch mode be changed to altitude hold.

In the process of checking the properties, counter-examples are generated by the model checker that reflect inadequacies in specification of the model or the property.

In [2] some initial counter-examples are produced that indicate situations where the pilot might take action to stop the aircraft from climbing (for example, changing the vertical speed). When the property is changed to prune those behaviours a new counter-example is produced which indicates that under specific circumstances the aircraft performs an implicit mode change that might lead subsequent user action to cause the aircraft to keep climbing past the target altitude. Palmer [14] reports that a similar problem was detected during simulation.

This counterexample prompts the designer to consider whether there is enough information provided by the device's user interface so that the pilot may be kept in the loop. The designers and human-factors experts can be called upon to clarify the full consequences of the counterexample. How aware will the pilot be of the mode change performed by the automation? Is this issue adequately covered in the manuals, and during training? Should the system be redesigned and how? What engineering constraints come into play regarding the design? Being able to raise these issues against a formal proof background in early design stages will undoubtedly allow for a better/safer design from the start. It will also reduce downstream costs of failing to discover these problems until too late.

5 Modelling the user to limit the analysis

As mentioned the approach above binds constraints about which paths can occur into the properties. There are other ways of restricting only to paths that are relevant. Rushby [17] models simple assumptions about the user's view of the system. The approach can be characterised as embedding elements of a user model into the device model and thereby constraining the behaviours of the system that must be analysed. The specification is modified to capture the changes in the user model as a result of actions by the system.

The problem to be explored using the model checker is whether there are discrepancies between the expectation that the pilot has of the ALT mode and its actual state. Considering the state of the altitude capture as defined by the device mode logic (ALT), and as idealised by the user (idealALT), the property could be expressed as: AG(ALT=idealALT)

A number of questions arise about how the user model is constructed. It must be decided what the user will or not be aware of, and whether the user will remember relevant information. These issues must be dealt with in the context of a model of cognitive behaviour. When the approach is used by software engineers there is a risk that wrong assumptions about cognition may be built into the model through ignorance of the science behind the model. These assumptions, however, are central to the reasoning process and because they are embedded in the specification their analysis becomes less open to discussion and dispute. In the example above, assuming the user does not notice the relevant indicator, then the mode problem is detected when we try to verify the property. When the user is assumed to notice the indicator then the problem does not show up in the analysis.

More elaborate attempts have been made to model what the operator is likely to do with a given device design. Two examples are syndetic modelling [4] which involves modelling the user as a process based on assumptions from a cognitive architecture (Interactive Cognitive Subsystems) and programmable user modelling (PUM) [19] which uses a planning model. In both cases, the analysis is based on describing the joint behaviour of device and user models. While the primary goal of the syndetic modellers has become to use formalism to make assumptions made in cognitive models more explicit and thereby more precise, PUM assumes that a cognitive architecture is programmed with domain and device specific knowledge. The aim of PUM therefore is to better understand and design the interactive system.

In this case, instead of enriching the device model with a (state-based) model of user's knowledge, a separate model of the knowledge needed by the user to operate the device is first developed, and then combined with the device model in order to investigate the behaviour of the resulting system. An *instruction language* is used to describe the goals of the user in using the device as well as the operators or methods as understood by the user to achieve these goals. Unlike the approach above these operators do not necessarily correspond to device level operations. Conceptual operators are defined in terms of pre- and post- conditions on the state as idealised by the user.

Once the models are fully developed a planning engine is used to show how the assumptions made in the model lead to a sequence of actions at the device level that meet the goals expressed at the domain level. This is typically done using means-ends analysis. The execution of a PUM is therefore an attempt to mimic how a rational user will behave when faced with the device. In most of the PUMA work as represented by [1] this process is ultimately performed relatively informally by hand.

6 Using the task to drive the analysis

In previous sections two approaches have been discussed for analysing an interactive system. In the first approach the model of the device is the primary focus for exploration. Assumption about context and user may be embedded within properties implicitly. In the second approach one model is used to describe the device in terms of the methods or actions that are available to users as understood by users, and a second model describes the goals as understood by users and encodes assumptions about cognition. Task analysis is a third approach. The motivation is similar to the other two in that the aim is to generate sequences that can be used as a basis for analysing the device. Task analysis however is a discipline that involves observing work as practiced, interviewing users, developing training material in order to understand the goals, sub-goals and actions as perceived by the users. The task analyst then represents the tasks that have to be carried out in terms of the goals, subgoals and procedures or plans that should be followed. The problem with this approach is the use of the word "should". The approach is normative and therefore may ignore important classes of activity that were unforeseen.

A variety of representations of tasks have been explored. Either using specific languages such as ConcurTaskTrees (CTT) or using more general languages, for example Ofan and *Murϕ*. These task representations are used in a variety of ways to analyse the interface. Paternò in [15], develops an interface model using LOTOS which is derived from a task description. The model therefore reflects the structure of the tasks that the device is supposed to support. More recently Paternò has used CTT to model and analyse the tasks models directly, no device model is used. Normative behaviour drives the analysis completely and properties such as *reachability* are analysed only in terms of the paths permitted by the task description.

Fields [5] has produced a more encompassing approach in which separate device and task models are developed, and the behaviour of their combination is analysed. In this case the analysis, while based on the normative behaviour represented by the task, systematically generates sequences that are based on perturbations of these normative task paths using simple mutations: omission, commission, repetition, reversal and so on, as used in hazard analysis techniques. Device models are written in the *Murϕ* language, and task models in a special purpose notation which is automatically translatable into the *Murϕ* language.

A further approach uses Ofan Statecharts which encode not just the device but also the task and the environment. These state charts are rendered checkable using the SMV model checker [10].

It would be straightforward to add a task model to figure 2. Whatever language is used, task models will describe the intended traces of behaviour. If only user actions are considered the actions in the task model will be the subset of visible actions of the device model, and the task model describes the intended traces of user behaviour. A possibility to representing those traces is using C/E Petri nets [13]. For example, the Petri Net in figure 3 represents the task of setting an altitude and toggling the altitude capture.

The Petri Net can be translated into an interactor for analysis. Each place is modelled by a boolean state variable representing whether it is marked. Each transition is modelled

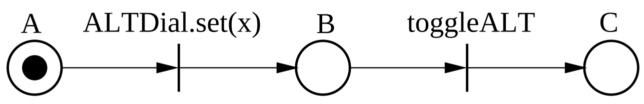


Figure 3. A simple Petri Net

by two axioms. A permission axiom stating when the transition is allowed to fire, and a modal axiom stating the effect of the transition on the marking of the net. For example, transition toggleALT is modelled by axioms:

per(toggleALT) $\rightarrow$ (placeB $\wedge\neg$placeC)

stating that it can fire when place B is marked and place C is unmarked (i.e. after ALTDial.set()); and

[toggleALT] $\neg$placeB' $\wedge$ placeC' $\wedge$ placeA'=placeA

stating that after toggleALT being fired place B becomes unmarked, place C becomes marked, and the marking of place A does not change.

The task model represented by the Petri Net can be combined with the device model binding task and device visible actions together. Properties can be checked of the form AF(goal state of the system) (the goal can be achieved by performing the task) or AG($\neg$hazardous state) (no hazardous state can be reached when performing the task).

7 Discussion

Ensuring the usability of a system is a complex and difficult endeavour. It has been shown elsewhere [3] that it is possible to reason about designs, from models of those designs, in order to improve usability prior to actual implementation and deployment of the system. Typical approaches, however, lack the rigour and thoroughness that only formal (mathematical) models can provide.

When attempting to model an interactive system for analysis we are faced with the problem of how to capture the diversity of concepts and concerns that are involved in the analysis of usability. Three basic aspects can be identified. The designed device, the user(s), and the work to be carried out (usually expressed in terms of tasks or goals). In a good interactive system design the device will support the user in carrying out the work. The quality of such support can be measured in many ways: efficiency, effectiveness, enjoyability, etc.

We have presented a number of proposals for the systematic analysis of usability that use different combinations of these ingredients. In the approach of [2] only the device is explicitly modelled. Considerations about user and work are used to drive the analysis. This style of unconstrained approach is best suited to detect problems with unpredicted device, or user, behaviour. This can be especially relevant for complex systems where tasks can interact in unexpected ways. It does not, however, provide much direct support when investigating how users will actually use the system. That step is left outside the formal reasoning phase. Additionally, some of the behaviours identified by the automated analysis might not be relevant to the real system. This analysis is also to be done outside the tool.

To address these issues, authors have proposed the use of user models in conjunction with the device models. This directly introduces human factors issues into the models, thus lessening the dependency of the analysis on human-factors expertise, and making it more accessible to software engineers. The general idea behind this style of approach is to set a specific goal, and see whether the *system* can find a way to satisfy it. This is specially interesting when investigating how a user will interact with a device in order to achieve a certain goal. In this aspect, it can be related back to Cognitive Walkthroughs. However, the planner may generate unexpected solutions that might lead the analyst to appreciate unforeseen consequences and scenarios that would not be otherwise explored.

Adding the user model means adding assumptions about user behaviour which will restrict the possible behaviours of the device. This bias in the analysis can lead to key issues in the device's design being ignored. User model architectures such as PUM help in guiding the process and in spelling out what assumptions are being made.

An alternative is to model, not the user, but the tasks that the user must perform. The approach will not give much direct support when investigating how easy users will find performing the tasks to be, but enables the analysis of how the device supports the proposed tasks. This can be done both in terms of whether performing the task achieves the desired goal, and of whether performing the task might lead the device into unwanted states. The consequences of erroneous behaviour are investigated by introducing errors in the task model.

It has become clear that the different styles of approach are complementary in that they support different styles of reasoning about the system. Unconstrained device analysis enables a more thorough exploration of the device. The interpretation of the results must then be performed with the aid of human-factors experts. By adding user models to the analysis one aims at analysing how users will be able to use the device. Using task models one analysis how the device supports the tasks the user is expected to perform.

8 Conclusions

A number of studies indicates that human factors account for a very large proportion of failures in systems [7, 11]. In the past, software development methods have attached relatively little significance to human factors issues. Unless that is changed the proportion of failures attributed to human factors will keep increasing as other aspects of software development improve. Clearly there is the need for a better integration of human factors concerns into the software engineering life cycle.

One specific facet of such integration is the need to reason about the usability of systems' designs from early in the development process. Usability analysis must not be left to the latter stages of software development when changes will be more difficult and expensive to make. Instead, a number of lightweight methods are needed that enable reasoning about the usability of systems from the early stages of design, thus enabling the design to be shaped by the usability criteria and concerns.

A number of discount (analytic) methods for the analysis of usability have been proposed, and studies have shown that they can be useful in detecting potential usability problems [3]. Discount techniques have distinct advantages over empirical methods. They are cheap to use, they do not require extended advanced planning and they can be used in the early stages before a design is implemented.

There are however problems when these techniques form part of a process where it is intended that they be applied by software engineers because they require human factors expertise. Another problem is the sheer size and complexity of the models that must be considered when it comes to complex systems. Usually the methods tend to focus on what could be called surface issues in the interaction, with little consideration of more complex behavioural issues of the interaction between user and device that will arise in real usage conditions. In this respect it can be argued that not even empirical evaluation can guarantee absence of such errors since for complex systems it is usually not viable to test all possible usage conditions/situations.

This paper presents some recent proposals of more formal and systematic usability analysis methods that attempt to provide answers to these problems. The claim is that these techniques, although narrower in scope, provide a more thorough analysis in which human factors claims are more clearly identified and substantiated using complementary expertise from other parties to the design process. Hence software engineers carry out part of the process but there are well defined stages where human factors input is required. In an initial phase human factors expertise is brought in to help select and model relevant system features, the analysis can then progress in a more typical software engineering setting, finally the analysis of the results must go back to human factors expertise.

The problems with using formal techniques relate to how well the techniques scale; how easy is it to use the methods; and whether the methods bias the analysis so that key issues may be ignored. We do not contend that this type of approach is the answer to all of the usability engineering problems. We do however feel that it has a role to play during interactive systems design and analysis. In the context of design problems where there is no obvious pay off using formal techniques, for example where more exploratory techniques are currently used, it might be argued that costs of applying these techniques are too high. However the situations where early analysis can reduce substantial downstream costs, either because the system is safety critical or because the cost of shipping volumes of less than usable systems cannot be countenanced, are increasing. More in depth and realistic studies are required to justify these claims.

References

[1] Ann Blandford, Richard Butterworth, and Jason Good. Users as rational interacting agents: formalising assumptions about cognition and interaction. In M. D. Harrison and J. C. Torres, editors, *Design, Specification and Verification of Interactive Systems '97*, Springer Computer Science, pages 45–60. Springer-Verlag/Wien, June 1997.

[2] José C. Campos and Michael D. Harrison. Model checking interactor specifications. *Automated Software Engineering*, 8(3/4):275–310, August 2001.

[3] Heather W. Desurvire, Jim M. Kondziela, and Michael E. Atwood. What is gained and lost when using evaluation methods other than empirical testing. In A. Monk, D. Diaper, and M. D. Harrison, editors, *People and Computers VII*, pages 89–102. Cambridge University Press, September 1992.

[4] D.J. Duke, P.J. Barnard, D.A. Duce, and J. May. Syndetic modelling. *Human-Computer Interaction*, 13(4):337–393, 1998.

[5] Robert E. Fields. *Analysis of erroneous actions in the design of critical systems*. DPhil thesis, Department of Computer Science, University of York, 2001.

[6] M. Harrison and H. Thimbleby, editors. *Formal Methods in Human-Computer Interaction*. Cambridge Series on Human-Computer Interaction. Cambridge University Press, 1990.

[7] E. Hollnagel. *Human reliability analysis: context and control*. Academic Press, London, 1993.

[8] B. Kirwan. *A Guide to Practical Human Reliability Assessment*. Taylor and Francis, 1994.

[9] Clayton Lewis, Peter Polson, Cathleen Wharton, and John Rieman. Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In *CHI '90 Proceedings*, pages 235–242, New York, April 1990. ACM Press.

[10] K. Loer and M. Harrison. Formal interactive systems analysis and usability inspection methods: Two incompatible worlds? In P. Palanque and F. Paternó, editors, *7th International Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS)*, volume 1946 of *Lecture Notes in Computer Science*, pages 169–190. Springer-Verlag, 2001.

[11] D. MacKenzie. Computer related accidental death: an empirical exploration. *Science and Public Policy*, 21(4):233–248, 1994.

[12] J Nielsen. *Usability Engineering*. Academic Press, Inc, 1993.

[13] Ph. Palanque, R. Bastide, and V. Senges. Task model - system model: towards an unifying formalism. In *Proceedings of HCI International conference*, pages 489–494, Yokoohama, Japan, July 1995. Elsevier.

[14] Everett Palmer. "Oops, it didn't arm." - a case study of two automation surprises. In Richard S. Jensen and Lori A. Rakovan, editors, *Proceedings of the Eighth International Symposium on Aviation Psychology*, pages 227–232, Columbus, Ohio, April 1995. Ohio State University.

[15] Fabio D. Paternò. *A Method for Formal Specification and Verification of Interactive Systems*. PhD thesis, Department of Computer Science, University of York, 1996.

[16] S. Pocock, M. Harrison, P. Wright, and P. Johnson. THEA: A technique for human error assessment early in design. In M. Hirose, editor, *Human-Computer Interaction INTERACT'01 IFIP TC.13 International Conference on human computer interaction*, pages 247–254. IOS Press, 2001.

[17] John Rushby. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering and System Safety*, 75(2):167–177, February 2002.

[18] S. Shorrock and B. Kirwan. Development and application of a human error identification tool for air traffic control. *Applied Ergonomics*, 33:319–336, 2002.

[19] Richard M. Young, T. R. G. Green, and Tony Simon. Programmable user models for predictive evaluation of interface designs. In K. Bice and C. Lewis, editors, *CHI'89 Proceedings*, pages 15–19. ACM Press, NY, May 1989.

Where SE and HCI Meet: A Position Paper

Mary Jane Willshire, Ph.D.
University of Portland
willshir@up.edu

Abstract

One way to remedy the gap that currently exists between software engineering and human computer interaction is to expose undergraduate students to the ideas, concepts, processes, methodologies and jargon of each field of endeavor. In teaching both software engineering and human computer interaction courses to undergraduates, I have found places where topics from one disciple can be inserted into the course material for the other to emphasize the connections that exist between the two. My hope is that by showing these bridges to students, they will enter the working world or graduate studies with an increased sensitivity to the issues. Perhaps this sensitivity will manifest itself into changed practice. This position paper discusses the approaches that I use to teach students.

1. Introduction

There are many challenges in teaching students with little or no industry experience either software engineering or human computer interaction. However, I feel that it is vitally important to introduce undergraduate students to the worlds of software engineering, human computer interaction, process, documentation and their interdependences. Cheston and Tremblay who point out that software engineering concepts can become part of introductory computing courses [2] also discuss the difficulties faced by those of us who attempt to do so. Typical undergraduate students have no concept of the scope of work or amount of effort involved in building a large, complex system. The ideas of systematic processes and documentation consisting of anything other than comments in their code are totally alien to them. In their minds, if they write a program and it compiles and runs, then they have built a "perfect" software system. Their idea of an interface is at best a menu driven, text based interface or maybe some Java graphics with simple buttons and sliders. Their programs are mostly written by one person, and tested by that same one person (or maybe their instructor or grader). Given these challenges, I have developed some techniques for teaching traditional undergraduate students these subject areas and especially some bridging points between software engineering and human computer interaction. This paper is an overview of those techniques—approaches that have worked and approaches that need more development.

2. Commonalities

What are the commonalities between the standard way that one teaches software engineering and the way that one teaches human computer interaction? For many, no commonality exists. However, if one looks, there are indeed places where in each course one can cross-reference to the other. Table 1 summarizes some of these commonalities and how I have introduced them in each subject area. For instance, every software project should have a detailed requirements document. How is this any different from the requirements that an HCI person would need? Or what should be added to the requirements document to reflect the users needs?

Table 1. Commonalities

SE in HCI	HCI in SE
Requirements for usability	Usability included in requirements
Use cases—task analysis	Use cases from the users
Detailed scenarios for system tasks	User-focused detailed scenarios
Role of prototypes in system development	Prototypes of user interface
Usability testing	Usability testing
Software process models and work products	User-centered development

Another commonality is the role of use cases and scenarios. What are these but descriptions of how a system will be used by the users—hence, these can form the basis for usability studies and the setting of usability goals. A third, though somewhat weaker commonality, is prototyping the system. A simple paper (or low fidelity) prototype is sufficient for early phases of usability testing. It can then be used by the developers in building the screens for their functional prototypes. The rest of this paper discusses in detail how these commonalities are brought forward in teaching software engineering and human computer interaction courses.

3. Teaching SE in an HCI course

When one is teaching a course with HCI as its major topic, one can use either a psychology-oriented approach or a software development approach. If one chooses a psychology-oriented approach, there are large selections of excellent texts to use for the course. Many of these texts are targeted to the graduate level, but can be used by an undergraduate course by selecting some chapters and omitting others. This approach builds on the human factors that have been studied for years by specialists in HCI. While we have much to learn from their efforts, this approach tends to ignore the software engineering issues entirely. For traditional undergraduate students with the mistaken idea that all of computer science is writing small programs for their own use, there is little understanding of or appreciation for this body of knowledge. I have tried this approach with undergraduates and felt that the course was a dismal failure.

If, instead, one decides to try a software development approach then one immediately encounters particular challenges. The first such challenge is that there is a severe lack of texts written that target an undergraduate audience in HCI with a software focus. One such text is Hix and Hartson [3] but it assumes a software background that many undergraduates do not have. A newer text is that by McCracken [4]. The McCracken text uses web page design as its theme. This topic is understandable to undergraduate students and has great appeal. I used this text (in manuscript form) this past academic year and my students really liked it. The one downside is that the text does not in and of itself, present actual software engineering. However, it does emphasize a process-oriented approach to interface design. This provides a way for the teacher to bridge to the software development processes.

What I have done that works well with undergraduates is to select a text that is not too heavy on the psychology, supplement it with lectures and assignments on software process, software lifecycle issues and use software work products such as requirements documents, use cases, scenarios and test plans as parts of the course project. For example, students are told that they have to do a project to design an interface and that there are several deliverables for this project. The first stage in the project is to select the "product" for which they will develop an interface. They must write up a description of this product, that is, provide a requirements document for their product. Initially, I don't call this assignment a "requirements document" but I tell them that they have to turn in a written description of the product, what it is, what it will do and so on. I gather these, read them and then start asking the students questions about their products. Based on these questions, they revise their descriptions. At this point, I point out that what we are doing is really eliciting the requirements for the project and writing a requirements document. That then leads into a discussion of what are requirements, how are they elicited, how are they used and where this fits into a development project.

After students begin to understand requirements, I then introduce the idea of use cases. Many of my students have never to this point in their studies considered the idea that some things are part of the system and some things are outside of the system. They easily understand that human users are not part of the system but provide inputs to the system; however, they do struggle with the concept of other external actors such as the network or a database. Unfortunately, an HCI class only has time to go so far down this path and for most undergraduates that means just working with humans as the only external actors in a use case diagram. Even so, this does make the students start to think about the interaction between the system and the users and makes them document how a user might want to interact with the system. Aha, now they are beginning to understand the context of a system and the use of a standard software modeling technique. Then I ask them what sorts of wrong things a user might do and have them do one or two of those as use cases. Recently, an article by Alexander [1] discussed this approach when taken to its extreme of hostile actions as "misuse cases".

Frequently, students realize that their requirements are incomplete as they begin to identify their use cases by doing a task analysis. They may be surprised by this, which then leads to class discussion about problems with incomplete or incorrect requirements. When the students have a good set of use case names in their use case diagrams, I then point out that these really don't have much detail and aren't all that helpful in designing the interface. So, from the use case diagrams the students move on to developing the scenarios for their use cases. For scenarios, the students write detailed interaction scripts for both correct and incorrect user actions (primary and secondary scenarios). They only have time to do this for one or two use cases. We discuss the value of doing a more complete set of scenarios, but typically, there isn't time in this setting to really expect the students to do so. Again, this introduces students to how one develops scenarios for the use cases, how one models user-machine interactions through a standard software modeling technique, and how this often causes the revisiting and revision of the requirements documents. The idea of revising something that they have already "finished" is another strange concept for many students. "You've already graded it, why should I redo it?" is a typical question. Well, my dear students, welcome to the world of iterative development.

By the time students have written scenarios for their interfaces, they think they are ready to start doing the design and truthfully, many have already done so in their heads if not on paper. However, I make them take one more step before going on to design. I have the students set up usability goals for their interface. I make them decide how good is good enough for their design. This is quite a shock to undergraduates. What do you mean, the users will make errors? No they won't, my interface will be so easy to understand that no one will make mistakes. I tell them that any software has to have acceptance criteria and they have to set the criteria for this product—how fast can a user be expected to perform some task? How many errors might a user make but still have acceptable completion of the task? And so on.

Setting usability criteria goes along with writing test scripts for later usability trials. Some students have some sense of testing software, but many feel that compilation alone is sufficient testing! They reluctantly set the criteria and start to think about testing. At this point, I usually relent and let them do their design and build a paper prototype for their interface. We discuss prototyping, its purpose and place in the development lifecycle and how it can be misused. We talk about different sorts of prototyping and why a paper prototype is often the best choice for interface design. We also discuss the disadvantages to low fidelity prototypes versus higher fidelity prototypes.

At last, the big day arrives when the students have written a usability test script and have a paper, or low fidelity, prototype of their design ready for testing. With great confidence they launch into their first experience at having a user who is not part of the team (but is a class member) try to use their design. They are very surprised when their classmates make errors in executing the tasks or don't understand how to begin a task, much less complete it. Back to the drawing board—iterative development and an easy to change paper prototype make more sense now. A second round of tests is scheduled and the project concludes with a set of revised documents and a lessons learned paper submitted for grading.

To drive home the points on software process, I like to have the students do a second, more extensive project. The second time around, they really understand that each work product will feed into a next phase of development and that discoveries in later phases mean the revision of earlier documents. They also have a greater appreciation for the level of detail that is required in the earlier phases in order to have a successful design at the end.

For the second project, I usually ask the students to find usability trial participants who are not members of the class. Here they realize that outside users are not the same as "inside" users and that significant effort is involved in finding participants. This then opens up a discussion of why organizations often do not use HCI best practices in their software development efforts. Students report learning more from this second attempt than they did from the first project.

4. Teaching HCI in SE class

Teaching software engineering to undergraduates in some ways has fewer hurdles and in other ways has higher ones. Good textbooks exist [5] [6], students have written programs and have used various software systems. This means that the terminology is more familiar or at least often connects with their experience in some way. However, most undergraduate students have never thought of a process for development; they have never had to consider what goes into a requirements document; testing means compile the program and run it once; and documentation means writing a few comments in the code as an afterthought. This means they come into a software engineering course thinking that they already know all there is to know about software development.

The first hurdle is to make students aware of software process and process models. I include a user-centered development approach as part of this presentation. As I discuss the phases of a project, I ask the students to identify stakeholders for each phase. Since users of the final product are indeed important stakeholders, it is not difficult to repeatedly bring the user' s perspective into the discussion. However, it is usually the instructor, not the students, who must identify this perspective. Students are not attuned to the user' s needs. They simply want to get to the coding part of the process.

Throughout the software engineering course, I inject usability terminology into the software jargon as often as possible. When we discuss requirements, I ask the students to consider what should be in the requirements document that would reflect the usability levels for the product. I ask them how they would elicit this information, how would they document it and so on. When we discuss what is meant by software quality, I make sure that usability is on the list. Most software engineering texts (suitable for undergraduate courses) do not address these issues. Likewise, when we do use cases and their scenarios, I require the students to consider the users of the system. The next time I teach this course (Spring 2003) I will have students add diagrams (a low fidelity paper sketch) of interface components to the documentation of use cases for the primary and secondary scenarios as is done in the book by Schneider and Winters [7].

One thing that I have not tried is to ask software engineering students to design a usability trial. If I add this to the course, I will have to drop or reduce coverage of other topics. Currently, I talk briefly about usability testing and encourage the students to take the HCI course to learn more. Ideally, students would take both courses. Until recently, each course was considered a technical elective and rarely did students take both. Our new students are required to take the software engineering course and HCI is an elective. This gives me at least one chance to teach our undergraduates that users are important and must be considered throughout the software development process. Of the two courses, HCI and software engineering, I feel that the software engineering course could use the most improvement in bridging the issues.

5. Summary

Teachers of undergraduate computer science students have the opportunity to introduce new ideas and concepts to open minds. Thus we can teach software engineering students the fundamentals of usability and teach HCI students the fundamentals of software processes. To be successful, the commonalities of each discipline must be identified and then those commonalities given a prominent place in the course structure. In this position paper, I discussed some of these commonalities and how I incorporate them as I teach the two courses.

6. References

[1] Ian Alexander. "Misuse Cases: Use Cases with Hostile Intent", *IEEE Software*, IEEE Computer Society, v20, n1, January/February 2003, pp 58—66.

[2] Grant A Cheston and Jean-Paul Tremblay. "Integrating Software Engineering in Introductory Computing Courses", *IEEE Software,* IEEE Computer Society, v19, n 5, September/October 2002, pp 64—71.

[3] Deborah Hix, and H. Rex Hartson. *Developing User Interfaces Ensuring Usability Through Product and Process,* Wiley Professional Computing, New York, 1993.

[4] Daniel D. McCracken and Rosalle J. Wolfe. *User Centered Web Site Design A Human Computer Interaction Approach,* to be published, Prentice Hall, 2003.

[5] Roger S. Pressman. *Software Engineering A Practitioner' s Approach, Fifth Edition,* McGraw Hill, 2001.

[6] Stephen R. Schach. *Object-Oriented and Classical Software Engieering, Fifth Edition,* McGraw Hill, 2002.

[7] Geri Schneider and Jason P. Winters. *Applying Use Cases, Second Edition,* Addison-Wesley, 2001.

Scenario-based Assessment of Software Architecture Usability

Eelke Folmer, Jilles van Gurp, Jan Bosch
Department of Mathematics and Computing Science
University of Groningen, PO Box 800, 9700 AV the Netherlands
mail@eelke.com, Jilles@cs.rug.nl, Jan.Bosch@cs.rug.nl

Abstract

Over the years the software engineering community has increasingly realized the important role software architecture plays in fulfilling the quality requirements of a system. The quality attributes of a software system are, to a large extent determined by the system's software architecture .Usability is an essential part of software quality. The usability of software has traditionally been evaluated on completed systems. Evaluating usability at completion introduces a great risk of wasting effort on software products that are not usable. A scenario based assessment approach has proven to be successful for assessing quality attributes such as modifiability and maintainability [12]. It is our conjecture that scenario based assessment can also be applied for usability assessment. This paper presents and describes a scenario based assessment method to evaluate whether a given software architecture (provided usability) meets the usability requirements (required usability). The Scenario-based Architecture Level UsabiliTy Assessment (SALUTA) method consists of five main steps, goal selection, usage profile creation, software architecture description, scenario evaluation and interpretation

1. Introduction

The quality attributes of a software system are to a considerable extent defined by its software architecture. In addition, design decisions in the beginning of the design process are the hardest to revoke. Therefore it is important to have an explicit and objective design process. Various researchers in the software engineering research community have proposed software architecture design methods: SAAM [1], ATAM [2] and QASAR [3]. The latter, the Quality Attribute-oriented Software ARchitecture design method (QASAR), is a method for software architecture design that employs explicit assessment of, and design for the quality requirements of a software system.

The architecture design process depicted in Figure 1 can be viewed as a function that transforms a requirement specification to an architectural design. The requirements

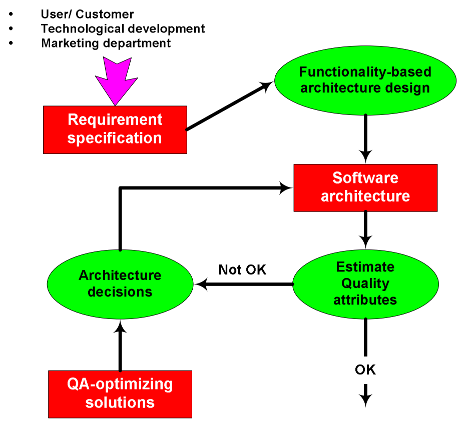

User/ Customer
Technological development
Marketing department
Requirement specification
Functionality-based architecture design
Software architecture
Estimate Quality attributes
Not OK
Architecture decisions
QA-optimizing solutions
OK

Figure 1: Software architecture design method

are collected from the stakeholders; the users, customers, technological developments and the marketing departments. These groups often provide conflicting requirements and have to agree on a specific set of requirements before the design process can start. The design process starts with a design of the software architecture based on the functional requirements. Although software engineers will not design a system on purpose that is unreliable or performs poorly, most non-functional requirements are typically not explicitly defined at this stage.

