LNCS 3425

Rémi Bastide
Philippe Palanque
Jörg Roth (Eds.)

Engineering Human Computer Interaction and Interactive Systems

Joint Working Conferences EHCI-DSVIS 2004
Hamburg, Germany, July 2004
Revised Selected Papers

ifip

Springer

TP11-53
E57.3
2004

Rémi Bastide Philippe Palanque
Jörg Roth (Eds.)

Engineering Human Computer Interaction and Interactive Systems

Joint Working Conferences EHCI-DSVIS 2004
Hamburg, Germany, July 11-13, 2004
Revised Selected Papers

Springer

Volume Editors

Rémi Bastide
LIIHS-IRIT, Université Toulouse I
Place Anatole France, 31042 Toulouse Cedex, France
E-mail: bastide@irit.fr

Philippe Palanque
LIIHS-IRIT, Université Paul Sabatier
118, route de Narbonne, 31062 Toulouse Cedex, France
E-mail: palanque@irit.fr

Jörg Roth
Universität Hagen
Praktische Informatik II
Universitätsstr. 1, 58084 Hagen, Germany
E-mail: Joerg.Roth@Fernuni-hagen.de

Library of Congress Control Number: 2005928449

CR Subject Classification (1998): H.5.2-3, H.5, I.3, D.2, H.3, H.4, K.4, F.3

ISSN 0302-9743
ISBN-10 3-540-26097-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-26097-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 11431879 06/3142 5 4 3 2 1 0

Preface

As its name suggests, the EHCI-DSVIS conference has been a special event, merging two different, although overlapping, research communities: EHCI (Engineering for Human-Computer Interaction) is a conference organized by the IFIP 2.7/13.4 working group, started in 1974 and held every three years since 1989. The group's activity is the scientific investigation of the relationships among the human factors in computing and software engineering.

DSVIS (Design, Specification and Verification of Interactive Systems) is an annual conference started in 1994, and dedicated to the use of formal methods for the design of interactive systems. Of course these two research domains have a lot in common, and are informed by each other's results. The year 2004 was a good opportunity to bring closer these two research communities for an event, the 11th edition of DSVIS and the 9th edition of EHCI. EHCI-DSVIS was set up as a working conference bringing together researchers and practitioners interested in strengthening the scientific foundations of user interface design, specification and verification, and in examining the relationships between software engineering and human-computer interaction.

The call for papers attracted a lot of attention, and we received a record number of submissions: out of the 65 submissions, 23 full papers were accepted, which gives an acceptance rate of approximately 34%. Three short papers were also included. The contributions were categorized in 8 chapters:

Chapter 1 (Usability and Software Architecture) contains three contributions which advance the state of the art in usability approaches for modern software engineering. Bonnie John and her colleagues discuss that, in contrast to other software quality attributes such as performance, reliability and maintainability, usability is not usually tackled at the software architecture level. Their contribution is to propose usability-supporting architectural patterns, assorted with sample solutions. The second paper, by Brinkman et al., proposes three usability measures designed to be applied in a component-based environment. These measures can be objective, based on event logs, or subjective, obtained through questionnaires. An experimental study assessing the value of these measures is also described. The third paper, by Folmer and her colleagues, also deals with the relationships between usability and software architecture. They show how explicit evaluation of usability during architectural design may reduce the risk of building a system that fails to meet its usability requirements and may prevent high costs incurring adaptive maintenance activities once the system has been implemented.

Chapter 2 is devoted to issues regarding task modelling, which is a traditional topic of choice for both the EHCI and DSVIS series of conferences. The paper by Dittmar et al. investigates the slow adoption of task modelling by software practitioners. A thorough examination of the leading-edge tools for task modelling reveals how this situation can be improved by better integration of scenario-based design elements. The work of Clerckx et al. investigates the improvement that can be brought to usual task, environment and dialogue models by tackling the new application domain of context-sensitive user interfaces. The paper by Eicholz et al. explores the relationships between task modelling and workflow, or business process modelling.

