Clarifying the Relationship between Software Architecture and Usability

Natalia Juristo, Ana M. Moreno
School of Computing - Universidad Politécnica de Madrid, Spain
natalia@fi.upm.es, ammoreno@fi.upm.es

Maria Isabel Sánchez
School of Computing - Universidad Carlos III de Madrid, Spain
misanche@inf.uc3m.es

Abstract

This paper examines in a problem posed recently concerning the relationship between software system usability and architecture. Here, we try to empirically clarify this relationship, focusing on the concept of architecture-sensitive usability mechanism. This concept represents specific usability issues that can improve software usability and that have demonstrated architectural implications. Accordingly, this paper outlines how usability needs to be decomposed to be dealt with from an architectural point of view and how the architecture-sensitive usability mechanism emerges. A list of architecture-sensitive usability mechanisms is presented and the procedure for outputting their respective architectural implications is discussed.

1. Introduction

Usability is an important component of software quality. Although there is no established set of critical software quality attributes, several classifications agree on the importance of considering usability as a quality attribute [1][2][3]. Additionally, usability is increasingly recognized as a quality attribute that has a big impact on software development [4].

To understand the depth and scope of the usability of a system, it is useful to make a distinction between the visible part of the user interface (buttons, pull-down menus, check-boxes, background color, etc.) and the interaction part of the system. By interaction we mean the coordination of information exchange between the user and the system. A system's usability deals not only with the user interface, but mainly with the user-system interaction. This interaction must be carefully designed and should be considered when designing not just the visible part of the user interface, but also the rest of the system. For example, the provision of continuous feedback for users is a primary usability feature, and its implementation needs to be considered when designing the system. System operations have to be designed so as to allow information to be frequently sent to the user interface to keep users informed about the current status of the operation. So, although this information could be displayed by different means (percentage-completed bar, a clock, etc.) and these means are interface or presentation issues, the feedback feature is not just an interface aspect. It is a functionality that affects system usability and should be considered during design, as the design is affected by the decision on whether or not to include this usability feature.

However, seminal interactive system architectures, such as Model-View-Controller (MVC) and Presentation Abstraction Control (PAC) [5] seem to assume that usability only affects the presentation and dialogue components of an interactive application. Based on this assumption, these architectures decouple the application features from the user interface, such that each can be designed and modified more or less independently of the other. This assumption does not consider the fact that functionalities buried in the application logic can sometimes affect the usability of the whole system.

Recently, some groups have been working on identifying specific usability aspects with connections in the software architecture to try to clarify this relationship [6] [7]. These papers show how even if the presentation of a system is well designed, system usability can be greatly compromised if the underlying architecture and designs do not make the proper provisions for user concerns.

In this paper, our aim is to contribute to this clarification by empirically studying the relationship between software usability and software architecture [1]. Note that it is important to clarify this relationship, because, as mentioned above, if any such relationship exists, developers should bear usability issues in mind when defining the overall system and not just when working on the user interface.

To deal with this relationship, we have decomposed usability into lower level concepts more related to the software solution. As we will see in section 2, these concepts are usability attributes and usability properties.

[1] The content of this paper is part of the research done in the STATUS project: European Union funded project IST–2001–32298.

Then the concept of architecture-sensitive usability mechanism is introduced in section 3, identifying specific usability features that will address a particular usability property and whose inclusion in a software system will have a specific effect on its architecture. Section 4 shows an example of the architectural implications for one such architecture-sensitive usability mechanism (Undo). It also describes the empirical process followed to identify these implications, and refers to the site where the implications of the other architecture-sensitive usability patterns identified can be found. From our research, we conclude that there is a relationship between usability and software architecture and that it is, therefore, dangerous to assume that usability will only affect the presentation component of our software systems. Usability also needs to be dealt with when designing the logic of applications.

