



Linking usability to software architecture patterns through general scenarios

Len Bass^{a,*}, Bonnie E. John^{b,1}

^a *Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA*

^b *Human-Computer Interaction Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA*

Received 9 January 2002; received in revised form 9 April 2002; accepted 26 April 2002

Abstract

Usability is an important quality attribute to be considered during software architecture design. Up to this point, usability has been served only by separating a system's user interface from its functionality to support iterative design. However, this has the effect of pushing revisions to achieve usability toward the end of the software development life cycle. Many usability benefits link directly to a variety of architectural tactics in addition to separation of the user interface and these benefits can be discovered early in the life cycle. For each of 27 scenarios, we identified potential usability benefits a user could realize and an architectural pattern that supports achievement of those benefits. We organized the scenarios into an emergent hierarchy of potential benefits to the user and into an emergent hierarchy of architectural tactics used in the supporting patterns. The range of architectural tactics identified in this hierarchy demonstrates that separation is far from the only architectural tactic necessary to support usability. We present techniques that permit important usability issues to be addressed proactively at architecture design time instead of retroactively after user testing. © 2002 Elsevier Science Inc. All rights reserved.

Keywords: Usability; Software architecture; Software patterns; Scenarios

1. Introduction

Stakeholders recognize that usability is important for most interactive systems. In the past 20 years, a substantial amount of research has gone into supporting the detailed design of the user interface (UI). Supporting usability concerns by constructs of software architecture has also been an avenue of research since the 1980s. Architectural patterns such as the model view controller (MVC) and presentation–abstraction–control (PAC) (Buschmann et al., 1996) have been developed to simplify the modification of the detailed UI design. The area of User Interface Management Systems (UIMSS) (Pfaff, 1983) had as its goal simplifying the construction of the UI and, consequently, improving the ability to perform iterative development.

Implicit in these methods and their focus on iterative development, is the assumption that getting the details of the UI correct is equivalent to making a system usable. This assumption implies that being able to modify aspects of the UI such as menu structure, dialog boxes, and the content and presentation of information to the user, in the face of usability analyses or data, is the primary path to producing a usable system. From an architectural point of view, it turns usability concerns into modifiability concerns. Separation of the UI from the functionality of the system is sufficient to support modifiability. However, having a good UI is only one aspect of producing a usable system. Giving the user the ability to perform “undo”, for example, is also important for usability and one that is not tied to the design of the UI (except for providing the facility for invoking the undo command) or supported by separation.

There has been little research on software architectural support for those aspects of usability that are not connected with iterative development and design of the UI. We present an initial investigation into developing software architectural support for those neglected

* Corresponding author. Tel.: +1-412-268-6763; fax: +1-412-268-5758.

E-mail addresses: ljb@sei.cmu.edu (L. Bass), bej@cs.cmu.edu (B.E. John).

¹ Tel.: +1-412-268-7182.

aspects of usability. Our approach is to identify a collection of usability scenarios that involve more than detailed design of the UI and have architectural impact beyond separation in support of modifiability. We present software architectural patterns in the sense of Buschmann that will achieve the scenarios. The existence of the scenarios acts as a checklist for both the software designer and the usability engineer so they can discuss the desirability of achieving the scenario in a particular system. The suggested pattern provides the software engineer with a basis to estimate the effort (cost) involved in realizing the scenario.

We begin by exploring in more detail our assertion that the basis for the existing work on the connection between usability and software architecture is supporting iterative development. We then present our approach in terms of scenarios and architectural patterns to achieve the scenarios. We describe how we developed two hierarchies based on the scenarios and the patterns and give an example of using these hierarchies in practice. We close by discussing the limitations of the work we present here and give areas for future work.

1.1. Prior work

1.1.1. Software architectural patterns and user interface reference models

During the 1980s and into the early 1990s, both practitioners and researchers developed a collection of UI reference models (software architectural patterns in today's jargon). Beginning with MVC and the Seeheim Model these models evolved over time to a common basis. See Chapter 6 of (Bass et al., 1998) for a discussion of these models and their evolution. The one element all of these models have in common is the separation of the UI from the remainder of the application (the core functionality). Separation of concerns is a basic engineering technique that divides problems into distinct sub-problems with minimal overlap. The smaller sub-problems are, presumably, easier to solve and the fact there is minimal overlap enables integrating the solution of the parts into a solution of the whole. Making the overlap minimal is the key to making separation of concerns successful. In the software engineering world, separation is done in order to make modifications to each of the smaller portions easier. Modification of the UI is the essence of iterative development. An interface is developed, the interface is analyzed and tested with users, modifications are discovered and made, and this process is expedited by separating the UI.

In 1992, the state of the practice was summarized thus:

A common approach for developing such models (MVC, PAC, and others) is to examine the func-

tionality of an interactive system, decide that separating the UI functionality from other functionality is the most important design goal, and derive an architecture that supports this separation. (UIMS Developers Workshop, 1992, p. 32.)

It is worth noting that these models are UI reference models. No one claimed that these models solved the problems of usability that were not tied to the UI. Nielsen (Nielsen, 1993) is an example of an author who discusses the non-UI aspects of usability but he (and the others) do so without discussing any implementation considerations.