Chapter 3 is concerned with the "browsing and searching" application domain, which is of high industrial relevance considering the current interest in Web-based applications. Ormerod et al. present new browser concepts to support the sharing of digital photographs and also report on the combined use of ethnographic, experimentation and design methods they used for their project. Gonçalves and Jorge propose a new classification scheme for document retrieval systems, where users "tell a story" about their document, in order to make the later retrieval of the document more natural.

Chapter 4 deals with model-based approaches. It is made up of six contributions, making it the longest chapter of the book, witness to the fact that the definition and use of models is at the core of the EHCI-DSVIS community. Campos and Nunes, in this chapter's first paper, emphasize the need for a better integration of models and tools. They present a new UI specification language bridging the gap between envisioned user behavior and concrete user interfaces. Macías and Castells bring the field of programming-by-example to the domain of Web-based applications by detecting iteration patterns in user behavior and generating a programmatic representation of a user's actions. Navarre et al. integrate two different notations in order to offer a tool-supported approach for the prototyping of advanced multimodal applications. Limbourg and his colleagues apply their USIXML language to show how a user interface can be specified and produced at and from different, and possibly multiple, levels of abstraction while maintaining the mappings between these levels. The chapter is concluded by two short contributions: In the paper by Schaefer et al., a novel dialogue model for the design of multimodal user interfaces is proposed. Ziegler and Specker conclude by proposing the use of "Navigation Patterns," pattern systems based on structural mappings.

Chapter 5 is devoted to a rapidly developing application domain, ubiquitous computing. Borkowski et al. propose several software tools with the assorted interaction techniques to develop multisurface computer-augmented environments. Evreinov and his colleagues explore the use of vibro-tactile interaction, especially useful for new mobile devices such as palmtop computers.

Chapter 6 is called "Bridging Viewpoints": this refers to an ongoing activity of the IFIP 2.7/13.4 working group, which is to find ways to reconcile the fundamental paradigms of user-centered design and software engineering. For instance, Blandford, Green and Connel analyze the misfits between the user's conceptualization of the domain and device with which they are working and the conceptualization implemented within those systems. Barbosa et al. discuss the role of an enhanced extended lexicon as a shared communicative artefact during software design. They describe how it may act as an interlingua that captures the shared understanding of both stakeholders and designers. López-Jaquero et al. contribute a short paper on a design process for adaptive interfaces.

Chapter 7 is concerned with the emerging application domain of plastic and adaptive interfaces. Increasingly often, the same application has to be delivered on widely different platforms, ranging from a complete workstation to a PDA or a cell phone. Clearly, advances in design approaches are needed to avoid redesigning the user interface from scratch for each platform. Dobson's work is concerned with laying out such principles, in particular for pervasive computing systems. Calvary and her colleagues present a software widget explicitly dealing with plasticity of the user interface. Gilroy and Harrison propose the incorporation of interaction style into abstract UI specification, in order to accommodate with different UI platforms. Correani et al. present a new version of the TERESA tool supporting flexible development of multidevice interfaces.

Chapter 8 (Groupware) concludes the book with two papers, both concerned with supporting collaborative software construction. Wu and Graham present the Software Design Board, a prototype collaborative design tool supporting a variety of styles of collaboration and facilitating transitions between them. Gutwin et al. explore ways to improve group awareness in collaborative software design.

The conference was held in the beautiful, quiet and secluded Tremsbüttel Castle, near Hamburg, Germany, providing a studious atmosphere propitious to after-hours discussion. As usual for the EHCI conference series, the discussion that followed each paper presentation was transcribed, revised and appended to the edited version of the paper. From these, the reader may catch a glimpse of the lively debates that were held at the conference.

Rémi Bastide
Philippe Palanque
Jörg Roth

Programme Committee

¥736.32元

Table of Contents

Bringing Usability Concerns to the Design of Software Architecture[1]

Bonnie E. John[1], Len Bass[2], Maria-Isabel Sanchez-Segura[3], Rob J. Adams[1]

[1] Carnegie Mellon University, Human-Computer Interaction Institute, USA
{bej, rjadams}@cs.cmu.edu
[2] Carnegie Mellon University, Software Engineering Institute, USA
ljb@sei.cmu.edu
[3] Carlos III University of Madrid, Computer Science Department, Spain
misanche@inf.uc3m.es

Abstract. Software architects have techniques to deal with many quality attributes such as performance, reliability, and maintainability. Usability, however, has traditionally been concerned primarily with presentation and not been a concern of software architects beyond separating the user interface from the remainder of the application. In this paper, we introduce usability-supporting architectural patterns. Each pattern describes a usability concern that is not supported by separation alone. For each concern, a usability-supporting architectural pattern provides the forces from the characteristics of the task and environment, the human, and the state of the software to motivate an implementation independent solution cast in terms of the responsibilities that must be fulfilled to satisfy the forces. Furthermore, each pattern includes a sample solution implemented in the context of an overriding separation based pattern such as J2EE Model View Controller.

