Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 1506–1516
The Journal of Systems and Software
www.elsevier.com/locate/jss

Analysing the impact of usability on software design ☆

Natalia Juristo [a], Ana M. Moreno [a,*], Maria-Isabel Sanchez-Segura [b]

[a] *School of Computing, Universidad Politécnica de Madrid, Campus de Montegancedo s/n, 28660 Boadilla del Monte, Madrid, Spain*
[b] *Computer Science Department, Universidad Carlos III de Madrid, Avda. De la Universidad, 30, 28911 Leganés, Madrid, Spain*

Received 20 July 2006; received in revised form 3 January 2007; accepted 5 January 2007
Available online 20 January 2007

Abstract

This paper analyses what implications usability has for software development, paying special attention to the impact of this quality attribute on design. In this context, the aim is twofold. On the one hand, we intend to empirically corroborate that software design and usability are really related. This would mean that this, like other quality attributes, would need to be dealt with no later than at design time to develop usable software at a reasonable cost. On the other hand, we present a possible quantification, calculated from a number of real applications, of the effect of incorporating certain usability features at design time.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Software usability; Software design

1. Introduction

Software usability is a quality attribute listed in a number of classifications (IEEE, 1998; ISO9126, 1991; Boehm, 1978). Although it is not easy to find a standard definition of usability, Nielsen gave one of the most well-known descriptions concerning the learnability and memorability of a software system, its efficiency of use, its ability to avoid and manage user errors, and user satisfaction (Nielsen, 1993). Similarly, ISO 9241-11 (ISO9241, 1998) 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 specific context of use". In short, usability is also generally referred to as "quality in use" (ISO14598, 1999).

A number of studies have pointed out a wide range of benefits stemming from usability (Trenner, 1998; Battey, 1999; Donahue, 2001; Griffith, 2002; Black, 2002): it improves productivity and raises team morale, reduces training and documentation costs, improves user productivity, increases e-commerce potential, etc. Additionally, and contrary to what one might think, the cost/benefit ratio of usability is highly worthwhile. Donahue states that every dollar spent on usability offers a return of $30.25 (Donahue, 2001). There are also studies for e-commerce sites that show that a 5% improvement in usability could increase revenues by 10–35% (Black, 2002).

Accordingly, usability is moving up the list of strategic factors to be dealt with in software development. For instance, IBM is convinced that usability "makes business effective and efficient and it makes business sense" (IBM, 2005). In the same line, the Boeing Co. changed the way it buys software making a product's usability a fundamental purchasing criterion (Thibodeau, 2002).

In spite of the relevance of usability in software development it is still insufficient in most software systems (Seffah and Metzker, 2004; Bias and Mayhew, 2005).

The most widespread view in the field of software engineering (SE) is that usability is chiefly related to the user interface (UI) (Seffah and Metzker, 2004; Folmer et al., 2004). In other words, usability mainly affects the UI and not the core of the system. Therefore, the good design practice that separates the UI from the core functionality would be sufficient to support the development of a usable

☆ This research was funded by the Spanish Ministry of Education – Project TIN 230005-00176.
* Corresponding author. Tel.: +34 91 3366929; fax: +34 913366917.
E-mail addresses: natalia@fi.upm.es (N. Juristo), ammoreno@fi.upm.es (A.M. Moreno), misanche@inf.uc3m.es (M.-I. Sanchez-Segura).

software system. This view means that dealing with usability could be put off until the later stages of software system development (commonly during system evaluation), as the changes required to improve this quality attribute should not involve a lot of rework.

Voices questioning this hypothesis started to be heard some time ago. For example, authors like Perry and Wolf claimed that usability issues place static and dynamic constraints on several software components (Perry and Wolf, 1992). On the other hand, Bass and John stated more recently that separating the user interface from the core functionality is not sufficient to support usability and that other tactics must be put in place in the software design if a usable product is to be delivered to the customer (Bass and John, 2003).

If this relation between usability and software design is confirmed, the rework cost to achieve an acceptable level of usability would be much higher than expected according to the hypothesis of separation: the improvements in system usability would involve modifications beyond the interface, specifically, to the design of some software components and, therefore, to its core. If this is the case, usability should be dealt with earlier in the development process in order to define and evaluate its impact on design as soon as possible. Notice that this approach is consistent with the tendency in SE to carefully consider quality attributes early on in the development process (Barbacci et al., 2003). This strategy has already been applied to other quality attributes like performance, modifiability, reliability, availability and maintainability. A number of authors have proposed techniques to deal with these attributes at architectural design time (Klein, 1999; Bass, 1999; Eskenazi et al., 2002; Bosch and Lundberg, 2003).

As we will detail later, some authors have already illustrated, albeit informally, a possible relationship between usability and architectural design (Bass and John, 2003; Folmer et al., 2004). However, the knowledge of this relationship needs to be further investigated to be able to deal with usability during software design. On the one hand, there is a need to move from illustrations to facts to confirm whether or not usability really does have an impact at this development time. And, on the other, this impact has to be analysed and quantified.

This paper aims to contribute in this direction. To this end, Section 2 analyses what different types of impact usability heuristics and guidelines stated in the Human Computer Interaction (HCI) literature are likely to have on a software system. As we will see, some usability features appear to have some effect on software design. The next step then is to examine what sort of effect this is. With this aim in mind, we have conducted a study of several real systems into which we have incorporated the usability features in question. Section 3 details, for one such usability feature, our analysis in terms of new classes, methods and relations derived from adding this usability feature at design time. As a result of this analysis we are able to demonstrate that usability really is not confined to the system interface and can affect the system's core functionality. With the goal of quantifying as far as possible the effect of usability features with an impact on design, Section 4 discusses the data gathered from applying a number of usability features in the development of several real systems. We used these data to demonstrate the relationship between usability and software design and to get an informal estimation of the implications of building these features into a system. Finally, Section 5 reviews the related work there is on the subject and describes how our research contributes to a further study of usability within the SE framework.

In sum, this work tries to demonstrate that particular usability issues have a real impact on software design. Consequently, (1) although the separation strategy might be valid for addressing other quality attributes like maintainability, it does not guarantee easy improvements in usability; and (2) usability needs to be considered from the former development moments to avoid design rework.

2. What impact does usability have on software development?

The field of HCI has traditionally dealt with the usability of software systems. The HCI literature therefore includes many recommendations on how to improve software system usability. Different authors refer to such recommendations differently: design heuristics (Nielsen, 1993), usability rules (Shneiderman, 1998), usability principles (Preece et al., 1994; Constantine and Lockwood, 1999), ergonomics principles (Scapin and Bastien, 1997), usability patterns (Tidwell, 2005; Welie, 2003; Brighton, 1998), etc. Some authors propose tens of recommendations (Brighton, 1998) and others close to 50 (Welie, 2003). Unfortunately, the HCI community has not reached agreement on the generation of an accepted list of usability recommendations. This difference of opinion is an obstacle in the way of software engineers using appropriate guidelines for building usable software. A software developer set to consider HCI suggestions would have to: (1) catalogue the bibliography about all existing recommendations; (2) analyse and compare recommendations to get a set of agreed recommendations; and (3) classify and typify recommendations (some are guidelines on how to relate to the user, other are features that the system should contain, etc.). To do all this, the developer would need a wide range of knowledge related to psychology, ergonomics, linguistics, etc. proper to the HCI field. This type of knowledge is far removed from the expertise of software engineers who are versed in techniques, methods and tools for building applications.