The design process results in a preliminary version of the software architecture design. This design is evaluated with respect to the quality requirements by using a qualitative or quantitative assessment technique. Using the assessment results, the estimated quality attributes are compared to the values in the specification. If these are satisfactory, then the design process is finished. Otherwise, the architecture transformation or improvement stage is entered. This stage improves the software architecture by selecting appropriate quality attribute optimizing or improving design solutions.

When applying architecture design solutions, generally one or more quality attributes are improved whereas other attributes may be affected negatively. By applying one or more architectural design solutions, a new architectural design is created. The design is evaluated again and the process is repeated, if necessary, until all non-functional requirements have been satisfied as much as possible. Other design methods such as SAAM or ATAM take a similar approach with respect to iterative refinement of the design. Generally some compromises are necessary with respect to conflicting non-functional requirements. The design process described here depends on two requirements:

- It is required to determine when the software design process is finished. Therefore, assessment techniques are needed to provide quantitative or qualitative data, to determine if the architecture meets the non-functional requirements.
- Development or identification of architectural design solutions that improve quality attributes.

As of yet, no architectural assessment techniques for usability exist. The goal of this paper is to outline and present an assessment technique for usability that fulfills one of the requirements to be able to design for usability on the architectural level.

2. Architecture assessment of usability

Most usability issues are only discovered late in the development process, during testing and deployment. This late detection of usability issues is largely due to the fact that in order to do a usability evaluation, it is necessary to have both a working system and a representative set of users present. This evaluation can only be done at the end of the design process. It is therefore expensive to go back and make changes at this stage. Next to that practice and experience [12] shows that improvement of, or design for quality attributes often requires the use of certain design patterns or styles. For instance, to improve portability and modifiability it may be beneficial to use a layered architecture style. From this example we can conclude that a large number of issues associated to usability may also require architectural support in order to address them.

One of the goals of the STATUS[1] project is to develop techniques and methods that can assess software architectures for their support of usability. The reason for developing such techniques is because the quality attributes of a software system are, to a large extent determined by a system's software architecture. Quality attributes such as performance or maintainability require explicit attention during development in order to achieve the required levels. We believe this statement not only holds for quality attributes such as maintainability or modifiability but also for usability.

Being able to assess the quality attributes such as usability during early development therefore is very important. Three types of architecture assessment have been identified [3]

- Scenario based assessment: In order to assess a particular architecture, a set of scenarios is developed that concretizes the actual meaning of a requirement. For instance, the maintainability requirements may be specified by defining change profiles that captures typical changes in the requirements, underlying hardware and so on. For each scenario the architecture is assessed for its support of this scenario.
- Simulation: Simulation of the architecture uses an executable model of the application architecture. This comprises models of the main components of the system composed to form an overall architecture model. It is possible to check various properties of such a model in a formal way and to animate it to allow the user or designer to interact with the model as they might with the finished system.
- Mathematical modeling: By using mathematical models developed by various research communities such as high performance computing, operational quality attributes can be assessed. Mathematical modeling is closely related to, or an alternative to simulation.

In our industrial and academic experience with scenario based analysis we have come to understand that scenario based analysis is a good technique to analyze software architectures because the use of scenarios allows us to make a very concrete and detailed analysis and statements about their impact or support they require even for quality attributes that are hard to predict/assess from a forward engineering perspective. A scenario based assessment approach has proven to be successful for assessing quality attributes such as modifiability and maintainability [12]. It is our conjecture that scenario based assessment can also be applied for usability assessment. The usage of scenarios is motivated by the consensus it brings to the understanding of what a particular software quality really means. Scenarios are a good way of synthesizing individual interpretations of a software quality into a common view. This view is both more concrete than the general software quality definitions [4] and also incorporates the uniqueness of the system to be developed, i.e. it is more context sensitive.

Usability is often defined in a very abstract fashion. Scenarios can make abstract usability requirements more specific. For example a usability requirement like "the system should be learnable" is much harder to evaluate for a system than a usage scenario defined as: "For a novice user operating on a helpdesk context, inserting a new customer in the sales database should be learnable", which is a more concrete statement.

[1] STATUS is an ESPRIT project (IST-2001-32298) financed by the European Commission in its Information Society Technologies Program. The partners are Information Highway Group (IHG), Universidad Politecnica de Madrid (UPM), University of Groningen (RUG), Imperial College of Science, Technology and Medicine (ICSTM), LOGICDIS S.A.

Before presenting our scenario based assessment technique we discuss in the next section the results of researching the relationship between usability and software architecture.

3. Usability Framework

One of the first goals of the STATUS project was to investigate the relationship between usability and software architecture. A framework has been developed [6] which illustrates this relationship. This framework provides the basis for developing assessment tools for usability. The framework is used for extracting information regarding the architectural information related to usability required for the assessment. The framework consists of the following concepts:

- Usability attributes.
- Usability properties.
- Usability patterns.

Figure 2 gives some examples of attributes, properties and patterns, and shows how these are related to illustrate the relationship between usability and software architecture. The concepts used are defined below.

3.1. Usability attributes

A comprehensive survey of the literature [7] revealed that different researchers have different definitions for the term usability attribute, but the generally accepted meaning is that a usability attribute is a precise and measurable component of the abstract concept that is usability. After an extensive search of the work of various authors, the following set of usability attributes has been identified for which software systems in our work are assessed. No innovation was applied in this area, since abundant research has already focused on finding and defining the optimal set of attributes that compose usability. Therefore, merely the set of attributes most commonly cited amongst authors in the usability field has been taken. The four attributes that are chosen are:

- Learnability - how quickly and easily users can begin to do productive work with a system that is new to them, combined with the ease of remembering the way a system must be operated.
- Efficiency of use - the number of tasks per unit time that the user can perform using the system.
- Reliability in use - this attribute refers to the error rate in using the system and the time it takes to recover from errors.
- Satisfaction - the subjective opinions that users form in using the system.

These attributes can be measured directly by observing and interviewing users of the final system using techniques that are well established in the field of usability engineering.

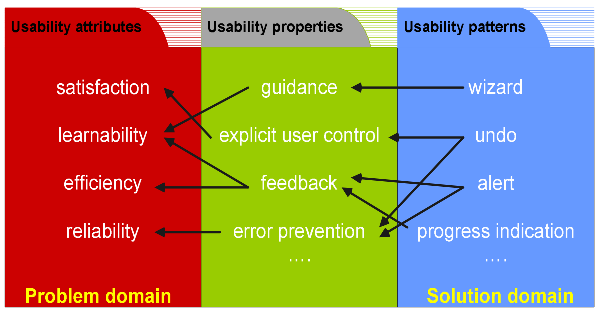


Figure 2: Usability framework

3.2. Usability properties

Essentially, usability properties embody the heuristics and design principles that researchers in the usability field have found to have a direct influence on system usability. Usability properties cannot be observed when evaluating usability for an implemented system. These properties can be used as requirements at the design stage, for instance by specifying "the system must provide feedback", however they are not strict requirements in a way that they are requirements that should be fulfilled at all costs. Usability properties should be considered as higher-level design primitives, which have a known effect on usability. It is up to the software engineer to decide how and at which levels these properties are implemented by using usability patterns of which it is known they have an effect on this usability property. The following properties have been defined:

- Providing feedback - the system provides continuous feedback as to system operation to the user.
- Error management - includes error prevention and recovery.
- Consistency - consistency of both the user interface and functional operation of the system.
- Guidance - on-line guidance as to the operation of the system.
- Minimize cognitive load - system design should recognize human cognitive limitations, short-term memory etc.
- Natural mapping - includes predictability of operation, semiotic significance of symbols and ease of navigation.
- Accessibility - includes multi-mode access, internationalization and support for disabled users.

3.3. Usability patterns

The term usability pattern refers to a technique or mechanism that can be applied to the design of the architecture of a software system in order to address a need identified by a usability property at the requirements stage (or iteration thereof). Various pattern collections have been defined [8], [9], the difference with other collections is that our collection considers only patterns which should be applied during the design of a system's software architecture, rather than during the detailed design stage. There is not a one-to-one mapping between usability patterns and the usability properties that they affect. A pattern may be related to any number of properties, and each property may be improved (or impaired) by a number of different patterns. The choice of which pattern to apply may be made on the basis of cost and the trade off between different usability properties or between usability and other quality attributes such as security or performance. 20 patterns have been identified and a detailed analysis of each usability pattern and the relationship between this patterns, usability properties and usability attributes can be found on http://www.designforquality.com/

The next section presents our scenario based assessment method for architectural assessment of usability. The assessment technique uses the framework described in this section as a source of input for extracting the information required for assessment.

4. Usability assessment technique: SALUTA

Our experience with developing and successful applying assessment techniques for maintainability has led us to develop a technique for usability which is based on the same principles. Assessing a software architecture for its support of a particular quality attribute basically comes down to a comparison between the required values of that particular quality attribute versus the provided value of that quality attribute. For usability assessment the required usability 'levels' are compared to the provided usability 'levels'. In case of scenario based assessment the required levels are specified by scenarios. The Scenario based Architecture Level UsabiliTy Analysis method (SALUTA) comprises the following steps:

1. Determine the goal of the assessment.
2. Describe required usability: create usage profile.
3. Describe provided usability: describe the SA.
4. Evaluate scenarios.
5. Interpret the results.

The steps are discussed and defined in detail in the following subsections:

4.1 Determine the goal of the assessment

The first step in the analysis method is to determine the type of results that will be delivered by its analysis. The following goals are distinguished:

- Predict the level of usability: give an accurate indication of the support of usability for an architecture.
- Risk assessment: detect usability issues for which the software architecture is inflexible.
- Software architecture selection: compare two candidate software architectures and select the optimal candidate which has the best support for usability

4.2 Create usage profile

Before an architecture can be assessed for its support of usability, first a way to describe the required usability is required. Preece [10] and Hix [11] suggest various techniques for the specification of usability. The way traditional techniques specify usability such as proposed by Preece and Hix are not suited for architectural assessment because of the following reasons:

- Very little is mentioned about usability requirements in scientific literature. In addition, real-life examples are rarely provided. Preece for example, presents much advice on usability requirements, but in a rather abstract setting without real-life examples. Most usability specifications are rather defined in an abstract fashion and therefore not suited for architectural assessment.
- Focus on interface evaluation rather than on dialog, whereas dialog is likely to have a much greater effect on usability.
- Traditionally usability requirements have been specified such that these can be verified for an implemented system. However, such requirements are largely useless in a forward engineering process. For example, we could say that a goal for a system is that it should be easy to learn, or that new users should require no more than 30 minutes instruction, however, requirements at such a level are hard to assess on an architectural level, because those can only be measured when the system is in use.

A more suitable format, as argued in section 2 for specifying required usability for architectural assessment is by using scenarios. A scenario profile describes the semantics of software quality attributes such as maintainability or safety for a particular system by means of a set of scenarios. In other words scenario profiles are an interpretation of the quality attributes in the context of the requirements. Scenario profiles are increasingly often used for the assessment of quality attributes during the architectural design of software systems [12]. A usage profile describes usability requirements in terms of a set of usage scenarios which are defined in the next section.

4.2.1 Scenario definition for usability

One of the main assumptions of usability is that usability depends on the context of use: the tasks, goals, the people and the environment. For usage scenarios the following variables have been identified that define a usage scenario:

- The *users*: the users play an essential part in the definition of a usage scenario. Users are part of the stakeholder population that interacts with the system. Examples of users: novice users, advanced users, system administrators and so on. A precise definition of users is not stated here because the type of users depends on type of system the user is working with.
- The *tasks* that a user can perform. The functionality that a system presents to its users is also a scenario variable of a usage scenario. The tasks directly relate to the interaction part of the definition of a scenario.
- The *context* in which the user operates. The context of operation is included because for some users performing a particular task the required usability may be different which can only be explained by the different contexts in which they operate.

A usage scenario is therefore defined as "an interaction between the users and the system in a specific operation context". The variables context, user and task which we have included in our usage profile definition are more or less recognized in the various definitions of usability. For example the ISO 9126-1 [14] definition of usability: the capability of the software product to enable specified users to achieve specified goals with effectiveness, productivity, safety and satisfaction in a specified context of use. Shackel [15] defines the usability of a system as the capability in human functional terms to be used easily and effectively by the specified range of users, given specified training and user support, to fulfill the specified range of tasks, within the specified range of scenarios.

Scenario profiles for describing required usability is created using the following steps:

1) Identify the users.
2) Identify the tasks.
3) Identify the context of operation.
4) Create attribute preference table
5) Scenario Selection
6) Scenario Prioritization

4.2.2 Identify the users: A representative list of distinct users has to collected and defined. Examples: Novice users, expert users or system administrators.

4.2.3. Identify the tasks: The next step is identification and selection of distinct tasks. Most systems have a lot of different tasks; therefore a representative selection of these tasks that are distinct has to be made. For example a task could be: insert new customer in database.

4.2.4. Identify the contexts of operation: The third step is determination of the unique contexts in which each user operates. Examples: helpdesk context or training environment.

4.2.5. Create attribute preference table: The attribute preference table (APT) is defined to relate a scenario to usability. However a scenario specified in such a way does not state anything about required usability as was our goal of creating usage scenarios. Therefore a way to relate it to usability has been defined. Stating that a scenario should be usable is not specific enough for analysis. In section 3.1 we have presented usability attributes. To express the required usability for a user performing a task in a specific context the scenarios are related to these usability attributes. In Table 1 an example of an APT is presented which is the result from an industrial case study. In this case the following usage scenario was defined: "end-users performing task quick search in training context". The required usability for the system is expressed by determining for each scenario values for the usability attributes. The goal of this analysis is to determine for each scenario an ordering or ranking between usability attributes which can then be used in the evaluation part of the assessment. The results for each scenario are then summarized in the (APT). The APT expresses the required usability for the system by stating for each scenario which specific usability attributes are important. In the example below we have assigned values between 1 and 4 to each attribute for each of the three scenarios. There are various ways to determine quantitative values for the preference to usability. It can be done as part of requirements collection process: typical users or experts assign values, for example they assign values between 1 to 5 to each attribute for each task and context. The assigning of values can also be done as a post requirements process (during assessment), where an expert (or a team of experts) determine values for the usability preferences, the usability requirements that are collected during requirements analysis can then be used as an informative source for assigning the values.

Table 1: Example APT

APT for Web-platform							
no	**Users**	**Tasks**	**Context**	**Satisfaction**	**Learnability**	**Efficiency**	**Reliability**
1	End-user	Quick search	training	2	4	1	3
2	End-user	Navigate	-	1	4	2	3
3	Content Administrator	Edit object	Helpdesk	2	1	4	3

4.2.6. Scenario selection: the attribute preference table that was created in combination with a descriptive list of users, tasks and contexts of operation can be used to summarize and describe the different scenarios that have been created. From this table, which holds all scenarios a scenario, profile is created by selecting scenarios that are representative. Scenario selection is the process of selecting those scenarios that are to be used in the assessment step of the analysis. Scenario selection results in a scenario profile which holds the set of relevant scenarios which will be evaluated. Different profiles may be defined depending on criteria for selecting the scenarios into the profile. The selection criteria influence the representativeness of the scenario profile, since in essence it is a kind of population sampling strategy. Two types of general scenario profiles have been identified:

- Complete scenario profile: "all scenarios that can potentially occur"
- Selected scenario profile: "a representative subset of the population of all possible scenarios

4.2.7. Scenario prioritization: Scenarios may be assigned additional properties, such as an associated weight, priority or probability of occurrence within a certain time. The selection of usage scenarios also depends on the goal of the analysis, if the goal is to:

- Predict the level of a quality attribute: Select scenarios that have high probability of occurring.
- Risk assessment: select scenarios that expose those risks.
- Software architecture selection: select scenario that highlight differences

The process of identifying users, tasks and contexts of operation, creating and APT, selecting an prioritizing scenarios will often be performed in one step. This subsection has discussed how to create a usage profile. Going back to our assessment approach we need to compare this usage profile (required usability) to the provided usability. But to be able to do this we need to know how to describe the provided usability which is discussed in the next subsection.

4.3 Describe the software architecture

The third step, architecture description, concerns the information about the software architecture that is needed to perform the analysis. Generally speaking, usability analysis requires architectural information that allows the analysis to evaluate the scenarios. The result of this step is a description of the provided usability. Information related to the architecture; for example, box and line diagrams or documented design decisions, may provide data about various quality attributes but since our interest lies in usability only the information that is related to usability is required. To achieve this, the information required is extracted using our framework described in section 3. Different types of assessment techniques have been defined depending on the amount of architectural information that is available for assessment or what information one is willing to acquire to get a more accurate result from the assessment. The following subsection discusses the different assessment types defined and the architectural information necessary to perform that type of assessment.

4.4 Evaluate Scenarios

Assessing an software architecture for its support of usability is done by comparing the required usability 'levels' to the provided usability 'levels'. The levels are specified by usage scenarios. In section 4.2 a technique is discussed for capturing and describing the required usability using a usage profile. For each usage scenario in the profile the architecture is analyzed for its support of that scenario. The process that identifies the support for the scenarios is defined as architectural support analysis. Eventually the results from the analysis are summarized into an overall result. For example the number of supported scenarios versus the number scenario not supported. This number will be an indication of the support of the architecture for its support of usability. The architecture will be accepted or rejected based on the number of usage scenarios in the usage profile that are

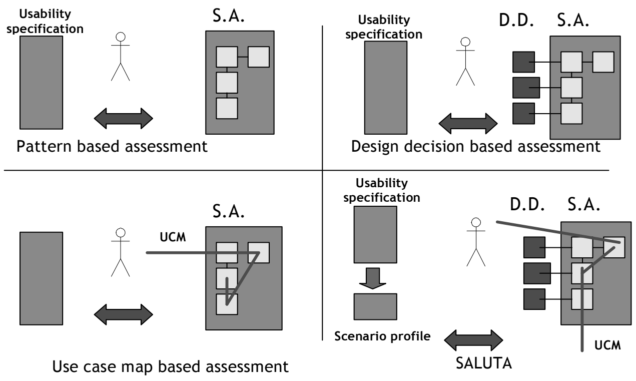


Figure 3: Scenario based assessment techniques

accepted. Three different types of scenario evaluation techniques have been defined (as depicted in Figure 3):

- **Pattern based**
- **Design decision based**
- **Use case map based**

The framework is used to extract the architectural information required for each assessment technique.

4.4.1. Pattern based. By analyzing an architectural description of the system an expert assesses the architectures support of usability. The architecture designs present within the development are used as a source of input for this type of assessment. The architectural design can be a simple box and line diagram or for example a 4+1 view on the architecture. These designs can provide a lot of information about quality attributes and since the subject of our evaluation is usability we are only interested in those parts of architecture information that are related to usability. To acquire this information we use the framework to extract the required information. For expert based analysis an identification of patterns that influence usability in the system is required. By heuristically evaluating the system using the list of patterns identified in [6] a list of patterns or possible derivatives of those patterns implemented can be identified. The list usability patterns present in the software system should provide the information necessary for the software engineer to decide if a scenario will be supported by the architecture. For each scenario the software engineer will determine which patterns are involved and whether the usage scenario is sufficiently supported.

4.4.2. Design decision based. Not only a description of the structure of a system as it is decomposed into components and relations with its environment may be used for analysis. The design decisions that led to that particular architecture are also very important. The earliest design decisions may have a considerable influence on various quality attributes of the resulting system. However such design decisions which are made during design are most often not documented. If they have been however they may be used as a source of input for this type of assessment. For design decision based

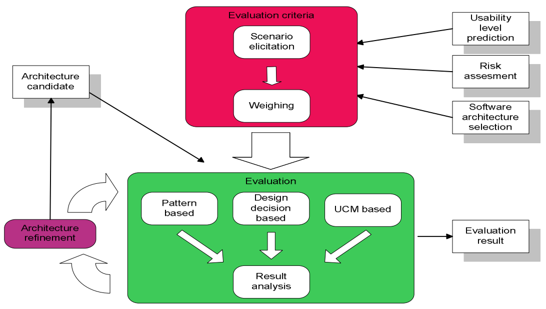


Figure 4: Assessment process

analysis it needs to be determined which design decisions have been made with regard to usability. By heuristically evaluating the design decisions made during design, using the list of usability properties defined in our framework the required information for the assessment (the design decisions that relate to usability) is extracted. This type of assessment heavily depends on the amount of information documented during or after initial architectural design. If no design decisions have been documented, this information could be retrieved by interviewing the system architect(s). For design decision based analysis, the list of design decisions that have been extracted using the framework is used to determine the support for each usage scenario. For each scenario we analyze if a scenario is affected by the design decisions and whether this has resulted in sufficient support for that scenario.

4.4.3. UCM based. An even more detailed way of assessing is to use use case maps (UCM) for describing the architecture. Using UCM for describing the architecture has the following benefits:

- Use case maps describe behavioral and structural aspects of systems at a high (architectural level) of abstraction
- Can capture user requirements when very little design detail is available
- UCM are easy to learn & understand but precise.

Architecture designs and design decisions made during design can be provided by the software architect who assists the analysis. Use case maps in case not present can be constructed with the assisting software architect. Based on the scenarios in the scenario profile for each scenario a use case map is build. Some tasks may have the similar or the same use case maps. The use case map allows us to analyze various static properties that relate to the usability attributes layer in our framework. For example a use case map may visualize the number of steps or time it takes to perform a task. The number of steps may be an indication to the efficiency or learnability attribute. Next to providing static information use case maps allow close analysis of architectural components (such as a patterns) involved in that particular scenario. The information gathered during this analysis is an extra source of input for the architectural support analysis of the scenarios.

4.4.4. Summary: The types of assessment techniques presented here are complementary as shown in Figure 4. In general expert based assessment can be applied in most cases, assuming that at least some basic form of architectural description has been made for design which allows for identification of patterns. Design decision and use case map based assessment may give additional information for the architectural support analysis. However because these types of required information are not always present these can be retrieved or created by interviewing the system architects, which has its costs.

4.5 Interpret the results

When the scenario evaluation has been finished we need to interpret the results to draw our conclusions concerning the software architecture. At this stage we go back to our architecture design approach where we wondered if this architecture had sufficient support for usability. The interpretation of the results depends entirely on the goal of the analysis and the system requirements. If the architecture proves to have sufficient support for all quality attributes the design process is ended. Otherwise we need to apply architecture transformations or design decisions to improve certain quality attribute(s). The choice to use particular transformations may be based upon results from the analysis. For example: Consider a system, which proves to have a low support for usability, for example learnability for some usage scenarios is not supported. To improve learnability we could use the design primitive of guidance, to address guidance we could implement for example a wizard pattern or provide context sensitive help. The framework we have developed is then used as an informative source for design and improvement of usability.

5. Conclusions

The work presented in this paper is motivated by the increasing realization in the software engineering community of the importance of software architecture for fulfilling quality requirements. We have presented a provisional assessment technique for usability based on scenarios, which has potential to improve current design for usability. Future case studies should determine the validity of our approach to refine it, possibly redefine and elaborate the steps that should be taken to make it generally applicable. Several issues need to be resolved during case studies, which have been summarized below:

- Relevance of framework: The relationships depicted in our framework indicate potential relationships. Further work is required to substantiate these relationships.
- Use case maps: may provide information about static properties of usability. More research is required to determine whether use case maps can provide that kind of information.

The main contribution of this paper is the formulation and derivation of an architectural assessment approach for usability.

6. References

[1] R. Kazman, G. Abowd and M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architectures", *Proceedings of the 16th International Conference on Software Engineering*, 1994, pp. 81-90

[2] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson and J. Carriere, "The Architecture Tradeoff Analysis Method", *Proceedings of ICECCS'98*, 1998

[3] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*, Pearson Education (Addison-Wesley and ACM Press).2000.

[4] IEEE Architecture Working Group. Recommended practice for architectural description. Draft IEEE Standard P1471/D4.1, IEEE.

[5] P. O. Bengtsson; N. Lassing; J. Bosch and H. van Vliet, “Architecture-Level Modifiability Analysis (ALMA)", *Conditionally Accepted for the Journal of Systems and Software*, 2002.

[6] E. Folmer and J. Bosch, "Usability patterns in Software Architecture", *Accepted for HCI International 2003*, 2003

[7] E. Folmer and J. Bosch ,"Architecting for usability; a survey", *Accepted for the Journal of systems and software*, 2002.

[8] M. Welie and H. Trætteberg, "Interaction Patterns in User Interfaces", *7th Conference on Pattern Languages of Programming (PloP)*, 2000.

[9] J. Tidwell, "Interaction Design Patterns", *Conference on Pattern Languages of Programming 1998*, 1998.

[10] J. Preece, Y. Rogers, H. Sharp, D. Benyon, S. Hollandand T. Carey, *Human-Computer Interaction*, Addison Wesley.1994.

[11] D. Hix and H. R. Hartson, *Developing User Interfaces: Ensuring Usability Through Product and Process.*, John Wiley and Sons.1993.

[12] J. Bosch and P. O. Bengtsson, "Assessing optimal software architecture maintainability", *fifth European Conference on Software Maintainability and Reengineering*, 2002

[13] P. O. Bengtsson and J. Bosch ,"An Experiment on Creating Scenario Profiles for Software Change", *special issue on Software maintenance in Annals of Software Engineering (ISSN: 1022-7091),* vol. 9 59-78, 2000.

[14] ISO 9126-1 Software engineering - Product quality - Part 1: Quality Model.

[15] B. Shackel, Usability - Context, Framework, Design and Evaluation, in *Human Factors for Informatics Usability*. B. Shackel, and S. Richardson, Cambridge University Press, 1991.

Software Engineering Overlaps with Human-Computer Interaction: A Natural Evolution

Allen E. Milewski
Monmouth University
amilewski@monmouth.edu

Abstract

It is argued that overlap between the Software Engineering and Human-Computer Interaction disciplines is part of a natural evolution that has been developing throughout the history of both fields. It is further proposed that education and training is the most effective and long-lasting solution to some of the problems of communication and efficiency that have developed. Finally, it is argued that the education curricula are already evolving to reduce these problems.

1. Introduction

The disciplines of Software Engineering and Human Computer Interaction have each evolved over the past ten years to meet the needs of their customers and the responsibilities of their work assignments. In the course of evolving, each has seen the need to evolve toward the other. Software Engineering has developed practices for understanding the needs of users and other stakeholders in order to obtain reliable requirements and has developed evolutionary process models to iterate with users during the design phase. Human-Computer Interaction Engineers, for their part, have begun to include analyses of technical platform capabilities and constraints early in their designs, and now regularly develop software prototypes for user evaluation [1, 4].

This evolved situation has many problems. The most significant one is that the two disciplines don't know enough about each other to realize that the have evolved similarly. There is often a striking lack of communication between these two disciplines despite the fact that they often work side by side on a daily basis on software projects. They use different terminology for similar activities and artifacts. In most cases, there is a loss of efficiency since two engineers are performing highly-overlapping functions, at twice the cost, when in some situations, one person could do it. And, worst of all, there is the increased chance of confusing customers and users alike when two organizations schedule interviews and two organizations handover overlapping requirements documents for validation and sign-off.

But, in spite of these problems, it is argued, the current situation is much improved over the historic relationship, and is a natural evolution of the two fields that can be, and already is, being facilitated by fairly simple changes in the training process.

It is important to remember that the introduction of users into the computing environment is not a recent development. Historically, the relationship between software "builders" and Human-Computer Interaction Professionals was almost always entirely antagonistic. There are many stories where HCI professionals needed to plead with a developer to try to get even a small usability change incorporated. The response was often, "the software can't do that", or "the impact of that change is just too large". Conversely, there are also many stories where HCI engineers attempted power-plays to force developers to make seemingly arbitrary changes that were needlessly difficult and complex.

These organizational atrocities are relatively rare in the current, evolved situation. It is more difficult now to tell an HCI engineer what the software can or cannot do because s/he is more knowledgeable in platform characteristics. And, it is more difficult to tell a Requirements Engineer that they don't know what the user needs because they, too, have likely utilized an assortment of techniques for finding out.

This would suggest that the overlap between fields, contrary to being a problem, has actually been a good thing and a natural evolution for disciplines with the mutual goal of producing effective systems. The increased breadth of knowledge in these disciplines has increased empathy and made the negotiation processes far more realistic and efficient.

2. Some Prescribed Measures may have Small Effects

But, problems of communication and efficiency certainly persist, so that it is useful to consider how they can be reduced further. Some measures have been proposed, but it is argued that they are unlikely to have major effects on current SE/HCI problems, at least when taken in isolation.

1) It will not solve the problem to combine Software Engineers and Human Computer Interaction into a single discipline or a single role in the lifecycle of a project. First, while user interface issues generally account for half (or even more) of the lines of code, there are still many application functions that have nothing directly to do with the user. Second, experience has shown the utility of having a "user advocate", semi-separated from the schedule and budget demands of the rest of the project. Finally, there is simply too much to do to combine positions and the interfaces to others in the project (systems engineers, developers, designers, marketers) are typically too numerous for one person to handle.

2) It is unlikely that creating a common, integrated process model will, by itself make the current situation more manageable. In practice, process models are chosen and adapted to fit the specifics of the environment and, often, the needs of the specific project. As such, process models are more descriptive of what happens than prescriptive of what must happen. The history of process models is one of trial and error, where processes that are developed in an ad hoc fashion in real-life projects are retained and formalized if they turn out to be effective. –same with tools --may be the only resort for people already in industry

3) It is similarly unlikely that anything will be solved by initiating a formal effort to create a common terminology for the two disciplines. Like process models, the development of a set of useful working terms has a strong grass-roots element. Of course, many of our professions' terms have been introduced by leading thinkers in the fields, but only those that are meaningful in practice remain and become popular. Besides, what set of terms would be chosen to describe overlapping concepts? Existing terms are all laden with baggage for one discipline or the other. And, the strategy of creating brand new, neutral terms inevitably complicates the situation Since the older terms are seldom dropped, the net effect is to magnify confusion by increasing the number of synonymous terms (or worse- not quite synonymous).

3. Evolution Through Education and Training

The most significant and lasting solution lies in the education and training programs for both Software Engineering and HCI. If the Academic disciplines begin to acknowledge the overlap and specifically explore it, students will enter the workforce in a better position to collaborate with the entire interdisciplinary team required for a successful software project. From this interdisciplinary education will come the abilities and attitudes required to continue the evolution of the two fields.

Several aspects of training are especially important in this context:

?? First, students in each discipline need to be exposed to and encouraged to explore the terminology of the other and decide for themselves the mapping between them and the most useful commonalities. In general, it is more useful to be aware of terminology differences than to be shoehorned into a single one.