Since the software architecture is the artifact that embodies the earliest design decisions, it is very difficult to modify once designed. Our search is to find those aspects of usability that affect the software architecture once the UI has been separated from the remainder of the system. These are the aspects of usability difficult to instill into the system if their absence has been discovered during the iterative design process.

1.1.2. User interface management systems and user interface development environments

UIMSs were systems predicated on separating the UI from the functional core that were developed in parallel with the architectural patterns just discussed. They usually consisted of a UI builder or specification language together with a run-time infrastructure that supported the integration with the functional core and the incorporation of the UI specified through the builder or specification language. A special case of UIMS research investigated User Interface Development Environments (UIDEs) (Foley et al., 1991). UIDEs attempted to build a model of the UI and use this model to construct the instance of the UI relevant in particular contexts. They extended UIMSs in terms of the support for the construction process and the run time support for the UI but were identical in terms of the basic underlying architecture.

UIMSs were based on the assumption that the run-time infrastructure captured all of the common aspects of the UI and the UI specification captured all of the variable aspects. They also were based on separation of the UI as we have discussed above.

By assuming that all of the commonalities of a good UI were captured in a run-time infrastructure, UIMSs assumed that there was one basic solution that all good UIs exhibit.

We do not believe this. In our search for scenarios, we were very careful to avoid the “one-size-fits-all” phenomena. That is, when we have two similar scenarios, we kept them separate as long as the solutions were distinct rather than attempting to combine the scenarios and their solutions. Our belief is that software designers

are capable of combining distinct patterns and as long as a possibility exists that not all systems will, by necessity, implement both scenarios we kept the scenarios distinct.

1.2. Our approach

With this background, our approach is as follows:

1. We identified a collection of scenarios. We identified this collection through a search of the literature, through discussions with colleagues, and through personal experience. The collection has the following characteristics.
 - (a) The scenarios are common to many interactive software systems. By necessity, these scenarios are not related to the domain functionality of any one system.
 - (b) The scenarios are architecturally significant. By this we mean that the solution to each scenario affects the functional core in a software architectural pattern based on separation of the UI.
 - (c) The scenarios are distinct either with respect to their subject or their solution.
2. Each scenario has an architectural pattern that provides a solution to implementing the scenario. We do not claim that the pattern provided is the only solution, or even the best solution for every system, but that it is one possible solution to consider.
3. We organized the scenarios into a hierarchy based on the benefits to the user from realizing the scenario.
4. We organized the patterns into a hierarchy based on the software architectural “tactics” used within the pattern. Tactics is a coined word in this context and we discuss its use in a subsequent section.

We now discuss our scenarios in more detail.

2. General usability scenarios

Quality scenarios have been widely used both in analyzing for software architectures (Clements et al., 2001) and for designing software architectures (Bass et al., 2001b). Our first step in investigating the relationship between usability and software architecture was to generate scenarios that expressed a general usability issue and seemed to have architectural implications. For example, a common usability scenario is that a user changes his or her mind about issuing a command and wants to cancel that command before it has completed. This is generally applicable to many software systems and has architectural implications because the system must be attentive to the cancel command and be able to restore state.

We generated scenarios in several ways. We read several standard HCI textbooks and used their examples and definitions of usability to inspire scenarios (e.g. (Gram and Cockton, 1996; Newman and Lamming, 1995; Nielsen, 1993; Shneiderman, 1998)). We generated scenarios from our own experiences. We discussed scenarios with colleagues and we literally asked people we sat next to on buses “have you had any problems with computers lately?” Thus, the initial generation process was not systematic or comprehensive, but it was sufficient to produce substantial evidence that the link between usability benefits and software architecture is much deeper than simple separation of UI from core functionality. The full set of scenarios that we are currently considering can be found in Appendix A.

2.1. Software architectural patterns

For each scenario we generated a software architectural (SA) pattern to achieve a solution to the scenario. The SA patterns are described fully in (Bass et al.,

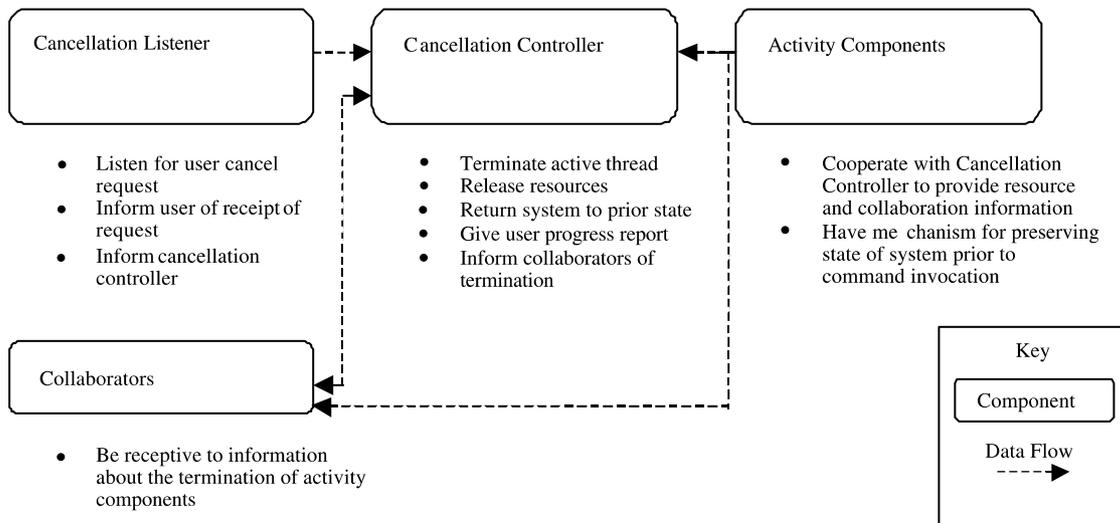


Fig. 1. Module view of the pattern for the cancellation scenario.

