Making Usability a Respectable Quality Attribute in the Engineering Lifecycle

by: Ahmed Seffah and Jonathan Benn,
Concordia University, Montreal, QC

Abstract

In this paper, we introduce usability as a quality attribute of a software system, and describe the disastrous results of ignoring it. We will explain why usability has been neglected in current software system development approaches. In particular, we discuss the fundamental, fallacious belief that the user interface and the software system are independent concepts. Finally, we will reflect on methods for completely incorporating user-centered design into the system engineering lifecycle.

Sommaire

Dans cet article, nous introduisons l'utilisabilité comme un des critères importants de qualité d'un logiciel. Nous donnons aussi un apercu sur les conséquences néfastes de sa non prise en compte dans le cycle de developpement logiciel. Nous expliquerons pourquoi, actuellement, l'utilisabilité est négligée pendant le développement des logiciels. En particulier, contrairement à l'opinion répandue, nous démontrerons que l'interface utilisateur et le système logiciel sous-jacent sont des composants indissociables. En conclusion, nous exposerons les méthodes pour intégrer la conception centrée-utilisateur et l'utilisabilité dans le cycle de vie du logiciel.

1.0 Introduction

As a software system user/consumer, you may have experienced something like this: With the hope of becoming more productive, you buy and install a sophisticated piece of software on your desktop computer, PDA or mobile phone. Instead, you end up using only about 20% of its features, either because you are not aware of all the things it can do, or because you don't have time to learn them. This scenario illustrates some of the damage of unusable software.

A lack of usability can increase the time and cost of learning as well as technical support. It can decrease productivity and sales while increasing the cost of maintenance. A strong commitment to usability offers enormous benefits. Among the measurable benefits of usable systems, one can mention decreases in terms of costs and learning time, easier transition to new versions of a system, human performance, productivity enhancements, better quality of work, and fewer user errors in data entry [3].

Furthermore, several studies have shown that 80% of total maintenance costs are related to problems users have with what the system does and not with technical bugs [1]. Of the problems, 64% are with usability [3]. In a recent survey of 8 000 projects, the Standish Group found that lack of user involvement and incomplete user requirements represent the major reasons for project success or failure [6]. One of this situation's causes is that software engineering methodologies, when used for developing highly interactive software with a significant user interface, have a major limitation. Most of them do not propose any mechanisms for:

- Explicitly and empirically identifying and specifying user needs and usability requirements,
- Testing and validating requirements and user interface prototypes with end-users before and during development.

These are among the reasons why usability is becoming an increasingly critical software development issue. Usability assurance and user acceptance are about to become the ultimate measure for the quality of today's e-commerce web sites, mobile services and tomorrow's proactive assistance applications.

2.0 Usability: The Well-known Yet Oft-neglected Quality Factor

For an interactive software product, usability refers to its ease of use, and its ease of learning. Many definitions of usability exist, which sometimes makes usability a confusing concept. ISO alone proposes two different definitions for usability:

- ISO/DIS 9241-11 advocates that usability is a high-level quality objective, and that it is defined 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".
- In ISO 9126 usability is seen as one relatively independent contribution to software quality. Usability is defined as: "A set of attributes of an interactive system that bear on the effort needed for use and on the individual assessment of such use by a stated or implied set of users".

Besides the conflicting definitions of usability within the software engineering and human computer interaction (HCI) communities, there are few integrated software quality models for objectively specifying and measuring our current meaning of usability. One of the ISO standards' weaknesses are that they are not well integrated into our day-to-day software quality assurance procedures and practices.

For example, some software managers will likely feel that their project can't afford so much time spent on usability requirements and user interface design. They will worry that the iterations of prototypes will never end, with "those HCI people" trying to make everything perfect. There are two solutions to this.

First, setting down measurable usability objectives, as part of the project plan, will decrease the project workload, not increase it. It is much easier to build close to the mark right from the beginning (thanks to effective user feedback throughout the process) and then tweak the project until it's correct, than it is to miss the mark by a mile and then try to push the project back on track at the end of the development cycle. If this seems obvious, ask yourself why so many developers wait until their project is nearly complete before they seek user feedback for the first time.