While experts in HCI are reaching agreement, the authors have reviewed different types of usability recommendations in the HCI literatures (Nielsen, 1993; Shneiderman, 1998; Preece et al., 1994; Constantine and Lockwood, 1999; Hix and Hartson, 1993; Rubinstein and Hersh, 1984; Heckel, 1991; Scapin and Bastien, 1997; Tidwell, 2005; Welie, 2003; Brighton, 1998; Coram and Lee, 1996;

Laasko, 2003) with the aim of analysing its effect in a software system from a SE view. Readers should not expect this analysis conducted by software engineers to provide an exhaustive classification of HCI recommendations or compile a conclusive list; it should identify what type of impact the inclusion of certain usability recommendations would have on development.

Let's detail before the concept of usability. Usability deals with the whole user–system interaction, not just the user interface. The user interface is the visible part of the system (buttons, pull-down menus, check-boxes, background colour, etc.). Interaction is a wider concept. Interaction is the coordination of information exchange between the user and the system. For example, suppose that the feedback feature is to be built into the system to improve usability. In this case, system operations have to be designed to allow information to be frequently sent to the user interface to keep the user informed about the current progress of a task. This information could be displayed for the user by different means (percentage-completed bar, a clock, etc.). These means are interface or presentation issues. But this usability feature has another more important part: the pieces of code in charge of getting the information from the system operations and which decide how often this information needs to be refreshed, which information to display depending on the operation, etc. These issues are not part of the interface, but are related to the user–system interaction.

After examining the different usability recommendations provided by the HCI literature we have identified three different categories, depending on their effect on software development:

• *Usability recommendations with impact on the UI*. These are HCI recommendations that affect system presentation through buttons, pull-down menus, check-boxes, background colour, fonts, etc.

The impact of these features on software development can be illustrated by an example. Suppose we have a developed system and we want to change the colour of one of the windows to highlight certain aspects of interest for the user (Tidwell, 2005; Constantine and Lockwood, 1999) or the text associated with certain buttons so that user can get a clearer idea of what option to select (Hix and Hartson, 1993; Nielsen, 1993). These changes that help to improve the software usability would be confined to altering the value of some source code variable related to the window or buttons. In other words, building these recommendations into a system involves slight modifications to the detailed UI design, and no other part of the system needs therefore to be modified. If good design practices have been followed, this change would be confined to the interface component or subsystem, having no impact on the system core.

• *Usability recommendations with impact on the development process*. These are recommendations that are not confined to one or more software system products, but can only be taken into account by modifying the development process, that is changing the techniques, activities and/or products used during such process. These recommendations include guidelines to encourage all user–system interaction to take place naturally and intuitively for the user. This is the goal of usability recommendations like natural-mapping (Constantine and Lockwood, 1999), develop a user conceptual model (McKay, 1999), know your user (Heckel, 1991; Hix and Hartson, 1993), involve the user in software construction (Heckel, 1991; Hix and Hartson, 1993; Constantine and Lockwood, 1999), reduce the user cognitive load (Preece et al., 1994), etc., which intend the user to be able to confidently predict the effect of his/her actions with the system.

Note that to be able to incorporate such recommendations the software development process needs to be modified by making it more user centred. This involves including, for example, more powerful elicitation techniques, probably coming from other disciplines like HCI (to satisfy the "natural-mapping" or "know the user" recommendations); determining when to get the user involved in the development process, apart from during traditional elicitation, describing the techniques and products to be used for this aim (to comply with "involve the user in the software construction"), etc.

There are already a few proposals in the HCI literature (Mayhew, 1999; ISO18529, 2000) aimed at modifying the software process with the aim of explicitly satisfying the usability recommendations in this group. These proposals have, however, been stated from the HCI field's perspective of software development, far removed on many points (terminology, development approach, etc.) from SE's view. Therefore, they need to be tailored for use in the SE process.

• *Usability recommendations with impact on design*. These are recommendations involving building certain functionalities into the software to improve user–system interaction. For example, features like cancel an ongoing task (Brighton, 1998), undo a task (Shneiderman, 1998; Welie, 2003; Laasko, 2003), receive feedback on what is going on in the system (Hix and Hartson, 1993; Shneiderman, 1998; Tidwell, 2005) or adapt the software functionalities to the user profile (Rubinstein and Hersh, 1984; Welie, 2003).

Suppose that we want to build the *cancel* functionality for specific commands into an application. HCI experts mention "provide a way to instantly cancel a time-consuming operation, with no side effects" (Tidwell, 2005)" or "if the time to finish the processing of the command will be more than 10 s, along to the feedback information, provide a cancel option to interrupt the processing and go back to the previous state (Nielsen, 1993)". To satisfy such requirements the software system must at least: gather information (data modifications, resource usage, etc.) that allow the system to recover the status prior to a command execution; stop command execution; estimate the time to cancel and inform the user of progress in cancellation; restore the system to the status before the cancelled command; etc.

Table 1
Preliminary list of usability features with impact on software design

Functional usability features	Goal
Feedback (Tidwell, 1999; Brighton, 1998; Coram and Lee, 1996; Welie, 2003; Tidwell, 2005; Nielsen, 1993; Constantine and Lockwood, 1999; Shneiderman, 1998; Hix and Hartson, 1993; Heckel, 1991)	To inform users about what is happening in the system
Undo (Tidwell, 2005; Welie, 2003; Brighton, 1998)	To undo system actions at several levels
Cancel (Tidwell, 2005; Brighton, 1998; Nielsen, 1993)	To cancel the execution of a command or an application
Form/field validation (Tidwell, 2005; Brighton, 1998; Shneiderman, 1998; Hix and Hartson, 1993; Rubinstein and Hersh, 1984; Constantine and Lockwood, 1999)	To improve data input for users and software correction as soon as possible
Wizard (Welie, 2003; Tidwell, 2005; Constantine and Lockwood, 1999)	To help do tasks that require different steps with user input
User expertise (Tidwell, 1999; Welie, 2003, Hix and Hartson, 1993; Heckel, 1991)	To adapt system functionality to users' expertise
Multi level help (Tidwell, 2005; Welie, 2003; Nielsen, 1993)	To provide different help levels for different users
Use of different languages (Nielsen, 1993)	To make users able to work with their own language, currency, ZIP code format, date format, etc.
Alert (Brighton, 1998; Welie, 2003)	To warn the user about an action with important consequences

This means that, apart from the changes that have to be made to the UI to add the cancel button, specific components must be built into the software design to deal with these responsibilities. Therefore, building the cancel usability feature into a finished software system involves a substantial amount of design rework rather than minor changes to the UI (as is the case of the usability recommendations with impact on the UI). Therefore, it should not be dealt with at the end of the development process but well in advance like any other functionality. The same applies to other usability recommendations that represent specific functionalities to be incorporated into a software system, their inclusion should be addressed early in the development process.