2. Decomposing Usability from the Architectural Viewpoint

One of the problems of working with usability from a design perspective is that it is a broad and abstract concept that is hard to grasp. Therefore, the best way of addressing the concept of usability is to decompose it. The first level of the usability decomposition is what is called usability attributes in the (Human Computer Interaction) HCI field. Usability attributes are precise and measurable components of the abstract concept that is usability. Usability has been decomposed into attributes in the HCI field mainly for evaluation purposes. Although different authors have proposed different usability attribute classifications, the view that appears to be shared by most of the prominent authors in the field is that the main *usability attributes* are [11] [12] [14]:

- **Learnability**, which is composed of two complementary aspects: how quickly users can learn to use the system for the first time and how easy it is to remember how to operate the system after not having used it for some time.
- **Efficiency of use**, which refers to how efficiently the user performs a task using the system, that is, this attribute measures the efficiency of the software system used by the user. Note that this attribute is not the same as the classical quality attribute of efficiency, understood as system efficiency.
- **Reliability of use**. Again, this parameter is not to be confused with system reliability. It refers to the reliability of the user performing a task using the system. Therefore, this attribute refers to the errors made by the user when using the system, not the system errors.
- **Satisfaction** is the most subjective attribute and refers precisely to the user's subjective view of the system.

However, these usability attributes are very far removed from software design, that is, the effect that these attributes have on software architecture cannot be determined directly. Therefore, the approach that we have followed has been to decompose these attributes into intermediate levels of concepts that are increasingly closer to the software solution. The first one of these concepts is usability property.

We have identified *usability properties* from the HCI field. HCI researchers have defined some concrete aspects to help developers to build usable systems. Each author has named these tips differently: design heuristics [8], rules of usability [9], principles of usability [10][11], ergonomic principles [12], etc. We have compiled these design heuristics and principles that different authors suggest for developing more usable systems [8][9][10][11][12][13][14] and have arrived at the following usability properties for a software system:

- **Keeping the user informed**. The system should inform users at all times so that they know what is going on.
- **Error management**. The system should provide a way to manage errors. This can be done by error correction or error prevention.
- **Consistency**. The system should be consistent in all aspects of interaction, that is, in the interface and in the way we provide functionality.
- **Guidance**. We should provide informative, easy-to-use and relevant guidance and support both in the application and in the user manual to help the user understand and use the system.
- **Minimize cognitive load**. Systems should minimize the cognitive load, e.g., humans have cognitive limitations, and systems should bear these limitations in mind.
- **Explicit user control**. Users should feel that they are in control of the interaction.
- **Natural mapping**. The system should provide a clear relationship between what the user wants to do and the mechanism for doing it.
- **Ease of navigation**. Systems should be easy to navigate.
- **Accessibility**. Systems should be accessible in every way that is required. This property includes internationalization, multi-channeling and accessibility for disabled people.

Although this classification could contribute to somehow structuring the field of design heuristics, an important problem still remains to be addressed.

Usability properties may be useful as possible sources of requirements to be satisfied by a usable software system. However, developers have no systematic way of incorporating them into their developments. In other words, they need to know what particular elements a software system has to include to satisfy a usability property. Therefore, usability properties need to be further elaborated if we want developers use them to incorporate specific functionalities to improve the usability of the software systems.

3. Architecture-Sensitive Usability Mechanisms

Very recently, the HCI community has developed the concept of usability pattern. There are several a few lists of usability patterns, the most commonly referenced being the Amsterdam Collection [15] and Common Ground [16]. HCI usability patterns provide usability solutions (allow the user to undo at least the last couple of actions, provide feedback to the user every two seconds of command processing, in forms to be filled by users arrange the blanks in an order that makes sense semantically, use different colors to identify the major sections of the screen, etc.) to common problems.

Note that, on the one hand, the inclusion of some of these usability solutions in a software system will help to address specific usability properties. On the other, the inclusion of some of these solutions in a software system could have an effect on its software architecture and not only on its user interface.

So, we have developed the concept of *architecture-sensitive usability mechanism*, to refer to specific usability features that have an impact on the software architecture (as we will see in the next section) and address particular usability properties. In other words, we have descended another level in our approximation of usability to architectural design, defining the concept of architecture-sensitive usability mechanisms. An architecture-sensitive usability mechanism addresses a need identified by a usability property at the requirements stage and that has a specific effect on the design of the software system.

Note that we avoid to use the concept of usability pattern, as from a software engineering perspective, patterns should provide validated design solutions to repetitive problems [17], while, architecture-sensitive usability mechanisms represent usability features that affect software architecture. As noted at the end of this paper, we intend to pursue this work in the future by approximating these mechanisms to architectural sensitive usability patterns, adding to the usability solutions proposed by the HCI community particular design solutions.

