CLOSING THE GAPS:
Software Engineering and Human-Computer Interaction

Edited by

Morten Borup Harning
Jean Vanderdonckt

IFIP

Closing the Gap:

Software Engineering and Human-Computer Interaction

Edited by

Morten Borup Harning
Dialogical ApS/Open Business Innovation
Værløse
Denmark

and

Jean Vanderdonckt
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Published by Université catholique de Louvain, Institut d'Administration et de Gestion (IAG)
on behalf of the International Federation for Information Processing (IFIP)

Published by
Institut d'Administration et de Gestion (IAG),
Université catholique de Louvain (UCL)
Place des Doyens, 1
B-1348 Louvain-la-Neuve, Belgium

under the reference

Working Paper WP 89, September 23th, 2003.

Table of Contents

Introduction

Almost half of software in systems being developed today and thirty-seven to fifty percent of efforts throughout the software life cycle are related to the system's user interface. For this reason issues and methods from the field of human-computer interaction (HCI) affect the overall process of software engineering (SE) tremendously. Yet despite strong motivation amongst organizations to practice and apply effective SE and HCI methods there still exist major gaps of understanding both between suggested practice, and how software is actually developed in industry, and between the best practices of each of the fields. There are major gaps of communication between the HCI and SE fields: the methods and vocabulary being used in each community are often foreign to the other community. As a result, product quality is not as high as it could be, and (avoidable) rework is often necessary. In addition, SE methods and techniques are often perceived by HCI specialists as tools that are only re-served to computer scientists and of little or no relevance to HCI. And vice versa: HCI contents are often perceived by software engineers as after-thoughts or side-tools that do not necessarily affect the quality of software. For instance, no methodologies in the domain of object-oriented programming offer explicit support for HCI and existing HCI methods are integrated in development practices in a way that is more opportunistic than systematic.

The theme of this workshop is to attempt to enumerate and understand these gaps of under-standing and communication, with an eventual goal of proposing ways to bridge these gaps.

For instance, SE frequently employs requirements elicitation techniques that involve soft goals, procedures, and operators. HCI typically uses task modelling involving task, sub-tasks, and temporal operators between. While these two techniques are different in purpose, they are surprising close to each other.

This workshop can improve software engineering and HCI education and practice by raising awareness of HCI concerns among SE researchers, educators, and practitioners, and vice-versa. It can also show the places where an attention to concerns from one field can inform the other field's processes, and showing how methods and tools can be augmented to address both SE and HCI concerns.

Morten Borup Harning and Jean Vanderdonckt
Værløse and Louvain-la-Neuve, September 2003

http://www.se-hci.org/bridging/interact
http://www.interact2003.org/workshops/ws9-description.html

Architecture for Personal Text Entry Methods

Poika Isokoski & Roope Raisamo

TAUCHI, Department of Computer and Information Sciences,
University of Tampere, FIN-33014 Finland

{poika,rr}@cs.uta.fi

Abstract: Text entry is special kind of human-computer interaction in one respect: it is a highly learned activity, and unlike some others it needs to be so. Writing is not an inborn or trivially acquired skill like pointing and reacting to simple stimuli. Because learning a writing method is expensive, users benefit from using the same method in different situations. We describe an architecture that supports text entry methods that follow the user from device to device. The architecture consists of a server and text entry method implementations that the server downloads from the Internet. Text entry method selection is based on user preference and input device compatibility. We use Java for implementing the text entry methods so that the same code can be run on different platforms. The server is not similarly platform independent. Currently we have implementations for Linux and Microsoft Windows.

Keywords: Text input, device independence, personalization, Java.

1 Introduction

Text entry is an important task in human-computer interaction. A large part of our daily computer-mediated communication is done in writing. This is true even for devices that were originally intended for other forms of communication. For example, some people use mobile phones more for text messaging than for talking.

Some form of text entry is needed in many computing devices. Desktop computers, mobile phones, tablet PCs, two-way pagers, game consoles, and even TV sets have some way to enter textual information. The very different design goals of these devices lead to different choices of input devices, which results in a great variety of text input methods. Good examples of this are the various text input methods fashioned for the 12-key telephone keypad (MacKenzie and Soukoreff, 2002).

For a user with several computing devices, the need to learn how to enter text into all of them can become a problem. The rapid aging of the devices aggravates this problem. A new device generation may have a different text entry system. This means that the user has to go through the expensive and frustrating learning process repeatedly. Some users can cope with this learning burden, but many would rather not – at least if it can be avoided.

We propose solving this problem through a clearer separation of the device and the text entry method. Instead of being device-specific the text entry methods should follow the user. This can happen in at least two ways. One way is to have a physical personal text entry device such as a small keyboard or a digital pen that interfaces with the computing devices that the user encounters. The other way is to make the text entry software adaptive so that instead of bringing his or her own input device, the user can utilize the available input devices in the most familiar possible way. We have explored this latter option by implementing prototypes of a software architecture. Our architecture does not exclude the hardware solution or any of the existing text entry systems. Rather it is a new service that can offer advantages to those that choose to use it.

We will describe the benefits of our architecture by focusing on its good properties. These include:

- Support for different input devices and methods that are compatible with multiple devices.
- Multi-language support.
- Support for emerging text entry methods.
- Good environment for personalized text entry methods.

- Lower overall implementation effort.
- Coexistence with other systems.

These issues have been addressed in some ways in existing systems. Our architecture extends these capabilities.

First we describe the architecture and our current implementations. Then we explain the above-mentioned properties in more detail. Finally we discuss our experiences with the architecture so far and lessons learned in the implementation.

2 The Architecture

We describe our architecture in two parts. First, the essential main components that allow text entry methods to follow the user and then the peripheral components that are needed to support this functionality.

2.1 Main Components

The main idea – to make the text input methods follow the user – dictates that a text input method is a self-contained module that can be transferred between computing devices. In order for this to be possible the computing devices must be capable of running these modules. Thus, we have the two main parts of our system: text input modules and servers that can run these modules.

Figure 1 shows these two components in the normal operating mode. The server gives the module input events to process. The module translates the input into characters and then gives the characters back to the server. The scope of interest frame in Figure 1 shows the part of the system that has been in our main focus so far. We have been aiming to build a minimal pair of interfaces that would allow text entry modules to operate. This is also the part of the system that needs to be standardized to guarantee module compatibility between server implementations. Everything else can essentially be left to the implementers to do as they see most convenient.

Interaction Between the Server and a Module

A text input module translates input primitives such as keystrokes and pointer coordinates to a text stream. Each module is capable of dealing with a limited set of input primitives. For example, a module that does handwriting recognition cannot function if only keyboard input is available. Because of this, a module begins its setup task by querying the server for the available input devices. Depending on the response its input needs are either met or not met.

In addition to suitable input, the module must make sure that the server can accept the output that it produces. In a manner similar to the input queries, the module negotiates with the server the characters that the server will accept. Again the module may conclude that its needs are met to a sufficient degree or that it cannot cooperate with the server.

If the module concludes that it cannot function under the conditions that the server has listed, it will tell this to the server. The server then unloads the module. Otherwise the module asks the server to start sending input as it arrives.

In addition to input primitives and text output capabilities a module may require graphics output capabilities. These are queried and requested through a procedure similar to acquiring input.

The text that the module produces is given to the server one character at a time. The server forwards these characters to the operating system in the form of suitable events that produce text in the user interface components.

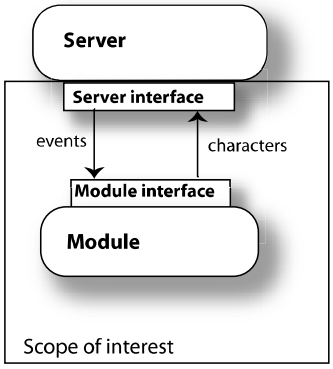


Figure 1. The two main components of the architecture.

2.2 Peripheral Components

For the server to be able to behave as the modules expect, several other components are needed. These are illustrated in Figure 2. Starting from the top the functions of the first components, *input devices*, *operating system*, and *applications* are as usual. Everything below them belongs to the text input architecture implementation. The operating system offers an interface for accessing the input device data. The *input adapters* use this interface to get the input and translate it to the internal format of our architecture. This data is then passed to the *server* who dispatches it to the interested *module* if any can be found. The module processes the input, and if a character is produced, it is given to the server for delivery. The server delivers the character to the operating system through an *output adapter*. The output adapter translates the character to the message format of the operating system. The operating system then forwards these messages according to its own

rules to the *application* or window that has the keyboard focus.

The *configuration handler* component in the middle of Figure 2 waits for the user to insert a user key that contains the user preferences. It then reads the preferences and commands the server accordingly. Usually this amounts to unloading all modules and then loading the modules that the user prefers.

Our current implementations run the server, the key handler and the adapter layers as separate processes that communicate through TCP connections. Because the platform-specific parts are isolated from the core of the server, we can use the same server on different platforms. The two platforms (Microsoft Windows and Linux), that we have implemented the system for, both run on desktop computers that usually have adequate resources for running all these processes. A different approach that minimizes the number of processes may be better for mobile devices with limited resources to allocate for text input. Portability of the server is a nice feature, but by no means necessary for achieving our primary design goal of creating text entry modules that follow the user.

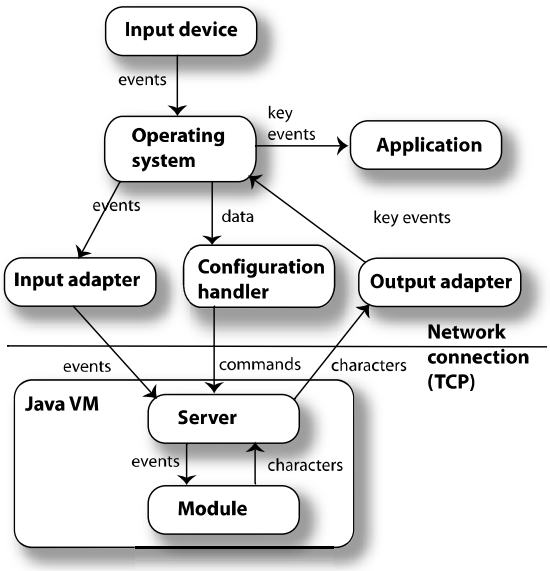


Figure 2. Overview of the components in the current implementations.

User Key

Before loading a text entry module the server must know which module to load. This information can be configured into the server or provided by the user. Because the user does not necessarily have text input capability before loading a module, there should either be a default module available or a way to enter the user preferences without writing. We use generic CompactFlash cards on a USB reader. The Subscriber Identity Module (SIM card) would be a natural choice in GSM phones. Even floppy disks or CD-ROMs could be used, when applicable. Naturally, a consensus on the memory device is necessary or otherwise the user has to carry all of these.

The medium that delivers the user preferences may contain all the needed information including the text entry modules, or it may contain network addresses where the modules can be found.

Module selection

Regardless of the way that the location of the preferred modules is delivered to the server, it will proceed to load the most preferred one and initiate the module setup sequence described above. If the module is not satisfied, it is unloaded and the next module on the user's list is loaded. This continues until a module is satisfied or until there are no more modules in the user preferences. A consequence of this module selection algorithm is that the user should choose the modules in his or her list so that they cover all the input/output conditions that he or she wishes to use.

The server can be configured with default modules and used without per-user module selection or with text entry method selection through explicit interaction with the user just like with the systems and devices that are currently available. The default module configuration is implemented in our server. Implementing interactive module selection involves writing a new configuration handler.

3 Features of the Architecture

To clarify the kind of situation in which our architecture is useful we describe a scenario where a user owns several computing devices and enters text to all of them. In the following we assume that our text input architecture is available on all computing devices.

3.1 Use Case: User with Several Devices

Our user has a mobile phone, a PDA, a digital TV set with a web browser, a game console, and a desktop computer. The user knows how to touch-type on a QWERTY keyboard, but his handwriting seems problematic to most handwriting recognizers. This is why he has learned how to use Quikwriting (Perlin, 1998) for entering text into his PDA. Luckily Quikwriting can be operated with a 9-key keypad and a joystick as well. The setup that requires the least amount of learning new text entry methods is to

Touch-type on the desktop computer and use Quikwriting on the other devices. For the mobile phone and the PDA doing this is straightforward. The setup procedure consists of copying the Quikwriting module onto the device and configuring the system properties to set it as the preferred text entry method.

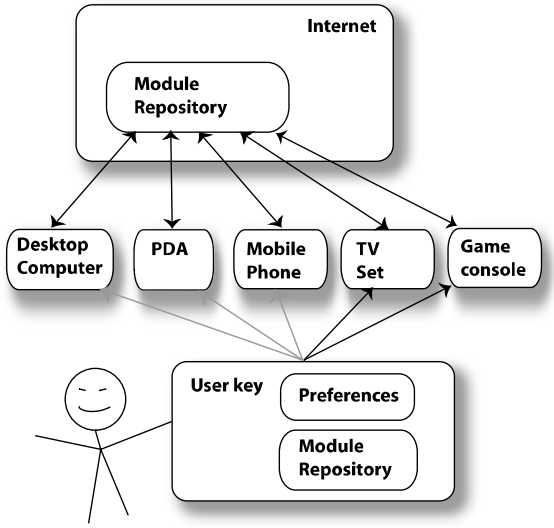


Figure 3. User key, devices, and Internet-based module repository.

The setup for the TV and the game console requires some more work because other members of the family use them too. These devices are configured to use per-user module selection with a small memory card as the user key. The overall situation is depicted in Figure 3. The user key could be used with all devices as shown, but using it makes no sense if a device has only one user or all users of the device use only one text input system as is the case with the mobile phone, PDA, and desktop computer.

Overall, instead of needing to bother with five different text entry methods, our user needs to know only two. This is a nice achievement, but is it worth the trouble of standardizing the architecture and implementing it on every computing device? We believe so, especially when we take into consideration the other good properties of the architecture. Next we will describe these properties in more detail.

3.2 Support for different input devices

A module can be written to support different input devices. Currently text input systems such as Thumbscript (2003), Quikwriting (Perlin, 1998), MDITIM (Isokoski and Raisamo, 2000), and MessageEase (2003) that can be operated with many input devices remain a curiosity. This is because text entry systems are built specifically for a certain device and a device usually has only one input device for text entry. Currently it makes no sense to support many input devices. However, in an environment where all text entry methods are available on all devices, users may prefer to learn one system well and use it with different input devices rather than spread their effort to several text entry methods. Accordingly, modules that support many devices are valuable. Our architecture makes writing such modules easier.

3.3 Multi-language support

Languages that use different characters are a problem in text entry. Supporting all possible characters at the same time is not a good solution with any text entry method, but the problem is especially bad in the case of handwriting recognizers. Building a recognizer that can reliably recognize all writing systems at the same time is most likely impossible. In our architecture different languages would most naturally be written using different modules. Only one of these is loaded at a time according to the user's preference.

3.4 Emerging text entry methods

It is difficult for a new text entry system to enter the market. The inventor must convince one or more of the device or operating system manufacturers to include the system in their products or the users will never have a chance to try it. This is mostly a good thing. As long as the manufacturers make decisions to protect the users from bad inventions and support the useful ones, everything is OK.

However, the current practice seems unnecessarily stiff. Early adopters of new technology face technical difficulties in trying out new text entry methods not supported by the device manufacturers. Because different devices often require different implementations or at least different build of the executables, most of the consumers do not want to see the trouble of figuring out if they can find an implementation that will work on their device. Our system allows the use of any text entry method on a particular device as long as the server that runs the modules is implemented for that device.

3.5 Personalized text entry methods

Aside from language preference, there are other reasons for loading a specific module for a specific user. For example handwriting recognizers can benefit from user-specific training. Presumably one of the reasons for the limited success of trainable recognizers is the difficulty that arises once the user upgrades the device. So far there has been no guarantee that the trained recognizer can be transferred to the new device. Also, if the user owns

multiple devices, he or she has to train them all separately or explicitly distribute the training database. In our system the training database resides with the recognizer module in the Internet or on the user key and is downloaded when needed. Personalized databases of disambiguation systems for mobile phones, such as those of T9 (Tegic, 2003), and word completion databases can be handled in the same way.

3.6 Lower overall implementation effort

Currently a prospective text entry method producer has to implement the logic of the text entry method and the interfacing to each of the operating system and hardware platforms that the system needs to run on. In our architecture all text entry methods use the same server that implements the interface to the device drivers and to the operating system for delivering the written text. Thus, the text entry method implementation needs to interface with only one system. Furthermore, the logic of the text entry method needs to be compiled only once. The implementation effort for the text entry method producer is reduced significantly. However, somebody needs to implement the server for each device. Even so, the overall implementation effort is smaller with our system than without it if the number of text entry methods is large. This is the case at least in the global perspective.

3.7 Coexisting with other systems

Even when our system is running, the operating system can choose not to send input to it. This effectively stops the text entry method from interfering with the rest of the user interface. Thus, for example on a mobile phone, the keypad can be used for text input only occasionally and for other purposes such as gaming, keying in phone numbers at other times. Similarly, users can be offered the choice of using the native text entry systems or our system just like they can switch between soft keyboarding and handwriting in the current PDAs.

4 Related work

User interfaces that support some degree of device independence have been built in operating systems since the first interactive systems were introduced. The mechanisms for text input exist in several levels of sophistication, ranging from simple key mapping to architectures that are distributed across several networked computers. Below we discuss some of these systems and their relationship to our architecture.

4.1 Platform-Specific Architectures

Some writing systems require user input beyond simply pressing a key to produce a character. For example, after pressing a sequence of keys to input the pronunciation of a character or a word some text input systems for Chinese require the user to select the intended written representation of the character from a list of matching characters (Sun, 2003; Wang et al, 2001).

As there are several languages that require this kind of interaction and there are several ways to implement the interaction, the need to separate these issues into a replaceable module in the operating system arises naturally. Accordingly, systems that are intended for world wide use are usually built to include a framework with a component that acts as an intermediary between the user interface components that receive the text input and the language-specific text input modules. An application that uses the standard text components offered by the operating system does not need to worry about the text input technology and the text input modules do not need to interact with applications directly.

In the following we summarize key points of some of these systems regarding our objective of making the text input method follow the user.

Java 2 Input Method Framework

The Java 2 platform includes an Input Method Framework, J2IMF, to enable text input components to receive their input from different text input methods (Sun, 2003). The kinds of methods that are mentioned in the documentation include language-specific input methods, methods for keyboards with less than the normal amount of keys, speech recognition, and handwriting recognition.

The text input methods in Java 2 are installed as extensions to the runtime environment. The intention in Java 2 is not to change input methods in runtime depending on who happens to be using the device. However, there is nothing to stop an input method developer from writing a Java 2 input method that can itself load code in runtime and execute it to produce different text input behavior. A system like this would in essence amount to implementing our server component for the Java 2 platform.

We claim that J2IMF is not an ideal solution. Still we have implemented our architecture in Java. The difference between the systems is that the Java Input Method framework works directly with the Java based GUI components in the applications. Our implementation interfaces with the operating system to support non-Java applications. In addition our approach requires only a small subset of the standard Java libraries making it more suitable for resource-

constrained devices. Generally speaking, our module implementation language does not need to be Java. What our architecture needs is platform independent binaries (i.e. they are run in a virtual machine) and that only one such language is used. Java seems a reasonable choice, as it is available in most devices.

Microsoft Text Services Framework

The current text input solution in Microsoft Windows is the Text services framework, TSF (Microsoft, 2003). The parts of the framework are connected using the Component Object Model (COM) framework. In addition to providing access to input devices, TSF provides access to the text buffers within applications enabling context-sensitive interpretation of input. The binaries that implement TSF-compliant text services are in general compiled separately for each hardware platform. However, just like in the Java 2 platform it is possible to provide wide platform independence within TSF by implementing a text service that loads platform independent modules that implement the actual functionality. This is just not done at the present.

IIIMF

The Internet/Intranet Input Method Framework (IIIMF) has been described with many of the same arguments that we have for our architecture. The goal is to build a system that is easy to integrate with existing text input architectures and utilize the services that they can offer over the network (Hiura, 1999).

IIIMF has two major parts separated by a network connection. The IIIM Server is platform independent software that offers services to IIIM clients. Platform independence in this case means independence of the platform on which the text input happens. The server is, of course, dependent on the platform that it runs on. A client is run on the computer that the user is using whereas the servers may reside on different computers. Parts of the system can be loaded over the network at runtime. This includes the code that does local preprocessing in the client computer before sending the input data to the IIIM server, and the code that displays feedback to the user. The servers run language engines that provide the translation from keystrokes to text.

IIIM Client Frameworks have been implemented for several platforms such as Java 2, X Windows, Emacs, and Microsoft Windows. Although one server can run the language engines for any of these, the code that is sent for the client to execute must be able to run in that system. All text input systems do not require code to run on the client system, but those that do are not independent of the client system.

4.2 Theoretical Frameworks

If viewed through the well known Seeheim (Green, 1985) and Model-View-Controller (Krasner and Pope, 1988) architectures, the text input systems discussed above including ours reside outside the application. TSF and J2IMF utilize the same user interface widget instances that the application uses. Overall, the systems do recognize the need to separate text input from the application thereby following the ideas of the Seeheim and MVC models.

So far mobile computing devices have shown some evidence of text entry adaptability. For example in mobile phones, text input method can be chosen from a set of usually two alternatives. In addition to the traditional multi-tap a more sophisticated disambiguation-based method like T9 (Tegic, 2003) or iTap (Motorola, 2003) is usually available. The systems themselves can further adapt to the vocabulary of the user, but overall the choice of text input method is limited to the implementations that are pre-installed in the device.

Although we use partly the same rhetoric that has been used for supporting adaptive user interfaces and multimodality in general, we are not proposing a general solution. Our work considers only text entry techniques. This distinction is important because supporting adaptivity, adaptability, and plasticity (Thevenin and Coutaz, 1999) with or without multimodality in all aspects of user interfaces is a huge undertaking with slim chance of success. Personalization, through these means is much easier to achieve within a limited domain. Because the domain of text entry involves expensively acquired skills personalization can offer clear payoffs in saved time and effort.

5 Discussion and Future Work

While our experience with the architecture has shown that it is realizable, the implementation is to be taken as a proof of concept rather than as a finished product. Here we discuss some important aspects that need to be improved if the system is to be used outside research laboratories.

5.1 Security

Java provides a framework for running code downloaded from the network relatively safely. There are aspects of the proposed architecture that make it necessary to use all the security features available.

For example, a simple but effective malicious use of the system would be to insert a module that saves all that the user writes and sends it to the interested parties over the Internet. Another way to cause

security problems would be to write a module that gives commands to the system as if they were coming from the user.

To avoid these harmful scenarios, the modules must be authenticated. The user can authenticate the fact that he or she really wants to use this specific module. An outside authority can authenticate that this module has been approved for general use. Which combination of authentications is required depends on the system administrator. Manufacturers and network operators might require both forms of authentication to be used whereas an individual user might only require his or her own authentication for modules used in a personal computer.

5.2 Standardization

Our text entry architecture is not the only system that requires widespread agreement on interfaces. Initiatives like the Open Mobile Alliance, OMA (2003), aim to provide common software architecture for mobile computing devices. Text input and other user interface issues will undoubtedly emerge in discussions within these cooperation organizations. We are not affiliated with any of these organizations. Our goal is to independently experiment with text input architectures to provide evidence for or against adopting them for general use.

5.3 Unsupported Text Entry Methods

Naturally our minimal programming interface excludes some types of text input methods. There is presently no mechanism for informing a module about the input context such as the text surrounding the cursor or more general context such as whether numeric input or text is expected. Thus, for example handwriting recognizers cannot utilize this information to increase recognition accuracy. The strength of the text input frameworks built into user interface frameworks is that they can get this kind of information. In addition, the closely coupled systems like TSF and J2IMF can offer things like in-place editing with a stylus.

Support for context information could be added. It would make the interface slightly more complicated, but that is not the main reason for not having it already. The justification for keeping it out so far has been our doubts on its usefulness. Text entry methods that fully utilize the context are rare in the mobile devices.

The list of supported input devices indicates more unsupported text entry methods. We have support for pointers and keys but for example a microphone is not supported. Assuming a recognition server based approach such as in the MiPad system (Huang et al, 2000), useful but compact speech based modules for text entry could be constructed. Support for microphones and other input devices can be added. All server implementations do not need to support all devices. The important thing is that those that do support a device all do it in the same way.

5.4 Practical Lessons Learned

So far we have implemented mostly incomplete modules to test the server implementations. The more complete ones include an extension of QuikWriting (Perlin, 1998), shown in action in Figure 4. We adapted Quikwriting to be used with a joystick and keyboard in addition to the original stylus. Another working module implementation is MDITIM (Isokoski and Raisamo, 2000). Here we discuss the practical issues that have risen in this work.

It is desirable to have as compact modules as possible. The size of currently implemented modules varies between 10 and 40 kilobytes. Modules that are to work with a wide variety of input devices need code that examines the properties of the available input and output, and then adapts to the available resources. Selecting the optimal input device configuration without human intervention is a difficult problem. We do not claim to have found a perfect solution. The current approach is to divide the input into basic input devices such as absolute and relative pointers, and key matrices. One physical device may contain many of these. For example mouse has a key matrix and a relative pointer. In the relatively simple cases that we have encountered so far our modules have been able to select the correct input devices. However, very difficult situations are easy to construct especially with odd key arrangements.

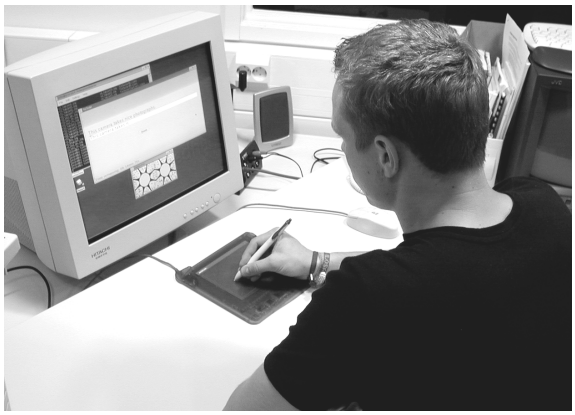


Figure 4. A participant at work in an experiment that utilized the architecture.

Implementing the input and output adapters for Linux and Windows is straightforward because both systems have good support for this kind of work. Currently we have adapters for mice, tablets, game

controllers and keyboards for Linux and for game controllers and tablets for Windows. In Linux these use the evdev interface to USB human interface devices and in Windows we use DirectInput.

Implementing the modules is not simple. Although a module may consist of as few as 500 to 1000 lines of Java code, the code needs to be very adaptable and efficient. Efforts to save in the module size, runtime memory footprint, and CPU load may easily lead to bad programming practices resulting in code that is difficult to comprehend and maintain. Despite these hardships we believe that implementing the module once in a hard way is less expensive than implementing it dozens of times to serve all devices with a platform specific implementation.

6 Conclusions

We presented the rationale and implementation of text entry architecture for mobile devices and users. The architecture has many benefits. It allows the users to freely choose the text entry methods they want to use and to retain the usefulness of their text input skills as computing devices change. Another important benefit of the architecture is that software vendors only need to support one platform – namely the implementation of the described architecture – rather than all of the current operating system platforms.

In its current state the system serves as a platform for doing text entry experiments using multiple text entry methods. We use it because it saves effort in implementing the text entry methods that we need especially when testing text entry methods that can be used with more than one input device. If adapted to wider use, the system could offer the same benefits to a wider audience.

Acknowledgments

This work was funded by Tampere Graduate School in Information Science and Engineering and by the Academy of Finland (project 50724, grant 73987).

References

Green, M. (1985), Report on Dialog Specification Tools. User Interface Management Systems, Springer-Verlag, 9-20.

Hiura, H. (1999), Internet/Intranet Input Method Architecture, Sun Microsystems Inc.. available at: http://www.li18nux.org/subgroups/im/IIIMF/whitepaper/whitepaper.html

Huang, X., Acero, A., Chelba, C., Deng, L., Duchene, D., Goodman, J., Hon, H., Jacoby, D., Jiang, L., Loynd, R., Mahajan, M., Mau, P., Meredith, S., Mughal, S., Neto, S., Plumpe, M., Wang, K., & Wang, Y. (2000), MiPad: A Next Generation PDA Prototype, Proceedings of ICSLP 2000. Beijing, China.

Isokoski P., & Raisamo R. (2000), Device independent text input: A rationale and an example. Proceedings of the Working Conference on Advanced Visual Interfaces AVI 2000, ACM Press, 76 - 83.

Sun (2003), Java 2 Input Methods Framework, http://java.sun.com/j2se/1.4/docs/guide/imf/.

Krasner G. E. & Pope S. T. (1988), A cookbook for using model-view controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, 1 (3), 26-49.

MacKenzie, I. S., & Soukoreff, R. W. (2002), Text entry for mobile computing: Models and methods, theory and practice. Human-Computer Interaction, 17, 147-198.

Message Ease (2003), http://www.messagease.com/

Microsoft (2003), Text Services Framework, http://msdn.microsoft.com/library/en-us/tsf/tsf/text_services_framework.asp

Motorola (2003), iTap, Intelligent keypad text entry technology. http://www.motorola.com/lexicus/html/itap.html.

OMA (2003), Open Mobile Alliance, http://www.openmobilealliance.org.

Perlin, K. (1998), Quikwriting: continuous stylus-based text entry. Proceedings of the ACM UIST '98 Symposium on User Interface Software and Technology, ACM Press, 215-216.

Tegic (2003), Tegic Communications, Inc., T9 text input. http://www.tegic.com.

Thevenin, D., & Coutaz, J., (1999), Plasticity of User Interfaces: Framework and Research Agenda, Proceedings of Interact '99, IOS Press, 110-117.

Thumbscript (2003), http://www.thumbscript.com

Wang J., Zhai S., & Su H. (2001), Chinese input with keyboard and eye-tracking – an anatomical study. In Proceedings of Human Factors in Computing Systems, CHI 2001, ACM Press, 349 - 356.

The Software Quality Star:
A conceptual model for the software quality curriculum

Ronan Fitzpatrick

School of Computing,
Dublin Institute of Technology, Kevin Street, Dublin 8, Ireland.
Tel: +353 (1) 4024835, Fax: +353 (1) 4024985
Email: ronan.fitzpatrick@comp.dit.ie

Abstract

Usability is a significant factor in the creation of quality software products which is an important focus of the software engineering syllabus. Usability is also a significant factor in the study of Human-Computer Interaction (HCI). Despite the connection, software quality and the rigours of the software engineering discipline are not used in the teaching of HCI and visa versa. Consequently, students undertaking software engineering courses are not fully exposed to usability issues and HCI students often miss out on the disciplines of software engineering. This paper presents a quality focused conceptual model – the Software Quality Star - which can be used in both syllabi.

1 Introduction

Software quality is a well researched and understood discipline in software engineering. Quality is an aspiration of everyone. External quality factors are accepted as those that impact end users. Fitzpatrick and Higgins (1998) argue that the combined set of external quality factors constitute usability which is a significant consideration for HCI professionals. However, while quality is an objective of HCI professionals, the HCI curriculum is not driven by a quality perspective. At the same time software engineering professionals want to produce quality products which will be used by end users, but their syllabus is not driven by a focus on usability.

The aim of this paper is to introduce to IT educators and practitioners a conceptual model for software quality, a model of quality perspectives, which is motivated by the International Standard ISO/IEC 12207 (1995). This standard addresses life cycle processes and the model considers quality throughout those processes. User considerations are core to those processes, so, the model incorporates elements, which impact HCI. The model was developed specifically for undergraduate programmes, where it is being successfully used.

Section 2 introduces the Software Quality Star and explains the originating philosophy underpinning the model. Section 3 provides an overview of the principal topics addressed by the various components of the model, illustrating how quality is the principal motivator. Section 4 clarifies how the model supports the transfer of software engineering theory to the discipline of Human-Computer Interaction. Section 5 outlines current usage in an academic context and Section 6 offers some concluding comments.

2 The Software Quality Star

This section presents the conceptual model of the Software Quality Star and clarifies the philosophy underpinning its origin. The section also comments on the comprehensive paper that supports the understanding of the model.

2.1 The conceptual model

The software quality star is a conceptual model for presenting the different perspectives of software quality as seen by the different software product stakeholders and is based on the Acquirer and Supplier as defined in ISO/IEC 12207 (1995). The model is illustrated in Figure 1.

There are three significant elements in the star – the Producer (Supplier), the Procurer (Acquirer) and the Product. In the model the Procurer enters into a contract with the Producer to create a software product. This contract will clearly specify the

quality characteristics of the product. The Procurer's perspective of the producer organisation is that they will use best project management practice and engage in first-rate processes to create a quality product. The Procurer's perspective of the product is that it will be acceptable by the user community and that it can be serviced and maintained by their IS professionals.

The model considers the Procurer to be the lead party in any contractual arrangement as it is the Procurer's users and technical support professionals

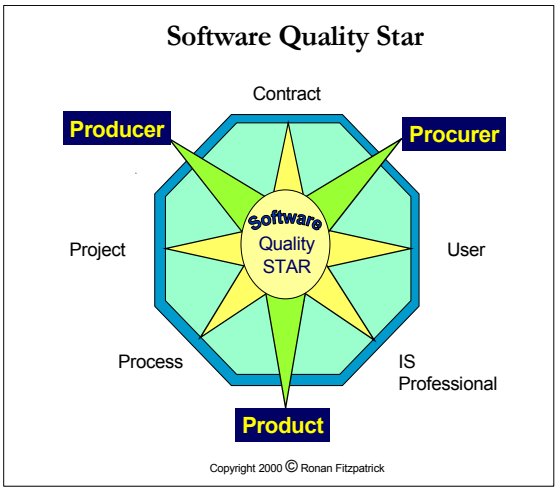


Figure 1 – *Software Quality Star*

who will dictate the success or failure of the software product. It is also the Procurer who will dictate the profile and maturity of the selected Producer organisation.

The model also accommodates the Producer's perspective of software quality and focuses on the maturity of the Producer organisation as software developers and the development processes that they used to create quality software products.

2.2 Motivating philosophy

The motivating philosophy for the model is software quality and the principal aim is to show that quality permeates all of the stakeholder's perspectives. The philosophy is not limited to the belief that quality is "*conformance to specification*" or "*fitness for purpose.*" Instead, the model relies on the philosophy of Kaoru Ishikawa, the founding father of Japanese thinking on quality who writes "*that broadly interpreted, quality means quality of work, quality of service, quality of information, quality of process, quality of division, quality of people including workers, engineers, managers and executives, quality of system, quality of company, quality of objects*" (Ishikawa, 1985). Section 3 illustrates this quality focus in more detail.

2.3 Supporting papers

All of the perspectives outlined in the model are comprehensively explained from a quality perspective in a supporting paper. These explanations rely on acknowledged and seminal work in the area and are fully illustrated. The extended *Software Quality – Strategic Driver Model* (Fitzpatrick, 2001) is also used as supporting reading.

A summary of the components and topics addressed are set out in the next Section.

3 Components and topics

There are three main perspectives in the model, the Producer's perspective, the Procurer's perspective and the product perspective. Each is explained now in turn.

3.1 Producer's perspective

The Producer's perspective is driven by the desire to be a quality organisation employing first-rate staff who engage in first-rate processes using first-rate tools and techniques to create quality software products. In keeping with the House of Quality model (Hauser and Clausing, 1988), the Producer will be driven by an enlightened philosophy and leadership. This philosophy is underpinned by the belief that to achieve quality there must be political (organisational) stability and that there is a need for continuing education among the actors involved. The organisational ability will support organisational desire to create quality products and the education will support the ability to specify quality requirements. This is especially important for aligning the Acquirer's corporate processes and their software requirements. The need to address software development strategies like developing re-usable code and software portability are significant issue for the Producer organisation. These topics considerably impact the cost of creating software products and might not be of interest to the Acquirer. The accepted approach to successfully creating these

first-rate software products is to engage in software project management and an expression often used to provide focus for the objective of project management is "*to complete the right project, on time and within budget*".

Engaging in a project is a software industries approach to creating a software product and ISO/IEC 12207 (1995) gives project management guidance in section 5.2 of the standard. This means that project management best practice is used in order to assure the successful delivery of the product and this best practice employs planning, organising, controlling and directing throughout the development life cycle. It is through this best practice that the Supplier demonstrates to the Acquirer their standing and competence as a Supplier organisation.

ISO 12207 (1995) places the onus on the Supplier *to* develop *and* document *project management plans* (section 5.2.4.5) and continues that *the Supplier shall* implement *and* execute *the project management plans* (section 5.2.5.1). So, there are two critical aspects to the Supplier's responsibility (develop and document AND implement and execute) and their level of competence in these is what the Acquirer should evaluate.

In this perspective the process for creating the software product is all-important. As part of the creation of software products, supplier organisations will engage in a set of processes. In the early days of software development these processes were established almost by trial and error and sets of standards evolved to suit developer understanding and practice. More recently, organisations like the International Organisation for Standardisation, the Software Engineering Institute and different developer companies have devised standards and models, which quantify full and comprehensive sets of processes. The philosophy behind this approach is that by addressing these comprehensive processes, supplier organisations will create quality software products. The International Organisation for Standardisation (ISO) and the International Electrotechnical Commission (IEC), have published ISO/IEC 12207 (1995) relating to software life cycle processes, the Software Engineering Institute has developed the Capability Maturity Model (Paulk *et al*., 1993b) and ISO/IEC are developing the SPICE standard (ISO/IEC TR 15504:1998). The discussion in this chapter will be confined to these three approaches but the reader should be aware that there are other solutions, especially in the commercial sector.

3.2 Procurer's perspective

In the context of software quality, the Procurer is obviously interested in knowing that the Producer is a first-rate organisation, which uses first-rate processes to create software products that incorporate all of the most appropriate quality factors. However, there are also strategic issues, which the Procurer must address. For example, the Procurer will be interested to know that there is alignment between the software product and the organisation's business processes. The Procurer will also be concerned to know that the software product will be usable, that it contains the highest technical excellence and that the organisation can afford the software and secure a return on the investment. There are also legal consideration relating to the quality of software which the Procurer must be satisfied are conformed to by the software. Competitive advantage or competitive survival is also a Procurer's concern. Typically, these are Strategic Quality Drivers which have been addressed by Fitzpatrick (2001)

For the purpose of this section the IS professionals being considered are the technical professionals of the Acquirer organisation who have line responsibility for IS. Their role begins with advise to management regarding the specification, selection and acquisition processes and would typically address technical excellence, user empowerment, corporate alignment and investment efficiency together with supplier organisation profile. They have responsibility through to the retirement of the software at the end of its operational life.

During the specification, selection and acquisition processes they will identify the quality characteristics required of the software product and will ensure that contract documents address these issues.

During the operational stage they are responsible for supporting the users or operators of the software and for servicing and maintaining the software during its operational life. As part of their supporting role these IS Professionals are concerned with all of the quality factors as outlined in section 3.4. For example, the installability of the software might be especially of interest to them, or, the

reliability of the system might seriously impact their workload. From the time that a new system is delivered, technical support relies heavily on user and technical manuals. So, from this IS professional's perspective an essential requirement of a quality software product is a full and comprehensive set of these.

As part of the servicing role they will be addressing such quality issues as reinstalling sections of software where modules become corrupted and will require systems that support this type of servicing activity.

And, as part of their maintenance role they are required to adapt software to suit changes such as government directives, changing rates like pay scales, tax rates and similar items. They will be required to correct any bugs or other errors that manifest themselves during the life of the product and they will be required to perfect the software especially by fine-tuning algorithms to improve their efficiency and to meet new user requirements. To assist them in these activities they will be especially interested to know that good designs, documentation and best programming practice has been used during the project phases when the software was being created.