Secondly, overloaded managers should consider the long-term effect of quality work on the self-esteem (and hence productivity) of their developers. DeMarco and Lister [2] have a hypothetical manager saying, "Some of my folks would tinker forever with a task, all in the name of 'Quality'. But the market doesn't give a damn about that much quality - it's screaming for the product to be delivered yesterday..." DeMarco and Lister agree: "People may talk in glowing terms about quality or complain bitterly about its absence, but when it comes time to pay the price for quality, their true values become apparent." And continue on to say that "the client's perceived needs for quality in the product are often not as great as those of the builder," but that letting the builders of a product (the software developers) apply their own judgment as to when the software is ready for release will result in higher productivity in the long run.

3.0 The Fallacy Of A Cartesian Separation Between The User Interface And The System's Functionality

There is a common (false) conception that a software system's functionality exists independently of the user interface - in fact, there is no need for functionality except for what is needed by the user. If the user inter-

face doesn't provide access to a certain piece of internal functionality, that functionality is dead code that might as well not exist. The link exists along the reverse direction as well: a system with poor functionality will have a poor user interface.

The term user interface is perhaps one of the underlying obstacles in our quest for usable programs since it gives the impression of a thin layer sitting on top of the other software that is the "real" system. This dichotomy between the perceived situation and the real situation is explained by the peanut butter theory of usability. This is the specimen of software in which usability is seen as a spread that can be smeared over any software model, however dreadful, with good results if the spread is thick enough. If the underlying functionality is confusing then spread a graphical user interface on it… If the user interface still has some problems, smear some manuals over it. If the manuals are still deficient, smear on some training which you force users to take.

This fallacy dichotomy does not take into account the intimate relationship that exists between internal attributes and external factors that affect the usability of a system. As an example, the user and developer are both interested in the software's performance, but the user could see this attribute as response time to the event entered by him, while the developer thinks of it as data structure depth or path length. Furthermore, a requirement for quality in use may be that the system will increase the user's performance by 20% when doing routine tasks. If a search engine function is to give a fast response time, the information may need to be indexed in a certain way to support fast retrieval. Only by ensuring that goals at the internal functionality level mirror goals at the user level can this 20% performance increase be achieved.

4.0 Involving The User In The Software Development Lifecycle Is A Beginning

Most usability professionals agree on the basic approach to user-centered design. Key steps include requirements gathered through observation and interview, creating a conceptual design, iterative development, testing and refinement. While specific situations may call for different techniques and different levels of formality, the basic structure is generally similar. This commonality is the basis for the emerging ISO Standard 13407: Human-Centered Design Process for Interactive Systems. However, although some software engineering standards claim to have similar goals to those promoted in ISO standard 13407, in practice they often seem very different. This is because they are formulated using different terminology, notations and languages. An example of this would be the IEEE standards on software quality and the ISO collection on quality in use (see the IEEE-1061 Standard on Software Quality Metric Methodology and the ISO-9126 Standard on Quality Characteristics and Guidelines for their Use).

Too often, user-centered design remains the province of visionaries and a few enlightened software practitioners and organizations such as IBM, Microsoft and SUN, rather than the everyday practice of programmers and analysts. One barrier to the wider practice of user-centered design is that its structure and techniques are still relatively unknown, underused, difficult to master, and essentially inaccessible to small and medium-sized software development teams and common developers. While software developers may have high-level familiarity with such basic concepts as requirements analysis and usability testing, few understand the complete process at a level that allows them to incorporate it into the larger software development lifecycle.

5.0 Moving From Technology-driven To Human-centered Engineering Practices

User-centered design is a philosophy opposed to the system-driven development philosophy that is the traditional way of seeing and doing things in software development. This philosophy consists of involving the software's end-users in all of its development stages [5]. User-centered, or human-centered, design is the direction in which we want software development to evolve, but it has yet to become established practice. Hopefully, it will become the recognized, accepted way of doing things. However, dislodging the system-oriented approach will take an enormous amount of effort or a miracle. This miracle may be on its way thanks to new technologies and decentralized software development.