Table 1 shows the relationship between usability properties (rows) and architecture-sensitive usability mechanisms (columns) that we have considered. A detailed description of this relationship is given in [18].

Table 1. Relationship between Usability Properties and Architecture-sensitive Usability Mechanisms

	Architecture-sensitive Usability Mechanisms											
Usability Properties	Different languages	Feedback	Undo	Form/Field validation	Wizard	User Profile	Cancel	History Logging	Command Aggregation	Action for multiple objects	Workflow Model	Provision of Views
Keeping the user informed		X										
Error management												
Error prevention		X		X	X		X		X		X	
Error correction			X				X	X				
Consistency												
Guidance		X			X							
Minimize cognitive load									X		X	
Explicit user control			X				X			X		X
Natural mapping												
Ease of navigation										X		
Accessibility	X											
Adaptability						X		X				X

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

4. Studying the Implications of Usability Mechanisms into Software Architecture

To analyze the architectural implications of the architecture-sensitive usability mechanisms presented in Table 1, we worked with different practitioners asking them to incorporate these mechanisms into their developments, once they had made the design for the system considering none of such mechanisms. Specifically, we worked on two small real applications developed by final-year Computing students, one real application developed by one of our Master students, and another real application developed by one of the industrial partners of the STATUS project. If the practitioners modified their designs to incorporate a specific mechanism, then the respective mechanism can be considered to be architecture sensitive.

The exact process followed to study the relationship between the usability mechanisms and the software architecture was:

- We worked with a list of usability mechanisms longer than the one that appears in Table 1, and compiled from HCI literature about specific software elements that improve system usability.
- We asked designers to build the design models for the systems without including usability mechanisms.
- We asked designers to modify their original developments to include the functionality for each of the mechanisms under consideration.
- If the modifications made affected the design models, for example, involved the inclusion of new components or different interactions between existing components, we considered that the mechanism was architecture sensitive and generalized the design solutions provided by the different practitioners for these mechanisms.
- If the modification did not affect the design models (typically they affected in this case to lower level functions or pseudocode) then the mechanisms was considered non architectural sensitive.

An example of the architectural implications of one of the architecture-sensitive usability mechanisms (Undo) is shown in Figure 1. The complete demonstration of the architectural impact of the mechanisms shown in Table 1 appears in [19], including a detailed description of the design solutions provided for the practitioners for each mechanism, how they were derived, and an example of the inclusion of these mechanisms in a specific application. Note that the generalized architectural solutions for each mechanism (like the one shown in Figure 1) represents just one possible way of incorporating such usability mechanisms into a software design. Its goal is just show the architectural implication of a mechanisms but not at all the only solution to design such mechanisms.

5. Conclusions

Usability is a key issue in software development. This paper has shown an approach for dealing with usability from an architectural point of view. In particular, we have shown how usability has a real impact on software architecture, not only affecting the user interface as usually thought. Therefore, it is important to bear in mind the concept of usability when designing the overall system functionality and not just when designing the user interface.

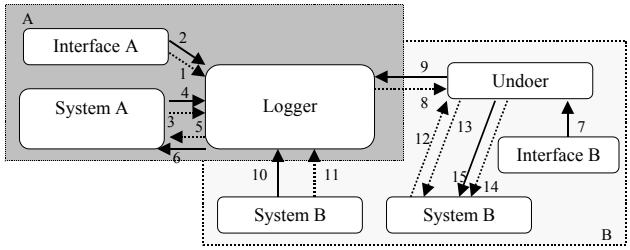
The approach followed to illustrate the relationship between usability and software architecture focused on decomposing the concept of usability into lower levels that are progressively closer to the solution domain: usability attributes, properties and mechanisms. While usability attributes come from traditional HCI attributes, usability properties are taken from existing tips and heuristics that can be found in HCI literature. Finally, architecture-sensitive usability mechanisms represent specific usability issues to be incorporated into a software system and that have a demonstrated impact on software architecture.

By the time being, developers can use this work to consider usability mechanisms to incorporate into their systems during software architecture design. However, we are expanding this work to better serve developers. Specifically, we are developing what we have referred to as architecture-sensitive usability patterns which package both usability solutions and design solutions to mechanisms. In these patterns we customize architectural implications of each mechanism for specific architectural

restrictions, for example, the use of MVC or PAC architectures; also we make explicit the user interface implications of these mechanisms to inform developers of what effect these mechanisms have on both the software architecture and the user interface.