2001a) for those who wish more details than we are able to present here. Our goal in presenting these patterns was to avoid a one-size-fits-all solution and provide only a minimum solution for the particular scenario at hand. One of the reasons for the failure of the UIMS research cited earlier was the attempt by its proponents to solve all of the usability problems that a system might ever have. This made the UIMSs inherently cumbersome and unattractive to developers.

Fig. 1 shows a module view of the SA pattern for the cancellation scenario. The key to this pattern is that a component cannot be responsible for managing its own cancellation because it may be blocked when the cancel command is issued. Hence, there is a listener component that is always listening for the user to issue the cancel command and a Cancellation Controller that is responsible for actually performing the cancellation and cleaning up after the component that is being cancelled.

3. Classifying the scenarios

After generating about two dozen scenarios, we classified them in two ways. We first looked at all the scenarios from the point of view of what usability benefits could be delivered to a user if a good solution to the scenario were implemented and then we looked at the scenarios from the point of view of classifying the architectural patterns. In both of these cases we used a bottom-up process called affinity diagrams (Beyer and Holtzblatt, 1998). We now discuss these two classifications.

3.1. Usability classification

We classified the scenarios from the bottom up rather than starting with an existing definition of usability and

sorting the scenarios into it, because it was not clear that architecturally sensitive scenarios would cover the typical range of usability benefits. However, the resulting hierarchy, shown in Fig. 2, is similar to organizations of usability given by other authors (e.g. Newman and Lamming, 1995; Nielsen, 1993; Shneiderman, 1998), encompassing benefits of efficiency, problem-solving and learnability, and user satisfaction. One item in the hierarchy that differs from other authors' characterizations is reducing the impact of system errors. Several scenarios highlighted the inevitable occurrence of system errors (e.g., networks going down, systems crashing). A carefully designed architecture can mitigate the damage of such errors to the users' work. Each scenario occurs in one or more positions in the hierarchy; the rationale for each assignment can be found in (Bass et al., 2001a). We provide the rationales for the canceling command scenario in Fig. 3.

3.2. Software architectural pattern classification

After completing the affinity diagram from the point of view of usability benefits, we began again with the unordered set of scenarios and did another affinity diagram to uncover similarities in the SA patterns for implementing a good solution to these scenarios.

When examining the original set of scenarios from this point of view, we sometimes found that a scenario had no substantive implications for the architecture of the system and we removed it from further consideration. This examination also revealed that some scenarios should be split in two because different SA patterns highlighted different aspects of a scenario, and, conversely, that some scenarios should be merged into one because their SA patterns were identical.

The bottom-up classification process found similarities in what we call "architectural tactics" inherent in the

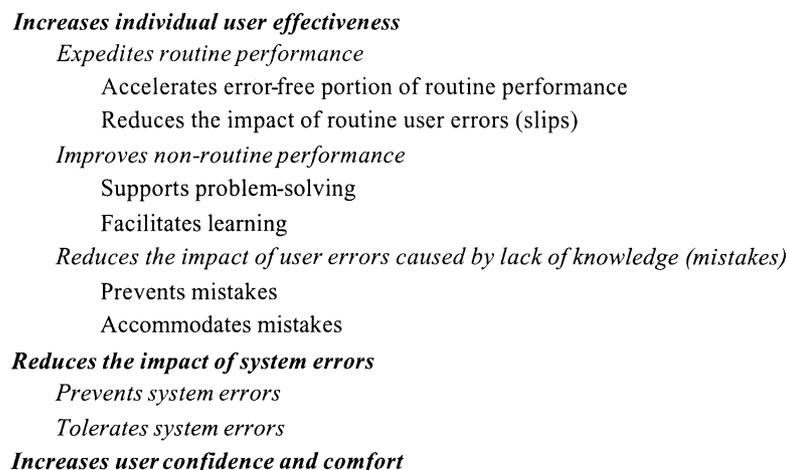


Fig. 2. Usability benefits hierarchy.

- Increases individual effectiveness**
Expedites routine performance
Reduces impact of slips
 Cancellation reduces the impact of slips by allowing users to revoke accidental commands.
- Increases individual effectiveness**
Improves non-routine performance
Supports problem-solving
 Cancellation facilitates problem-solving by allowing users to apply commands and explore without fear, because they can always abort their actions.
- Increases individual effectiveness**
Reduces the impact of user errors caused by lack of knowledge (mistakes)
Accommodates mistakes
 Cancellation accommodates user mistakes by allowing users to abort commands they invoke through lack of knowledge.
- Reduces the impact of system errors**
Tolerates system errors
 Cancellation helps users tolerate system error by allowing users to abort commands that aren't working properly (for example, a user cancels a download because the network is jammed).

Fig. 3. Allocation of cancel command to benefit hierarchy.