?? Second, students need to be steeped in the problem-solving approaches of the other discipline. The pace of software projects in Industry is very fast, and most of the communication required has to be implicit. Successful collaboration requires knowing how other team-members think and approach their tasks. Courses on topics such as the Psychology of Programming and Empirical Behavioral Methods are useful in this respect.

?? Third, the issues of overlap between Software Engineering and Human-Computer Interaction need to be covered in detail from a Management standpoint. I recently cited for a friend the potential problem of separate SE and HCI requirements being delivered to the customer for validation. He, a longtime manager, asked: "what manager in his right mind would ever let that happen?". His reminder that inefficiencies come from management shortcomings was a sound one. Management Training courses need to deal with these issues.

The theme here has been that the Software Engineering and Human-Computer Interaction fields have both been evolving in a positive way and that overlap is part of that continual evolution. Similarly, the educational curricula for these fields are evolving to solve current issues of communication and efficiency. There is an increasing number of programs that acknowledge the overlap and giving their students the opportunity to explore it in detail. Interdisciplinary faculty are becoming more common. Major Software Engineering textbooks have increased the sophistication of their HCI coverage [3]. Finally, nearly every survey textbook on Usability Engineering covers Process at least in part from a traditional Software Engineering standpoint [2, 4].

Many educators in both Software Engineering and Human Computer Interaction have examples of students discovering the similarities between the fields, and the differences as well. It is gratifying that the differences they see are not the artificial differences of the historic relationship, but the differences the really exist, and ought to exist for software projec ts to be efficient and successful.

10. References

[1] Mayhew, D., *The Usability Engineering Lifecycle*, Morgan Kaufmann, San Francisco, 1999.

[2] Preece, J., Rogers, Y. and Sharp, H.., *Interaction Design, Wiley, New York*, 2002.

[3] Pressman, R.S.., *Software Engineering: A practitioner' s approach*, Fifth Edition, McGraw Hill, New York, 2001.

[4] Rosson, M.B.. and Carroll, J.M., *Usability Engineering: Scenario-Based Development of Human-Computer Interaction.*, Academic Press, London, 2002.

Undergraduate Software Engineering Curriculum Enhancement via Human-Computer Interaction

Stephanie Ludi
Software Engineering Department, Rochester Institute of Technology
sludi@mail.rit.edu

Abstract

More needs to be done to train students to deliver usable software. The current Software Engineering curriculum includes Human-Computer Interaction (HCI) topics in terms of a lecture. This paper presents how an undergraduate Software Engineering curriculum can be enhanced with HCI principles and techniques. The intent is to produce software engineers who value usable software and who can produce usable software.

Creating an atmosphere where HCI is not only presented, but expected in the student projects will provide ample opportunity for students to apply these newly acquired skills. The continued application of these skills will improve software.

1. Introduction

The usability of software, or rather the lack thereof, is at best an office nuisance or at worst the root cause of a tragic accident [2]. Some users groan and trudge through tasks while others become visibly frustrated when goals seem unattainable. The conduit to the system is the user interface, which is often a graphical user interface nowadays. Poorly designed user interfaces persist in a variety of domains.

Prospective software engineers hone their craft through coursework, including group projects. All too often the user interface design the result of one person's effort rather than an effort by the team. The success of the user interface is attributed to the efforts of one person's skill and vision. Such effort is reminiscent of the Capability Maturity Model's Initial level (CMM Level 1). CMM level 1 behavior is not acceptable in software development, and it should not be acceptable to the process of ensuring that users can interact with systems effectively and efficiently.

Many software engineers are simply not trained in HCI concepts and how to merge them with their technical training. Some students receive a lecture on user interface design principles, while others take a Human Factors course from the Industrial Engineering department. As Software Engineering programs are increasing in number, the time has come for these programs to take a serious look at merging aspects of Human-Computer Interaction into the curriculum.

2. General Curricular Revision

Revision is needed in the undergraduate Software Engineering curriculum. Two different perspectives are needed when shaping the curriculum.

2.1. The HCI Course Within the Department

In order for the Software Engineering to exert the most control over the content and objectives of the course, the offering of the HCI course within the Software Engineering department would be ideal. The ACM SIGCHI Curricula for Human-Computer Interaction Curricula can be used as a starting point [1]. Assuming that one course will be required, introductory HCI topics need to be put into the context of software development.

For a course being proposed at RIT, the course description is:

The course stresses the importance of good interfaces and the how to apply the theories of HCI to user interface design. The course surveys the techniques available in the discipline, demonstrates where and when they are applicable and proceeds to demonstrate via a combination of scientific theory understanding and engineering modeling, to the solution of design problems. Other topics include: interface quality and methods of evaluation; interface design examples; dimensions of interface variability; dialogue tools and techniques; user-centered design and task analysis; prototyping and the iterative design cycle; and prototyping tools.

The proposed course will initially be offered as a seminar. After such a course becomes a permanent part of the curriculum, it will be required for all Software

Engineering students. A sample course could consist of the topics in Table 1.

Table 1. Course topics for an HCI course in a software engineering department

Course Topics
1) Introduction to the course a) The specific disciplines that comprise HCI, and how each discipline contributes to HCI b) Examples illustrating the importance of user interface design c) The relationship of the discipline of user interface design to the science of human-computer interaction
2) Interface quality and evaluation a) Measures of user interface quality b) Methods for observation and evaluation
3) Dimensions of interface variability a) Languages, communication and interaction b) Dialogue genre; the role of metaphor c) Dialogue techniques (including menus, icons, etc.) d) User support and assistance, documentation
4) Social organization and work a) Small group dynamics b) Organizational information flow c) Models of work, workflow and cooperative activity Impact on design
5) Methodology a) Methods for capturing, analyzing and applying data at the organizational and social level of human behavior b) Problems of validity c) Questionnaire design d) Conducting surveys e) Observation
6) User-centered design and task analysis a) Relationship to software engineering design models; user-centered design, participatory design b) Socio-technical issues c) Task analysis d) Prototyping and the iterative design cycle; the evolution of designs e) The role of principles and guidelines f) Examples of designs
7) The Human Information Processor a) Description of human architecture and performance of critical subunits (e.g., memory, perception, motor skills, etc.) b) Models of human activity (e.g., GOMS models, Keystroke Level model, etc.) c) Formal specification d) Applications of model human information processor to example problems
8) User interface implementation topics a) Prototyping tools and environments b) Ergonomic issues i) Arrangement of displays and controls ii) Design for disabilities iii) Fatigue and health issues c) The role of graphic and industrial design
9) Evaluation revisited; learning from HCI research; the role of models a) A deeper look at evaluation b) Learning from HCI research; applying science to interface design c) Conducting and analyzing usability studies
10) Human-machine fit and adaptation a) Nature and theory of adaptive systems b) Theories of system adoption and methodology used to ascertain and motivate adoption c) User adaptation: ease of learning, training methods d) System adaptation to user types e) Relationship to system design
11. Beyond the Graphical User Interface

While smaller assignments are provided, a large project will be a significant part of the course. Students will apply concepts and techniques for a course-long project. In the course project, each student team will design and implement a prototype system. The main facets of the project consist of the following:

- First, the students will elicit requirements through the use of questionnaires, interviews and unobtrusive observation (likely not all of these though).
- Second, the students will analyze these requirements and create a paper design, followed by modeling and evaluation of the design with the human information processing model.
- Third, each team will create a working prototype and conduct user testing of the design.
- Fourth the feedback will be used to redesign the prototype, based on the information gathered during the evaluation.
- Lastly, each team will present their prototype to the class.

Such context will allow students to identify with the potential applications in software development. While the course topics appear to be general the discussion, examples, and assignments are presented so as to be more relevant to Software Engineering students. By contrast, the proposed course seeks to improve upon the Human Factors course taken from the Industrial Engineering department. In this course (which several Engineering departments require), HCI is only covered for one week.

While the proposed course is not presented as the ultimate example, it can serve as a starting point.

Another advantage of having an HCI course is to explore special topics with HCI that may not be able to be explored within other courses due to time constraints. Such topics include accessibility and global design issues. Many students (and professional designers) think that the users of the delivered systems are just like themselves. The diversity with the national and international community shows that this is not the case.

2.2. Integrating HCI into Software Engineering Courses

For some, the development of an HCI is not possible or likely in the immediate future. An alternative is weaving HCI topics into existing Software Engineering courses. This arrangement is also a good complement to the presence of an HCI course.

Several courses have the potential to introduce HCI concepts and/or techniques that are companions to Software Engineering concepts/techniques. The depth of the HCI material introduced should be appropriate to the level of the course.

The introductory (generally lower-division) courses provide students with a foundation to build upon during their academic career. Potential pairings of HCI topics and Software Engineering courses are in Table 2.

Table 2. Matching HCI topics with existing lower-division courses

Software Engineering Course	Sample HCI Topic
Freshman Seminar	The relationship of the discipline of user interface design and usability to the science of human-computer interaction
Introduction to Software Engineering	The specific disciplines that comprise HCI, and how each discipline contributes to HCI The psychology of using software User Interface design heuristics. Usability and its role in nonfunctional requirements.
Engineering of Software Subsystems	Relationship to software engineering design models; user-centered design, participatory design Prototyping and the iterative design cycle; the evolution of designs The role of principles and guidelines

The upper-division courses provide students the opportunity to study a phase of software development in more depth or to learn about special topics.

Table 3. Upper-division pairings between HCI topics and courses

Software Engineering Course	Sample HCI Topic
Formal Methods	Modeling human activity
Requirements & Specification	Task Analysis Participatory design Designing and analyzing data from questionnaires
Software Architecture & Design	Design of Graphical User Interfaces (layout, structure, elements) Alternative interfaces
Metrics	Usability metrics used during evaluation
Verification and Validation	User interface verification and validation techniques

3. Expectation Within the Program

While the HCI concepts and techniques are needed in the curriculum, more is still needed. The critical element is expectation.

Unfortunately many students limit the use of the techniques and concepts to the course that they are acquired. After the course, students often revert back to what technique or process has worked for them in past projects, no matter how unstructured or haphazard the technique or process is. The follow-through to HCI concepts and techniques is no exception.

In order to truly have an impact on usability, academic programs must ask students to continuously practice what is learned and to incorporate the concepts and use the acquired techniques in subsequent projects. There must be the expectation that students will apply HCI best practices along with Software Engineering best practices. The expectation builds as the students progress through their course of study.

Such expectation can be accomplished in the grading of student projects. While the main concept being conveyed by the assignment will continue to be the main focus of grading, a portion of the project grade should be allocated to those concepts that the department also deems important at the stage of the student's academic career.

For example, a student is assigned a project in a design course. The majority of the grade is for the design concepts being applied. In addition, 10% of the overall grade can include the correct application of user interface design. This is akin to the expectation that students organize and cite references in their papers.

An alternative is to not include the user interface design tasks in the grade at all. Instead the omission of the tasks (and any others deemed important) is cause for a penalty in the overall grade.

In any case, as the students build upon their knowledge base and skill set, they should be expected to apply these in subsequent projects in other courses.

4. The Role of Faculty

The successful grafting of HCI onto a Software Engineering program will not be possible without faculty support. Since many Software Engineering faculty are not trained in HCI, the gap in expertise can be addressed in different ways.

When new faculty needs to be hired, a candidate who has expertise in HCI and Software Engineering can be sought after. This would be the ideal, as such a faculty member can teach HCI-specific courses as well as Software Engineering courses in order to meet the general needs of the department. Such faculty can also play an instrumental role in enhancing the Software Engineering curriculum with HCI.

Existing faculty who do not have expertise in HCI but are interested in the field, can pursue professional development in general HCI or specialized areas within HCI that relate to current expertise such as Requirements, Design, or Process. Such faculty can also play an instrumental role in shaping curriculum, especially since he or she has more experience with the needs of the department than that of new faculty (who need some time to adjust).

Another option for a department to pursue when the desire and resources exist is to create a joint appointment. The joint-appointment can be made with another department that teaches HCI-related courses. For example a professor in Psychology or Computer Science with HCI expertise and technical experience can potentially fulfill most needs of a Software Engineering department when no other alternative exists. Such an arrangement would work best when HCI-specific course are offered rather than asking a faculty member from another department to teach a design or testing course with integrated HCI objectives. The downside to such an arrangement is the fact that the faculty member is not full-time. The part-time appointment means that the faculty member cannot devote the same amount of time needed to shape curriculum as a full-time faculty member.

5. The Role of Industry

In order for academia to take usability seriously, industry must make it clear that they believe that usability is important.

Many departments have councils consisting of representatives of local software development companies. These councils provide input on curricula in terms of trends in industry and issues that should be addressed. The departments listen to industry since it is the customer for their graduates.

When industry discusses the importance of usability in software, departments will take notice. Departments will be motivated to act more quickly if industry vocalizes the importance of usability and the need for students to be trained in HCI.

6. Summary

In order to produce software engineers who place value in producing usable software, both academia and industry must do their part. Academia needs to prepare software engineers by training them to produce usable software at every phase of development. When the academic program imparts the expectation that students apply the concepts and techniques in all courses, the students will make HCI a part of their repertoire.

Industry can impart its need for software engineers who can deliver usable software to academia. Software Engineering departments listen to industry in order to make their graduates more appealing to industry. While industry does not directly drive curriculum, the influence exists when courses are revised from year-to-year.

The time for change is here. Usable software is attainable, and software engineers are capable of delivering it when adequately trained.

7. References

[1] ACM SIGCHI Curricula for Human-Computer Interaction Curricula, available at: http://www.acm.org/sigchi/cdg/cdg3.html

[2] N. G.Leveson, and C. S. Turner, “An investigation of the Therac-25 accidents”, *Computer*, 26(7), July 1993, pp.18-41.

Improving UML Support for User Interface Design: A Metric Assessment of UML*i*

Paulo Pinheiro da Silva
Department of Computer Science, Stanford University
Gates Building, Stanford, CA 94305, USA.
E-mail: pp@ksl.stanford.edu

Norman W. Paton
Department of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, England, UK.
E-mail: norm@cs.man.ac.uk

Abstract

The Unified Modeling Language (UML) has been widely accepted by application developers, but not so much by user interface (UI) designers. For this reason, the Unified Modeling Language for Interactive Systems (UMLi) has been proposed to improve UML support for UI design. UMLi introduces a diagram notation for modeling UI presentation and extends activity diagram notation to describe collaboration between interaction objects and domain objects. This paper demonstrates using design metrics, in a quantitative way, that UMLi models are significantly structurally, behaviorally and visually less complex than standard UML models when describing the same set of properties of an interactive system.

1 INTRODUCTION

The development of techniques for constructing UI models and generating running UIs from these models has been studied since user interface management systems (UIMSs) were proposed during the 80s. More recently, model-based user interface development environments (MB-UIDEs) [6] have provided additional experience on the construction of user interface models. The use of many different and sometimes user interface specific notations is one of the well-known shortcoming of MB-UIDEs [15]. Indeed, the modest adoption of MB-UIDEs as part of software development practice can at least partly be ascribed to the difficulties associated with the integration of MB-UIDEs with mainstream application development techniques. Therefore, the idea of modeling UIs using UML [12], a standard modeling notation in widespread use for modeling mainstream systems, has attracted the attention of both HCI and software engineering communities [9, 11, 14, 16].

UML, however, suffers from its lack of support for modeling UIs [14]. For example, class diagrams are not entirely suitable for modeling UI presentation. As a result of these difficulties, research has taken place with a view to improving the effectiveness of UML for UIs. For example, Markopoulos's approach [9], Wisdom [11] and UML*i* [16] are conservative extensions of UML, while ConcurTaskTree [14] that proposes the introduction of a new notation for task modeling constructors into standard UML. By contrast, the notion of task is represented by classes in Wisdom and by activities in UML*i* and Markopoulo's approach. The existence of more than one approach to improving UI support in UML indicates the necessity of evaluating the benefits of such approaches compared with standard UML. Indeed, it is still unclear how best to improve support for UI design in the context of UML.

Qualitative evaluations based on the verification of the conformance of proposed improvements with design practices and guidelines are supported by most of these approaches. For instance, [9], [14] and [16] are approaches derived from MB-UIDEs. However, such qualitative evaluations do not identify the extent of the benefits achieved. Therefore, this paper presents, for the first time, a quantitative assessment of the benefits of extensions to UML for modeling interactive systems. This assessment is described in terms of the structural and behavioral complexity of UML*i* and UML models for interactive systems.

2 RELATED WORK

Few assessment strategies for UI designs evaluate the quality of conceptual UI models. Considering toolkits and UI builders, no conceptual specification of UIs is usually produced at all. Indeed, the produced UI codes are already concrete UI presentations, generally committed to a specific layout and a selected set of widgets. Considering UIMSs and MB-UIDEs, an interaction with generated UI prototypes and UI codes allows designers and users to evaluate the quality of a combination of UI models, techniques used to generate running UIs, and toolkits that may be used to implement the running UIs rather than the quality of the UI models in isolation. Some quality evaluation can be achieved by the simulation of these conceptual models, as in ConcurTaskTree [13]. Simulations, however, restrict assessment to the behavioral part of the models.

The quality of models, however, can be assessed from internal attributes of the models rather than from attributes of artifacts produced from them. For instance, object-oriented design metrics [1, 2, 4, 10] have been used to evaluate the quality of several aspects, such as the collaboration (coupling) between classes [3, 4], the cohesion among the operations of a class [2], and the complexity of the control-flow [10] of as object-oriented design.

Therefore, object-oriented design metrics are used in this paper to assess the benefits of using UML*i* to model an interactive system when compared with the use of standard UML. In terms of structural complexity, this study uses the suite of metrics proposed by Chidamber and Kemere (CK metrics) [4] because:

- Basili et. al. [1] have validated CK metrics for class fault-proneness [1] (disregarding the *lack of cohesion on methods* LCOM).

- Basili et. al. [1] have provided a valuable indication of the impact of UI classes (or *widgets*) on the the validation of each CK metrics. Indeed, the experimentation described there was composed of eight medium-sized interactive systems built in C++. Further, the user interfaces of those systems were built using the OSF/MOTIF toolkit. Therefore, due to the controlled nature of the experiment, it was possible to describe the impact of CK metrics on the database and user interface classes, the two categories of classes described in the paper. Furthermore, the comments in [1] are contrasted to the CK metrics of the case study in this paper.

- It has influenced many other metric proposals [2].

In terms of behavioral complexity, this study considers the McCabe's cyclomatic complexity [10] since it is a long-term well-established metric the definition of which has been naturally translated from a code-level metric to a design-level metric.

3 CASE STUDY

The modeling of a library system in both UML and UML*i* is used as the case study in this paper. This section characterizes the aspects of a generic interactive system that are commonly modeled in UIMSs and MB-UIDEs. From this characterization, a brief description of the library system modeled in the following sections is provided.

3.1 Aspects of Interactive Systems

Many aspects of an interactive systems can be described by models. Therefore, there are many possible combinations of models that can together describe an interactive system. Despite how different these models can be, structural and behavioral aspects of systems should be considered if the aim is to build UI models that can be used, for instance, to generate running user interfaces.

- *Structural models.* Classes and objects are the main structural elements of a system. Relationships between classes and objects, i.e., associations, compositions and generalizations, are also structural elements. Thus, structural models describe properties of classes, objects and their relationships.
  - *Presentation models.* Classes and objects responsible for the visual appearance of user interfaces are structural elements. Interaction objects are usually called *widgets*. Presentation models are structural models describing properties of widgets and their classes.
  - *Domain models.* Classes and objects modeling the entities of a system are elements of the domain. Many of them may be related to interaction classes and widgets, despite the fact that they are not inherently related to UIs. Thus, domain models describe properties of classes and objects of the domain.

- *Behavior models.* Dynamic elements used to alter the states of structural elements, i.e., tasks, actions, events, are behavioral elements of a generic system. Thus, behavioral models describe properties of behavioral elements.

A description of how domain, presentation and behavioral models for the library system can be built using UML and UML*i* are discussed in the following sections. First, however, the functionalities considered in the case study are described.

3.2 The Library System Case Study

Books, BookCopies and LibraryUsers classified into Borrowers and Librarians are the main entities of the library system. Considering these entities, the top-level functionalities specified for the library system are defined as follows. LibraryUsers are entitled to connect to the system (Connect use case), search for books (SearchBook use case) and check the status of a book (CheckBookStatus use case). Librarians can additionally check books in (deleting loans), check books out (creating loans), renew loans, and maintain the Book, BookCopy and LibraryUser catalogues.

Many other functionalities are not considered in the case study. For instance, users are not allowed to browse the book catalogue, to get a list of their loans or to make a book reservation. However, this simple specification has provided sufficient functionality to require the construction of quite substantial models for use in the study using metrics. Indeed, the library system must be entirely modeled in order to make the quantitative assessment a result of *the effort of building complete models of interactive systems*.

UML and UML*i* models of the library system are partially described in the following two sections. Documentation containing a complete description of these models along with the actual UML and UML*i* models is available at http://img.cs.man. ac.uk/umli/metrics. Moreover, the models can be viewed and adapted using Argo/UML*i*, which is a UML editor tool that implements the UML*i* extensions and is also publicly available from http://img.cs.man.ac.uk/umli/software.html.

4 CASE STUDY IN UML

Class diagrams are used in UML for modeling classes and their relationships. Furthermore, the popular use of class diagrams is mainly to model the *domain* of software systems.

As structural models, presentation models can be built using class diagrams, as presented in Figure 1. Unlike in the modeling of the domain model, the use of class diagram for presentation modeling is not a popular choice. Indeed, it is difficult to realize that the class diagram in Figure 1 represents the presentation of a user interface, even when many important properties such as interaction object containments in a UI are actually modeled. However, this kind of abstract UI presentation is conceptually equivalent to the abstract presentation in MB-UIDEs [6]. Further, the InteractionClass, InvokeActionClass and PrimitiveInteractiveClass classes in Figure 1 specify methods similar to those presented by Holub [7], a long-term practitioner working with UI design.

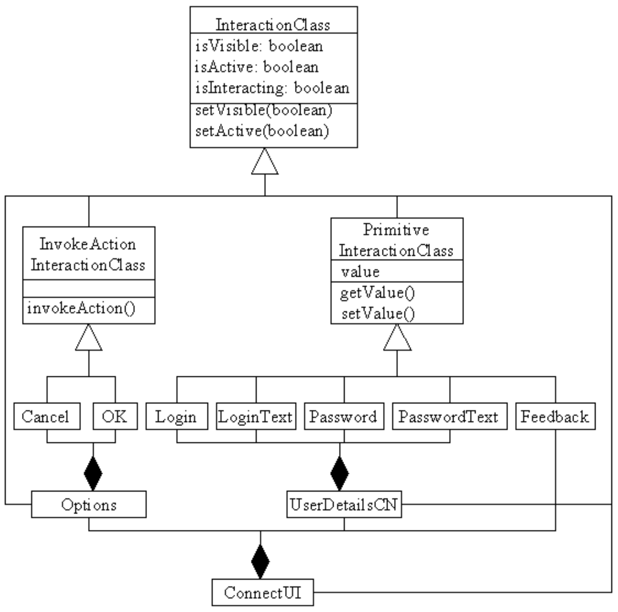


Figure 1. The ConnectUI presentation modeled in a UML class diagram.

Behavior can be modeled in UML using activity diagrams and interaction diagrams (i.e., collaboration diagram and sequence diagram). Interaction diagrams are more popular than activity diagrams for modeling behavior and specially for eliciting structural elements required to support a specific functionality. However, interaction diagrams suffer from difficulties when representing the non-sequential behaviors commonly required to model UI behaviors. In this case, activity diagrams as presented in Figure 2 can be used to describe the possible behavior of a user interface. The Connect activity in Figure 2 represents the behavior associated with the Connect use case. Therefore, for example, the cn1 object of class Cancel is made active and awaiting for an action invocation immediately after the widgets in the ConnectUI are instantiated and made visible in the InitiateConnectUI activity. Further, the uq object of class UserQuery is instantiated by the new UserQuery action state leading to the execution of the GetUserDetails activity, all of these actions happening in parallel with the activation of the cn1 object.

Although uncommon, this approach of using activity diagrams for modeling UI behavior is observed in UI designs produced by practitioners [8]. Furthermore, recalling the design guidelines of Shneiderman [17], activity diagrams allow designers to verify if UIs can *offer informative feedback*, if their sequence of actions are logically grouped in order to *yield closure*, and if they can *permit easy reversal of actions*. For instance, in Figure 2 it is specified that a user interacting with the `Connect` service can either gain access into the library system or receive "Invalid Information" feedback. Further, the `ok1` object is designed to be the last action in the `Connect` service, which provides a sense of service closure. Finally, the `cn1` object allows users to cancel an attempt to connect to the library system when the `ConnectUI` is visible.

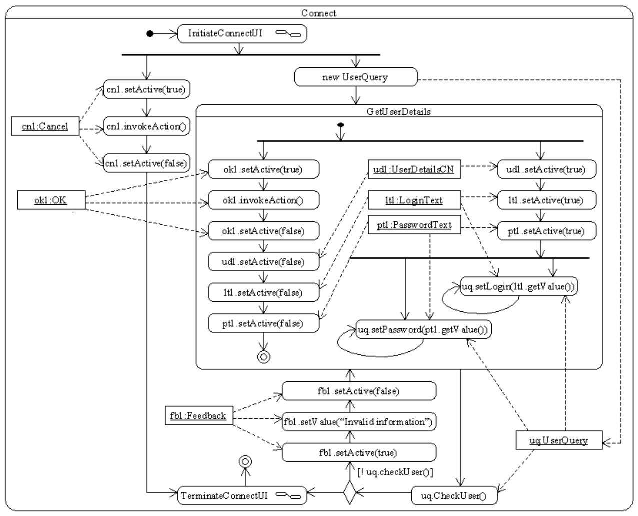


Figure 2. The behavior associated with the `ConnectUI` user interface modeled in a UML activity diagram.

Another benefit of activity diagrams is that they provide an explicit way to represent the complex relationship between structural and behavioral diagrams. Indeed, with the use of object flows as represented in Figure 2, activity diagrams can model the data flow in addition to the control flow of an interactive system.

As stated earlier, there are many different ways of modeling an interactive system. The UML model presented here is one approach which conforms with modeling techniques suggested by practitioners [7, 8] as well as with design guidelines suggested by HCI experts [17].

5 CASE STUDY IN UML*i*

Some of the limitations of UML, such as the difficulty of visualizing widget containment in Figure 1, are apparent. To cope with these limitations, UML*i* proposes some extensions to UML. The *user interface diagram*, a specialized version of the class diagram, aims to support the design of UI presentations. Figure 3 presents a user interface diagram for the presentation modeled in Figure 1. There, the dashed cube, ⬚, is a `FreeContainer` representing

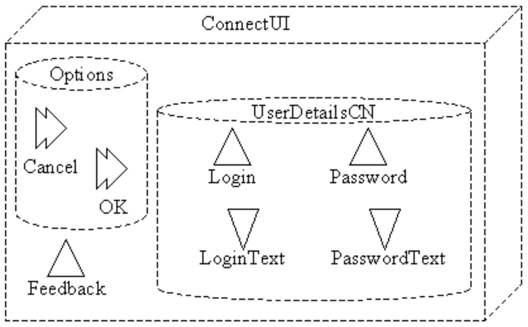


Figure 3. The `ConnectUI` presentation modeled in a UML*i* user interface diagram.

a top-level interaction object which cannot be contained by any other interaction object, e.g. a top-level window. The dashed cylinders, ⛁, are `Containers`, which can group interaction objects that are not FreeContainers. The downward triangles, ∇, are `Inputters`, which are responsible for receiving information from users. The upward triangles, △, are `Displayers`, which are responsible for sending visual information to users. The pairs of semi-overlapped triangles pointing to the right, ⊳ are `ActionInvokers`, which are responsible for receiving information from users in the form of events. Essentially, the user interface diagram notation provides two benefits when compared with class diagrams. Firstly, it emphasizes the notion of containment among interaction objects. Secondly, it identifies the abstract roles that widgets are playing in a user interface presentation.

The modeling of UI behavior is simplified in UML*i* by extending the notation of activity diagrams, as presented in Figure 4. For example, the `order independent` Selection State, ⊖, simplifies the modeling of a state where users have a range of choices (or *selectable states*) that can be performed however many times required, if any. Then, the action states `uq.setLogin(getValue())` and `uq.setPassword(getValue())` are the selectable states for the selection state in Figure 4. Further, these selectable states correspond to the `uq.setLogin(lt1.getValue())` and `uq.set Password(pt1.getValue())` action states respectively in Figure 2. Moreover, the pattern of action states, transitions and forks within the `GetUserDetails` in Figure 2 corresponds to the `order independent` Selection State in Figure 4. Finally, there is a set of powerful stereotypes, e.g., ≪*presents*≫, ≪*cancels*≫, ≪*interacts*≫, and ≪*confirms*≫, based on the use of interaction object flows that reduce the necessity of modeling action states related to the process of making widgets visible, invisible, active and inactive. A description of these stereotypes in terms of UML constructors is presented later in the paper.

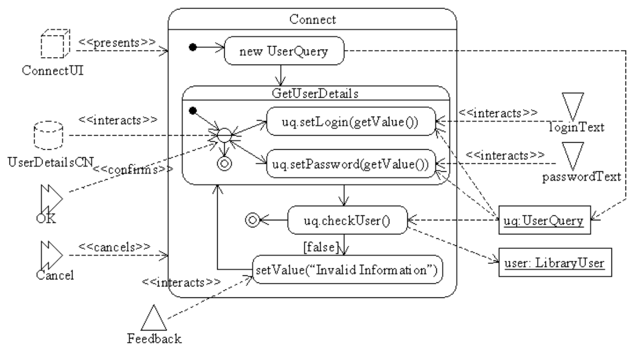


Figure 4. The behavior associated with the `ConnectUI` user interface modeled in a UML*i* activity diagram.

6 DESIGN METRICS

This section describes the metrics selected to assess the models produced. It also describes the process of translating UML*i* into UML in order to produce a pair of models specifying an *equivalent* set of properties of the library system.

6.1 Selected Metrics

Structure, behavior and visual appearance are the three dimensions of interactive system designs considered in this case study.

Structural Complexity. CK metrics [4] is considered in this paper for measuring structural complexity. However, from the CK metrics, only the CBO and RFC metrics, defined as follows, are mentioned in this paper. In fact, according to the study in this paper, the other CK metrics were not significantly affected when modeling UIs using standard UML.

- *Coupling Between Object Classes (CBO)* is defined as the number of classes to which a class is coupled. This metric measures the complexity of modifying and testing a class in relation to other classes. The assumption regarding this metric is that highly coupled classes are more fault-prone than weakly coupled classes.

- *Response For a Class (RFC)* is defined as the number of methods that can be executed in response to a message received by an object of that class. This metric measures the number of activities and action states reached by transitions triggered by operations of a class. The assumption regarding this metric is that the larger the response set of a class, the higher is the complexity of the class, and consequently the more fault-prone and difficult to modify is the class.