In addition to supporting other users and maintaining systems, these professionals are often themselves the users of products like network operating systems and management tools, so, they too will be impacted by quality of use issues. While for some, the internal quality factors may be their primary interest they will also be concerned that the external quality factors fully support users.

3.3 Product perspective

From this perspective a software product is considered to be a quality product if it supports a set of quality factors or product characteristics. Software quality factors were first defined in the 1970's by researches like McCall *et al.,* (1977) and Boëhm, (1978). Their research was later complemented by standards like IEEE and ISO. More recently, Fitzpatrick and Higgins (1998) conducted a methodical analysis and synthesis of three strands - quality (as explained by McCall *et al.* and by Boëhm), statutory obligations, and human-computer interaction, which influence software quality. This established a comprehensive set of quality factors, and those factors that related to users, they called the attributes of a usable software product. More recently this set has been extended to show that, depending on the domain, additional quality factors will be involved. Figure 2 shows Fitzpatrick and Higgins original set of CORE QUALITY FACTORS together with the DOMAIN SPECIFIC QUALITY FACTORS appropriate to the World Wide Web.

Figure 2 is divided to categorise the quality factors into External quality factors (those that significantly impact end users), Internal quality factors (those of a "technical" nature which are of specific interest to IS professionals, and a new category - Strategic quality factors – which are of special interest to strategic managers and IS Procurers. This last category is new in that

	EXTERNAL QUALITY FACTORS	INTERNAL QUALITY FACTORS	STRATEGIC QUALITY FACTORS
CORE QUALITY FACTORS	• Suitability • Installability • Functionality • Adaptability • Ease-of-use • Learnability • Interoperability • Reliability • Safety • Security • Correctness • Efficiency	• Maintainability • Testability • Flexibility • Reusability • Portability	
DOMAIN-SPECIFIC QUALITY FACTORS	• Visibility • Intelligibility • Credibility • Engagibility		• Differentiation

Figure 2 – *Quality factors*

researchers have not addressed it in the past and is now the focus of some researchers. The grid also shows how it is necessary to consider special domains (in this case the World Wide Web) and illustrates the additional quality factors required for this domain. For example, if the domain is mobile computing or virtual computing then new, as yet unquantified quality factors might be needed for those domains. The set of Core quality factors was identified in relation to traditional data processing applications. So, it is also necessary to interpret the Core quality factors for each new domain. For example, in a traditional data processing environment, maintenance was concerned with corrective, adaptive and perfective maintenance. Maintenance of a Web site is a very different occupation with content in particular often needing to be updated daily.

4 Bridging the SE-HCI gap

There are two significant contributions that the Software Quality Star makes towards bridging the gap between software engineering and Human-Computer Interaction.

First, it is quite common that students of the many disciplines that impact information systems don't fully appreciate how the different study modules fit together. It is not until they experience a project in action that their understanding and appreciation is completed. Using quality as a common strand and an international standard of life cycle processes the Software Quality Star becomes a tool which helps to overcome this difficulty.

Second, the Software Quality Star creates an awareness and understanding of the contribution that can be made by other disciplines. Consequently, it becomes appropriate for informed students of one discipline to question if practice in the other discipline might be appropriate for use in theirs. A typical questioning might refer to quality metrics which are well understood in the software engineering discipline, but they are not researched or studied to the same extent in the HCI domain. This is significant because HCI for E-Commerce is becoming a major consideration for Web site owners who are looking for quality Web sites to secure competitive advantage.

5 Conclusion

The Software Quality Star was originally developed for teaching software quality topics It has been successfully used at undergraduate level with computer science and business studies students in their software engineering, project management and HCI modules. It can also be successfully used as a framework for teaching other modules which address legal and contracting issues, and similar IT management topics. At postgraduate level during Information Technology and Strategic Management seminars an enhanced version of the model, *Software Quality – Strategic Driver Model* (*SQ_SDM*), is used. This extended version of the Software Quality Star focuses further on the competitive advantage perspectives of the Acquirer and the Supplier. Both models could be easily interpreted for new domains like the WWW and new and evolving technologies like mobile solutions.

References

Boëhm, B. (1978) *Characteristics of software quality*, Vol 1 of TRW series on software technology, North-Holland, Amsterdam, Netherlands

Fitzpatrick, R. and Higgins, C. (1998). Usable software and its attributes: A synthesis of software quality, European Community law and human-computer interaction, In: *People and Computers XIII. Proceedings of HCI'98 Conference*, Springer, London, UK

Fitzpatrick, R. (2000) Additional Quality Factors for the World Wide Web, *Proceedings of the Second World Congress for Software Quality*, Yokohama, Japan, Union of Japanese Scientists and Engineers (JUSE), Tokyo, Japan

Fitzpatrick, R. (2001) Strategic Drivers of Software Quality: Beyond external and internal software quality, *Second Asia-Pacific Conference on Quality Software, Proceedings of APAQS 2001*, Hong Kong; IEEE Computer Society Press, California, USA.

Ishikawa, Kaoru (1985) What is Total quality control? : The Japanese way. Prentice Hall, Englewood Cliffs, London, UK, p44/5

ISO/IEC 12207 (1995) *International Standard. Information technology - Software life cycle processes*, International Organisation for Standardisation, Genève, Switzerland

McCall, J., Richards, P. and Walters, G (1977) *Factors in software quality*, Vols I-III, Rome Aid Defence Centre, Italy

HCI Designers and Engineers: It is possible to work together?

Jorge Belenguer, Jose Parra, Ismael Torres, & Pedro J. Molina.

CARE Technologies S.A., Pda. Madrigueres 44, 03700, Denia, SPAIN

{jbelenguer|jparra|itorres|pjmolina}@care-t.com

Abstract: This position paper states the point of view of the authors about the problems and some possible solutions about integrating and making effective the work of Software Engineers and HCI professionals. These two groups of people use quite different jargon and tools to build User Interfaces. Try to bridging the gap and improve the communication between these two groups is essential to obtain better products in terms of efficiency (SE) and usability (HCI).

Keywords: Software Engineering, HCI, heterogeneous development teams.

1 Introduction

It is quite clear that the development of good systems requires the collaboration of different professionals: software engineers and programmers (SE) from one side and HCI professionals (including interaction designers, usability experts, graphic designers, user experience experts, etc.) from the other side are obliged to reach and agreement to satisfy the user.

Different groups of people use different jargon, methods, and techniques to develop UIs. The main problems arise from such differences and produces communication problems among the staff.

2 Background of the authors

All the authors of this position paper work for a Research & Development company interested in the automatic production of business applications (http://www.care-t.com). Therefore, we should warn in advance that our approach could be understood in some way a bit Software Engineering biased.

However, we want to describe our ideas in this workshop to stimulate discussion and obtain feedback from the research community, especially from approaches more HCI oriented.

2.1 Software Engineering approach

The approach we have taken is to use Conceptual Modelling to produce abstract and precise specification describing the system to be built. Such a system is specified in terms of structure, behaviour, functionality and abstract user interface. Design details and platform selection are explicitly not collected in this phase to focus on *what it is needed* instead of *how it is implemented*. Abstract specifications are produced using CASE tools with validation support and stored in XML files.

2.2 Code Generation Driven

After the specification is completed, it is used as input for specialized code generators (transformation engines) to produce the persistence, business and user interface layers of the application for a given architecture. For example: MS SQL Server, VB/MTS & VB client code or Oracle 8i, EJB components & JSP web clients, to cite two classical configurations.

On the client layer, we have worked with the following technologies: Visual Basic 6.0, Java/Swing (desktop environments), JSP, ColdFusion, ASP (web environments) & eVB (pervasive computing).

In some of them, we have developed transformation engines (code generators), in others: applications and prototypes.

Depending on the target platform different usability problems have arisen.

2.3 Desktop UI experiences

From our point of view, desktop UIs are the powerful and simplest ones. **Powerful** in the sense that they provide the richest toolkits to build user interfaces. **Simplest** in the sense that are well-known by developers, well-supported by tools and IDEs to create and manipulate such UIs and well documented (there are good books teaching the rationale behind the design of good desktop user interfaces).

Guidelines and style guides are well established in desktop environments. Each platform establishes a style guide (a look & feel) to be followed, making easy some design choices and providing standardisation of UI features accordingly to the style guide.

For this reason, customers easily agree with the general look & feel of the application if it is compliant with the environment and other similar applications.

Nevertheless, sometimes the look & feel of applications generated by code generators does not satisfy the customers.

For example: the form shown in Figure 1 allows to retrieve objects from a database and interact with them: executing services within the objects, navigating to other related objects or limiting the set of displayed objects using filters. The widget used to represent this behaviour is considered like a 'black box' and always has the same appearance (called a *user control* using VB terminology).

Figure 1. Example of Desktop window.

The customer (or the UI designer) could not agree with this appearance. Maybe, he would prefer to see the vertical toolbar on the left instead of displaying it on the right. Or maybe, he would prefer to view the horizontal toolbar as a list control.

2.4 Web UI experiences

However, Web applications are a bit trickier. There is no single language to develop (you will need at least HTML, CSS, Javascript and server side scripting). Browsers are not compatible with one another, forcing developers to program for and test it in different browsers. UI widgets are of course less rich that Desktop ones. Latency delays and net breakdowns influence your application. Stateless programming is harder that using state and simulating the state using cookies or session management has a non universal solution.

Furthermore, there is no style guide for the web. On the contrary, applications must follow the *look & feel* used in the customer corporate site.

The automated user interfaces are produced with the aim that such interfaces could be the final product and to minimize the manual code. But nowadays, the scope of a web application (number of potential users) is wider than some years before and the profiles of the Internet users are not always well defined. For these reasons, it is more difficult to satisfy the necessities of possible users.

Figure 2. Example for Web environment.

Figure 2 shows the same example as previously shown in Figure 1 but implemented in a web environment.

2.5 PDA UI experiences

The recent emergence of Pocket PC and palm-sized devices compelled us to consider the potential benefits of these mobile, wirelessly connected handheld devices. The main problem with pervasive devices is their constraints in terms of screen size, keyboard, toolkits, memory, etc.

Conventional GUI techniques appear to be ill-suited for some of the kinds of interactive platforms now starting to emerge, with ubiquitous computing devices having tiny and large displays, and recognition-based user interfaces using speech and gestures.

Personal Digital Assistants (PDAs), personal organizers, digital cell phones, Tablet PCs, WebTVs, wall-size displays, large plasma panels and other devices are becoming more and more popular. Interfaces on these very large and very small displays cannot typically use the standard desktop model, and people will not necessarily expect these devices to act like "regular" computers.

The implication of these changes is that we can expect a dramatic increase in the diversity of both the types of computing devices in use, and the task contexts in which they operate. This in turn implies that we are poised for a major change in user interfaces, and with it dramatic new needs for tools to build those interfaces.

Another problem is there is no standard for UI development across different vendors for pervasive devices. Some devices have rich UI widgets meanwhile others have poor ones. Some devices are Internet-capable (have a WAP-Browser, a HDML-browser, a HTML-browser or other kind of browser) meanwhile others are not. Some devices have graphical UIs meanwhile others have textual UIs (or even speech UIs). In fact, some PDA UIs can be seen as a subset of Desktop UI or Web UI (or both), but having constrained screen size and screen resolution.

The code generated for mobile devices applications has the same kind of problems as mentioned in section 2.3. In addition to these problems, we have to consider the reduced screen size of this kind of devices.

Figure 3. Example for PocketPC.

For example: the form shown in Figure 3 provides equivalent functionality to that shown before in Figure 1 & Figure 2. Some features are implemented in different forms due to the reduced screen size.

3 The problems found

During these years working on the described context, some problems in this area were found.

3.1 Automatic UIs are good but are not the best ones

Code generation can save a lot of resources and prevent human mistakes. However, full automation can be undesired in the following context:

1. You need to change the look and feel to adapt to a specific platform or application.
2. The computer will not be as creative as a good HCI specialist or UI designer. It will follow a deterministic algorithm to take decisions and place widgets following a fixed layout algorithm.
3. You found a problem that is out of the scope of your code generator. You are neither able to specify nor generate it.

3.2 Recognize different capabilities

Different people have different capabilities. In the same way SE and HCI guys usually think in a very different way. They have different main concerns and solve problems using different approaches. Engineers try to focus on functionality, efficiency, scalability, etc. whereas HCI professionals are more concerned about usability, suitability or funology!

3.3 Different Languages, Similar concepts

We have found that in early stages of the development, the conception of the UI follows a similar approach for each group. Using sketches and wire-frames helps to draw an initial mock-up without being lost in the details. Refining the sketches, tasks, functionality, navigation and contents will appear soon or later.

Is quite interesting to observe how different UI professionals use specialized jargon used by a small group of colleagues.

3.4 Complexity of functionality/design

Small systems are easy to deal with. However, in industrial ones, scalability problems appear on the scene quickly.

HCI and graphic designers tends to treat the functionality of an application as a black box. They do not know how it works and they do not care: a programmer will do, in this case, the hard job.

On the other hand, the same occurs with aesthetics and graphic design: software engineers and programmers will do not touch the graphic design. Otherwise, they could break it down.

For big-sized systems the necessity of splitting functionality and UI design is crucial in order to have

both kind of people working together instead of fighting against each other.

3.5 The not so standard web look & feel

As anticipated previously, there is no standard web look & feel, no consensus of how it should look like. And for marketing reasons, this will probably never be possible: every company on the Internet wants to offer differentiated services respect to the competitors.

Moreover, depending on the user profiles some design options could be good or bad options, for example depending on the expertise level of the users.

This makes harder the web design, in the way that it is necessary to adopt or design a new style guide for the web application to our new customer.

These problems points out the necessity to have code generated easily updateable to support such changes and maintenance. Transformation engines have to be design taking into account the necessity to easily change such look & feel.

4 Some solutions

In our daily work, we have found some useful techniques to deal with these problems. In this section we will describe some of them.

4.1 Separate of Concerns

Separate of Concern is an effective principle to split the responsibilities of different roles during the development. UI Designers and Software Engineers can concentrate in their respective tasks with controlled interference if an appropriate frontier can be clearly drawn.

Patterns like MVC (Goldberg, 1983) or architectures as PAC (Coutaz, 1987) or ARCH (SIGCHI, 1992) helps to implement such a principle for separating UI from functionality.

4.2 Patterns

User Interface Patterns have been revealed as a good resource to be used in UI development: Patterns can be considered as a *lingua franca* (Erickson, 2000) shared by the development team.

Design patterns (Tidwell, 1999) & (van Welie 2000) are quite useful to learn good solution to recurrent design problems.

More over, Conceptual UI Patters (Molina, 2002 & 2003) are also useful in early stages of the development allowing to include users in the requirement elicitation process.

Patterns languages used in a UI also provide homogeneity in such a UI: in the way that the same problem is always solved in the same way and is easily recognisable by the user. Recent workshops addressed this usage of patterns for UI Design (Mullet et al., 2002) & (Fincher et al., 2003).

4.3 Sketches and diagramming

Sketching the UI and making paper diagrams helps to understand the user requirements and to envision the UI.

Papers based techniques (pens, paper and coloured sticky notes) (Ruble, 1997) and (Constantine & Lookwood, 1999) are useful during the requirement elicitation phase to work directly with customers and stakeholders.

Wireframes techniques (van Welie, 2001) or UI computer supported sketching like in DENIM (Landay, 2001) have the advantage that eliminates all distracting design choices and concentrates in the negotiation of an abstract UIs to focus the discussion in what it is needed instead of how it could be implemented.

All these documentation artifacts are quite valuable for engineers and HCI professionals. They can work using these materials to identify and resolve pending ambiguities and problems.

4.4 Mock-ups and early prototyping

Prototypes and mock-ups are also quite helpful. Users can test the system in early stages before it is built. Users, HCI professionals, and engineers can take advantage of mock-ups to detect usability issues even after the final product is built.

4.5 Functionality driven development

What we call a *"functionality driven development"* usually starts trying to find all the functionally need and later on, organize such features in a suitable user interface.

In this approach, the functionality is ready before the UI. It is useful when designing well-known applications that should follow a strict style guide. Transformation engines can be used to produce a high percentage of the final code. If 80% of the generated UI scenarios are ready to use, the saved time obtained by the usage of code generation can be used to redesign or tweaking the UI of the other 20% pending scenarios.

The need to change the look and feel of the UI will hardly ever occur.

4.6 UI driven development

On the other hand, a more *"UI driven development"* starts identifying user tasks, scenarios and prototyping the user interface. In this process, the functionality is incrementally discovered by HCI specialist, interaction designers, etc.

The UI is built manually and it is ready before the functionality is implemented. Code generation is less suitable for these tailored UIs, however is still very valuable for the functionality of the application.

4.7 Templates

Templates can be an effective way to implement SoC (Separate of Concerns). Static templates (applied in design-time) like the used in Macromedia Dreamweaver (Macromedia, 2002) or dynamic templates (applied in run-time) as introduced in Velocity (Velocity, 2002) or Smarty (Smarty, 2002) helps to maintain contents, behaviour and presentation in separate files. Using templates, in this way, can be an effective way to maintain separately the design and layout of an UI from the functionality behind it.

4.8 Wizards

In *"UI driven development"* the UI designers do not need to know too much about the implementation of the functionality. They only need to invoke such behaviour in certain points during the user interaction.

In this scenario, Wizards tools driven by functional specification metadata can help to select the functionality needed for a scenario and provide the code in a particular language to invoke such functionality. Such a code can be inserted automatically in the UI design after the designer choice.

Doing it in this way, designers will only need to know what functionality they need and how the invokers work. Meanwhile, the implementation will be kept as "top secret" by SE specialists.

4.9 Customisation

Static applications are more rigid that configurable ones.

Despite the user profiles and differentiated user interfaces, each user can be seen as unique.

In this way, if we can also provide generic customisation mechanisms in an automatic way, we will allow users to personalize their application to their needs.

5 Conclusions

The problem stated by this workshop is not an easy pie, on the contrary is a real problem nowadays we must cope with. The possible solutions proposed in this position paper are probably not enough to solve the general problem in the area. However, in our domain our two cents seems to help in our daily work.

We hope these ideas promote discussions during the workshop and we are looking forward to getting feedback about that from the rest of the workshop attendees.

6 References

Constantine L. & Lockwood L. *"Software for Use. A practical Guide to the Models and Methods of Usage-Centered Design"*. Addison-Wesley, Reading, MA, USA, 1999.

Coutaz J. *"PAC – An Object Oriented Method for Dialogue Design"*. In Proceedings of the INTERACT'87 Conference, Elsevier Science, Stutgart, Germany, 1987.

Erikson T. & Thomas J. *"CHI 1997 Workshop. Putting It All Together: Towards a Pattern Language for Interaction Design"*, 1997. Summary available at http://www.pliant.org/personal/Tom_Erickson/Patterns.WrkShpRep.html.

Fincher S., Finlay J, Greene S. Jones L., Matchen P., Molina P.J., Thomas J. CHI-2003 workshop: *"Perspectives on HCI patterns: concepts and tools"*. Fort Lauderdale. USA. 2003. http://nitro.watson.ibm.com/chi2003Workshop

Goldberg A., Robson D. *"Smalltalk-80: The Language and Its Implementation"*. Addison-Wesley, Reading, MA, USA 1983.

James A. Landay and Brad A. Myers, *"Sketching Interfaces: Toward More Human Interface Design." IEEE Computer*, vol. 34, no. 3, March 2001, pp. 56-64.

Macromedia. *Macromedia Dreamveaver MX*. http://macromedia.com, 2002.

Molina P.J., Meliá S., & Pastor O. "*User Interface Conceptual Patterns"*. In Forbrig, Limbourg, Urban & Vanderdonckt (editors), "Proceedings of the 4th

International Workshop on Design Specification & Verification of Information Systems DSV-IS'2002", Springer Verlag, Berlin, Germany, 2002. ISBN 3-540-00266-9. In Lecture Notes in Computer Sciences. http://www.springer.de/cgi/svcat/search_book.pl?isbn=3-540-00266-9.

Molina P.J. *"User Interface Specificacion: from requirements to automatic code generation"* (In Spanish) PhD Thesis. Technical University of Valencia, Valencia, Spain, 2003. Available at: http://www.dsic.upv.es/~pjmolina/EN/thesis.html

Mullet K., McInerney P., van Welie M. CHI 2002 Workshop. *"Patterns in Practice: A Workshop for UI Designers"*, Minneapolis, USA, 2002.

Ruble D.A. *"Practical Analysis and Design for Client/Server and GUI Systems"*. Yourdon Press Computing Series, 1997.

SIGCHI. The UIMS Workshop Tool developers: *"A Metamodel for the Runtime Architecture of An Interactive System."* SIGCHI Bulletin, 1(24), pages. 32–37, 1992.

Smarty, PHP template engine. http://smarty.php.net/ 2002.

Tidwell J. *"Common Ground: A Pattern Language for Human-Computer Interface Design"*, 1999. http://www.mit.edu/~jtidwell/common_ground.html

van Welie, M. *"Wireframes"* 2001, http://www.welie.com/articles/wireframes.pdf.

van Welie, M., van der Veer G., Eliëns A. *"Patterns as Tools for UI Design"*. In International Workshop on Tools for Working with Guidelines, pages 313–324. Biarritz, France, 2000.

Velocity, Java Template Engines. http://jakarta.apache.org/velocity/ 2002.

Integration of HCI Needs with SE Methods Using OODPM Methodology

Offer Drori

The Hebrew University of Jerusalem, SHAAM – Information Systems,
P.O. Box 10414, Jerusalem, ISRAEL
offerd@cc.huji.ac.il

Abstract: Many information systems softwares currently being developed relate to the system's user interface. Notwithstanding, the separation between the fields of HCI and SE is customary only in a part of the academic arena. In most cases, human computer interaction (HCI) affects the overall process of software engineering (SE). The challenge, as we see it, relies on systems analysts' and software developers' understanding of organization and user needs, and the ability to translate these needs into functional information systems. In this paper, we will describe OODPM methodology that offers explicit support for HCI using object-oriented methods from the SE field.

Keywords: OODPM, SE and HCI, Prototype methodology

1 Introduction

In object-oriented design, the object is the building block upon which the system is constructed. It comprises a method (the algorithm according to which the object operates), data that the object activates, and an area in which data is transferred between the object and other programs or objects (Alter, 1996) (Booch, 1994) (Rumbaugh et al., 1991). Although in OO design an object is a building block, objects do not exist (or behave) in isolation. Rather, they participate in actions together with other objects in order to satisfy some objective. Also, relationships between objects defined by relationship invariants define the stable properties of the system that should not be violated by those actions (Kilov, 2002) (ISO/IEC).

Prototype methodology bases system planning and design upon the construction of a prototype, which is used to test, demonstrate, and evaluate the proposed system. Even though there are several kinds of prototypes, all allow the system to be perceived from the user's point of view. Instead of relying on the user's imagination, the prototype provides a tool that fully simulates the future system, thereby minimizing the risk of errors and misunderstandings (Martin, 1991).

OODPM embodies an integration of these approaches. The system is planned by defining objects (as well as relationships and behavior), while the user interface (the graphical design) is presented by means of a prototype. This approach has been presented in several publications (Drori, 2001) (Drori, 2000) (Drori, 1998) (Drori, 1996). A recent survey of large software houses in Israel (Drori, 2003), which examined the use of methodologies and tools in information systems development phases, found that 42% of the firms use UML methodology, 15% use OODPM (Drori, 2002), 12% use OMT, 10% use Mafteach (Mafteach), and 7% use Code & Yourdon.

2 OODPM Components

In the OODPM model, the activity (object) comprises six components (see Fig. 1):

(a) The user interface - window/screen of the computer system, depicting the activity of the future system.

(b) Verbal description of the process used by the activity (free text or pseudo-code). If the process will be automated, this description must be explicit, integral, and unambiguous – in other words, pseudo-code.

(c) Input data that will be entered into the window/screen of the information system.

(d) Output of the computer system window/screen (for example, a computer record consisting of the input fields and additional data, such as a user ID, date, etc.).

(e) Data format used in the activity.

(f) System state (Common communication area for objects, for example, Response code table).

Components (a), (b), (c), (e), and (f) are preconditions of the activity (i.e., they should be true before the activity can occur), and components (d) and (f) are post conditions of the activity (i.e., they should be true as an immediate result of the activity).

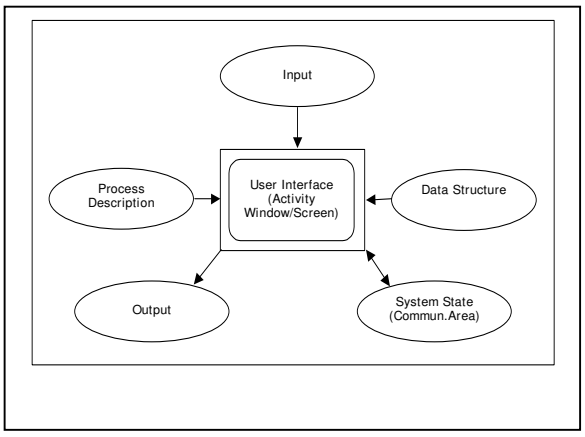


Figure 1: Object Components in OODPM

3 Template for OODPM (Ver. 5)

A template for the methodology has been developed to achieve the goal of planning an information system that gives the user the ability to influence his needs. The template includes 5 chapters that assist the system analysis to define the requirements of the future system with the user's collaboration. The five chapters are: 1. Initiation And Introduction;
2. Description of Existing System; 3. Feasibility Research; 4. Defining the New System;
5. Designing the System. The most important chapter in the user point of view is chapter 5 since he gets from it the user interface of the future system as defined by the system analyst.

The template paragraphs are:

1 Chapter On Initiation And Introduction

The process of initiating the project; the central problem that led to the project; the initiator, referral to the analyst, etc.

1.1 Initiation

1.2 Description of Background

1.2.1 General description of the organization

{Include here general organizational structure diagram}

1.2.2 Characteristics of activity

1.2.3 Quantitative operational data

1.3 Definition of the Problem

1.3.1 General diagnosis of the problems from the point of view of the consumer.

1.4 Aims

1.4.1 Aims of the organization

1.4.2 The aims of the organizational unit examined

1.4.3 Aims and gauges of an information system

1.5 Fundamental Assumptions, Framework, and Limitations

1.5.1 Scope of operation and its domain

1.5.2 Budgetary limitations

1.5.3 Timetable

1.6 Program For Carrying Out the Survey; Previous System Analysis Work

1.6.1 Method

1.6.2 Budget estimate

1.6.3 Timetable

1.6.4 Approval from the authorized decision-making level

2 Chapter on Description of Existing System

OOPDM focuses primarily on system planning, but also addresses the business specification stage. According to this approach, user needs to be implemented in the future system must be studied, but time must also be dedicated to studying the current situation in order to complete the requirements definition. Experience has shown that users tends to focus on those needs that have not been met by the current system, and tend to ignore the parts of the system that have met their needs. Without a stage to examine the current situation, only a partial definition of the requirements is likely to achieved.

2.1 System and Structure

2.1.1 Organizational structure

{Include here organizational unit structure diagram}

{Include here organizational work team structure diagram}

2.1.2 Technological character of the organization

2.1.3 Management characteristics

2.1.4 Development plans

2.2 Flow of Information

2.2.1 The flow of information among principals in the organization

{Include here functional flow chart}

{If needed use here Data Flow Diagram (DFD) content only}

2.2.2 Analysis of information

2.3 Procedures

2.3.1 Description of procedures

2.3.3 The timing and coordination of times between the procedures

2.4 Processes

2.4.1 Description of processes

2.4.2 The timing and coordination of times between the processes and activities

2.4.3 Handling inadequacies

2.5 Forms

2.5.1 Description and characteristics of forms

2.5.2 Examples of forms

2.5.3 Division of fields in forms

2.6 Files

2.6.1 Description, characteristics, and structure of files

2.6.2 Records

2.6.3 Special fields and specifications

2.7 Reports

2.7.1 Description, structure, and characteristics of reports

2.7.2 Contents of reports

2.7.3 Examples of reports

2.8 Linkages with Other Systems

2.8.1 Direction and depth of linkages

2.9 Resources in the System

2.9.1 Manpower

2.9.2 Equipment

2.9.3 Software

2.9.4 Hardware

2.9.5 Communication

2.9.6 Miscellaneous

2.10 Definition of Problems of Existing System

2.10.1 Process problems

2.10.2 Information problems

2.10.3 Coordination problems

2.10.4 Technological problems

3 Chapter on Feasibility Research

This chapter deals with the description of all alternatives that can be used to solve the problems as responses to requirements and problems from the previous chapter.

3.1 Requirements of the New System

3.1.1 Required tasks

3.1.2 Inadequacies to be corrected

3.1.3 Additional improvements

3.1.4 Aims and gauges

3.1.5 Constraints

3.2 Alternative Solutions

3.2.1 Description of solutions

3.2.2 Describe the solutions as responses to requirements and problems

3.2.3 Advantages and disadvantages of each solution

3.3 Economic Feasibility

3.3.1 Lifetime of the system

3.3.2 Development cost

3.3.3 Operating cost

3.3.4 Comprehensive analysis for each solution.

3.4 Technological Feasibility

3.4.1 Possibility of development from a technological standpoint

3.4.2 Possibility of integration into an existing technology

3.4.3 Technological stability and its reliability

3.4.4 Comprehensive analysis for each solution

3.5 Operational Feasibility

3.5.1 Extent of required change to the existing situation

3.5.2 Anticipated resistance and/or assistance

3.5.3 Comprehensive analysis for each solution

3.6 The Chosen Solution

3.6.1 Comprehensive analysis

{Include here Delphi model's weighted grading table summary}

3.6.2 Recommendations for deciding on the system

3.6.3 Decision

3.7 Action Program for Project Development

3.7.1 Manpower required for development

3.7.2 Hardware and software required for development

3.7.3 Timetable

3.7.4 Approval of the authorized decision-making level

4 Chapter on Defining the New System

This chapter deals with the definition of the new system and, in practice, with its design.

4.1 Perceptions and Principles

4.1.1 Perception of the computerized system

4.1.2 Organizational aspects

4.2 Constraints and Limitations

4.2.1 Constraints and limitations of the computer system

4.2.2 Constraints and limitations of the organization

4.3 General Description

4.3.1 System components

{Include here a structural chart of the system}

4.3.2 System flow

4.3.3 Scope and frequencies of operations

4.4 Data Structure for System Components

4.4.1 Input contents

4.4.2 Output contents

4.4.3 Structure of files and their contents

4.5 Sub-systems

4.5.1 Components of each sub-system.

4.5.2 Inputs of each sub-system.

4.5.3 Outputs of each sub-system.

4.5.4 Files of each sub-system

4.5.5 Operations of each sub-system.

4.5.6 Operational screens of the sub-system.

4.5.7 Developing phases recommended

4.6 Linkages with Other Systems

4.6.1 Technical linkages

4.6.2 Operational and organizational links

4.7 Hardware and Software Requirements

4.7.1 Equipment requirement

4.7.2 Software requirements

4.7.3 Information security and backup requirements.

4.8 Operation, Conversion, and Assimilation Requirements

4.8.1 Testing and trial requirements

4.8.2 Conversion requirements

4.8.3 Essential requirements for preliminary operation

4.9 Control Requirements and Feedback

4. 9.1 Quality

4.9.2 Timetable

4.9.3 Resources

4.9.4 Reporting

4.10 Possibilities For Expansion

4.10.1 Possible requirements for increasing the volume of data

4.10.2 Possible requirements for the increase of functions

4.10.3 Possible requirements for technological changes

4.10.4 Possible requirements for linkages with other systems

5 Chapter on Designing the System

This chapter relates to designing the system, which is the last stage before programming.

5.1 Detailed Description of the System

5.1.1 Description of the system

5.1.2 Function of various components

5.1.3 Processing procedure

5.2 Detailed Description of Each System Object (Component)

5.2.1 Drawing of the system screen (or of another main component, such as a printout)

{Include here drawing of the system screen/window comprise the user interface of the object (GUI)}

5.2.2 Input from each object

5.2.3 Output to each object

5.2.4 Processing of each object (the method)

5.2.5 Messages from each object

5.2.6 Data structure of each object

5.3 System Navigation Diagram

{Include here system navigation diagram}

5.4 Files / Tables

5.4.1 Description and characteristics

5.4.2 Examples of records

5.5 Databases

5.5.1 Logical organization

5.5.2 Physical organization

5.6 Software and Development Environment

5.6.1 Infrastructure software

5.6.2 Programming languages and application generators

5.7 Equipment / Hardware

5.7.1 Power of processors

5.7.2 Peripheral equipment

5.7.3 Communication equipment

As we can see, chapters 4 and 5 are based on the user's needs. In chapter 5, paragraph 5.2 (detailed description of each system object) is the most important part of the planning work. In this paragraph, the designer expresses his or her understanding of the user's needs by demonstrating the user interface of the future system. Using this technique, the user can test the system's attributes and processes, and, most importantly, can modify options or attributes and assert his or her needs.

4 Advantages of Designing Information Systems with OODPM

OODPM combines the SE method for developing information system with HCI needs. Prototyping is a low-cost means of demonstrating the idea behind a design. A prototype can be easily modified, and the design evolves gradually to an optimal solution before the system is implemented. Prototyping offers

savings in time and cost to develop something that can be tested with real users (Luqi, 2003). Using the prototyping environment represents significant support for the usability of the prototyping effort. It can be expected to improve the quality of the systems developer, decrease the time and reduce the cost of software development, enhance the developer's satisfaction and productivity, and in most ways makes software development a more delightful and exciting task (Johnson, 1999). Using Prototype as a main part of the object definition is one of the major integrations of SE and HCI. The advantages of designing an information system with OODPM are:

1. It creates greater understanding between the developer and the user.
2. It improves the understanding between the designer and the developer.
3. Part of the developing work is dictated by the design process
4. The system is finely tuned to the user's needs.
5. It enables a reuse of the software, thereby economizing on organizational resources.
6. Reusing proven software components improves the reliability of the system.
7. It enables the creation of object-oriented systems in conventional programming languages.

5 Summary

In most cases, human-computer interaction affects the overall process of software engineering. Prototyping is the main bridge between these two fields. SE is a platform that empowers software designers to develop software based on the requirements specification. Requirements specification is based, first of all, on the human factor that is involved in the HCI process. In this paper, we describe methodology (OODPM) in the domain of the object-oriented field that offers explicit support for HCI using the SE method. A template for word 2000 is available for convenient use of OODPM methodology. The first version of OODPM was published in 1994, and the most recent version (5.0) was published in February 2003. The latest version is the first one to be in English (all other versions were published in Hebrew). More information about the methodology can be found at the methodology site (OODPM). The template may also be downloaded at the site.

References

Alter, S. (1996), *Information Systems - A Management Perspective, 2nd ed.*, Menlo Park, CA: The Benjamin/Cummings Publishing Company.

Booch, G. (1994), *Object Oriented Analysis and Design with Applications. 2 ed.*, Redwood City, Calif.: Benjamin/Cummings.

Drori, O. (2003), Use of CASE Tools and Object Oriented Methodologies, Maydaon - The Israeli Journal for Information Technology, Tel-Aviv: The Israel Chamber of System Analysts, No. 133, June 2003, 21-26 (in Hebrew).

Drori, O. (2002), *Planning information systems using OODPM methodology - user guide, version 5*, Jerusalem, Academon, 2002 (in Hebrew).

Drori, O. (2001), HyperCASE - Case Tool which Supports the Entire Life Cycle of OODPM, *Proceedings of the ECOOP2001 conference workshop on Automating Object-Oriented Software Development Methods* (June 2001, Budapest, Hungary).

http://shum.huji.ac.il/~offerd/papers/drori062001.pdf

Drori, O. (2000), Analysis and Design of Information Systems Using OODPM - Practical Implementation, *Proceedings of the OOPSLA 2000 Workshop on Behavioral Semantics* (Minneapolis, Minnesota, USA, October 15, 2000).

http://shum.huji.ac.il/~offerd/papers/drori102000.pdf

Drori, O. (1998), Definition of requirements for an OODPM-Based information system using hypertext, in: *Proceedings of the ECOOP'98 Workshop on Precise Behavioral Semantics with an Emphasis on OO Business Specifications* (Brussels, Belgium, July 24, 1998), ed. by H. Kilov, B. Rumpe, Munich University of Technology, TUM-I9813, pp. 75-84.

http://shum.huji.ac.il/~offerd/papers/drori071998a.pdf

Drori, O. (1996), Planning and Design of Information Systems Using OODPM, in: *SIGSOFT Newsletter*, New York: ACM SEN (Software Engineering Notes), Volume 21 Number 5, p. 95.

http://shum.huji.ac.il/~offerd/papers/drori091996.pdf

ISO/IEC JTC1/SC21. Open Distributed Processing – Reference Model: Part 2: Foundations (ITU-T Recommendation X.902, ISO/IEC 10746-2).

Johnson, C. (1999), Why Human Error Analysis Fails to Support System Development, *Interacting with Computers*, Vol. 11, No. 5, 517-524.

Kilov, H. (2002), Business Models: A Guide for Business and IT, N.J.: Prentice-Hall.

Luqi, Z. Guan. (2003), A Software Prototyping Framework and Methods for Supporting Human's Software Development Activities, *Proceedings of Bridging the gaps between SE and HCI workshop*, held at ICSE'03, 114-121.

Martin, M. (1991), *Analysis and Design of Business Information Systems*, New York: Macmillan Publishing Company.

Mafteach http://www.methoda.com/

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991), *Object-Oriented Modeling and Design*, Englewood Cliffs, N.J.: Prentice-Hall.

OODPM http://oodpm.huji.ac.il

Quantitative Architecture Usability Assessment with Scenarios

Mugurel T. Ionita[1], Pierre America[2], Henk Obbink[2] & Dieter Hammer[1]

[1]Technical University Eindhoven, 5600 MB, Eindhoven, The Netherlands

M.T.Ionita@tue.nl, D.K.Hammer@tue.nl

[2]Philips Research, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands

Pierre.America@philips.com, Henk.Obbink@philips.com

Abstract: In this paper, we propose a quantitative scenario-based method for dealing with the quality attributes for future software systems before any architecture implementation effort is actually done. The exploration of the possible design variants for a future system is done using so-called user scenarios. The method supports the software architects in developing future systems that meet their specific quality requirements. The method was validated in an industrial case study focusing on the usability aspects.

Keywords: Usability, quantitative assessment, scenarios.

1 Introduction

When designing software intensive systems, we need to know about the functional aspects, but also about the quality attributes of the system: How fast will it be? How many users will it support simultaneously? How long it will take a user to complete a certain action? These quality attributes are determined by the architecture of the future system, and must therefore be considered at an early stage of the design. Bosch (2000), Clements et al (2002) and Folmer et al (2003) proposed scenario-based methods for dealing with the various quality attributes. In these methods, the architecting team initially proposes the system architecture. The architecture is then assessed by its stakeholders to judge how well it will support specific quality requirements with so-called assessment scenarios. In this context the assessment scenarios are descriptions of possible changes that may appear in the future, very similar to use cases. The architectures are often redesigned as a result of these processes, because the envisaged scenarios are created at the time of assessment and are therefore not supported in the proposed system architecture.

In this paper we propose a different scenario-based approach to minimize the need for the architecture re-designs. This approach considers the uncertainty associated with various possible future scenarios before any effort has been invested in designing the architecture. This is done by considering different user scenarios, which describe the future systems and the possible user interactions with these systems. The architecture is then proposed based on these user scenarios.