We have termed these HCI recommendations *functional usability features* (FUF), as most of these heuristics/rules/principles make recommendations on functionalities that the software should provide for the user. Table 1 shows the most representative FUFs that can be foreseen to have a crucial effect on system design. As we have seen with *cancel*, this effect is inferred from the detailed descriptions of the FUFs provided by HCI authors (the respective references are also listed in Table 1).

In the remainder of the paper we will focus on this last category of usability features, that is, FUFs, since, as already mentioned, we would like to examine the possibility of dealing with usability at software design time like other quality attributes. To do this, the following section illustrates the impact derived from incorporating one type of FUF on a particular design. This impact will be quantified in further depth in Section 4 for all FUFs in Table 1.

3. Analyzing the relationship between usability and design

In order to confirm what the relationship between FUFs and software design is, we have worked on a number of real development projects carried out by UPM Master in Software Engineering students as part of their MSc dissertations from 2004 to 2005. Students originally developed the respective systems without any FUFs. These designs were then modified to include the FUFs listed in Table 1.

This section discusses an example illustrating the incorporation of one FUF on a system developed to manage online table bookings for a restaurant chain. The FUF addressed is one particular kind of *feedback*, *system status feedback* (Tidwell, 1999; Coram and Lee, 1996). This usability feature involves the system notifying the user of any change of state that is likely to be of interest. HCI experts recommend that the notification should be more or less striking for the user depending on how important the change is. The status changes that the client considered important referred to failures during the performance of certain system operations, owing either to runtime faults, insufficient resources or problems with devices or external resources (specifically Internet).

Fig. 1 illustrates part of the UML class model for this system, including the classes involved in the restaurant table booking process only. The original classes designed without considering *system status feedback* are shown on a white background in the diagram and the classes derived from the later inclusion of this feature on a grey background. Let us first analyse this diagram without taking into account the *system status feedback*. The Interface class represents the part of the bookings terminal with which the user interacts to book a table at the restaurant over the Internet. The Restaurants-Manager and Reservations-Manager are two classes specialized in dealing with restaurants and bookings, respectively. They have been added to make the system more maintainable by separating the management processes so that they can be modified more efficiently in the future. The Reservations class accommodates the information related to each booking that is made at the restaurant, where the restaurant is represented by the Restaurant class and each of the tables at this restaurant that are not taken are identified by the Table class.

According to the previous definition of *system status feedback*, its inclusion into a software system involves the following specific responsibilities:

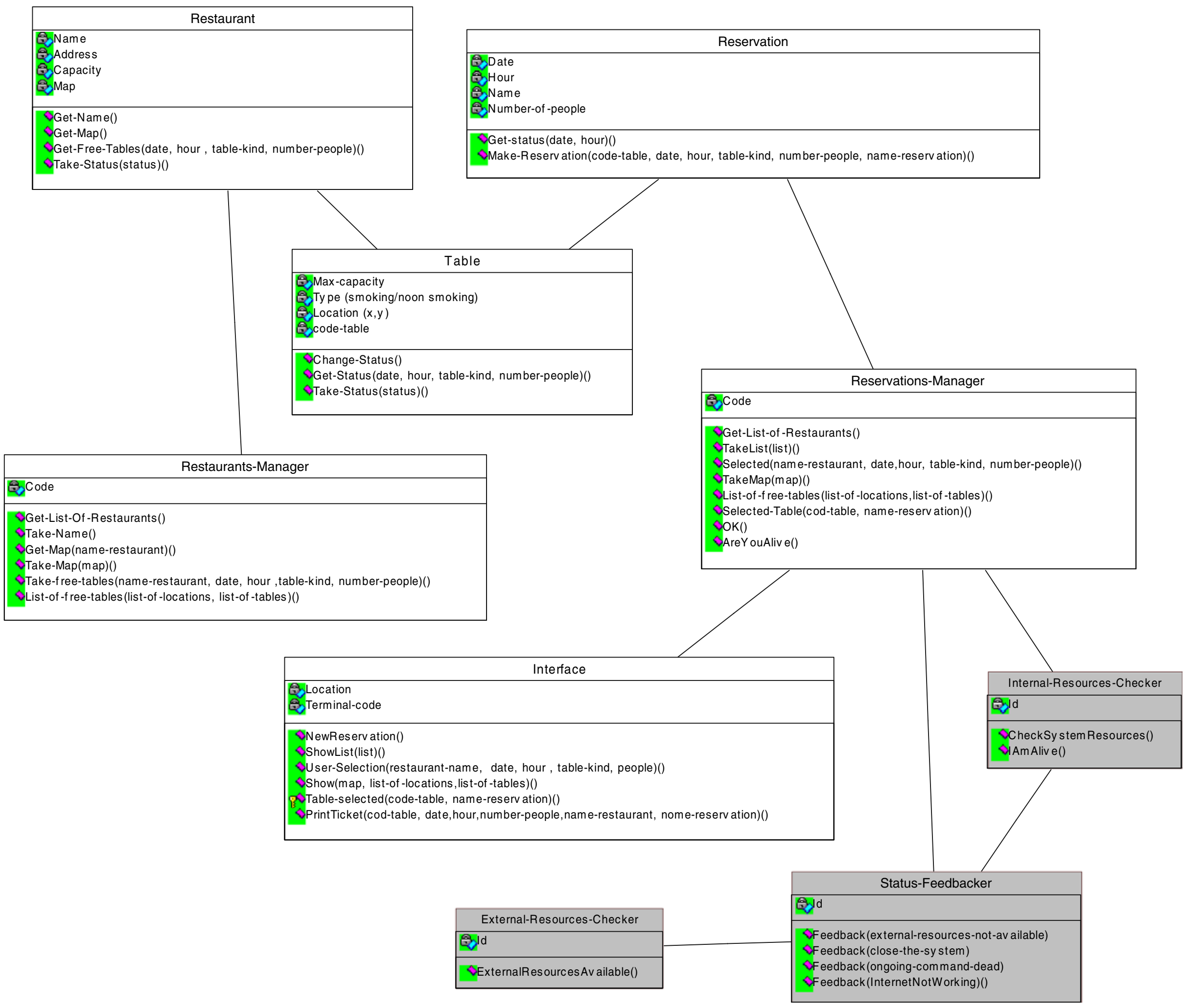
Restaurant
Name
Address
Capacity
Map
Get-Name()
Get-Map()
Get-Free-Tables(date, hour , table-kind, number-people)()
Take-Status(status)()
Reservation
Date
Hour
Name
Number-of -people
Get-status(date, hour)()
Make-Reserv ation(code-table, date, hour, table-kind, number-people, name-reserv ation)()
Table
Max-capacity
Ty pe (smoking/noon smoking)
Location (x,y)
code-table
Change-Status()
Get-Status(date, hour, table-kind, number-people)()
Take-Status(status)()
Reservations-Manager
Code
Get-List-of -Restaurants()
TakeList(list)()
Selected(name-restaurant, date,hour, table-kind, number-people)()
TakeMap(map)()
List-of -f ree-tables(list-of -locations,list-of -tables)()
Selected-Table(cod-table, name-reserv ation)()
OK()
AreY ouAliv e()
Restaurants-Manager
Code
Get-List-Of -Restaurants()
Take-Name()
Get-Map(name-restaurant)()
Take-Map(map)()
Take-f ree-tables(name-restaurant, date, hour ,table-kind, number-people)()
List-of -f ree-tables(list-of -locations, list-of -tables)()
Interface
Location
Terminal-code
NewReserv ation()
ShowList(list)()
User-Selection(restaurant-name, date, hour , table-kind, people)()
Show(map, list-of -locations,list-of -tables)()
Table-selected(code-table, name-reserv ation)()
PrintTicket(cod-table, date,hour,number-people,name-restaurant, nome-reserv ation)()
Internal-Resources-Checker
Id
CheckSy stemResources()
IAmAliv e()
Status-Feedbacker
Id
Feedback(external-resources-not-av ailable)
Feedback(close-the-sy stem)
Feedback(ongoing-command-dead)
Feedback(InternetNotWorking)()
External-Resources-Checker
Id
ExternalResourcesAv ailable()