Behavioral Complexity. McCabe's cyclomatic complexity (CC) metric [10] is defined as the number of decisions (or predicates) specified in models, plus 1. Since decisions are specified in behavioral models, this is a metric for behavioral complexity. The reasoning behind this metric is that CC corresponds to the number of possible execution paths specified in the models. The assumption regarding this metric is that models with high CC are more difficult to understand and consequently maintain than models with low CC.

Visual Complexity. There is no well-established metric for measuring the complexity of visual languages [5]. In the case of Argo/UML*i*, the diagrams can be stored and exchanged using the Precision Graphics Markup Language (PGML) format. Therefore, as PGML is a textual representation for the diagrams in Argo/UML*i*, the number of lines of code (LOC) of the PGML files is the metric for measuring the size of the UML and UML*i* diagrams in this study. Size, however, may not be an appropriate metric for visual complexity since designs with long textual representations can be very simple ones. Therefore, the following metrics are used in this paper for measuring the visual complexity of the models.

- *Density of coupling between objects in diagrams (DCBOD).* This is defined as the ratio of the level of CBO in the models and the total number of LOC of the PGML files representing the diagrams. The non-validated assumption is that high densities indicate that more relevant structural specification, e.g., structural complexity, can be represented by fewer graphical elements than with low densities.

- *Density of cyclomatic complexity in diagrams (DCCD).* This is defined as the ratio of the cyclomatic complexity in the models and the total number of LOC of the PGML files representing the diagrams. The non-validated assumption is that high densities indicate that more relevant behavioral specification, e.g., behavioral complexity, can be represented by fewer graphical elements than with low densities.

6.2 Using the Design Metrics

Measuring the metrics in the models of the library system is a straightforward task. The major concern regarding the use of these design metrics is the production of models using two different notations to model a common set of properties from the specification of a system. Therefore, a systematic mapping strategy should be defined in order to ensure that the same reuse strategy is used in both models.

The diagrams in Figures 3 and 4 correspond to the modeling of the `Connect` use case using UML*i*, while the diagrams in Figures 1 and 2 correspond to the systematic translation of the UML*i* models into UML models. Therefore, the mapping rules from UML*i* into UML can be explained with reference to the Figures 1, 2, 3 and 4.

InteractionClasses to Classes. `InteractionClass`, `InvokeActionInteractionClass` and `PrimitiveInteractionClass` in Figure 1 are classes of the UML*i* metamodel used to represent the `InteractionClasses`, viz., `FreeContainers`, `Containers`, `Inputters`, `Displayers`, `Editors` and `ActionInvokers`. Therefore, the `InteractionClasses` in Figure 3 are mapped into `Classes` in Figure 1. This is a natural mapping since the UML*i* `InteractionClass` is a subclass of UML `Class`. As a consequence of this mapping we can observe the following.

- *Placement to Composition*. In UML*i*, an `Interaction-Object` which is not a `FreeContainer` must be associated with a `Container`. Thus, the association of an *InteractionObject* to its `Container` is specified in UML*i* by the placement of the `InteractionObject` into the `Container` in the diagram. In UML, however, this association is specified by a composition. Therefore, a composition is created from each `InteractionClass` in Figure 1 that is not a `FreeContainer` to its immediate `Container`. For instance, `Cancel` is placed in `Options` in Figure 3, so there is a composition between `Cancel` and `Options` in Figure 1.

- *Visible(), Active() and InvokeAction() operations to explicit class methods*. These operations originally embedded in the UML*i* metamodel are explicitly modeled in the UML presentation model in Figure 1.

Interaction Object Flow to Object Flow. Interaction object flows in Figure 4 were translated into object flows in Figure 2. Despite missing the information as to which role each object can play in a UI, this is a natural mapping since `InteractionObject` is a subclass of `Object` in the UML*i* metamodel.

Interaction object flow stereotypes into fragments of a standard activity diagram. There is no one-to-one mapping between UML*i* constructors and UML constructors. Thus, the mapping of each stereotype must be individually explained.

- The ≪*Presents*≫ stereotype in Figure 4 specifies that the `ConnectUI` FreeContainer is a presentation unit. This means that the widgets directly and indirectly contained by `ConnectUI` must be instantiated (if not previously explicitly instantiated) and must be made visible when the `Connect` activity is reached. Further, these widgets must be made invisible and destroyed when the `Connect` activity is left.

- The ≪*Cancels*≫ stereotype in Figure 4 specifies that the `Cancel` ActionInvoker is active and can finish the `Connect` activity anytime when the control-flow is there. This behavior is modeled in Figure 2 by the `Cancel` object that is made active immediately after the instantiation of `ConnectUI`.

- The ≪*Interacts*≫ stereotype can be associated with a `Container` or `Primitive-InteractionObject`. If the stereotype is associated with a `Container`, this means that contained interaction objects are made active when the associated activity or action state is reached. If associated to a `PrimitiveInteractionObject`, this means that the object is made active (if its `Container` was not previously activated by another ≪*Interacts*≫) and ready to interact with a user through the `getValue()` and `setValue()` operations.

- The `OrderIndependent` SelectionState in Figure 4 corresponds to the fork with each selectable action being recursively invoked, as presented in Figure 2. Thus, the ≪*Confirms*≫ stereotype in Figure 4 is mapped into three sequential action states associated to the `OK` object in Figure 2. The first action state activates the associated `ActionInvoker`, the second waits for the performance of the `invokeAction()`, and the last deactivates the `ActionInvoker`. Therefore, the performance of the `invokeAction()` is responsible for *confirming* the end of the selection state in Figure 2.

- The ≪*Activates*≫ stereotype, not used in Figure 4, is mapped into three sequential action states, as in ≪*Confirms*≫. The important characteristic of ≪*Activates*≫, though, is that these sequential action states are placed before the activity/action state to be triggered by the associated ActionInvoker.

7 RESULTS

Two set of files containing a textual representation of the UML and UML*i* models were produced using Argo/UML*i*.

In terms of LOC of these sets, the consolidated size of the textual representations of the models and diagrams are 3.44 times higher and 4.01 times higher in UML than in UML*i* respectively. Size usually does not say much about how difficult it is to construct model of an interactive system or how difficult it is to understand such models. By contrast, the metrics in this paper are designed to quantify the inherent difficulties of constructing and understanding models, since they measure many dimensions of the complexity associated with the models. Table 1 presents the metrics for the UML and UML*i* models. The following results can be observed analyzing Table 1.

Model		Structural Complexity		Behav. Cmplx.	Visual Complexity	
	Data	CBO	RFC	CC	DCBOD	DCCD
UML	Mean	2.7582	6.7582	77	0.0019	0.0006
	Sum	251	615			
UML*i*	Mean	2.8523	1.1818	66	0.0080	0.0021
	Sum	251	104			
UML/ UML*i*	Mean	0.9670	5.7185	1.1667	0.2484	0.2891
	Sum	1.0	5.9135			

Table 1. Design metrics of the UML and UML*i* models of the library system.

- The total of CBO is same in both models. The difference in the mean is due to the abstract `InteractionClass`, `InvokeAction-InteractionClass` and `Primi-tiveInteractionClass` classes which are explicitly specified in the UML models and implicitly specified in the UML*i* metamodel. This means that CBO is not affected at all by UML*i* even with a model corresponding to 29% the size of the UML model.

- RFC has been reduced 4.88 times when using UML*i* since much of the behavior in UML*i* classes is embedded within the UML*i* constructs, making the models more straightforward and easier to maintain. As a result of such improvement, Figure 5 presents a graphical representation of the distribution of RFC per number of classes. In this distribution, small areas are better than big areas since low RFCs are better than high RFCs. Further, according to [4], the reduction of RFC indicates that UML*i* classes are less fault-prone and easier to maintain than UML classes. Therefore, *the reduction in RFC is a significant achievement of UMLi*.

- CC is 16% higher in UML than in UML*i*. This is due to the ≪*cancels*≫ and ≪*confirms*≫ stereotypes that eliminate the necessity of specifying how activities can be canceled by users and how *option selection states* can be confirmed by users. This reduction may have a significant impact in the design of large-scale interactive systems since it provides an uniform treatment for the canceling of tasks in order to allow users to reverse their actions [17].

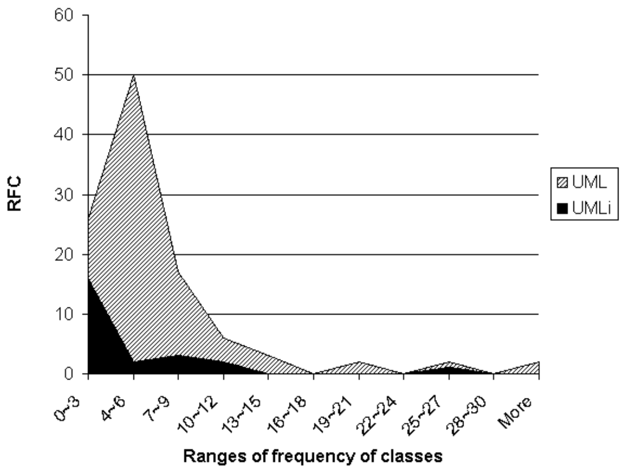


Figure 5. Distribution histogram comparing the RFC of the UML and UML*i* models.

- DCBOD is 3 times lower and DCCD is 2.45 times higher in UML than in UML*i*. This indicates that the capability of UML*i* to visually represent both structural and behavioral complexity is higher than that of UML. Thus, with a fixed number of graphical elements, UML*i* should be able to represent more relevant information than UML.

From a metric point of view there is no trade-off involved in the use of UML*i* compared with UML for modeling interactive systems. In fact, no metric has demonstrated any disadvantage to the use of UML*i* for modeling interactive systems when compared with UML.

8 CONCLUSIONS

This study has demonstrated in a quantitative way that the UML*i* proposal [16] is an improvement on UML for UI design. Further, it has demonstrated that UML can usefully be enhanced for modeling UIs, as qualitatively identified in studies on the use of UML for modeling UIs [14]. In fact, significant reductions in structural complexity (reduction of 87% in RFC) and visual complexity (increase of 402% in DCBOD and 345% in DCCD) along with considerable reductions in behavioral complexity (reduction of 14% in cyclomatic complexity) are achieved in UML*i* models by improving the support of UML for UI design. These metric improvements, according to their validated assumptions, mean that the construction and maintenance of models of interactive systems should be simpler and easier in UML*i* than in UML. Indeed, the introduction of the user interface diagram with its interaction classes has simplified the visualization of UI presentations. The introduction of the interaction object flow with its stereotypes has simplified the modeling of actions related to UI widgets. The introduction of selection states has simplified the modeling of behaviors usually observed in UIs.

As well as providing a systematic comparison of UML and UML*i*, the paper establishes a foundation for a productive discussion on how to assess UML extensions for UI design using design metrics. Any other proposal to extend UML for UI design, e.g., [9], [11] and [14], can be contrasted, for example, with UML*i*, by comparing the metrics presented with those for standard UML. This approach would allow researchers to systematically identify the best combination of improvements in UML to support UI design.

Finally, this paper has demonstrated that well established object-oriented design metrics can contribute to the evaluation of user interface designs early in the process of developing UI software. For instance, using design metrics, UI models in UIMSs and MB-UIDEs could be evaluated before any running UI is produced from them. Indeed, user interface design attributes can be directly evaluated from their internal attributes rather than from any other artifact, e.g., UI prototypes or UI code, generated from them.

References

[1] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Trans. Software Engineering*, 22(10):751–761, October 1996.

[2] James M. Bieman and Byung-Kyoo Kang. Measuring Design-level Cohesion. *IEEE Trans. Software Engineering*, 24(2):111–124, February 1998.

[3] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. Defining and Validating Measures for Object-Based High-Level Design. *IEEE Trans. Software Engineering*, 25(5):722–743, September/October 1999.

[4] Shyam R. Chidamber and Chris F. Kemerer. A Metric Suite for Object Oriented Design. *IEEE Trans. Software Engineering*, 20(6):476–493, 1994.

[5] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thonson, London, UK, second edition, 1996.

[6] Tony Griffiths, Peter J. Barclay, Norman W. Paton, Jo McKirdy, Jessie B. Kennedy, Philip D. Gray, Richard Cooper, Carole A. Goble, and Paulo Pinheiro da Silva. Teallach: A Model-Based User Interface Development Environment for Object Databases. *Interacting with Computers*, 14(1):31–68, December 2001.

[7] Allen I. Holub. Building User Interfaces for Object-Oriented Systems. *JavaWorld*, July-November 1999, January 2000, March 2000.

[8] Ari Jaaksi, Juha-Markus Aalto, Ari Aalto, and Kimmo Vättö. *Tried & True Object Development: Practical Approaches with UML*. Cambrigdge University Press, Cambridge, UK, 1999.

[9] Panos Markopoulos and Peter Marijnissen. UML as a representation for Interaction Designs. In *Proceedings of OZCHI 2000*, pages 240–249, 2000.

[10] Thomas J. McCabe and Charles W. Butler. Design Complexity Measurement and Testing. *Communications of the ACM*, 32(12):1415–1425, December 1989.

[11] Nuno J. Nunes and João Falcão e Cunha. Towards a UML profile for interaction design: the Wisdom approach. In *Proceeding of UML2000*, volume 1939 of *LNCS*, pages 101–116, York, UK, 2000. Springer.

[12] Object Management Group. *OMG Unified Modeling Language Specification*, June 1999. Version 1.3.

[13] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer, Berlin, 1999.

[14] Fabio Paternò. Towards a UML for Interactive Systems. In *Proceedings of EHCI2001*, LNCS, pages 7–18, Toronto, Canada, May 2001. Springer.

[15] Paulo Pinheiro da Silva. User Interface Declarative Models and Development Environments: A Survey. In *Proceedings of DSV-IS2000*, volume 1946 of *LNCS*, pages 207–226, Limerick, Ireland, June 2000. Springer-Verlag.

[16] Paulo Pinheiro da Silva and Norman W. Paton. UMLi: The Unified Modeling Language for Interactive Applications. In *Proceedings of UML2000*, volume 1939 of *LNCS*, pages 117–132, York, UK, October 2000. Springer.

[17] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, Reading, MA, third edition, 1997.

Interaction Modeling as a Binding Thread in the Software Development Process

Simone Diniz Junqueira Barbosa, Maíra Greco de Paula
Departamento de Informática, PUC-Rio
R. Marquês de São Vicente, 225
Rio de Janeiro, RJ, Brazil – 22453-900
{simone, mgreco}@inf.puc-rio.br

Abstract

This paper proposes the use of an interaction modeling language called MoLIC to graphically represent scenarios as an additional resource in software development. MoLIC brings human-computer interaction (HCI) concerns to software engineering processes. It does so by representing potential user-system interaction paths, which will not only allow software designers to make decisions about the HCI aspects of software, but also provide a blueprint — from the users' point of view — of what needs to be implemented and tested, and how users should perceive it. Using MoLIC as an input artifact, software engineering techniques may be used to decide how to build the software that will make it possible for the represented interaction to happen. As such, MoLIC provides a basis not only for communication and understanding among team members, but also as a concrete resource for software design and development.

1. Introduction

The value of using scenarios in software development has been investigated for almost a decade [2]. In the area of human-computer interaction (HCI), it has been extensively used [3], especially in the preliminary stages of software design.

A few attempts have been made to use scenarios as a representation for bringing HCI aspects to software engineering processes [6]. However, as the collection of natural language scenarios gets larger for a single application, it becomes harder to make consistent decisions about what must be done. In addition, due to frequent ambiguities found in natural language texts, there are too many decisions to be made in moving from scenarios to software specification. These decisions are seldom recorded for future reference or verification of the final product. Unfortunately, more often than not what is developed is very different from what has been represented in the scenarios.

So, in order to maximize the benefits brought about by using scenarios, we need to reduce the gap between scenario construction and software specification. To achieve this, we propose to use an intermediate representation language called MoLIC. MoLIC stands for "Modeling Language for Interaction as Conversation". It is a design language devised under the semiotic engineering theory of human-computer interaction.

2. Semiotic Engineering and Scenarios

MoLIC is rooted in semiotic engineering [5], a theory of HCI which views the user interface as a metamessage sent from designers to users. This message is designed in such a way as to be capable of exchanging messages with users, i.e., allowing human-system interaction. In semiotic engineering, interaction design is viewed as conversation design. This conversation is of a unique kind, however, because the designer is no longer there at the time it is carried out. This means that he/she must anticipate every possible conversation that may occur, and embed in the system his/her "deputy": the designer's spokesman in the system, with whom users will communicate during interaction. We have devised MoLIC, "Modeling Language for Interaction as Conversation", in order to support the creation of this deputy, i.e., to design the whole range of interactions that may take place in the application. Our goal is to support the designer's reflective and decision-making processes about his/her design throughout the design process [11], under a semiotic engineering perspective.

One way to support the creation of the designer-to-user message is to help designers express *what* they want to say, before going into *how* the designer's deputy will say it. This what–how coupling may be achieved by using scenarios [2] together with MoLIC.

Scenarios can be used throughout the development process, starting from the analysis of the domain and the users' tasks and characteristics. A major goal of using scenarios at this stage is to explore or confirm, together with the users, the designers' understanding of the goals and tasks to be supported. By means of scenarios, designers not only learn about the users' vocabulary and thought processes, but they have a chance to make this knowledge explicit in a language users fully understand, so it can be verified by users themselves. Scenarios are also very useful in uncovering and exploring typical and atypical courses of action in specific circumstances.

It is important to notice that designers should avoid including in the early scenarios references to concrete interface elements, such as text elements and widgets. By avoiding an early commitment and raising user's expectations about the interface solution that will be adopted, both designers and users will be open to explore alternative solutions at later stages.

One of the major advantages of using scenarios in HCI is to provide a detailed setting for better understanding and conceiving contextualized pieces of user–system interaction. Taken from another perspective, however, the contextualized and detailed nature of scenarios may be, to some extent, a drawback in HCI design. When using scenarios alone, designers may have difficulty in grasping a global view of the application as a whole system, and also in understanding the interrelations between the tasks and goals it should support. This limitation hinders the design of a coherent communicative artifact, an essential requirement for the Semiotic Engineering of Human-Computer Interaction.

In order to fill this gap in HCI design, we propose to complement scenarios with MoLIC. MoLIC was conceived to be applied between the initial analysis and the interface specification phases. It has shown to be useful in helping designers grasp a global view of the conversations that comprise the application, and thus design a coherent designer-to-users message, keeping in mind the communicative role of the designer's deputy.

MoLIC may be viewed a first step towards user interface specification. It allows the representation of users' goals and steps to achieve them, as well as the information to be exchanged between users and the designer's deputy at each step. The focus is always on the communicative capacities of both user and designer's deputy: interaction occurs only as a result of the communication between these two agents.

When writing scenarios, the designer should have in mind what the purpose of each scenario is: to investigate certain aspects of a domain or socio-cultural environment, to propose a task structure or an interaction solution, to contrast alternative task structures or interaction solutions, and so on. These purposes should be made explicit when creating scenarios. One way to achieve this is through a set of questions that are expected to be answered by the users' feedback for each scenario.

Another benefit from asking questions based on scenarios is to uncover hidden information, so that the signs[1] (what) and conversations about signs (how to) are made explicit. In semiotic engineering, investigating "how to" means not only supporting users in how to manipulate signs, but also how to understand them in context. In addition, we propose to include a set of questions to investigate the motives underlying each decision the user will be making during interaction (why), in an attempt to capture the design rationale. The answers to these questions will provide valuable information for the designer to proceed. Some of these answers may generate further questions, which in turn may give rise to unanticipated scenarios. Viewed under this perspective, this approach is similar to systematic questioning as illustrated in [4].

As a running example, we will use an application designed using scenarios and MoLIC. It is an extended annotation system for a small research group. Every user may post documents and annotations to documents in order to start discussions about them. Users may belong to one or more groups, according to common interests, level of expertise in certain research areas, or some other user-defined criteria. Users may also evaluate documents and annotations, recommend documents to other users, and change information about their profile.

A sample scenario would be the following:

Carol has just joined the graduate program at UC. In order to get to know the work of her research group, she decides to access the annotation system they use to discuss about their ongoing work and published papers, as well as relevant papers from the international research community. As she enters the system, she realizes she doesn't have a password, and is glad that there is a possibility of logging into the system as a guest [Q1]. She searches for the documents in her area of interest [Q2], views some of the papers being currently discussed and decides to print two of them to read at home [Q3]. She sends a request for a password, so that in the future she will be able to actively participate in the discussions [Q4].

Some of the questions regarding this scenario excerpt might be:

Q1 Who can access the system? (Should the system be available to the general public? If so, which parts, and why?)

Q2 How do users like or need documents to be organized? What is the purpose of document classification?

[1] The most widely used definition of signs in Semiotics is: "anything that stands for something to someone" [10]. We use signs here to stand for the whole range of representations used by designers and users, from conceptual information to user interface elements.

Q3 How can documents be viewed? (Are users online all the time? Should the system have both online and printable versions of documents?)

Q4 Should registration in the system be automatic? Who can register, and how?

From the scenarios, users' goals are identified and may be structured in a hierarchical goal diagram. A user's goal extracted from the sample scenario is, for instance, "searching documents". As we will see in the next section, for each identified goal, a piece of interaction model is built in such a way as to be interrelated with the paths of interaction corresponding to other goals, aiming to form a coherent whole.

3. Using MoLIC in Software Design

MoLIC comprises three interrelated representations: a diagram of users' goals, an ontology of the domain and application signs, and an interaction model.

3.1 User Goals

The first step in using MoLIC is to extract, from the scenarios, the top-level goals users may have when using the application, i.e., goals that are explicitly supported by it. These goals are then organized in a hierarchical diagram, according to some classification the designer finds useful: what are the classes of users that will form this goal; how primary is the goal with respect to the system scope definition (or whether it is just a supporting goal); the frequency with which the goal is expected to be achieved, and so on.

3.2 Domain and Application Signs

The next thing to extract from the scenarios are the domain and application signs that are meaningful to users. While these signs are usually treated as data, in semiotic engineering they acquire a new status, going further than establishing the vocabulary to be shared between designer's deputy and users. Defining domain and application *sign systems* help designers uncover users' expectations about how knowledge should be shaped and built. Signs allow designers to establish what the system is all about, and how users will be able to understand and use it.

A widespread example of a simple sign system is the use of ontologies to organize domain and application concepts [1]. A designer should be able to straightforwardly derive an ontology from usage scenarios. However, this is seldom the case. Scenarios are often incomplete and ambiguous, and it is hard to keep the whole set of scenarios consistent and coherent. Thus, in order to build ontologies that define the application signs, designers should explore as many dimensions or classifications of signs as necessary to grasp their full meaning.

As in typical data classification, signs may be grouped according to the kind of information they may have as a value. The most simple classification is probably the one that distinguishes only between textual and numeric values. In order to be useful for HCI design, this classification needs to be complemented with knowledge about the range of values these signs may assume. For instance, a user interface element used for providing unconstrained textual information is different from that for providing a piece of textual information that belongs to a known set of values.

An interesting classification is related to the degree of familiarity to a sign users are expected to have. In this classification, domain signs are those directly transported from the users' world, such as "full name". Application signs, on the other extreme of the scale, are those that were introduced by an application, and have no meaning outside it. Still in this classification, there is an intermediary kind of sign: a transformed sign is derived from an existing sign in the world, but has undergone some kind of transformation when transported to the application. This transformation is often related to an analogy or metaphor.

The reason for this classification to be interesting in HCI is that different kinds of signs may require different kinds of user interface elements to support users. In general, a domain sign would require an explanation only inasmuch there are constraints imposed by the application. For example, the concept of "full name" is clear to users, but a restriction about the number of characters allowed in the corresponding piece of information might not be, and thus need some clarification from the designer's deputy. A transformed sign would require an explanation about the boundaries of the analogy. For example, a folder in the desktop metaphor might require an explanation about its "never" getting full, except when the hard disk which contains it lacks space. At the end of the scale, an application sign may require a complete explanation about its purpose, utility and the operations that can manipulate it. An example might be a sign for "zooming" in a graphics application.

There are of course some signs that can be classified in either group. For example, a password may be thought of as a transported sign, derived from the existing domain sign: signature. In these cases, it is the designer's responsibility to decide, based on the analyzed characteristics of users and their tasks, the amount of support to provide in order to have users fully understand each sign.

It is important to note that users may become familiar with certain signs in one application and then transport this knowledge to another application. When this is done unsuspectingly, however, it may cause unpredictable distortions in the users' conceptual model of the latter application.

There is also the typical input/output classification, which establishes who will be responsible for manipulating the sign at a certain point during interaction: the user or the system (via the designer's deputy). This classification, however, changes during interaction, according to the user's current task, and thus may be considered a task-dependent property of the sign. Task-dependent signs will be explored in the next section, in which we describe MOLIC's interaction notation.

After having established these diverse sign classifications, designers can follow traditional ontology engineering techniques to represent the acquired knowledge.

3.3 Interaction Modeling

From the user-approved scenarios and their corresponding signs, HCI designers have enough elements to build an interaction model and thus shape the computational solution.

When interaction is viewed as conversation, an interaction model should represent the whole range of communicative exchanges that may take place between users and the designer's deputy. In these conversations, designers establish *when* users can "talk about" the signs we extracted from the scenarios. The designer should clearly convey to users *when* they can talk about *what*, and what kinds of *response* to expect from the designer's deputy. Although designers attempt to meet users' needs and preferences as learned during user, task and contextual analyses, designing involves trade offs between solution strategies. As a consequence, users must be informed about the compromises that have been made. For instance, MoLIC allows the representation of different ways to achieve a certain result, criteria to choose one from among them, and of what happens when things go wrong.

MoLIC supports the view of interaction as conversation by promoting reflection about how the design decisions made at this step will be conveyed to users through the interface, i.e., how the designers' decisions will affect users in their perception of the interface, in building a usage model compatible with the designers', and in performing the desired actions at the interface. This model has a dual representation: both an abbreviated and an extended diagrammatic views. The goal of the diagrammatic interaction model is to represent all of the potential conversations that may take place between user and system, giving designers an overview of the interactive discourse as a whole.

The interaction model comprises scenes, system processes, and transitions between them. A scene represents a user–deputy conversation about a certain matter or topic, in which it is the user's "turn" to make a decision about where the conversation is going. This conversation may comprise one or more dialogues, and each dialogue is composed of one or more user/deputy utterances, organized in conversational pairs. In other words, a scene represents a certain stage during execution where user–system interaction may take place. In a GUI, for instance, it may be mapped onto a structured interface component, such as window or dialog box, or a page in HTML. In the diagrammatic representation, a scene is represented by a rounded rectangle, whose text describes the topics of the dialogues that may occur in it, from the users' point-of-view (for instance: `Search documents`). Figure 1 illustrates the representation of a scene.

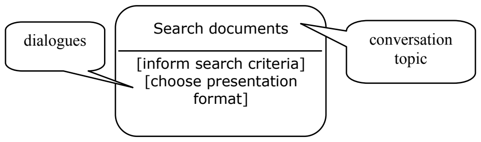


Figure 1. Diagrammatic representation of scene "Search documents".

The signs that make up the topic of dialogues and scenes may also be included in an extended representation. When this is the case, we adopt the following convention: when a sign is uttered by the deputy, i.e., presented to the user, it is represented by the sign name followed by an exclamation mark (e.g. `date!`); whereas a sign about which the user must say something about, i.e., manipulated by the user, is represented by a name followed by an interrogation mark (`login?`, `password?`). Figure 2 illustrates the extended representation of a scene[2].

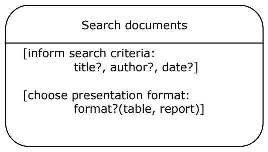


Figure 2. Extended scene representation including signs

A system process is represented by a black rectangle, representing something users do not perceive directly. By doing do, we encourage the careful representation of the deputy's utterances about the result of system processing, as the only means to inform users about what occurs during interaction.

[2] The detailed specification of the extended representation is found in [8]

Transitions represent changes in topic. This may happen due to the user's choice or to a result in system processing. Transitions are represented by labeled arrows. An outgoing transition from a scene represents a user's utterance that causes the transition (represented by a bracketed label, such as u: `[search document]`), whereas an outgoing transition from a process represents the result of the processing as it will be "told" by the designer's deputy (represented by a simple label, such as: `d:document(s) found`). In case there are pre-conditions for the transition to be made, they should come before the transition label, following the keyword `pre:`. Analogously, the `post:` keyword after the label is used to mark a post-condition, or a new situation or application state that affects interaction.

Some scenes may be accessed from any point in the application, i.e., from any other scene. The access to these scenes, named ubiquitous access, is represented by a transition from a grayed scene which contains a number following the letter U, for "ubiquitous". Moreover, there are portions of the interaction that may be reused in various diagrams. Stereotypes are created to represent parameterized interaction diagrams, represented by a rounded rectangle with double borders (such as `View(document)`).

In semiotic engineering, error prevention and handling are an inherent part of the conversation between users and system, and not viewed as an *exception*-handling mechanism. The designer should convey to users not only how to perform their tasks under normal conditions, but also how to avoid or deal with mistaken or unsuccessful situations. Some of these situations may be detected or predicted during interaction modeling. When this is the case, we extend the diagrammatic representation with breakdown tags, classifying the interaction mechanisms for dealing with potential or actual breakdowns in one of the following categories:

Passive prevention (PP): errors that are prevented by documentation or online instructions. For instance, about which users have access to the system, what is the nature of the information expected (and not just "format" of information).

Active prevention (AP): errors that will be actively prevented by the system. For instance, tasks that will be unavailable in certain situations. In the interface specification, this may be mapped to making widgets disabled depending on the application status or preventing the user to type in letters or symbols in numerical fields, and so on.

Supported prevention (SP): situations which the system detects as being potential errors, but whose decision is left to the user. For instance, in the user interface, they may be realized as confirmation messages, such as "File already exists. Overwrite?")

Error capture (EC): errors that are identified by the system and that should be notified to users, but for which there is no possible remedial action. For instance, when a file is corrupted.

Supported error handling (EH): errors that should be corrected by the user, with system support. For instance, presenting an error message and an opportunity for the user to correct the error.

Figure 3 presents a diagrammatic representation of the interaction model for the goal "search documents".