Although there is an undeniable advantage in designing system architectures based on the user scenarios, we have identified a potential problem: how can we make sure that the proposed architectures will also meet their future quality requirements, and in particular, their usability requirements? The solution is rather trivial: the systems must be designed for that quality. The architects must therefore ensure that the proposed user scenarios describe a system with a specific, measurable, and adequate level of quality. However, once this has been achieved, a related problem will arise: which of these scenarios should be considered in the design?

To deal with these problems we propose a systematic quantitative method for assessing the quality level expressed at a user scenario level. The method also provides a mechanism for informed decision making with respect to those user scenarios that should be further considered for the system architecture design. This is called the Systematic Quantitative Analysis of Scenarios' Heuristics (SQUASH) method.

This paper is organized as follows. Section 2 introduces the SQUASH method and. Section 3 describes a case study conducted for Philips Medical Systems, where the method was applied for a set of user scenarios. Section 4 presents some conclusions. Section 5 completes the paper with ideas for future work.

2 The SQUASH Method

2.1 The Method Description

The SQUASH method consists of two phases: the information gathering phase, and the decision-making phase. Each phase contains one or more steps as listed below.

Information Gathering Phase

1. Identify Stakeholders
2. Identify Quality Objectives
3. Make the Quality Objectives Quantifiable
4. Analyse Scenarios
5. Present Scenarios Quality Profile
6. Improve the Scenarios

Decision Making Phase

7. Select the Scenarios

A detailed description of the method's steps is provided below, and an overview of the method is given in Figure 1.

Information Gathering Phase

Step 1 - Identify Stakeholders

The first step in SQUASH is the identification system's stakeholders. The stakeholders will provide information about the different qualities expected from the future system.

Step 2 - Identify Quality Objectives

In step two we identify the stakeholder objectives with respect to the stated quality attributes. This can be done directly by asking the stakeholders about their quality requirements, or indirectly by studying existing requirement specifications or business proposals.

Step 3 – Make the Quality Objectives Quantifiable

The third step is to define the stakeholders' quality objectives in a specific and measurable way, so that they can be used for the quantitative assessment of the user scenarios. The identification of proper metrics for the various quality objectives will help to focus the analysis and provide concrete points for improving the scenarios.

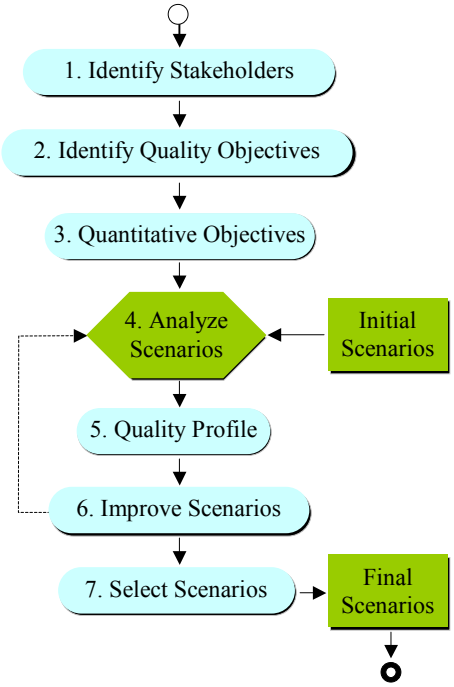


Figure 1: The SQUASH Method Overview.

Step 4 – Analyse Scenarios

In step four the user scenarios are analysed to find out how well they support the various quality objectives of the stakeholders, as defined in step three. The user scenarios should be structured into scenes beforehand. A scenario scene consists of a task or a combination of tasks that the user has to perform to achieve a result. The result of a scene is usually used as input for the next scene along in the scenario story line. The quality objectives are then assessed and quantified per scene. This type of analysis is called *bottleneck analysis,* as it will point out which scenes contain certain quality attributes that are limited or uncontained. The analysis is performed for each of the user scenarios.

Step 5 – Present Scenarios Quality Profile

The result of the analysis is a comparative chart containing the stakeholders' quality objectives and their estimated values per scene in the various user scenarios. This representation is called the scenario quality profile.

Step 6 – Improve the Scenarios
If the resulting quality profiles of the scenarios are acceptable to their stakeholders, the scenarios remain unchanged. Else, the user scenarios will have to be improved. The improvements particularly affect those scenario scenes that hinder certain qualities.

Decision Making Phase
Step 7 – Select the Scenarios
At this stage, the user scenarios contain sufficient qualitative information to be used later in the design of the future system. However, only a small set of these user scenarios can be considered further for the design. Depending on the situation, we could use subjective reasoning or objective selection to decide on the final scenario set. Subjective reasoning implies that the stakeholders and the architects use the hard data provided by the quantitative analysis to make a decision; the decision is taken intuitively. The objective selection implies that a certain multiple-criteria decision-making algorithm is applied in selecting the optimum scenario set; the algorithm makes use of the hard data provided by the analysis. If a selection algorithm is applied, the results will be re-discussed with the stakeholders to see if these results meet their actual expectations.

2.2 The Method Applicability for Usability Assessment

The SQUASH method has been developed in particular for estimating the usability of a future system at a scenario level. The method provides the means to reason about the usability of a system, before it is implemented, by quantifying the factors that contribute to the final usability of the system.

In general usability can be define as: ease of use, ease of learning, and user satisfaction, Jordan et al (1996) and Rosson et al (2002). ISO 9241-11 standard defines usability as, "*the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use",* where

- *Effectiveness refers to the accuracy and completeness of accomplishing a task,*
- *Efficiency refers to the expenditure of physical and human resources for a particular task in relation to effectiveness,*
- *Satisfaction refers to the freedom from discomfort, and positive attitudes towards the use of the system.*

For our case study we will use the usability definition as proposed in ISO 9241-11 standard, but specialize it for our context.

In order to assess these usability factors like effectiveness, efficiency and satisfaction in general subjective measures are used. For example the users are asked to rank these factors on a subjective scale from 1 to 5. Although the results of this subjective assessment will show the relative preference of the users for these factors, they don't provide any information on why the users gave those rankings and furthermore what should be done to change that. We want to improve on this aspect therefore we propose that before assessing the usability factors at a scenario level, these factors should be defined in a specific and measurable way. In this way the assessment results will provide more objective results in terms of quantitative information about the specific levels of the usability factors.

In the next section we will illustrate in a concrete case study how we quantified and assessed the general usability attributes like efficiency, effectiveness or satisfaction.

3 The CathLab Case Study

We applied the SQUASH method to assess the usability expressed at a scenario level in a real industrial project for Philips Medical Systems. The user scenarios of a professional medical product were studied. The context of this case study is given in the following subsection.

3.1 The Context

The context of our case study is the medical domain, namely the Catheterisation Laboratory (CathLab). CathLab is a specialized laboratory in which the cardiologist treats the patients who suffer from obstructions of their blood vessels, in particular the vessels of the heart (i.e. coronary artery disease). The cardiologist first has to localize the obstructed area of the blood vessel, referred to as stenosis, measure its length and width, and finally enlarge that blood vessel. The localization of the stenosis is done with a special tube, called a catheter. For inserting the catheter, the cardiologist makes a small opening in one of the patient's blood vessels (e.g. veins which are located in an accessible area of the body such as the neck or the leg). Through this opening the catheter is inserted and then guided towards the heart. The catheter is guided using fluoroscopy, which means inserting low dose of contrast fluid via the catheter and exposing the patient's region of interest to low intensity X-rays. Fluoroscopy is used

to visualize the position of the catheter's tip within the patient's body in real time. When the catheter reaches the heart area, the cardiologist performs an exposure, which means inserting high dose contrast fluid and exposing the relevant region of the patient's body to high intensity X-rays. The exposure provides accurate images of the heart blood vessels. The accuracy is important to localize the stenosis. The X-ray images acquired are displayed on a monitor, Figure 2.

Figure 2: CathLab Picture – The Intervention Room.

Next, a stent is deployed via the catheter's guide wire. The stent is a special thin aluminum cylinder that can be enlarged in order to correct the stenosis During the intervention, the cardiologist uses different systems situated in the two rooms of the CathLab. The intervention room houses the X-ray system and the patient monitoring system, while the control room houses the patient data logging system, and reviewing workstations for patient studies (e.g. magnetic resonance studies, or X-ray studies).
The need for increased efficiency of the CathLab requires the integration of these systems. The goal of the integration is to reduce the number of systems, tasks and medical personnel required in the CathLab, and to reduce the intervention time. Therefore, the usability of the new integrated CathLab is an important quality required.

3.2 The User Scenarios

Five main user scenarios have been proposed to explore the design variants of such an integrated medical system. The scenarios start with a minimal integrated CathLab in which the systems are still distributed within the CathLab, being operated by a cardiologist or technician, and gradually build towards a fully integrated CathLab in which the systems are translated into individual applications, running on a single workstation, and are controlled only by the cardiologist. An overview of the different user scenarios is given in the following paragraphs.

The Minimal Integration Scenario, on short called Min., presents the case where only standard DICOM is used for integrating the CathLab systems.

The Data Integration Scenario, on short called Data, focuses on sharing of data. If data is produced by one system, can a second system, read, understand or even change it?

The Presentation and Control Integration Scenario, on short called PC, are two sides of the same coin. Presentation integration is accomplished when two or more systems can present their results in a similar way. Control integration on the other hand, means that two or systems can be controlled in similar ways. In practice, these types of integration are rarely separated, but are joined to create a common look and feel.

The Workflow Integration Scenario, on short called Wf., presents the case where the systems work together to support the workflow of their users. It is important that this behaviour is flexible and can be adjusted to meet specific workflow needs and habits.

The Full Integration Scenario, on short called Full, presents a fully integrated CathLab system. Besides the above-mentioned integration levels, new features that are enabled by this high level of integration are described.

3.3 Usability in the CathLab

For defining the usability we started from the ISO 9241-11 standard and specialized it for the CathLab context. In our case, usability will be perceived as, "*the extent to which the integrated CathLab can be used by the medical personnel (cardiologists, technicians, and nurses) for interventional purposes with effectiveness, efficiency and satisfaction in detecting, assessing and correcting human coronary artery diseases",* where:

- *Effectiveness* refers to the accuracy (the absence of mistakes or errors) and completeness (supporting all the steps) of the intervention,
- *Efficiency* refers to the expenditure of physical and human resources (time, people, and materials) for a particular task in relation to effectiveness, and
- *Satisfaction* refers to the freedom from discomfort, and positive attitudes towards the use of the CathLab systems.

These three factors will be detailed and defined in a measurable way to be further used in when assessing the usability of the CathLab at a user scenario level.

3.4 Applying the SQUASH Method

The following sections describe how the SQUASH method was actually applied for the quantitative usability assessment of the user scenarios in our case study.

Step 1 - Identify Stakeholders

From a usability point of view, the medical personnel, i.e. the cardiologists, the nurses and the technicians operating in the CathLab, and the hospital administration, are the only people who have a stake in the new integrated CathLab.

Step 2 - Identify Quality Objectives

The hospital administration is interested in increasing the efficiency of the CathLab. The cardiologist wants a CathLab that is easy to use and has a lower X-ray activity during the intervention (i.e. effectiveness and satisfaction). The nurse assists the cardiologist in the intervention room. The nurse's main usability objective is a sterile and tidy CathLab environment, a comfortable physical mean of support for the patient during the intervention, and as low dose as possible of X-ray radiation received during the intervention (i.e. effectiveness and satisfaction). The technician also assists the cardiologist, but from the control room, where he or she operates the patient monitoring systems. The technician's usability objectives are a CathLab with fewer user actions per task, and with functions that are easy to learn (i.e. efficiency).

We used this information to translate the usability objectives into more concrete usability factors, as shown in columns 1 and 2 of Table 1. Note that the shaded usability factors will not be considered further in the user scenario assessment.

Step 3 – Make the Quality Objectives Quantifiable

In this step, we associated the usability factors identified in the previous step with proper metrics, as shown in column 3 of Table 1, and acceptance levels, Table 2. Although we had identified a large number of usability factors and metrics, the user scenarios presenting the integration of the CathLab did not improve on all these issues, i.e. the shaded factors in Table 1. We therefore only considered a reduced set of these factors in the analysis, as shown in Table 2. This set contains the following:

- *Personnel involved* measures the number and type of persons involved to complete a certain activity, such as cardiologists, nurses and technicians.
- *Number of atomic actions* accounts for the number of indivisible operations the user has to do to complete a task. The metrics used for this factor are: the number of walks between the two rooms of the CathLab, the number of re-sterilizations due to an interaction with non-sterile items, and the number of operated controls (e.g. buttons pressed).
- *Patient comfort* measures the patient's overall level of satisfaction. Although satisfaction here is a very subjective usability factor, which depends on many other issues, we tried to identify it for as many metrics as possible. From an integration point of view, we can estimate patient comfort measures as being the intervention duration and the physical support for the patient in the various scenarios. For these metrics we assumed that the longer the duration of an intervention, and the less physical support (eye protection, soft table top, etc.), the more uncomfortable the patient would feel.
- *Invasiveness* involves those factors that would infringe the privacy of the persons in the CathLab (i.e. the patient or medical personnel). The factors identified for quantifying the invasiveness are: the total duration of the X-ray exposures, the total duration of the fluoroscopy study, and the total amount of contrast agent used. The smaller the X-ray dose is received during the interventions, the better for both the cardiologist and the patient.

Usability Objective	Usability Factor for the CathLab	Metrics
Efficiency	Personnel involved	Number of Cardiologists
		Number of Nurses
		Number of Technicians
	Number of atomic actions	Number of walks
		Number of re-sterilizations
		Number of buttons
Effectiveness	Error rate	Image quality
		Number of buttons
	Stent accuracy	Image quality
Satisfaction	Patient comfort	Intervention total duration
		Physical support in the CathLab
	Invasiveness	X-ray Exposure Time
		Fluoroscopy Time
		Contrast agent amount

Table 1: The stakeholders' usability objectives and relative metrics

Step 4 – Analyse Scenarios

In step four we assessed the usability factors at a scenario level based on the metrics and acceptance levels provided in Table 2. The assessment was an elaborate process that involved scenario walkthrough sessions with architects and stakeholders. The scenarios were analysed per scene and quantified with respect to each usability attribute. This is called bottleneck analysis because it reveals the particular scenes where the values of certain usability factors may exceed the acceptance levels. The value of each usability attribute was studied per scenario scene, which than allowed us to reason about the entire scenario quality levels by summing up, or taking the maximum, of the individual usability estimates obtained per scene. The hard data obtained in this way will be used in the decision making process when selecting the final user scenario(s) to be considered for the design.

Usability Attribute	Usability Acceptance Levels		
	Poor	Good	Excellent
Number of Cardiologists	> 1	1	0
Number of Nurses	>1	1	0
Number of Technicians	> 1	1	0
Number of walks	□1	0	0
Number of re-sterilizations	□1	0	0
Intervention duration	> 70 min	45 – 70 min	45 min
X-ray Exposure Total Duration	> 40 sec	20 – 40 sec	20 sec
Fluoroscopy Total Duration	> 20 min	10-20 min	10 min

Table 2: The Usability Metrics – Acceptance Levels

We did not always have sufficient data at a scenario level to estimate or measure the value of certain usability factors. To overcome this problem we consulted other information sources such as specifications of existing systems, or domain experts. In general the experts provided similar estimated. However, if there are significantly differences in the provided estimates, you should either organize joint meetings with all the domain experts and discuss these values until agreement is achieved, or you should consider all of the data and explain the differences. The second suggestion is more likely to be adopted, because joint workshops are usually hard to organize, require a lot of preparation time, and rarely result in single values being agreed upon by all participants.

Step 5 – Present Scenarios Quality Profile

The results of the scenarios' usability analysis is then summarized and presented in a comparative form, as shown in Table 3. In order to visualize the relationship between the estimates obtained from the scenarios and the acceptance levels set in Table 2, the cells containing the values from the different usability estimates are shaded. Dark shading (red) represents a poor usability level, which in some cases could be unacceptable. Light shading (yellow) represents a good usability level, which usually corresponds to the planed level. No shading (green) represents an excellent usability level.

Usability Attribute		Min	Data	PC	Wf	Full
Number of Cardiologists		1	1	1	1	1
Number of Nurses		1	1	1	1	1
Number of Technicians		1	1	0	0	0
Number of walks		4	3	0	0	0
Number of re-sterilizations		0	0	0	0	0
Intervention duration	Avg.	65 min	50 min	45 min	45 min	30 min
	Max.	100 min	90 min	65 min	65 min	45 min
X-ray Exposure Time	Avg.	20 sec	20 sec	20 sec	20 sec	15 sec
	Max.	30 sec	30 sec	30 sec	30 sec	25 sec
Fluoroscopy Time	Avg.	10 min	10 min	10 min	10 min	10 min
	Max.	15 min	15 min	15 min	15 min	15 min

Table 3: The Scenarios' Usability Profile

Step 6 – Improve the Scenarios

We could not improve the scenarios content in our case study, as the scenarios were from an existing project. Our objective was simply to conduct the usability analysis and observe the results when applying SQUASH. However, we do have a couple of suggestions for the improvement of the scenarios.

Decision Making Phase

Step 7 – Select the Scenarios

We are now able to interpret the results provided in Table 3. From a usability point of view the most preferred scenario is the Full Integration Scenario followed, in descending order, by Workflow, Presentation and Control, and Data, with the least preferred scenario being the Minimal Integration Scenario. However, a selection should not only be based on the usability profile of the various scenarios. Other qualities such as performance, costs and risks have to be considered during the selection.

4 Conclusions

This method contributes to the existing software architecture development methodology in the following ways:

- The method proposes a systematic quantitative way of analysing user scenarios of a system before committing effort and budget into designing the system's architecture.
- The accuracy of the analysis improves by assessing the various qualities per scenario scene. The scenes that may hinder the usability of the final system can be explicitly identified.
- Working with scenarios in form of stories has been an efficient method of communication for the architects and the stakeholders of the system. The analysis of the usability at a user scenario level means that real system prototypes do not have to be built.
- The system's stakeholders are given a clear overview of the relative benefits of the different scenarios in the form of written user scenarios annotated with quantitative information about their various quality levels. With this quantitative information at hand, the decision makers are better able to decide which scenarios should be considered for the design.
- The usability levels specified in the user scenarios can be used later to measure and validate the implemented systems.

5 Future Work

The SQUASH method presented in the present paper has been applied for the scenarios of a real product. However, we have seen that decisions must be based on more than one quality attribute, namely usability. In the future will look at integrating more quality attributes into the decision-making process, such as performance, cost, and the risks associated with the various user scenarios.

Acknowledgements

We would like to thank our colleagues Eelco Rommes, Jan Gerben Wijnstra, Marc Stroucken, Maarten Bodlaender and André Postma, for their valuable feedback during the intermediate phases of the work presented in this paper. This research was founded by Stichting Technische Wetenschappen (STW), Project EWI.4877 and conducted within Philips NatLab.

References

Jordan, P.W., Thomas, B., Weerdemeester, B.A. & McClelland, I.L. (1996), *Usability Evaluation in Industry*, Taylor & Francis, pp.189-192.

Bosch, J. (2000), *Design and Use of Software Architectures,* Addison Wesley.

Rosson, M.B. & Carroll J. (2002), *Usability Engineering: Scenario-Based Development of Human Computer Interaction*, Morgan Kaufmann Publishers.

Clements, P., Kazman, R. & Klein, M. (2002), *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley.

Folmer, E., Gurp J. & Bosch, J., (2003), *Scenario-based Assessment of Software Architecture Usability,* ICSE 2003 Workshop Proceedings, Bridging the Gaps Between Software Engineering and Human-Computer Interaction, Portland, Oregon, USA.

Multi-faceted development

Ebba Thora Hvannberg
University of Iceland, Hjardarhaga 2-6, 107 Reykjavik, Iceland
ebba@hi.is

Abstract: In this paper we present an example of how two disciplines Software Engineering and HCI can work side by side on the same problem, having the same goal but with different approaches. Such work can help us understand the similarities, differences and the synergies that multi-faceted development can bring. It should motivate us to understand the goals of the disciplines, and their role in the future development environments. Experience with working on this particular case has convinced us that it is important to continue to develop two strong disciplines but that interfaces between them must be built.

Keywords: HCI, Software Engineering, Quality, Case study, Facet

1 Introduction

Human computer interaction (HCI) is an interdisciplinary field, interacting with many areas. It is a part of computer science, but has strong roots in psychology, sociology and industrial design. "Human Computer Interaction is concerned with the design, evaluation, and implementation of interactive computing systems for human use and with the study of major phenomena, surrounding them" (Hewett et. al 1996). HCI includes knowledge from computer science in the area of application design and interaction design and the engineering of user interfaces. To understand what it means to engineer the user interfaces, we can relate to the definition of software engineering as defined in IEEE 610.12 (IEEE 1990) "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."

The standard 13407 (ISO 13407 1999) recommends using an interdisciplinary team in user interface design. Yet, applications show there is much research needed to understand how they can interact (Bryan-Kinns and Hamilton, 2002).

More interesting ideas are often created in an interdisciplinary collaboration. Solutions to problems appear when you look at them from multiple perspectives. We should not limit ourselves using just one model to describe user interfaces or software or one view but multiple ones. We may connect these models with interfaces so that they can be used together. All too often we have tried to build large modeling or programming languages that either encompass everything or dilute ideas that are reduced to some least common denominator.

The types of interaction between humans and computers are evolving. We have different software and hardware technologies. The problems we need to solve, the users and the physical and spatial contexts are changing. As we discuss the two areas, we need to be careful not to delve too long on the past but look at the roles and the interaction of these disciplines in the future and in relationship with other disciplines too.

In this paper we present an example of the two disciplines working side by side. We argue that it is not necessary to close the gap but to bridge it with appropriate interfaces so that cooperation between disciplines and people studying them can take place. Such case studies of multi-faceted development will help us understand the differences between HCI and SE and the synergies that the disciplines can form.

2 Unified goals

Software engineering is concerned with process and product quality. ISO 9126-1 (ISO 9126-1 2001) divides product quality into external and internal quality on one hand, and quality of use on the other. External and internal quality includes functionality, reliability, usability, efficiency, maintainability and portability. According to the standard, quality in use

is the user's view of quality and includes effectiveness, productivity, safety and satisfaction. Internal quality indicates external quality that again indicates quality in use.
Additionally, software engineering monitors the use of resources while building, maintaining and operating the system.
Standards that cover quality of HCI have similar quality goals. One can continue to specify such quality goals and I can for example name trust as being a perceived measure (Corritore, Kracher, Wiedenbeck, 2003). However, HCI has also been concerned with the effect the system has, such as its social impact (Shneidermann and Rose, 1990). There are yet other types of goals, but they are related to functionality of the system for solving the problems, such as sustainability, economy, physical and mental health, sociability, learning etc.
Although the goals of the two areas look similar, the perceived difference between the two areas may be that software designers think that the system should solve as much of *a problem* as possible, and look at the user as an operator. HCI looks at the human as a primary actor of the system that completes *a task*.

3 Network of human and system actors to solve problems

Undoubtedly, in the future there will be more automation and less human operation. This is especially apparent in areas where users are mediators between the system and primary initiators of tasks. Twenty five years ago I worked as a telephone operator in a telecommunication company. When a caller wanted to make an international call, he had to call the calling center where he was connected. There, operators wrote requests on paper forms that were given to other operators that made the call. The length of the telephone call was then recorded on -paper. Automating this bookkeeping of the telephone requests would have been solving the wrong problem. As we know, the telephone operator has become mostly obsolete. Today, we see many examples of similar human computer interaction patterns, e.g. in air traffic control, hamburger fast food restaurants, the classroom etc. The third party operator will gradually have a diminishing role and even disappear as a mediator, and the initiator of the task can directly solve his problem with the aid of a computer. People are no longer constrained in performing work in some certain environment, they are free to stay or move around.

Traditionally, software developers have looked at one type of user and one system. Users are now participants in a world assisted by diversity of computers. Development methods are beginning to acknowledge this change. Newer approaches must view a user of many systems and even many users collaborating in using those systems.
Michael Jackson (Jackson, 1995) reminds us to look at the problem domain, to analyse what problem needs to be solved, not who solves it or how. We have been taught to separate the what from the how, but there are three stages. The first stage, the problem stage, is to define what problem we need to solve, the second part, the boundary definition, is to decide who solves the problem, that is how the work is divided among people and systems, and the third part, the design including the architecture, how it should be done and how the subsystems and people interact.
We will further focus on the second stage. Before we design how to solve a problem, we need to define who should solve it. Should it be our new system, other systems, and should people solve a part of the problem even manually. Normally, the people we ask to solve a problem will do this with the help of systems. Just as we define the interfaces between our new system and other systems, we also define the interfaces between our new system and the users that are solving a part of the problem cognitively. We even may go as far to define the interfaces between humans, i.e. human to human interfaces. This may very well influence the human to computer interaction. Since it is a network, we want to find out to how all the components communicate. All that may affect the performance of the solution. As in a network, a single component can slow down the end-to-end performance.
In the next section we will describe the role of people in our networked system as has been described above.

4 The role of humans in an evolving world view

4.1 Humans as resources

In the beginning of this paper, we described HCI as being a part of computer science. We can look at HCI in a similar way as we treat databases, networks, and operating systems. Those are different disciplines that help us design and implement certain parts of the system – interaction design is comparable with these areas. HCI is broader since it covers also the second stage, i.e. Boundary

specification, as is described in the previous section. Once the boundary specification is in place we can carry on with Architecture specification where we decide who does what and how they interact. Note however that HCI is not as broad as to cover the problem definition, i.e. the first stage because at that stage we have not decided who does what. Some methodologies define the interaction between human and computer prematurely, before having decided the distribution of the workload.

The human is a resource, similar to the physical network, CPU or disk space. They are constraints just as disk access or bandwidth. However, requirements specification usually doesn't put cognitive capabilities forward as a measurable entity or formal constraint. What makes it of course harfder to design for is its variability. Unlike physical resource, a human resource does not have a guaranteed capacity but can be increase through learning or decreased under external influences.

4.2 Humans as natural systems

Computer systems receive information from mechanical and electrical systems and provide feedback to them, e.g. airplanes or fish processing assemblies. Computer systems also receive such input from natural systems, e.g. earthquakes, indirectly through sensors and meters. In much the same way, people are sources of data and action and receivers of input.

We thus need a model of the human to understand how it interacts with the computer system, similar to other natural systems that a computer system interacts with. Computer systems capture information from natural systems and try to store information about them and even predict their behaviour.

4.3 Resource capabilities and constraints

When determining who should solve the problem, we need to know what resources we have. How can the resources meet the required achievements? How performing are the computers, the network, and the people. Do all these together meet our performance expectations for the problem solving or do we need to boost their performance. Additional resources e.g. of memory and processing power can be added to a computer, networks can be scaled and people can be trained to increase their performance. The software that we are building can also help meet the gap by designing it to compensate for the lack of human resources. Operating systems can manage resources better; network services can balance the load to computers. Functionality can be added to systems to increase people's capabilities, increase their perception, processing and projections. Caches do the same for client systems in distributed systems; they enhance the performance of clients by giving fast access to data or functions of servers. As with network bandwidth, interaction between user and system can be enhanced.

On one hand we define the quality of the solution, e.g. its expected performance, our confidence in the solution (security, reliability, usability), and its adaptability to change (maintainability, portability etc.). On the other we are constrained by the computational and network resources, organizational (including cost to produce the system), regulatory, physical and social (including ethical) contexts as well as the human resources to solve the problem as participants in the network and their cognitive capabilities.

5 Different approaches – an example from air traffic control

This section describes two different approaches of people addressing a problem of integrating the user interfaces of three different systems of an air traffic control workstation. The systems that should be integrated are electronic flight strips, radar and communication system. Although I talk about software engineers and HCI engineers I should warn that the other might not exclusively in either group but somewhat influence people.

When software engineers design a system, they will first try to describe the problem or the tasks by visiting its most abstract form, almost naively. Then as more knowledge is gained about the problem and the interfacing environment, they will refine the description with more detail, using generalization and aggregation hierarchies. This is the approach we used when given the tasks of integrating different systems, like radar, communication and electronic strips in the air traffic controller's workstation. Software engineers would then take the description and verify it by asking the users or observing them at work. We described entities, behaviour, relationships between entities, communication between behaviour, events, actions as responses to events, users and other stakeholders and the environment. SE described these as text or diagrams. The definition was done in several stages, with the most abstract one first, followed by a refined one including the technologies that are used for air traffic control. The problem with this approach is that we users are not very good at interpreting

abstract knowledge without seeing any details. Yet, it may be the only way to manage complexity.
People with HCI background took a different approach. They went directly to site visits, prepared with questions that would answer what the constraining factors were. These are common approaches in the HCI area such as Contextual design (Beyer and Holzblatt 1998) and Cognitive Work Analysis (Vicente 1999). The context was primarily viewed by understanding how controller gained awareness of the situation (perceive, gather, project – (Endsley 1995)) followed by a plan of solution and an action (from the Norman Action cycle (Norman 1986)). The approach also included drawing the flow of data or the relationship between system components. This approach is more bottom up, and tends to look at details and then to classify and analyse the work.
Prototypes, and scenarios are also detailed forms that give users a concrete model to look at and judge. Reviewing methods such as heuristics analysis looks also at details.
A secondary goal of this study is to understand how we can benefit from looking at a problem from different sides. Our next task will be to see whether these two approaches, i.e. their models cannot be bridged with two interfaces. We should not necessarily try to find a common denominator but build interfaces that will allow the two research areas to cooperate.
Psychologists are looking at detail, but software engineers at abstraction. There doesn't always seem to be so much a difference between SE and HCI, but when an SE looks at a problem, he tries to abstract in the beginning, classify entities and define abstract interfaces to entities. Then as the development proceeds refinement occurs that adds more detail and implementation. Anthropologists and psychologists prefer to look at details, and learn how humans interact with systems or solve problems. This tends to cause conflicts between the two disciplines. (Bentley et. al 1992) Similar discussion (Diaper 2002) has taken place when debating how appropriate scenarios are for user interface design. Whereas scenarios may seem to be detailed and concrete, other task models are often viewed as more abstract. The conflict rises because we want to show users concrete examples because we don't trust them to be able to think abstractly about problems. On the other hand we want to build abstract task models that free us from deciding on the interaction design. We also want to use more abstract models, and try to stimulate ideas of future use. In our work when designing interfaces for a workstation that has new and heterogeneous technological facilities, we have found that abstract models that hide e.g. technical and organizational detail become durable.
To understand one another we need to understand what is the different in models originating from the HCI and SE areas. There are user interface models, cognitive models and system models. Is there a difference in what they show e.g. functionality or non-functionality attributes such as performance and fault tolerance and how, e.g. abstraction.

6 Learning opportunities

The two areas have used different approaches to development but they have also influenced one another. One can see that action research originated from social sciences, soft systems methodology and participative design has had influence on software life cycle models. Software engineering has been rather late at realizing the close cooperation with clients. It is only now with XP (Beck 1995) and Evolve that close cooperation with the customer is being formalized into a framework. Iterative approaches are gaining popularity. Empirical research studies and benchmarks have been performed in the area of HCI and software engineering needs to make intelligent decisions based on evidence. Claims analysis has been used in scenario based design for user interfaces (Rosson and Carroll et al 2001) and scenarios are also used in architecture trade off assessments (Clements, Kazman, Klein 1995).
There are certainly many areas where the two disciplines can learn from one another. HCI can look at how software engineers have tested systems and built architectures. Software engineers can listen better to stakeholders, not just for functional requirements but also for technical, social and economical constraints under which a system is built, maintained and operated. Not only at the onset but also throughout the lifetime of the system.

7 Building interfaces between disciplines

Our case study has convinced us that we need not find a common denominator but build interfaces between disciplines that will allow the two research areas to cooperate. It may also be of benefit to try to make the distinction between the two disciplines clearer, e.g. so that both are not trying to specify requirements. As we have seen in papers of previous

HCI-SE workshops (Willshire, 2003 and Clemmensen and Nörberg, 2003), it becomes very difficult to merge two fields in a curriculum because the combined course will suffer from lack of time. Instead, we should continue to build strong disciplines of HCI and Software Engineering and they can learn many things from each other but also from other disciplines and apply it to their own areas but alas to a different problem. We also need to build good interfaces between HCI and Software Engineering so that they can exchange views. Not only should the interfaces be built so that they can divide the work but also so that there can be multi-faceted development, each taking their own road to their destination and hopefully benefiting from the multiple facets, either by extending the understanding or via the choice of one facet over the other.

To motivate the discussion on what these interfaces are, I give several examples such as Abstract and Concrete, Intuitive and Reasoning, Aesthetic vs. Heuristics, Human and System, Empirical and Action research, Models vs. Prototypes. Although these are pairs to show that you can apply both techniques, these are not set forward to put one of the items in the pair into the SE category and HCI in another. The objective is simply to show that one can benefit from developing multiple facets.

8 Conclusion

In this paper we have discussed the two disciplines HCI and SE. We have shown their relationship with one another and pointed out their unified goals in quality. The need for considering the changing role of human in future system architectures was also discussed. We presented a brief description of two different approaches to building an air traffic control workstation. The comparison of the two approaches will be a part of future work. The case study is described here to motivate more such multi-faceted development that hopefully will have meaningful common interfaces on the facets' boundaries.

References

Beck, Kent, *extreme Programming Explained,* Addison Wesley, 1995

Bentley, Hughes, Randall, Rodden, Sawyer, Shapior, Sommerville (1992), Ethnographically-inofrmed systems design for air traffic control, *Proceedings CSCW 1992*

Beyer, Holtzblatt, (1998) *Contextual Design,* Morgan Kaufman

Brian-Kinns, N & Hamilton, F. (2002) One for all and all for one? Case studies of using prototypes in commercial projects. *NordiCHI: Proceedings of the Second Nordic Conference on Human-Computer Interaction, October 19-23, 2002, Aarhus, Denmark,* p. 91-100

Clements, Kazman, Klein (1995), *Evaluating Software Architectures,* Addison Wesley,

Clemmensen, Nörberg (2003), Separation in Theory – Coordination in Practice, *SE-HCI Workshop at ICSE, 2003* http://www.se-hci.org/

Corritore, Cynthia L. Beverly Kracher, Susan Wiedenbeck, On-line trust: concepts, evolving themes, a model (2003), *International Journal of Human-Computer Studies* 58 737-758

Diaper (2002) Task scenarios and thought *Interacting with computers,* 14 (2002) 629-638

Endsley, Mica R. (1995). Toward a theory of situation awareness in dynamic systems. *Human Factors*, 37(1), 32-64.

Hewett, Baecker, Card, Carey, Gasen, Mantei, Perlman, Strong and Verplank (1996) ACM SIGCHI

IEEE STD 610.12-1990 (1990) *IEEE Standard Glossary of Software Engineering Terminology*

ISO 13407:1999 (1999) Human-centred design processes for interactive systems

ISO 9126-1 (2001) Software engineering – Product quality – Part 1: Quality model

Jackson, M. (1995) *Software Requirements & Specification,* Addison Wesley, ACM Press

Norman D A, (1986) Cognitive engineering. *In User centred system design, D A Norman and S W Draper Eds* Erlbaum Hillside, NJ 31-62

Rosson, Carroll et al (2001) Scenario-Based Development of Human Computer Interaction, Morgan Kaufman

Shneiderman, B., A. Rose (1996), "Social Impact Statements: Engaging Public Participation in Information Technology Design," in C. Huff, ed., *Computers and the Quality of Life: The Proceedings of the Symposium on* Computers and the Quality of Life, ACM Press, New York,1996, pp. 90-96.

Vicente, Kim J. (1999) *Cognitive Work Analysis*, Lawrence Erlbaum associates,

Willshire, M. J. (2003) Where SE and HCI Meet: A Position Paper, *SE-HCI Workshop at ICSE, 2003* http://www.se-hci.org/

Relating Human-Computer Interaction and Software Engineering Concerns: Towards Extending UML Through an Interaction Modeling Language

Maíra Greco de Paula, Simone D.J. Barbosa, Carlos José P. de Lucena

Departamento de Informática, PUC-Rio
R. Marquês de São Vicente, 225 – Gávea
Rio de Janeiro – RJ – Brasil – 22453-900

{mgreco, simone, lucena}@inf.puc-rio.br

Abstract: The UML suite of modeling languages fails to properly model the human-computer interaction. On the other hand, newly conceived HCI modeling languages need to foresee their role as members of the family of languages that constitute the UML representations for software design, due to their wide acceptance by both researchers and practitioners. MoLIC, our proposed HCI modeling language, seems to be a natural family member since many consistency checks seem to be possible between MoLIC and other software design notations used in UML. MoLIC is based on Semiotic Engineering and represents interaction as threads of conversation users may have with the system.
Keywords: interaction modeling language, UML, HCI and software design, HCI-SE integration

1 Introduction

Almost 20 years after Brooks' seminal "No Silver Bullet" (Brooks, 1986), we still face the challenges of traversing the gap between essence and accident in software construction, i.e. between "the mental crafting of the conceptual construct" and the implementation process.

Many methods, models and tools have been proposed since then to try and face this challenge. Currently, we find object-oriented design with the Unified Modeling Language (UML, 2003) amongst the most widely discussed in the academia and put in practical use in the software industry. In the realm of software engineering (SE), this approach has been working fine for most purposes. When we take into account human-computer interaction (HCI), however, we see that it does not provide designers with modeling languages and associated tools to explore the relevant HCI aspects of interactive software. Modeling languages that express the HCI concerns and that can be integrated with other notations that address software engineering concerns are necessary to bridge both understanding and communication gaps between the areas of SE and HCI.

This paper proposes to use an interaction modeling language called MoLIC (Paula, 2003) as a tool to help bridge this gap. We illustrate how an interaction model can help define the system behavior from the user's point of view, or the "architecture of a system" as defined in (Brooks, 1975). We do not claim it is ready to be fully integrated to other SE notations (e.g. the UML suite of languages), but instead that it may be considered as a first step towards HCI-SE integration.

2 OO Design with UML

In software development, designers use models that help them understand, organize and represent the system's architecture and functionality. In object-oriented development, the UML provides us with a large set of models to represent complementary aspects of the system. Use cases, class diagrams, sequence diagrams and activity diagrams are

perhaps the most widely used UML models in the initial design stages.

Figure 1 summarizes the relationship between these models. Although UML is process independent, the following steps are usually carried out. Use cases are created to represent a sequence of system actions to produce an observable result, relevant to at least one of the actors. They describe what the system does, and not how it does it.

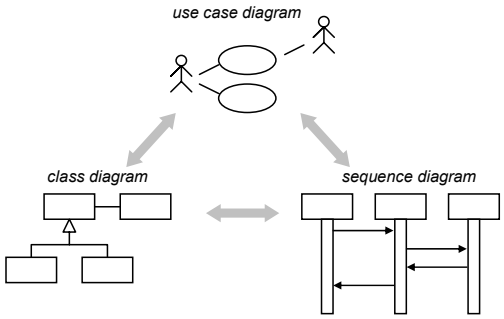


Figure 1: Typical UML models used in early stages of software design.

From the set of use cases, several features of class diagrams are created, which represent the system's static structural model: the system classes, their attributes and methods, and the relationships between the classes.

To each use case, a sequence diagram is associated. It represents the possible interactions between instances, the messages that can be exchanged between them, and in which sequence (in time).

Naturally, as with any design process, the construction of each model itself provides additional information that may cause refinements to the other models, hence the bidirectional arrows in the figure. From an HCI perspective, one of the major drawbacks of UML is that most of its models concern the system only, leaving most decisions about the user interface to the later stages of development, or even to the implementation phase.

