Architectural Patterns for Usability

Len Bass and Bonnie E. John
Software Engineering Institute/Human Computer Institute
Carnegie Mellon University
Pittsburgh, Pa, 15213 USA
412.268.6763
{len.bass,bonnie.john}@cmu.edu

ABSTRACT

Facets of usability that require architectural support such as cancellation, undo, and progress bars are identified. For each facet, an architectural pattern is described that supports the achievement of the facet. Facets of usability that require architectural support are difficult to add after the initial design of a system has been completed and, hence, it is critical to identify these facets prior to initial system design.

Keywords

Software Architecture, Usability, Architectural patterns

1. Introduction

Usability is important for interactive systems. No one would dispute this. Hence it is important to design systems to support usability. Achieving usability for an interactive system depends on a number of factors. The functions provided by the system must accomplish the intended tasks, fit within the work context of the user, be understandable and clearly visible through the user interface. The input and output devices and layout must be correct for the class of users of the system and their physical work environment. The software architecture must be designed to support the user in the achievement of the necessary tasks.

In recent work undertaken at the Software Engineering Institute, we have focussed on determining the facets of usability that require software architectural support.

2. Software Architecture

Software architecture comprises the earliest design decisions for a system. They underlie subsequent decisions and are the most difficult to modify. In previous work on supporting usability through software architecture, the focus was on supporting the iterative design process by allowing the presentation and input/output devices to vary. This was accomplished by separating the presentation from the remainder of the application. Separation is an architectural technique that restricts the impact of modifications. Separating the presentation thus makes modification of the user interface relatively easy after user testing reveals problems with the interface. MVC, PAC, Seeheim, and Arch/Slinky are examples of architectural patterns based on separating the user interface from the remainder of the system. See Chapter 6 of [2] for a discussion of these patterns and for references to additional sources.

Separation of the user interface has been quite effective, and is commonly used in practice, but it has problems. These problems are, first, there are some aspects of usability that are not impacted by separation and, secondly, the later changes are made to the system, the more expensive they are to achieve. Forcing usability to be achieved through modification means that time and budget pressures will cut off iterations on the user interface and result in a system that is not as usable as possible.

3. An example of our approach

Our approach is more proactive and complementary to separation. We identified a set of specific usability scenarios that have software architectural implications. This set currently contains 26 scenarios. Although we make no claim, yet, that these scenarios are exhaustive, they seem to cover much of what people typically mean by usability (e.g., efficiency, learnability, some aspects of user satisfaction). For each one of these aspects, we identified an architectural pattern that enables the achievement of that scenario. We then organized these scenarios and patterns in a fashion we will discuss after we introduce an example.

Consider cancellation. Users require the capability to cancel an active command for most commands. It does not matter why they wish to cancel, they may have made a slip of the mouse or changed their mind, they may have been exploring the effects of an unknown command but do not wish to wait. Once a command has been cancelled, the system should be returned to the state it was in prior to the issuance of the command. One architectural pattern for achieving cancellation is for the system to maintain a thread of control that listens for the user to issue a cancellation command. This thread should be separate from the thread executing the command Furthermore, the component that implements the issued command must save its state prior to beginning work on the command so that this state can be restored if the user does, in fact, cancel the command. The architectural pattern we have produced for

the cancellation scenario discusses these and other aspects of supporting cancellation.

Other examples of usability facets that require software architectural support are undo, progress bars for feedback to the user, and propagation of known information to input forms. This last scenario would support, for example, allowing the user to input name or address once and then automatically filling in this information in portions of the user interface that require it.

Once we generated our 26 scenarios, we then organized them in two separate fashions. We organized them into a hierarchy that represents benefits to the organization from applying these scenarios. For example, "accommodates users' mistakes" and "supports problem solving" are entries in this hierarchy. Each scenario was placed into one or more positions in this hierarchy. The second organization was into a hierarchy of software architectural mechanisms. "Recording state" and "preemptive scheduling" are two of the items in this hierarchy. Again, each scenario was placed into one or more positions in the hierarchy.

The hierarchies were then treated as the two axes of a matrix and each scenario was placed in the appropriate cells. That is, if a scenario provided a single benefit to the users and required a single architectural mechanism to be in place, then it would show up in a single cell in the matrix indexed by its position in the two hierarchies. Most scenarios provide more than one benefit and require more than one architectural mechanism and therefore occupy several cells in the matrix. Figure 1 gives a small portion of this matrix.