Fig. 1. Class diagram for restaurant management with system status feedback feature.

- R1: The software should be able to listen to commands under execution, because they can provide information about the system status. If this information is useful to the user, the system should be able to pass this information on as, when and where needed.
- R2: As the system may fail, the software should be able to ascertain the state of commands under execution, because users should be informed if the command is not working owing to outside circumstances. The system should be equipped with a mechanism by means of which it can check at any time whether a given command is being executed and, if the command fails, inform users that the command is not operational.
- R3: The software should be able to listen to or query external resources, like networks or databases, about their status, to inform the user if any resource is not working properly.
- R4: The software should be able to check the status of the internal system resources and alert users in advance to save their work, for instance, if the system is running short of a given resource and is likely to have to shut down.

The following changes (highlighted on a grey background) had to be made to the design illustrated in Fig. 1 to deal with these responsibilities:

- Three new classes:
  - Internal-Resource-Checker, responsible for changes and for determining failures due to internal system resources like memory, etc.
  - External-Resource-Checker, responsible for changes and determining failures due to external system resources like networks, databases, etc.
  - Status-Feedbacker, responsible for providing the client with information stating the grounds for both changes and system failures in the best possible format and using the most appropriate heuristic in each case.
- Five new methods:
  - ExternalResourcesAvailable, to determine whether the external resources are available.
  - CheckSystemResources, to determine whether the system has the resources it needs to execute the booking operation for the restaurant.

- ◦ AreYouAlive, to be able to ask whether the command under execution is still running correctly.
- ◦ IAmAlive, for the Reservations-Manager to be able to tell the system that it is still running, that is, that it is alive.
- ◦ Feedback, to tell the user of any changes or failures identified in the system while the table is being booked at the restaurant.

Table 2
Correspondence between responsibilities and static structure diagram classes

Class Name	Responsibility
Reservations-Manager	R1, R2
Internal-Resources-Checker	R1, R2, R4
External-Resources-Checker	R3
Status-Feedbacker	R1, R2, R3, R4

- Four new associations between:
  - ◦ Reservations-Manager and StatusFeedbacker, so that the ReservationsManager class can tell the StatusFeedbacker about changes or system failures that are relevant for the user and about which the user is to be informed.
  - ◦ Internal-Resource-Checker and Status-Feedbacker and External-Resource-Checker and Status-Feedbacker, to be able to tell the Status-Feedbacker class that a change or failure has occurred.
  - ◦ Reservations-Manager and Internal-Resources-Checker, so that the Internal-Resources-Checker can check that the reservation process is still active and working properly.

Table 2 summarises the classes responsible for satisfying each of the identified responsibilities.