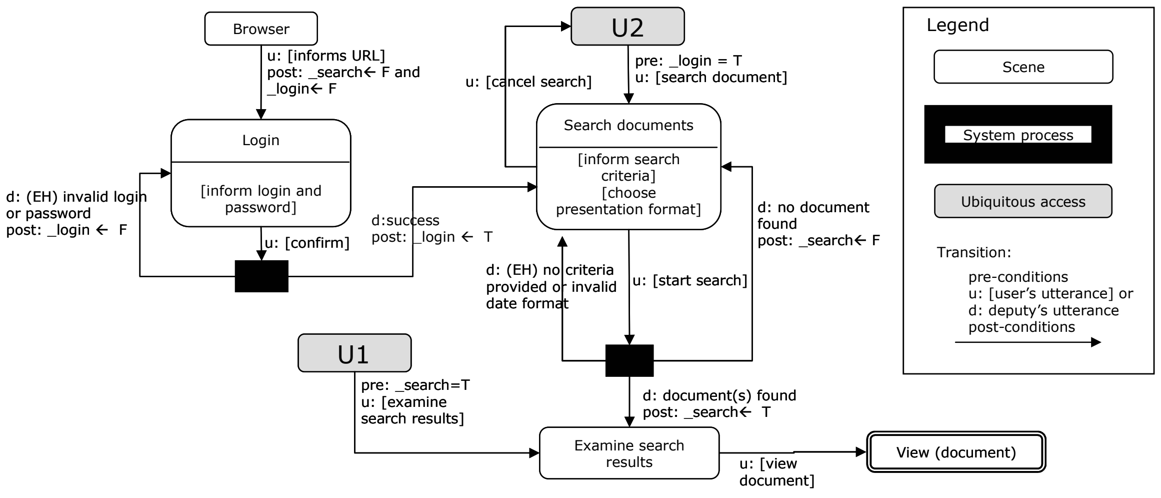


Figure 3. Sample abbreviated interaction diagram.

In order to illustrate the benefits of using MoLIC, we now exemplify two kinds of decisions made based on the diagrammatic interaction model, in two distinct situations, both related to searching:

1. Negative search result. When no document is found in a search task, the initial designer's solution was to present a message informing that no document was found, and then take the user back to the Search documents scene, where he/she could try to perform another search (Figure 4). This solution, though common in Windows applications, was inadequate in the Web, where the cost-per-click is higher. The adopted solution then was to return directly to the Search documents scene (Figure 3), which should then be modified (during execution) to reflect the source of the incoming transition: 1) a user utterance requesting a search; or 2) a deputy's response, indicating that no document was found.

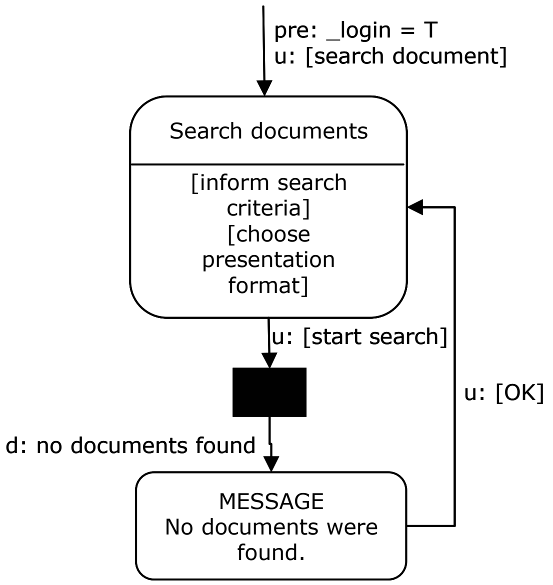


Figure 4. Alternative negative search result (no documents were found).

2. Alternative access to successful search results. When the search is successful, i.e., when there are one or more documents found, the design solution initially investigated consisted in allowing access to the search result right after the search, or from the visualization of one of the documents found (highlighted in Figure 5). However, designers and users found it useful to keep the list of found documents throughout the session, thus enabling future visits to this list (ubiquitous access to Examine search results, Figure 3). For this latter solution, it was necessary to introduce a _search control sign, which indicates whether there is a valid search result.

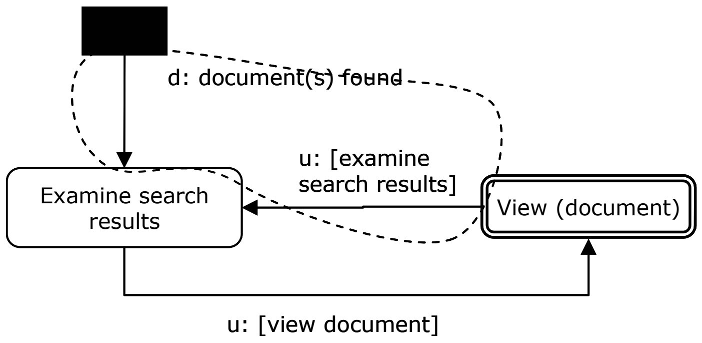


Figure 5. Alternative access strategy to search results.

The interaction model does not present details about interface or technological environment. However, designers must take into account such factors when making design decisions, for the characteristics of the computational environment in which the system will execute may determine whether a certain interaction solution is adequate or not. In other words, the *design decisions* depend on the interface style and technology, but not the *interaction representation*.

The design decisions made during interaction modeling may be presented to users for their appreciation. It may be necessary to generate new scenarios that reflect the proposed solutions, exploring the distinctions between them. For instance, in order to evaluate the users' opinions about the alternatives presented in Figures 4 and 5, scenarios were generated and presented to three users, who then worked with the designers to decide for the solutions depicted in the final interaction model (Figure 3).

4. Discussion

The goal of the work presented in this paper is to bring HCI concerns to software engineering. We argue that software engineering design models may be complemented with representations that favor the domain and task understanding (mostly scenarios), as well as the reflection about alternative solutions and interaction specification (mostly interaction model).

The use of HCI models in interactive systems design is essential for allowing designers to reflect about their solutions from the initial steps of the project on, and not just as a post-implementation evaluation. However, current HCI approaches usually tend to use complex task models which —sometimes awkwardly— intermingle issues of a different nature without calling attention to the fact that many diverse factors are at play which could be better analyzed from different perspectives. This makes it hard to understand and predict the effects of a certain solution. We believe it is best to use simpler representations, each with a well-defined focus: scenarios focusing on establishing a common ground for mutual understanding designers and users, and MoLIC focusing on the discourse structures being designed.

No method has yet been developed that is capable of efficiently taking advantage of the results from the field of HCI into software development processes. On the one hand, scenarios provide contextual information about the application, from the users' point of view, but they do not map easily into software specification. On the other hand, storyboards and static visual elements (such as dialog boxes, field, and labels) provide concrete specification, but at a level of detail too fine to be of practical use during design or to support designers' decisions about the interactive solution. In general, design decisions made at one phase to augment the software's quality of use are lost due to the lack of an adequate representation at a later phase.

We believe MoLIC is a representation suitable for bringing HCI aspects to software engineering design. It may be used to represent scenarios in a semi-formal way, composing a blueprint of the system, from the user's point of view. This blueprint can then be used in later software design phases as input to what must be implemented and tested, interaction-wise. MoLIC also makes it easier to define an application's storyboards. Roughly speaking, each scene in MoLIC is a good candidate for a complex user interface element such as window or dialog box. The information needed to decide which widgets should be included in the storyboard comes from both the ontology of signs and the interaction model, which represent the nature of the information and the way it is to be presented or manipulated, respectively.

As we have said, interaction design in the semiotic engineering perspective is concerned with building a coherent and cohesive message, focusing on the users' understanding of the designers' conception of the application. We believe that, if the designer's assumptions and design decisions are conveyed successfully, with few communicative breakdowns or misunderstandings, i.e., through a well-designed user interface, users will not only make better use of the application, but also have more resources to use it creatively when unanticipated needs arise.

A decisive factor in conceiving MoLIC was to keep it as independent of specific user interface issues and technological platform as possible. This consideration not only facilitates the (re)use of models in various stages of software design and development, but also avoids that decisions about the user interface are forced to be made prematurely, making it harder for designers to explore different alternative solutions.

MoLIC has been used in the design and redesign of interactive systems in different environments. It has allowed a deeper and more organized understanding of the domain, and promoted the reflection about alternative design solutions, offering better support for the design team's decision making processes, before proceeding to the interface specification, storyboarding and implementation. Although reflection occurs when any design model is used, existing models generally allow the representation of solutions at too low an abstraction level, where important aspects of the solution are omitted, such as the representation of alternative or failure courses of action, and the communicative exchanges that take place between user and designer's deputy. In particular, this representation of "conversation" allows designers to perceive more clearly the effect of their design decisions, and thus makes it easier to determine which solution is adequate for the situation at hand.

We are currently investigating the integration of MoLIC with system specification models used in software engineering, aiming at augmenting HCI design quality without causing a negative impact on the software development cycle. Also underway there is a study about the use of MoLIC for building HCI patterns [9]; and another one for modelling synchronous multi-user environments.

5. References

[1] CACM (2002) Ontology: different ways of representing the same concept. Communications of the ACM, Volume 45, Issue 2 (February 2002).

[2] Carroll, J. M. (ed) (1995). Scenario-based design: envisioning work and technology in system development, New York, Wiley.

[3] Carroll, J. M. (ed) (2000) Making use: Scenario-Based Design of Human-Computer Interactions. The MIT Press. Cambridge, MA.

[4] Carroll, J.M.; Mack, R.L.; Robertson, S.P.; Rosson, M.B. (1994). "Binding Objects to Scenarios of Use", International Journal of Human-Computer Studies 41:243-276.

[5] de Souza, C.S., Barbosa, S.D.J., da Silva, S.R.P. (2001) "Semiotic Engineering Principles for Evaluating End-user Programming Environments", Interacting with Computers, 13-4, pp. 467-495. Elsevier.

[6] Imaz, M. & Benyon, D. (1999) How Stories Capture Interactions. Proceedings of IFIP TC.13 International Conference on Human-Computer Interaction, Interact'99. pp. 321–328.

[7] Norman, D. e Draper, S. (eds., 1986) User Centered System Design. Hillsdale, NJ. Lawrence Erlbaum.

[8] Paula, M.G. (2003) "Projeto da Interação Humano-Computador Baseado em Modelos Fundamentados na Engenharia Semiótica: Construção de um Modelo de

Interação" (in Portuguese). Masters Dissertation. Informatics Department, PUC-Rio, Brasil. March, 2003.

[9] Paula, M.G. & Barbosa, S.D.J. "Bringing Interaction Specifications to HCI Design Patterns". CHI 2003 *Workshop on Perspectives on HCI Patterns: Concepts and Tools*. Fort Lauderdale, Florida, April, 2003.

[10] Peirce, C.S. (1931) Collected Papers. Cambridge, Ma. Harvard University Press. (excerpted in Buchler, Justus, ed., Philosophical Writings of Peirce, New York: Dover, 1955)

[11] Schön, D. (1983) The Reflective Practitioner: How Professionals Think in Action, New York, Basic Books.

Finding Boundary Objects in SE and HCI: An Approach Through Engineering-oriented Design Theories

Andrew Walenstein
Center for Advanced Computer Science
University of Louisiana at Lafayette
walenste@ieee.org

Abstract

This paper outlines an approach of using engineering-oriented design theories to build bridges between software engineering and human–computer interaction. The main thrust of the approach is to try to use design theories to create "boundary objects", which are intellectual tools that are shared by both disciplines and enable cooperation. The approach is illustrated and supported by relating it to ongoing domain-bridging work which is exploring the application of cognitive support theories to SE research problems.

1. Introduction

Gaps currently exist separating human-computer interaction (HCI) and software engineering (SE) research and practice. These gaps are due, in part, to differing terminology, concepts, education, and methods. In the past, such gaps have been identified as being problematic, and efforts have been made to to try to bridge these gaps (e.g., John *et. al* [13], Carroll *et. al* [4]). These advancements are encouraging, but new methods for bridging the gaps are still needed. In this paper I shall argue that, over the long haul, critical struts in key bridges between SE and HCI will be in the form of *engineering-oriented design theories*, and in the intellectual tools these make possible.

Pragmatic, engineering-oriented theories have historically been key contributors to the maturation of engineering disciplines. Why should HCI engineering be any different? In other engineering disciplines, explicit theories—and the intellectual tools they make possible—have served to reduce the reliance on skill and craftsmanship, and thus paved the way for more deliberate and systematic engineering. For example, Vincenti recounted the way in which airplane wing design was transformed by the emergence of theories of wing performance [19]. After the initial amateur experimentation phase but before the advent of the design theories, wing design relied critically on special labs. These labs had plenty of experience in testing and developing various known wing designs, relied to a great degree on this experience to generate plausible designs, and leaned heavily on their capabilities to use wind tunnels to test prototypes. The similarity to modern-day HCI design consultants and user testing facilities is readily apparent. New wing performance theories facilitated a deeper and broader shared understanding of both the overall design problems and their solutions. This allowed wing design decisions to be more closely integrated into the other engineering decision processes. These processes typically involved various tradeoffs such as flight range, load limits, and engine power requirements [19]. With suitable theories available, the primacy of wind tunnel tests in wing design abated. Such changes represent important transformations away from primarily craft-based empirical practices using post-design prototypes, and towards significantly theory-based practices which can work in a more deliberate, anticipatory manner.

I believe that design theories should be able to play a similar role in transforming SE practice by bridging gaps between HCI and SE activities. HCI theories can help transition "invent and test" practices into "specify then implement" practices [6]. It might take a long time to complete this transition, but it is unlikely to be sped along by avoiding the challenge. Overall, then, I agree with the assertions by Newell *et. al* [16] and John *et. al* [13] that "design is where the action is". They argued that HCI has to a great extent organized itself around evaluation and user testing, and that by the evaluation stage HCI is already in a sense too late. Although I concur with this argument, I would also point out that nothing appears to stop us from bringing theory to bear on the whole spectrum of SE activities (engineering requirements, assuring quality, etc.). Their argument also does not settle the question of how SE and HCI will relate to each other as the fields evolve. The maturation of engineering practice is likely to involve transitioning HCI theory into HCI practice and SE theory into SE practice. Moreover—

and this is critical—it might involve building *both* SE and HCI theories into structures that *bridge* SE and HCI.

This paper explores the use of engineering-oriented design theories to bridge gaps between SE and HCI. First, the particular theory-based route being advanced will be clarified. The central theme is that the bridging can be done by building so-called "boundary objects" in the form of practitioner-oriented intellectual tools. Second, I will try to both illustrate and support the argument by describing an ongoing effort to bridge gulfs between SE and HCI using *cognitive support* theories. These are theories that say why artifacts can help or aid cognition. Such theories can form the foundation for bridging structures. Implications for future research directions are outlined.

2. Bridge building with design theories

It is not *a priori* clear what sort of bridgework is needed between SE and HCI, if any. For example, one may argue that merely *bridging* the two does not go far enough, and that HCI and SE should, in fact, be *merged* into a single whole. After all, the overwhelming majority of software is made ultimately for nobody but humans. Thus it is not beyond reason to claim that all computing invariably needs to conform to human requirements, goals, and needs, be they cognitive, social, economic, political, organizational, emotional, spiritual, philosophical, etc. Surely the problem of effectively building software to satisfy human requirements is a theme which is central to both HCI and SE.

Curious, then, that normally dispassionate people can be made to bristle when it is coyly suggested that HCI be positioned as a part of SE or vice versa. I just may have enough personal experience on this matter to credibly offer the following etiquette guidance for how to behave at future SE–HCI social functions. Beware when suggesting that HCI is merely a part of SE—the part most directly connected with humans. Avoid suggesting that HCI can be comfortably slotted as parts of requirements engineering and quality assurance; do *not* call it "the annoying part most easily cut if pressed for time"! Likewise, bite your tongue before proposing that SE is simply that expensive but basically unproblematic portion of HCI consisting of the final step from idea to silicon. And leave out suggesting, even if you are merely waxing philosophical, that SE is actually the junior partner to HCI since HCI is the final arbiter of quality and suitability.

Playful ribbing and turf wars aside, there are good reasons why HCI and SE are not comfortably subsumed by one another, and why they may forever maintain quite distinct journals, conferences, courses, development departments, and value systems. In my mind the main argument in favor of perpetual separation is that they break off different portions of the overall challenge of building computing solutions fitting human needs. For to build software that truly fills human needs one must solve two frequently baffling—and separate—riddles. First, *what satisfies the human needs of the clients or users*? This is HCI's home turf. Software is only part of the answer. And even if you assume you can perfectly write the software for free, you are effectively no closer to answering the question. Second, *what software is feasible and how do you actually build the software effectively, efficiently, and reliably*? This is what SE can arguably claim is in its foveal view. Even if HCI can be relied upon to independently figure out the human suitability issues, some software artifacts must be engineered. And not all envisionable software can be feasibly built, so the parameters for HCI solutions are at least in part determined by SE's software factors such as reliability, cost, safety, etc. So, really, to a great extent SE and HCI have different scopes that force them to specifically concentrate on different aspects of software development.

There may be good reasons to wish this separation to persist. The loose coupling between the two permits specialized training, gives practitioners the freedom to use their own specialized techniques and methods when they work best, and allows researchers and innovators to explore new directions unencumbered (at least initially) by requirements from the other domain. The same basic argument is used in SE to defend personnel selection to ensure mixed and synergistic expertise in software development teams. It also underlies the advantages of heterogeneous systems in AI and cognitive science (e.g., Newell's blackboard system argument [15]). Thus it is reasonable to ask: need we bridge these two disparate disciplines?

Absolutely. Experiences with the problems in current development practices seem to demand it. Furthermore, admitting that HCI and SE are distinct in essential ways can, at most, argue only that they should not be merged into a truly monolithic entity. Some sub-structure is still admissible; and their distinctness in no way absolves them from needing to smoothly inter-operate. In large organizations it might be feasible to more-or-less forcibly separate the HCI and SE into different people, units, and sub-processes. Then, perhaps, the main issue is merely to get SE and HCI people, units, or processes to adequately mesh when they cross paths (which, thinking optimistically, is rarely). But surely is not always possible or desirable for HCI and SE to be highly decoupled; indeed it seems quite likely that the *same people* will sometimes need to wear both SE and HCI hats *at the same time*. SE and HCI are therefore tied together by their shared problem and, indeed, are often bound up in the same people, activities, and events. And in both research and practice, SE and HCI are currently too loosely coupled to adequate facilitate the needed interplay. Bridges—or *more* or *better* bridges—need to be constructed. But how?

I propose that engineering-oriented design theories can be effectively employed to bridge the gaps between SE and HCI. I am certainly not the only one to propose a theory-laden prescription for delivering HCI to SE, or for imbuing HCI with more of SE's sensibilities such as its focus on tools, processes, and formal methods. In fact, my argument will take significant cues from authors such as Newell and Card [2, 16], Green *et. al* [8–10], and Barnard [1]. Although I have gratefully borrowed from this existing heritage, it is still important to clearly lay out a position on what sort of bridges can be built, and how to do it using design theories. This is done in two stages by first explaining what *boundary objects* are and how they might bridge gaps between HCI and SE, and then by proposing how design theories might be helpful as a foundation for organized collections of boundary objects.

2.1. Boundary objects as domain bridges

The launching point for this proposal is Star's analysis of the workings of scientific communities [17]. Star employs her concept of "boundary objects" to explain how it is that a loosely coupled heterogeneous network of scientists can coordinate to solve problems. Green [8] pointed out that Star's concept of a boundary object is also useful in describing how to bridge between different domains such as cognitive psychology and HCI or SE.

Based on historical case studies of scientific work involving both professional scientists and amateurs, Star and her colleagues found that the participants:

> (1) cooperate without having good models of each other's work; (2) successfully work together while employing different units of analysis, methods of aggregating data, and different abstractions of data; (3) cooperate while having different goals, time horizons, and audiences to satisfy. [17, pg. 46]

The reader may immediately recognize the similarity between that description and the gaps which exist between SE and HCI. SE and HCI people often fail to completely understand each other, are accustomed to different representations, and are often focused on different goals, audiences, etc. HCI and SE nonetheless must work together as a coordinated whole to solve a common problem. Specifically, they must work smoothly together to build good software.

It makes sense, therefore, to look at Star's analysis to try to see if lessons learned there can apply to build better bridges between SE and HCI. Star studied heterogeneous and loosely coupled communities having divergent conceptions, goals, methods, and knowledge. If these gulfs *truly* separated the heterogeneous communities, would coordination not break down? Something must have been bridging the gulfs.

Star suggested that in the activity she observed it was the *boundary objects* that made cooperation possible. Boundary objects are "objects that are both plastic enough to adapt to local needs and constraints of the several parties employing them, yet robust enough to maintain a common identity across sites," [17, pg. 46] and that they sit "in the middle of a group of actors with divergent viewpoints" [17, pg. 46]. From her description it seems that their value stems partly from their ability to provide a common structuring resource for the communities on either side of it, and partly also from their ability to make it possible to map data or knowledge from one domain onto another.

This idea of a boundary object between communities is similar to Barnard's conception of "bridging representations" for bridging gulfs between theory and practice [1]. His bridging representations may be thought of as representations of knowledge from one community that have been transformed and tailored for use in another. Compared to Star's conception of boundary objects, Barnard's description of bridging representations makes them appear less shared, and more one-dimensional in the way they facilitate coordination. Thus Star's boundary object model may be a more suitable starting point for discussing how to bridge gulfs *within* practice or theory. Barnard's emphasis on transformation may be more helpful in understanding how to bridge gaps *between* practice and theory.

If we take these ideas of boundary objects and import them into the debate about how to bridge between SE and HCI, then we should be looking to create boundary objects that allow loose coordination in similar ways. This attempts to go beyond merely packaging contents from HCI for use by SE, improving cross-discipline training, or massaging HCI practice to make it follow or interleave with typical SE processes. It would involve creating objects that are expected to sit between loosely-coupled HCI and SE processes and coordinate them. These boundary objects would leave flexibility in interpretation and usage for parties on different sides of the SE/HCI divide.

Star emphasized that there should be a plurality of boundary objects, each suited to sharing different types of information from the different communities. In fact, she documented a taxonomy—i.e., a collection of boundary object *categories*—which she extracted from her historical case studies. Boundary objects from any category shared similar characteristics and coordination benefits. These categories included:

Label. Keywords or other conceptual categories are shared between two communities. These are "indexes" that can be used to refer to other entities of interest within the domain. The labels establish common terminology even though implications and interpretations of the terms may vary between communities.

Repository. An indexed collection of units such as a library or museum. Multiple domains can import these atomic units for use in work done in each domain. This implies that the units combine ideas and issues from multiple domains, and do so in suitably fine-grained units.

Ideal Type/Platonic Object. Star described a Platonic object as "an object such as a map or atlas which in fact does not accurately describe the details of any one locality. It is abstracted from all domains, and may be fairly vague. However, it is adaptable to a local site precisely because it is fairly vague" [17, pg. 49].

Terrain with Coincident Boundaries. These occur when objects from different domains share commonalities. I find it helpful to de-emphasize Star's physical description (e.g., "terrain"), and think of them instead in terms of homomorphic structures which are utilized differently within different domains. Her example is of geographic maps which are tailored to show different information for different communities, but which can nonetheless be correlated because of their common geography.

This derived taxonomy of boundary object types may also apply well to SE-HCI. A short survey of previously proposed bridges between HCI and SE yields several candidates which match qualities of Star's categories. For instance, the now ubiquitous GUI toolkit may be considered a repository since it usually contains an organized library of reusable interface components. GUI toolkits simultaneously binds SE concerns (reuse, framework design, etc.) and HCI concerns (interaction primitives, style, etc.). For another repository example, we might also point to Sutcliffe's proposal for building an indexed collection of reusable HCI design analyses [18]. Star's description of Platonic objects is highly reminiscent of the aims of design patterns. HCI design patterns (e.g., Erickson [7]) might therefore reasonably be called Platonic objects. It might also be argued that "usability architectures" [13] match Star's intent for Platonic objects since architectural-level descriptions abstract over lower-level details. In addition, the role sometimes envisioned for scenarios in coordinating HCI and SE [4] seems consonant with Star's description of terrain with coincident boundaries.

Star's particular taxonomy of boundary objects may not be exhaustive. It may also not apply well enough to SE–HCI. However it at least hints at the possibility of understanding SE–HCI gulf-bridging in abstract, principled ways. It suggests also that it should be feasible to inform SE–HCI bridging by studying and modeling HCI–SE coordination in real organizations.

2.2. Design theories and boundary objects

When I propose to use "engineering-oriented theories" I have in mind theories that are explicitly designed with effective engineering decision making and judgment in mind [14]. Typically this means an eager acceptance of approximation, simplification, and general rules. Even theories that have been proven inaccurate and false can be good engineering theories if they answer engineering questions adequately well in known circumstances [14]. Thus the widely known GOMS [14] framework qualifies as an engineering-oriented theory.

By "design theory" I mean to include only those theories that can make direct contributions to the design of artifacts. Metaphorically speaking, the designer gives such theories information about engineering goals and tradeoffs, and the theory responds with hints (i.e., predictions) as to which artifacts might satisfy these goals. Significantly, my definition explicitly requires that the theories need no existing tool, prototype, or design in order to be invoked. This definition excludes theories such as GOMS since it requires a design to be given before it can be invoked. Design theories can be engineering-oriented. "HCI design theories" facilitate the design of interactions between humans and computers. Thus the Cognitive Dimensions framework (CDs) [8] can justly be called an engineering-oriented HCI design theory. CDs give names to common HCI-related design tradeoffs and choices. The CDs are explicitly advertised as a "broad-brush" intellectual tool so they are of a different character than GOMS [9]. As such, they do not fall into the calculational style of "hard science" theory thought by Newell and Card [16] to be necessary. They are engineering-oriented, nonetheless, in that they can be used proactively by designers to reason about design decisions and tradeoffs. Therefore they can do just what Newell and Card wished for in a practical theory for HCI.

Now there is, I believe, an important relationship between design theories and desirable boundary objects for SE–HCI. Many—if not all—key boundary objects for SE–HCI will embed within them design theories. These theories may not be very explicit. The situation is directly analogous to the idea from Carroll and Campbell [3] that artifacts embed theories. For instance a skyscraper embeds a theory of local weather since it reliably predicts the expected maximal wind speeds (they are usually built to withstand expected storm forces). In the same way SE–HCI boundary objects can be expected to embed design theories from SE and HCI. More specifically, they can be expected to *simultaneously* embed theories from both SE and HCI.

An example of such a boundary object is a usability architecture [13]. John and Bass argued that usability architectures closely weave SE and HCI design issues together in its architectural description. For instance, their description of a usability architecture highlights the frequent need and utility of an undo mechanism and also indicates the need to consider undo mechanisms at the architectural level. Such architectures embed an HCI theory regarding the usability impact of an undo mechanism (including its likely uses, etc.). It also embeds a SE design theory as to what software architectures are suitable for implementing flexible and reusable undo mechanisms. The particular theories might be neither explicit nor obvious, but it seems to me that they must exist or else the design results would likely be unacceptable to one or both of the SE or HCI audiences.

At this point it seems plausible that most or all of the boundary objects one could point to would similarly rest on fundamental design theories from both domains. It also seems likely that certain theories would be able to contribute to the foundation of several different boundary objects. Furthermore, it seems eminently sensible to desire whole *families* of boundary objects which apply across the entire software lifecycle and which are related by a common theoretical framework. Its reasonable to expect that SE processes would generally be aided by a coherent collection of boundary objects for requirements engineering, design, implementation, and testing. Thus instead of merely pursuing an *ad hoc* collection of boundary objects, it might be feasible to generate suites of related boundary objects to coordinate HCI and SE at several levels and during several activities.

These thoughts lead to an ambitious vision for bridging SE and HCI. Imagine, for example, that one has a general theory of the usefulness of undo mechanisms. Suppose the theory states when they are useful, what features are useful, provides quantities for their impact, and so on. Presumably the theory relies on a reasonably well defined ontology and a vocabulary to match. Further, suppose that the theory is found useful in generating usability specifications [5], as well as for defining a related usability architecture. An object-oriented framework might be built to implement the architecture. These (specs, architecture and OO framework) might well be used in a variety of HCI *and* SE activities related to requirements engineering, design, and evaluation or quality assurance. Presumably the theory's vocabulary, ontology, and statements about usability would be preserved in these, which should help ensure coherence within the family. The result is a purposefully-designed, coherent collection of boundary objects built from a common theory. What is more, the theory is exposed to SE and HCI scientists for independent scientific validation. With such a theory pervasively backing the boundary objects, this could lead to an engineering discipline built more solidly on a scientifically sound foundation.

This vision is ambitious and the route to fulfilling it is likely full of serious hurdles. But I strongly feel the potential payoffs make it worth trying. The main catch, of course, is having useful design theories and knowing how to weave these into various boundary objects. Are useful theories even possible? These questions are addressed in the next section.

3. The case of cognitive support theories

This section attempts to illustrate the envisioned bridge-building approach. The illustration is intended to help convey the overall vision and to help inspire new ways forward. This section contains no definitive results or conclusions as the work being described is very much in its beginning stages. However if it succeeds in providing grist for current SE–HCI debates then it will have served its purpose.

The illustrations in this section are based on the work stemming from my own dissertation work [20]. The main thrust of the dissertation was to try to set SE on a firmer scientific footing regarding cognitive issues in HCI. Specifically, the dissertation worked towards establishing a comprehensive, applied theory of *cognitive support* which could be used to develop pragmatic resources for SE and HCI practitioners. This background is briefly described, and the way it is proposed to be used is then overviewed and related back to Star's boundary object ideas.

3.1. Cognitive support background

Cognitive support can be defined as the capacity an artifact has for assisting in cognitive work. Artifacts that support cognition make cognition easier, faster, or less errorful—i.e., better in some way. Cognitive support is only one HCI issue in a sea of many, but it is a crucial one: it is a critical factor in the usefulness of essentially all software designed to support knowledge work. A design environment, a library interface, a financial forecasting tool, or a cockpit would need to consider how cognitive performance is improved by using the software tools. If any of these failed to adequately support the cognition of its users, it would likely be considered to be a failure.

An important cog in my dissertation is the definition of a high-level theory of how artifacts can support cognition. This theory is called RODS [20, 21]. It assumes an umbrella framework for understanding HCI in terms of theories of "distributed cognition" (DC). DC treats cognition as a type of computation: one that is distributed between humans and artifacts. For example, in a cockpit human pilots rely on external media such as marks on dials to hold data that, without such external aids, would otherwise need to be held in their heads. DC effectively treats such a cockpit as a heterogeneous distributed computing system where pilots are computing nodes and the dials are external memories they can access [11].

The basic insight behind RODS is that cognitive assistance can be traced to rearrangements in computation. For instance, if a dial marker is introduced into a cockpit, it can offload data from internal memory onto external memories, thereby reducing the cognitive burdens of the pilots. The support introduced reengineers the cognition in the cockpit according to principles of distributed memory computing optimization. RODS claims that all cognitive supports are instances of computational reengineering. This is a significant statement. It asserts that improvements in cognition can be related back to established computational theories. To put it another way, psychological effects are explained in computing sciences' own terms.

