The Journal of Systems and Software 81 (2008) 1845–1852

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Reconciling usability and interactive system architecture using patterns

Ahmed Seffah *, Taleb Mohamed, Halima Habieb-Mammar, Alain Abran

Human-Centered Software Engineering Group, Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada

ARTICLE INFO

Article history:
Received 26 July 2005
Received in revised form 9 April 2008
Accepted 10 April 2008
Available online 4 May 2008

Keywords:
Usability
User interface design patterns
Software architecture
Usability factors
Software quality metrics
HCI design principles

ABSTRACT

Traditional interactive system architectures such as MVC [Goldberg, A., 1984. Smaltalk-80: The Interactive Programming Environment, Addison-Wesley Publ.] and PAC [Coutaz, J., 1987. PAC, an implementation model for dialog design. In: Interact'87, Sttutgart, September 1987, pp. 431–436; Coutaz, J., 1990. Architecture models for interactive software: faillures and trends. In: Cockton, G. (Ed.), Engineering for Human–Computer Interaction, Elsevier Science Publ., pp. 137–153.] decompose the system into subsystems that are relatively independent, thereby allowing the design work to be partitioned between the user interfaces and underlying functionalities. Such architectures extend the independence assumption to usability, approaching the design of the user interface as a subsystem that can designed and tested independently from the underlying functionality. This Cartesian dichotomy can be fallacious, as functionalities buried in the application's logic can sometimes affect the usability of the system. Our investigations model the relationships between internal software attributes and externally visible usability factors. We propose a pattern-based approach for dealing with these relationships. We conclude by discussing how these patterns can lead to a methodological framework for improving interactive system architectures, and how these patterns can support the integration of usability in the software design process.

1. Introduction

Software architecture is defined as the fundamental design organizations of a system; they are embodied in its components, their relationships to each other and the environment, and the principles governing its design, development and evolution [ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems]. In addition, it encapsulates the fundamental entities and properties of the application that generally insure the quality of application (Kazman et al., 2000).

In the field of interactive systems engineering, architectures of the 1980s and 1990s such as MVC (Goldberg, 1984) and PAC (Coutaz, 1987, 1990) are based on the principle of separating the functionality from the user interface. The functionality is what the software actually does and what information it processes. The user interface defines how this functionality is presented to end-users and how the users interact with it. The underlying assumption is that usability, the ultimate quality factor, is primarily a property of the user interface. Therefore separating the user interface from the application's logic makes it easy to modify, adapt or customize the interface after user testing. Unfortunately, this assumption does not ensure the usability of the system as a whole.

We now realize that system features can have an impact on the usability of the system, even if they are logically independent from the user interface and not necessarily visible to the user. Bass observed that even if the presentation of a system is well designed, the usability of a system could be greatly compromised if the underlying architecture and designs do not have the proper provisions for user concerns (Bass et al., 2001; Raskin, 2000). We propose that software architecture should define not only the technical interactions needed to develop and implement a product, but also interactions with the users.

At the core of this vision is that invisible components can affect usability. By invisible components, we mean any software entity or architectural attribute that does not have visible cues on the presentation layer. They can be an operation, data, or a structural attribute of the software. Examples of such phenomena are commonplace in database modeling. Queries that were not anticipated by the modeler, or that turn out to be more frequent than expected, can take forever to complete because the logical data model (or even the physical data model) is inappropriate. Client-server and distributed computer architectures are also particularly prone to usability problems stemming from their "invisible" components.

Designers of distributed applications with Web interfaces are often faced with these concerns: They must carefully weigh what part of the application logic will reside on the client side and what

* Corresponding author. Tel.: +1 514 848 2424x3024; fax: +1 514 848 4568.
E-mail address: seffah@encs.concordia.ca (A. Seffah).
URL: http://hci.cs.concordia.ca/www.

0164-1212/$ - see front matter © 2008 Published by Elsevier Inc.
doi:10.1016/j.jss.2008.04.037

part will be on the server side in order to achieve an appropriate level of usability. User feedback information, such as application status and error messages, must be carefully designed and exchanged on the client and server part of the application, anticipating response time of each component, error conditions and exception handling, and the variability of the computing environment. Sometimes, the Web user interface becomes crippled by the constraints imposed by these invisible components because the appropriate style of interactions is too difficult to implement.

Like other authors (Bass et al., 2001; Folmer and Bosch, 2003), we argue that both software developers implementing the systems features and usability engineers in charge of designing the user interfaces should be aware of the importance of this intimate relationship between features and the user interfaces. This relationship can inform architecture design for usability. With the help of patterns, this relationship can help integrate usability concerns in software engineering. Beyond proposing a list of patterns to solve specific problems, our long-term goal is to define a framework for studying and integrating usability concerns in interactive software architecture via patterns.