SA patterns. These tactics are widely used software engineering approaches either for structuring systems or for managing the execution of systems. Examples of our architectural tactics include “separation” and “replication”. A particular pattern will provide an implementation of one or more tactics. For example, the pattern proposed to satisfy the cancellation scenario uses the tactic of “pre-emptive scheduling” so that the listener component can respond to the request for cancellation as well as the tactic of “recording” so that the cancellation component can restore the system to its state

- Separation**
Encapsulation of function
Data from commands
Data from the view of that data
Authoring from execution
- Replication**
Data
Commands
- Indirection**
Data
Function
- Recording**
Preemptive scheduling
- Models**
Task
User
System

Fig. 4. Software architectural tactics hierarchy.

prior to the invocation of the command that was cancelled.

Fig. 4 shows the hierarchy that resulted from the bottom up classification process. We cannot compare this to similar software architecture hierarchies because, to our knowledge, no comparable hierarchies exist.

The bulk of this hierarchy should be of no surprise to software engineers, as this hierarchy contains many familiar tactics: various forms of separation, replication, indirection, etc. However, the fact that these tactics have implications for the usability of the system may be surprising. This range of tactics belies the assumption that separating the UI from the core functionality is sufficient to support usability at architecture-design time. Clearly, other tactics must be in place in the architecture if a usable product is to be delivered to the customer.

As with the benefits hierarchy, the affinity process produced one unusual item in the hierarchy, the notion of models. To provide good solutions to several scenarios, the system should include a model of the task (e.g., a spell-checker in a word-processor can use the frequency of occurrence of words in the language to order suggestions), the user (a spell-checker can also use the most common typing errors, e.g., transposition of characters), and sometimes of the system itself (e.g., progress bars need to “model” the functioning of the system to estimate time to completion).

Each SA pattern uses one or more architectural tactics. The rationale for the assignment of tactics to SA patterns can be found in Bass et al. (2001a). An illustration of the rationale for the canceling command is given in Fig. 5.

Recording

The cancellation component must record its initial state so that the system can be returned to the state prior to the invocation of the cancelled components.

Preemptive Scheduling

To adequately implement cancellation, the cancellation listener and cancellation controller must occupy independent threads.

Models/System

After a command has been cancelled, the system must consult an explicit model of itself in order to predict state restoration time and to report progress.

Fig. 5. Architectural tactics found in the software architectural pattern that provides a solution to canceling commands.

4. The benefit/tactic matrix

With each scenario categorized into what usability benefits a good solution has the potential to deliver and what architectural tactics are required to implement a good solution, we constructed a two-dimensional matrix of benefit by tactic (Fig. 6).

The matrix can be used by a design team to evaluate or design an architecture in three ways. First, the team may decide that a particular scenario is important to the design goals of their product. They may identify important scenarios from examining our list directly, from performing a Heuristic Evaluation (e.g., Nielsen's heuristic of providing "emergency exits" maps to our cancellation scenario (Nielsen and Mack, 1994)), or from other scenario-based design techniques that may produce specific scenarios analogous to our general ones. The team can then go into the matrix and find the cells occupied by that scenario. From the scenario's positions in the matrix, the design team can understand the potential benefits to the user and understand the tactics necessary to include in the SA in order to achieve those benefits. Thus, our list of scenarios differs from checklists or heuristics in other HCI literature because they not only raise consciousness about what scenarios should be supported (e.g., "provide undo"), but also supply guidance as to how to implement a good solution to each scenario. Second, the design team can use the matrix by identifying which usability benefits are most important in their situation. For example, if they are designing a walk-up-and-use information kiosk then supporting problem solving or learning may be more important than efficiency. Alternatively, if the team is designing a specialized information system for well-trained, long-term users, they may value efficiency over learnability. The team can then examine only those scenarios that appear in the columns populated by the valued benefits. They then translate our general scenarios into specific scenarios for their system and follow the rows to specific architectural tactics to implement solutions to those scenarios.

Finally, if an architecture is already proposed, the design team can enter the matrix at the rows that contain the tactics in that architecture. By examining the sce-

narios in that row, they may discover additional scenarios that could be solved with little additional effort, thereby accruing additional usability benefits for their users. This last method for using the matrix is more speculative than the first two because if a tactic exists in service of a particular scenario, there is no guarantee that the same implementation of that tactic will serve another scenario. For example, if one component records state to allow easy evaluation of a system, it is not necessarily the same recording needed to support cancellation or undo. However, we anticipate that thinking about architectural tactics in a systematic way through the matrix will facilitate the team asking relevant questions at architecture-design time and making informed decisions about relative benefits and costs of architectural design decisions. Furthermore, one of our applications of this work provides evidence that having tactics to support one scenario facilitates the supporting of another. We discuss this application in the next section.

An important point about the benefit/tactic matrix is its density. If the matrix were very sparse, any particular usability benefit could be mapped to a single tactic and a list of benefit/tactic tuples would be sufficient to capture architectural design knowledge. If the matrix were very dense, then a single complex architecture could be designed once that would solve all usability problems for future systems. This, indeed was the philosophy behind UIMs, but was resisted by practicing developers as too cumbersome. In reality, the matrix is neither particularly sparse nor particularly dense. This means that providing a solution to most scenarios promises more than one usability benefit (22 out of 27), but that few scenarios can be solved with only one architectural tactic (6 out of 27). The number of tactics required to implement a solution to a scenario ranges from 1 to 5, with an average of 2.3. Therefore, the benefit/tactic matrix indicates that developers' resistance to UIMs was probably justified because not every usability problem requires all architectural tactics. Thus, the number of decisions about architectural design to produce a usable system is far greater than simply separating the UI from the functionality. It requires customization of the architecture for each design situation, with guidance embodied in architecture patterns, but the potential rewards are also great.