So, although a lot of work still remains to be done to elucidate the exact details of the relationship between software usability and software architecture, we have presented a first step that empirically demonstrate that there is such a relationship, and we have explicitly identified which usability issues involves such relationship.

- **Usability Mechanism:** The ability to undo an action and return to the previous state.
- **Example of design solution:**
  - **Diagram**:



  - **Participants**: This mechanisms design has two clearly separate parts. These parts have been labeled in the illustration as A and B, respectively. Part A collects the actions performed in the system (the number of actions to be stored will have to be specified when the system is developed) so that they can be later undone. Part B manages the respective undo.
    - Interface A: receives the request to execute an operation in the system, which may contain both the operation and data (1) (2). As we will see later, this execution request can also come from the actual system (3) (4).
    - System A: this module sends the functions and data executed in the system to the logger (3) (4) and also, optionally, if the logger does not store the actions internally, will send the information to the part of the system that manages these actions (5) (6).
    - Logger: this module receives the actions and the data requested by the user or from another part of the system (1) (2) (3) (4) and stores the logged action and data either internally or in another part of the system, in which case it will have to send this action and data to the system (5) (6) to be processed by the respective part of the system. Logger receives the undo request from Undoer (9) and, if the logged actions are stored in the logger, it then sends them one by one to Undoer (8). If they are not stored in the logger, it will receive both the data and the operation to be undone from another part of the system, which we have named System B, through (11) and (10), respectively.
    - Interface B: receives the undo request and sends it to Undoer through (7).
    - Undoer: sends the undo request to logger (9) and also sends each of the actions to be undone that it receives from logger to System B (13), as well as receiving the opposite operation to the one performed from System B (12). When it knows which opposite operation is to be performed, it sends the operation to System B along with the data associated with the operation in question through (14) and (15).
    - System B: it will search the system for both the action performed and the data associated with this operation (10) (11) if the data are not stored internally in the logger. It receives the actions to be undone (13) and provides the opposite operation (12) (for which purpose it will have to store what the opposite is for each action, see implementation section for example). The opposite action and the respective data will be sent to the respective part of the system ((15) and (14)).
- **Related mechanisms**: History logging is equivalent to part A of this mechanism. Therefore, if undo is provided, history logging could be provided at no extra cost.
- **Mechanisms implementation in OO**: This mechanism will generate an "undoer" class responsible for triggering the entire undo process. Additionally, there are the "listener" and "action-done" classes, which are used to store the actions that are performed as the system operates. A "system-action" class also has to be included to establish what the opposite is for each action that can be undone through the "is-the-opposite" relationship.
- **Example**: See [19] for a full example, not included here for reasons of space.

Figure 1. Architectural implications of the "Undo" mechanism

References

[1] IEEE. *IEEE Std 1061: Standard for a Software Quality Metrics Methodology*. IEEE, 1998.

[2] ISO. *ISO 9126-1 Software Engineering – product quality – part 1: Quality Model*. ISO, 2000

[3] B. Boehm, J.R. Brown, H. Kaspar, M. Lipow, G.J. Macleod, M.J. Merritt. *Characteristics of Software Quality*. North Holland, 1978.

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

[5] L. Bass, P. Clements, R. Kazman. *Software Architectures in Practice*. Addison Wesley, Reading, MA, 1998.

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

[7] J. Bosch and N. Juristo. "Designing Software Architectures for Usability". *ICSE Tutorial*. Portland, OR, May 2003.

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

[9] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 1998.

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

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

[12] D.L.Scapin, J.M.C. Bastien, *Ergonomic criteria for evaluation the ergonomic quality of interactive systems,* Behaviour & Information Technology, vol 16, no 4/5, pp. 220-231

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

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

[15] M. Welie. The Amsterdam Collection http://www.welie.com, visited September 2003.

[16] Tidwell. The Case for HCI Design Patterns. http://www.mit.edu/jdidwell/common_ground_onefile.htm, visited September 2003.

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

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

[19] N. Juristo, A. Moreno, M Sánchez. "Techniques and Patterns for Architecture-Level Usability Improvements". *Deliverable 3.4. STATUS project*. Http://www.ls.fi.upm.es/status May 2003.