It is important to use the term user-centered design unequivocally. It is a bit unfortunate that for the software engineering community, usability engineering has become the way of thinking about user-centered design. Usability engineering focuses on requirements and evaluations, thus preserving a technical, engineering-oriented attitude to software development. User-centered design, on the other hand, addresses designing with the users. Figure 1 clarifies the differences between human-centered and technology-driven development. It also illustrates some avenues for bridging the gap between software engineering and UCD practices.

6.0 Establishing Usability In The Software System Engineering Lifecycle

Engineering a product for usability requires attention to the user interface but also to all the other elements that might affect usability including: user manuals, training materials, help system, technical support as well as installation and configuration procedures. It takes a shift from the waterfall development model to an interactive process that comprises the following milestones:

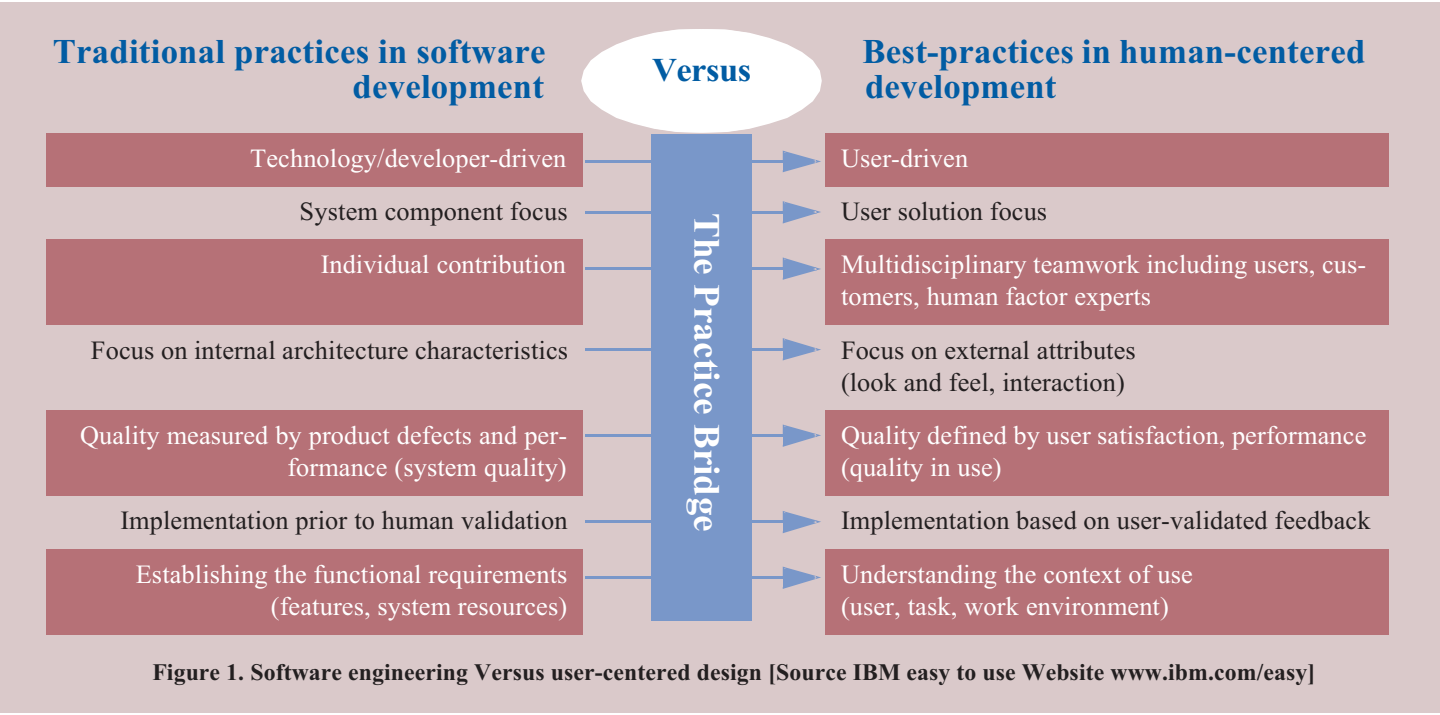

Figure 1. Software engineering Versus user-centered design [Source IBM easy to use Website www.ibm.com/easy]