The second section discusses the related word dealing with the architectures for interactive software. In Sections 3 and 4, we focus on specific ways in which internal software properties can have an impact on usability criteria. In Section 5, we attempt to provide a more general, theoretical framework for the relationships between usability and invisible software attributes. Finally, we conclude with the future investigations.

2. Related work

A large number of architectures for interactive software have been proposed, e.g., Seeheim model, model-view-controller (MVC), arch/slinky, presentation abstraction control (PAC), PAC-amadeus and model-view-presenter (MVP) (Bass et al., 1998). Most of these architectures distinguish three main components: (1) abstraction or model, (2) control or dialog and (3) presentation. The model contains the functionality of the software. The view provides graphical user interface (GUI) components for a model. It gets the values that it displays by querying the model of which it is a view. A model can have several views. When a user manipulates a view of a model, the view informs a controller of the desired change. Fig. 1 summarizes the role of each these three components for an MVC-based application.

The motivation behind these architecture models is to improve, among others, the adaptability, portability, complexity handling and separation of concerns of interactive software. However, even if the principle of separating interactive software in components has its design merits, it can be the source of serious adaptability and usability problems in software that provides fast, frequent and intensive semantic feedback. The communication between the view and the model makes the software system highly coupled and complex.

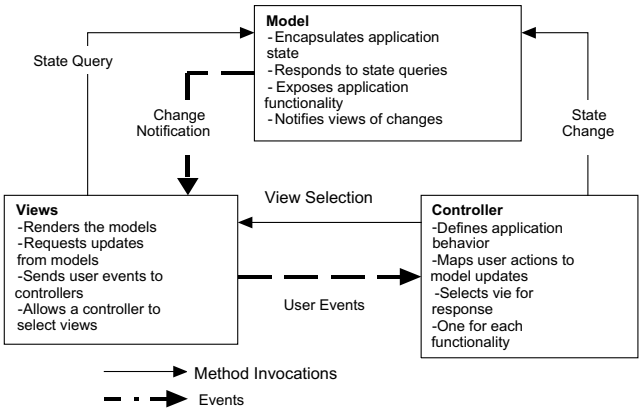

Fig. 1. The roles of the MVC architecture components.

The major weakness of this architecture is the lack provisions for integrating usability in the design of the model or abstraction components.

Len Bass and his colleagues (Bass et al., 2001) identified specific connections between aspects of usability (such as the ability to "undo") and the model response (processed by an event handler routine). Their attention was limited to single-user desktop systems only and the scenarios need to be validated in practice.

Folmer and Bosch (Folmer and Bosch, 2003) discussed a usability framework which consists of three levels: problem domain, solution domain and the usability properties level. This framework expresses the relationship between design methods that allow for design for usability at the architectural level and the evaluation tools that allow assessment of architectures for the support of usability. This research needs case studies to determine its validity and consider other application domains rather than e-commerce software.

To study these intimate relationships between the model and the interface, we proposed the following methodological framework to:

1. Identify and categorize typical design scenarios that illustrate how invisible components and their intrinsic quality properties might affect the usability,
2. Model each scenario in terms of a cause/effect relationship between (a) the attributes that quantify the quality of an invisible software entity and (b) well-known usability factors such as efficiency, satisfaction, etc.,
3. Suggest new design patterns or improve existing ones that can solve the problem described in similar scenarios,
4. Illustrate, as part of the pattern documentation, how these patterns can be applied within existing architectural models such as MVC.

3. Identifying and categorizing typical scenarios

The first step in our approach for achieving usability via software architecture and patterns is to identify typical situations that illustrate how invisible components of the model might affect usability. Each typical situation is documented using a scenario. Scenarios are widely used in HCI and software engineering (Carroll, 2000). Scenarios can improve communication between user interface specialists and software engineers who design invisible components – this communication is essential in our approach. Within our approach, we define a scenario as a narrative story written in natural language that describes a usability problem (effect) and that relates the source of this problem to an invisible software entity (cause). The scenario establishes the relationship between internal software attributes that are used to measure the quality of the invisible software entity and the external usability factors that we use for assessing the ease of use of the software systems.

The following are some typical scenarios we extracted from our day-to-day experiences and from a literature review. Other researchers also proposed other scenarios (see Kazman and Leonard, 2002). The goal of our research was not to build an exhaustive list of scenarios, but rather to propose a methodological framework for identifying such scenarios and to define patterns that be used by developers to solve such problems. The scenarios are therefore intended as illustrative examples.