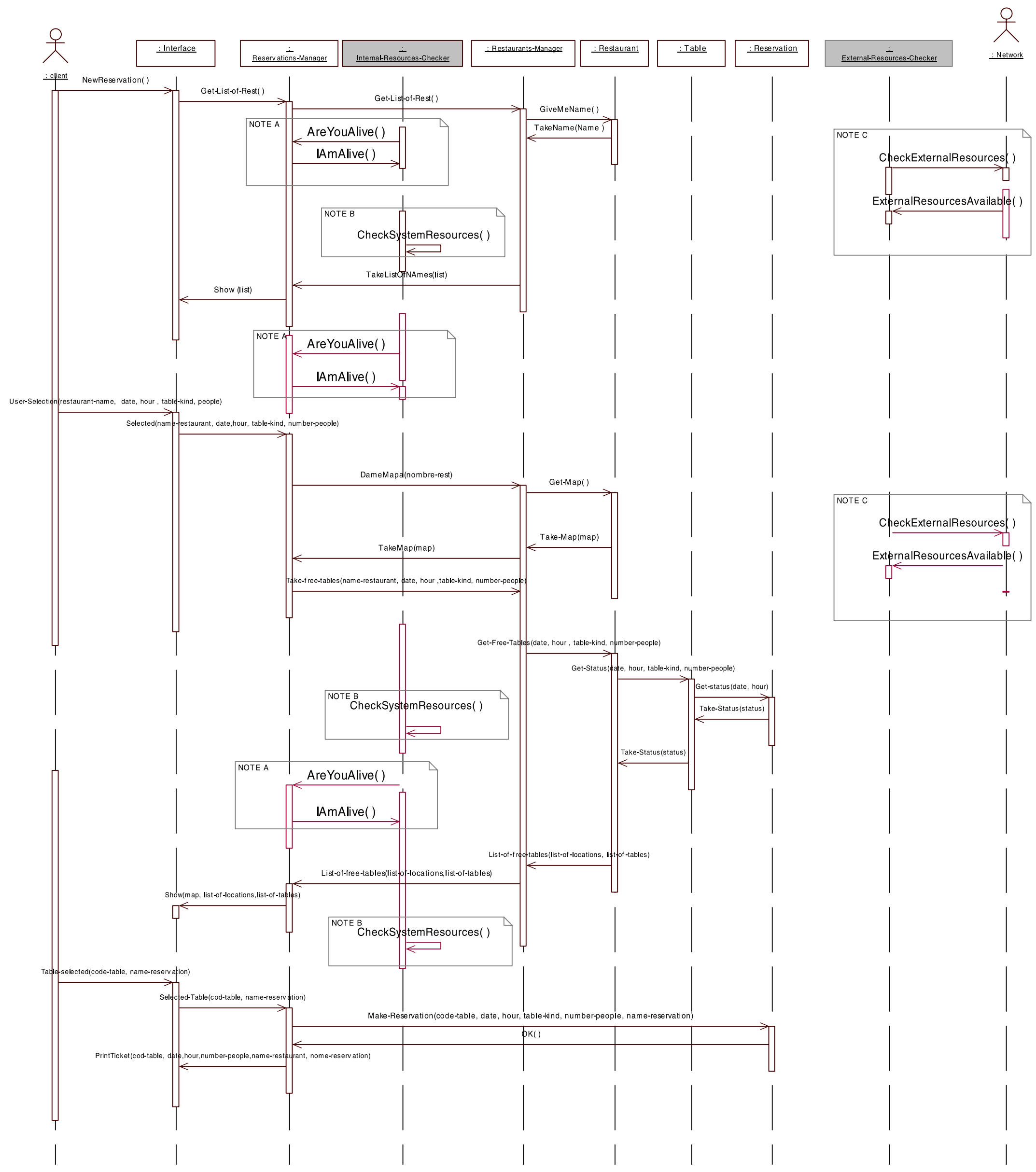
: client
: Interface
: Reservations-Manager
: Internal-Resources-Checker
: Restaurants-Manager
: Restaurant
: Table
: Reservation
: External-Resources-Checker
: Network
NewReservation( )
Get-List-of-Rest( )
Get-List-of-Rest( )
GiveMeName( )
TakeName(Name)
NOTE A
AreYouAlive( )
IAmAlive( )
NOTE C
CheckExternalResources( )
ExternalResourcesAvailable( )
NOTE B
CheckSystemResources( )
TakeListOfNAmes(list)
Show (list)
NOTE A
AreYouAlive( )
IAmAlive( )
User-Selection(restaurant-name, date, hour , table-kind, people)
Selected(name-restaurant, date,hour, table-kind, number-people)
DameMapa(nombre-rest)
Get-Map( )
NOTE C
CheckExternalResources( )
ExternalResourcesAvailable( )
Take-Map(map)
TakeMap(map)
Take-free-tables(name-restaurant, date, hour ,table-kind, number-people)
Get-Free-Tables(date, hour , table-kind, number-people)
Get-Status(date, hour, table-kind, number-people)
Get-status(date, hour)
Take-Status(status)
NOTE B
CheckSystemResources( )
Take-Status(status)
NOTE A
AreYouAlive( )
IAmAlive( )
List-of-free-tables(list-of-locations, list-of-tables)
List-of-free-tables(list-of-locations,list-of-tables)
Show(map, list-of-locations,list-of-tables)
NOTE B
CheckSystemResources( )
Table-selected(code-table, name-reservation)
Selected-Table(cod-table, name-reservation)
Make-Reservation(code-table, date, hour, table-kind, number-people, name-reservation)
OK( )
PrintTicket(cod-table, date,hour,number-people,name-restaurant, nome-reservation)

Fig. 2. Sequence diagram for reservations management with System Status Feedback feature.

Fig. 2 illustrates the sequence diagram for the case in which the user books a restaurant table over the Internet. Like Fig. 1, this diagram shows the original classes designed without taking into account *system status feedback* on a white background, whereas as the classes derived from the inclusion of this usability feature are shaded grey. Let us analyse this diagram first without taking into account the information entered as a result of *system status feedback.* Looking at the sequence diagram, it is clear that the Client actor interacts with the Interface class to start to book a restaurant table. As of this point, the Reservations-Manager takes over to request the names of available restaurants from which the client is to select one and enter the date, time and table type (smoking or non-smoking).The system can use this information to check what tables have already been booked at the restaurant and determine which meet the conditions specified by the client. Once the system has gathered what information there is about tables, it displays a map of the restaurant showing the position of the vacant tables from which the user can select the one he or she likes best. After the user has selected the restaurant table, the system proceeds to materialize the booking providing the client with a booking code stating the table booked and giving him or her the option to print a ticket stating the booking information.

After including *system status feedback*, the following checks need to be run: (i) the system has not stop while it was making the booking (see UML NOTE A); (ii) the system has enough resources, memory, etc. (see UML NOTE B) for as long as it took to book the table, and (iii) the Internet connection has worked properly (see UML NOTE C). As you can see, the above checks are done at different times throughout the booking process and, therefore, the interactions represented by UML NOTES A, B and C appear more than once.

As shown in Fig. 3, if an external resource, in this case the network, failed, the External-Resources-Checker would take charge of detecting this state of affairs (as no ExternalResourceAvailable message would be received) and it would alert the user through the Status-Feedbacker. If the shaded classes were omitted from this diagram, the system would not be able to detect the network failure. Note that a skilful programmer could have implemented this operation with a time-out in anticipation of a possible network failure. In this case, however, the potential usability of the application would depend on the programmer's skill, without guarantee of it working properly. Explicitly incorporating the components for this usability feature at design time provides some assurance that it will be included in the final system.

The above example illustrates what impact the inclusion of the *system status feedback* feature has on system design (and therefore on later system implementation). Other designers could have come up with different, albeit equally valid designs to satisfy the above-mentioned responsibilities. For example, a less reusable class model might not have included new classes for this type of feedback, adding any new methods needed to existing classes. In any case, the addition of this new feature to the system can be said to involve significant changes at design time.

Note that this example only shows the effect of this usability feature for a specific functionality (table booking). This effect is multiplied if this feature is added to the other system functionalities, and, as we will discuss in next section, even more so if we consider all the other FUFs.

With the aim of analysing what effect FUFs have on design in more detail, the next step should be trying to quantify the impact of each feature. This would help to ascertain whether all the FUFs are equally complex and, therefore, have the same level of impact on design. The next section shows the results of this study.

4. Quantification of the impact level of functional usability features on design

Similarly to the *system status feedback*, we have incorporated the other FUFs listed in Table 1 into the different