- Analyzing and specifying user needs and requirements. Besides giving details on the functional requirements, these also consist of collecting information about user personas and their tasks, as well as the technical and organizational environment in which the system will be used.
- Using digital images, sound, video and animations to develop proofs of concept, design solutions and prototypes. Prototypes may range from simple paper mock-ups of screen layouts to prototypes with greater fidelity, which run on computers.
- Planning and conducting usability evaluation and user-oriented tests for collecting user feedback and understanding user behaviors. Here we can use an audio and video monitoring system for conducting ethnographic interviews and online customer satisfaction surveys. We can also conduct performance measurements, where the users attempt to accomplish "real world" tasks, using a prototype or the final system. The quantitative feedback from these empirical studies are then transformed into insights and patterns that can be used to develop the improve the original design.

Figure 2 describes the main activities suggested by the ISO 13407 standard on human-centered processes for interactive systems. Constantine (1999) and Mayhew [4] both proposed a detailed development lifecycle including these activities.

The only presently feasible approach to successful design is an empirical one, requiring observation and measurement of user behavior, careful evaluation of feedback, insightful solutions of existing problems, and strong motivation to make design changes. User-derived feedback about ease of use and ease of learning is collected directly and/or indirectly from users, and then transformed into design recommendations, decisions, principles, guidelines, design patterns and look and feel guidelines that can be used as proven design solutions to common user problems, or as best design practices.

7.0 Usability Versus The Other Quality Attributes And Safety In Particular

As we have already mentioned, usability is intimately coupled with other software quality attributes, including safety and security. The Therac-25 device is a perfect historical example of this intimate relationship.

This appliance was a cancer irradiation device whose faulty operation led to a number of deaths. Eleven Therac-25s were installed in the US and Canada. Six accidents involving massive overdoses to patients occurred between 1985 and 1987. It may seem intuitive that a device that is easy to use and learn is safer than one that isn't, but this is not always the case. One of the safety features in the original Therac-25 design was that all the settings for the device had to be entered through a terminal, as well as on a control panel. Users as well as developers saw this as redundant, and the original design was changed before release so that the settings could be entered on the terminal alone. Using the new user-friendly GUI, once the settings were accepted by hitting the return key, the user was asked to confirm that the settings were correct - by hitting the return key again. This extra step was considered a replacement for the control panel, and in the opinion of the developers it would increase the system's ease of use while reducing its complexity.

Unfortunately, users started pressing the return key twice in succession, as a reflex. With repetition, the action became like double-clicking a mouse and the machine's settings were never really reviewed. Because of a fault in the Therac-25's software, some data entries weren't properly recorded. The fault was a race condition created because proper resource locking of the data wasn't exercised. Since the crosscheck in the user interface had been removed, the fault was never detected in time to save lives. Here was an example of a software system where the design was altered to favor usability, but the safety of the device was fatally compromised.

The story of the Therac-25 holds many powerful lessons, including:

- Designing the correct user interface for a system is, contrary to popular opinion, very difficult. It requires research, user validation and the careful balancing of many trade-offs. The Therac-25's human-computer interface required more thorough thought than it received.
- The inseparability of the system and the user interface. A hastily improved user interface could not cover up fatal flaws inside the software system.
- Better usability did not automatically equate to better safety. In fact, in this case ease-of-use and safety were trade-offs of each other. Better user validation would have revealed the error in allowing return key double-clicks.
- Usability is one quality factor among many. In this case, safety was of critical importance, and more essential than ease-of-use.

The most important lesson we can pull from the tragedy of the Therac-25 is this: the deaths due to the irradiation appliance could have been prevented (in spite of the internal system flaws) had there been a greater emphasis on user validation and feedback. Had the potential users (doctors and nurses) been consulted throughout the development process, the system engineers would surely have had an easier time concluding that either (1) The system was better off retaining the control panel in favor of safety, or (2) A confirmation mechanism other than a second return key press was required in order to ensure that the data entries were reviewed. However, because user validation and feedback was left to the end of the process, these important conclusions were never reached.