1. Introduction

For the past twenty years, software architects have treated usability primarily as a problem in modifiability. That is, they separate the presentation portion of an application from the remainder of that application. This separation makes it easier to make modifications to the user interface and to maintain separate views of application data. This is consistent with the standard user interface design methods that have a focus on iterative design – i.e. determine necessary changes to the user interface from user testing and modify the system to implement these changes. Separating the user interface from the remainder of the application is now standard practice in developing interactive systems.

Treating usability as a problem in modifiability, however, has the effect of postponing many usability requirements to the end of the development cycle where they are overtaken by time and budget pressures. If architectural changes required to

[1] This work supported by the U. S. Department of Defense and the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298.

implement a usability feature are discovered late in the process, the cost of change multiplies. Consequently, systems are being fielded that are less usable than they could be.

Recently, in response to the shortcomings of relying exclusively on separation as a basis for supporting usability, several groups have identified specific usability scenarios that are not well supported by separation, and have proposed architectural solutions to support these scenarios [2,3,5,6,11]. In this paper, we move beyond simply positing scenarios and sample solutions by identifying the forces that conspire to produce such scenarios and that dictate responsibilities the software must fulfill to support a solution. Following Alexander [1], we collect these forces, the context in which they operate, and solutions that resolve the forces, into a *pattern*, in this case a *usability-supporting architectural pattern*.

In the next section, we argue that software architects must consider more than a simple separation-based pattern in order to achieve usability. We then discuss why we are focusing on forces and why the forces that come from prior design decisions play a special role in software creation. In section 4, we describe our template for these patterns and illustrate it with one of the usability scenarios previously identified by several research groups. We also comment on the process for creating these patterns. Finally, we conclude with how our work has been applied and our vision of future work.

2. Usability Requires More than Separation

The J2EE Model-View-Controller (J2EE-MVC) architectural pattern [12], appears in Fig. 1. This is one example of a separation based pattern to support interactive systems. The model represents data and functionality, the view renders the content of a model to be presented to the user, and the controller translates interactions with the view into actions to be performed by the model. The controller responds by selecting an appropriate view. There can be one or more views and one controller for each functionality.

The purpose of this pattern is explained by Sun as follows [12]: "By applying the Model-View-Controller (MVC) architecture to a Java™ 2 Platform, Enterprise Edition (J2EE™) application, you separate core business model functionality from the presentation and control logic that uses this functionality. Such separation allows multiple views to share the same enterprise data model, which makes supporting multiple clients easier to implement, test, and maintain." Modifications to the presentation and control logic (the user interface) also become easier because the core functionality is not intertwined with the user interface. A number of such patterns have emerged since the early 1980s including the original Smalltalk MVC and Presentation Abstraction Control (PAC) [8] and they have proven their utility and have become common practice.