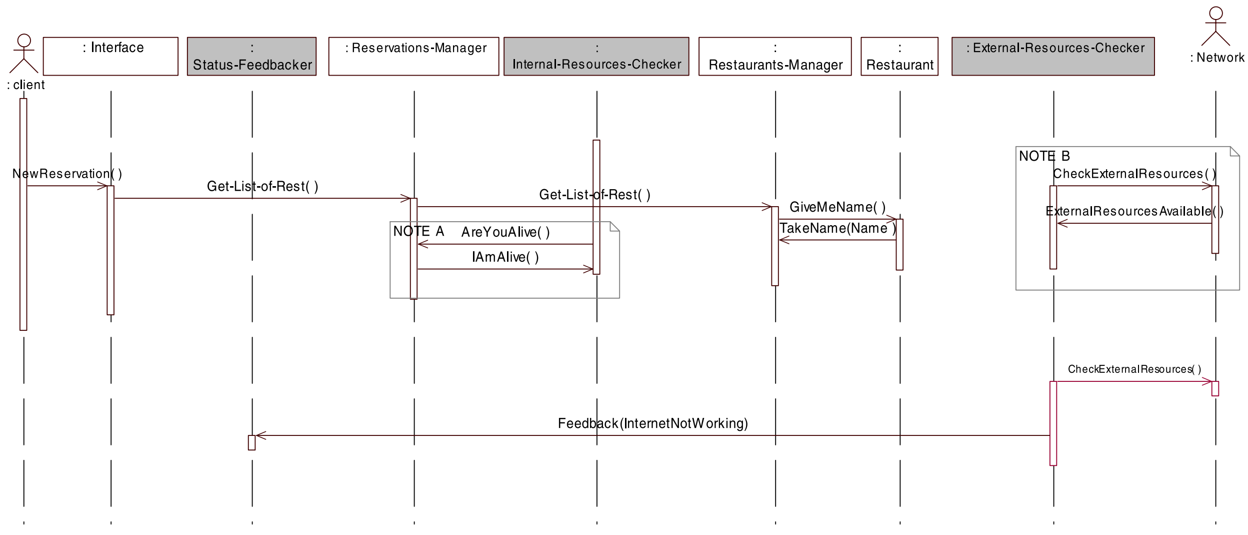

Fig. 3. Sequence diagram for bookings management with System Status Feedback feature and a network failure.

software systems used as MSc dissertations of our UPM students. Besides confirming the relationship between usability and software design, we aimed to have some kind of indication of the impact resulting from incorporating these features, which can be a useful piece of information for developers working on FUFs.

Specifically, we incorporated the FUFs from Table 1 into seven interactive systems (an on-line table booking for a restaurant chain, an outdoor advertising management system, a car sailing system, an adaptable surface transport network system, a computer assembly warehouse management system; an on-line theatre network ticket sale and booking; and an employee profile and job offer processing and matching software system). We deliberately chose interactive systems because usability is more relevant in such systems, and FUFs can be expected to have a bigger impact.

The systems were developed according to the object-oriented paradigm. In all cases, developers started from a software requirements specification without usability features to which the FUFs listed in Table 1 were added. As mentioned earlier, all the systems were developed by UPM Master in Software Engineering students, as part of their MSc dissertation.

For each of the systems to which the FUFs listed in Table 1 were added, we quantified a number of criteria, based on standard metrics used to measure object-oriented systems complexity (number of use cases, number of classes, number of messages, etc.) (Fetcke et al., 1997; Chidamber and Kemerer, 1994):

- *FUF impact on system functionality* (FUF-Functionalities). This parameter mirrors the number of functionalities (in terms of expanded use cases) affected by the FUF in question. To assess this criterion we calculated the percentage of expanded use cases affected by each FUF, which was allocated the value of low, medium, high depending on the interval to which the percentage belongs (under 33%, from 33% to 66%, over 66%).
- *FUF-derived classes* (FUF-Classes). This criterion refers to the number of classes that appear in the design as a result of adding a FUF. This has been assessed by calculating the percentage of new classes derived from the feature, which has been allocated the low, medium, high value depending on the interval to which the percentage belongs (under 33%, from 33% to 66%, over 66%).
- *FUF-derived methods complexity* (FUF-Methods Complexity). The criterion refers to how complex the methods that need to be created as a result of incorporating a given FUF into the system are. It is not easy to provide a measure of the complexity of a method at design time. For the purposes of our study, however, we have classified the possible class methods based on their functionality as follows:
  - Methods related to displaying information, running checks, etc., have been allocated a complexity value of low.
  - Methods related to filters, error corrections, etc., have been allocated a complexity value of medium.
  - Methods related to returning to the earlier state of an operation, saving the state, etc., have been allocated a complexity value of high.
- *Interaction with other system components* (FUF-Interaction). This parameter represents how the classes involved in FUF design couple with the other system classes. To assess this parameter, we measured the percentage of interactions between the FUF-derived classes or between these and other system classes that can be observed in the interaction diagrams. The value of this criterion will be low, medium and high depending on what third this percentage belongs to (under 33%, from 33% to 66%, over 66%).

The need to build different FUFs into a particular project will depend on the project features. For example, *use of different languages* will have a low value for impact on system functionality (FUF-Functionality) if only part of the system can be displayed to users in more than one idiom, whereas it will have a high value if users can access the entire system in more than one language. Similarly, the other FUFs could be designed to affect more or fewer parts of the software system. In our study, all usability features addressed were specified as to be included in the whole system and related to the maximum number of functionalities to which they applied. For example, when *feedback* was added, it was considered that the whole breadth of this feature was needed, including progress bars, clocks, etc., for all the tasks that have need of this functionality.

Moreover, the FUF-Classes, FUF-Methods and FUF-Interactions criteria will very much depend on the type of design. The output values for our systems should, therefore, not be construed as absolute data. On the contrary, they are intended to serve to somehow illustrate the possible effect of adding the respective FUFs on design.

Table 3 shows the data collected for the restaurant system. It is clear from this table that the *cancel* and *undo* FUFs have the biggest impact on design. Not many more classes (FUF-Classes) are added (as the chosen design is one in which a single class is responsible for saving the last state for whatever operations are undertaken, another equally valid design could have envisaged a separate class to save the state for each operation). However, the complexity of the methods (FUF-Methods-Complexity) that need to be implemented is high, as is the number of interactions between the different classes (FUF-Interactions). Moreover, in the *cancel* case especially, this feature is closely related to all system functionalities (FUF-Functionalities), because the HCI literature recommends among other things that easy exit or cancellation should be provided for each and every one of the tasks that the user uses the system to do (Tidwell, 1999). Therefore, this FUF affects almost all system functionalities.

Another FUF with a big impact on all system functionality is *feedback*. Apart from the *system status feedback*

Table 3
Effect of FUFs on restaurant table booking system

On-line booking for a restaurant	FUF-Functionality	FUF-Classes	FUF-Methods Complexity	FUF-Interactions
Feedback	High (87%)	Low (3.3%)	Medium (calculation of times, error control, clocks, etc.)	High 76%
Undo	Medium (50%)	Low (12%)	High (go back to earlier states, save states)	High/medium 66%
Cancel	High (95%)	Low (10%)	High (go back to earlier states, save states, exit operations)	High/medium 66%
User input errors prevention/correction	Medium (53%)	Low (6%)	Medium (filters, user error correction, etc.)	Low 10%
Wizard	Low (20%)	Low (2%)	Low (searches and checks only)	High 72%
User expertise	Low (10%)	Low (4%)	Low (checking and information display)	Low 7%
Help	Low (5%)	Low (8.8%)	Low (information display)	High 69%
Use of different languages	High (67%)	Low (5.8%)	Medium (filters)	Low 15%
Alert	Low (20%)	Low (5.8%)	Low (warnings)	Medium 54%

discussed in the last section, the HCI literature also recommends that the user should receive feedback reporting the *progress* of the operations when the user is doing long tasks (Tidwell, 2005; Brighton, 1998; Coram and Lee, 1996; Welie, 2003), when the *tasks are irreversible* (Brighton, 1998; Welie, 2003) and, additionally, every time the user *interacts* with the system (Brighton, 1998). It is this last recommendation that specially leads to the high FUF-functionality for this feature, as it means that feedback affects all a software system's non-batch functionalities.