Another important claim that RODS makes is that many different types of cognitive support can be decomposed into just four primitive types of computational rearrangements. These four transformations give RODS its name. For instance, an external memory such as a bookmark list in a browser falls into the support category termed "distribution" because it distributes data (the bookmarks). The "D" in RODS stands for "distribution". The remaining letters of the RODS acronym name the other support categories. All four RODS categories are transformations commonly known to computer scientists familiar with standard programming tricks. It does not add enough value to the present argument to discuss the theories in more detail here. Details are in the dissertation [20].

Another key part of the work is the realization that one can apply the four generic computational transformations defined by RODS to any DC model of human–computer systems. In particular, if one considers a particular model of a human interacting with a computer (or other artifact), then RODS transformations on that model can be systematically considered. These transformations correspond to promising design moves. For instance, the data and processing in the model can be analyzed for opportunities for distribution onto external memory or processing. By using a generic cognitive model, a taxonomy of 16 different classes of cognitive support was generated and considered. Each class of cognitive support abstractly identifies a way to support cognition. For example the browser history kept in a web browser was identified as a specific type of cognitive support, namely the distribution of history states. Any one tool could involve multiple support types.

RODS is not quantitative: it does not model or predict quantities such as the amount of memory that might be offloaded, or the relative importance of various cognitive supports. It also does not directly address tradeoffs. For instance, an external memory is normally expected to have update costs. It is possible to have external memories that are too costly to update to make them worthwhile. These issues are not factored into RODS.

3.2. Boundary object considerations

Although the above introduction is a terse description of the theories, it covers enough to discuss the sort of boundary objects that can potentially be derived from them. Some of these are under investigation. Others are merely speculative at this point however they help convey the current general direction of the research. These boundary objects are described below and their relation to SE and HCI theories are briefly explained. How they coordinate HCI and SE work is also indicated.

Vocabulary and conceptual framework.

RODS and its overall framework identifies and names concepts relating to cognition and its support in artifacts. For example, the terms "external memory", "distribution", "data distribution", and "history state" are defined in order to be able to define the category of cognitive support that involves the distribution of history states.

An extremely important aspect of this vocabulary is that the definitions bind cognitive issues and models in HCI directly to computational terminology and common computing science theories. In this sense, they can be classified as "Labels" in Star's taxonomy of boundary objects. For instance "data distribution" indexes into well-understood concepts from computing science. The notion will be familiar to most software engineers, and implications for design (e.g., storage methods, search techniques, caching, etc.) is likely to be keyed into. The terms "external memory" and and "history state" will also have implications for HCI specialists, who might wish, for instance, to focus in on the how users rely on external encoding of history states to simplify backtracking.

Given the above, the vocabulary of RODS and its associated conceptual framework serves as a type of boundary object between SE and HCI, embedding and combining theories from both computing and HCI. It is well-positioned to be adopted by SE and HCI people alike. Much of the terminology and concepts are familiar to SE people; getting SE people to be aware of HCI concepts and meanings is known to be frequently difficult due to educational differences.

A shared vocabulary and conceptual framework are likely to be important for all phases of software construction. The fact that RODS is not quantitative or does not address tradeoffs does not adversely hinder its applicability. HCI and SE people frequently communicate in informal design meetings, including hallway conversations. A suitable shared vocabulary tied to appropriate abstractions and theories can help lift the level of conversation. This motivation is nicely described by Green and Petre when they say

> Explicitly presenting one's ideas as discussion tools... is doing nothing more than recognizing

that discussion among choosers and users carries on interminably, in the corridors of institutes and over the Internet. Our hope is to improve the level of discourse and thereby to influence design in a roundabout way. [10, pg. 132]

Cognitive support patterns.

RODS is created as a decompositional analytic framework in which complicated forms of cognitive support can be broken down into combinations of various primitive types of cognitive support. Certain compositions of cognitive support appear to work particularly well together. For example, consider some of the advantages offered by a typical hierarchical outline processor such as one finds in Microsoft PowerPoint in Outline View mode. The outline processor allows items to be stored externally, alleviating internal memory bottlenecks and making it possible to directly work with larger presentations. Once externalized, the items in the outline can be manipulated externally by directly manipulating them by dragging them with the mouse. This more visceral manipulation can replace mentally juggling the outline. Once the outline is finished it acts as a plan for writing out the contents of the slides. Each of these features is classifiable by the cognitive support theories. Their composition resolves several problems simultaneously.

The above sort of synergistic compositions—and many others—occur frequently in a variety of contexts and implementations. The compositions are abstract in critical ways. They are independent of the implementing technologies. For instance, paper-and-pencil based "outline processors" share most of the same features as electronic ones. The compositions are also at least partly task-independent. For instance, the utility of the above sort of outline view support can be the same for nearly any task where planning the authoring of a hierarchical document is required, such as in programming or proof writing.

Because they are general, can occur in many contexts, and solve multiple issues simultaneously such compositions might be called *patterns* of cognitive support [20, 21]. This would be a form of HCI design pattern. As it was mentioned before, HCI design patterns are likely candidates to be considered "repositories" in Star's taxonomy of boundary objects. In this case the patterns are specified using combinations of cognitive support terms (i.e., labels) defined by RODS and its framework.

Cognitive support reference designs.

For any specific domain or problem category, one particular generic composition of cognitive support patterns may be prototypical. For example many software reverse engineering tools can be seen as implementing a medium for externalizing and manipulating imperfect knowledge [12]. By this I mean that the tools allow reverse engineers to externally represent and manipulate knowledge about the system, and that this knowledge may be uncertain, contradictory, incompletely, or otherwise imperfect. The reverse engineering process then seeks to repair this imperfect knowledge. As in the case with outline processors, several types of cognitive support combine to make this general solution pattern desirable. The tools allow offloading of memory, make it possible to externally manipulate the knowledge instead of needing to do it internally, and can be made to automatically process the knowledge [12]. The cognitive support analysis of such tools provides a high-level understanding of them [22].

These considerations present the possibility for describing new boundary objects. First, it may be possible to define a type of reference, high-level design for a given task or problem domain (as indicated by Jahnke *et. al* [12] in the domain of reverse engineering tools). This would be useful for kickstarting development, or for generating a reusable framework for building reverse-engineering tools. This reference design can be associated with a pre-existing analysis of the cognitive support provided by the implementation (as in Walenstein [22]). The reference design and its associated cognitive support analysis can serve as reusable boundary object. Since the analysis and reference design can be expected to use the same feature and cognitive support vocabulary, it appears to be much like what Star described as a "Terrain with Coincident Boundaries" (the common structure of the design and analysis is the coincident part). Furthermore, a collection of such reference designs+analysis combinations may be considered to be a repository.

4. Summary and discussion

This paper examined the possibilities for bridging rather than eliminating the gaps between HCI and SE. The analysis focused on a proposal to view the bridging problem in terms of building so-called "boundary objects" that mediate activities between SE and HCI practitioners or researchers. A specific proposal for building such boundary objects was advanced, and was argued to have promise. The proposal involves embedding engineering-oriented design theories in a collection of related boundary objects. Embedding theories in this manner should help ensure coherence and continuity, and assist in the push to solidify the scientific foundations of SE practice.

As an illustration of this overall bridge-building approach, a research effort closely following this proposal was summarized. The theories being employed to build boundary objects were cognitive support theories. From a relatively simple cognitive support theory, several potential boundary objects were described. It was argued that these could help in the coordination of HCI and traditional SE streams. For example, a vocabulary, pattern catalog, and design/analysis repository may collectively form a family of boundary objects related by being created from a common cognitive support design theory. For instance imagine a case where, during design brainstorming, conversation informally drifts toward the tradeoffs between the user's cost of using an external memory versus the potential benefits for remote collaboration. During design elaboration, HCI participants might consider various cognitive support patterns for resolving such tradeoffs. The pattern candidates are used by SE teams to select usability architectures implementing the identified patterns. While the software is being developed the HCI team can use the cognitive support analysis of a reference architecture to plan evaluations and run initial walkthroughs. The theory and its concepts form a theme running through the entire development cycle.

History has shown that engineering disciplines get ideas essentially right even before explicit theories are developed to explain the reasons why. It seems prudent to empirically study real SE–HCI coordination. Latent theories may be discovered, and successes could be replicated in new or updated boundary objects. In short, the work to bridge SE and HCI can proceed by deliberate steps of investigating design theories and then empirically examining practice for opportunities for improving current practice.

References

[1] P. Barnard. Bridging between basic theories and the artifacts of human-computer interaction. In J. M. Carroll, editor, *Designing Interaction: Psychology at the Human-Computer Interface*, chapter 7, pages 103–127. Cambridge University Press, 1991.

[2] S. K. Card. Theory-driven design research. In G. R. McMillan, D. Beevis, E. Salas, M. H. Strub, and R. Sutton, editors, *Applications of Human Performance Models to System Design*, pages 501–509, New York, 1989. Plenum.

[3] J. M. Carroll and R. L. Campbell. Artifacts as psychological theories: The case of human–computer interaction. *Behaviour and Information Technology*, 8(4):247–256, 1989.

[4] J. M. Carroll, R. L. Mack, S. P. Robertson, and M. B. Rosson. Binding objects to scenarios of use. *International Journal of Human–Computer Studies*, 41(1/2):243–276, 1994.

[5] J. M. Carroll and M. B. Rosson. Usability specification as a tool in iterative development. In H. R. Hartson, editor, *Advances in Human–Computer Interaction*, volume 1 of *Human/Computer Interaction Series*, pages 1–15. Ablex, Norwood, NJ, 1985.

[6] J. Dowell and J. Long. Conception of the cognitive engineering design problem. *Ergonomics*, 41(2):126–139, 1998.

[7] T. Erickson. *Lingua Francas* for design: Sacred places and pattern languages. In *Proceedings on Designing Interactive Systems: Processes, Practices, Methods, and Techniques*, pages 357–368. Association for Computing Machinery, 2000.

[8] T. R. G. Green. Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay, editors, *People and Computers V: Proceedings of the Fifth Conference of the British Computer Society Human-Computer Interaction Specialist Group*, pages 443–460. Cambridge University Press, 1989.

[9] T. R. G. Green, S. P. Davies, and D. J. Gilmore. Delivering cognitive psychology to HCI: the problems of common language and knowledge transfer. *Interacting with Computers*, 8(1):89–11, 1996.

[10] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[11] E. Hutchins. How a cockpit remembers its speed. *Cognitive Science*, 19:265–288, 1995.

[12] J. H. Jahnke and A. Walenstein. Reverse engineering tools as media for imperfect knowledge. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'2000)*, pages 22–31. IEEE Computer Society Press, 2000.

[13] B. E. John and L. Bass. Usability and software architecture. *Behaviour and Information Technology*, 20(5):329–338, 2001.

[14] B. E. John and D. E. Kieras. Using GOMS for user interface design and evaluation: Which technique? *ACM Transactions on Computer–Human Interaction*, 3(4):320–351, Dec. 1996.

[15] A. Newell. Some problems of the basic organization in problem-solving programs. In M. C. Yovits, G. T. Jacobi, and G. D. Goldstein, editors, *Proceedings of the Second Conference on Self-Organizing Systems*, pages 393–423, New York, 1962. Spartan Books.

[16] A. Newell and S. K. Card. The prospects for psychological science in human-computer interaction. *Human Computer Interaction*, 1(3):209–242, 1985.

[17] S. L. Star. The structure of ill-structured solutions: Boundary objects and heterogenous distributed problem solving. In M. N. Huhns and L. Gasser, editors, *Distributed Artificial Intelligence 2*. Morgan Kaufmann, 1989.

[18] A. Sutcliffe. On the effective use and reuse of HCI knowledge. *ACM Transactions on Computer–Human Interaction*, 7(2):197–221, June 2000.

[19] W. G. Vincenti. *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*. The Johns Hopkins University Press, Baltimore, 1990.

[20] A. Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. PhD thesis, School of Computing Science, Simon Fraser University, May 2002.

[21] A. Walenstein. Foundations of cognitive support: Toward abstract patterns of usefulness. In *Proceedings of the 14th Annual Conference on Design, Specification, and Verification of Interactive Systems (DSV-IS'2002)*, volume 2545 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 2002.

[22] A. Walenstein. Theory-based analysis of cognitive support in software comprehension tools. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02)*, pages 75–84. IEEE Computer Society Press, 2002.

Separation in Theory – Coordination in Practice

Torkil Clemmensen[a], Jacob Nørbjerg[b]

Department of Informatics, Copenhagen Business School, Howitzvej 60, DK-2000, Frederiksberg, Denmark
tc.inf@cbs.dk[a],jan.inf@cbs.dk[b]

Abstract

The lack of a common language and mutual understanding between the disciplines of systems development/software engineering and HCI does create challenges for both teaching and practice in systems and software development. In this paper we propose, however, not to attempt to 'bridge the gap' between the disciplines when teaching information systems development. Using our own teaching as an example, we argue that it would require a considerable effort to reconcile the differences in terminology and approaches to systems development between the two disciplines and that the reasonable way forward is to teach the students how to coordinate in practice, while keeping the theories separate. Coordination of HCI and SE curricula can take place by using a vehicle of tacit knowledge, the software prototype, which enables students to learn through practical experience the richness and usefulness of both approaches. The implications of this position are discussed.

1. Introduction

The authors of this paper teach systems development methods and HCI at a university program in Informatics. With a background in psychology and human-computer interaction, respectively computer science and software engineering, we've come to see HCI and software engineering as two widely separate 'bodies of knowledge', both occupied with the design and implementation of useful information systems, but with very separate world views, research traditions and theoretical foundations. Building shared concepts and languages, and integrating the two fields in a joint curriculum, is therefore a considerable undertaking, which may furthermore reduce the students' appreciation of the richness and diversity of each topic. Our position is, therefore, that as far as teaching is concerned, the two disciplines should be coordinated rather than integrated by being presented and discussed separately but concurrently, with constant consideration of and attention to their practical interactions and dependencies, but as disciplines in their own right, each with their own vocabulary, theoretical foundations, and research traditions. The dependencies and interactions between the two fields can be demonstrated and appreciated through practical assignments where the students apply methods and techniques from HCI and software engineering in small projects building prototypes of a system. In this way, they demonstrate that they know how to apply a systems development methodology as well as analyze users' interactions with a system.

Our position builds on our experience from jointly planning and teaching a course in information systems development where we try to apply the above principles. The course is part of the 2nd year of a bachelor program in business administration and computer science which – on a larger scale – applies similar ideas to the whole program.

The significance of our contribution is that we state a general principle for coordinating HCI and SE curricula: coordination can take place by using a vehicle of tacit knowledge, the software prototype. Furthermore, choosing the prototype as a point of coordination – instead of trying to integrate everything in e.g. a new shared language or common method – allows students to learn an important medium for practitioners: prototype design, which computer scientists, usability specialists and buyers and users all can relate to.

The rest of the paper is organized as follows: section 2 outlines the bachelor program in business administration and computer science. Section 3 describes the HCI and systems development curricula of the 2nd year course in informatics and argues why we want to teach these as different subjects. In section 4 we present and discuss how the two fields can be systematically combined in practical assignments. Section 5 concludes the paper.

2. The bachelor program in business administration and computer science

The bachelor program in Business administration and computer science at the Copenhagen Business School

(CBS) is a three year, higher education study program aiming to provide students with the necessary qualifications for participation in functions concerning formulation of requirements, development, implementation, operation and sale of IT-based systems on the basis of knowledge of relevant business and societal conditions. Each year 200 students enroll in the program. After the three years, the bachelors will typically be employed as "bridge builders" between users of IT systems within companies and those persons who deliver IT systems (hardware and software).

The program has a fixed structure and all courses are mandatory, though there is no compulsory attendance. The teaching consists of lectures based on textbooks and exercises supervised by student instructors. The exams combine comprehensive written assignments (projects) and oral exams based on the curriculum. Thus, the study program provides a basic theoretical knowledge within the disciplines taught, but is also very problem-oriented and give priority to the application of methods and theories in cases and practical project work, including learning how to manage IT projects.

The structure of the 3-year study program builds on three parallel lines of courses, one for each of the disciplines: 1) Organization: management and cooperation, conflict resolution, etc. 2) Economy: national and business economy, cost estimation related to information systems development, etc. and 3) Informatics: programming and database design, operating systems and networks, and information systems development. The study program ends with a 6 month group project, in which the students integrate the three disciplines taught in the program and applies what has been learned on a real problem in cooperation with an external organization.

This paper focuses on the informatics part of the program.

3. The Informatics courses

The first year course in informatics focuses on databases and introduction to programming. Thereafter, the second year introduces complex IT systems through four course modules: operating systems, object oriented system development, man-machine interaction, and distributed systems. During the third year, students take two courses related to large scale systems development: definition of information systems and computer supported systems development.

The authors of this paper are responsible for the systems development module and the interaction design module in the 2nd year course. These modules together take 44 lectures and a similar number of exercise sessions and are followed by a two week assignment where the students apply the methods and techniques from the course to a practical problem. The curriculum focuses on methods and techniques but emphasizes underlying theories and concepts too. When we in this paper discuss HCI and SE curricula, it is therefore these kinds of curricula that we talk about. Below we briefly describe the two course modules followed by a discussion of the differences in terminology usage and perspectives between systems development and interaction design that we've experienced, and how these affect the way we plan and conduct the course.

3.1 Object Oriented systems development module

The object oriented systems development module takes up 32 lectures and a similar number of exercises. The module's core curriculum consists of a text-book in an object oriented analysis and design (OOA/D) methodology [9]. The methodology uses the UML notation and applies many of the same ideas as the Unified Process (see e.g. [6]) but with higher emphasis on the business modeling step. In addition to the methodology, the module introduces students to other SE topics such as process modeling, quality assurance and configuration management.

The OOA/D methodology divides analysis and design into five main steps: definition of system scope and purpose, development of a model of the system's problem domain, analysis and definition of functionality and interface requirements, architectural design, and object design. The methodology emphasizes a layered architecture and the use of analysis and design patterns.

The textbook focuses on clear and concise descriptions (diagrams, tables, structured descriptions) to document work and support subsequent work; i.e. programming and maintenance. It explicitly recognizes that some of the activities in the methodology – particularly definition of system scope and purpose, and determination of interaction requirements and functionality – depend on skills and techniques not covered by the methodology; e.g. organizational analysis, work process analysis, and interaction design. These skills and techniques are briefly mentioned and some short examples of their use are given but in general they are treated as belonging to (important) activities 'outside' the methodology, which can be used to produce input to the analysis and design process; e.g. in the form of use cases and interface specifications.

The teaching and the evaluation of the students' work emphasize consistent application of the method's principles and development of analysis and design models of high technical quality.

3.2 Interaction design module

The students follow the Interaction Design module in parallel with the systems development module,. The Interaction Design module spans 12 lectures and a similar number of exercise sessions. The curriculum consists mainly of a textbook in Interaction Design [11]. The rationale for using this book is that our everyday interaction with systems and products require more than a flawless technical design. It is also necessary to design products that are easy, effective and fun to use. Hence the purpose of the module is to allow the students to familiarize themselves with theories and methods for user-oriented analysis, design and evaluation of IT interfaces.

The module combines lectures that introduce concepts, models and principles of interaction design with exercises where the students apply these principles. For example, a lecture may introduce the students to the principle that beneath every physical design lies a conceptual model and in the exercises the students will study calendar systems and their underlying conceptual models. Thus, the module is not a course in the construction of graphical interfaces, but an introduction to user-oriented design. The aim is that the students:

- acquire a terminology to reason about a product's user-friendliness,
- understand how conceptual models influence the design and use of products,
- can apply different techniques to identify users' needs and describe their requirements,
- can demonstrate hands-on experience with the use of prototypes to involve users in the design process,
- have insight in strengths and weaknesses in usability test methods.

The module is the students' first encounter with HCI and it is therefore of utmost importance that they become able to see the usefulness and richness of this approach. They can later (at the graduate level) engage in more deep level studies within the discipline.

3.3 Differences in concepts and approaches

During our discussions on how to organize and run the second year informatics course we have come to realize how difficult it will be to integrate the disciplines in a coherent manner which at the same time respects both areas' research traditions, theoretical underpinnings, and perspectives on systems development and humans. We illustrate this point with examples of the issues and problems we've encountered followed by a brief reflection on the two fields' position in the larger landscape of IS and SE research and practice.

Our first example illustrates a terminological and conceptual confusion caused by the different meaning and use of the concepts *problem domain* and *problem space* Within systems development 'problem' or 'business' domain – in some instances also denoted application domain – is a common term to denote the area of concern of an information system; e.g. accounting, information exchange, or process control. A central activity in systems development methods is the construction of a suitable model of the problem domain to be included in the final computerized system. Differences between methodologies concern the nature of the model and the modeling steps as well as the final representation and use of the model in the computerized system; e.g. as data flow diagrams and data models in classical structured analysis, and use cases and object models in the Unified Process.

The OOA/D textbook used in the course defines problem domain as 'that part of a [system's] context that is administered, monitored or controlled by a system' [9, p. 6]; i.e. accounts, transactions, invoices etc. in an accounting system, or planes, radars, flight paths etc. in a flight control system. The method stipulates that the information system shall model the state and behavior of the problem domain. Thus, a pivotal activity in the methodology concerns modeling the problem domain as objects, classes and structures and identification of the (real world) events and event sequences that influence the state of individual objects.

The Interaction Design textbook uses a similar concept – *problem space* – in a very different manner. "Problem space" is the analyst's "space", i.e. a subjective space, and its elaboration requires the analyst to reflect upon his or her perceptions of the world, and the idiosyncrasies, and social relations of the human (future) user of an information system. In order to avoid beginning design at the physical level, i.e. beginning with decisions on whether to use menus, commands, icons, gestures etc., the interaction designer should think through how his or her design will support people in their everyday work activities [11]. This means defining the usability and the user experience goals by thinking through design ideas and explicating assumptions and claims. These may relate to design ideas that need to be better formulated or user needs that should be determined by identification of problematic human activities or potential useful applications of new technology. In this explication process, frameworks for looking at the system from the users' point of view may be useful. A central output from the process of defining the problem space is conceptual models, i.e. models that describe "the proposed system in terms of a set of integrated ideas and concepts about what it should do, behave and look like, that will be understandable by the users in the manner intended" [11, p. 40]. Thus, working out the *problem space* in Interaction Design is a different process from that of modeling the *problem domain* in OOA/D.

Our second example illustrates the danger inherent in attempting to share a new language between SE and HCI. In 1992, [7] introduced *use cases* into the object oriented community and since then this approach has spread to the HCI community. Today, a use case in HCI is a recognized form of task description, which focuses on the user-system interactions [11]. The interactions of interest are those necessary to achieve the goal most users want in the situation. Thus, the main use case describes what is called the 'normal course' (alternative courses are also considered) through a use case, i.e. the set of actions that the analyst believe to be most commonly performed. For example, if data gathering has led the analyst to believe that most users of a library go to the catalog to check the location of a book before going to the shelves, then the 'normal course' for the use case would include this sequence of events [11, p. 227]. In this way, use cases in HCI is useful in the conceptual design stage, and less so during requirements or data gathering [11, p. 228].

Use cases in HCI are thus a possible way to express insights gained from in-depth enquiry into a use situation whereas OOA/D apply them as means to *determine* the application domain [9, p. 120]. This is done - not by deep analysis of work tasks in the current physical system - but by following a 'blitzing' strategy of making a brief survey and then jump directly to a specification of functional requirements. Thus, the first step in defining the target systems usage is to identify actors by determining their different roles, the second step is to describe the actors by stating their goals and characteristics. The final step is examination of the application-domain tasks in sufficient detail to distinguish between different use cases that describe the interactions between the target system and the actors. Thus, OOA/D models the application domain by structuring actors and use cases. And though the wisdom in following a 'blitzing' strategy vs. doing a deep work analysis may be hard to judge, the point is that HCI and SE enter the matter of use case descriptions from opposite directions and therefore have opposite views and backgrounds. Use cases seems to function as figure-ground pictures; what is the ground for HCI in a use case, is the figure for SE, and inversely.

These, and other examples of terminological differences and inherently different perspectives on shared concepts, indicate that it will require a considerable effort to develop shared curricula and teaching approaches for the two disciplines. However, the example also points to deeper differences in the basic assumptions and underlying world views which are even more difficult to overcome than differences in terminology.

The problem domain analysis step in OOA/D requires the analyst to abstract out relevant (with respect to the system's purpose) features of the 'real world' in order to construct a model which – after some further transformations – will be included in a computerized system. Objects representing human beings; e.g. customers, pilots, or operators may be included in the model, but they are seen as no different from objects that represent machines, tools or abstract entities. Aspects of the real world not suitable for modeling purposes, such as emotions, conflicts, or social relations are (deliberately) ignored as are artifacts or concepts that the analyst deems outside the scope of the system under consideration.

On the other hand, the main idea in the concept of "problem space", within interaction design is that the analyst works out and explicates his or her *implicit assumptions about the use of the design,* preferably by using an explicit theoretical framework and associated techniques. Plurality of perspectives and representation are encouraged when working out a problem space and the purpose of the resulting descriptions is not to build an unambiguous representation of a part of the world, but to inspire reflection and generate ideas about the system's purpose, scope and relation to human work.

The differences outlined above relate to the two disciplines' foundation in widely different research traditions. OOA/D is rooted in software engineering and hence in a tradition that focuses on the *construction* of computer systems. Focus is therefore on the production of unambiguous abstract descriptions of the world that are suitable for representation in a computer program or that can be used to define the interaction between the system and its surroundings. Human beings are relevant insofar that we need to produce interfaces that allow for effective and efficient interaction with them. In this respect OOA/D shares the perspective and approach of the majority of systems development methods, and like these belong in the functionalist systems development paradigm as discussed by [5] (see also [1-3] for an elaboration and discussion of this point.)

Interaction design is, on the other hand, rooted in a different research tradition that emphasizes *understanding* of the social world rather than the construction of technical systems. The purpose of the study of humans and their (possible) interactions with a technical system is therefore to understand human work and – eventually – to improve it with or without a computer system. Plurality of perspectives and inclusion of a wide range of aspects of human and organizational behavior and interaction; e.g. emotions, social relations, (political) conflicts, are accepted and encouraged. HCI – as taught by us – is therefore rooted in the social-relativist paradigm as discussed in [5].

3.3 Integration or coordination?

These examples indicate that attempts to integrate the two disciplines in one course curriculum face several difficulties and challenges. Not only will it require a considerable effort to reconcile the differences in terminology and approaches to systems development, but there is also a risk that a joint or shared curriculum becomes disengaged from the underlying theories and perspectives of one or both of the disciplines. One could – as an example – resolve the ambiguity of problem domain/problem space by inventing new concepts or terminology. This would enable us to discuss "problem space" and techniques for analyzing and describing it within a coherent curriculum, but it would cut the students off from HCI research and theories. We would further risk reducing the teaching of Interaction Design to a set of ready-to-use techniques with no appreciation of the paradigmatic differences between software engineering and interaction design and of the implications of these differences. A similar point has been raised by [10].

Practical software development on the other hand also requires a careful balancing of the two areas to avoid the danger of one approach becoming dominant. Dittrich and Lindeberg has observed how software developers engaged in the study of a user organization risk 'going native'. Like ethnographers risk being immersed in the culture of the natives, software developers can become too immersed in the culture of the users if they adopt a use oriented approach without at some point stepping back and reflecting upon their assumptions [4]. Thus 'working out the problem space' is not an innocent phrase in the Interaction Design vocabulary; it requires considerable insight into and experience with applying frameworks for understanding users.

Instead we propose to teach the two disciplines as different but equally important subjects. This will allow teachers to present and discuss concepts, ideas and techniques within the framework of the research tradition that produced them. Such an approach furthermore opens up for critical reflection and discussion of the differences and similarities between the fields; e.g. the reasons for working with different – equally valid and useful – ambiguous descriptions of "problem spaces", vs. the production of an abstract and precise "problem domain model".

The challenge facing the teacher (researcher, practitioner) is of course how to combine the two disciplines in practical systems development. We describe how we deal with this problem below.

4. The assignments – using the prototype as a means of coordination

We propose to overcome the above mentioned challenge in two ways: First, by constantly emphasizing the relations between the two areas when teaching. When teaching OOA/D, for example, we emphasize that the use cases produced during interaction design and function specification are the visible end-result of a complex analysis of human users' work processes and interaction requirements. Conversely, when teaching the interaction design module we emphasize how some of the artifacts produced when studying users and experimenting with various interface designs relate to the more 'formal' modeling activities in OOA/D.

Second, we use the final two-week project assignment as a means to demonstrate how to combine interaction design and OOA/D in practical project. To do so, we ask the students to develop two different prototypes of the same software system:

A *horizontal* prototype that demonstrates the user interface of a complete system. The prototype should be the end-result of systematically applying theories, tools and techniques taught in the interaction design module and it should conform to accepted user interface design principles. This prototype and the associated documentation documents the students' ability to use the material taught in the interaction design module and thus proves that they can do the analysis necessary to get a firm grasp of the interaction between the user and the design.

A *vertical* (running) prototype that implements a small subset of the (intended) final system's functionality. The prototype – and its associated analysis and design documentation – should be developed according to the methodology and conform to general software engineering analysis, design and programming principles. Thus, the vertical prototype demonstrates the students' ability to systematically apply the OOA/D methodology and produce a running program that is consistent with their analysis and design.

As indicated, the above prototype(s) allow the students to apply the theories and techniques from each of the modules taught within the framework of an assignment of a reasonable size. By deliberately letting the students work in a 'grey zone' between the two clearly different approaches, we enable them– through their own practical experience - to realize how the fields of interaction design and software engineering together contribute to the construction of a system. Thus, they can either choose to make two independent prototypes and describe in words how they are related, or they can choose to develop one prototype that combines an implementation of a subset of the system's functionality with 'dummy' design of the user's interaction with a full system. Either way, they need to reflect on how to coordinate the two approaches.

For example, deliberations over the 'problem space' and experiments with different user interfaces inform the scoping of the system and definition of functional requirements. It can also help the students become aware of different aspects of the 'problem domain' – thus serving to inspire and validate the 'problem domain' analysis and subsequent design activities. On the other hand, the problem domain analysis and design activities inspire and lead to the design of new user interface prototypes, which can help surface implicit assumptions about the use of the system, and thus expand the students' understanding of the 'problem space' of the design. Ideally, such iterations should provide the students with deeper insights into the purpose of the design of a system.

From a project management point of view, this approach allows the students to learn how to coordinate interaction design activities with other analysis and design activities.

5. Discussion and conclusion

The students' iteration between human-computer interaction and software engineering methods cannot be taken for granted; obviously the design problem can be solved by choosing some linear approach. Students may choose to do all the OOA/D first, then the Interaction Design part. In such cases, reflections on how to coordinate the two approaches will be retrospective and unnecessary. However, learning such a linear approach to design is not the goal with the course; therefore we will advice the students to use the two approaches in parallel.