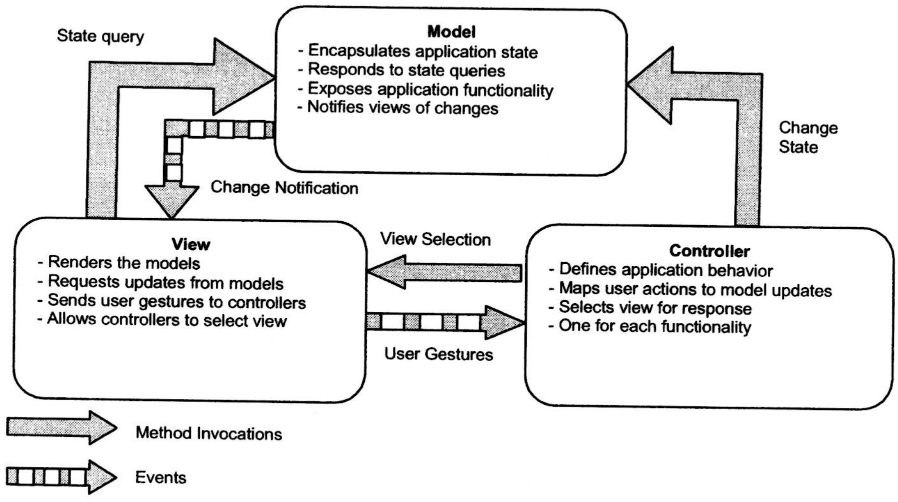
State query
Model
- Encapsulates application state
- Responds to state queries
- Exposes application functionality
- Notifies views of changes
Change State
Change Notification
View
- Renders the models
- Requests updates from models
- Sends user gestures to controllers
- Allows controllers to select view
View Selection
User Gestures
Controller
- Defines application behavior
- Maps user actions to model updates
- Selects view for response
- One for each functionality
Method Invocations
Events

Fig. 1. J2EE-MVC structure diagram (adapted from [12]).

The problem, however, is that achieving usability means more than simply getting the presentation and control logic correct. For example, consider cancelling the current command, undoing the last command, or presenting progress bars that give an accurate estimate of time to completion. Supporting these important usability concerns requires the involvement of the model as well as the view and the controller. A cancellation command must reach into the model in order to terminate the active command. Undo must also reach into the model because, as pointed out in [10], command processing is responsible for implementing undo and command processing is carried out in the model in J2EE-MVC. Accurate time estimates for progress bars depend on information maintained in the model. This involvement of multiple subsystems in supporting usability concerns is also true for the other separation based patterns. Thus, usability requires more than just separation.

3. The Forces in Usability-Supporting Architectural Patterns

The patterns work pioneered by Christopher Alexander in the building architecture domain [1] has had a large impact on software engineering, e.g. [8,10]. Following Alexander's terminology, a pattern encompasses three elements: the context, the problem arising from a system of clashing forces, and the canonical solution in which the forces are resolved. The concept of forces and their sources plays a large role in defining the requirements that a solution must satisfy.

As we mentioned above, previous work [2,3,5,6,11] focused on identifying usability scenarios not well served by separation and providing an example solution, architectural or OOD. These solutions did indeed support the scenarios, but included design decisions that were not dictated by, nor traceable to, specific aspects of the scenarios. In the work presented here, this lack of traceability is remedied by Alexander's concept of forces.

Figure 2 depicts the high-level forces acting on a system of people and machines to accomplish a task. In general, forces emanate from the organization that causes the task to be undertaken.

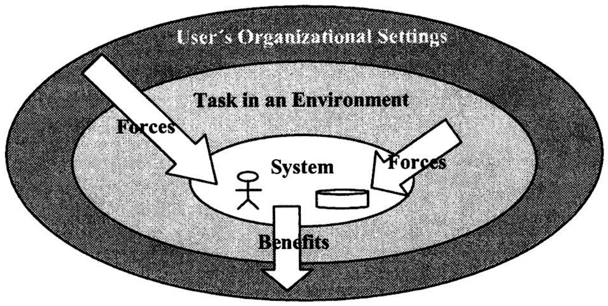

Fig. 2. Forces influencing the solution and benefits of the solution.

That is, the organization benefits from efficiency, the absence of error, creativity, and job satisfaction, to varying degrees, forcing the people to behave and the machines to be designed to provide these benefits. The costs of implementing, or procuring, software systems that provide such benefits is balanced against the value of those benefits to the organization. Although the balance is highly dependent on the specific organization and will not be discussed further, our work provides a solid foundation for determining costs, benefits, and the link between them.