On the other hand, we find that the impact of including other FUFs, like for example *user expertise*, are less costly because they can be easily built into a software system and do not interact very much with the other components. A similar thing applies to *help*. In this case, however, despite its low impact on functionality (because this functionality was designed as a separate use case, yielding a 5% and therefore low FUF-functionality value), its interaction is high, as it can be called from almost any part of the system.

We conducted this study for the other systems mentioned at the start of this section. For brevity's sake, Table 4 shows the mean values derived from incorporating the FUFs in those systems. As can be seen, the same pattern applies than the one discussed for the REST system, in terms of the impact of the different FUFs. This was expected according to the similarity of the applications addressed, that is, management systems. Note that these same FUFs may have a slightly different impact on other software systems types, for example, control systems (in which FUFs like *user input errors prevention/correction* or *alerts* may have a bigger impact than the one in Table 4 due to the criticality of the tasks performed) or less interactive systems (in which *feedback* or *cancel* will have less impact).

In sum, the data in Table 4 corroborate that there are some usability recommendations that affect the core functionality of a software system suggesting that specific design components should be created to deal with these features. Therefore, we can confirm that the strategy of separation is insufficient for developing usable software system within a reasonable cost.

Furthermore, although illustrative, the data in Table 4 can provide developers with guidance as to the effort involved in the design of each feature. This kind of information is likely to be of use when discussing with stakeholders which features to incorporate into a software system. It should be important to agree on the incorporation of FUFs with highest design impact as soon as possible, as they will be the ones that will lead to most rework if they are considered in later on. Additionally, those usability features with a bigger impact on system functionality will tend to be more visible in the software built and their inclusion can help to improve system usability more directly. These points are likely to be of interest to developers who have to weigh up the different usability recommendations against the cost of complying with them.

Table 4
Mean values for design impact of FUF

Summary	FUF-Functionality	FUF-Classes	FUF-Methods Complexity	FUF-Interactions
Feedback	High 90%	Low 27%	Medium	Medium/high 66%
Undo	Medium 40%	Low 10%	High	Medium/high 66%
Cancel	Medium 95%	Low 8%	High	Medium/high 66%
User input errors prevention/correction	Medium 36%	Low 11%	Medium	Low 6%
Wizard	Low 7%	Low 10%	Low	High 70%
User Profile	Low 8%	Medium 37%	Medium	Low 10%
Help	Low 7%	Low 6%	Low	High 68%
Use of different languages	Medium 51%	Low 10%	Medium	High 70%
Alert	Low 27%	Low 7%	Low	Medium/high 66%

5. Related work

There are few papers analysing the relationship between usability and software design. A preliminary approach for studying the effect of usability at specific development times beyond UI design was proposed by Bass et al. (2001) and Bass and John (2003). It was their intent to intuitively illustrate that the practice of separating the UI from the system core is not enough to support modifiability, when such modifiability is necessary to improve software system usability. They exemplified a set of scenarios that describe an interaction of a stakeholder with a software system and that is related to the usability of this system. Based on their experience they illustrate how designing these scenarios would require modifications not only to the system interface but also to other parts of the system core. In the authors' words, this approach "was not systematic or comprehensive, but was sufficient to produce substantial evidence that the link between usability benefits and software architecture is much deeper than simple separation of UI from core functionality" (Bass and John, 2003). Additionally, Bass and John's scenarios were based mainly on the "authors' expertise and discussion with colleagues" (Bass and John, 2003). Hence the relation of some scenarios (for example, checking resources) to usability is according to HCI literature questionable.

On the other hand, the researchers of the European STATUS (IST-2001-32298) project have also investigated the relationship between usability and software architecture. To do so, the researchers and practitioners involved in the project used an inductive line of reasoning to describe, based on their expertise, how some usability issues are related to software architecture. For example, as regards the *cancel* feature, we claim "the components processing actions need to be able to be interrupted and the consequences of the actions may need to be rolled back" (Andrés et al., 2002; Folmer et al., 2004). Notice that this approach resembles the work done by Bass and John and can be considered a first step aimed at intuitively illustrating the relations between usability features and software design. Additionally, the usability recommendations that the STATUS project deals with include some features that represent specific software system functionalities (emulation, multi-channelling) rather than HCI recommendations derived from the HCI literature such as we present in this paper.

The research presented in this paper aims to further our knowledge of the relationship between usability and software design. On the one hand, we looked exclusively at those recommendations provided by HCI literature and whose incorporation into a software system is therefore guaranteed to improve software system usability. Additionally, we screened HCI authors that provide enough detail about the usability features to be able to select a preliminary list of FUFs that are likely to have a significant impact on design. Additionally, we applied the preliminary list of FUFs to a number of real cases and observed their impact on the design models to thus confirm that the predicted impact on design matches reality. Finally, we quantified the impact of these FUFs on the systems that we built. As a result we were able to move forward from the illustration of a possible relationship between usability and software design to the practical justification of this relationship.

6. Conclusions and future research

The goal of this paper was first to justify the relationship between usability and software design and second give an idea of the impact of the inclusion of particular usability recommendations into a software system. We do not claim that the values that we have gathered about this impact are a universal measure for all types of applications. Indeed, they will depend, as already mentioned, on the type of system under consideration, the type of design being implemented, the type of customer demands, etc. However, building the responsibilities associated with these mechanisms into a software system will in one way or another lead to significant changes to system design.

The results of this paper are a starting point for justifying to deal with usability at design time as is done with other quality requirements. Specifically, we are working on a possible approach for this purpose, using patterns, similar to design patterns, to provide the developer with guidance as to which classes, responsibilities, etc., should be built into a software system to provide each FUF. At the same time, we are analysing the effect of these FUFs on the requirements elicitation and specification process.

Finally, it is worth recalling that, although dealing with usability during software system design contributes to improve this quality attribute, it is not the only action to be taken to achieve this purpose. As already mentioned, special attention also has to be paid to other usability features with impact on both the UI and the development process to produce highly usable software. Research in this field is, therefore, an open issue.

References

Andrés, A., Bosch, J., Charalampos, A., Chatley, R., Ferre, X., Folmer, E., Juristo, N., Magee, J., Menegos, S., Moreno, A., 2002. Usability Attributes Affected by Software Architecture. Deliverable. 2. STATUS project. <http://www.ls.fi.upm.es/status>.