In UML, use cases do concern users, but only in a fragmented way, in which each user goal is dealt with in isolation. As a result, designers lose the global view of the application as it will be perceived by the users. This makes it more difficult for HCI designers to envisage how the interaction will take place, in order to plan for and evaluate the quality of use of the final product. Thus, we claim that it is not enough to have use cases as a basis for the construction of the other diagrams from the user`s perspective.

Moreover, the basic UML diagrams do not differentiate between what is internal to the system and what will surface at the user interface. If we examine the aforementioned diagrams with respect to the elements that are relevant to HCI designers, we find, in a use case diagram: the actors which represent classes or roles of users and the use cases directly related to them, as representing the users' goals that may be achieved by interacting with the application. In a class diagram, we cannot distinguish which attributes or methods will have a (direct or indirect) representation at the user interface, and which of them are internal to the system. In a sequence diagram, we may have instances that correspond to actors, but the interaction is dispersed between each actor and a number of different instances.

From the user's point of view, the user interface is a single unified artifact. In order to design the user interface, HCI designers adopt decomposition strategies that are different from those adopted in SE, deriving task models and storyboards, for instance. Before we start each decomposition, however, we need to carefully craft the solution at a conceptual level. At this level, we need representations that provide a global view of the application. In SE, we might accomplish this by representing the system architecture. We claim that, in HCI, we need a modeling language that allows designers to build a blueprint of the application that will reveal its apparent (from a user's perspective) behavior. Such a blueprint could then be used as a reference point for global design decisions, and would be an additional resource for deriving both HCI and SE models. As such, this blueprint could act as a target for the application design to aim at. Figure 2 illustrates the relations between the UML models and this “blueprint”:

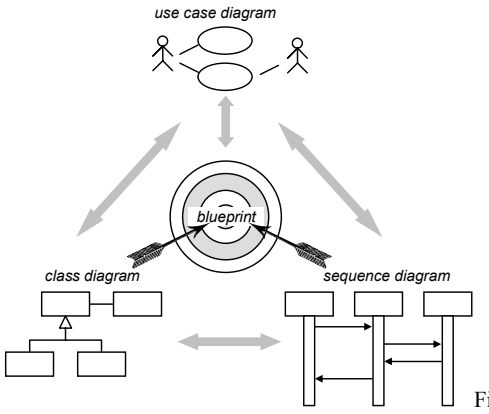


Figure 2: Proposal for an interaction blueprint as a reference point to UML models.

In the next section, we will describe an interaction model proposed in HCI to act as this application-as-the-user-will-experience-it blueprint. We represent in it how the user will be able to interact with the system, different interaction paths to accomplish his/her goals, and also how interaction breakdowns will be handled.

3 Interaction Modeling with MoLIC

HCI researchers and practitioners have proposed a number of methods and models to deal with the complexity of interactive systems design. Task models, user interface specifications and storyboards are the most widely used (see, for instance, (Paternò, 2000) and (Hix & Hartson, 1993)). However, none of these models gives designers a global view of the apparent behavior of the system, including alternative and breakdown interaction paths.

Paula proposed an interaction model named MoLIC, which stands for "Modeling Language for Interaction as Conversation" (Paula, 2003). Based on Semiotic Engineering (de Souza, 1993; de Souza, forthcoming), MoLIC represents interaction as threads of conversation users may have with the system[1], without yet specifying in detail the storyboards or the user interface itself. Each thread of conversation represents courses of action to accomplish a certain goal or to take remedial action when a communicative breakdown occurs. MoLIC was devised to explicitly represent all possible interactions, allowing designers to inspect the model for inconsistencies.

A MoLIC diagram may be seen as a graphical view of the whole set of scenarios or use cases, from the user's perspective. In HCI design, we do not aim to substitute scenarios, but to organize the relevant information they contain. Although high-quality scenarios contain all the information represented in MoLIC, 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. Some of the difficulties are also due to frequent ambiguities found in natural language texts.

[1] In fact, the conversation takes place between the user and a part of the system's user interface called "designer's deputy", as defined in the Semiotic Engineering theory of HCI. The semiotic engineering concepts used on the theoretical foundation for MoLIC and the complete MoLIC notation may be found at (Paula, 2003).

Many HCI designers generate task models from the scenarios, and then proceed to the user interface specification. However, task models usually assume the "correct" way of achieving a goal, without much consideration to breakdowns that may occur during interaction.

Finally, there are often too many decisions to be made in moving from scenarios and/or task models to software specification. And, unfortunately, these decisions are seldom recorded for future reference or verification of the final product.

3.1 MoLIC's Notation

The interaction model basically comprises scenes, system processes, and user/system's utterances — called transitions— that connect scenes and processes to form conversation threads. A scene represents a user–system 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/system utterances. 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 a window or dialog box, or a page in HTML.

In MoLIC's 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.

A system process is represented by a black rectangle, representing something users do not perceive directly. By doing so, we encouraged the careful representation of the system's utterances about the result of system processing, as the only means to inform users about what occurs during interaction.

Transitions represent changes in the conversation topic or focus. 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 system (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:`. Also, if there is a postcondition, it should be marked by the `post:` keyword after the

label. A postcondition is typically represented when there is a change in the application state that affects interaction.

A user goal is represented by a gray elliptical shape behind the thread of scenes and system processes with which the user will need to interact to accomplish it. User goals are extracted from scenarios (Carroll, 1995; Carroll, 2000) or use cases, and provide some level of traceability between MoLIC and them.

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".

3.2 Interaction breakdowns in MoLIC

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 illustrates a MoLIC diagram for a banking application which achieves the following goals: check balance, check account statement, and transfer between accounts.

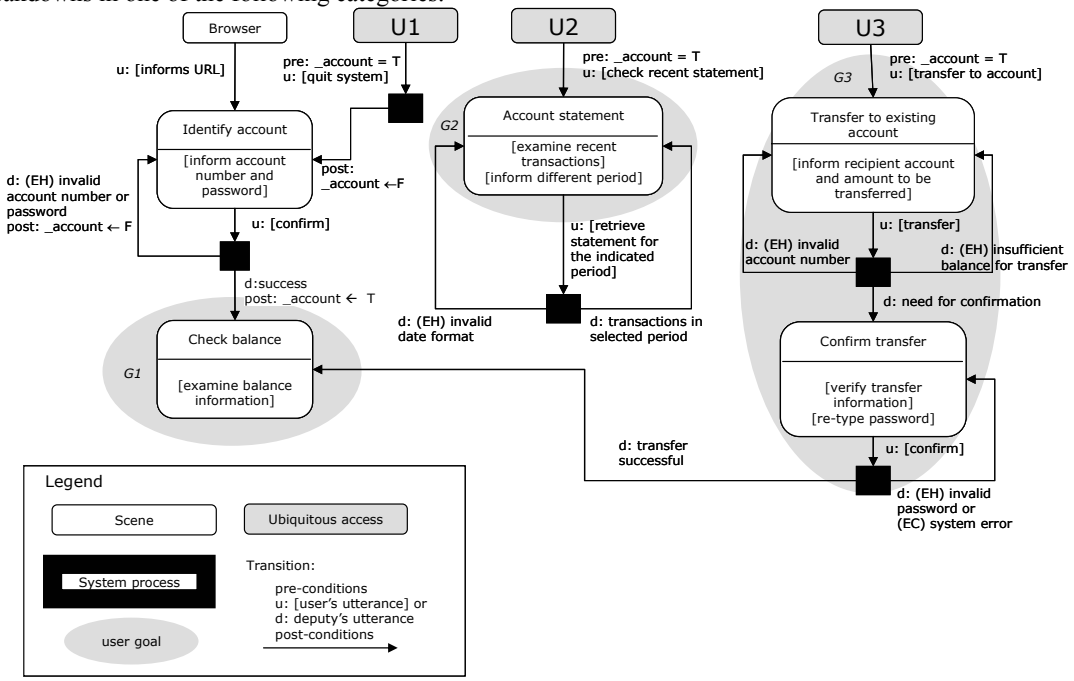


Figure 3: An abbreviated MoLIC diagram.

MoLIC has both an abbreviated and an extended diagrammatic representation. The extended MoLIC diagram represents not only dialogues, but also what can be talked about in each dialogue. This is represented by the signs at the user interface that either the user or the system manipulates at each moment. Here, we use the term *sign* to denote any given element at the user interface to which a user may attribute meaning with respect to his/her goal or task, or to the application itself.

When a sign is uttered by the system, 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 (`account number?`, `password?`).

Besides the role responsible for manipulating a sign (user or system), the extended representation may also include additional information about each sign, such as: whether the user must provide a value for the sign, whether the system provides a default value for the sign (and how this default value is calculated), the abstract user interface widget associated to the sign (simple choice, free text, etc.), the degree of knowledge the user has about the sign, additional information to be conveyed to users to help them understand and/or manipulate the sign, and any additional annotations designers may want to represent. Figure 4 shows the extended representation for the "Identify account" scene.

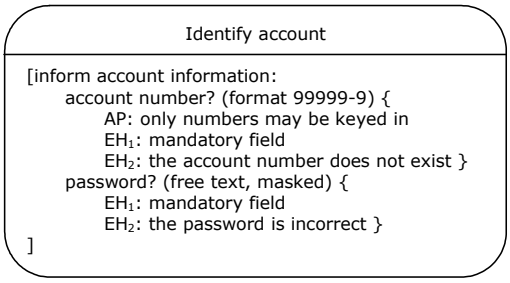


Figure 4: Extended representation of a scene.

3.3 MoLIC's Contributions to Design

As seen in section 2, we needed a language to represent the apparent behavior of a system as a whole, and from the user's point of view. It is necessary to identify the user–system turn-taking during interaction, but the system should be represented as a single entity, instead of already decomposed in internal software components. This way, HCI designers will be able to anticipate interaction problems or inconsistencies, and make appropriate design decisions early on.

We believe MoLIC contributes to HCI design by overcoming some limitations of existing HCI models. In particular, MoLIC motivates the uncovering of interaction problems or breakdowns, allowing designers to reflect upon the remedial actions to be taken and additional signs to present to users in order to help them recover from challenging situations.

In the next section, we will illustrate the use of MoLIC together with UML diagrams, as an important resource for software design.

4 Interaction Modeling within Software Engineering

We believe MoLIC can act as the blueprint illustrated in Figure 2, i.e., as a reference point to other design models (Barbosa & Paula, 2003). As such, MoLIC would be responsible for representing the "essence" of the application, i.e., the conceptual solution, from the user's point of view. This kind of approach has been proposed a long time ago by Frederick Brooks, when he stated that "the separation of architectural effort from implementation is a very powerful way of getting conceptual integrity on very large projects", and "by the *architecture* of a system, I mean the complete and detailed specification of the user interface" (Brooks, 1975).

From an HCI perspective, MoLIC is an important resource for building the storyboards or rigorously specifying the user interface. The simplest way to proceed would be to map each scene to a screen or window in a storyboard, each sign to a widget, and each user-initiated transition to a button or menu item (for activating an operation, usually between a scene and a system process), or even to a hyperlink (for navigation only, usually between two scenes). More elaborate constructions can, of course, be derived. For instance, one scene with multiple incoming transitions may be mapped to multiple variations of a single storyboard. Each variation would provide information about the alternative course of action taken to reach the scene, such as the goal the user is trying to achieve or the error he/she should correct.

But how can MoLIC be integrated to or collaborate with the aforementioned UML models? A MoLIC model may be built from scenarios or use case descriptions. In building the MoLIC model, the

designers will be motivated to find and correct problems in these information sources, such as inconsistencies and incompleteness. The advantage of using MoLIC here is that no system decomposition needs to be made or revised before this step, and thus the cost of changes is not very high. In Figure 3, we may substitute the ellipses representing user goals by a reference to the use cases that are directly related to the user-actors in a use case diagram.

With respect to the construction of the class and sequence diagrams, we may cite additional points where MoLIC may contribute:

- mapping from MoLIC signs to class attributes or methods: most of the signs present at the user interface have a counterpart in the data model. In a class diagram, these signs are usually mapped onto a class attribute, such as a person's name or birthdate. The value of a few signs is not directly stored, but calculated from one or more pieces of data. These may be mapped onto class methods, such as a person's age or whether he/she may obtain a driver's license (calculated based on his/her birthdate).
- mapping from MoLIC transitions to messages in sequence diagrams: for every sequence diagram in which one of the instances represent a user role, the messages incoming or outgoing from this instance may be retrieved from or verified against MoLIC's transitions.

Figure 5 illustrates these mappings.

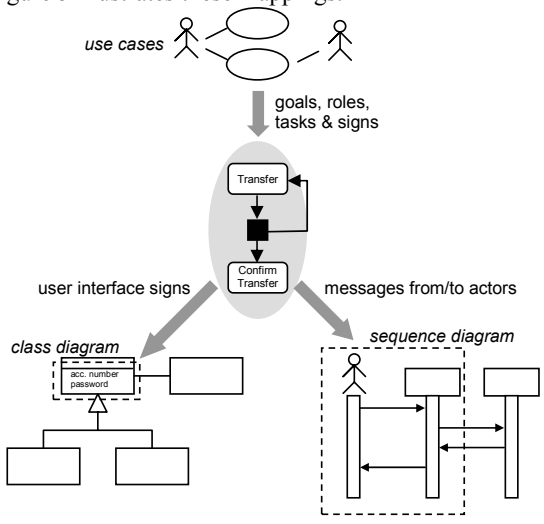


Figure 5: Sample mappings from use cases to MoLIC, and from MoLIC to UML class and sequence diagrams.

The UML suite of modeling languages (a de facto standard) fails to properly model the human-computer interaction. On the other hand, newly conceived HCI modeling languages need to foresee their role as a member of the family of languages that constitute the UML representations for software design. MoLIC appears to be a natural family member since many consistency checks seem to be possible between our proposed HCI modeling language and other software design notations used in UML. Incorporating MoLIC to the UML family of modeling languages will probably require changes in several existing UML diagrams to allow for its proper integration. Investigating process models based on an extended version of UML that encompasses HCI modeling appears to be a promissing research area.

5 Final considerations

We have briefly illustrated an initial attempt to use an HCI design model together with SE design models. Our goal was to provide a common reference point for both HCI designers and software engineers upon which to base their solutions.

There have been some attempts to incorporate user interface concerns into UML diagrams, such as UMLi (Silva, 2002). The main difference between our approach and UMLi is that we adopt a specific theory of HCI —called Semiotic Engineering— which provides us with an ontology for describing and evaluating relevant HCI phenomena, always keeping the focus on the quality of use of the proposed solution.

Other approaches, UMLi included, typically follow a bottom-up approach, whose main resource is a collection of case studies and user interface aspects present in user interface development environments. These approaches adopt a system-centered decomposition of user interface representations, losing sight of important HCI concerns. As such, they do not adequately support the designer's reflection about the intended solution at a conceptual level, and that a proper representation of HCI at this level is necessary for bridging the gaps between HCI and SE.

We are currently investigating what kind of design decisions are facilitated by our approach, and what are the impacts (good and bad) in the HCI and SE design processes. We need to assess whether this approach effectively promotes collaboration between SE and HCI and, if so, how this collaboration may be further propagated to other models, both in HCI and in SE.

Acknowledgements

Simone Diniz Junqueira Barbosa and Carlos José Pereira de Lucena thank CNPq for providing financial support to their work.

References

Barbosa, S.D.J.; Paula, M.G. (2003), "Interaction Modelling as a Binding Thread in the Software Development Process". Workshop *Bridging the Gaps Between Software Engineering and Human-Computer Interaction*, at ICSE 2003. Oregon, USA. May, 2003. Available online at http://www.se-hci.org/bridging/icse/proceedings.html.

Brooks, F. P. (1975), *The Mythical Man-Month*. Addison-Wesley.

Brooks, F. P. (1986), No Silver Bullet, *Proceedings of the IFIP Tenth World Computing Conference*, pp.1069–76.

Carroll, J. M. (ed) (1995). *Scenario-based design: envisioning work and technology in system development*, New York, Wiley.

Carroll, J. M. (ed) (2000) *Making use: Scenario-Based Design of Human-Computer Interactions*. The MIT Press. Cambridge, MA.

de Souza, C.S. (1993) The Semiotic Engineering of User Interface Languages. *International Journal of Man-Machine Studies*. No. **39**. pp. 753-773. 1993.

de Souza, C.S. (forthcoming) The Semiotic Engineering of Human-Computer Interaction.

Hix, D. and Hartson, H. (1993) Developing User Interfaces: Ensuring Usability Through Product and Process. John Wiley and Sons.

Paternò, F. (2000) *Model-Based Design and Evaluation of Interactive Applications*, London, Springer-Verlag.

Paula, M.G. (2003), *Designing the Human-Computer Interaction Based on Semiotic Engineering Models: Building an Interaction Model* (in Portuguese). Master dissertation. Informatics Department, Pontifícia Universidade Católica do Rio de Janeiro.

Silva, P.P. (2002) *Object Modelling of Interactive Systems: The UMLi Approach*. PhD's thesis, Department of Computer Science, University of Manchester, United Kingdom.

UML (2003) *OMG Unified Modeling Language Specification, v. 1.5*. March, 2003. Available for download at http://www.uml.org

Bridging the HCI-SE Gap: Historical and Epistemological Perspectives

Effie Lai-Chong Law

Computer Engineering and Networks Laboratory, Eidgenössische Technische Hochschule Zürich, Gloriastr. 35, CH-8092 Zürich, Switzerland

law@tik.ee.ethz.ch

Abstract: Assuming that the gap between HCI and SE can be better understood through exploring their historical developments, we studied several related literature available. The insights thus gained are: thesis-antithesis-synthesis approach can be used to clarify fundamental concepts by inviting discussions between different parties; ongoing within-discipline evaluation should be based upon cross-discipline observations; balanced views about humans and technologies are essential for understanding their interactions. The fact that knowledge as dynamically-constructed personalized entity accounts for infeasibility for closing the gap between HCI and SE. The conjecture that knowledge domain can be served as a tool to sustain the control and interest of the privileged people leads us to consider that a mesh of political, social and economic issues is lurking behind the persistent separation of disciplines. Furthermore, user-inspired basic research model is regarded as ideal as it can meet dual goals of understanding and use. Finally, standards, as anchor points for collaborative activities, should be rendered more consistent and comprehensible.

Keywords: historical development, research models, knowledge, situated cognition, standards

1 Introduction

Recent interests in investigating the gap between Human Computer Interaction (HCI) and Software Engineering (SE) have invited the attention and efforts of researchers and practitioners of both disciplines to work towards resolution of different related problems (see e.g., INTERACT 2001 Workshop; ICSE'03 Workshop). Among them, problems that are commonly addressed include usability analysis and assessment techniques, scenario- and pattern-based design, curricular reforms, and user-system interaction modelling. However, limited description or discussion is dedicated to the histories of the two disciplines. Indeed, the literature in this regard is relatively meagre (e.g., Brennecke & Keil-Slawik, 1996; Glass, 1997; John & Bass, 2001; Myers, 1998). Similarly, discussion about the possible role of standards in remedying the gap between the two disciplines is scanty. In fact, several standards (IEEE-1601, ISO/IEC-9126) have been developed to define the key concept "usability", which essentially hinges the two communities of practice. However, these standards seem inconsistent. Consequently, those who attempt to link HCI with SE by anchoring them in some common standardized definitions may probably be disappointed.

Traditionally, the two disciplines were ascribed certain distinct or even antagonistic features (Table 1), while they definitely share some commonalities (Willshire, 2003). How can the distinct features be traced back to the historical developments of both disciplines?

In fact, their distinctions have faded with time when the professionals of the two disciplines initiated the dialogues, reciprocally recognized the significance of each other and even adopted the approaches developed by each other (Constantine, Biddle & Noble, 2003). Specifically, SE has put on

more humanistic flavour by taking users' needs into serious consideration (e.g., user-centred design) whereas HCI has taken up more engineering flavour by looking deeper into software development and software analysis issues. For instance, the application of design patterns (Gamma et al., 1995) that has been widespread in SE community during the last five years is currently taking ground in HCI. Another example is the optimisation of software architecture for enhancing usability of user interface (John & Bass, 2001).

Attribute	SE	HCI
Human resources	Predominantly computer scientists	Interdisciplinary professionals
Focused tasks	Coding	Evaluation
Affiliated disciplines	Mathematics, Engineering	Social Sciences
Research Paradigm	Pragmatic: Practical → Theoretical → Practical	Empirical: Theoretical → Practical → Theoretical
Major Units of analysis	Technological	Social, psychological

Table 1: Traditional distinct features of SE and HCI

When studying their histories, a basic question to be explored is: Were both disciplines conceived from the same or different inspirational sources? In fact, SE and HCI have a common ancestor – computing! SE, according to Glass (1997), was conceived in the mid 1950s, whereas its cousin HCI, according to Myers (1998), emerged in the early 1960s. However, they were rooted in different concepts and visions with SE aiming to overcome the so-called "software crisis" and HCI aspiring to bring users' needs to the forefront (i.e., a central rather than peripheral role) in the enterprise of software systems design.

Since their inceptions, the two disciplines have been nurtured by academic and industrial professionals of various backgrounds, who deployed different research methodologies, tools and techniques to examine questions of interest. Consequently, they grew into varied traditions and styles. Interestingly, in their respective histories, both disciplines encountered the same problems of experiencing "identity crisis" and gaining recognition in the overarching field "Computer Science". In particular, the status of HCI is even more ambiguous. After years of struggles, HCI has somewhat earned its autonomy as an independent institute or unit in higher education institutions. However, it can be found under Faculty of Computer Science, Faculty of Social Sciences or Faculty of Business Studies.

In fact, the relationship between SE and HCI has been evolving during the last 20 years. Both disciplines endeavour to create useful, usable, and high-quality software systems by adopting and adapting methods and theories inputted from different avenues. Nonetheless, the two disciplines have progressed too independently (John & Bass, 2001). While such independence might be inevitable or even desirable at the early stage of their formation, now most of the professionals from both disciplines see the urgency to re-examine their inherent interdependence, considering the ever-increasing complexity of software artefacts that necessitate the expertise of both disciplines. While there may be many possible ways to improve the relationship between SE and HCI, it is crucial that the professionals from both disciplines are enabled to communicate their ideas effectively and efficiently. We propose that the key is to unify their languages by developing certain standards, which should define core common terms consistently and comprehensibly.

2 Synopsis of the Review Studies

2.1 Historian-Software Engineer Seminar

The Dagstuhl seminar report (Brennecke & Keil-Slawik, 1996) documents the views of two groups of scholars - historians and software engineers, whom we normally do not associate, about the problems identified in SE. Echoing the presumption of the coordinators of the seminar, we are convinced that "In a highly dynamic scientific environment with ongoing debates about the fundamental issues, a historic investigation may help to better understand the issues …" (p.1).

Nonetheless, instead of settling down the provocative debate whether the so-called 'software crisis' has continued to exist, the seminar participants

tended to conclude that there was an 'identity crisis' in SE. Since its emergence more than 30 years ago, computer scientists are still investigating the history of other established branches of engineering to define what should be done to transform SE into a sound engineering discipline. A seminar participant even claimed that SE was then still at the pre-engineering phase (Baber, 1996). Actually, the term "Software Engineering" has the implication that software manufacture should be based on the types of theoretical foundations and practical disciplines that are established in the traditional branches of engineering. Given this assumption, SE might tend to shape itself in a way that it aligns with so-called 'hard sciences' such as most other engineering schools while distancing itself from so-called 'soft sciences' such as psychology, sociology, etc. Though no solid clear-cut conclusion could be drawn, the event registered many facets, issues and methods in SE that would be worth further exploration. It is expected that the current workshop may have similar or even more impacts than this specific seminar. Specifically, through open dialogues between SE and HCI professionals deep-seated ideas in both disciplines can be surfaced and tackled.

2.2 Evolving Views of Software Engineers

John and Bass's paper nicely summarizes the evolving role of software architecture in SE and HCI from the 1980s to the beginning of the new millennium and describes how it affected the relationship between the two intimately related disciplines. An interesting observation is how software engineers changed their views about the relation between usability and software architecture. "Software engineering in the 1990s focused on technical attributes … and process attributes ... Although usability is mentioned occasionally as a quality attribute, some authors explicitly declare that it is not related to [software] architecture …" (p. 332). Through the authors' meticulous analyses of general usability scenarios and their personal experience using them in SE processes, they have come to the recent conclusion: "… many facets of usability are indeed architecturally sensitive" (p. 335). Thanks to the collaboration undertaken between HCI and SE institutes in the same university where the authors resided, the earlier erroneous assumption could be amended. This case can clearly illustrate the importance of ongoing self-assessment within a community of practice while keeping open eyes to the development of the neighbouring disciplines.

Furthermore, the same authors criticise that, despite the call from Newell in the early 1980s that the action of HCI professionals should shift from evaluation to design, their role seems to be squarely in the arena of testing or evaluation, as shown by the publications in recent conferences. Nonetheless, we strongly support the recommendation of the authors that for HCI professionals, on top of their evaluation endeavour, *design is where the action is*, and that software engineers prioritise usability as a first-class quality attribute among the others.

2.3 HCI History on One Side

The main message Myers (1998) aimed to convey through his paper is that the advances of HCI have been contributed by both academic researchers in universities and their counterparts in corporate research labs. Anyway, looking up the item "History" in the HCI Bibliography website (http://hcibib.org/hci-sites/HISTORY.html, last modified on 03.07.2003), only 18 records are found and most of them, if not all, like Myers' paper, cover only the technical aspects of HCI. Some only report certain survey results (e.g., Nielsen, 2002). Nonetheless, Myers called for the documentation of the historical development of the other equally, if not more, important side of HCI – the social. Unfortunately, such literature seems still lacking.

As a matter of fact, studying the history of a discipline is an enormous project by itself, which should basically investigate the origin of the discipline (i.e., scientific, political, economic, and social reasons for its emergence), the facilitating and hindering factors for its development as well as sustainability (e.g., deployment of selected research methodologies, evaluation techniques, dissemination activities, professional training programs), and its respective evolving roles in university, industry, government at both the local and global levels. Presumably, with such historical data, the questions why and how two disciplines resemble and differ from each other can be systematically answered. Based on their respective past successes and failures as well as their current trends of the development, we

probably can envision a picture about the future relationship between the two disciplines - a co-existent partnership as the status quo or a merged enterprise.

3 Reflection on Nature of Knowledge and Research

3.1 Knowledge as Dynamic Entity

It is a natural tendency to explain the gap between SE and HCI with reference to software artefacts that entail contributions from both disciplines. However, perhaps the gap can be explicated from an alternative perspective – sociological views of knowledge.

Knowledge is conventionally defined as consisting of coherent islands whose boundaries and internal structure exist, putatively, independent of individuals (Lave, 1988). So conceived, "culture [and knowledge] is uniform with respect to individuals, except that they may have more or less of it" (ibid, p.43). This conventional epistemological view is problematic because knowledge, individuals and the societal context in which they are embedded are inherently inseparable (Law, 1994). Knowledge is interpreted and assimilated in an entirely individual and contextualized manner. Hence, to put it an extreme way, there would be as many interpretations of what SE or HCI should be as the number of professionals in the respective discipline. Consequently, it would be very difficult, if not impossible, to cohere a diversity of personal and idiosyncratic views.

Arguing along the same line, the concept of "knowledge domains" has several problematic implications. First, it leads to separate the study of problem solving from analysis of the situation in which it occurs. Moreover, the maintenance of strict boundaries among various academic disciplines by the professionals concerned can reflect their intended control of a body of knowledge. Hence, the seemingly unbridgeable gap between SE and HCI could be attributed to a cluster of highly intertwined political, economic and societal reasons. People of vested interests in a certain field may be reluctant to surrender their "territories". Moreover, research fund is a practical issue. Presumably, more funds will be granted to a discipline that has unique status, research goals and activities than to a discipline that shares a number of features with the others or even is subsumed by them.

3.2 Nature of Research

Can the nature of research conducted in HCI and SE be one of the possible reasons for the existing gap? It may simply be a chicken-and-egg problem! Traditionally, research was dichotomised into two categories, namely basic research with the goal of understanding and applied research with the goal of use. Furthermore, it was assumed that the knowledge acquired from the former could be transferred to the latter, somehow like a conduit transferring water from a higher to lower point. Note also basic research was regarded as more prestigious than applied research. Not until recently, criticisms have been charged against this one-dimensional view of knowledge transfer (Fischer et al., 2003). In its place, Stokes (1997) proposed two-dimensional model by conceptualising goals of understanding and goals of use as two distinct continua. Three different perspectives on the relationship between theory and practice arise from that assumption (Figure 1).

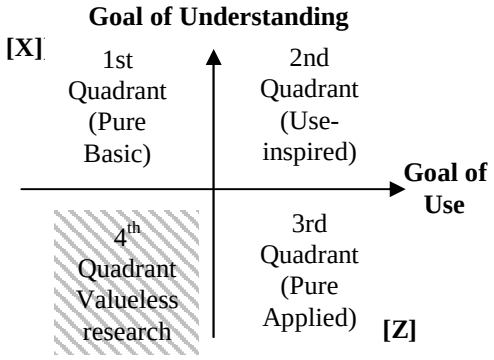


Figure 1: Research matrix
(Adapted from Stokes, 1997)

The Point [X] of the first quadrant represents pure basic research, which sets its focus exclusively on goals of understanding and has minimal concern on the application of knowledge to technological development and production. The Point [Z] of the third quadrant represents pure applied research, which only pursues goals of use and is not interested in theoretically understanding the phenomena under

investigation. The Point [Y] of the second quadrant represents *use-inspired basic* research, which combines goals of understanding and use in conducting basic research and aims to resolve important practical problems. Of these three kinds of research, use-inspired basic research has its special appeal in that it systematically links insight into learning processes and intervention for change (Fischer et al., 2003).

What kinds of research are predominant in SE and HCI? Interestingly, the three keynote papers authored by Dye, Rabardel and Cordier et al. in Proceedings of Human-Computer Interaction (HCI) 2001 could be classified as representing user-inspired basic research, basic research, and applied research, respectively. While each of the authors presents some significant achievements in HCI, Dye's paper perhaps is more appealing as it relates goals of use to goals of understanding and vice-versa. By scanning the remaining 33 papers in the same Proceedings, it is estimated that whilst most of them fall into the category "user-inspired basic" or "applied research", a few of them are in the category "basic research". When imposing the same scheme on the papers in Proceedings of International Conference for Software Engineering (ICSE) 2001, which are divided into Technical, Case Study and Educational sections, one tends to say that a large portion of all the 61 papers fall into the category "applied research", some into "user-inspired basic research", and a very small number into "basic research". Noteworthy is that the two dimensions – goals of use and goals of understanding - are continua in the way that some researches are not different in nature, but only in degree of the two attributes. Nonetheless, it is worrisome to observe that limited research efforts have been invested in exploring theoretical issues that lay the foundation of the discipline.

In analysing Stokes' framework, Fischer and his colleagues (2003) have drawn some implications that are relevant to our study of the gap between HCI and SE. As noted by these authors and corroborated by our reading of the two aforementioned Proceedings, we believe that methodological issues is a compelling concern for research communities (cf. Gray & Salzman, 1998). There are a plethora of methodologies deployed in HCI and SE, because every researcher tends to adapt selected methodologies to meet his or her specific needs. As standards on how to use different methodologies are lacking, there are no ways to evaluate their quality of use. Consequently, we propose that comparison of methodologies to highlight essential elements of the favoured paradigms used in HCI and SE should be the first step of working towards this standardization endeavour. Furthermore, we echo the proposition of Fischer et al. (2003) that it is necessary and useful to establish so-called "practical impact standards", which can provide figures of merit for the potential of some contribution to improve people's lives. To reify the dual goal of understanding and goal of use mentioned earlier, we should develop and apply standards for both scientific and practical impacts.

3.3 What are Standards for?

In the previous sub-section, we addressed the issue about the lack of standards for methodology usages and for assessing scientific and practical impacts of scholarly contributions. Nonetheless, a caveat needs to be made that developing standards is not a panacea. Nor does it guarantee that the gap between HCI and SE can be so bridged. Indeed, poor standards could be worse than no standards at all.

Take a look of the current software engineering standards, especially those related to usability, to name just a few: IEEE-1061 (Standard on Software Quality Metric Methodology), ISO/IEC-9126-1:2000 (Software Engineering –Product Quality-Quality Model), ISO 9241-11:1998 (Guidance on Usability), ISO 8402 (Quality Vocabulary), ISO 9001: 1994 (Quality Systems), ISO 9001:2000 (Quality Management Systems), etc. The concept of usability is defined differently in the original and revised versions of ISO/IEC 9126 and in ISO 9421-11. The recent attempt is to use the term "quality of use" (ISO/IEC 9126-1) to cover dual aspects of usability (Bevan 1999).

It is confusing, particularly for newcomers of the community of practice, to decide which standards to adhere, given their inconsistent terminologies, notations and languages. Eventually, they may give up following any of them. It may explain why although some software developers have some theoretical knowledge of user interface design guidelines or usability standards, they seldom use them in practice, simply because they do not know

which, when and how to apply them in their context. Despite these rather frustrating observations, we should not deny the values of standards that have actually contributed to the progress of both SE and HCI. However, what we need to do is to develop some implementation strategies through which software developers are enabled to "situate" or "localize" the principles described in standards to specific application contexts.

4 Summary & Conclusion

Based on the conviction that "the present is built upon the past", we posit that the existing gap between the two intimately related disciplines – Software Engineering and Human Computer Interaction - can be better understood through studying their historical developments, specifically their origins, their successes and failures, and their changing roles in various societal institutions. Unfortunately, the literature in this regard is scanty. Nonetheless, we were able to get some insights from the accessible references:

- The Dagstuhl seminar (Brennecke & Keil-Swalik, 1996) was successful to a certain extent to surface some issues deeply entrenched in SE by confronting software developers with the views from historians. We assume that direct dialogues between SE and HCI perhaps are not sufficient or provocative enough to uncover deeply seated problems. The presence of a third party like historians or scholars in philosophy of science may serve as observers, mediators or even provocateurs to bring the dialogues forward and enrich them. This will align with the so-called "thesis-antithesis-synthesis" approach (Hegel, 1998).
- The review on the relationship between usability and software architecture developed over the last three decades (John & Bass, 2001) showed us the significance of ongoing self-evaluation within one discipline. More important is that such evaluation activities are based on the understanding of the development of other related fields. Open-mindedness is indispensable for reflection that is in turn essential for making advancement in any discipline. Furthermore, the role of HCI professionals seems still overwhelmed by usability evaluation tasks. It is the high time for them to balance their works on both design and evaluation or even invest more efforts in the former.
- The one-sided review on the technological aspect of HCI (Myers, 1998) revealed the biases of some HCI professionals. On the contrary, HCI is distinct from SE by its human or social concerns. However, it seems that HCI has become hCI with (too) much emphasis being put on "C" rather than 'h". Nonetheless, neither hCI nor HcI is appropriate. What we should focus on is I – interaction as well as interface. Without understanding both H and C, we cannot say anything meaningful about I. It is challenging for HCI professionals and, however, it is exactly what makes HCI appealing.

Apart from the historical perspective, we believe that the gap between HCI and SE can be analysed from the epistemological perspectives. Challenged by the so-called "situated views" movement in the mid 1980s (Lave, 1988; Suchman, 1987), the traditional concepts of knowledge, knowledge domain and knowledge transfer have been challenged. Knowledge, no longer seen as a fixed commodity-like construct, is dynamically generated by learners. Consequently, each individual has idiosyncratic interpretations of the same subject matter. It may explain why the HCI-SE gap is so difficult to bridge, because it is almost impossible to coalesce a diversity of interpretations. To reach the highest possible common understanding or so-called "mutual intelligentibility" (Suchman, 1987), negotiation or open dialogues between learners or professionals is deemed indispensable.

The neo-Marxist view that knowledge domain serves as an artificial boundary to sustain the control and protect the interest of a privileged group may sound too radical. Nonetheless, it reveals the fact that sustaining knowledge domains or disciplines is not simply an academic manner, but rather it is meshed with political, economic and societal factors.

The conventional conduit metaphor for knowledge transfer from basic to applied research was proved to be inappropriate. As a matter of fact, current researches tend to achieve dual goals of understanding and use, though to a varying extent. However, one worrisome phenomenon noted is that researches in both HCI and SE tend to be too application-oriented while neglecting the theoretical aspect. In fact, the essential factor which accounts for so-called "design correctness" (i.e., all requisite functionalities present and can be operated with highest possible effectiveness, efficiency and satisfaction) is that there exists a solid foundation, be it a mathematical model of artefacts or a model of human behaviours, and that the practitioners apply this very foundation regularly and as a matter of course in their work.

Lastly, most, if not all, professionals in HCI and SE agree upon the importance of mutual communication so as to narrow, while impossible to close, the gap between the two disciplines. Standards can serve as good anchors point to focus the dialogues and collaborative activities. However, the existing standards are rather inconsistent and thus confusing. More efforts should be invested to render these tools more usable and useful. Specifically, it is worthy to develop implementation strategies for situating or localizing the standards so that they can be applied effectively in particular contexts.

Reference

Bade, R.L. (1996). Comparison of electrical 'engineering' of Heaviside's times and software 'engineering' of our times. In A. Brennecke & R. Keil-Slawik (Eds.), *Positions papers for Dagstuhl Seminar 9635 on History of software engineering* (pp. 4-9). Available on http://www.dagstuhl.de/9635/index.en.phtml.

Bevan, N. (1999). Quality in use: Meeting user needs for quality. *Journal of System and Software*. Available on http://www.usability.serco.com/papers/qiuse.pdf.

Brennecke, A., & Keil-Slawik, R. (Eds.). (1996). *Positions papers for Dagstuhl Seminar 9635 on History of software engineering*.

Constantine, L., Biddle, R., & Noble, J. (2003). Usage centred design and software engineering: Models for integration. A Position Paper in *ICSE'03 Workshop on Bridging the Gap between Software Engineering and Human-Computer Interaction*, Portland, Oregon, May 3-11 (pp. 106-113). Available on http://www.se-hci.org/bridging/icse/proceedings.html.

Cordier, F., Lee, W.S., Seo, H.W., & Magnenat-Thalmann, N. (2001). From 2D photos of yourself to virtual try-on dress on the web. In J. Vanderdonckt, P. Gray & Blanford, A. (Eds), *People and computers XV – Interaction without Frontiers* (Proceedings of HCI 2001) (pp. 31-46). Springer.

Dye, K. (2001). An easy way to use as a banking machine. In J. Vanderdonckt, P. Gray & Blanford, A. (Eds), *People and computers XV – Interaction without Frontiers* (Proceedings of HCI 2001) (pp. 3-16). Springer.

Fischer, F., Bouillon, L., Mandl, H., & Gomez, L. (2003). Scientific principles in Pasteur's quadrant: Integrating goals of understanding and use in learning environment research. In B.Wasson, S. Ludvigsen & U. Hoppe (Eds.), *Proceedings of Computer-Supported Collaborative Learning (CSCL) 2003* (pp. 493-502), 14-18 June, Bergen, Norway.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns, elements of object-oriented software*. Reading, MA: Addison-Wesley.

Glass, L. R. (1997). *In the beginning: Personal recollections of software pioneers*. Wiley-IEEE Press.

Gray, W.D., & Salzman, M.C. (1998). Damaged merchandise? A review of experiments that compare usability evaluation methods. Human-Computer Interaction, 13, 203-361.

Hegel (1998). Hegel's Science of Logic, tr. by A. V. Miller. Humanity.

John, B.E., & Bass, L. (2001). Usability and software architecture. *Behaviour and Information Technology, 20*(5), 329-338.