Usability Benefits →		Increases individual effectiveness						Reduces impact of system errors		Increases confidence and comfort
		Expedites routine performance		Improves non-routine performance		Reduces impact of mistakes		Tolerates system errors	Prevents system errors	
		Accelerates error-free portion	Reduces impact of slips	Supports problem-solving	Facilitates learning	Prevents mistakes	Accommodates mistakes			
Architectural Tactics ↓	Separation	Encapsulation of function	4, 13, 14, 15, 20, 23		4, 13, 20	4, 13, 20	4, 13, 20	9, 14		
	Data from the view of that data	12, 13, 24, 25	12	12, 13, 22, 24, 25, 26	12, 13, 24	12, 13, 22, 24	12			12
Data from commands	1, 24, 25	5, 17	5, 17, 24, 25, 26	5, 17, 24	1, 5, 17, 24	1, 5, 17			17	
Replication	Authoring from execution	1, 2	2			1, 2	1, 2			
	Data	16								
Commands	2	2	22		2, 22	2				
Indirection	Data	7, 11, 14	11	7, 11			14			
	Function	6, 14, 20, 27	27	6, 20	20	20, 27	14		6	27
Recording		2, 7	2, 3, 21	3, 7, 21		2	2, 3, 21	3, 8		
Preemptive Scheduling		15, 18, 19	3, 5, 17, 18	3, 5, 10, 17	5, 10, 17	5, 17, 19	3, 5, 17	3		17, 18
Models	Task	18, 19	5, 17, 18	5, 10, 17	5, 10, 17	5, 17, 19	5, 17			17, 18
	User	12, 18	5, 12, 17, 18	5, 10, 12, 17, 22	5, 10, 12, 17	5, 12, 17, 22	5, 12, 17			12, 17, 18
	System	4, 6, 19, 23	3, 5, 17	3, 4, 5, 6, 17	4, 5, 17	4, 5, 17, 19	3, 5, 17	3	6, 23	17

KEY

- | | | |
|-----------------------------------|------------------------------------|--|
| 1 Aggregating data | 10 Providing good help | 19 Predicting task duration |
| 2 Aggregating commands | 11 Reusing information | 20 Supporting comprehensive searching |
| 3 Canceling commands | 12 Supporting international use | 21 Supporting undo |
| 4 Using applications concurrently | 13 Leveraging human knowledge | 22 Working in an unfamiliar context |
| 5 Checking for correctness | 14 Modifying interfaces | 23 Verifying resources |
| 6 Maintaining device independence | 15 Supporting multiple activities | 24 Operating consistently across views |
| 7 Evaluating the system | 16 Navigating within a single view | 25 Making views accessible |
| 8 Recovering from failure | 17 Observing system state | 26 Supporting visualization |
| 9 Retrieving forgotten passwords | 18 Working at the user's pace | 27 Supporting personalization |

Fig. 6. Matrix linking potential usability benefits to software architectural tactics through 27 usability scenarios (named in the Key). More information on usability general scenarios and their assignment to cells in the matrix can be found in [4].

5. Uses in practice

The SEI has both design and evaluation methods that are based on understanding the relationship between the achievement of desired *quality attributes* and SA. Traditional quality attributes in architecture design are attributes like performance, security, and availability. Both design and evaluation methods have steps that require the generation of scenarios that characterize quality requirements for the system under consideration. Since quality attribute scenarios are already an essential portion of these methods, including usability into the methods via the scenarios listed here was easily accomplished.

We now describe our application of these results to an application of ATAM (Architecture Tradeoff Analysis Method, Clements et al., 2001) to a large commercial information system. The designers of this system

had involved professional usability engineers in the design but still our results were able to assist them.

One of the scenarios generated during the ATAM mapped to our usability scenario 22 (Working in an unfamiliar context). The particular ATAM scenario involved a helper at a central site who was called by a remote user having a problem using a customized interface. The problem, then, became, how can the helper see the same interface that the remote user sees in order to provide assistance?

The system under discussion supported multiple languages (our scenario 12). Fig. 7 shows the central server and the client machines. Observe in the benefit/tactic matrix that both scenarios utilize the tactic “separate data from the view of that data”. Within the architecture being reviewed for the ATAM, the description of the view of the data was “pushed” every night from the server to the client machine. The solution