For the designer, the matrix can be used in two fashions. If the desired benefits to the organization are known, then the matrix can be used to determine which scenarios are relevant to that benefit and which software architectural mechanisms should be included in the system to achieve those benefits. If one scenario is to be included in the system then an examination of the matrix will yield other scenarios that could be incorporated into the system at a small incremental cost. See [4] for a full description of the scenarios, the hierarchies, the architectural patterns, and the matrix.

We have used these scenarios in three different design or analysis situations on two commercial and one prototype systems. The design context was using the Attribute Driven Design method [1] and the analysis context was using the Attribute Trade-off Analysis Method [5]. The systems for which the usability scenarios were applied were a large financial system, a driver information system, and a multi-user meeting support system. That is, we have applied these in a variety of different domains. In each case, the scenarios discovered issues in the user interface design that would have otherwise gone undiscovered, at least through initial implementation.

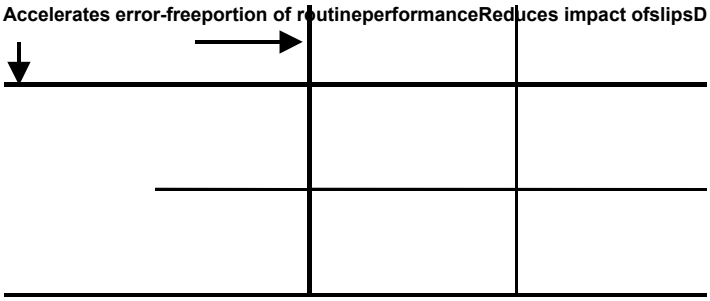


Figure 1: A sample of four cells of the benefit/mechanism hierarchy. The full hierarchy is 9 benefits by 13 mechanisms, or 117 cells.

4. Trade-offs with other attributes

Usability is important but it must be considered in the context of other important attributes. That is if a particular architectural pattern is used to support usability, what are its performance, availability, security, and modifiability impacts. The usability work we are reporting on here is a portion of a larger project to understand the trade-offs inherent in achieving any quality attribute. [3] gives an overview of this larger effort.

5. References

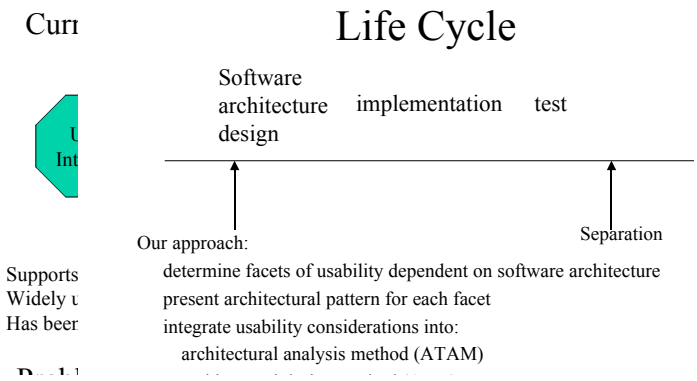
1. Bachmann, F. and Bass, L. "Designing Software Architecture for Quality: the ADD Method", OOPSLA 2001 Conference Companion

2. Bass, L., Clements, P. and Kazman, R., "Software Architecture in Practice", Addison-Wesley Longman, Reading, Ma. 1998

3. Bass, L., Klein, M., and Bachmann, F., "Quality Attribute Design Primitives", CMU/SEI-TN-2000-017, December, 2000.

4. Bass, L., John, B.E., and Kates, J., "Achieving Usability Through Software Architecture", CMU/SEI-TR-2001-005. March, 2001.

5. Clements, P., Kazman, R., and Klein, M. "Evaluating Software Architectures: Methods and Case Studies", Addison-Wesley Longman, 2001.

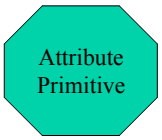


Examples

- Cancellation
  - pre-emptive scheduling
  - recording of state of components that may be cancelled
- Undo
  - recording of state before any command by all important components
- Information reuse (e.g. user's name propagated to all relevant forms)
  - global data dictionary and repository
- Progress bars for feedback
  - system state available and current

Must Understand Multiple Attributes

Basis for achieving one quality attribute
usability
performance
availability
modifiability
security



Understand primitive's effect on other quality attributes.
E.g. what are usability, performance, security, modifiability implications of using redundancy to support availability