Lave, J. (1988). *Cognition in practice: Mind, mathematics, and culture in everyday life*. Cambridge: Cambridge University Press.

Law, L.-C. (1994). *Transfer of learning: Situated cognition perspectives* (Research Report No. 32). München: Ludwig-Maximilians-Universität, Lehrstuhl für Empirische Pädagogik und Pädagogische Psychologie.

Myers, B. A. (1998). A brief history of human-computer interaction technology. *Interactions* (March/April), 44-54.

Nielsen, J. (2002). *Top research laboratories in Human-Computer Interaction*. Available on http://www.useit.com/alertbox/20020331.html.

Stokes, D.E. (1997). *Pasteur's quadrant: Basic science and technological innovation*. Washington: Brookings Institution Press.

Rabardel, P. (2001). Instrument mediated activity in situations. In J. Vanderdonckt, P. Gray & Blanford, A. (Eds), *People and computers XV – Interaction without Frontiers* (Proceedings of HCI 2001) (pp. 17-30). Springer.

Suchman, L. (1987). *Plans and situated actions – The problem of human-computer communication.* Cambridge, MA: Cambridge University Press.

Willshire, M.J. (2003). Where SE and HCI meet. A position paper in *ICSE'03 Workshop on Bridging the Gap between Software Engineering and Human-Computer Interaction*, Portland, Oregon, May 3-11 (pp. 57-60). Available on http://www.se-hci.org/bridging/icse/proceedings.html.

Integrating formal approaches in Human-Computer Interaction methods and tools: an experience

Patrick Girard[1], Mickaël Baron[1], Francis Jambon[2]

[1] LISI/ENSMA, 1 rue Clément Ader, Téléport 2,
BP 40109, 86961 Futuroscope Cedex, France
{girard,baron}@ensma.fr

[2] CLIPS-IMAG, 385 rue de la bibliothèque,
BP 53, 38041 Grenoble cedex, France
Francis.Jambon@imag.fr

Abstract: Formal methods are increasingly used by HCI researchers. Nevertheless, their usage in actual interactive developments is not so common. In this paper, we describe the use of a specific formal method from two viewpoints. On the one hand, we demonstrate how it is possible to use a formal method on real development from specification to actual code. Doing so, we notice that HCI concepts, such as architecture models, may have to be adapted. On the other hand, we show how it is possible to bring more usability to formal methods by the way of a complete integration into HCI tools. We conclude in eliciting the lessons learned from these works.

Keywords: Formal methods, HCI tools, interactive development

1. Introduction

Software engineering (SE) methods are well known to be relatively far from Human Computer Interaction (HCI) methods and tools. From the HCI point of view, we can say that beyond use case models in UML, task-based approaches are not really in use in most projects. From the SE point of view, HCI tools and methods remain partial and somewhere "anecdotal".

In this contribution, we would like to introduce our experience in attempting to incorporate formal methods (issued from SE point of view) in interactive design and development. Doing so, we try to elicit the points that make all the difficulty of actually realizing our goals.

Our approach is based on the use of the formal B method in HCI, in order to address security in critical interactive applications, such as traffic air control or nuclear power plant supervision. We do not intend to focus on the use of this particular formal method. Adversely, we only use this experience to illustrate the gap between the two fields.

In the next part –part 2– we give a short list of formal approaches that have been used in HCI, and we give several points that explain their poor usage in that field. In part 3, we relate the first attempts in applying the B method in interactive design. We particularly focus on architectural problems, which might constitute a solid bridge between SE and HCI. In part 4, we show how actual HCI tools might incorporate secure development methods by the way of leaning on formal semantics all along the HCI design and development. Last we conclude on discussing the lessons learned in these works.

2. Formal approaches in HCI

Formal specification techniques become regularly used in the HCI area.

On the one hand, user-centred design leans on semi-formal but easy to use notations, such as MAD (Scapin and Pierret-Golbreich, 1990) and UAN (Hix and Hartson, 1993) for requirements or specifications, or GOMS (Card et al., 1983) for evaluation. These techniques have an ability to express relevant user interactions but they lack clear semantics. So, neither dependability nor usability properties can be formally proved.

On the other hand, adaptation of well-defined approaches, combined with interactive models, brings partial but positive results. They are, for example, the interactors and related approaches (Duke and Harrison, 1993, Paternò, 1994), model-oriented approaches

(Duke and Harrison, 1993), algebraic notations (Paternò and Faconti, 1992), Petri nets (Palanque, 1992), Temporal Logic (Brun, 1997, Abowd et al., 1995). Thanks to these techniques, some safety as well as usability requirements may be proved.

However, these formal techniques are used in a limited way in the development process, mainly because of three points:

Few of them can lean on usable tools, which allow real scaled developments. Case studies have been demonstrated, but no actual application has been completely designed with these methods.

Formal notations are currently out of the scope of HCI designers. Their usage by non specialists it everything but easy.

Formal studies are currently disconnected from usual HCI tools. No commercial tool and very few research one really incorporates semantically well defined approaches.

In this paper, we relate our studies one model-oriented approach, the B method (Abrial, 1996), whose one great advantage is to be well instrumented. But we do not allege it is the best nor the perfect formal method to be used. Our claim is that this model-oriented technique that uses proof obligations can be used with profit in a HCI context; more, it might be used together with model checking techniques, where automatic proofs of properties can be performed.

3. The B method and interactive design and development

This section presents the different steps that have been made in the attempt to use the B method in the field of HCI. Starting from the reduced aspect of verifying software specifications, we show how it has been possible to reach the implementation step in a complete formal development. Then, we focus on architecture problems. Last, we conclude in analyzing the difficulty of this extreme approach.

3.1. Using B for HCI specifications

In (Aït-Ameur et al., 1998a, Aït-Ameur et al., 1998b), the authors use for the first time the B method for the verification of interactive systems. Lying on a pure interactive case study (see below), these works suggest formally based solutions which allow solving difficulties that are inherent to interactive systems specification, such as reachability, observability or reliability.

The case study is a co-operative version of a Post-It®[i] Note software. With this case, it is possible to address highly interactive problems due to the direct manipulation style, such as drag and drop, Iconfication/Deiconification, resizing, and so on. A special attention is paid on mouse interaction.

This use of the B method on a non-trivial case study has illustrated the capability of B to handle different aspects of the software life cycle in the area of interactive systems. The described approach demonstrates:

Complete formalisation: the approach is completely formalised and most of the proof obligations are automatically proved. The other ones need only few steps of manual proof.

Property checking: it is possible to check properties on specifications, thanks to the weakening of preconditions.

Reverse engineering aspects can be handled with this approach and the specifications of already existing programs can be used to develop new ones. Therefore, reusability issues appear.

Incremental design: the specifications are incrementally. Indeed, programming in the large operator allows to compose abstract machines and therefore to build more complex specifications. Yet, this process needs to follow a given methodology issued from the area of interactive system design.

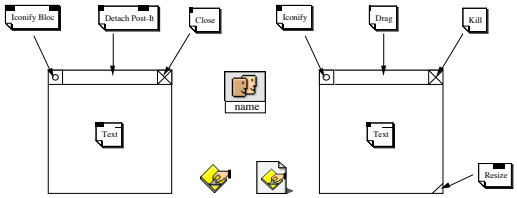


Figure 1: From the left to the right, The Post-It® block, the three icons (Post-It® block, User, and Post-It®), and the Post-It® Note itself.

One can object that this case study is situated at a too low level for the interactive viewpoint. Properties such as keeping the mouse pointer into the screen are not relevant in current graphical systems where this is ensured –or supposed to be ensured– by the operating system. In fact, this emphasizes the problem of using formal methods in actual interactive environments. Is it acceptable to use formal techniques when we lean on graphical layers that are not formally defined? One

[i] "Post-it" is a registered trademark of 3M

solution, as described in this work, might be to make a reengineering analysis of such tools.

The first step reached by this study is the one of a complete specification of an interactive system, with respect to some interactive properties. As many works in the field of formal methods in HCI, it is possible to concentrate on some properties, but two drawbacks can be given:

> Because of the strong relation to the coding activities, interactive properties are not related to the user activity;
>
> Formally ensuring that specification are consistent, and respect properties, does not ensure that the actual code will respect specification, without a link between implementation and specification.

One of our major goals in exploring the usage of formal methods in the context of HCI design and development was to ensure that other people than pure formal methods specialists could use the method. So, with help of B tools, we tried to realize the whole development of an interactive application, from high-level specifications to running code. We first propose a architecture model to assist the designer (3.2), and then define heuristics to implement this model (3.3).

3.2 Formal development versus software architecture models

The case study is here a control panel for a set of three rechargeable batteries. It is an elementary safety-critical process-control system: the operator can control side effects on hardware –the switches– whereas the hardware state –the batteries levels– is altered asynchronously. Both safety and usability properties have to be ensured. This required first step of the design process consists in modeling the battery control panel requirements with the B language. Three kinds of requirements must be fulfilled:

> The system must be safe, i.e., the system must avoid shortcuts and it must not be possible to switch on an empty battery.
>
> The system must be honest, i.e., the user interface widgets must display exactly the batteries levels and switches positions.
>
> The system must be insistent, i.e., the system must warn the operator when a battery is going to be empty.

Our first idea for designing such a system was to use a well-known multi-agent model, such as MVC (Goldberg, 1984) or PAC (Coutaz, 1987), because acceptability of formal methods is greatly influenced by using domain standard methods. The interactive system specifications must however stay in the boundaries of the B language constraints. We selected three kinds of constraints that relate to our purpose. These main constraints are:

> Modularity in the B language is obtained from the inclusion of abstract machine instances –via the INCLUDES clause– and, according to the language semantics, all these inclusions must build up a tree.
>
> The substitutions used in the operations of abstract machines are achieved in parallel. So, two substitutions –or operations– used in the same operation cannot rely on the side effects of each other.
>
> Interface with the external world, i.e. the user actions as well as the updates of system state, must be enclosed in the set of operations of a single abstract machine.

Classic software architecture models such as PAC or MVC are not compliant with these drastic B language constraints. That is why we proposed a new hybrid model from MVC and PAC to solve this problem. The design of this new software architecture model –CAV– cannot be detailed here. The reader should refer to (Jambon et al., 2001) for a more detailed description of the model design.

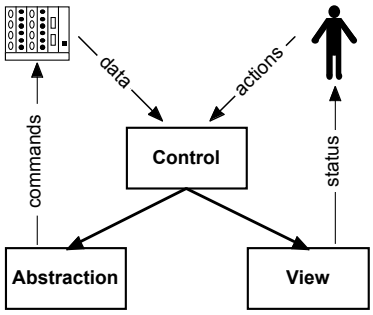


Figure 2: The three components of the Control-Abstraction-View software architecture model

The CAV model uses the external strategy of MVC: the outputs of the system are devoted to a specific abstract machine –the View– while inputs are concerned by another one –the Control– that also manages symmetrical inputs from the reactive system which is directed by the third abstract machine –the Abstraction. The Control machine synchronizes and activates both View and Abstraction machines in response to both user and reactive system events.

Among the usability properties, the system is in charge of warning the user if a battery is going to be empty. This usability requirement has to be specified as: if the battery switch is in position ON and the level is below or equal 10%, a warning message must

be shown. This is specified in the INVARIANT clause of the View. As a consequence, the operations of the View must be specified to fulfil this invariant whatever the way they are computed. This insistence property specification is restricted to the View abstract machine. So, it is fairly easy to handle. On the contrary, the Conformity property requires the Control mediation between Abstraction and View. Its specification is similar to the specification of safety below.

Among the safety requirements, we detail now the prevention of user error: the operator must not be able to switch on an empty battery. At first, this safety requirement deals with the functional core of the system, i.e., it must be specified in the Abstraction. Moreover, this requirement is not a static but a dynamic property: the battery can become empty while switched on, but an empty battery must not be switched on. This requirement is not static predicate, so, it cannot be specified in the invariant clause of the abstract machine. In the B language semantics, this category of requirement must be specified in a precondition substitution of operations.

In fact, we delegated to the Control abstract machine –that includes the Abstraction– this safety requirements, i.e. the Control is in charge of the verification of the semantic validity of the parameters when it calls the operation of the Abstraction abstract machine. We name this technique the delegation of safety. This generates two consequences: (1) The operator cannot be aware of the fact that a battery could not be switched on ; (2) An action on a pushbutton can be generated with a empty battery number as parameter, so some required proofs obligations cannot be proved.

The first consequence is easy to set up. We have to improve the interface layout and to update the state of the button: enabled or disabled. Of course, if a button is disabled, it is well known that this button cannot emit any action event. This assertion may seem to be sufficient to solve the second consequence above. That is not exact: the B semantics cannot ensure that a disabled button cannot emit events because the graphic toolkit is not formally specified. So, the Control abstract machine must filter the input events with the button states specified in the View abstract machine. This is required by the formal specification. The benefit of this consequence is that our system is safe whether the user interface is defective.

3.3. From formal specifications to implementation

The final program must be a set of software modules in which some of them are formally specified and implemented, and some others are developed with classic software engineering methods. In order to dissociate these two antagonist types of modules, interfaces have been inserted in between. So, at the implementation step, the CAV architecture supports some add-ons as shown on figure 3. We now focus on these three types of modules: secure code, interface and native modules.

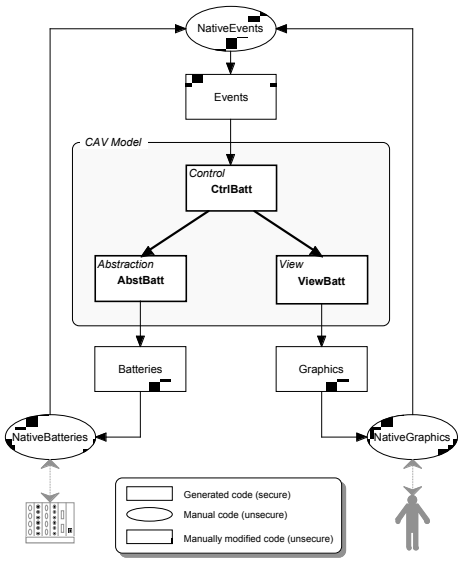


Figure 3: The CAV software architecture with interface and native modules

3.3.1. Secure Code

The core of the interactive system has been specified in three B abstract machines. These machines specify the minimum requirements of the system but do not give any implantation solution. To do so, the B method uses implementation machines that refine abstract machines. The implementation machines are programmed in BØ pseudo-code that shares the same syntax with the B language, and is close to a generic imperative programming language. In implementation machines, the substitutions are executed in sequence. BØ pseudo-code can be automatically translated into C code.

As implementation machines refine abstract machines, they must implement all the operations of the abstract machines. Moreover, the B method and semantics ensure that the side effects on variables of the implementation machine operations do respect the invariant as well as the abstract machine operations they refine. Providing the proof obligations are

actually proved, the implementation machines respect the safety and usability requirements. So, the code is secure providing the specifications are adequate.

3.3.2. Native Code and Interfaces

A working program cannot be fully developed with formal methods because most of graphic widgets and hardware drivers libraries are not yet developed with formal methods. As a consequence, the battery control panel uses three native modules:

> The NativeGraphics software module controls the graphic layout of the user interface. It uses the GTk library.
>
> The NativeBatteries software module simulates the batteries with lightweight processes. It uses the POSIX thread library.
>
> The NativeEvents software module is in charge of merging the events coming from the user or the hardware and formats them to the data structure used by the BØ translator.

These three modules are not secure. However, the modules can be tested with a reduced set of test sequences because the procedures of these modules are only called by the secure code that does respect the formal specification. For example, the bar graph widget of NativeGraphics module is to be tested with values from 0 to 100 only because the secure modules are proved to use values from 0 to 100 only. Abnormal states do not have to be tested.

The interfaces module roles are to make a syntactic filtering and translation between native modules and secure code:

> The Events software module receives integer data and translates them to 1..3 or 0..100 types. This module is secure because it as been specified and fully implemented in BØ but is called by non-secure modules.
>
> The Graphics and Batteries modules are specified in B and the skeleton of the modules is implemented in BØ and then manually modified to call the native modules NativeBatteries and NativeGraphics respectively.

3.3.3. Programming Philosophy

At last, the project outcome is a set of C source files. Some of these files are automatically generated from the BØ implementation, while others are partially generated or manually designed. The formal specification and implantation require about one thousand non-obvious proof obligations to be actually proved. All these proof obligations can be proved thanks to the automatic prover in a few dozen of minutes with a standard workstation.

The core of the system is formally specified and developed. The programming philosophy used is called the offensive programming, i.e., the programmer does not have to question about the validity of the operations calls. The B method and semantics ensure that any operation is called with respect to the specifications. Most of the dialogue controller as well as the logic of the View and the Abstraction are designed with this philosophy. As a consequence, most of the dialog control of the system is secure.

On the opposite, the events coming from the real-word –user or hardware– have to be syntactically and semantically filtered. This programming philosophy is defensive. On the one hand, the syntactic filtering is done by the Event module that casts the parameter types –from integer to intervals. On the other hand, the semantic filtering is achieved by the Control module, which can refuse events coming from disabled buttons. So, the system is resistant to graphic libraries bugs or transient errors with sensors. This filtering is required by the proof obligations that force upon the operation calls to be done with valid parameters.

There is no need to use the defensive programming philosophy in native modules. The procedures of these modules are called only by secure modules, so the parameters must be valid anytime. Neither verification nor filtering is necessary. The programming philosophy looks like the offensive philosophy except that the native modules are not formally specified but must be tested, so we name this philosophy half-offensive. As a consequence the development of high-quality native code can be performed with a reduced programming effort.

3.4.Formal method in HCI: what kind of user?

As we write upper, one of our first goals was to ensure that other people than pure formal method specialists could use the method. Did we succeed?

We must admit that this goal is not reached today. In our first attempts on the Post-It® case study, even if the B tool automatically demonstrated most proofs, it remained some of them to be demonstrated by hand. This task cannot be made by non B specialists.

In the second case, for the battery case study, we obtained a fully automated process with the B tool. But it required to pay strong attention on condition writing; more, despite of the smallness of the study, the number of generated proof obligation let us think that a much more example might “explode” the tool.

4. Incorporating formal methods in HCI tools

Another way to allow cooperation between SE and HCI is to lean on formal semantics while building a tool for HCI. We describe in this section such an approach, and show how it can bring different solutions.

In section 4.1, we shortly review the area of HCI tools, mainly GUI-Builders and Model-Based tools. Section 4.2 describes the fundamentals of our proposal: connecting directly and interactively a GUI-Builder to a functional core, by the way of formal semantics. Section 4.3 relates how to incorporate task-based analysis in this process.

4.1. A glance at HCI tools

HCI tools for building interactive software are numerous. In the meantime, few of them handle formal approaches.

On the one hand, GUI builders and tools from suites such as Visual Basic® or Jbuilder® do not provide any way to handle any kind of formal method. Code generation is another difficulty, because it enforces a code organization that does not conform to good SE practices.

On the other hand, Model-Based tools (Puerta, 1996, Puerta et al., 1999) deal with models, but are currently not usable for actual software development. Some research tools such as Petshop (Ousmane, 2001) incorporate formal methods, for some parts of software development.

Our goal is to try to incorporate formal methods in HCI tools in a deep way, with respect to usability for HCI user.

4.2A semantic link between the functional core and the HCI tool

The basic idea of our approach is to build HCI tools that lean on formal semantics to ensure that properties are maintained all along the development process. At the same time, we do not expect the user to become a formal method specialist

Our first step was to demonstrate how it is possible to build a tool that ensures a semantic formal link. We start from a formally developed functional core. We assume that this functional core, which has been specified with the B method, delivers services through an API. It is possible to automatically link such a functional core to a tool that exploits function signatures and formal specifications to help building interactive software.

In figure 4, we can see a screen copy of the GenBUILD tool. On the left, the animator consists in fully generated interface that allows to interactively run the functional core. Every function of the functional core is usable through button activation. When parameters are required, a dialog box appears to allow the user to enter them. Functions are textually described, and current state of the functional core can be estimated through the result of all functions. It is important to notice that all that part is fully automatically generated. It allows the user to "play" with his/her functional core, and to be aware of functional core state.

In the right part of the figure, we can see the GUI-Builder view, where widgets can be dropped to build the concrete user interface. In the center, as in any GUI-Builder, we can see a property window, which allows the user to finalize the presentation. Below this window, the last window permits associating events to functions from the functional core.

The great two originalities at this point are: first, at any time, we can switch from design mode to test mode where functional core can be called from either the presentation or the animator (the context of the functional core remains consistent); second, the

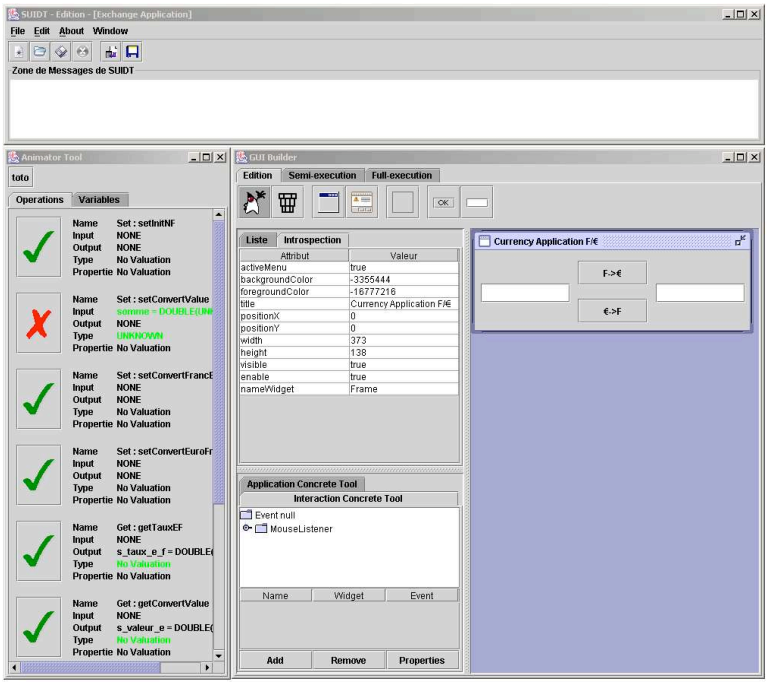


Figure 3: the GenBUILD system

system leans on formal specifications from the functional core to ensure that calls are correct.

This study demonstrates that it is possible to incorporate formal approaches in interactive tools. The benefit is not very important at this stage, because interactive model is poor: we assume that the link between widgets and functional core is direct. In the next part, we show how it is possible to enhance this model.

4.3. Linking task based analysis and formal semantics

The second step of our study consists in focusing on task-based analysis. We incorporated task-based analysis into our system by the way of two task models (abstract and concrete task models) using the CTT formalism (Paternò, 2001). In figure 5, we see on the upper left a view of the abstract task model. While CTTE (Paternò et al., 2001) provides a purely graphical view of CTT trees, we chose to draw them in a tree widget. This avoids many problems like sizing or beautifying. The original part of the study consists in the link that exists between the abstract task model and the functional core. In tools such as CTTE, we can animate the task model, in order to ensure that the specifications of the system are consistent. In GenBUILD, we can go one step further. We can animate the system itself; we exploit the possibility to interactively run the functions of the functional core, with respect to the formal specifications of this one. More, we can also link the pre- and post-conditions of CTT to functions, in order to dynamically control the execution. This is shown in the front window on figure 5.

We do not illustrate here the concrete task model, which allows the same kind of links, but on the presentation (widget) side (Baron and Girard, 2002).

With GenBUILD, we use formal specifications in an interactive way that allows non-specialists to take advantage of formal methods without heavy learning or careful usage.

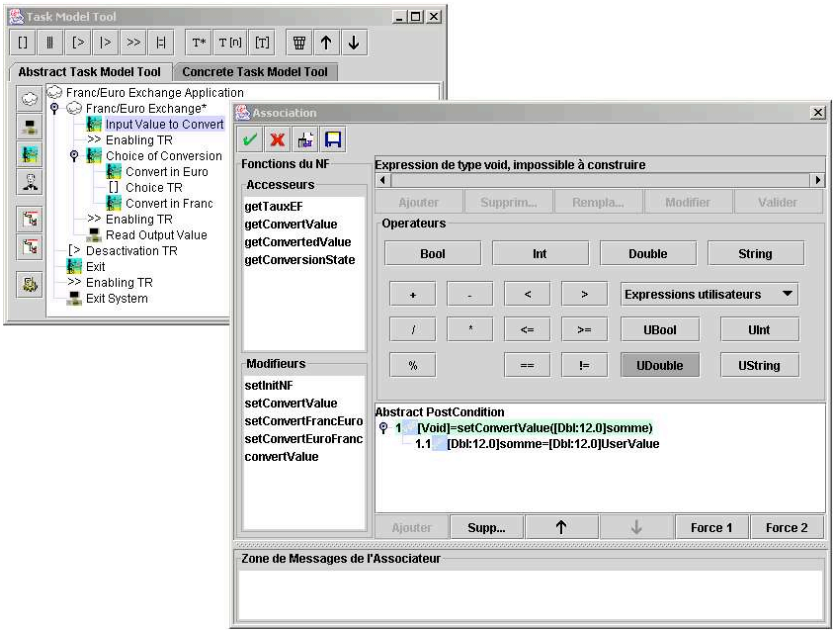


Figure 5: task-based aspects of GenBUILD

5. Lessons learned

5.1. Consideration about methods

Our studies bring partial solutions in the field of formal methods for HCI. On the one hand, they demonstrate how formal methods are really usable in HCI design. In the meantime, their usage is restricted to specialists that come to grips with mathematical fundamentals. Automatic proving is not possible in real developments. “Manual” proving is mandatory. Incorporating formal approaches into HCI tools may bring a solution: hiding formal language complexity allows HCI designers to use these methods in a blink mode.

On the other hand, we did not work on a global method to build application with such tools. We assumed that functional cores have to be designed first. In many cases, this is not the best way to work. In some cases, task analysis may turn up new needs. Modifying the functional core and its formal specifications to rebuild a new solution might be difficult. Is the opposite way possible? Is it possible to start form task analysis, to design the presentation, and then to develop the functional core, with respects to properties that might be enforced in the functional core by strong formal methods?

5.2. What is the user

One of the strongest questions that have been raised by these studies is: what kind of user for formal methods in HCI?

One the one hand, manipulating formal methods

themselve is often hard. Complete formal development is very difficult, and formal tools such as “Atelier B” are not really able to manage real scaled applications.

On the other hand, manipulating formal methods through HCI tools seems very interesting. But wher is the place for formal development? And who might make it?

All these points are to be discussed, and solutions to be bring by further work.

6. References

Abowd, G. D., Wang, H.-M. and Monk, A. F. (1995) A Formal Technique for Automated Dialogue Development, in DIS'95, Design of Interactive Systems (Eds, Olson, G. M. and Schuon, S.) ACM Press, Ann Arbor, Michigan, pp. 219-226.

Abrial, J.-R. (1996) The B Book: Assigning Programs to Meanings. Cambridge University Press.

Aït-Ameur, Y., Girard, P. and Jambon, F. (1998a) A Uniform approach for the Specification and Design of Interactive Systems: the B method, in Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'98), Vol. Proceedings (Eds, Markopoulos, P. and Johnson, P.), Abingdon, UK, pp. 333-352.

Aït-Ameur, Y., Girard, P. and Jambon, F. (1998b) Using the B formal approach for incremental specification design of interactive systems, in Engineering for Human-Computer Interaction, Vol. 22 (Eds, Chatty, S. and Dewan, P.) Kluwer Academic Publishers, pp. 91-108.

Baron, M. and Girard, P. (2002) SUIDT : A task model based GUI-Builder, in TAMODIA : Task MOdels and DIAgrams for user interface design, Vol. 1 (Eds, Pribeanu, C. and Vanderdonckt, J.) Inforec Printing House, Romania, Bucharest, pp. 64-71.

Brun, P. (1997) XTL: a temporal logic for the formal development of interactive systems, in Formal Methods for Human-Computer Interaction (Eds, Palanque, P. and Paternò, F.) Springer-Verlag, pp. 121-139.

Card, S., Moran, T. and Newell, A. (1983) The Psychology of Human-Computer Interaction. Lawrence Erlbaum Associates.

Coutaz, J. (1987) PAC, an Implementation Model for the User Interface, in IFIP TC13 Human-Computer Interaction (INTERACT'87) North-Holland, Stuttgart, pp. 431-436.

Duke, D. J. and Harrison, M. D. (1993) Abstract Interaction Objects. Computer Graphics Forum, 12, 25-36.

Goldberg, A. (1984) Smalltalk-80: The Interactive Programming Environment. Addison-Wesley.

Hix, D. and Hartson, H. R. (1993) Developping user interfaces: Ensuring usability through product & process. John Wiley & Sons, inc., Newyork, USA.

Jambon, F., Girard, P. and Aït-Ameur, Y. (2001) Interactive System Safety and Usability enforced with the development process, in Engineering for Human-Computer Interaction (8th IFIP International Conference, EHCI'01, Toronto, Canada, May 2001), Vol. 2254 (Eds, Little, R. M. and Nigay, L.) Springer, Berlin, pp. 39-55.

Ousmane, S. (2001) Spécification comportementale de composants CORBA. PhD of Univ. Université de Toulouse 1, Toulouse.

Palanque, P. (1992) Modélisation par Objets Coopératifs Interactifs d'interfaces homme-machine dirigées par l'utilisateur. PhD of Univ. Université de Toulouse I, Toulouse.

Paternò, F. (1994) A Theory of User-Interaction Objects. Journal of Visual Languages and Computing, 5, 227-249.

Paternò, F. (2001) Model-Based Design and Evaluation of Interactive Applications. Springer.

Paternò, F. and Faconti, G. P. (1992) On the LOTOS use to describe graphical interaction, Cambridge University Press, pp. 155-173.

Paternò, F., Mori, G. and Galimberti, R. (2001) CTTE: An Environment for Analysis and Development of Task Models of Cooperative Applications, in ACM CHI 2001, Vol. 2 ACM Press, Seattle.

Puerta, A. (1996) The MECANO project : comprehensive and integrated support for Model-Based Interface development, in Computer-Aided Design of User interface (CADUI'96) (Ed, Vanderdonckt, J.) Presse Universitaire de Namur, Namur, Belgium, pp. 19-35.

Puerta, A. R., Cheng, E., Ou, T. and Min, J. (1999) MOBILE : User-Centered Interface Building, ACM/SIGCHI, Pittsburgh PA USA, pp. 426-433.

Scapin, D. L. and Pierret-Golbreich, C. (1990) Towards a method for task description : MAD, in Working with display units (Eds, Berliguet, L. and Berthelette, D.) Elsevier Science Publishers, North-Holland, pp. 371-380.

Investigating User Interface Engineering in the Model Driven Architecture

Jacob W Jespersen* **and Jesper Linvald**

IT-University of Copenhagen, Glentevej 67, 2400 NV, Denmark

*corresponding author: jwj@itu.dk

Abstract: This paper describes a model-based approach to user interface engineering that we are currently investigating. The investigation is part of a project that utilizes OMG's Model Driven Architecture (MDA) to generate domain specific J2EE applications from a purely declarative application description. We discuss basic conditions for engineering a user interface in this context, primarily the central role of the *user interface model* and the model's relation to the concrete system. Finally, we argue that the MDA may be able to bridge some of the gaps between HCI and System Engineering under certain circumstances.

Keywords: HCI, model-based interface development, user interface specification, user interface modelling, Model Driven Architecture, program generation

1 Introduction

The Model Driven Architecture (MDA) is an approach to IT system specification that separates the specification of system functionality from the specification of that functionality on a specific technology platform (OMG, 2001). Notably, the MDA prescribes a Platform Independent Model (PIM) and a Platform Specific Model (PSM), which, as the names suggest, are two different ways to describe a given system.

A common use of the MDA – and the one we are pursuing – is to specify the functionality of a system in a domain specific language. Such a specification constitutes a PIM, the Platform Independent Model of the wanted system. Details about the domain we address are omitted here, but are available in a separate paper (Borch et al., 2003).

For a PIM to be realized as a running system, the declarations it contains must be turned into *artefacts* (often software programs, database tables, etc.) that can be deployed on a computing platform. The artefacts needed to realize a given PIM on a given platform constitute a corresponding PSM for the platform. In the MDA terminology, it is said that a PIM is *transformed* into a PSM.

The model-based approach to user interface engineering is not a new concept in HCI, yet it has not been successful in commercial environments (Szekely, 1996). One likely cause is the generally low adoption of model-based approaches in commercial system engineering (OMG, 2003).

Possibly, model-based user interface development can benefit from the awareness that OMG's Model Driven Architecture gets, and leverage advances in modelling and transformation technology.

In this paper we sketch our current understanding of user interface engineering in the specific MDA context and outline a basic approach that we consider relevant from an HCI perspective.

2 HCI in the MDA

An important motivation for the MDA in general is the prospect for *separation of concerns* when modelling a system. Ideally, design and specification activities within each set of concerns are independent of other concerns. The typical example of the separation is that of system functionality in terms of the application domain versus concerns for the computing platform's performance, scalability, transaction support, security mechanisms, etc.

It is not clear that HCI concerns *per se* have a role in the MDA. Most HCI concerns cannot easily be described independently from other concerns for a system; usually, every aspect of a system has the potential to impact its users. We do not see that the MDA requires new HCI virtues, however, the attempt to model (i.e. make recipes for) systems in independent 'slices' to the benefit of core system engineering will affect associated HCI activities.

2.1 User interface models

In this paper, we discuss some implications of the MDA for user interface specification and engineering activities. In brief, we believe that the MDA puts additional focus on the means to create user interface specifications as *models*. While it is trivial to model the common graphical user interface components (window, icon, menu and widget) on their own, it is not straightforward to model user interfaces at higher abstraction levels and to cover all aspects, e.g. visual design, task guidance, error handling, and help facilities.

Yet, a *user interface model* is fundamental to circumscribe user interfaces in the MDA. Such models must encompass completely the intended structure and behaviour of a user interface, its temporal as well as permanent features, and it must be *bound* to the data it represents. Binding is discussed later on, but a further qualification of the user interface model concept is beyond the scope of this paper. See (Szekely, 1996) for a discussion of the nature of user interface models.

3 Our approach

We set out with the ambition to generate complete Java 2 Enterprise Edition (J2EE) applications (Sun Microsystems, 1999) in a fully automated way from a declarative system specification in a domain specific language. Our program generator tool reads the system specification (the PIM) and generates the required J2EE artefacts (the PSM) that can be installed and run on a J2EE application server. We chose J2EE as it provides a realistic, yet manageable platform for running a distributed application.

3.1 The concrete user interface

Among the generated artefacts is the client application that allows users to use the system. The choice of client technology posed a principal design question. First and foremost, the choice was between a standard client (e.g. an internet browser) and a custom-designed 'fat' client application. Traditionally, fat clients are considered expensive to develop and maintain, but this may no longer be the case if the fat client is automatically generated (qua the MDA) from a user interface model in a sufficiently expressive language. To incorporate changes to a generated fat client in this case, the system engineer would need only to update the model to reflect the changes and then generate a new application.

We decided to utilize the MDA for the client application as well, and make our generator tool generate a fat client that runs on top of the Java Foundation Classes (Sun Microsystems Inc., 2003), which provide basic WIMP style user interaction services.

3.2 When to *bind* the user interface

The second principal design question on the client-side was that of *time of binding*, i.e. when is the user interface tied to the data it represents?

Regardless of presentation issues and visual design (which we do not discuss in this paper) the client application obviously needs data to support its operation. Data may come from any source; it can be placed directly in the model, or the model can contain references to the location where the data exists, such as a database. Either way, the model must directly or indirectly specify the data that goes into the user interface. *Binding* is loosely the process of making a connection between *one that holds* data (e.g. a database) and *one that uses* data (e.g. a user interface component).

In principle, binding can take place at any time during the user interface model transformations (PIM-to-PSM) or, ultimately, at some point during runtime. Generally speaking, *early binding* benefits runtime performance (allowing for low response times) but freezes specific choices; *late binding* adds levels of indirection, which provides variability but incur the cost of time-consuming look-ups (as well as an extra complexity to the engineering part).

There are compelling reasons to differentiate binding strategies dependent on the kind of data involved, e.g. to incur look-up overhead only when necessary. However, in order to reduce complexity we have so far not employed more than one binding strategy. For the time being, we use only late runtime binding, as this is a single binding strategy that allows us to bind to all relevant kinds of data, as explained next.

3.3 What to bind in the user interface

We identified three fundamentally different kinds of data that need representation in the user interfaces of the generated client applications, and thus need to be bound:

Type 1. The business data from the domain. This data is residing in a database on the server.

Type 2. Data about the state of the system itself, which is used e.g. to show system progress when the user is waiting for the system. This data is available by system reflection or similar service.

Type 3. Static data for help facilities, graphics, etc. This data is available locally or on the server.

Binding to the data about the state of the system's operations (Type 2) is what demands the late runtime binding strategy, since this data cannot effectively be bound before the system actually exists.

Now, deciding on the kind of data to bind is obviously only one side of the story; specifying the proper user interface components, these components' static and temporal relationships, and which components should represent which data is the necessary other side. In its current incarnation, our generator tool does only a simple mapping from user interface model element to user interface component. Details of our presentation specification style are omitted in this paper.

We use the ubiquitous Model-View-Control pattern (Krasner & Pope, 1988) to integrate data, presentation and user controls when the client application runs.

4 Concluding remarks

We believe the MDA holds some promises from an HCI perspective, primarily – as is routinely said about model-based development – because it may directly lower the cost of constructing and maintaining high-quality systems, and thus benefit users and the people involved in bringing the systems to them.

But paradoxically, the modelling aspect of the MDA seems to be a limiting factor to it being an all-round approach when it concerns user interfaces. First of all, it takes a considerable effort to reach mature *ways* to express user interface models, which we loosely call *modelling styles*. If a system engineering team is not sure that it can re-use a particular modelling style, it can be too risky to devote resources to craft and/or adopt it.

Secondly, by definition, the MDA models are the result of abstractions "suppressing selected detail to establish a simplified model" (OMG, 2003). Thus, in the course of finding modelling styles, compromises are struck between generality and individuality. As design philosophies and practices look at the world differently, it follows that no single modelling style can aspire to become universal. For example, even if user tasks and domain models are natural ingredients in a user interface model, it is not obvious whether a task orientation or a domain orientation should dominate, and current model-based development tools have chosen differently in this respect (Szekely, 1996).

Finally, one can argue that all systems are profoundly unique. Then every system will require a separate user interface modelling style, in which case the MDA has little to offer.

4.1 Domain specific modelling styles

On the other hand, it seems that when groups of systems have commonalities at a level that allow their user interface models to share a common modelling style, the MDA promises can hold.

In our case, generation of the client application complements the generation of the complete system, as previously described. Consequently, besides a user interface modelling style, we have also a 'core system modelling style' (i.e. the *style* of the PIM) that covers the specific business domain we target. So it is not surprising that the J2EE applications coming from our tool have commonalities that we can exploit when choosing the modelling style for the user interfaces. For example, since we work in a specific business domain, we know in advance the *types* of user tasks that the generated user interface must support. Thus, our user interface models can refer directly to these domain task types, and we need not model the task types explicitly. By referring to domain specific task types, we obviously make our user interface models domain specific as well. That is not a problem in our case, since we use our own equally domain specific generator tool.

4.2 System reflection at runtime

We found that the issue of binding the user interface components to the core system was a little less trivial than expected. We wanted the client application to have the means to query the system server (at the J2EE application server) about current processing status in order to inform the user during long waits. A trivial problem, yet it required us to either control the order of artefact generation (to make sure the server parts existed before the client did) or to postpone binding until runtime, where we can reasonably expect that everything exists.