To instruct the students in how to coordinate the use of interaction design and OOA/D methods requires that we can point to obvious points of coordination. To identify these points is an issue for further discussions, but we have a few ideas for our course: First we may ask the students to assign, among themselves and within their project group, advocates for 'problem space' and 'problem domain', and thus have a continuous discussion of these two perspectives of the analysis and design. Second, we may explicitly instruct the students to reflect upon 'use cases' as a concrete meeting point between the disciplines but also as having slightly different connotations and purposes. This approach resembles that of [8] in that we recommend building different kinds of knowledge and insights during a development project by applying different perspectives and sets of tools.

We have argued that coordination between SE and HCI can take place by using a vehicle of tacit knowledge, the software prototype, which also provides students with the opportunity to pay equal attention to interaction design and systems analysis and design in their work. Furthermore using the prototype approach to learn to coordinate will help make implicit assumptions behind the HCI and SE methods more explicit, e.g. assumptions of the power of modeling tools or the precision of usability design principles.

Finally, we have argued that choosing the prototype as a point of coordination – instead of trying to integrate everything in e.g. a new shared language or common method – allows students to learn an important medium for practitioners: prototype design, which computer scientists, usability specialists and buyers and users all can relate to. In this way, the students already during their basic education learn (on a small scale) to coordinate different approaches to design. Hence they are not left to acquire this competence on their own, when they, after becoming bachelors, stand face to face with the 'reality shock' in their first years as practicing designers.

6. References

[1] Bansler, J., Systems Development in Scandinavia: three theorethical schools, *Office Technology and People*, 1989, 4, pp. 117-133.

[2] Bansler, J.P. and K. Bødker, A Reappraisal of Structured Analysis: Design in an Organizational Context, *ACM Transactions on Information Systems*, 1993, 11 (2), pp. 165-193.

[3] Dahlbom, B. and L. Matthiassen, *Computers in Context. The Philosphy and Practice of Systems Design*, 1993, Malden, MA, Blackwell.

[4] Dittrich, Y. and O. Lindeberg, Can Software Development be too Use Oriented? Going Native as an issue in Participatory Design, in *IRIS*, 2001, Bergen, Norway.

[5] Hirschheim, R. and H. Klein, Four Paradigms of Information Systems Development, *CACM*, 1989, 32 (10), pp. 1199-1216.

[6] Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Object Technology Series, ed, I. Jacobson, G. Booch, and J. Rumbaugh, 1999, Boston, Addison-Wesley.

[7] Jacobson, I., et al., *Object Oriented Software Engineering: A Use-Case-Driven Approach*, 1992, Reading, MA, Addison-Wesley.

[8] Kensing, F. and A. Munk-Madsen, PD: Structure in the Toolbox, *Communications of the ACM*, 1993, 36(4), pp. 78-83.

[9] Matthiassen, L., et al., *Object Oriented Analysis and Design*, 2000, Aalborg, Marko.

[10] Nyce, J.M. and G. Bader, On Foundational Categories in Software Development, in *Social Thinking - Software Practice*, R. Klischewski, C. Floyd, and Y. Dittrich, Editors, 2002, The MIT Press, Cambridge, MA, pp. 29-44.

[11] Preece, J., Y. Rogers, and H. Sharp, *Interaction Design: Beyond Human-Computer Interaction*, 2002, John Wiley & Sons.

Usage-Centered Design and Software Engineering: Models for Integration

Larry Constantine
Constantine & Lockwood, Ltd.
University of Technology, Sydney (Australia)
lconstantine@foruse.com

Robert Biddle
Victoria University of Wellington (New Zealand)
robert@mcs.vuw.ac.nz

James Noble
Victoria University of Wellington (New Zealand)
kjx@mcs.vuw.ac.nz

Abstract

This paper argues for a model-driven framework for integrating human interface engineering with software engineering. The usage-centered design process, a robust and proven process grounded in software engineering, is briefly described and contrasted with traditional user-centered approaches. The connections between the core models of usage-centered design and their software engineering counterparts are outlined, and the advantages of this approach to bridging the gap are discussed.

1. Introduction

All software ultimately serves human needs and interests, hence provision for effective interaction between human users and software would reasonably be expected to be integral to all software engineering. Such has not been the case, historically. Until recently, software engineering methods and practice have focused primarily or exclusively on the internal structure and functioning of computer programs, leaving the interaction with users and the user interface that supports that interaction to other disciplines and professionals. For its part, the human-factors community has shown relatively little interest in the problems of software engineering as such.

Recent years have seen an upsurge in interest in narrowing the gap between human-computer interaction (HCI) and software engineering (SE) perspectives, particularly in regards to object-oriented methods [1]. The gap is, of course, multifaceted, with differences relating to historical evolution, training, professional orientation, and technical focus, as well as in methods, tools, models, and techniques. Although the academic, research, conceptual, and professional aspects of this gap are interesting in their own right, in this paper we shall focus on pragmatic matters in integrating the means and methods by which software, with its human interface, is engineered.

2. Designing the Human Interface

Responsibility for human interface design, while eschewed by software engineering, is claimed by a number of professions with varying perspectives. Terminology is often marked by vehement debate over the professional responsibilities of graphic designers versus visual designers or the distinctions between user interface design and interaction design or the precise domain covered by information architecture.

In the final analysis, however, the engineering of the interfaces between humans and software requires solving design problems in three intimately interrelated areas: architecture, presentation, and interaction. The architecture of the interface refers to the overall and large-scale organization of the interface, that is, the way in which the interface as a whole is partitioned into distinct and recognizable regions or parts, how these parts are related and interconnected, and how the user navigates among these parts. Presentation design addresses the specification of how information is presented to users by visual, audible, haptic, or other means. In conventional graphical user interfaces, presentation design involves the selection and design of visual elements as well as their layout within and distribution among the various parts of the user interface, as well as details of appearance, such as, color and shape. Interaction design addresses the specification of the means by which users interact with a system and includes the organization of discrete steps in processes, the selection or specification of gestures or idioms for interaction, the sequencing of actions, and workflow.

These three aspects of interface design interact strongly and are ultimately inseparable, even if some designers and methods strive to address them independently. The choice of a particular selection component, for example, implies constraints on appearance and forms of interaction with effects on both layout and the use of "screen real estate," which in turn may impact what features can be combined within a single part of the interface and which must be separated.

While we are keenly aware that others have used other terms and definitions, in this paper we will use the term human interface engineering in preference to user interface design and will consider it to embrace the three areas of interface architecture, presentation design, and interaction design.

Ultimately, any integrated approach must not only address these three areas but also the design of the supporting software as such. In the usage-centered software engineering approach described in this paper, a single set of models forms the basis of integration and coordination of human interface and software engineering.

3. Usage-centered or user-centered?

Usage-centered design [2, 3] is a systematic, model-driven approach to human interface engineering for software and Web-based applications. Beginning with early work on task modeling based on use cases [4, 5], it has evolved into a sophisticated process that has proved itself on projects of widely varying scope and scale in a variety of application areas [3, 6, 7, 8, 9]. Because of its focus on designing software for use based on robust task models, the approach has proved particularly effective in delivering innovative—even award-winning—designs that both enhance performance and are easily learned [9, 10, 11].

Usage-centered design can be distinguished from the more widely recognized and practiced user-centered design [12]. In one form or another, the latter has become the dominant design philosophy of human-computer interaction. As one indication of this dominance, a Google search on user-centered design yields over 74,000 hits, in contrast to about 2,000 hits for usage-centered design. Nevertheless, the application and influence of usage-centered approaches has grown steadily, and its core concepts and techniques have been incorporated into other methods, including the Rational Unified Process [13, 14] and even in some cases under the rubric of user-centered design [15, 16].

The tensions between software engineering and user-centered methods are widely recognized [17]. An emphasis on thorough investigation and extensive field observation, particularly in ethnographic approaches, combined with repeated cycles of prototyping and user feedback, leads to a substantial, often unpredictable, analysis and design commitment that typically must precede software design and development. Usability testing, a cornerstone of user-centered approaches, typically comes late in the development cycle when its findings are all too often ignored or deferred for later releases, especially when testing uncovers architectural defects or problems in the basic organization of the user interface.

Usage-centered design differs from its older and better established counterpart in several important ways. As the name suggests, the center of attention is not users per se but usage, that is, the tasks intended by users and how these are accomplished. This difference in emphasis is reflected in differing practices that have a significant impact on the development life cycle and on integration with software engineering. Table 1 summarizes salient differences in the two approaches.

Unlike user-centered design, whose methods and traditions are firmly rooted in the human factors and human-computer interaction world, usage-centered design is grounded in a strong engineering orientation reflecting the background of its co-developers, including one of the early pioneer software methodologists (Constantine). Usage-centered design emerged directly from software engineering and particularly from object-oriented software engineering [18]. Precisely because usage-centered design is built around extensions and refinements to well established software engineering models and techniques, such as actors and use cases [19], integration with software engineering is more straightforward.

Perhaps the most fundamental difference is in the basic organization of the design process itself. Underlying much of user-centered design as practiced is a view of user interface design as a process of successive approximations wherein a final solution emerges

Table 1. Salient Differences between User-Centered and Usage-Centered Design

User-Centered Design	Usage-Centered Design
Focus on users: user experience, user satisfaction	Focus on usage: improved tools supporting task accomplishment
Driven by user input	Driven by models
Substantial user involvement: user studies, participatory design, user feedback, user testing	Selective user involvement exploratory modeling, model validation, structured usability inspections
Descriptions of users, user characteristics	Models of user relationships with system
Realistic or representational design models	Abstract design models
Design by iterative prototyping	Design through modeling
Varied, often informal or unspecified processes	Systematic, fully specified process

principally through repetitive cycles of trial design alternated with user testing and feedback. Although the profession may be loathe to admit it, such approaches are appropriately described as design by trial-and-error.

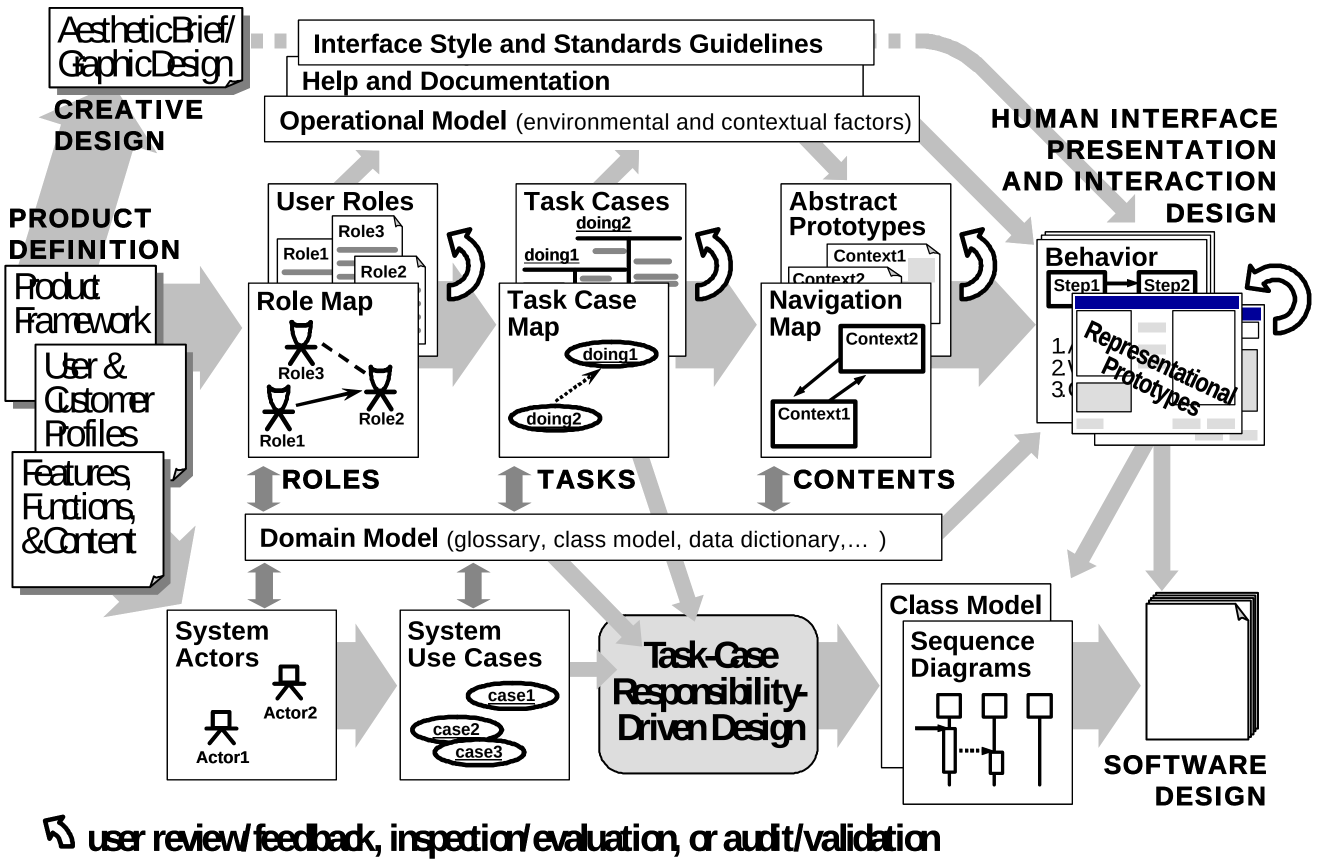
In contrast, usage-centered design is conceived as an integrated concurrent engineering process aimed at producing an initial design that is essentially right in the first place. This is not to say that refinement and improvement through evaluation and feedback are not employed, but these are not the driving forces in the design process. Instead, usage-centered design is driven by interconnected models from which a final visual and interaction design are derived more or less directly by straightforward transformations. Iterative refinement applies more to the models from which the final design is derived than to the design.

4. The usage-centered design process

The process, outlined schematically in Figure 1, is based on concurrent engineering. Although the core models are connected in a logical sequence, in reality experienced practitioners develop them concurrently, moving from model to model as information and insight emerge and as the needs of project management and problem solving dictate. In addition, the process divides into concurrent but interdependent threads, one primarily focused on designing the human interface, and the other primarily focused on designing the internal software.

A number of variants to this process and its models have been devised to suit various contexts and varying degrees of formality, ranging from highly structured forms [2, 9, 20] to agile design for Web-based applications [3, 21], and even incorporation with extreme programming [22].

In lieu of more unstructured or open-ended investigation typical of contextual inquiry or other ethnographic techniques, a model-driven process is used even for problem definition and requirements gathering In this exploratory modeling [23] (not shown in the diagram), questions and areas for investigation are identified by constructing provisional user role and task models based on whatever background information or knowledge is at hand. Limited, sharply focused inquiries, observation, surveys, or investigations are then conducted to answer questions, resolve ambiguities, and supply



Aesthetic Brief/ Graphic Design
CREATIVE DESIGN
Interface Style and Standards Guidelines
Help and Documentation
Operational Model (environmental and contextual factors)
HUMAN INTERFACE PRESENTATION AND INTERACTION DESIGN
PRODUCT DEFINITION
Product Framework
User & Customer Profiles
Features, Functions, & Content
User Roles
Role1
Role2
Role3
Role Map
Task Cases
doing1
doing2
Task Case Map
Abstract Prototypes
Context1
Context2
Navigation Map
Behavior
Step1
Step2
Representational Prototypes
ROLES
TASKS
CONTENTS
Domain Model (glossary, class model, data dictionary,…)
System Actors
Actor1
Actor2
System Use Cases
case1
case2
case3
Task-Case Responsibility-Driven Design
Class Model
Sequence Diagrams
SOFTWARE DESIGN
user review/feedback, inspection/evaluation, or audit/validation

Figure 1. Schematic Outline of Usage-Centered Design Process

missing information as identified through the exploratory modeling.

4.1. Usage-centered models

Presentation and interaction designs, as is typical, are embodied as representational prototypes, that is, paper or other prototypes representing or approximating the actual appearance of the user interface. Usually these take the form of annotated drawings or renderings of screens, pages, or other interaction contexts supplemented by descriptions of the behavior of the interface in narrative or other convenient form, such as, flow charts or decision tables.

In the usage-centered process, the presentation and interaction designs derive directly from the contents of the three tightly integrated core models: a user role model, a user task model, and an interface content model. Although some form of user and task modeling is common to most user-centered approaches, usage-centered models are distinctive in the use of abstraction and simplification to focus precisely on matters most essential for informing the design process.

4.1.1. User roles. Rather than modeling actual users or user communities broadly, as is often the case with user-centered techniques, the user role model focuses exclusively on salient aspects of the relationships between users and the system as represented by the various roles users assume. In its simplest informal form, the role model describes context, characteristics, and criteria, that is, the context within which each role is assumed including the overall contextual purpose and responsibilities of the role, the characteristics of interaction within the role, and criteria for support of the role, such as specific functions or the relative importance of various design objectives. Relevant characteristics of roles have been cataloged and compiled into templates for structured modeling of user roles [2]. In practice, a simple summary of context, characteristics, and criteria on an ordinary index card may suffice in many projects. For example,

Routine-Test-Running Role

CONTEXT: Unsophisticated but trained operator runs predefined standard tests with limited modification (under supervision).

CHARACTERISTICS: frequent, regular performance of relatively simple but crucial repetitive tasks.

CRITERIA: simple, efficient, error-free operation.

4.1.2. Task cases. Developed originally by Ivar Jacobson for his object-oriented methodology [18], use cases have become ubiquitous in modern software engineering practice. As originally conceived and conventionally employed, use cases comprise "sequences of actions…that a system…can perform by interacting with outside actors" [24]. They are usually expressed in terms of sequences of user actions and system responses.

For guiding human interface design, the focus on the system and on concrete action and interaction proved to be problematic, leading to the development of task cases, a more abstract form modeling user intentions rather than actions and system responsibilities rather than responses. An example is shown in Figure 2. Task cases, also known as essential use cases, are expressed from the perspective of users in roles.

Running Standard Test

USER INTENTIONS	SYSTEM RESPONSIBILITIES
	1. show available standard tests
2. pick test	
3. optionally [modify test] }	4. show test configuration
5. confirm & start	6. run test
	7. report results

Figure 2. A task case or essential use case.

Because task cases are essential in the sense introduced by McMenamin and Palmer [25]—that is, abstract, simplified, generalized, and technology- and implementation-independent—they come closer to the essence of user needs than typical use cases, offering advantages for requirements modeling [26] and for reducing the complexity of sophisticated systems [9]. The advantages of abstraction and goal orientation in use cases are widely accepted [3, 27, 28, 29, 30].

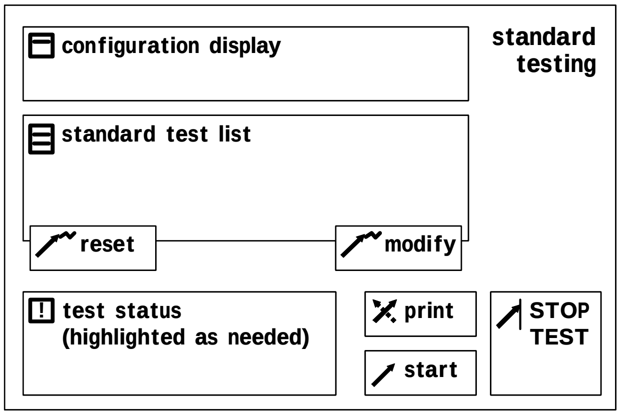
In practice, task case models are more fine-grained than typical use case models for software engineering. Increased granularity has advantages for user interface design in exposing small, reusable tasks and in clarifying the interrelationships among user tasks. This facilitates devising a more robust and flexible interface architecture that closely conforms to the underlying structure of user task needs.

4.1.3. Content models. User-centered designers typically move quickly into sketching paper prototypes, but in usage-centered design, content modeling is used to first work out the architecture or overall organization of the interface and its subparts independent of their realization as actual interface components and apart from the details of appearance and behavior. The interface content model consists of a set of abstract prototypes representing the contents of the various parts of the user interface and a navigation map representing the interconnections among all these parts.

Abstract prototypes [31] range from simple content inventories or lists of the contents of each part of the user interface to highly structured and formalized abstract prototypes based on canonical abstract components [32].

Abstract prototyping facilitates exploration and problem solving regarding the contents and organization of the various component parts of the interface without having to plunge ahead into details of appearance and layout. In their preferred form, abstract prototypes are constructed from a canonical collection of abstract components that allow for precise description of interfaces in terms of form and function independent of details of appearance and behavior and that promote the recognition of common patterns and problems in presentation and interaction design. An example of a canonical abstract prototype is shown in Figure 3.

Figure 3. Example of an abstract prototype using



canonical abstract components.

5. Interrelationships among models

In the process model of Figure 1, models form the bridge between human interface engineering and software engineering, providing traceability and interdependence and serving as the means for coordinating concurrent processes.

Software engineering typically begins with some form of high-level representation of the system scope in terms of the system boundary and interfaces with the environment. In the contemporary unified process and related methods, a use case diagram serves this purpose by identifying all the actors interacting with the system.

An actor, as the term is used in UML and the UP, is any entity that interacts with a system. For a usage-centered approach, non-human actors, referred to as system actors, are distinguished from user actors. Indirect users, whose interaction with the system is mediated by other users, are excluded, as they are part of the context of user actors (direct users) and are not directly involved with the user interface. User actors interact with the interface within the roles they play in relation to the system. A given actor may play in a number of different roles and the same role may be played by more than one actor.

Task cases support user roles. A given role typically requires a number of task cases, and a given task case may support multiple roles. On the software engineering side, the picture is more straightforward: system actors are supported by system use cases. Both task cases and system use cases become input for the object design, which must support them all to meet requirements. However, many details of the required software may not be determined until the presentation and interaction design are complete.

The presentation and interaction designs are based on abstract prototypes of the various interaction contexts and the navigation map modeling the interrelationships among distinct contexts. Particular user interface components with specific behavior and appearance are chosen or designed to realize the abstract components.

Abstract components are incorporated into abstract prototypes based on the particular tools and materials needed to realize each step in those task cases to be supported together within a given part of the user interface. The partitioning of the total user interface into subparts, as represented by the navigation map, is determined based on the interrelationships among task cases. A task case map in its most specific form models inclusions, extensions, and specializations relating task cases, but for many projects a clustering based on the likelihood of task cases being used together suffices.

The domain model, which captures the core concepts of the application, is developed and refined concurrently and collaboratively throughout the process and serves to link the various design models. For example, object classes referred to in the narrative body of task cases correspond to those in the contents of abstract prototypes on which the realized presentation and interaction design are based. The domain model in turn maps to the internal object design.

From this overview it should be clear that use cases, as task cases or system use cases, form the common thread that interconnects the various models and activities in the process. Indeed, task cases can directly guide the organization and contents of documentation and online help [2, 33], thus providing uniform and comprehensive traceability throughout the delivered system.

6. Distributed responsibilities and software

For use within conventional object-oriented software engineering approaches, such as the Unified Process, task cases (essential use cases) need to be transformed into conventional or concrete use cases. Abstract narratives defined in terms of user intentions must be elaborated into a more complex and detailed "flow of events" that refers to actual user actions in relation to the user interface as designed. In many situations, differences in granularity require combining closely related task cases into a composite concrete use cases that incorporates many if not all of the exceptional or alternative flows. Because this process of translation begins with task cases in the abbreviated form understood by the human interface designers, the detailed concrete use cases needed for software engineering must be developed by the human interface designers. Although this somewhat tedious translation introduces an added step with its attendant resource needs and opportunities for error, it has been used successfully on even very large projects [8].

The abstraction to intentions and responsibilities in the defining narrative of task cases suggests a possible more direct link to object-oriented software engineering by way of responsibility-driven design [34] and so-called CRC cards [35], both of which employ the concept of responsibility to guide software design decisions. Indeed, as has been noted [26], the role of responsibility in task cases is entirely consistent with the role of responsibility in software design. Both describe behavior without describing implementation, a commonality that enables linking the methods for determining requirements, for designing human interfaces, and for designing software.

The distributed responsibility technique [26, 36] developed at Victoria University of Wellington takes task cases and system use cases in the same essential form as used for requirements definition and human interface design and uses them to drive the derivation of an object-oriented design. This technique has several demonstrated advantages. The abstraction and brevity of essential use cases allows a more agile start to early object modeling and makes possible conducting object modeling and user interface design in parallel. The key role of responsibility in both task models and object models supports traceability between the two models, as well as providing better operational guidance to developers and facilitating review processes in design evaluation.

The iterative process begins with a system object, a single class whose responsibilities are those identified as system responsibilities in the task cases and system use cases, which means the design is guaranteed to provide the behavior specified in the requirements. In successive iterations, additional classes are identified as suggested by repeated terms within the class responsibilities or by virtue of their inclusion in the evolving domain model.

As classes are identified, responsibilities are delegated and the resulting provisional model is checked for reasonableness and refactored as appropriate. CRC cards are often used as convenient tools for representing classes, responsibilities, and collaborations. For greater precision, sequence diagrams can also be developed to clarify and detail responsibilities and collaborations.

7. Prospect

The usage-centered software engineering process outlined here is not a proposal but an already well established "industrial strength" process that has proved successful in numerous projects ranging up to 50+ person years completed by organizations around the world. Current areas of continued investigation include refinement in notation, compilation and elaboration of patterns based on canonical abstract components [32] and task cases [26], and continued work on common theory and metrics, such as cohesion and coupling, underlying both object design and human interface design [2, 37, 38, 39].

Perhaps most pressing is the need for tools that support usage-centered software engineering by incorporating its models and exploiting their interconnections for flexible concurrent modeling and systematic requirements tracing. The dominance of contemporary software engineering standards, in particular UML [24], is itself an impediment, as UML lacks constructs for user roles, task cases, and interface contents, forcing practitioners and process mentors to emulate the needed models in a procrustean bed constructed of UML "stereotypes" that are ill-suited to the purpose [13].

8. References

[1] van Harmelan, M. *Object Modeling and User Interface Design.* Addison-Wesley, Boston, 2001.

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

[3] Constantine, L. L., and Lockwood, L. A. D. "Usage-Centered Engineering for Web Applications." *IEEE Software, 19* (2), March/April. 2002.

[4] Constantine, L. L. "Essentially Speaking." *Software Development, 2* (11), November 1994.

[5] Constantine, L. L. "Essential Modeling: Use Cases for User Interfaces," *interactions 2* (2), March/April 1995.

[6] Anderson, J., Fleek, F., Garrity, K., and Drake, F. "Integrating Usability Techniques into Software Development. *IEEE Software, 18* (1), January/February, 2001.

[7] Patton, J. "Extreme design: Usage-Centered Design in XP and Agile Development." In L. Constantine (Ed.), *forUSE 2002: Proceedings of the First International Conference on Usage-Centered, Task-Centered, and Performance-Centered Design.* Ampersand Press, Rowley, MA, 2002.

[8] Strope, J. "Putting Usage-Centered Design to Work: Clinical Applications." In L. Constantine (Ed.), *forUSE 2002: Proceedings of the First International Conference on Usage-Centered, Task-Centered, and Performance-Centered Design.* Ampersand Press, Rowley, MA, 2002.

[9] Windl, H. "Designing a Winner: Creating STEP 7 Lite with Usage-Centered Design." In L. Constantine (Ed.), *forUSE 2002: Proceedings of the First International Conference on Usage-Centered, Task-Centered, and Performance-Centered Design.* Ampersand Press, Rowley, MA, 2002.

[10] Windl, H., and Constantine, L. "Performance-Centered Design: STEP 7 Lite." Winning submission, Performance-Centered Design 2001, http://foruse.com/pcd/

[11] Constantine, L. L., and Lockwood, L. A. D. "Instructive Interaction." *User Experience, 1* (3), Winter 2002.

[12] Norman, D. A., and Draper, S. W. *User-Centered System Design.* Erlbaum, Hillsdale, NJ, 1986.

[13] Heumann, J. "Use Cases, Usability Requirements, and User Interfaces." Tutorial Notes, OOPSLA 2002, 4-8 November. ACM, New York, 2002.

[14] Kruchten, P., Ahlqvist, S., and Byland., S. "User Interface Design in the Rational Unified Process." In M. van Harmelen, ed., *Object Modeling and User Interface Design.* Addison-Wesley, Boston, 2001.

[15] Scanlon, J., and Percival, L. "User-Centered Design for Different Project Types, Part 1: Overview of Core Design Activities." IBM DeveloperWorks, March 2002. ftp://www6.software.ibm.com/software/developer/library/us-ucd.pdf

[16] Percival, L. and Scanlon, J. "User-Centered Design for Different Project Types, Part 2: Core Design Activities by Project Types." IBM DeveloperWorks, March 2002. ftp://www6.software.ibm.com/software/developer/library/us-ucd2.pdf

[17] McCoy, T. "Letter from the Dark Side: Confessions of an Applications Developer. interactions 9 (6), November/December, 2002: 11 - 15.

[18] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Reading, MA, 1992.

[19] Constantine, L. L., and Lockwood, L. A. D. "Structure and Style in Use Cases for User Interface Design." In M. van Harmelan (Ed.), *Object Modeling and User Interface Design.* Addison-Wesley, Boston, 2001.

[20] Armstrong, C., and Underbakke, B. "Usage-Centered Design and the Rational Unified Process." In L. Constantine (Ed.), *forUSE 2002: Proceedings of the First International Conference on Usage-Centered, Task-Centered, and Performance-Centered Design.* Ampersand Press, Rowley, MA, 2002.

[21] Constantine, L. L. Process agility and software usability: Toward lightweight usage-centered design. *Information Age, 8* (8), August 2002. Also in L. Constantine (Ed.), *Beyond Chaos: The Expert Edge in Managing Software Development.* Addison-Wesley, Boston, 2001.

[22] Patton, J. "Extreme Design: Usage-Centered Design in XP and Agile Development. In L. Constantine (Ed.), *forUSE 2002: Proceedings of the First International Conference on Usage-Centered, Task-Centered, and Performance-Centered Design.* Ampersand Press, Rowley, MA, 2002.

[23] Windl, H. "Usage-Centered Exploration: Speeding the Initial Design Process." In L. Constantine (Ed.), *forUSE 2002: Proceedings of the First International Conference on Usage-Centered, Task-Centered, and Performance-Centered Design.* Ampersand Press, Rowley, MA, 2002.

[24] Rumbaugh, J., Jacobson, I., and Booch, E. G. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, MA, 1999.

[25] McMenamin, S. M., & Palmer, J. *Essential Systems Analysis.*: Prentice Hall, Englewood Cliffs, NJ, 1984.

[26] Biddle, R., Noble, J., and Tempero, E. "From Essential Use Cases to Objects." In L. Constantine (Ed.), *forUSE 2002: Proceedings of the First International Conference on Usage-Centered, Task-Centered, and Performance-Centered Design.* Ampersand Press, Rowley, MA, 2002.