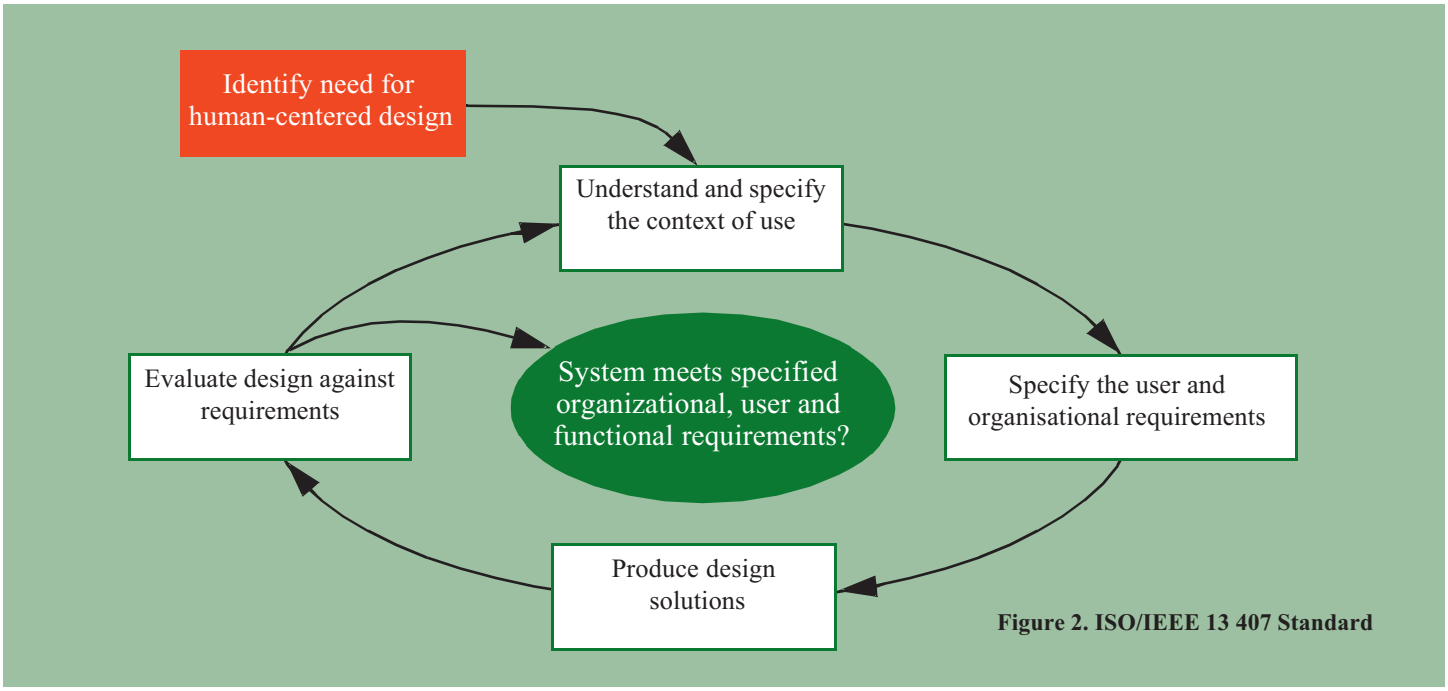

Figure 2. ISO/IEEE 13 407 Standard

Usability..., *continued from previous page*

8.0 Concluding Remarks

A successful usability development approach can entail setting up environments and methods to monitor users doing things to better understand how to help them work well; it can include developing methods to normalize user input without bias; and perhaps most importantly, it can facilitate the creation of interfaces that make good use of the gathered information. In any case, it is also pertinent that users be involved in every step to ensure that it's their input that is being reflected, and not the opinions of those applying the usability engineering.

It is now acknowledged that software engineering as an engineering discipline involves the development of software through accepted practices to facilitate economic success. It is important that it be recognized early on that usability engineering is a key component to meeting the above description. Hopefully, much emphasis will be placed on this aspect of software engineering in the near future, and more importantly, corresponding pressure will be applied to create the appropriate courses at the graduate and undergraduate levels in order to supply the market with appropriately equipped software engineers.