Again, this aspect turns up in our case only because the core system is also being generated; usually, model-based user interface development tools can assume known data types from the core system.

Of course, other solutions can be found to solve such binding problems, but so far we haven't had reason to do so. On the contrary, the availability of a *system reflection* interface has turned out to be a valuable asset for the user interface designer.

4.3 Further work

We continue to aim at getting the most out of model-based user interface engineering in a way that is aligned with the MDA approach. Our investigation has proved valuable to realize a working user interface modelling style (and corresponding generator capabilities) that seems suitable to the specific business domain we generate systems for.

It is evident that because we generate user interfaces in a specific domain, we have been able to avoid a lot of complexity. For example, our user interface models need not cope with the modelling of user interfaces in general; we have merely identified suitable user interface constructs to act as 'types' and then determined the required variability of each type. Only the chosen variability of the types is available for specification in the model; types are implicit due to the chosen modelling style.

We see a substantial reliance on a specific domain as the most straightforward way to realize the promises in the MDA on the short term. This holds also for the model-based user interface development.

In domain specific environments it seems feasible to agree on specialized user interface modelling styles, which allows the corresponding models to be semantically rich and focused descriptions suitable for fully automated transformations to concrete user interfaces.

In this case, the MDA has the potential to successfully bridge some of the gap between HCI and system engineering.

References

Borch et al., 2003, A Model Driven Architecture for REA based systems, In the Proceedings of the Workshop on Model Driven Architecture: Foundations and Application, TR-CTIT-03-27, University of Twente

Krasner & Pope, 1988, A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system. Journal of Object Oriented Programming, 1988. vol. 1, no. 3, pp. 26-49, http://citeseer.nj.nec.com/krasner88description

OMG, 2001, Object Management Group, Model Driven Architecture, http://www.omg.com/mda/

OMG, 2003, MDA Guide Version 1.0

Sun Microsystems Inc., 2003, Java Foundation Classes, Cross Platform GUIs & Graphics, http://java.sun.com/products/jfc

Sun Microsystems, Inc., 1999, Simplified Guide to the Java 2 Platform, Enterprise Edition, http://java.sun.com/j2ee/j2ee_guide.pdf

Szekely, P.: Retrospective and Challenges for Model-Based Interface Development. In Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfaces, (Vanderdonckt, J. Ed.). Namur University Press, Namur, 1996. http://citeseer.nj.nec.com/szekely96retrospective

Towards a Model-Based Framework for Integrating Usability and Software Engineering Life Cycles

Pardha S. Pyla, Manuel A. Pérez-Quiñones, James D. Arthur & H. Rex Hartson

Virginia Tech, 660 McBryde Hall, Blacksburg, VA 24061, USA

{ppyla, perez, arthur, hartson}@cs.vt.edu

Abstract: In this position paper we propose a process model that provides a development infrastructure in which the usability engineering and software engineering life cycles co-exist in complementary roles. We describe the motivation, hurdles, rationale, arguments and implementation plan for the need, specification and the usefulness of such a model. Our approach does not merge one lifecycle's techniques into another; rather it coordinates each lifecycle's activities, timing, scope, and goals using a shared design representation between the two lifecycles. We describe potential shortcomings and conclude with an implementation agenda for this process model.

Keywords: Process model, integration, unified framework, software engineering, usability engineering

1 Introduction

1.1 Parts of interactive software systems

Interactive software systems have both functional and user interface parts. Although the separation of code into two clearly identifiable modules is not always possible, the two parts exist conceptually and each must be designed on its own terms.

The user-interface part, which often accounts for an average of half of the total lines of code (Myers and Rosson, 1992), begins as an interaction design, which finally becomes implemented in user interface software. Interaction design requires specialized usability engineering (UE) knowledge, training, and experience in topics such as human psychology, cognitive load, visual perception, task analysis, etc. The ultimate goal of UE is to create systems with measurably high usability, i.e., systems that are easy to learn, easy to use, and satisfying to their users. A practical objective is also to provide interaction design specifications that can be used to build the interactive component of a system by software engineers. In this position paper we define the usability role as that of the developer who has responsibility for building such specifications.

The functional part of a software system, sometimes called the functional core, is represented by the non-user-interface software. The design and development of this functional part requires specialized software engineering (SE) knowledge, training, and experience in topics such as algorithms, data structures, software architecture, database management, etc. The goal of SE is to create efficient and reliable systems containing the specified functionality, as well as implementing the interactive portion of the project. We define the SE role as that of the developer who has the responsibility for this goal.

To achieve the goals for both parts of an interactive system, i.e., to create an efficient and reliable system with required functionality and high usability, effective development processes are required for both UE (figure 1) and the SE lifecycles (figure 2). The UE development lifecycle is an iteration of activities for requirement analysis (e.g., needs, task, work flow, user class analysis), interaction design (e.g., usage scenarios, screen designs, information design), prototype development, and evaluation; producing a user interface interaction specification. The SE development cycle mainly consists of concept definition and requirements analysis, design (generally proceeds in two phases as shown in figure

2: preliminary and detailed design), design review, implementation, and integration & testing (I&T).

1.2 The problem: Coordinating the development of the two lifecycles

Given the fact that each of these development processes is now reasonably well established, that the two have the same high level goal of producing software that the user wants, and that the two must function together to create a single system, one might expect solid connections for collaboration and communication between the two development processes. However, the two disciplines are still typically separate and are applied independently with little coordination in product development. For example, it is not uncommon to find usability engineers being brought into the development process after the implementation stage. They are asked to 'fix' the usability of an already implemented system, and even then any changes proposed by the usability engineers that require architectural modifications are often ignored due to budget and time constraints. Those few changes that actually get retrofitted incur huge costs and modify the software modules that were not written anticipating changes. The lack of coordination between the usability and software engineers often leads to conflicts, gaps, miscommunication, spaghetti code due to unanticipated changes, brittle software, and other serious problems during development, producing systems falling short in both functionality and usability and in some cases completely failed projects.

In particular, there is a need within interactive system development projects for:

- communication among developer roles having different development activities, techniques, and vocabularies;
- coordinating independent development activities (usability and software engineers working together on role-specific activities);
- synchronizing dependent development activities (timely readiness of respective work products);
- identifying and realizing constraints and dependencies between the two parts.

1.3 Objective

The objective of our position paper is to describe a development process model that:

- integrates the two lifecycles under one common framework;
- retains the two development processes as separately identifiable processes, each with its own life cycle structure, development activities, and techniques; and
- is built upon a common overall design representation, shared by the two developer roles and processes.

The common design representation is the key to coordination of interface and functional core development, to communication among different developer roles, and identification and realization of constraints and dependencies between the two parts. The common design representation also identifies the possibility of future changes. This allows the two developer roles to

- design for change by keeping the design flexible, and to
- mitigate the changes that could be imposed on each lifecycle.

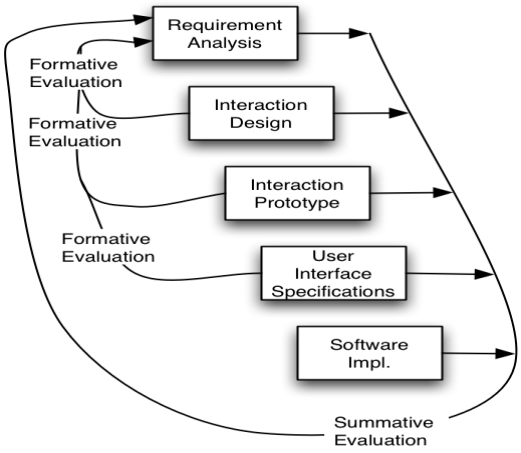


Figure 1: Usability engineering process model

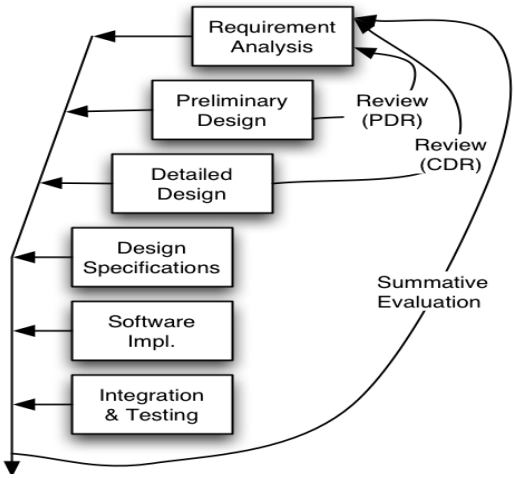


Figure 2: Software engineering process model

2 Background

2.1 Operating assumptions

A strong operating requirement for our work is to maintain UE and SE as separate lifecycle development processes for the two parts. It is not our goal to merge either development process into the other, but to establish a development infrastructure in which both can exist and function in parallel. UE and SE processes each require special knowledge and skills. Trying to integrate, for example, the UE lifecycle into the SE lifecycle, as done in (Ferre, 2003), creates a risk (and a high likelihood) of deciding conflicts in favor of software development needs and constraints, and against those of usability.

2.2 Similarities between lifecycles

At a high level, UE and SE share the same objectives:

- Seeking to understand the client's, customer's, and users' wants and needs;
- Translating these needs into system requirements; and
- Designing a system to satisfy these requirements
- Testing to help ensure their realization in the final product.

2.3 Differences between lifecycles

The objectives of the SE and UE are achieved by the two developer roles using different development processes and techniques. At a high level, the two lifecycles differ in the requirements and design phases but converge into one at the implementation stage (figure 3) because ultimately software developers implement the user interface specifications. At each stage, the two lifecycles have many differences in their activities, techniques, timelines, iterativeness, scope, roles, procedures, and focus. Some of the salient differences are identified here.

2.3.1 Different levels of iteration and evaluation

Developers of interaction designs often iterate early and frequently with design scenarios, screen sketches, paper prototypes, and low-fidelity, roughly-coded software prototypes before much, if any, software is committed to the user interface. Often this frequent and early iteration is done on a small scale and scope, often to evaluate a part of an interaction design in the context of a small number of user tasks. Usability engineers evaluate interaction designs in a number of ways, including early design walk-throughs, focus groups, usability inspections, and lab-based usability testing with the primary aim of finding errors in the design.

Software engineers identify the problem, decompose and represent the problem in the form of requirements (requirements analysis block in figure 2), transform the requirements into design specifications (preliminary and detailed design blocks in figure 2) and implement these design specifications. Traditionally these activities were performed using the rigid sequential waterfall model. Later these basic activities were incorporated into a more iterative spiral model (Boehm, 1988), with a risk analysis and an evaluation activity at the end of each stage. Even though these new development models, such as the spiral model are evolving towards the UE style by accommodating and anticipating changes at each iteration, functional software development for a system, for most part, is iterated usually on a larger scale and scope. The testing in this SE lifecycle is primarily done at the end and for verifying the implementation of the system and aims at checking the compliance of the system to specifications, completeness, and to ensure the accuracy of the integration stage.

2.3.2 Differences in terminology

Even though certain terms in both lifecycles sound similar they often mean different things. For example:

- in UE, 'testing' is a part of design and primarily aims at validating the design decisions (identified as formative evaluation in figure 1) whereas in SE 'testing' is an independent stage with the primary aim to check the implementation of the system and to verify its conformance to specifications. Analysis and validation of the design specifications performed in SE is often called 'review' (identified in figure 2) and once the specifications pass the review stage, they become a binding document between the client and the development team.
- a (use case) scenario in SE is used to "identify a thread of usage for the system to be constructed (and) provide a description of how the system will be used" (Pressman, 2001). Whereas in UE, a scenario is "a narrative or story that describes the activities of one or more persons, including information about goals, expectations, actions, and reactions (of persons)" (Rosson and Carroll, 2002).
- the SE group refers to the term 'develop' to mean creating software code, whereas the usability engineers use 'develop' to mean iterate, refine, and improve usability

Overall, the software engineers concentrate on the system whereas the usability engineers concentrate on users. Such fundamental difference in focus is one more reason why it is difficult to merge these two lifecycles.

2.3.3 Differences in requirements representation

Most requirement specifications documented by software engineers use plain English language and are generally very detailed. These specifications are written specifically to drive the SE development process. On the other hand, usability engineers specify interactive component issues such as feedback, screen layout, colors, etc. using artifacts such as prototypes, use cases, and screen sketches. These artifacts are not detailed enough to derive software specifications, instead they require additional refinement and design formulation before implementation. Therefore, they cannot be used to drive the software development process directly.

3 Current Practices

In spite of the extensive research and maturity achieved in each of these lifecycle areas, there has been a marked deficiency of understanding between the two. In general, the two teams do not understand the others' goals and needs and do not have an appreciation for the other's area of expertise. One apparent reason for this situation is the way computer science courses are typically offered in colleges: SE courses often omit any references to user interface development techniques (Douglas et al., 2002) and UE courses do not discuss the SE implications of usability patterns.

3.1 Lack of coordination

When translated into development activities, this lack of understanding between the two developer roles often leads to working separately as shown in figure 3, when they could be more efficient and effective working together. For example, both roles must do some kind of field studies to learn about client, customer, and users wants and needs, but they often do this without coordination. Software engineers visit the customers for functional requirements elicitation (Pressman, 2001), determination of physical properties and operational environments of the system (Lewis, 1992), etc. Usability engineers visit clients and users to determine, often through "ethnographic studies", how users work and what they need for computer-based support for that work. They seek task information, usage scenarios, and user class definitions. Why not do this early systems analysis together? Much value can be derived from working together on system analysis and requirements gathering in terms of team building, communication, and each lifecycle expert recognizing the value, and problems, of the other, and early agreement on goals and requirements. Instead, each development group reports its results in documentation not usually seen by people in the other lifecycle; each just uses the results to drive their part of the system design and finally merge at the implementation stage (figure 3).

Another important shortcoming of the practice shown in figure 3 is the fact that the independently generated user interface specifications on UE side and the design specifications on SE side are submitted to the development team at implementation stage. It is however, critical, to have

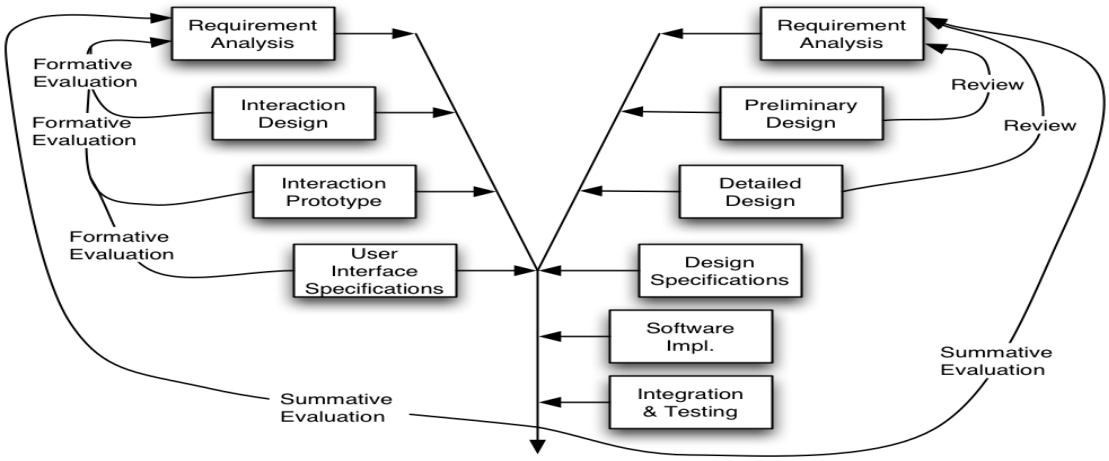


Figure 3: Current practices: Processes without communication/coordination

the user interface specifications before a design document can be generated by the software engineers because of the many dependencies and constraints between the interface on the functional core and vice versa. Moreover, such lack of coordination of development activities presents a disjointed appearance of the development team to the client. It is likely to cause confusion on the clients: "why are we being asked similar questions by two different groups from the same development team?"

3.2 Lack of provision for change

In interactive systems development, every iteration brings change. This change often affects both lifecycles because of the various dependencies that exist between the two process models. One of the most important requirements for system development is to identify the possibility for change and to design accordingly. Another important requirement is to try to mitigate the extent of change by coordinating the activities of the two lifecycles and by agreeing on a common structure upon which each developer role can base their design. The more the two developer roles work without a common structure (figure 3); the greater the possibility that the two designs efforts will have major incompatibilities.

3.3 Lack of synchronization of development schedules

Even though developers from each process can do much work in parallel, there are times when both the lifecycles must come together for various checkpoints. These checkpoints must be verified by the process during a verification and the validation stage. For example, during a final or near-final UE testing phase (identified as the formative evaluation in figure 3), both the usability and the software engineers should be present to integrate their parts and refine test plans.

However, as shown in figure 3, the more each team works independently of the other, the less likely both groups will be scheduling its development activities to be ready for the common checkpoints.

3.4 Lack of communication among different developer roles

Although the roles can successfully do much of their development independently and in parallel, a successful project demands that the two roles communicate so that each knows generally what the other is doing and how that might affect its own activities. Each group needs to know how the other group's design is progressing, what development activity they are currently performing, what insights and concerns they have about the project, and so on. But the current practice (figure 3) does permit such communication to take place because the two lifecycles operate independently and there is no structured process model to facilitate the communication between these two lifecycles.

3.5 Lack of constraint mapping and dependency checks

Because each part of an interactive system must operate with the other, many system requirements have both a user interface and a functional part. When the two roles gather requirements separately and without communication, it is easy to capture the requirements that are conflicting and incompatible. Even if there is some form of communication between the two groups, it is inevitable that some parts of the requirements or design will be forgotten or will "fall through the cracks."

As an example, software engineers perform a detailed functional analysis from the requirements of the system to be built. Usability engineers perform a hierarchical task analysis, with usage scenarios to guide design for each task, based on their requirements. Documentation of these requirements and designs is kept separately and not necessarily shared. However, each view of the requirements and design has elements that reflect counterpart elements in the other view. For example, each task in the task analysis can imply the need for corresponding functions in the SE specifications. Similarly, each function in the software design can reflect the need for support in one or more user tasks in the user interface. When some tasks are missing in the user interface or some functions are missing in the software, the respective sets of documentation are inconsistent, a detriment to success of the project.

Sometimes the design choices made in one lifecycle constrain the design options in the other. For example, in a PDA based navigational application, a usability feature could be that the application core automatically determines the location and provides navigational assistance accordingly. But for this feature to materialize the backend core should have the capability and processing power to use a GPS to pinpoint the user's location. This is an example of the backend core limiting the interaction (usability) feature. Similarly, a data visualization tool that uses dynamic queries (Ahlberg and Wistrand, 1995), in which a user is able to move a slider on the user interface to change

(filter) an attribute in the database being visualized is an example of interaction feature limiting backend core. This tool requires that the user interface be refreshed at the speed of the user action on the slider. This puts a finite time limit on the number of records in the database that the backend core can query and refresh the visualization within this time interval.

The intricacies and dependencies between user interface requirements and backend functionality have begun to appear in the literature. For example, in (Bass and John, 2001), user interface requirements and styles, such as support for undo, are mapped to particular software architectures required for the implementation of such features.

Because of the constraints on one another, independent application of the two life cycles (figure 3) would almost certainly fail and an integrated process model that facilitates communication between these two lifecycles is very essential.

4 Proposed Process Model

4.1 Activity awareness and lifecycle independence

Using our process model (figure 4), each developer role can see their own and the other's lifecycle status, activities, the iteration of activities, the timeline, techniques employed or yet to be employed, the artifacts generated or yet to be generated, and the mappings between the two domains if present. The view of each role would show only those activities that are relevant to that role. Each role views the shared design representation through its own filters (figure 5) so that, for example, the software engineers see only the software implications that result from the previously mentioned iterativeness in UE, but not the techniques used or the procedure followed. Similarly, if software engineers need iteration to try out different algorithms for functionality, it would not affect the usability lifecycle. Therefore, the process of iteration is shielded from the other role, only functionality changes are viewable through the UE filter. Each role can contribute to its own part of the lifecycle and the model allows each role to see a single set of design results, but through its own filter. Our process model places these connections and communication more on product design and less on development activities. This type of 'filter' acts as a layer of insulation, between the two processes, i.e. the process model helps isolate the parts of the development processes for one role that are not a concern of the other role. The layer needs to be concrete enough to serve the purposes, but not over specified so as to restrict the software design that will implement the user interface functionality. This prevents debates and needless concerns comparing processes and distrust on the other's techniques. Because our process model does not merge, but integrates, the two development processes, experts from one lifecycle need not know the language, terminology, and techniques of the other, and therefore can function independently.

4.2 User interface and functional core communication layer

Our process model advocates the need for the two development roles to specify a common communication layer between the user interface and the functional core parts. This layer is similar the specification of the communication between the model and the other two parts (view and controller) in the model view controller architecture (Krasner and Pope, 1988). This communication layer

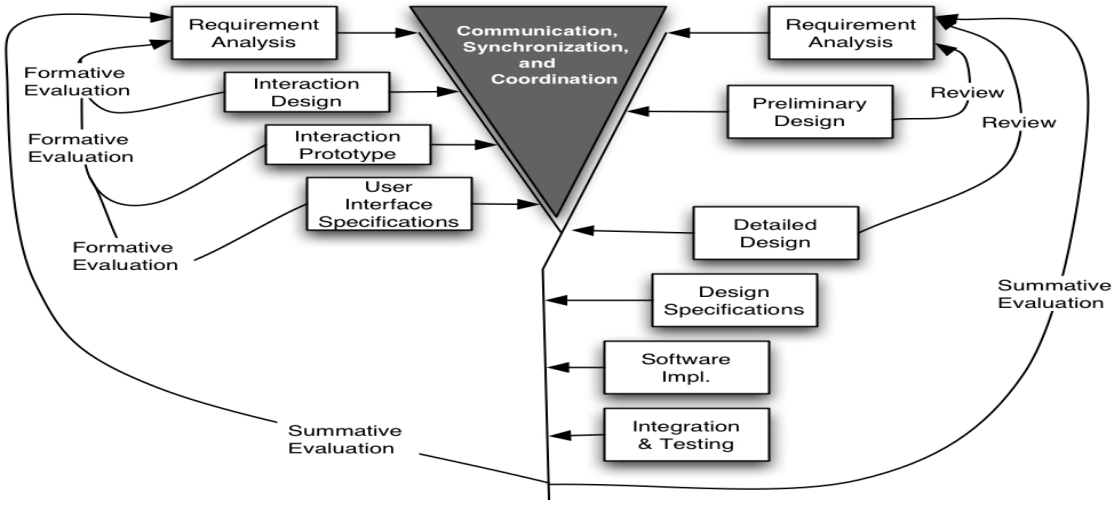


Figure 4: Proposed model: Processes with communication/coordination

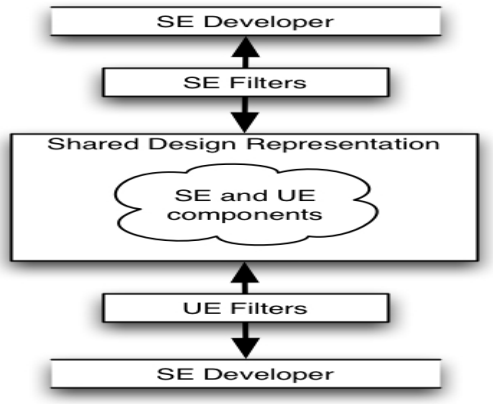


Figure 5: Shared design representation

describes the semantics and the constraints of each lifecycle's parts. For example, the usability engineer can specify that an undo operation should be supported at a particular part of the user interface and that in the event of an undo operation being invoked by the user, a predetermined set of actions must be performed by the functional core. This type of communication layer specification, which will be recorded by our process model, allows the software engineers to proceed with the design by choosing a software architecture that supports the undo operation (Bass and John, 2001). How the undo operation is shown on the user interface does not affect the SE activities. This type of early specification of a common communication layer by the two lifecycles minimizes the possibility of change on the two lifecycle activities. However, this common communication layer specification might change with every iteration and these changes should be made taking into account the implications such a change will have on the already completed activities and the ones planned for the future.

4.3 Coordination of life cycle activities

Our process model coordinates schedules and specifies the various activities that have commonalities within the two lifecycle processes. For such activities, the process model indicates where and when those activities should be performed, who the involved stakeholders are, and communicates this information to the two groups. For example, if the schedule says it is time for usability engineers to visit the clients/users for ethnographic analysis, the process model automatically alerts the software engineers and prompts them to consider joining the usability team and to coordinate for the SE's user related activities such as requirements analysis, etc.

4.4 Communication between development roles

Another important contribution of this process model is the facilitation of communication between the two roles. Communication between the two roles takes place at different levels during the development lifecycle. The three main levels in any development effort are: requirements analysis, architecture analysis, and design analysis. Each of these stages results in a set of different artifacts based on the lifecycle. The process model has the functionality to communicate these requirements between the two domains. For example, at the end of UE task analysis the usability group enters the task specifications into the model and the SE group can view these specifications to guide their functional decomposition activities. At the end of such an activity, the SE group enters their functional specifications to the model for the usability people to cross check. This communication also helps in minimizing the effects of change and the costs to fix these changes. By communicating the documents at the end of each stage, the potential for identifying errors or incompatibilities increases as compared to waiting till the usability specifications stage. This early detection of mismatches is important because the cost to fix an error in the requirements that is detected in the requirements stage itself is typically four times less than fixing it in the integration phase and 100 times less than fixing it in the maintenance stage (Boehm, 1981).

4.5 Constraints and dependencies

The design representation model incorporates automatic mapping features, which will map the SE and UE part of the overall design based on their dependencies on each other. For example, there exists a many-to-many mapping between the tasks on the user interface side and the functions on the functional side. In the event of identifying a new task after a particular iteration by the usability group, the design representation model will automatically alert the software group about the missing function(s) and vise versa. So when the software engineer tries to view the latest task addition s/he is given a description that clearly describes what the task does and what the function should do to make that task possible. This way the developers can check the dependencies at regular time intervals to see that all the tasks have functions and vice versa and that there are no 'dangling' tasks or functions that turn up as surprises when the two roles finally do get together.

4.6 Potential downsides of the model

Our process model has the following downsides due to the various overheads and additional tasks that arise because of the coordination of the two lifecycles:

- Increase in the overall software development lifecycle;
- Additional effort required by the experts in each lifecycle for document creation and entry into the design representation model;
- Additional effort required for coordination of various activities and schedules;
- Need for stricter verification process than conventional processes to enforce the various synchronisation checkpoints during the development effort; and
- Resource overhead to carry out all the above mentioned drawbacks.

5 Implementation and Test Plan

In order to build the above described integrated process model we plan to use the following agenda:

- Analyze each lifecycle in maximum detail and catalogue all activities and widely accepted domain terminology.
- Investigate each of the activities in both domains and list the artifacts that could result from these activities. These artifacts would become a part of the 'view' for the other group.
- Determine potential overlaps in these activities within the two processes and specify the type of overlap. Based on the type of overlap, the role of the developer who can perform this activity will be identified. This information would be used to alert the groups of a possible overlapping activity and also provide the profile of the person suitable for that activity.
- Investigate terminology mismatches in each of the overlapped activities and prepare a handbook for the experts.
- Create a list of potential dependencies or constraints between parts of software system design as produced by the two roles using the literature available and research any missing ones. An example of such a work is the use of architectural patterns to support usability (Bass and John 2001).
- Test the framework using a project in simulated real life settings. We plan to do this by offering the SE and UE courses in an academic semester and having half the teams use the current practices and the other half use our framework.

References

Ahlberg, C., and Wistrand, E. (1995), "IVEE: an Information Visualization and Exploration Environment." *IEEE InfoVis*, 66-73.

Bass, L. J., and John, B. E. (2001), "Supporting Usability Through Software Architecture." IEEE Computer, 113-115.

Boehm, B. W. (1981), *Software Engineering Economics*, Printice-Hall, Inc.

Boehm, B. W. (1988), "A spiral model of software development and enhancement." IEEE Computer, 61-72.

Douglas, S., Tremaine, M., Leventhal, L., Wills, C. E., and Manaris, B. (2002), "Incorporating human-computer interaction into the undergraduate computer science curriculum." *33rd SIGCSE Technical Symposium on Computer Science Education Conference Proceedings*, 211-212.

Ferre, X. (2003), "Integration of usability techniques into the software development process." *International Conference on Software Engineering (Bridging the gaps between software engineering and human-computer interaction)*, 28-35.

Krasner, G. E., and Pope, S. T. (1988), "A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80." *Journal of Object-Oriented Programming*, 1(3), 26-49.

Lewis, R. O. (1992), *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software*, John Wiley & Sons, Inc.

Myers, B. A., and Rosson, M. B. (1992), "Survey on user interface programming." *CHI '92 Conference Proceedings*, Monterey, CA, 195-202.

Pressman, R. S. (2001), *Software engineering: A practitioner's approach*, McGraw-Hill.

Rosson, M. B., and Carroll, J. M. (2002), *Usability Engineering: Scenario-based development of human-computer interaction*, Morgan Kaufman.

Extreme Evaluations – Lightweight Evaluations for Software Developers

Michael Gellner& Peter Forbrig

University of Rostock, Department of Computer Science, Software Engineering Group, Albert-Einstein-Str. 21, 18051 Rostock, Germany

{mgellner, pforbrig}@informatik.uni-rostock.de

Abstract: In most of the cases usability evaluations are done by usability experts. Employing such experts requires a certain size in business. Nevertheless users do not tolerate hard to use tools. So in a lot of small and middle sized companies developers are forced to learn handling usability aspects. This is not much easier than teaching usability engineers how to develop software. The usability evaluation process and its requirements also miss usable attributes. As a solution a light weighted usability evaluation model for software developers is created.

Keywords: usability engineering, usability testing, usability evaluation, interactive systems

1 Modeling Usability Evaluations and Usability Engineering

One of the most comprehensive views is given by Mayhew (Mayhew 1999). As shown in Figure 1 her lifecycle considers all aspects from the first steps to the successful installation. Fulfilling the requirements of this lifecycle will lead to a complex organizational structure. Further this process model does not suit well if several alternations are probable. These requirements are hard to fulfill for a

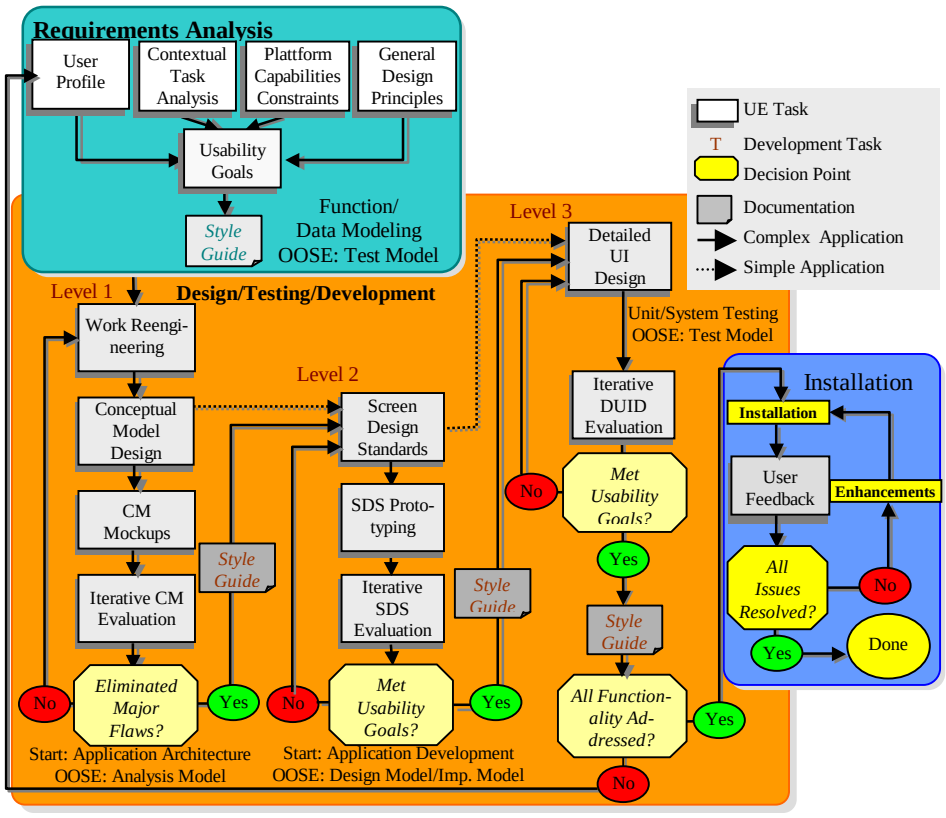


Figure 1: *Usability Engineering Lifecycle (Mayhew 1999)*

small or middle sized company. A whole staff is necessary to manage the various tasks. Mayhews lifecycle is directed to usability experts or deciders in a bigger environment that want to install an all including usability department. This approach seems not to be applicable for a small team of developers.

1. Know The user
   a. Individual user characteristics
   b. The user's current and desired tasks
   c. Functional analysis
   d. The evolution of the user and the job
2. Competitive analysis
3. Setting usability goals
   a. Financial impact analysis
4. Parallel design
5. Participatory design
6. Coordinated design of the total interface
7. Apply guidelines and heuristic analysis
8. Prototyping
9. Empirical testing
10. Iterative Design

Figure 2: *Usability Engineering Model (Nielsen 1993)*

Similar to Mayhews *Usability Engineering Lifecycle* is Nielsens *Usability Engineering Model*, see Figure 2 (Nielsen 1993). At the first view this looks like a list that has to be worked off. This is not right. Nielsons model contains circles and feedback paths, also, but it mentions only the components of usability engineering. The context is assumed to be known by the usability engineer. The correct application does not result in a kind of lightweight model. On the other hand the mentioning of the components only makes it hard for developers to apply this model.

Another model that is often shown in that context is the *Star Life Cycle* from Hartson and Hix, see Figure 3 (Preece 1994). In contradiction to Nielsens model the Star Life Cycle offers structural information beneath the components. This model is also not really helpful for developers: It shows only the steps that are known from the development models around the term »Evaluations«. This explains not much about how an evaluations could be conducted easily. Hartson and Hix do not consider the Star Life Cycle as a model for usability evaluations or for usability engineering. The focus is on an alternative to the waterfall or spiral model for development purposes since those models even do not mention evaluations or the term »usability«. So the Star Life Cycle is no solution for the problem mentioned above.

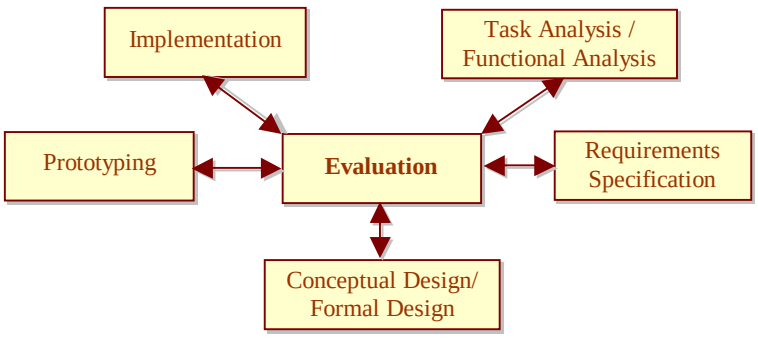


Figure 3: *Star Life Cycle [Pre 94]*

Further approaches to model usability engineering, usability evaluations or usability testing are shown by Rubin (Rubin 1994), Constantine and Lockwood (Constantine and Lockwood 1999), Dumas and Redish (Dumas and Redish 1999) and others. The approaches that show usability testing only are not discussed separately since this is only one aspect we want to cover.

2 Terminology

Before our approach is presented some terms should be determined. The first thing that is interesting is the *weight* of a model or a process. Some processes like *eXtreme Programming* (XP) or Hacking are considered as lightweight models. The counterpart to these one are models like the waterfall and the spiral model, also the Rational Unified Rrocess (RUP). The difference is the flexibility the process allows to its users (Eckstein 2000). Applying the waterfall model means to march one time from the first phase to the last one. Errors are hard to correct since it is allowed to go back only one stage if necessary. Practice shows that is often helpful to go back further. As a second criterion the quota of work is taken that counts for fulfilling the requirements of the model in relation to the work that is done to fulfill the project. Applied on the mentioned usability models Mayhews usability lifecycle and Nielsons usability engineering model are heavy weighted models. Our presented model offers a lot of liberties (see section *The Eight Phase Pattern*). It contains structural information but does not hinder to go back if it is necessary. This enables to decide nearly free how to combine phases. Since nearly no

work has to be done to fulfill the models requirements we tend to classify it as a lightweight model.
The next topic are the usability terms: *Usability engineering* includes every activity that is related to any stage of the product development to improve the usability properties. The term stands for a kind of a superset of workings and measures around this issue. For the purpose of evaluating results such models are to broad. Sub terms are *Usability Evaluation*, *Usability Inspection* and *Usability Testing*. Usability evaluation is one component of usability engineering that is realized by usability inspections and usability testing. Further components of usability engineering are *task analyses*, *requirements engineering* and others that are not considered here. Usability Testing means to test artifacts with testing persons. The artifacts can be software prototypes but also early paper and pencil mock-ups. An usability inspection is done by comparing artifacts against requirements, checklists or giving them to an expert review. See Figure 4 for a all-embracing view.

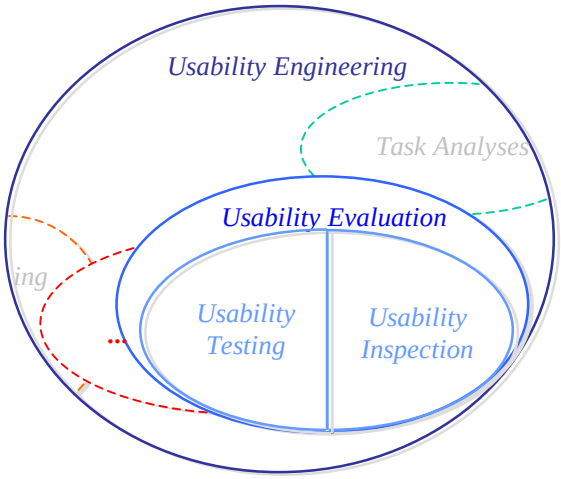


Figure 4: *Layers of Usability Terms*

The last term that should be explained is »e*Xtreme*«. The term *eXtreme Programming* (XP) is created by Beck in 1998 (Beck 1998). The consultant Beck became engaged in the *Payroll Project* at Chrysler, a project that should substitute the 15 different payroll systems that were running concurrently to that time. The project suffered on classical software engineering problems. This could be turned around with a set of innovative methods Beck introduced.
XP breaks with some established rules. The following shown approach is not as radical for usability engineering as Becks approaches are for software engineering. The term extreme is adapted to *extreme evaluations* since our approach wants to enable more agile workflows, also.

3 The Eight Phase Pattern

To be understood easily by developers the approaches is designed as a model that is known in that domain. Although the former waterfall model due to Boehm has some disadvantages it is really easy to understand and well known by software developers. For usability evaluations it is no problem to purge the prevention to go back only one phase (Gellner 2000). Our eight phased model is shown in Figure 5.