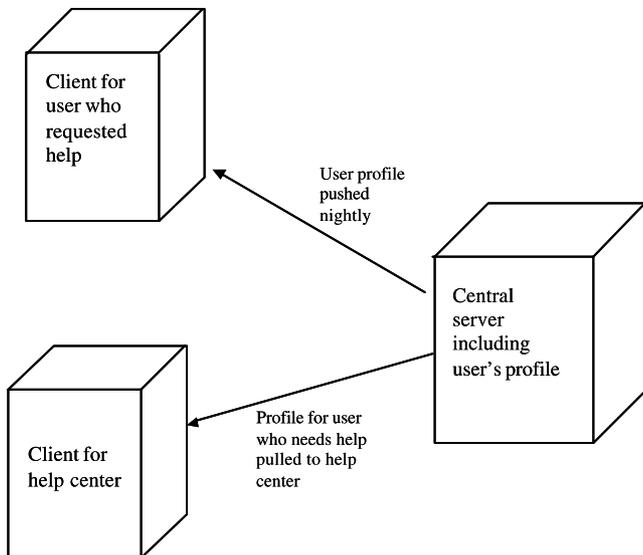


Fig. 7. System that supported internationalization via pushing user profiles modified to support working in an unfamiliar context via adding a pull feature.

to implementing scenario 22 lay in the recognition that the description of the data being used by the person needing helping must be made available to the helper. By adding a “pull” feature for the helper, the profile for the user who had requested assistance could be loaded onto the helper’s machine and the helper could see the same interface as the person requesting help. Thus, the matrix led to examination of the existing design to determine how the description of the data was made available to the user. This, in turn, led to a realization that the getting that description to the helper was the key to solving the problem. The matrix was the place where the connection between scenarios 12 and 22 was made.

6. Future work

This work is by no means complete. There are a number of activities that remain to be done to simplify the support of usability at the software architecture design stage of a system. These activities include the following.

Validating the usefulness and generality of our scenarios and quantifying the value of the potential benefits to users. This is a general goal for many HCI evaluation and design methods and is no less important in understanding the link between architecture and usability.

Fleshing out these scenarios and architectural solutions into usable HCI “patterns” similar to those in other disciplines and previous software engineering work (e.g., architecture (Alexander et al., 1977); object-oriented software (Gamma et al., 1995); software architecture (Buschmann et al., 1996); interactive music exhibit de-

sign (Borchers, 2001)). The cancellation scenario has been so expanded and summarized in (Bass and John, 2000).

Extending the list of general usability scenarios. We explicitly limited our attention to single-user desktop systems and make no claims about completeness of our scenario list. If possible, a more systematic approach to scenario generation should be developed to ensure coverage. But even without a systematic method, the set of scenarios should be extended to include other interaction paradigms like mobile and ubiquitous computers and multi-user environments. Although many of the general scenarios presented here will be applicable to other paradigms, these environments are likely to introduce their own additional usability requirements.

Understanding the effect on other attributes of usability architectural patterns. In order for a software architect to adopt a particular architectural pattern to support usability, the effect of this pattern on other quality attributes such as performance, modifiability, security and reliability should be understood. Each quality attribute community has its own analysis techniques. When a designer has to make a trade-off between, for example, performance and security, the designer will hypothesize a particular solution, analyze it for its security characteristics using security analysis techniques, analyze it for its performance characteristics using performance analysis techniques, and decide whether the hypothesized solution is acceptable. If it is not, another solution will be hypothesized and the process is repeated. If this process does not converge, then the definition of “acceptable” solution is changed so that one of the hypothesized solutions is acceptable.

It is no different when considering usability and security instead of performance and security. The designer chooses one of our scenarios, includes the associated pattern in the solution hypothesis and analyzes the result for its security characteristics using security analysis techniques. If the solution is not acceptable, another solution is hypothesized and the process is repeated.

Thus, usability plays a role in the design process exactly analogous to that played by the other quality attributes and trade-offs between usability and other quality attributes are handled using exactly the same process as trade-offs among other quality attributes. Ongoing work at the SEI is examining the relationships between all quality attributes, including usability.

Use the matrix, scenarios, and architectural patterns in a variety of real-world design situations. Our initial ventures into the real world of design have been encouraging, but additional applications of the information to design will prove the usefulness and usability of the method itself.

In particular, one potential problem with the use of general scenarios is the necessity for the software engineer and the usability engineer to make them system

specific. That is, Scenario 2 refers to the aggregation of commands. It is couched in terms “the user wishes to perform a multi-step procedure repetitively”. In particular domains, this scenario will appear in quite different forms. An example in the automotive domain is the use of “one button customization”. That is, pushing one button on a remote control will unlock the door, adjust the seat and mirror to your desired settings as well as set the radio volume and pre-set stations to be the ones you prefer. How is the usability engineer or the software architect to make this translation?

We have no definitive answer at this point, but we can cite our experience in several different settings.

We have presented this material at two masters-level classes and an ACM tutorial for professionals in both Software Engineering and Human Computer Interaction. In each case, we asked the attendees to consider several scenarios and apply them to domains with which they had experience. In every case, the people volunteering answers were able to generate scenarios within their domain that were indeed concrete examples of our general scenarios.

Future work should include a more systematic process to generate system specific scenarios but our experience in the classroom supports the belief that the generation of system specific scenarios is not an insurmountable barrier to applying this work.

7. Conclusions

Our major conclusion is that the link between SA and usability is much deeper than simply employing separation for easy modification of the UI. We have offered a collection of general usability scenarios that require architectural support beyond separation as evidence for this conclusion.

Given our use of the general scenarios and benefit/tactic matrix in architecture design and analysis processes, we further conclude that the scenarios, their architectural patterns, and the matrix, are promising tools to improve the development of real systems. Although supporting usability aspects through architectural design does not *guarantee* a usable system—too many implementation decisions must also be made before the system reaches the end user—at least early architectural design decisions can be made that will not *preclude* delivering a usable system.