Barbacci, M., Ellison, R., Lattanze, A., Stafford, J.A, Weinstock, C.B., Wood, W.G., 2003. Quality Attribute Workshop, 3rd ed. CMU/SEI-2003-TR-016, Pittsburgh, Software Engineering Institute, Carnegie Mellon University, USA.

Bass, L., John, B.E., 2003. Linking usability to software architecture patterns through general scenarios. Journal of Systems and Software 66 (3), 187–197.

Bass, L., et al., 1999. Architecture-Based Development. CMU/SEI-1999-TR-007. SEI/CMU.

Bass, L., John, B., Kates, J., 2001. Achieving Usability Through Software Architecture. Technical Report. CMU/SEI-2001-TR-005.

Battey, J., 1999. IBM's redesign results in a kinder, simpler web site. <http://www.infoworld.com/cgi-bin/displayStat.pl?/pageone/opinions/hotsites/hotextr990419.htm>.

Bias, R.G., Mayhew, D.J., 2005. Cost-Justifying Usability. An Update for the Internet Age. Elsevier.

Black, J., 2002. Usability is next to profitability. BusinessWeek Online. <http://www.businessweek.com/technology/content/dec2002/tc2002124_2181.htm>.

Boehm, B. et al., 1978. Characteristics of Software Quality. North Holland, New York.

Bosch, J., Lundberg, L., 2003. Software architecture – Engineering quality attributes. Journal of Systems and Software 66 (3), 183–186.

Brighton, 1998. Usability Pattern Collection. <http://www.cmis.brighton.ac.uk/research/patterns/>.

Constantine, L., Lockwood, L., 1999. Software for Use. A Practical Guide to the Models and Methods of Usage-Centered Design. Addison-Wesley.

Coram, T., Lee, L., 1996. Experiences: A Pattern Language for User Interface Design. <http://www.maplefish.com/todd/papers/experiences/Experiences.html>.

Chidamber, S., Kemerer, C., 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering (June), 476–492.

Donahue, G.M., 2001. Usability and the bottom line. IEEE Software 16 (1), 31–37.

Eskenazi, E.M., Fioukov, A.V., Hammer, D.K., Obbink, H., 2002. Performance prediction for software architectures, in: Proceedings of PROGRESS 2002 workshop, Netherlands.

Fetcke, T., Abran, A., Nguyen, T., 1997. Mapping the OO – Jacobson, Approach into Function Point Analysis, Software, Technology of Object-Oriented Languages and Systems, in: Proceedings of TOOLS 1997.

Folmer, E., Group, J., Bosch, J., 2004. Architecting for usability: a survey. Journal of Systems and Software 70, 61–78.

Griffith, J., 2002. Online transactions rise after bank redesigns for usability. The Business Journal. <http://www.bizjournals.com/twincities/stories/2002/12/09/focus3.html>.

Heckel, P., 1991. The Elements of Friendly Software Design, second ed. Sybex Inc, CA.

Hix, D., Hartson, H.R., 1993. Developing User Interfaces: Ensuring Usability Through Product and Process. John. Wiley & Sons, New York.

IBM, 2005. Cost Justifying Ease of Use. >http://www-3.ibm.com/ibm/easy/eou_ext.nsf/Publish/23> (current 18 May 2005).

IEEE, 1998. IEEE Std 1061: Standard for a Software Quality Metrics Methodology.

ISO, 1998. ISO 9241-11, 98: Ergonomic Requirements for Office work with Visual Display Terminals. Part 11: Guidance on Usability. ISO.

ISO, 2000. ISO 18529, 00: Human-Centered Lifecyle Process Descriptions. ISO.

ISO/IEC, 1991. ISO 9126: Information Technology – Software quality characteristics and metrics.

ISO/IEC, 1999. ISO14598-1, 99: Software Product Evaluation: General Overview. ISO/IEC.

Klein, M., et al., 1999. Attribute-Based Architectural Styles. CMU/SEI-99-TR-022. SEI/CMU.

Laasko, S.A., 2003. User Interface Designing Patterns. <http://www.cs.helsinki.fi/u/salaakso/patterns/index_tree.html> Visited October 2004.

Mayhew, D.J., 1999. The Usability Engineering Lifecycle. Morgan Kaufmann.

McKay, E.N., 1999. Developing User Interfaces for Microsoft Windows. Microsoft Press.

Nielsen, J., 1993. Usability Engineering, AP Professional, Boston, MA.

Perry, D., Wolf, A., 1992. Foundations for the study of software architecture. ACM Software Engineering Notes 17 (4), 40–52.

Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., Carey, T., 1994. Human–Computer Interaction. Addison Wesley.

Rubinstein, R., Hersh, H., 1984. The Human Factor. Digital Press, Bedford, MA.

Scapin, D.L., Bastien, J.M.C., 1997. Ergonomic criteria for evaluating the ergonomic quality of interactive systems. Behaviour & Information Technology 16 (4/5), 220–231.

Seffah, A., Metzker, E., 2004. The obstacles and myths of usability and software engineering. Communications of the ACM 47 (12), 71–76.

Shneiderman, B., 1998. Designing the User Interface: Strategies for Effective Human–Computer Interaction, third ed. Addison Wesley, Menlo Park, CA.

Thibodeau, P., 2002. Users Begin to Demand Software Usability Tests. ComputerWorld. <http://www.computerworld.com/softwaretopics/software/story/0,10801,76154,00.html>.

Tidwell, J., 2005. Designing Interfaces. Patterns for Effective Interaction Design. O'Reilliy, USA.

Tidwell, J., 1999. Common Ground: A Pattern Language for Human–Computer Interface Design. <http://www.mit.edu/%7Ejtidwell/interaction_patterns.html>.

Trenner, L. et al., 1998. The Politics of Usability. Springer, London, UK.

Welie, M., 2003. Amsterdam Collection of Patterns in User Interface Design. <http://www.welie.com/>.

Natalia Juristo is full professor of software engineering with the Computing School at the Technical University of Madrid (UPM) in Spain. She has a B.S. and a Ph.D. in Computing from UPM. She has been the Director of the UPM M.Sc. in Software Engineering for 10 years. Her research areas include Software Usability, Empirical Software Engineering, Requirements Engineering and Software Process. Contact her at natalia@fi.upm.es

Ana M. Moreno is Associate Professor with the Computer Science School at the Universidad Politecnica de Madrid. She has a B.S. and a Ph.D. in Computing. Since 2001 she is Director of the M.Sc. in Software Engineering. Her research interests are Software Usability, Requirements Engineering and Empirical Software Engineering. Contact her at ammoreno@fi.upm.es

Maria-Isabel Sanchez-Segura is an associate professor in the Computer Science Department at Carlos III University of Madrid since 1998. She received her Ph.D. in computer science from the Universidad Politécnica of Madrid. Her research interests include project management, software reuse, process improvement and process improvement usability using collaborative environments. Contact her at Departamento de Informática. Universidad Carlos III of Madrid, Avenida de la Universidad 30, 28911 Leganes, Madrid, Spain; misanche@inf.uc3m.es