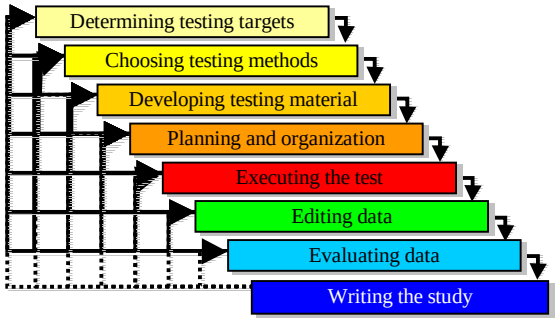


Figure 5: *Eight Phase Pattern*

Since it is part of a pattern language for extreme usability evaluations it is called *Eight Phase Pattern*. The pattern aspect can be ignored in this context. For further information see (Gellner 2003a).
Beneath the information, what subtasks has to be fulfilled also structural information are given. It is possible to go back as far as necessary at every time. In comparison to software development one cycle through the Eight Phase Pattern is a smaller task

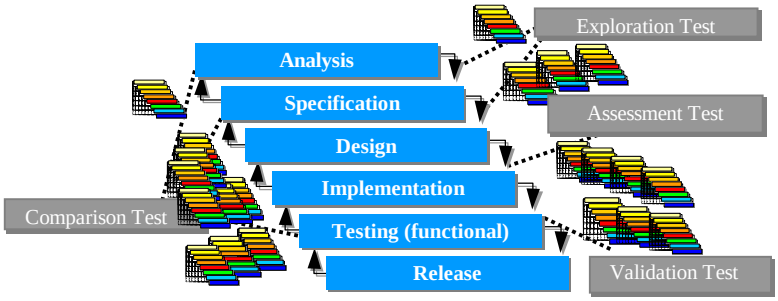


Figure 6: *Integrating the Eight Phase Pattern*

than one cycle in waterfall model for development. There one cycle stands for developing a whole release. Figure 6 shows the relations between the Eight Phase Pattern and the waterfall based development (based on). In the same way the integration of tests can take place by other development approaches (spiral model, incremental model, object oriented model etc.).
In contradiction to some other models the Eight Phase Pattern contains no component that limits the usage in external projects. The scenario of external consultants or usability experts is covered as well as in-house evaluation.

4 Applications

An important motivation for developing the Eight Phase Pattern was the fact that none of the existing models was suited well nor to categorize products for supporting usability evaluation neither to develop them. So the Eight Phase Pattern can be seen as a process pattern (take it as is and word due to it) and as a development concept (take it and cover every phase with tools). Nielsens model (see Figure 2) for example contains several points (participatory design, prototyping and others) that are hard to map into software tools, whereas every point in the Eight Phase Pattern can be seen as a component in a workflow management tool (represented in the most primitive case at least as wizards).
In comparison to software development we are nearly speechless if tools are methods have to be judges. In software development we can easily assign a tool to a certain phase in the waterfall view (IDE → implementation, UML-Painter → analyzing, specification, CASE Tool → spanning phases). This works even if the waterfall is not the underlying model. The Eight Phase Pattern enables such a communication for usability evaluations (see Figure 7).

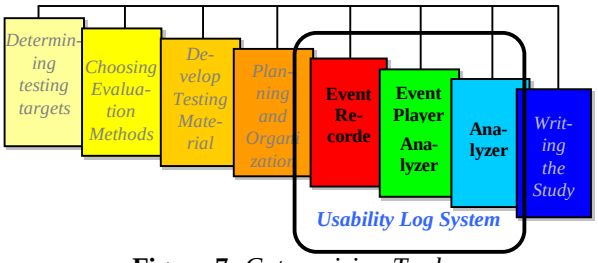


Figure 7: *Categorizing Tools*

It is also possible to analyze the costs that was caused by the works for each phase, see Figure 8.

This data can be used to compare with other labs to increase efficiency. Further it allows to see what areas or phases are not covered with tools.

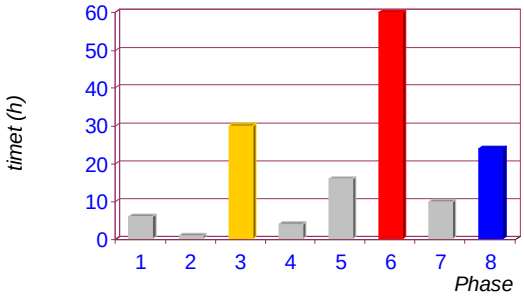


Figure 8: *Rating Performance per Phase*

Our intention to find approaches for software support led to a set of tools. Until now there are approaches and solutions for the phases 2 to 7. Tools that support phase embracing evaluations are considered as *Computer Aided Usability Evaluation Tools* (*CAUE*). This term again is created similar to the term *Computer Aided Software Engineering* (*CASE*).[i] Our most powerful approach *ObSys* combines the methods

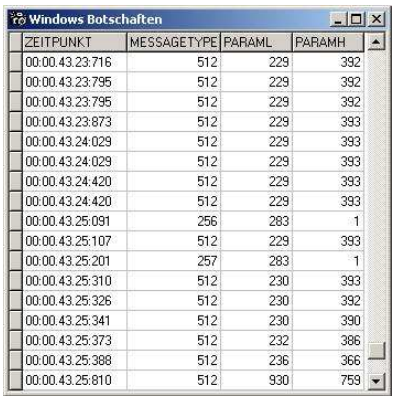
- *Event Logging* (using predefined short cuts manually to save observations)
- *Event Recording* (capturing automatically the message queue of an operating system)[ii]
- *Video Recording*
- *Screen Capturing*

Especially event recording has a high potential to automate product or prototype based usability evaluations (As known by usability experts it is to late to start with evaluations when there are prereleases. But this is not the only method we recommend.). The events are saved in databases and can be processed with SQL to find striking datasets (see Figure 9.
More concisely is our visualization called MouseMap. This multidimensional representation allows to watch the events graphically. The direction of mouse moves is visualized color gradient, clicks are pointed as round dots and the speed is

[i] In literature there is also mentioned the term *Computer Aided Usability Engineering (CAUsE)*. At the moment we see no basis for such a comprehensive demand.

[ii] At the moment determined to Microsoft Systems; beneath the *ObSyS*-Tool we have a VNC-based solution that works on all platforms that be connect to the internet

indicated by the thickness of the lines (Gellner and Forbrig 2003, Gellner 2003b).



ZEITPUNKT	MESSAGETYPE	PARAML	PARAMH
00:00.43.23:716	512	229	392
00:00.43.23:795	512	229	392
00:00.43.23:795	512	229	392
00:00.43.23:873	512	229	393
00:00.43.24:029	512	229	393
00:00.43.24:029	512	229	393
00:00.43.24:420	512	229	393
00:00.43.24:420	512	229	393
00:00.43.25:091	256	283	1
00:00.43.25:107	512	229	393
00:00.43.25:201	257	283	1
00:00.43.25:310	512	230	393
00:00.43.25:326	512	230	392
00:00.43.25:341	512	230	390
00:00.43.25:373	512	232	386
00:00.43.25:388	512	236	366
00:00.43.25:810	512	930	759

Figure 9: *List with recorded events*

MouseMaps summarize to an image what as a video sequence has to be watched as long as it took to record it, see Figure 10. On the other hand selecting to much events for a MouseMap are to complex to be analyzed qualitatively .

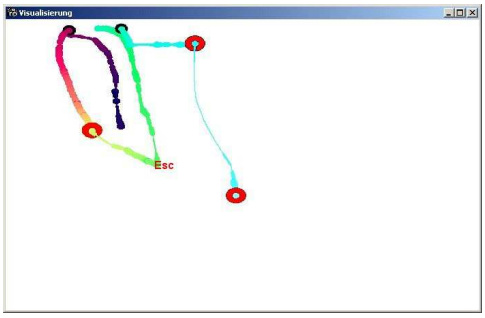


Figure 10: *MouseMap*

At the moment the error detection is not automated. An easy way to find error »candidates« is given with the time series of the events. If the time differences are visualized peaks can appear. Assumed that an user acts with a personal workflow a peak can indicate a tool based problem (but also someone asking something are else).

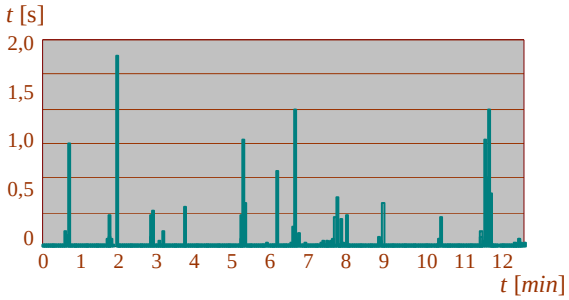


Figure 11: *Differences between events*

In an usability evaluation session all interruptions (reading scenarios, getting explanations etc.) are documented. So it is easy to distinguish between deflections and other sources for peaks.
The shown scenario in Figure 11 was observed by editing paragraphs in WinWord with built-in problems. Around the half of the peaks were caused by reading the scenario items. The other peaks indicate indeed problems. Analyzing six (± 3) exact positions in a 10 minute scenario is much more easier than watching the scenario three times on video to find fairly the same positions.

5 View and Further Work

For finding the described tools (and others) the Eight Phase Pattern was a helpful and effective approach. Further tools can be deduced and have to be realized. The most important step is the implementation of the observed error patterns for automating the detection. As a result the evaluator would get video snippets with relevant scenes and MouseMaps with the happenings around the error situation. Based on that material the evaluator must decide what steps has to be done next. A high amount of the time consuming summarizing and finding of the relevant positions were eliminated. With the shown approaches even software developers would be able to conduct usability tests and to evaluate the results. Other focuses are on the first phases. In the next month a tool for selecting well suited methods based on the algorithm in (Gellner 2002; Gellner 2003a) will be completed.
With the MouseMap visualization we found something like a body language users show by working with mice and keyboards. We assume that there are much aspects beyond simple causalities like Fitt's Law or the Steering Rule that can be investigated more in detail.

References

Beck, K. (1999), Extreme programming explained: embrace change. Addison-Wesley.

Constantine, L. L. and Lockwood L. A. D (1999), Software for use, a practical guide to the models and methods of usage-centered design. Addison Wesley Longman, Inc., Reading, Massachusetts.

Dumas, J. S. and Redish, J. C. (1999), *A Practical Guide to Usability Testing.* Revised Edition. Intellect Books Limited, Exeter, England.

Eckstein, J. (2002), XP – eXtreme Programming: Ein leichtgewichtiger Software-Entwicklungsprozess. In: basicpro, Vol. 35, 3, pp. 6-11.

Gellner, M. (2000), Modellierung des Usability Testing Prozesses im Hinblick auf den Entwurf eines Computer Aided Usability Engineering (CAUE) Systems. In: *Rostocker Informatik-Berichte*, Vol. 24, pp. 5-21. Rostock, 2000.

Gellner, M. (2002), A Pattern Based Procedure for an Automated Finding of the Right Testing Methods in Usability Evaluations. In: Forbrig, P., Limbourg, Q., Urban, B. und Vanderdonckt, J., Bricks & Blocks: *Towards Effective User Interface Patterns and Components*, Proceedings of the 9th International Workshop Design, Rostock, 2002, pp. 423-427.

Gellner, M. (2003a), Automated Determination of Patterns for Usability Evaluations. In: Hruby, P. und Sørensen, K. E.: *Proceedings of the VikingPLoP Conference 2002*, Micorsoft Business Solutions, ApS, 2003, pp. 65-80.

Gellner, M. (2003b), Mousemaps – ein Ansatz für eine Technik zur Visualisierung der Nutzung von Software und zur Automation der Entdeckung von Bedienungsfehlern. (submitted and accepted at *Mensch & Computer 2003*)

Gellner, M. und Forbrig, P. (2003), *ObSys – a Tool for Visualizing Usability Evaluation Patterns with Mousemaps*. (submitted, accepted and presented at *HCI International 2003*)

Mayhew, D. J. (1999), The Usability Engineering Lifecycle. Morgan Kaufmann Publishers, Inc., Kalifornien, San Francisco.

Nielsen, J. (1993), *Usability Engineering*. AP Proffessional, New Jersey.

Preece, J., (1994), Human-Computer-Interaction. Addison-Wesley, Harlow,.

Rubin, J., (1994) Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests, John Wiley & Sons, Inc..

AN APPROACH TO INTEGRATE HCI AND SE IN REQUIREMENTS ENGINEERING

Kenia Soares Sousa[1], Elizabeth Furtado[2]
[1]Mentores Consultoria - Rua Joao Carvalho, 800 - S.511 –CE – Brasil
[2]Universidade de Fortaleza - Av. Washington Soares, 1321 – CE – Brasil
kenia@mentores.com.br, elizabet@unifor.br

Abstract: This research work intends to present three workflows from a new Software Development Process (SDP) that, besides focusing on cost and schedule, also includes some Human-Computer Interaction (HCI) aspects along the software life cycle. Such HCI aspects include usability, accessibility, acceptability requirements, guidelines application, model-based User Interface (UI) generation techniques, and evaluation techniques. This study is based on the world-wide renowned Rational Unified Process (RUP), which is a basis for the definition of this new SDP, called the UPi (Unified Process for Interactive systems). The purpose of this integration is to develop interactive systems that are easy to learn and use, therefore, helping users in performing their daily tasks in an efficient manner. Besides that, this integration intends to bring together researchers from the HCI and Software Engineering (SE) areas of study, which started to be a concern in the research community in the last years.

Keywords: Human-Computer Interaction, Software Engineering, Interactive Systems, Software Development Process, RUP.

1 Introduction

This research work intends to present a way to integrate experts from the HCI and SE fields while working in a Software Development Process (SDP). A software development organization can apply a SDP that only focuses on delivering software on the pre-defined cost and schedule, and that accommodates users' expectations related to functionalities. On the other hand, a software development organization can choose to apply a SDP that focuses on the users' characteristics, the tasks they perform, the work environment where they are located, and the devices they use to perform their tasks. On the first case, most activities performed during the software lifecycle are related to: scope and risk control; functionalities definition; components design, implementation, and integration; functionalities tests; change control; tools definition; and documentation development. On the second case, the focus is more closely related to human factors (Shneiderman, 1998), usability (Nielsen, 1993), acceptability, accessibility, guidelines, evaluation techniques (Preece, 1994) and UI modeling. These aspects, when applied in a SDP, may bring benefits such as decrease in time to learn how to use the software, increase in the speed in performing tasks using the software, and increase in the user satisfaction towards the software.

This work is a continuous research on the integration of HCI aspects in SE, more specifically in the SDP. Our intention is to associate both areas in order to increase user satisfaction from the use of software derived from a user-centered SDP. We exposed our first ideas related to the addition of HCI aspects in only four RUP workflows in (Sousa, 2003a), and detailed and exemplified this work in (Sousa, 2003b). We extended this work by applying HCI aspects in all of the RUP workflows in (Sousa, 2003c). Now, we focus on the UPi business modeling, requirements, and analysis and design workflows in order to achieve better results in terms of trying to help HCI and SE experts to work as a team. Thus, this research focus on the workflows performed early in the life cycle in order to provide developers with a solid basis for implementing usable systems.

We intend to demonstrate that HCI and SE experts can work together efficiently, that is, their work can complement each other's work. Since most software development companies in the market-place do not even know HCI concepts, we intend to insert HCI activities, workers, and artifacts in a SDP in an

easy manner. Considering that the RUP is applied worldwide, we intend to provide those organizations a way to easily insert HCI in the SDP that they already apply in their software projects. Focusing on facilitating such application, we only recommend new activities, artifacts, and workers; maintaining the UPi workflows the same ones as in the RUP.

In this research work, we present how the RUP suggests the performance of three workflows (Kruchten, 2000), and we recommend the participation of HCI experts in those workflows. Each workflow is composed of workflow details, which have a set of activities to be performed by workers that produce valuable results, called artifacts.

2 Related Work

(Kruchten, 2001) presents the concern on UI design in the RUP by including the worker UI Designer, responsible for performing the activities UI Modeling and UI Prototyping in the Requirements Workflow. In the activity UI Modeling, the UI Designer analyzes the use case model, and creates a use case storyboard. This artifact is composed of a textual description of the interaction between the user and the system, interaction diagrams, class diagrams, usability requirements, references to the UI prototype, and a trace dependency to the use case. In the activity UI Prototyping, the UI Designer designs and implements the UI prototype, then obtains feedback from other project members, external usability experts, and users. Even though there is a concern on the UI design, there is no concern on the specification of users' tasks. Use cases differ from task descriptions in the sense that they do not provide details related to purpose, preconditions, frequency, and post conditions. Such specification is not a use case goal, but it is responsibility of task descriptions. We suggest use case and task modeling to be used in a complementary manner.

(Phillips, 2002) suggests the use of tabular representation of use cases in order to describe the flow of events, and the use of UI element clusters, which can be used as references to the UI prototype. Tabular use cases separate user and system actions. (Lauesen, 2001) argues that separating users' and systems' actions as early as in requirements may be a barrier for future decisions in the project. Therefore, he suggests the use of task descriptions, which specify what the users and the system shall do, not dividing the work between them. On the other hand, (Constantine, 1999) focuses on preparing task cases that address users' intentions rather than their actions and on system responsibilities rather than on responses. This approach offers a requirement modeling more technology-independent, providing designers the flexibility related to the actual interface components when designing the UI, but maintaining conformity to users' needs.

There are other research works related to extending the RUP, such as (Piveta, 2002) and (Massoni, 2002), but they concern changes in the process regarding aspect-oriented principles. Piveta suggests changes in the Analysis and Design Workflow in order to suit the modeling techniques necessary for the Aspect-Oriented Software Development. Massoni suggests changes in the Implementation Workflow by creating the Progressive Implementation Method, which provides the progressive implementation of three different aspects: persistence, distribution, and concurrency. Therefore, none of their extensions affect the generation of usable UIs.

3 The Business Modeling Workflow

The RUP business modeling workflow focuses on establishing a common understanding of the target organization in order to derive the software requirements. The understanding of the organizational structure (e.g. processes, roles, etc.) has an impact on the delivered product, which should help workers in accomplishing their daily tasks. This workflow is depicted in figure 1.

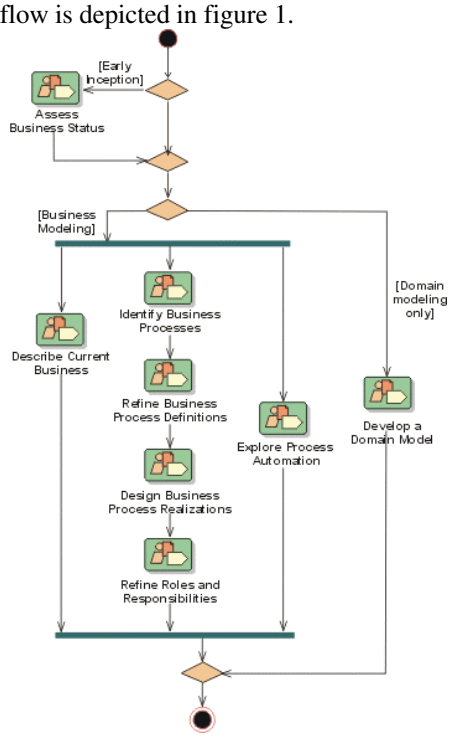


Figure 1: The Business Modeling Workflow from (RUP, 2003)

Most software development organizations perform the previously mentioned activities as part of the requirements workflow. Nevertheless, the RUP makes a clear distinction between these two workflows. The business modeling workflow results in an understanding of the organization, and the requirements workflow uses this outcome as an input in order to better define the system scope. Eventhough, they have distinct purposes, they are complementary, and their activities can be performed in parallel in order to provide a faster development process.

In the UPi, we suggest the focus on users' characteristics and environment, since they influence designers in better understanding the organizational structure. In order to fulfill this goal, two workflow details are changed:

- In the "Assess Business Status" workflow detail, the business-process analyst is responsible for understanding the target organization, that is, its vocabulary, goals and business rules, and documenting them on specific documents.

In the UPi, we recommend the business-process analyst and the UI designer to prepare the context of use model (figure 2). The first one collaborates with the knowledge on the business, and the latter collaborates with the knowledge on UI modeling. This model represents the environment where users are located in order to perform their tasks. The context of use model can be generated by the analysis of the environment where the interactive system will operate (e.g. analysis of users' surroundings, situation, and location). This model will be important to adapt the system's physical UI based on environment's constraints, especially for mobile devices that are used in a wide variety of locations. For instance, a palm will be necessary for a system while users work walking. The context of use model representation is based on the business use case model, and on scenarios' descriptions, which represent real situations faced by users while performing their daily tasks.

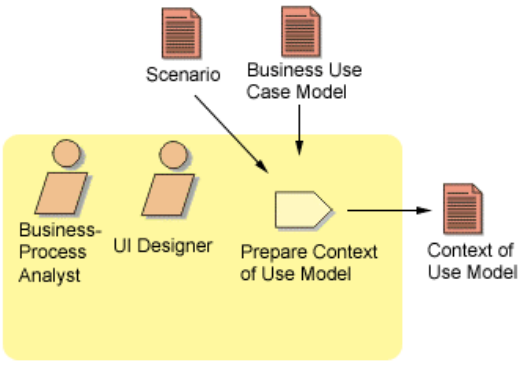


Figure 2: Extra Activities in the Assess Business Status Workflow Detail

- In the "Refine Roles and Responsibilities" workflow detail, the business designer is responsible for defining details of business workers and entities.

In the UPi, we recommend the business designer to use the user model while detailing business workers (figure 3). The user model, obtained from the requirements workflow, represents users' characteristics, focusing more on human factors than on work responsibilities. This addition brings a perspective related to users' reality, different from a business point of view. Considering that this workflow is performed in parallel with the requirements workflow, it is important to mention that this workflow detail needs to be performed after defining the system in the requirements workflow.

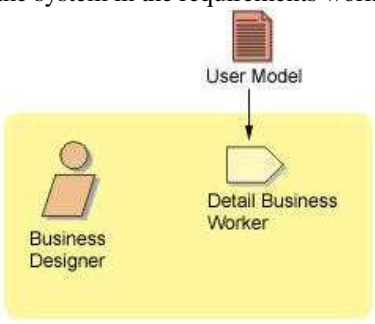


Figure 3: Extra Activities in the Refine Roles and Responsibilities Workflow Detail

4 The Requirements Workflow

The RUP requirements workflow has the purpose of defining the system scope. The system scope needs to be based on the detection of problems in the target organization and on understanding users' needs. Then, this scope needs to be managed, and refined in order to reflect users' changing requests. This workflow is depicted in figure 4.

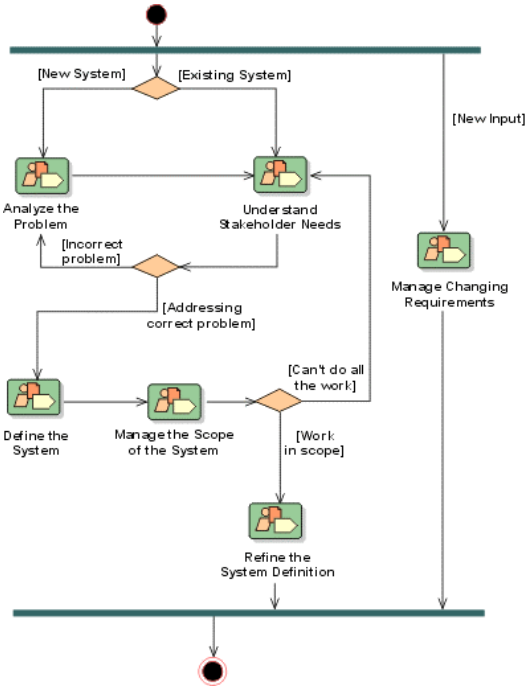


Figure 4: The RUP Requirements Workflow from (RUP, 2003)

The UPi requirements workflow focuses on users' needs, and requires the addition of activities that are related to the production of artifacts that are valuable to the generation of usable systems. For this reason, two workflow details in the RUP are changed, that is, HCI activities are added into them.

Before stating the changes suggested in the workflow details, it is important to explain the "Understand Stakeholder Needs" workflow detail. In this workflow detail, the system analyst is responsible for eliciting stakeholder requests in order to understand and document them, and prepare the use case model. Most of these requests are related to the system functionality; therefore, it is recommended that non-functional requirements are textually registered in a different document. Instead of that, we suggest the registration of non-functional requirements in the use-case, user, and task models, which are generated in the following workflow detail.

- In the "Define the System" workflow detail, the system analyst is responsible for analyzing the stakeholders' requests, and refining the use case model.

In the UPi, we, otherwise, recommend the participation of the UI designer, and the ergonomist in this workflow detail in order to produce the use case, the user, and the task model (figure 5). It becomes easier to achieve better results when users participate in these modeling activities.

This way, the system analyst provides the information that reflects stakeholder's needs as a basis for the UI designer and the ergonomist to prepare those models that are useful throughout the development process, especially in the UI generation.

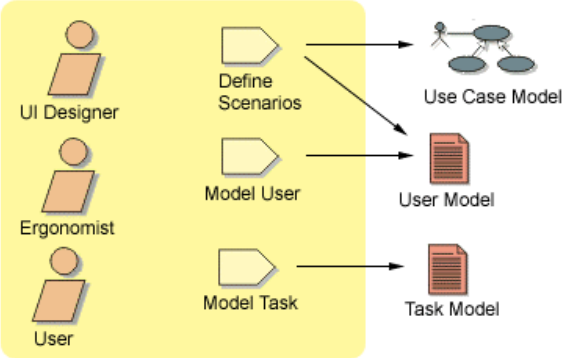


Figure 5: Extra Activities in the Define the System Workflow Detail

We recommend the preparation of scenarios, which are useful to define the use case model, the user model, and the context of use model, prepared in the business modeling workflow. Scenarios guarantee that the use case model represents users' reality, in terms of what are their real needs towards the system, not what systems analysts think the system should do.

The user model specifies characteristics of users who will interact with the system. The user model defines the scope of the user population, including characteristics, such as intellectual and physical abilities, previous experience with using interactive systems and performing the required task. Its greatest advantage is the impact of users' classification on the UI suitability to their individuality.

We also recommend the preparation of the task model, which defines the tasks performed by users. The task model is a hierarchical structure of tasks and sub-tasks graphically represented. In this work, we used the MAD formalism (Pierret-Golbreich, 1989), in which a task is divided in two or more subtasks that are performed in order to fulfill a certain objective. The task model is generated based on the use case model, defined during this workflow, which defines system's processes. The task model is useful to allow the system analyst to envision what tasks users perform, predict difficulties, evaluate systems, measure system complexity, and predict performance, among other aspects.

- In the "Refine the System Definition" workflow detail, the requirements specifier is responsible for detailing software requirements, and documenting the results in the software requirements specification and in the supplementary specifications

documents. Since these requirements are focused on functionality, we enhance this workflow detail with activities that result in HCI artifacts. Therefore, the requirements specifier has its artifacts complemented with information generated by the UI designer and the ergonomist.

In the UPi, we recommend the participation of the UI designer in this workflow detail in order to analyze the UI models generated in the previuos workflow detail aiming at preparing a usability requirements list (figure 6). These requirements are related to users' satisfaction and the performance of the system. Usability requirements directly influence aspects of the system quality of use (e.g. easy access to shortcuts, number of data users need to remember, etc.), because the guidelines applied in the UI during the implementation workflow, and the system evaluation during the test workflow are based on these requirements. These requirements can be gathered by analyzing the user, the task, and the context of use models prepared during this workflow; and they are organized in use cases. One usability requirement can be associated to many use cases. So, the matrix that represents the association of requirements with use cases is updated after each analysis.

Besides that, we also recommend the participation of the UI designer, and the ergonomist in order to define usability recommendations based on the users, their tasks, their working environment, and the system domain (figure 6), thus, generating guidelines. It becomes easier to achieve better results when users participate in these definitions.

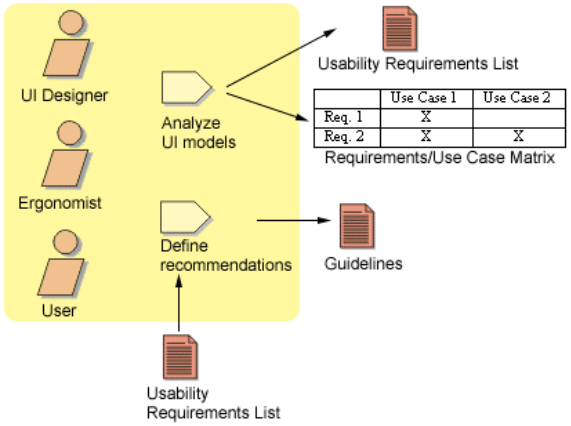


Figure 6: Extra Activities in the Refine the System Definition Workflow Detail

Guidelines definitions are important to provide consistency in the UI, increasing user productivity, decreasing number of errors, among other aspects related to usability. Guidelines defined during the requirements workflow are related to the system conceptual level, and are used as a basis for the definition of guidelines related to other workflows. For example, the guideline "use metaphors for operations" is useful to define the guideline "use a printer image for the print operation", necessary for the UI generation during the implementation workflow. Guidelines are defined by UI designers based on usability requirements. For instance, a use case "system maintenance" for any system that needs internal control, it could have "maintenance speed" as one usability requirement. Thus, the generating the following semantic guideline: use metaphors that group together actions that are familiar to users.

5 The Analysis and Design Workflow

The RUP analysis and design workflow has the purpose of transforming requirements into a design specification useful for the implementation workflow. This workflow is depicted in figure 7, in which the software architect is responsible for defining, and refining the system architecture. The system architecture is based on components, which can be tested individually and integrated gradually to compose a complete system.

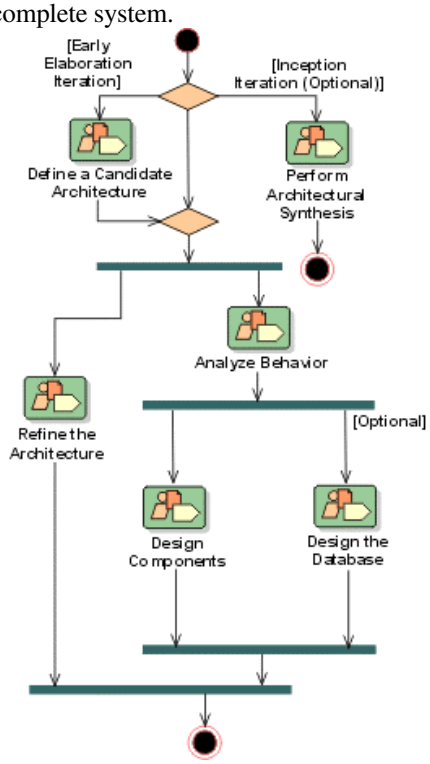


Figure 7: The RUP Analysis and Design Workflow from (RUP, 2003)

The UPi analysis and design workflow uses the models prepared in the requirements and business

modeling workflows in order to generate more usable UIs. With this goal in mind, HCI activities are included in three existing workflow details:

- In the "Analyze Behavior" workflow detail, the software architect is responsible for analyzing and transforming use case behavioral descriptions, and the interactive characteristics of tasks (e.g. sequence and repetition control) associated to the use case model into elements useful for design.

In the UPi, we recommend the participation of the UI designer in order to refine the task model, generated in the requirements workflow (figure 8). Such refinement may be done by considering characteristics of different kinds of users and different interaction styles.

This way, the software architect is identifying the system classes, and detailing use-case flow of events in terms of the system functionality, but also, taking into consideration the possibilities of system adaptation based on users specific characteristics.

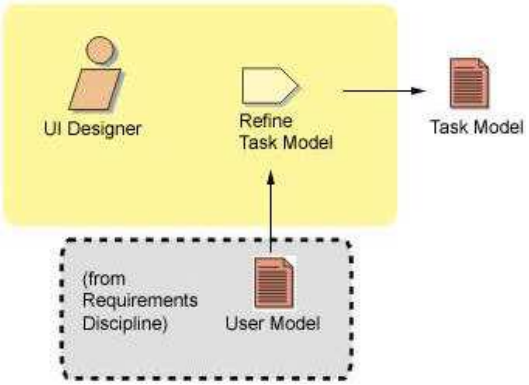


Figure 8: Extra Activities in the Analyze Behavior Workflow Detail

- In the "Refine the Architecture" workflow detail, the software architect is responsible for defining the archicteture by focusing on implementation and deployment.

In the UPi, we recommend the participation of the UI designer in order to prepare the interface conceptual model, and analyze UI design options (figure 9). The Interface Conceptual Model (MIC) (Furtado, 2002) that represents the interactive part of the system by graphically representing the navigational structure of the system. The MIC is generated based on the task model, on the context of use model, and on a set of guidelines applied to translate the task model into the MIC. For instance, there is one guideline that states that if a task in the task model shall only be performed when a user validates it, then two interaction objects have to be created in the MIC (e.g. confirm and cancel). There is another one that states that each interruptible task in the task model shall originate an interactive space in the MIC. The production of the MIC can generate design options because the UI designer is able to prepare this model for different kinds of users, e.g. experts and novices. The design options are generated using the MIC. This analysis is performed by the association of usability requirements to design options of a specific task in order to help the UI designer when trying to choose, along with the users, the best option.

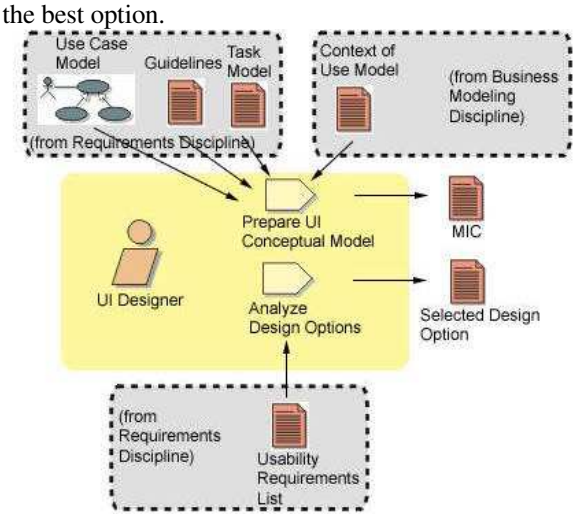


Figure 9: Extra Activities in the Refine the Architecture Workflow Detail

As a result of the participation of the software architect and the UI designer, we have a system that is well-structured in terms of its functionality, and usability.

- In the "Design Components" workflow detail, the class characteristics and relationships are completely specified.

In the UPi, we recommend the class model to be enhanced by being specified in a n-tier architecture (figure 10). 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. This approach allows the accordance to the independence dialogue principle (Lafon, 1991), which means modeling the system application independent from the system UI. The main goal is to facilitate changes in these layers. This way, a change in the UI will not affect the application, and vice-versa.

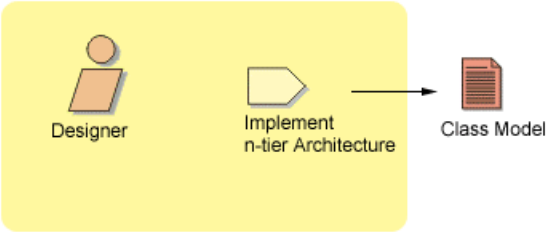


Figure 10: Extra Activity in the Design Components Workflow Detail

6 Conclusion

This research work presented three UPi workflows, the business modeling, requirements, and analysis and design workflow, focusing on how to integrate HCI and SE workers, activities, and artifacts. The Business Modeling Workflow is enhanced by focusing on users´ characteristics and environment. The Requirements Workflow is enhanced by focusing on model-based UI generation, on usability goals, and with the application of model guidelines. The Analysis and Design Workflow is enhanced by focusing on different kinds of users and interaction styles, dialogue independence, and usability requirements.

The focus of HCI aspects in the UPi workflows brings benefits both to the HCI and SE experts. The first ones benefit from system functionality definition, leading to better precision useful for the UI definition. The latter benefit from usability, acceptability, and accessibility aspects, leading into user satisfaction.

As for future work, we intend to continue this closer integration in the other UPi workflows. Besides that, we intend to suggest a model that integrates the most important characteristics from the user, task, context of use, and use case model. In other words, we want to visually demonstrate how specific users intend to perform their tasks in the envirnonment where they are located.

References

Constantine, Larry; Lockwood, Lucy. (1999), *Software for Use: A Practical Guide to Models and Methods of usage-Centered Design.* Massachusetts: Addison-Wesley.

Furtado, Elizabeth; Sousa, Kênia; Cólera, César. (2002), 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, pp. 84 – 92.

Kruchten, Philippe. (2000), *The Rational Unified Process - An Introduction.* 2 ed. New Jersey: Addison-Wesley.

Kruchten, Philippe; Ahlqvist, S; Bylund S. (2001), *User Interface Design in the Rational Unified Process, in Object Modeling and User Interface Design.* Massachusetts: Addison-Wesley.

Lafon, M. (1991), Interfaces Homme-machine : Vue d'ensemble et perspectives. *Actes du congrès Génie logiciel & Systèmes Experts, Interfaces homme-machine*, Maquettage & prototypage. Vol. 24, pp. 4-16.

Lauesen, Soren. (2001), Task & Support - Task Descriptions as Functional Requirements. *In: Proceedings of AWRE*, pp. 83-91.

Massoni, Tiago; Sampaio, Augusto; Borba, Paulo. (2002), A RUP-Based Software Process Supporting Progressive Implementation. *In: 2002 International Conference of the Information Resources Management Association (IRMA'2002)*, pp. 480-483.

Nielsen, Jakob. (1993), *Usability Engineering.* Califronia : Morgan Kaufmann.

Phillips, Chris; Kemp, Elizabeth. (2002), In Support of User Interface Design in the Rational Unified Process. *In: Proceedings of the Third Australasian User Interface Conference*, pp. 21-27.

Pierret-Golbreich, Christine.; Scapin, Dominique. (1989), MAD: Méthode Analytique de Description des tâches. *Colloque sun língénierie des interfaces homme-machine.* Sophia-Antipolis.

Piveta, Eduardo Kessler; Devegili, Augusto Jun. (2002), Aspects in the Rational Unified Process' Analysis and Design Workflow. *In: Proceedings of the Aspect Oriented Design (ADO).*

Preece, Jenny et al. (1994), *Human-Computer Interaction.* England: Addison-Wesley.

RUP. (2003), *The Rational Unified Process.* Available in: <http://www.rational.com/products/rup/tryit/eval/gen_eval.jsp>. Accessed in 23 apr. 2003.

Shneiderman, Ben. (1998), *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* 3 ed. England: Addison-Wesley.

Sousa, Kenia; Furtado, Elizabeth. (2003a), Integration of Human-Computer Interaction in a Software Development Process. *In: Human-Computer Interaction International* – HCII'2003.

Sousa, Kenia; Furtado, Elizabeth. (2003b), RUPi – A Unified Process that Integrates Human-Computer Interaction and Software Engineering. *In: International Conference on Software Engineering – ICSE'2003*, pp. 41-48.

Sousa, Kenia; Furtado, Elizabeth. (2003c) A Unified Process for Interactive Systems. *In: Latin American Conference on Human-Computer Interaction – CLIHC'* 2003.

What drives software development: issues integrating software engineering and human-computer interaction

Nuno Jardim Nunes
Universidade da Madeira, Campus da Penteada, Funchal, Portugal
njn@acm.org

Abstract: This paper discusses two important issues about integrating SE and HCI. The first issue is user-centred development and the role of users and their goals driving software development. User-centred development is usually conceived by software developers has a methodological guidance towards bringing users into the development lifecycle and iterating design alternatives. Here we discuss that the major advantage of user-centred development to software engineering is the intrinsic capability of driving development in a way that helps reduce the complexity of software projects, add-value to users and business and detect important software architecture problems. In this paper we present and contrast several examples from the Unified Process and the agile methods literature with a new proposal specifically tailored for interactive systems. We then argue that usability and user-interface concerns have an important impact on software-architecture.