Our goal is for software design decisions to be made explicitly in light of all of the consequences of those decisions. In particular, we want to ensure that the usability consequences of design decisions are understood before it is too late in the life cycle of the system to repair mistaken decisions. We believe the patterns linking usability benefits to SA is the first step towards that goal.

Acknowledgements

This work resulted from a period when Bonnie John was on leave from Carnegie Mellon’s HCI Institute and working at the SEI. We would like to thank the HCI Institute for granting that leave, and the SEI, in particular Steve Cross and Linda Northrop, for supporting it.

We also wish to acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298 and the United States Department of Defense through its sponsorship of the Software Engineering Institute.

Appendix A. General usability scenarios

This section enumerates the usability scenarios that we have identified as being architecturally sensitive. A general usability scenario describes an interaction that some stakeholder (e.g., end user, developer, system administrator) has with the system under consideration from a usability point of view.

1. *Aggregating data.* A user may want to perform one or more actions on more than one object. Systems should allow users to select and act upon arbitrary combinations of data.
2. *Aggregating commands.* A user wishes to perform a multi-step procedure repetitively. Systems should provide a batch or macro capability to allow users to aggregate commands.
3. *Canceling commands.* A user invokes an operation, then no longer wants the operation to be performed. Systems should allow users to cancel operations.
4. *Using applications concurrently.* A user may want to work with arbitrary combinations of applications concurrently. These applications may interfere with each other. Systems should ensure that users can employ multiple applications concurrently without conflict.
5. *Checking for correctness.* A user may make an error that he or she does not notice. However, human error is frequently circumscribed by the structure of the system; the nature of the task at hand, and by predictable perceptual, cognitive, and motor limitations. Depending on context, error correction can be enforced directly (e.g., automatic text replacement, fields that only accept numbers) or suggested through system prompts.
6. *Maintaining device independence.* A user attempts to install a new device. The device may conflict with other devices already present in the system. Alternatively, the device may not function in certain specific applications. When device conflicts occur, the system should provide the information necessary to

either solve the problem or seek assistance. (Devices include printers, storage/media, and I/O apparatus.)

7. *Evaluating the system.* A system designer or administrator may be unable to test a system for robustness, correctness, or usability in a systematic fashion. Systems should include test points and data gathering capabilities to facilitate evaluation.
8. *Recovering from failure.* A system may suddenly stop functioning while a user is working. Users should be provided with the means to reduce the amount of work lost from system failures.
9. *Retrieving forgotten passwords.* A user may forget a password. Retrieving and/or changing it may be difficult or may cause lapses in security. Systems should provide alternative, secure strategies to grant users access.
10. *Providing good help.* A user needs help. The user may find, however, that a system's help procedures do not adapt adequately to the context. Help content may also lack the depth of information required to address the user's problem. Help procedures should be context dependent and sufficiently complete to assist users in solving problems.
11. *Reusing information.* A user may wish to move data from one part of a system to another. Users should be provided with automatic (e.g., data propagation) or manual (e.g., cut and paste) data transports between different parts of a system.
12. *Supporting international use.* A user may want to configure an application to communicate in his or her language or according to the norms of his or her culture. Systems should be easily configurable for deployment in multiple cultures.
13. *Leveraging human knowledge.* People use what they already know when approaching new situations. Such situations may include using new applications on a familiar platform, a new version of a familiar application, or a new product in an established product line. System designers should strive to develop upgrades that leverage users' knowledge of prior systems and allow them to move quickly and efficiently to the new system.
14. *Modifying interfaces.* Iterative design is the lifeblood of current software development practice, yet a system developer may find it prohibitively difficult to change the UI of an application to reflect new functions and/or new presentation desires. System designers should ensure that their UIs can be easily modified.
15. *Supporting multiple activities.* Users often need to work on multiple tasks more or less simultaneously (e.g., check mail and write a paper). A system or its applications should allow the user to switch quickly back and forth between these tasks.
16. *Navigating within a single view.* A user may want to navigate from data visible on-screen to data not currently displayed. If the system takes too long to display the new data or if the user must execute a cumbersome command sequence to arrive at her or his destination, the user's time will be wasted. System designers should strive to ensure that users can navigate within a view easily and attempt to keep wait times reasonably short.
17. *Observing system state.* A user may not be presented with the system state data necessary to operate the system (e.g., uninformative error messages, no file size given for folders). Alternatively, the system state may be presented in a way that violates human tolerances (e.g., is presented too quickly for people to read. See: Working at the user's pace). The system state may also be presented in an unclear fashion, thereby confusing the user. System designers should account for human needs and capabilities when deciding what aspects of system state to display and how to present them.
18. *Working at the user's pace.* A system might not accommodate a user's pace in performing an operation. This may make the user feel hurried or frustrated. Systems should account for human needs and capabilities when pacing the stages in an interaction. Systems should also allow users to adjust this pace as needed.
19. *Predicting task duration.* A user may want to work on another task while a system completes a long running operation. If systems do not provide expected task durations, users will be unable to make informed decisions about what to do while the computer "works." Thus, systems should present expected task durations.
20. *Supporting comprehensive searching.* A user wants to search some files or some aspects of those files for various types of content. Search capabilities may be inconsistent across different systems and media, thereby limiting the user's opportunity to work. Systems should allow users to search data in a comprehensive and consistent manner by relevant criteria.
21. *Supporting undo.* A user performs an operation, then no longer wants the effect of that operation. The system should allow the user to return to the state before that operation was performed. Furthermore, it is desirable that the user then be able to undo the prior operation (multi-level undo).
22. *Working in an unfamiliar context.* A user needs to work on a problem in a different context. Discrepancies between this new context and the one the user is accustomed to may interfere with the ability to work. Systems should provide a novice (verbose) interface to offer guidance to users operating in unfamiliar contexts.
23. *Verifying resources.* An application may fail to verify that necessary resources exist before beginning an operation. This failure may cause errors to occur un-

expectedly during execution. Applications should verify that all necessary resources are available before beginning an operation.