9.0 References

[1]. Boehm, B.W. Software Engineering Economics, Englewood Cliffs, New Jersey: Prentice Hall, 1981.

[2]. DeMarco T. and Lister T. Peopleware: productive projects and teams. Dorset House, New York, 1987.

[3]. Landauer, T.K. "The Trouble with Computers: Usefulness, Usability and Productivity". MIT Press, 1995.

[4]. Mayhew, D.J.; The Usability Engineering Lifecycle: A Practitioner's Handbook for User Interface Design, Morgan Kaufman Publishers, 1999.

[5]. Norman D.A. and Draper S.W. (eds), User-Centered System Design, Laurence Erlbaum, Hillsdale NJ, 1986.

[6]. Standish Group. "CHAOS Chronicles or CHAOS: A Recipe For Success". 1995.

About the authors

Since January 2000, **Ahmed Seffah** has been an assistant professor of HCI and software engineering in the department of Computer Science at Concordia University. He is the Concordia research chair on Human-Centered Software Engineering and the co-founder of the Usability and Empirical Studies Laboratory. He holds a Ph.D. in software engineering from the École Centrale de Lyon (France). After completing his postdoctoral in 1994, he joined in 1995 the Computer Research Institute of Montreal (CRIM) as a member of the research staff. His research interests include usability requirements and testing, human-centered development, user interfaces engineering, quality in use measurement and metrics, and empirical software engineering. Dr. Seffah is the vice-chair of the IFIP working group on human-centered design methodologies.

Jonathan Benn is finishing his bachelor's degree in software engineering at Concordia University, in Montréal, Québec. Jonathan is also looking forward to obtaining a master's degree in human-centered software engineering. He is currently the technical manager of the Concordia Usability and Empirical Studies Lab. In the past, he has enjoyed tutoring fellow students on the finer points of C++ and PC assembly language programming. Jonathan's interests include improving the usability and safety of embedded systems, improving web usability, website design, software architecture, and C++ programming.

Self-Erecting Inverted Pendulum, *continued from page 19*

the motor voltage after this disturbance do not peak much over 4 V at times. The response of the motor is the important factor for this plot seen. The motor does not begin to generate much driving force until it nears the ±4 V range.

8.0 Conclusion

Image of the pendulum

The self-erecting inverted pendulum has been manufactured for a small cost and experiments have shown promising results. This system can be used as an educational tool for helping understand model dynamics and controller response.

Suggested improvements for this system would include a motor with an improved response. This would increase the robustness of the system and enhance disturbance recovery and swing up.

After many experimental tests the repeatability of the swing up controller is less than ideal. Since this controller is open loop, any disturbances such as slight bends in the electrical harness attached to the cart create friction causing the system to respond differently each time. Improvements could be implemented by designing a closed loop controller for the swing up.

9.0 Acknowledgments

The author would like to thank the following individuals: Student project members Brett Blyth and Christiaan Woodfield; project supervisor Dr. A. Tayebi; machinist and mechanical specialist Kailash Bhatia; PCB manufacturing and component locating support from Warren Paju and Manfred Klein; computer and programming support from Bruce Misner; swing up modelization theory support from Dr. K. Liu; and mechanical measurement support from Ed Drotar.

10.0 References

[1]. G. Perry, "Teach Yourself Visual Basic 6 In 21 Days", Sams Publishing, Indiana, 1998.

[2]. K. Ogata, "Modern Control Engineering: Third Edition", Prentice Hall, New Jersey, 1997, 3rd ed., pp. 952-957.

[3]. D. Galick, and D. Schelle, "Inverted Pendulum Degree Project", Lakehead University, 2001.

[4]. "Model-based Design of Control Systems" [Online Article], [cited November 2003], Available http://www.mathworks.com/products/controldesign/

About the author

Stephen McGilvray received his B.Eng degree in Electrical Engineering from Lakehead University in 2002. He was awarded the IEEE Life Member Award in 2003 for his student paper on control of a self-erecting inverted pendulum. He is currently working towards his M.Sc.Eng. in Control Engineering at Lakehead University in Thunder Bay. His main areas of research include nonlinear control of a VTOL four-rotor helicopter and force control of robot manipulators.