Keywords: Software engineering, human-computer interaction, user-centred development, user-interface design

1 Introduction

User-centred design (UCD) is currently defined in the ISO 13407 standard and typically entails involving users in the design and evaluation of the system so that feedback can be obtained. Therefore, UCD produces software products that are easier to understand and use; improves the quality of life of users by reducing stress and improving satisfaction; and improves the productivity and operational efficiency of individual users and the overall organization.

Activities required to achieve UCD are well-known: understand and specify the context of use, specify users and their organizational requirements, produce designs and prototypes and carry out user-based assessment.

Despite the fact that principles and activities behind UCD were identified during the 80s (if not earlier) there is still discussion about the integration of UCD practices in modern software development. However, careful inspection of modern lifecycle models (for instance the Unified Process (UP) and it's commercial counterpart the Rational Unified Process - RUP) clearly denotes that UCD principles and activities are an integral part of modern software development.

> The Unified Process is use-case driven to denote the emphasis on knowing and understanding what real users want and need to accomplish with the envisioned system.
>
> The Unified Process is architecture-centric, meaning that it focuses on the architecturally significant static and dynamic aspects of the system and the architecture grows out of the needs of the business and users reflected in the use-cases.
>
> The Unified Process is also iterative and incremental meaning that development evolves in terms of mini-projects that correspond to controlled and planned iterations that result in increments to the end product.

One can argue that those characteristics of UP are not sustained in specific process activities. However, the same argument could be used to discuss the practical applicability of UCD. In essence both UCD and UP (or RUP) are high-level models that define a set of principles and activities – discussing their practical applicability in terms of those high-level characteristics is an interesting but controversial (and perhaps pointless) exercise. Moreover, while UP and RUP define specific and well-documented workflows, activities and roles; UCD (at least at the standard level) lacks such a fine-grained description.

At the other extreme, of modern software development, we have the so-called agile movement

(Agile Alliance, 2003). Again we can find similarities between UCD principles and, for instance, the principles behind the Agile Manifesto (Agile Manifesto, 2003).

> Agile methods promote that their highest priority is to satisfy customers through early and continuous delivery of valuable software.
>
> Agile methods welcome changing requirements and deliver working software frequently.
>
> Finally the agile movement promotes that business people and developers must work daily throughout the project.

Again one could argue that agile methods are just another buzzword for chaotic development. But evidence exists that lightweight techniques - such us small and continuous releases, refactoring, pair-programming and on-site customers – contribute to promote communication, simplicity and feedback.

2 Driving Software Development

The use-case driven nature of the UP (and RUP), is grounded on two basic assumptions. On the one hand, the use-case strategy forces developers to think in terms of real value to the users and not just in terms of functionalities. On the other hand, use-cases drive the development process because they are used to understand the requirements, they drive the development and review of the analysis and design level models, and they also facilitate testing of the implementation components for conformance with the requirements.

The adoption of use-cases in the Unified Modelling Language (UML) acknowledges this importance of identifying the different roles users play when interacting with an application to support their tasks. However they are still mainly used to structure the application internals and don't provide an efficient way to support the usability aspects of interactive systems.

The major problems with these descriptions of user behaviour are related to the system-centric nature of use-cases. In (Constantine and Lockwood, 2001) the authors argue that "conventional use-cases typically contain too many built-in assumptions, often hidden and implicit, about the form of the user interface that is yet to be designed." Such argument led the authors to propose the essential use-case narrative, a technology-free, implementation independent, structured scenario expressed in the language of the application domain and end-users.

The essential quality (technology free and implementation independent) of descriptions of user behaviour can also be applied to diagrammatic representations. In (Nunes and Cunha, 2001) we present a proposal to detail use-cases with activity diagrams following the same essential and technology-free principle. Our proposal takes Constantine's approach even further because it enables developers to clearly identify architectural significant concepts from the essential use-case descriptions. Furthermore our approach enables developers to use participatory techniques to elaborate the use-cases with the end-users in a way similar to the techniques used in card games (such us CRC which is also used in agile methods like extreme programming – XP, and others).

Conventional Use-case: Withdraw Money

(Transcription from [Kruchten 1999, pp. 96-97])

1. The use case begins when the Client inserts an ATM card. The system reads and validates the information on the card.
2. System prompts for PIN. The Client enters PIN. The system validates the PIN.
3. System asks which operation the client wishes to perform. Client selects "Cash withdrawal."
4. System requests amounts. Client enters amount.
5. System requests account type. Client selects account type (checking, savings, credit).
6. System communicates with the ATM network to validate account ID, PIN, and availability of the amount requested.
7. System asks the client whether he or she wants a receipt. This step is performed only if there is paper left to print the receipt.
8. System asks the client to withdraw the card. Client withdraws card. (This is a security measure to ensure that Clients do not leave their cards in the machine.)
9. System dispenses the requested amount of cash.
10. System prints receipt.
11. The use case ends.

Figure 1: Conventional use-case description to withdraw money from an ATM machine.

To illustrate how our approach contrasts the conventional UP approach we present in Figure 1 a transcription from a well-known example of a withdraw cash use-case description taken from the RUP introductory book (Kruchten, 1999). Figure 2 presents a participatory and user-centred alternative to describe the same use-case. Our approach takes advantage of the evident superiority of essential use-case narratives but adapt the concept to a diagrammatical representation to foster user participation and enable a better integration with the UML. Therefore, we combine essential use-case narratives with participatory driven task flows as proposed in the Bridge method (Dayton et al., 1998). At the far left of Figure 2 is the outcome of a typical participatory session where the *withdraw cash* use-case is worked-out with end-users through manipulation of index cards (activities) and sticky arrows (dependencies between activities). To the right of Figure 2 is the corresponding UML activity diagram that results from the translation of the sticky arrows and index cards produced in the participatory session.

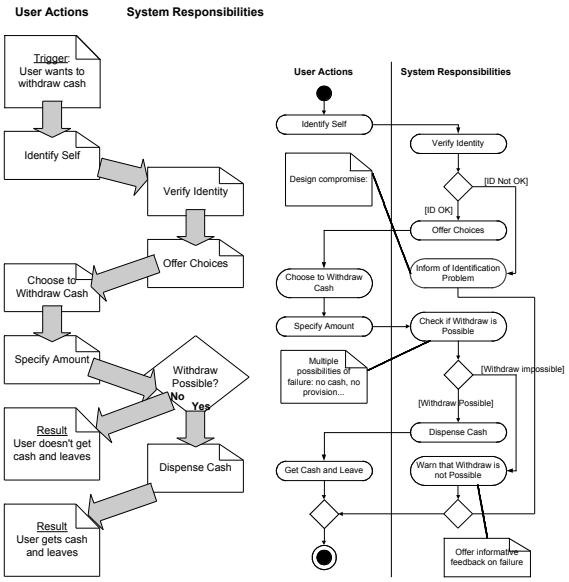


Figure 2: Participatory and user-centred use-case description to withdraw money from an ATM machine.

The above discussion clearly supports that conventional use-cases are not user-centred and don't express the requirements in a way that maximizes value towards end-users and business. Although this distinction seams marginal at the notational level, the impacts at the requirement level are considerable. Conventional use-cases (and in particular their descriptions) are complex and not suitable for cooperative participatory development. On the other hand, since use-cases drive development, a user-centred perspective is clearly fundamental to reduce the complexity of software systems. Driving development from a system-centric use-case perspective usually leads to the well-known problem of "featurism", that is, development is driven be functionalities that are not necessarily adding value to users and business.

Although the above discussion concentrated on contrasting conventional use-cases and a new approach based on essential use-cases, the same arguments could be applied to agile development methods. Even though agile methods don't endorse modelling, there is a specific reference to the role of "user stories" or similar requirement specification strategies in approaches like XP (Beck, 2000). There are important similarities with use-cases and their role in the UP and user-stories and their role in XP. On the one hand, both use-cases and user stories are used to prioritize development (user stories are used in XP to drive release planning). On the other hand, user stories are inherently user-centred, they are produced by XP customers and used throughout the lifecycle to drive the creation of acceptance tests (Beck, 2000).

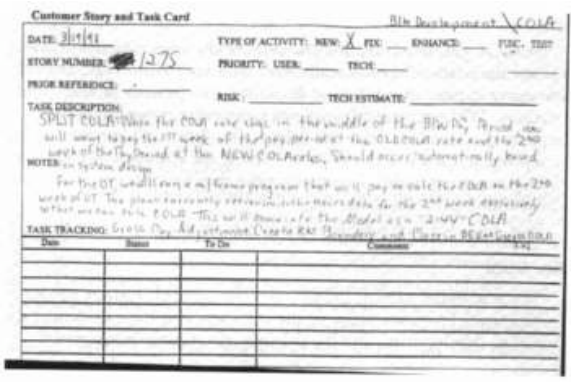


Figure 3: Example of a XP User Story (Beck, 2000, p. 88).

There is however an important distinction between UP and XP at this level. In UP use-cases drive the architecture, or in other words the architecture realizes the use-cases. In XP the architectural spike (and the system metaphor) are developed in parallel before release planning. Again careful inspection is required to analyse these subtle but important distinctions. Since XP doesn't promote modelling, user-stories are usually captured in natural language in the form of what is usually known as scenarios in the HCI field. Scenarios are an important technique to increase communication and discuss requirements. However, scenarios are the opposite of essential descriptions of use-cases. Scenarios are inherently full of assumptions and details, which increase their communication effectiveness (see Figure 3). However those same characteristics obstruct the creativity and reengineering required to generate simple and essential descriptions of requirements. Moreover, maintaining requirements in the form of natural language user stories is recognizable an overwhelming activity, in particular in agile environment and iterative development lifecycles. Despite the fact that there is strong evidence that XP and other agile methods work in practice, it is arguable that this effectiveness is related to the role of user-stories. The important role that XP's architectural spike plays in the overall lifecycle indicates that user-stories are mainly used to prioritize development and not to drive the software architecture.

3 Architecture-centric

The software architecture, in UP terms, refers to the most significant static and dynamic aspects of the system. The architecture grows out of use-cases but involves many other factors, such as platform restrictions, the existing reusable components, deployment considerations, legacy systems, and so on. Use-cases and the architecture correspond to the function and form of the software system and must evolve in parallel. The use-cases define the system functionalities and the architecture focuses on understandability, resilience to future changes and reuse. The UP promotes an architectural view (based on Kruchten's 4+1 model) sustained in parts of the different models that are architecturally significant.

The UP also promotes the boundary-control-entity pattern to describe the way use-cases are realized at a conceptual (architectural) level. The reason behind this partitioning of analysis classes into information (entity), behaviour (control) and interface (boundary) is to promote a structure more adaptable to changes by concentrating changes on different class stereotypes. This approach is conceptually similar, although at a different granularity level, to the PAC and MVC patterns. However, the UP pattern fails to map to the well-known physical architectural models of interactive systems (for instance Seeheim and Arch models (Nunes & Cunha, 2000)).

In (Nunes & Cunha, 2000) we propose an extension of the UP architectural pattern that includes additional dialogue and presentation dimensions to the original information structure of the UP boundary-control-entity pattern. Our approach aims at creating a new architectural framework more adapted to the requirements of interactive system development. Therefore, the boundary-control-entity pattern is extended with additional task and interaction space classes. The new class stereotypes introduced in this new proposal are:

task classes that are used to model the structure of the dialogue between the user and the system in terms of meaningful and complete sets of actions required to achieve a goal; and

interaction space classes used to represent the space within the user interface of a system where the user interacts with all the functions, containers, and information needed for carrying out some particular task or set of interrelated tasks.

Figure 4 illustrates the differences between our interactive system architecture and the conventional architectural descriptions used by the UP (and RUP). At the top of the figure is an analysis conceptual architecture provided in (Conallen, 1999) for a glossary web application. The architecture in the example supports three use-cases (*read glossary*, *search glossary* and *edit glossary entry*). As we can see from the top model in Figure 4, the conventional solution doesn't separate the user-interface from the internal functionality. For instance, *browse glossary* and *search glossary* are two control classes that contain both the business logic required to browse and search glossary entries, and the structure of use required to perform those tasks. Furthermore, the conventional model contains built-in assumption about the user-interface technology and the interaction styles used to implement the user-interface of the glossary application. For instance, the boundary classes *Home Page*, *Search form* and *Entry form* are obviously indicating a form-based

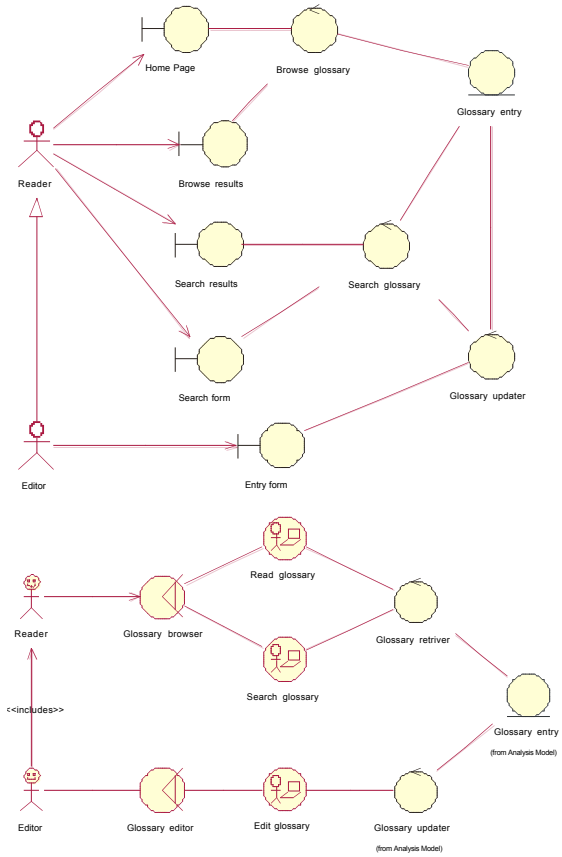


Figure 4: Application example of the UP architecture versus a revised proposal for interactive systems: top - transcription of a solution provided in (Conallen, 1999); bottom - the new solution based on (Nunes and Cunha, 2000)

interaction style and a web-based user-interface. Assuming technology constraints at the analysis level suggests that this architecture could not support a different technology or interaction style - for example a Java applet – therefore compromising the potential for reuse.

At the bottom of Figure 4 we illustrate a solution for the same problem based on the extended UI architecture. The new architecture clearly supports the well-known best practice of separation of concerns between the internal functionality and the user-interface specifics. The advantages are evident, not only the structure of use related to reading, searching and editing the glossary is contained in specific classes (the *read glossary*, *search glossary* and *edit glossary* task classes), but also the structure of the internal functionality becomes simpler because it doesn't have to contain the user-interface behaviour. Moreover, there is a clear separation between the presentation of the user-interface (*glossary browser* and *glossary editor* interaction space classes) and the structure of use. The resulting architecture is therefore, simpler (both in terms of the user-interface and the internal functionality), more robust (changes in the presentation of the user-interface don't impact the structure of use and the internal functionality, and vice-versa), and more reusable (the structure of use can be reused with respect to different implementation technologies). Finally our approach seamlessly maps different implementation architectures. For instance, assuming typical three-tier implementation architecture for a web-application, interaction spaces and task classes are candidates for client-side components, whereas control classes are candidates for the middleware tier and entities for the data tier.

Agile methods usually promote a simplified view of architecture. For instance, XP promotes an architectural spike with the main purpose of managing and assessing tough technical or design problems. This is usually accomplished by developing simple systems, which only address the problem under examination (architecturally significant) and ignore all other concerns – what is usually called a vertical throwaway prototype in HCI. The main goal of the architectural spike is to reduce the risk of a significant technical problem, or increase the reliability of a user story.

At the architectural level the communalities between agile methods and the UP are not evident. UP is architecture-driven but suggests a prescriptive approach towards architectural issues. The architecture in the UP emerges directly from the use-case model. The prescriptive nature of the conceptual architectural model implies that iterative and incremental development is preceded by an effort to analyse all the architectural significant classes emerging from the complete use-case model. That way UP aims to detect major technical problems a priori. Furthermore, there is no specific effort in trying to use the architecture to increase the reliability of requirements, something XP

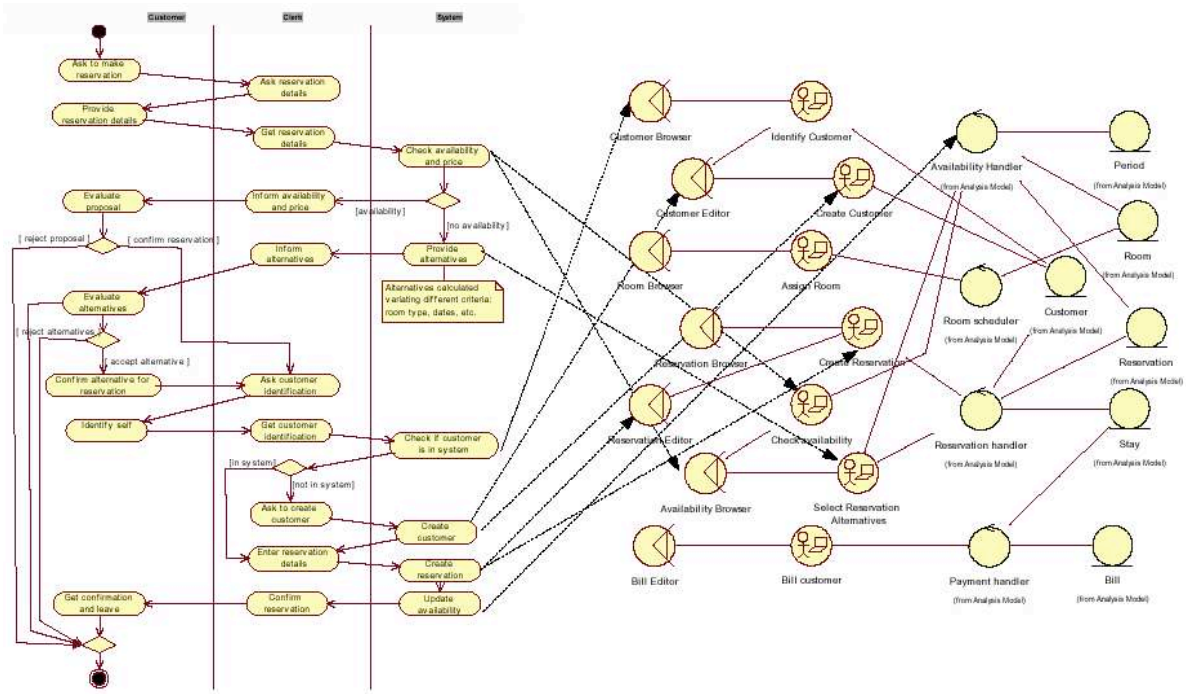


Figure 5: From task flow to the system architecture: an example for a hotel system (Nunes and Cunha, 2001).

emphasises.

There are two important issues when contrasting XP and UP at the architectural level. XP relies on vertical throwaway prototyping techniques to test potential architectural problems, both non-functional (technical problems) and functional (misconceptions emerging from user stories). The major problem with this approach is that foremost architectural problems emerge late during development when the different components of the system are integrated or implemented incrementally. XP assumes that developers can "guess" the main architectural problems, while also relying on refactoring techniques to simplify design as development progresses. This "wild guess" approach clearly contrasts the prescriptive architecture-centric nature of the UP. The second issue is related to the way both lifecycle models cope with architectural problems associated to misconceptions of user-requirements. XP emphasises this second issue, and relies on user stories, fast release cycles, refactoring and user evaluation to quickly identify and correct these problems. On the contrary, UP relies on defining a conceptual architecture model, based on prescriptive descriptions of requirements (the use-case model). Despite the fact that conventional use-cases are system-centric (and therefore the conceptual architecture - as we discussed in section 2), the UP practice assumes that architectural problems can only be identified when developers can argue over an orthogonal view of the foremost system components.

Another important aspect regarding the system architecture is the way architecturally significant classes are identified from the requirements descriptions. Both UP and XP promote that there should be a clear connection between the requirements (use-cases or user stories) and the conceptual architecture of the system. However, none of the approaches proposes a simple mapping between the two artefacts. In Figure 5 we illustrates how our approach supports the transition and traceability between task flows and the conceptual architecture. At the left hand side of Figure 5 is a task flow (UML activity diagram) expressing how a user makes a reservation in a hotel system. To the right of the activity diagram is a potential conceptual architecture that realizes the task flow. The arrows between those two diagrams represent mappings used to identify interaction spaces and tasks from the task flow. As we can see, the mapping is simple and straightforward.

4 Conclusion and Discussion

In this paper we tried to identify some important issues in modern software development that are clearly related to HCI and UCD.

Modern software development methods (such as UP and XP) are currently promoting some well-known UCD and HCI practices as a way to identify and manage requirements, drive and prioritize development and recognize architectural problems. In this paper we tried to look closer at the application of those UCD and HCI techniques in SE. We argue that discussing those issues at the lifecycle level is a pointless exercise - one that unfortunately has dominated the SE&HCI integration agenda for too long. In order to promote the integration of both fields we need to work closer and carefully examine how SE is using UCD and HCI techniques.

In this paper we presented the issue of driving software development from a UCD perspective. Both the UP and XP promote the UCD practice of involving users in the design and evaluation of the system. However, careful inspection of how that principle is achieved, in particular the specific techniques used in both UP and XP, reveals important misconceptions. We illustrate those problems by contrasting examples from the SE field with a new proposal that tries to selectively incorporate HCI techniques in a way that is not disruptive with current SE practice. The fact that the UML and UP promote use-case modelling is not per si a solution consistent with the UCD and HCI best practices. On the other hand, we cannot ignore that UML is a de facto standard language in the SE field. Therefore we present an approach that tries to incorporate participatory design and essential task flows in a way that can be successfully translate to UML constructs and efficiently managed by the tools available to the SE developers.

At the other extreme we have shown how the usage of scenarios (user stories) in XP is, not only inconsistent with agile development, but also misleading as an effective way to model user requirements. Scenarios play an important but focussed role in HCI that is usually disastrous when applied indiscriminately as a general requirements discovery technique. Moreover, natural language scenarios are particularly difficult to manage, above all with iterative methods and tools that currently proliferate in software shops.

This paper also discussed the architecture-centric nature of modern software development and how that issue relates to current UCD and HCI practice.

Software architecture plays an important role in modern SE, in particular since software products became ubiquitous, complex and involved a large number of developers and users. It was not our remit here to discuss software architecture, but instead how current SE methods build and convey the system architecture. Although both UP and XP concentrate heavily on the architecture there are substantial differences in the way those approaches tackle and convey the conceptual system architecture.

The UP relies heavily on use cases to drive the architecture, whereas XP promotes independent and concurrent creation of user-stories and the architecture spike. The specific examples provided in this paper reflect that UP promotes a system-centric view of the conceptual architecture that is clearly more complex and inconsistent with HCI practice. Contrasting the UP approach with our extended view of the conceptual architectural pattern, it is clear that the former doesn't provide separation of concerns between internal functionality and the user interface, lacks support for multiple platforms and usually leads to more complex conceptual architectures. In addition, conventional use-case descriptions don't provide guidance towards extracting architectural significant classes from user requirements.

At the other extreme XP relies on well-known HCI vertical prototypes to test potential architectural problems. By combining refactoring techniques, with frequent releases and user testing of story estimates, XP tries to tackle architectural problems but doesn't provide a high-level view of the system architecture. Again we witness heavy usage of some restricted HCI techniques, in particular scenario testing of prototypes, which could lead to potential problems when applied without specific knowledge. For instance, user testing of vertical prototypes involves problems because users have difficulties criticizing prototypes that "look and feel" like real systems. However the main problem with XP's approach is that there is no artefact that represents the conceptual system architecture and the connection between that architecture and the user-stories. Relying on code and refactoring as the sole representation for the system architecture is neglecting the importance of architecture at all.

Integrating HCI and SE is a longstanding issue. In this paper we try to take two steps forward by taking one step back: what drive software development?

References

Agile Alliance, (2003), www.agilealliance.org

Agile Manifesto (2003), www.agilemanifesto.org

Beck, K., (2000), Extreme programming explained: embrace change, *Addison Wesley Longman*, Reading, Mass.

Conallen, J. (1999), Building Web Applications with UML, *Addison-Wesley Longman*, Reading, Mass.

Constantine, L. L. and Lockwood, L. A. D. (1999), Software for use: a practical guide to the models and methods of usage-centered design, *Addison Wesley Longman*, Reading, Mass.

Dayton, T., McFarland, A. and Kramer, J. (1998), Bridging User Needs to Object Oriented GUI Prototype via Task Object Design, In User Interface Design (Ed, Wood, L. E.) *CRC Press*, Boca Raton - Florida - EUA, pp. 15-56.

Nunes, N. J. and Cunha, J. F., (2000), Wisdom – A UML based architecture for interactive systems, in Proceedings of DSV-IS'2000, *Springer-Verlag LNCS*, Limerick - Ireland.

Nunes, N. J. and Cunha, J. F., (2001) Whitewater Interactive System Development with Object Models, In Object Modeling and User Interface Design, (Ed. van Harmelen, M.) *Addison-Wesley Longman*, Reading, Mass.

UML for Interactive Systems: What is Missing

Philippe Palanque

LIIHS-IRIT, Université Paul Sabatier
118, route de Narbonne,
31062 Toulouse Cedex, France
palanque@irit.fr

Rémi Bastide

LIIHS-IRIT, Université Toulouse 1
Place Anatole France,
31042 Toulouse Cedex, France
bastide@irit.fr

Abstract
This position paper explores several design approaches put forward by HCI research, and investigates how the UML could or should support them.

1 Introduction

Over the years, the HCI research community has developed a set of techniques and approaches to the design of interactive systems. This trend of research puts the user at the center of the design process, and regards usability has the ultimate goal of design.
Meanwhile, the Software Engineering community was developing a set of notations and processes to support software design. The software system itself is at the center of this research, and software quality (dependability, reusability, etc.) is its ultimate goal. This research has matured into the UML [3], which is now an accepted standard.

It is unclear how these two trends of research can complement each other in order to keep the best of both worlds. This position paper explores this issue by looking at several design philosophies from the HCI side, and how much support is found for them in the UML (if any).

2 Engineering Interactive Systems

Several design philosophies emerge from HCI research, and can be classified according to the artefact that is put in the forefront:

- Abstraction first : architectures at the forefront
- Semantic first : metaphors at the forefront
- Implementation first : toolkits at the forefront
- Process first : User Centred Design
- Model first : Model-Based approaches

Each of these design philosophies is detailed below, along with its relationship to the UML

2.1 Abstraction first: architectures

The well known Seeheim and Arch [1] models are the result of an in-depth reflection on the very nature of interactive systems. They describe the typical structure of an IS in terms of abstract functionalities to be covered (e.g. Presentation, or Functional Core Interface).

These two models can be considered as an abstract software model of an Interactive System. They provide a very useful framework for the software designer to work in, since they provide a guide for a clear separation of concerns.

What kind of support can be found in the UML to support this architectural view of IS? A naïve idea is to use the notion of "package", provided by the UML as well as by most Object-Oriented programming languages. For instance, the topmost decomposition level of the software could mimic the Seeheim model by featuring four packages, Presentation, Dialogue, Functional Core Interface, and Functional core. This strategy, however, seems rather naïve since the "Dialogue" component of Seeheim cannot usually be conveniently isolated into a single package, but is usually distributed over several software components. Fined-grained dialogue (such as the answer to mouse click and key presses) is usually embedded into the code of widgets that are part of the "Presentation" toolkit. Coarse-grained dialogue is defined as the business logic of the application, usually in objects from the "Functional core" part.

Another approach to support the architectural view of IS would be a systematic use of the extension mechanisms of the UML such as stereotypes of tagged values.

2.2 Semantic first: metaphors

Designers often favour the use of metaphors to provide a unifying vision to their software artefact. The desktop or room metaphor has extensively been used for file managers, the "Village" metaphors is used for web sites or interactive TV, etc.

It is unclear how the UML could (or should) support the use of metaphors in the design. Maybe some support could be provided within the RUP to this end, but this requires further study.

2.3 Implementation first: toolkits

The HCI community has devoted extensive work and research on the design and implementation of UI toolkits, aiming at making the life of the software designer easier.
Although it would seem that UML is ideally suited to this kind of work, it is surprising to notice that no widely-used toolkit is described in terms of the UML (beyond providing an inheritance tree), while a UML description of a toolkit would be very useful for the software designer, especially interaction diagrams and StateCharts of the widget set. We can guess that 1) UML has not been used when designing these toolkits and 2) UML has been considered to costly for providing the documentation.

Has a result, most UI toolkits are hard to master and poorly documented, providing no or few rationale on their design.

2.4 Process first: iterative, UCD process

HCI people promote user-centred, iterative processes based on early evaluation of low-fidelity prototypes, incrementally evolved towards higher-fidelity prototypes. The UML users' guide Software development life cycle p. 33-34 states that "the UML is largely process-independent […]. However, […] you should consider a process that is 1) Use case driven, 2) Architecture-centric 3) Iterative and incremental". Prototyping per-se is not covered in the RUP. Proponents of eXtreme Programming (XP) also promote light-weight, incremental design, but the focus is on the incremental production and test of software, and the users are not usually formally involved in XP.

2.5 Model first: model-based approaches

Many models can be considered for model-based UI Design [4], [2]:

- Domain model (including data model)
- Task model and scenarios
- User model
- Platform model (link to toolkit and environment)
- Dialogue model (behaviour of the application I/O)
- Presentation model (appearance of the application)
- Application model (commands and data the application provides)
- …

Several of these models are supported in the UML:

- Domain model (including data model) are supported as class and object diagrams (organisational charts, …)
- Application model (commands and data the application provides) are the main focus of UML

Some models are only partially accounted for:

- Task model and scenarios can be described informally in UML use cases (Use Cases diagrams describe the **relationships** between use cases)
- Dialogue model state charts (state and events) and sequence diagrams

Several models are not considered at all in the UML

- User model
- Platform model (link to the toolkit and)
- Presentation model (appearance of the application)

3 Conclusions and Perspectives

For the team of methodologists (Rumbaugh, Jacobson, Booch) that shaped the UML, User Centred Design was not a central concern.

However it is clear that, in the present landscape of software development, Non-Interactive Systems are a special rare kind of system and not the other way round.
Our claim is that further work is needed to merge the achievement of HCI research into mainstream UML-based software design methodologies, and that this would be beneficial to the software industry as a whole.

4 References

1. Bass, Len, Little, R., Pellegrino, R., Reed, S., Seacord, R., Sheppard, S., and Szezur, M. R. "The Arch Model: Seeheim Revisited." *User Interface Developpers' Workshop*. Version 1.0 (1991)

2. Bodart, François, Hennebert, A-M., Leheureux J-M., Provot, I., and Vanderdonckt, Jean. "A Model-Based Approach to Presentation: A Continuum From Task Analysis to Prototype." *1st Eurographics Workshop on Design Specification and Verification of Interactive System (DSV-IS'94)*, Carrara, Italy. (1994)

3. Rational Software Corporation. *UML Summary*. 1.1 ed.1997.

4. Wilson, S., Johnson, P., Kelly, C., Cunningham, J., and Markopoulos, P. "Beyond Hacking: a Model Based Approach to User Interface Design. " *HCI'93*, Loughborough, U.K. Cambridge University Press (1993) 217-31.

Mind the Gap!
Software Engineers and Interaction Architects

Matthias Müller-Prove

Sun Microsystems, Sachsenfeld 4, 20097 Hamburg, Germany

mprove@sun.com

Abstract: This paper points out the different mentalities of software engineers and interaction architects, and shows how these can have a negative impact on the communication in product teams.

Keywords: software engineering, human-computer interaction, heterogeneous product teams, project management

1 Introduction

Software engineers and interaction architects need to cooperate with each other in order to create software products that work, and that are usable and useful for the target audience. A look at reality shows that the cooperation does not function as smoothly as it should. The cause for this can be on the engineer's side, or on the designer's – or on both sides. This paper identifies some differences in the mentalities that make it difficult to work together in one product team.

It needs to be said that successful product teams have many more components than just engineering and user interface design. To use Don Norman's metaphor: it is technology, marketing, and user experience that make up the three legs of a solid product (Norman, 1999, pp. 40). We also have to add product management, quality management, documentation, customer support, and upper management to the picture. Nevertheless, this paper focuses only on the relation between developers and HCI professionals.

2 Software Engineers

Software engineers live in their own world.[1] With few exceptions, they only focus on computers and themselves. A problem is seen as solved as soon as the algorithm is correct and the compiler does not come back with any syntax errors. As far as usability is concerned they are at best clueless, because usability was not part of their education. Many engineers fall into the trap of taking themselves as typical users by saying, "if I can use it then every other normal person can use it, too." – In fact, engineers are not normal or average people. Programming is a highly specialized profession that needs a lot of practice. The term software engineering reflects the scientific methods that are necessary to develop complex software systems. Sometimes software engineering even reaches the abstraction level of mathematics.

Engineers are the core members of a product development team. Nobody can do without them. And no one else is as difficult to replace. During the development process they collect so much detailed information about the product that it is even hard for another skilled engineer to continue the work at that point.

Sometimes engineers take advantage of their privileged position in order to steer the development process in a direction that seems most pleasant to them. They do not necessarily act intentionally. It is just too easy to become mixed up in this self-referential circle. Alan Cooper honored this position when naming his book *The Inmates Are Running the Asylum* (Cooper, 1999).

The attitude software engineers have is without a doubt fine in their own minds, and their standpoint is rarely challenged by other people in the product development process. Therefore, it is not surprising that engineers see no reason to take the suggestions of interaction designers seriously, since they usually just mean more work. Engineers block suggestions

[1] This section is based on *Usability im Unternehmen* (Müller-Prove, 2003, *Usability in Corporations*).

by responding that they are not feasible for technical reasons.

3 Interaction Architects

The term *interaction architect* was introduced by Bruce Tognazzini in one of his recent columns (Tognazzini, 2003). With this proposal he aims to give one unifying title to all those software designers, interaction engineers, human interface folks, user-experience flower-hoppers, or "whatever we are calling ourselves today." Interaction architects are in charge of designing the gestalt of the software product. They need to be creative in order to find solutions for product requirements that conform to specific style guides. Interaction architects tend to perceive themselves as artists.

On the other hand, *usability professionals* are those HCI experts that test the products in usability labs, and use other means like expert evaluations. They take the mockups and prototypes and criticize the usability of the artifacts. According to Tognazzini, the "usability professionals should report their results to the design team only, and then it's up to the design team how and when and where they want to incorporate those results (…)" (Dykstra-Erickson, 2000).

Now we are back at the point where interaction architects have to communicate with software engineers. For the interaction architect it is important to understand the engineer's attitude in order to be able to cope with the situation. This is less complicated for usability experts that have their roots in computer science than for those who come from graphics design or psychology. The latter per se do not speak the same language as the engineers. This additionally widens the gap between those groups and causes the usability experts to not be treated as fully accepted members of the product team.

Interaction architects need competence and substantiality. They have to take into account that some colleagues have no conception of the techniques of usability engineering while at the same time maintaining their point of view. This is a challenge for interaction architects. They have to continuously propagate the benefits of usability. They also have to explain the way in which they come to their design proposals in order to demonstrate a professional attitude. Showing highlights from usability lab studies can open the eyes of engineers. Also, supporting findings with quantitative data shifts the level of acceptance, because numbers are the native language of engineers.

A thorough approach is taken for instance by SAP (Latzina/Rummel, 2002). In-house training for engineers is conducted with the objective to build a conception of HCI methods. Role-plays are not just fun – they convince the engineers of the use of paper mockups for prototyping and other typical HCI techniques.

The conclusion is that it is necessary for both groups to gain respect for the different areas of expertise and have an understanding of each other's perspective. Otherwise, the fruits of beneficial cooperation will be a long time coming.

About the Author

Matthias Müller-Prove is a computer scientist. His reputation as a human-computer interaction architect is based on ten years of direct experience designing professional software interfaces for international operating companies. Before joining the StarOffice User Experience team at Sun Microsystems in 2002, he played a significant role in designing the user interface of Adobe GoLive.

References

Cooper, A. (1999). *The Inmates Are Running the Asylum*. Indianapolis: Macmillan Computer Publishing.

Dykstra-Erickson, E. (2000). Interview with Bruce Tognazzini. *Interactions, 7,* (2, Mar.), 41-46.

Latzina, M., & Rummel, B. (2002). Collaboration-Based Usability Training for Developers. In M. Herczeg, W. Prinz & H. Oberquelle (Eds*.), Mensch & Computer 2002* (pp. 285-291). Stuttgart: Teubner. http://mc.informatik.uni-hamburg.de/konferenzbaende/mc2002/konferenzband/mc2002_05_paper/mc2002-26-latzinarummel.pdf

Müller-Prove, M. (2003). Usability im Unternehmen. In S. Heinsen & P. Vogt (Eds.), *Usability praktisch umsetzen - Handbuch für Software, Web, Mobile Devices und andere interaktive Produkte*. München: Hanser.

Norman, D. A. (1998). *The Invisible Computer*. Cambridge, MA: The MIT Press.

Tognazzini, B. (2003). *It's Time We Got Respect*. AskTog. http://www.asktog.com/columns/057ItsTimeWeGotRespect.html

CLOSING THE GAPS:

Software Engineering and Human-Computer Interaction

Edited by Morten Borup Harning and Jean Vanderdonckt

This book comprises the proceedings of the Second IFIP WG2.7/13.4 Workshop on Closing the gap between Software Engineering (SE) and Human-Computer Interaction (HCI), sponsored by the International Federation for Information Processing (IFIP), which was held in Zürich, Switzerland in September 2003, 2-3. It contains research and position articles authored by practitioners and researchers concerned with integrating HCI techniques in SE and vice versa.

Almost half of software in systems being developed today and thirty-seven to fifty percent of efforts throughout the software life cycle are related to the system's user interface. For this reason issues and methods from the field of HCI affect the overall process of SE tremendously. Yet despite strong motivation amongst organisations to practice and apply effective SE and HCI methods there still exist major gaps of understanding both between suggested practice, and how software is actually developed in industry, and between the best practices of each of the fields. There are major gaps of communication between the HCI and SE fields: the methods and vocabulary being used in each community are often foreign to the other community. As a result, product quality is not as high as it could be, and (avoidable) re-work is often necessary. In addition, SE methods and techniques are often perceived by HCI specialists as tools that are only reserved to computer scientists and of little or no relevance to HCI. And vice versa: HCI contents are often perceived by software engineers as after-thoughts or side-tools that do not necessarily affect the quality of software. For instance, no methodologies in the domain of object-oriented programming offer explicit support for HCI and existing HCI methods are integrated in development practices in a way that is more opportunistic than systematic.

Please refer to the workshop homepage (hosted by IFIP WG2.7/13.4) for further details: http://www.se-hci.org/bridging/interact The 7th International Conference on Human-Computer Interaction, Interact'2003 (Zürich, 1-5 September 2003) hosted for the workshop. http://www.interact2003.org

Morten Harning is the Chief Design Officer at Open Business Innovation responsible for the overall design decisions both regarding the technical architecture and the non-technical design (such as user interface issues). Morten Harning is a member of the IFIP Working Group 2.7/13.4 on User Interface Engineering and is the founder of and currently chairing the danish special interest group on human-computer interaction - SIGCHI.dk.

Jean Vanderdonckt is Professor at the School of Management (IAG), Université catholique de Louvain (UCL), Belgium where he leads the Belgian Laboratory of Human-Computer Interaction (BCHI). See http://www.isys.ucl.ac.be/bchi. He is a co-founder of the belgian special interest group on human-computer interaction – www.belchi.be