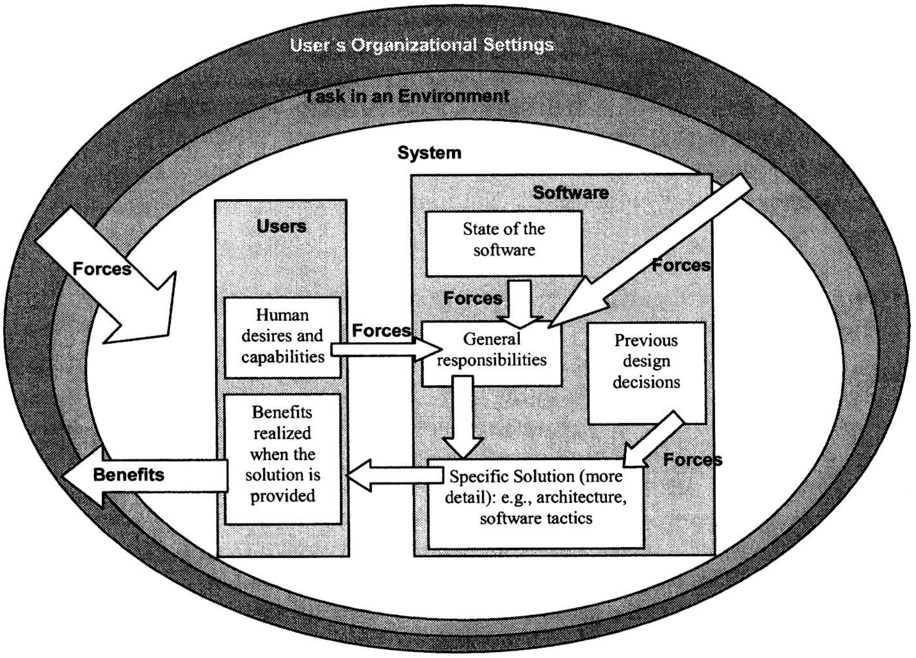
User's Organizational Settings
Task in an Environment
System
Users
Software
State of the software
Forces
Forces
Human desires and capabilities
Forces
General responsibilities
Forces
Previous design decisions
Benefits realized when the solution is provided
Forces
Benefits
Specific Solution (more detail): e.g., architecture, software tactics

Fig. 3. Forces impacting the software architecture.

Figure 3 gives more detail about the forces acting on the software that is the object of design. In addition to the general organizational forces that put value on efficiency, the reduction of errors and the like, there are specific forces placed on the design of a particular software application, which may conflict or converge, but are eventually resolved in a design solution. These forces have several sources: the task the software is designed to accomplish and the environment in which it exists, the desires and capabilities of humans using the software, the state of the software itself, and prior design decisions made in the construction of the software in service of quality attributes other than usability (e.g., maintainability, performance, security).

The first three sources of forces, task and environment, human, and software state, combine to produce a general usability problem and a set of general responsibilities that must be satisfied by any design purporting to solve the problem. These responsibilities can serve as a checklist when evaluating an existing or proposed software design for its ability to solve a given usability problem.

Combining these general responsibilities with the forces exerted by prior design decisions produces a specific solution, that is, an assignment of responsibilities to new or existing subsystems in the software being designed. If we assume, for example, the common practice of using an overall separation-based architectural pattern for a specific design, the choice of this pattern introduces forces that affect any specific solution. In this sense, our usability-supporting architectural patterns differ from other architectural patterns in that most other patterns are presented as if they were independent of any other design decisions that have been made.

We now turn to the elements of a usability-supporting architectural pattern, illustrated with an example.

4. A Template for Usability-Supporting Architectural Patterns: Example & Process

Table 1 presents a template for a usability-supporting architectural pattern, containing the context, the problem, and both a general solution and a specific solution. This template is based on the concepts in Alexander's patterns [1], past experiences teaching architectural support for usability problems [6,11], and usability evaluation of the pattern format itself. For example, the forces are listed in columns according to their source under the **Problem** section of the template. Each row of forces is resolved by a general responsibility of the software being designed. Even though the responsibilities constitute the **General Solution**, we place them in the rows occupied by the forces that they resolve because this spatial configuration emphasizes the traceability of responsibilities back to the forces. In the **Specific Solution** we repeat the general responsibilities rather than simply pointing to them, because it is easier for the designer to read the text of the general responsibility in proximity to the prior design decisions than to continually switch between different sections of the pattern template. As with the general responsibilities, the rows in the **Specific Solution** provide a traceability lacking in our previous presentations of similar material.