24. *Operating consistently across views.* A user may become confused by functional deviations between different views of the same data. Commands that had been available in one view may become unavailable in another or may require different access methods. Systems should make commands available based on the type and content of a user's data, rather than the current view of that data, as long as those operations make sense in the current view.
25. *Making views accessible.* Users often want to see data from other viewpoints. If certain views become unavailable in certain modes of operation, or if switching between views is cumbersome, the user's ability to gain insight through multiple perspectives will be constrained.
26. *Supporting visualization.* A user wishes to see data from a different viewpoint. Systems should provide a reasonable set of task-related views to enhance users' ability to gain additional insight while solving problems.
27. *Supporting personalization.* A user wants to work in a particular configuration of features that the system provides. The user may want this configuration to persist over multiple uses of the system (as opposed to having to set it up each time). Systems should enable a user to specify their preferences for features and provide the possibility for these preferences to endure.

References

- Alexander, C., Ishikawa, S., Silverstein, M., 1977. *A Pattern Language*. Oxford University Press, New York.
- Bass, L., Clements, P., Kazman, R., 1998. *Software Architecture in Practice*. Addison-Wesley, Reading, MA.
- Bass, L.J., John, B.E., 2000. Achieving Usability Through Software Architectural Styles. Extended Abstracts of CHI, 2000 The Hague, The Netherlands, 1–6 April 2000 ACM, New York. pp. 502–509.
- Bass, L., John, B.E., Kates, J., 2001. Achieving Usability Through Software Architecture, CMU/SEI-TR-2001-005 Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. <http://www.sei.cmu.edu/publications/documents/01.reports/01tr005.html>.
- Bass, L., Klein, M., Bachmann, F., 2001b. Quality Attribute Design Primitives and the Attribute Driven Design Method. In: *Proceedings of the Product Family Engineering*, vol. 4. Springer-Verlag, Berlin.
- Beyer, H., Holtzblatt, K., 1998. *Contextual Design*. Morgan Kaufmann Publishers, Inc., San Francisco, CA.
- Borchers, J., 2001. *A Pattern Approach to Interaction Design*. John Wiley & Sons, Chichester, UK.
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. *Pattern-Oriented Software Architecture—A System of Patterns*. John Wiley & Sons, Chichester, UK.
- Clements, P., Kazman, R., Klein, M., 2001. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, Reading, MA.
- Foley, J., Kim, W.C., Kovacevic, S., Murry, K., 1991. Sullivan, Tyler (Eds.), *UIDE—An Intelligent User Interface Design Environment in Intelligent User Interfaces*. Addison-Wesley, Reading, MA.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. *Design Patterns, Elements of Object-Oriented Software*. Addison-Wesley, Reading, MA.
- Gram, C., Cockton, G., 1996. *Design Principles for Interactive Systems*. Chapman and Hall, London, England.
- Newman, W., Lamming, M., 1995. *Interactive System Design*. Addison-Wesley Publishing, Wokingham, England.
- Nielsen, J., 1993. *Usability Engineering*. Academic Press Inc., Boston.
- Nielsen, J., Mack, R., 1994. *Usability Inspection Methods*. John Wiley & Sons, New York.
- Pfaff, G.P., 1983. (Ed.) *Proceedings of the Workshop on User Interface Management Systems*, 1–3 November 1983, Seeheim, Germany.
- Shneiderman, B., 1998. *Designing the User Interface*, third ed. Addison-Wesley, Reading, MA.
- UIMS Tool Developers Workshop, 1992. *A Metamodel for the Runtime Architecture of an Interactive Systems*, SIGCHI Bulletin January 1992.

Len Bass is an expert in software architecture and architecture design methods. Author of books on software architecture, documenting software architecture and developing software for the user interface, Len consults on large-scale software projects in his role as Senior MTS on the Architecture Trade-off Analysis Initiative at the Software Engineering Institute. His research area is the achievement of various software quality attributes through software architecture and he is the developer of software architecture analysis and design methods.

Bonnie E. John is an engineer (B.Engr., The Cooper Union, 1977; M. Engr. Stanford, 1978) and cognitive psychologist (M.S. Carnegie Mellon, 1984; Ph.D. Carnegie Mellon, 1988) who has worked both in industry (Bell Laboratories, 1977-1983) and academe (Carnegie Mellon University, 1988-present). She is an Associate Professor in the Human-Computer Interaction Institute and the Director of the Masters Program in HCI. Her research includes human performance modeling, usability evaluation methods, and the relationship between usability and software architecture. She consults for many industrial and government organizations.