[27] Cockburn, A. "Structuring Use Cases with Goals," *Journal of Object-Oriented Programming,* September/October 1997 and November/December 1997.

[28] Graham, I.(1996) "Task Scripts, Use Cases and Scenarios in Object-Oriented Analysis," *Object-Oriented Systems 3* (3), 1996.

[29] Kaindl, H. "An Integration of Scenarios with Their Purposes in Task Modeling." Proc. Symposium on Designing Interactive Systems. ACM Press, Ann Arbor, 1995.

[30] Lee, J., and Xue, N. "Analyzing User Requirements by Use Cases: A Goal-Driven Approach." *IEEE Software, 16* (4) July/August 1999.

[31] Constantine, L. L. "Rapid Abstract Prototyping." *Software Development, 6,* (11), November 1998. Reprinted in S. Ambler and L. Constantine, eds., *The Unified Process Elaboration Phase: Best Practices in Implementing the UP.* CMP, Lawrence, KS, 2000.

[32] Constantine, L. L., Windl, H., Noble, J., and Lockwood, L. A. D. "From Abstraction to Realization: Abstract Prototypes Based on Canonical Components." Working Paper, The Convergence Colloquy, July 2000.
http://www.foruse.com/articles/canonical.pdf

[33] Lynn, R. "Beyond Code Freeze: Use Cases that Do Not Leave Documentation and QA in the Cold." In L. Constantine (Ed.), *forUSE 2002: Proceedings of the First International Conference on Usage-Centered, Task-Centered, and Performance-Centered Design.* Ampersand Press, Rowley, MA, 2002.

[34] Wirfs-Brock, R. J., and McKean, A. Object Design: Roles, RESPOnsibilities, and Collaborations. Addison-Wesley, Boston, 2002.

[35] Wilkinson, N. *Using CRC Cards - An Informal Approach to OO Development.* Cambridge University Press, London, 1995.

[36] Biddle, R., Noble, J. and Tempero, E. "Essential Use Cases and Responsibility in Object-Oriented Development. In M. Oudshoorn (ed.), *Proceedings of the Australasian Computer Science Conference (ACSC2002).* Australian Computer Society, Melbourne, 2002.

[37] Constantine, L. L. "Visual Coherence and Usability: A Cohesion Metric for Assessing the Quality of Dialogue and Screen Designs." In Grundy, J., & Apperley, M. (eds.) *Proceedings, Sixth Australian Conference on Computer-Human Interaction.* IEEE Computer Society Press, Los Alamitos, CA, 1996.

[38] Constantine, L., and Noble, J. "Interactive Design Metric Visualization: Visual Metric Support for User Interface Design." In Grundy, J., & Apperley, M. (eds.) *Proceedings, Sixth Australian Conference on Computer-Human Interaction.* IEEE Computer Society Press, Los Alamitos, CA, 1996.

[39] Constantine, L. L. "Usage-Centered Software Engineering: New Models, Methods, and Metrics." In Purvis, M. (ed.) *Software Engineering: Education & Practice.* IEEE Computer Society Press, Los Alamitos, CA, 1996.

A Software Prototyping Framework and Methods for Supporting Human's Software Development Activities

Jennifer Z.Guan   Luqi
Software Engineering Automation Center
Naval Postgraduate School
833 Dyer Road, Monterey, CA 93940, USA
{zguan, and luqi}@nps.navy.mil

Abstract

Software development environment is a platform for supporting software designer to design software based on the software requirement specification. It is an interactive system with lots of human being involved. Human error, as the main threat for the dependability of the software development system, may greatly harm the quality of the produced software. In this paper, we present a practical framework for software prototyping, addressing two types of threats for the software prototype that will occur during designer's prototype efforts. By illustrating information with multi-level representation, building an iterative prototyping loop, and providing solid project management, the human prototyping efforts is enhanced, refined, and organized. Several approaches to make the specification and prototype of the software requirement to be more accurate, complete, and consistent are proposed. They prevent the occurrence of human omission/slip and help to recovery the system states from human mistake.

1. Introduction

An interactive system is the software whose purpose is to support two-way communication between a computer and its user. Typically, the user will perform an action on the system such as clicking a mouse button, typing text, or moving a joystick. Following this, the system will react accordingly, by invoking an application, displaying information on a screen, or simply waiting for yet another user action. The bi-directional communication between human and the computer supports the information exchange and helps to formulize knowledge and generate the result artifact.

Software development environment, or computer aided software environment is used to support software designer to design software based on the software requirement description. Its main responsibility is to provide representation of the software models and generated artifacts. It also displays the interface of tools in each of the development stages for user to interaction with. The generation, modification, and validation of the software artifacts, such as software specification, model or final program, can be done by manipulation of the interface elements of the tool in the environment. This kind of software tools or environment, as an intermediate tool for human to manipulate and interact with the computer artifacts, has high priority of requirement to be designed with the features of an interactive system.

The requirements for a software engineering tool, as an interactive system, are becoming broader than before. Not only the quality of the system, but also the usability aspect should be considered in the design of the software engineering tool. For usability issues, providing the people with an easy to understand, easy to use, seldom failure, and a friendly manipulation process can greatly increase the efficiency of the software development and decrease the risk of the project failure. The satisfactory of these properties are important for guarantee the operation of the system and the quality of the designed software.

A system that provides the user a satisfactory feeling of the operation and a satisfactory of the result product is called "dependable" system. Based on the definition of the "dependability" in [1], we extended the concept and defined a dependable interactive system as "a system on which the user's behavior can be performed with high reliability, and user can justifiably put reliance on certain aspects of the quality of service system delivers". By increasing the usability of the software development environment, therefore increasing the dependability, users can put more trust on the system operation, and have more endurance to the occasionally occurred software errors or failures.

In the software development process, the software requirement process has been proved to be the most human intensive process. Lots of human being is involved in the requirement elicitation, specification, and the analysis. The human participation will help the description of the requirement to be more complete, the formulization of the software requirement specification to be more accurate, the depiction of software functionality to be more consistent.

Among lots of approaches for the requirement acquiring and formulization, computer aided software prototyping has been proven to be an economic way to build a small-scale software system [8]. By building an executable version of the envisioned software, the clarified software requirements and the designed software models can be evaluated and adjusted before more efforts have been put into the detailed software implementation.

To ensure the quality of the produced software prototype and improve the prototype dependability, prototyping efforts need to be carried out with quality assurance methods. The correctness of the computing activity performance needs to be ensured. The effectiveness of the interactions between software development tools and human operators needs to be maintained and improved. The robustness of the performance of software design and prototyping tools needs to be sustained.

We had several year experiences and achievements on the computer aided prototyping research and application, the Computer Aided Prototyping System (CAPS) has been successfully used to build a software prototype automatically and clarify software requirements. We extend our foundational works and propose an expanded software prototyping framework based on the consideration of the human factors in the software prototyping process. To enhance the usability of the prototyping efforts, which therefore ensure the resulted prototype, our more research is concerning and focusing on the avoidance of the occurrence of the human errors and the recovery of the occurrence of human mistakes during the iterative prototyping process.

The prototyping framework developed here is an appropriate for the software development process that may include lots of human participations and collaborations. The iterative prototyping process provides the support for gradually accumulation of the software prototype models and the basis for the step-by-step refinement of the software requirement in terms of the feedback from the users about the performance of the software prototype.

The proposed several incorporated approaches, to at most extension, eliminate the potential risk of the error occurrence, is very useful and applicable to be integrated into a wide range of software development environment.

The paper is organized in five sections. In section 2, we briefly describe the related research work in the domain of computer aided software prototyping and human error. Section 3 introduces the framework of software prototyping with targeting at human erroneous efforts. Section 4 shows how we considered about how to resist the human erroneous actions. In this section, several design principles are described and we present the implementation of several approaches in the newly updated Computer Aided Prototyping System (CAPS-PC). In section 5, we draw our conclusions and discuss about future work.

2. Related Works

2.1. Computer Aided Prototyping

With increasing of the software complexity and scale, prototyping tends to be more than a simple set of techniques for software project development. It is called “prototyping methodology”. Several of activities are included in the prototyping methodology. They are modeling of prototype, evaluation of constraints, and automated program generation, which integrated as a whole to be the basis for prototyping methodology. When performing prototyping, several complex aspects are needed give more consideration with carefulness. To support the human prototype effort, a user-friendly prototyping tool is a prerequisite for larger use of prototyping base development [6][7][8].

As [2] indicated, the design and implementation of complex software-intensive system tend to product various errors at every development stages. For example, natural language is used to formulate requirement during requirements and design specifications in the early parts of the development. It would potentially result misunderstanding between users and designer of the future system and would produce errors that will have certain influence to reliability, safety, and cost of the system [8]. The software specification will possibly be interpreted in various ways, leading to “integration problem”[3].

Prototyping, as an economic way to build a small scale of software models and products, has been proved to be an efficient and effective methodology to evaluate proposed systems if acceptance by the customer or the feasibility of developments is in doubt [2].

2.2. Human Error

Since 1880, human error has been studied by lots of psychologists and biologists. It has shown that human error is the first source of damage of system dependability. But the analysis of human error has always stayed in the statistical analysis and quality description phase. Study on how to apply the result of human error analysis into real system development is very important to the increasing requirement of system dependability, which has rarely been touched before.

2.2.1. Human Error Constraints in Software Development. Peters define human error in the following ways [13]: "any significant deviation from a previously established, required or expected standard of human performance". Whereas, Sanders and McCormick [11], from the effect of human error point of view, gave a definition of human error as "an inappropriate or undesired human decision or behavior that reduces or has the potential for reducing effectiveness, safety, or system performance". According to the definition of Sander, a standard for evaluation of action as human error is if this action has undesirable or potential effect on the system criteria, possibly on people's safety. Also, Sanders thinks an error that is corrected before causing any trouble is still an error because it has the potential of causing an adverse effect on human or system criteria.

Johnson referred to three myths of human error that are often cited as barriers to the practical application of human error analysis [12]. Human error is described to be inevitable, unpredictable, and costly endurance. Following Johnson's original proposed problems, we gave a further discussion and possible solution for each of them, especially for human activities in software development.

A. <u>Human error is closely related with working situation;</u> from the psychology point of view, the occurrence of human errors depends on the environment and the user's understanding of situation, which provides the context for the user to derive from the supposed correct actions. Many researches have showed that, with correct and consistent understanding of the operation situation context, the human error can be prevented.

<u>B. Human error can be predicted and removed by get enough knowledge about the interacted system;</u> it is difficult to predict when and where a human will make an error. However, it is possible to predict and remove many of the local conditions that create the opportunity for user's error to have disastrous consequences, such as reducing inattention and fatigue.

<u>C. Human error can be guard and avoid by providing user satisfactory operation conditions;</u> normally, it is too expensive for companies to employ the analysis and prevention techniques that reduce the human contribution to major accidents. But, a feasible and useful way to reduce the human error is trying to enhance human being's understanding of system operation logic and improve the system's error endurance level. Some typical approaches for error endurance include the error prevention, fault tolerance and fault recovery.

2.2.2. Classification of Human Errors. Classification of human errors can help us to understand the human error within particular environments and special circumstances. Human error normally can be classified according to information processing and discrete actions.

Information processing is conducted coordinately during task performance. Some steps are performed to achieve the task, which include the observation of the system state, hypothesis formulation and testing, goal choosing, procedure selection for achieving the goal, and execution of the procedures. The information processing criteria for human error classification is often used in the analysis of complex systems.

According to classification criteria of discrete action, human error can be catalogued into error of omission, errors of commission, sequence error, and timing error. Error of omission means that the user forgets to do a task. Errors of commission imply performing an action incorrectly. Sequence errors include the errors of performing actions or tasks in the wrong order. Timing errors mean that the user fails to act in the allotted time period, due to performing either too slow or too fast.

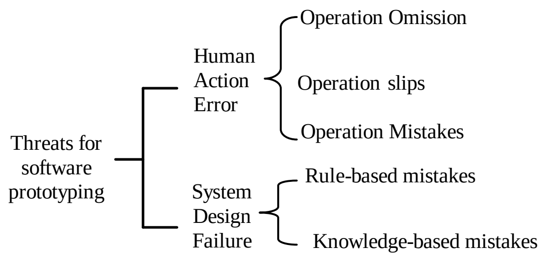


Figure 1. The classification of Human Threats for software prototype development

We define human threats to dependable software development in the context of the design and prototype efforts according to the effect of human discrete actions on the resulted software attributes. There are two types of source that would trigger the occurrence of errors. A human action error is an event where human action affects a part of the system state in a way that may cause a subsequent failure. Normally, the reaching of an error to the service interfaces that produces observable deviations from required service means the occurrence of a failure. A system design failure is an event that occurs when the system function service deviates from correct function service because of incorrect human action. A failure is a transition from correct service to incorrect service, such as not performing the required system function. These distinctions are illustrated in Fig. 1.

3. A software prototyping framework for supporting human design activities

Prototyping is a way of showing the idea behind a design in an inexpensive way. A prototype can be changed easily and the design evolved gradually to an optimal solution before the system is implemented. The idea behind prototyping is to save on the time and cost to develop something that can be tested with real users. Rapid prototyping has been found to be an effective technique for clarifying requirements. The early prototyping effort can provide the user a small-scaled software, which is more conceptual and understandable than that of the traditional software development approach which produce working code only near the end of the process.

Beside the simulation capability of the software prototype, we propose to adopt additional strategies for avoiding the context situation for error occurrence to improve the stability and robustness of the software. We proposed a framework for prototyping software with purpose to minimum the possibilities of error occurrence. Several human error avoidance methods and error recovery methods are designed, which can improve the user's understanding of software prototype and reduce the error of knowledge confusion.

Following our framework, an executable mini-system or a pilot version of an intended system will be built. The prototype can be treated as a partial representation of the target software, and can be used further as an aid in the design, analysis, validation, and verification of the system.

3.1. Prototyping Framework with Supporting of Error Elimination

Prototyping efforts are carried out from the step of "determine requirements", along the step of "construct prototype", to the step of "execute prototype". The resulted software executable prototype is the small scale of the designed software, which can be manipulated, tested, and verified. The real software can be built by performing the evolution of the approved software prototype. For each edges in this triangular diagram, several items are highlighted that act as the facilities to support the activities with duties of elimination of human errors or recovery of system derived state.

The original gathered natural language requirement are clarified and analyzed to provide a correct and consistent software descriptions for the prototype model formulization. By employing specification syntax error checking tool, the incorrect syntax structure can be modified to fit the software functional logics. By checking the data consistency between the higher-level components description with lower level module description, the omission of software functionalities during the complex system decomposition will be prevented. By providing context aware design hints in the prototype tool, the user can get a clear picture of whole system structure and the timeline of what has been done and what has not.

From the construction of the prototype to the execution prototype, design history retracing is useful for the design backward when it is found that a design effort is fail or the revise of past design efforts. Error recovery is to restore the software status from the mistaken status to formerly verified correct status by retracing the past design history. Run time testing is an execution effort to make sure the result prototype is executable. Several errors would be found when the designer was trying to execute the prototype.

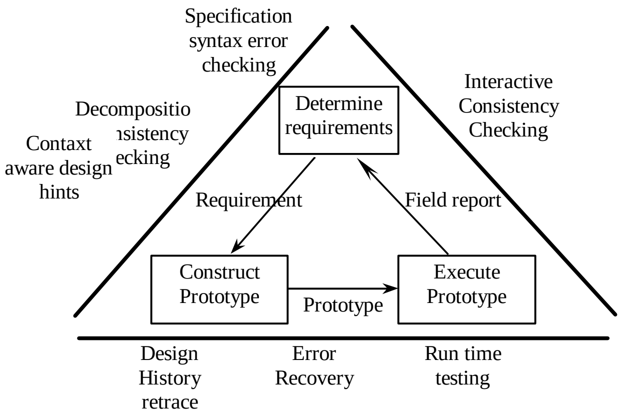


Figure 2. Framework for prototyping with supporting of error preventions

When the prototype is ensured to be executable, the checking of the execution functionalities of the software with the requirement description is very important for the end user's verification. Interactive consistency checking helps users to compare the performance of the prototype by the prototype interface with the imagined software in their brain to make sure what they got is what they want.

To enhance the capability of software prototyping and to make sure the generated prototype fit the requirement needs of the end user, we designed and implemented several methods to reduce the possibilities of the generation of the error, which therefore, will affect the quality of the resulted software. Some methods has implemented into the Computer Aided Prototyping System (CAPS-PC) developed in Naval Postgraduate School. Detailed descriptions and discussions for each method are presented in the following sections.

3.2. Multi-level Information Representation

The display of the too many information with classification and organization would reduce the efficiency of the information representation. The design model of the software prototype, although has been simplified, contains lots of information cohered with each designed module. To simplify the structure of the prototype model without the contained information, the software prototyping model can be displayed via multiple levels of view with different abstraction level and information hidden.

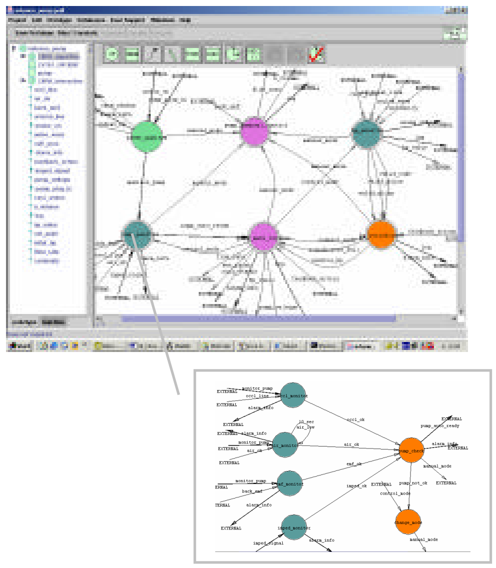


Figure 3. The decomposition of the second level of infusion pump prototype design

CAPS-PC provide multi-level point of view of the whole system design. The construction of the prototype begins with a top-level definition, followed with a number of sublevel refinement based on the functional decomposition. The high level of information illustration and lower level information decomposition prevent the designer from the information explosion and possible occurrence of design omission/slips. The designer can build the prototype with different granularity at different focus

3.3. Iterative Loop for Prototyping of Interactive System

The requirements are firmed up iteratively in a rapid prototyping approach through the examination of executable prototypes as well as by negotiations between customer and designer. The designer constructs a prototype based on the requirements, and examines the execution of the prototype together with the customer and potential users of the system.

The refinement of requirement is fulfilled by two kinds of approaches. From one hand, the requirements are adjusted based on the feedback from the customer, and the prototype is modified consequently until both the customer and the designer agree on the requirement. On the other hand, during the execution of the prototype, possible user's action can be generated to simulate human fault, which serves as an input to the system functionality or an anticipated feedback for a system function of an acquirement. Both of two methods are helpful to find and locate the software functional errors, on which further modifications can be carried.

4. User-centered Designs with Human Factor Considerations

An integrated software prototyping environment should not only have strong capabilities to building prototype, but also represent high support for usability of the prototyping effort. It is expected to improve the quality of the systems developer, reduce the time and cost of software development, enhance the developer's satisfactory and productivity, and in most extent make the software development a more delight and exciting task [11, 13]. Ease-of-use (usability) and ease-of-learning (Learnability) are two highlighted topic in the design and implement of an integrated development environment.

By considering the human factor in the design of CAPS-PC, we help the CAPS-PC user to correctly understand the functionalities carried on during the software prototyping. Five principles recommended in [11] are implemented in the design of CAPS-PC:

1) Making visible program artifacts and CAPS-PC functionalities when they are relevant and required. Irrelevant or rarely needed artifacts and functionalities compete with the relevant ones and diminish their relative visibility.

2) Minimizing the developer's memory load. Users of CAPS-PC don't need to remember functionalities and program components from one particular dialog or screen when they move to another area. The related artifacts and functionalities can be easily retrieved and made visible whenever they are needed

3) Speaking the developer's language. Although CAPS-PC used LAMPS as its bottom line language, which is a formal specification language as description in section 2, it is easy for user to understand the description of the control components and their supposed triggered action. The words, phrases and concepts displayed in the

CAPS-PC interfaces are illustrated by using common development language, which is comfortable and easy for developer to get acknowledge and familiar with.

4) Keeping the developer informed on the CAPS-PC status and the prototype being designed. During the prototype development process, contextualized feedback and appropriate message are displayed to developer at the time of happening, which efficiently inform the developer about the system status and the hints for possible consequential results.

5) Preventing developers from making mistakes. For a development environment, it would be an effective facility to prevent the problem from occurring in the first place. CAPS-PC used human error resistance strategies to prevent or limit the consequence of human error. Error resistance can be achieved by means of two strategies: error prevention and error handling. Error prevention can result from forcing functions, operator's selection, or training, which will not get rid of all human error occurrences. Then error handling mean to catch the error in the case of error occurrence, and control the evolution of error to minimize the error effect. To increase the usability of specifying and designing of system requirement model, CAPS-PC employed a backward error recovery strategy to reset the system state when an error or mistake occurs. Also, keeping of design histories and making them retraceable can remind the prototype developer the process of their past design steps, which can provide helpful information for developer to enforce most effective recovering approach.

4.1. Design History Retrace and Error Recovery

Backward error recovery is an attempt to restore the system state after an error has occurred. Backward recovery can be considered as the only real "recovery" function. The unexpected effects of errors will be totally removed with step by step backward until to the satisfied state.

Three kinds of backward error recovery commands, undo, cancel and stop, are implied in our prototyping tool, CAPS-PC.

Undo mechanism help user to go back the history of editing of the prototype specification and definitions of the designed software modules, interconnections, and their required properties.

The cancel function is used to abandon commands under specification. For example, a user can cancel the typing of a timing constraints property in a time critical module if it is checked to conflict with other timing constraints. Cancel function always to deal with in special situation that the user must know which command are concerned by the recovery.

The stop function is used to terminate a particular execution process. For example, in CAPS-PC, with the completed prototype model, the translation process can be triggered which will translate the prototype specification to generate pre-designed system skeleton and the control relationship between the modules. Once the process is trigger, the output of the execution will be displayed with the ongoing translation process. In the case of finding some mistakes before the process finished, users are always want to make modification right now. It will be necessary to stop the execution process and restore the initial state before translation.

CAPS-PC provides flexible mechanism to support the error recovery on time to prevent the effect of error. By employ multi-thread mechanism, translation, schedule, and compilation of prototype are triggered separately from the main editing of prototype graphic. During runtime execution of program, problems can be found and provided to user directly.

4.2. Automatic Syntax Error Checking

The grammar monitor for software requirement specification provides an excellent debugging support. Before saving the specification description, the syntax can be verified. In the case of syntax error, the grammar error information is displayed and selection suggestion is given on the interface for the user. This enhances the quality of the prototype produced.

When saving the requirement specification, the data flow diagram of the prototype will be checked to ensure that the data flow diagram has a correct input-output structure. Structural error messages are displayed to the user if the input of one module is fully used in its module functionality. An instant checking during each user's input is implemented in the prototype tool, CAPS-PC, to prevent human's operation omission/slips.

4.3. Interaction Consistency Check

During the design of software model, the interaction between the modules will be defined. The relationships between the modules include their data communication, input constraints and output constraints. All of this information would be defined in the connection-edge annotation. In the prototype specification annotation, edges are allowed to have the same name in the same level or in different levels of the tree structure. All the edges with the same name are regards as the same data stream and have the same property. The edge inconsistency problem may occur when the properties of the prototype edge will be changed in one of its instance [12]. We provide automatic consistency maintenance to check the differences between the interaction edge’s name, date type, and related time properties. The edges inconsistency checking include:

1) Dynamically detects the edges’ name changed and automatically fill in the edge property if it has the same name with en existing edge;

2) When any edge’s property changed and the OK button was clicked, searching all the edges existed to find every edge with the same name but different properties.

Once the prototype tool find the inconsistency between edges, an inconsistence table will display the detail information about the conflict edges, and provide the evaluation hints for user to select a correct property, which is showed in figure 4. When a verified property definition item is selected by the user, all the other edges that are list on the inconsistency table will be changed to be consistent with the verified edges.

4.4. Decomposition Consistency Check

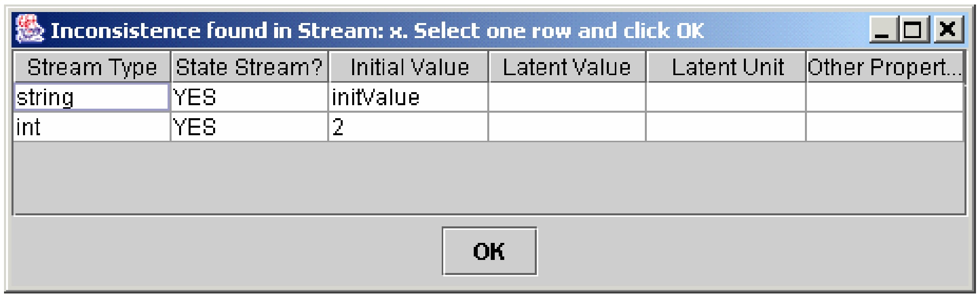
Based on the hierarchical design of software model, the high level model can be decomposed into low level model. The functionalities and properties can be detailed and realized in the decomposed second level. During the decomposition of software model, the high-level component definition should be consistent with lower level modules. Two types of consistencies should be maintained:

1) The number of data input/output of high-level component definition should be consistent with lower level data input/output. All the data input/output defined in high level should be realized into lower level as edges from or to terminal.

2) The properties of component defined in high level should be consistent with lower level properties definition. If the high level component For example, a data manipulation component is defined to have a periodic time constraints, then the lower level component event should also be triggered periodically. If Maximum Execution Time (MET) of the high level component is set to be 500ms, then the triggering condition of the low level component should be within the MET constraints, which means its maximum execution time should be less than 500ms.

4.5. Context Aware Design Hint

During the design of system, we provide several facilities to let the user know the current status of system and the coverage of designed prototype. Give more



Inconsistence found in Stream: x. Select one row and click OK

Stream Type	State Stream?	Initial Value	Latent Value	Latent Unit	Other Propert...
string	YES	initValue			
int	YES	2			

Figure 4. A checkable table for edge inconsistency properties

information for user to keep tracing of the design process will reduce the interruption in the design process. Contextualized feedback and appropriate message are displayed to developer at the time of happening, which efficiently inform the developer about the system status and the hints for possible consequential results.

During the design of software prototype, following information are keep updated and displayed to inform the developer:

1) The name of current prototype project and its version;

2) The current decomposition component and its located level;

3) The necessary to perform the saving of prototype according to the prototype modification actions every after saving the designed prototype;

4) Once the focus of mouse is put on a particular menu/icon/button, the related annotation of the interface component will be displayed beside the mouse focus. Its supposed action will be interpreted at the bottom of the interface of the prototype tool.

5. Conclusion

Prototyping is not recognized as a cornerstone of the successful construction of software system as it allows making users at the center of the development process. However, with the involvement of the human, prototyping tends to produce low quality software as no human factors or error resistance is undertaken. We have shown in this paper how methods of error checking and recovery can be integrated into the software prototyping framework and can contribute to the development process of software system through prototyping activities.

While the techniques for prototyping, such as specification, design, and code implementation, has reached a maturity level allowing moving with real application, the prototype tool with human factor consideration is still under development. A real application has been completely in the field of the CARA [19].

In addition, it has also shown the amount of work that is still required before the environment can be used by wider range of people. Several features is promising to be integrated into our prototype framework such as:

1) integration of property verification to ensure the inside accuracy of prototyped component functionality,

2) performance analysis in order to support more flexible technique selection, and

3) experience recovery with more widely time duration and task scope.

6. References

[1] J. C. Laprie. Dependable computing and fault tolerance: concepts and terminology. In Fault Tolerance Computing Symposium 15, pp. 2-11, Ann Arbor, MI, IEEE Computer Society, June 1985.

[2] Kutar, M., Britton, C., & Nehaniv, C. Specifying multiple time granularities in interactive systems. In Proceedings of Design, Specification and Verification of Interactive Systems Workshop, pp. 49-61. 2000

[3] James Reason, Human Error, Cambridge University Press, 1990

[4] Luqi, Valdis Berzins, Raymond T. Yeh, A Prototyping Language for Real-Time Software, IEEE Transaction on Software Engineering. Vol. 14, No. 10, Oct. 1988

[5] Rapid Evaluation of Interactive Systems, LTI-Annual Report 1999/00

[6] Nico Hamacher, Jory Marrenbach, Karl-Friedrich Kraiss, Formal Usability Evaluation of Interactive Systems, 8th IFAC/IFORS/IEA Symposium on Analysis, Design, and Evaluation of Human Machine System 2001, Sept. 18-20, Kassel, VDI/VDE-GMA, pp577-581, 2001

[7] Luqi, Real-Time Constraints in a Rapid Prototyping Language, Computer Language, Vol. 18, No.2, pp.77-103, 1993

[8] Luqi, Mohammad Ketabchi, A Computer-Aided Prototyping System, IEEE Software, pp.66-72, March, 1988

[9] Pete Broadwell, Naveen Sastry, Jonathan Traupman, FIG: A Prototype Tool for Online Verification of Recovery Mechanisms, ICS SHAMAN Workshop'02 New York, USA, 2002

[10] Sanders, M.S. & McCormick, E.J. Human Factors in Engineering and Design, 7th Ed., 1993, New York, McGraw-Hill

[11] Chris Johnson, Why Human Error Analysis Fails to Support Systems Development, Vol. 11, No. 5, Interacting with Computers, pp.517-524, 1999

[12] Peters G. Human Error: Analysis and Control, Journal of the ASSE, 1966.

[13] Sarter, N.B. & Woods, D. D., Strong, silent, and 'out-of-the-loop'. CSEL Report 95-TR-01, February, 1995

[14] Gail E. Kaiser, Cooperative Transactions for Multi-User Environments, Won Kim (ed.) Modern Database Management: Object-Oriented and Multidatabase Technologies, ACM Press, 1994

[15] Smith, D. E., & Marshall, S. P., Applying hybrid models of cognition in decision aids. In Zsambok, C. & Klein, G. (Eds.), Naturalistic Decision Making, Mahwah, NJ: Erlbaum, pp. 331-343, 1997

[16] David Navarre, Philippe Palanque, Remi Bastide, & Ousmane, A model-based tool for Interactive Prototyping of Highly Interactive Applications, pp 136-141, 2001

[17] Sanders, M and McCormick, E Human Factors in Engineering and Design (six edition) McGraw-Hill, New York, NY, USA , 1987

[18] Luqi, Ying Q., Lin Z., Computational Model for High-Confidence Embedded System Development, Montery Workshop 2002, Italy

[19] Luqi, Z. Guan, Requirements Document Based Prototyping of CARA software, to appear at International Journal on Software Tools for Technology Transfer