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

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

[*] 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.

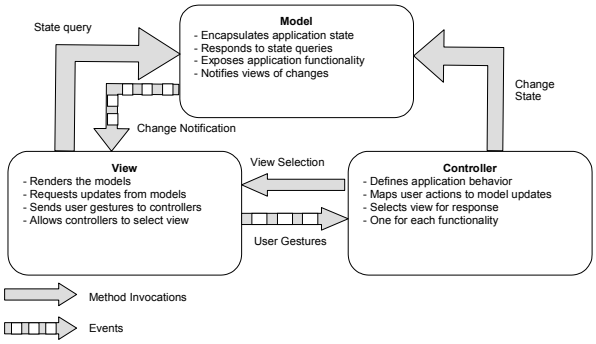

Fig. 1. J2EE-MVC structure diagram (adapted from [12]).

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.

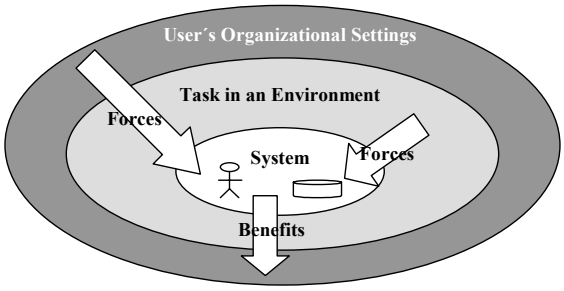

Fig. 2. Forces influencing the solution and benefits of the solution.

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.

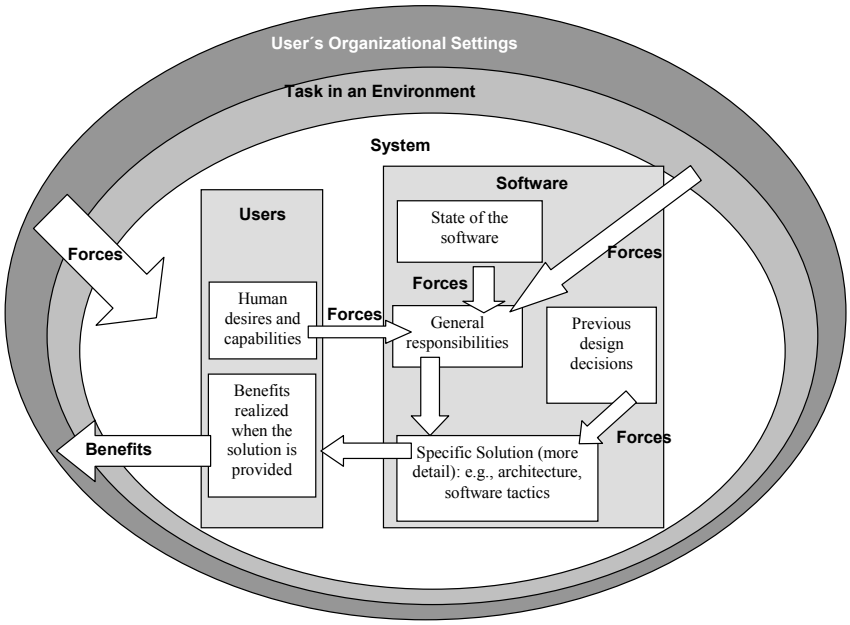

Fig. 3. Forces impacting the software architecture.

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.

Table 1. Usability-supporting architectural pattern template.

<table>
<tr><td colspan="4">Name: The name of the pattern</td></tr>
<tr><td colspan="4">Usability Context</td></tr>
<tr><td colspan="4">Situation: A brief description of the situation from the user's perspective that makes this pattern useful</td></tr>
<tr><td colspan="4">Conditions on the Situation: Any conditions on the situation constraining when the pattern is useful.</td></tr>
<tr><td colspan="4">Potential Usability Benefits: A brief description of the benefits to the user if the solution is implemented. We use the usability benefit hierarchy from [3,5] to express these benefits.</td></tr>
<tr><td colspan="3">Problem</td><td>General solution</td></tr>
<tr><td>Forces exerted by the environment and the task. Each row contains a different force</td><td>Forces exerted by human desires and capabilities. Each row contains a different force</td><td>Forces exerted by the state of the software. Each row contains a different force.</td><td>Responsibilities of the general solution that resolve the forces in the row.</td></tr>
<tr><td colspan="4">Specific Solution</td></tr>
<tr><td>Responsibilities of general solution (repeated from the General Solution column)</td><td>Forces that come from prior design decisions</td><td>Allocation of responsibilities to specific components.</td><td>Rationale justifying how this assignment of responsibilities to specific modules satisfy the problem</td></tr>
<tr><td colspan="4">Component diagram of specific solution</td></tr>
<tr><td colspan="4">Sequence diagram of specific solution</td></tr>
<tr><td colspan="4">Deployment diagram of specific solution (if necessary)</td></tr>
</table>

4.1 Cancellation: An Example of a Usability-Supporting Architectural Pattern

Consider the example of canceling commands. Cancellation is an important usability feature, whose value is well known to UI specialists and users alike, which is often poorly supported even in modern applications. This example shows the extent to which a usability concern permeates the architecture. Space does not permit us to include a completed pattern for this example, so we will illustrate specific points with selected portions of the pattern.

Usability Context. Table 2 contains the **Name** and the **Usability Context** portions of the usability-supporting architectural pattern for canceling commands. The **Situation** briefly describes the pattern from the point of view of the user, similar to the situation in other pattern formats. However, the **Conditions** section provides additional information about when the pattern is useful in the usability context. For example, cancellation is only beneficial to users when the system has commands that run longer than a second. With faster commands, users do not get additional benefit from cancellation over simply undoing a command after it has completed. The loci of control may also appear in the **Condition** section. In our example, the cancellation may be initiated by the user or by the software itself in response to changes in the

environment. The last section in the usability context is the **Potential Usability Benefits** to the user if the solution is implemented in the software. Quantifying these benefits will depend on the particular users, tasks, and organizational setting and is beyond the scope of this paper. However, the list of potential benefits and their rationale is a starting point for a cost/benefit analysis of providing the solutions in the pattern. The benefits are cast in terms of the benefit hierarchy given in [3,5] ranging from efficiency, to supporting non-routine behavior (i.e., problem-solving, creativity, or learning), to user confidence and comfort. The ability to cancel commands has the potential to benefit each of these categories.

The **Problem** and **General Solution**

Table 2. Usability context of the Cancelling Commands pattern.

Name: Cancelling Commands
Usability Context
Situation: The user issues a command then changes his or her mind, wanting to stop the operation and return the software to its pre-operation state. It doesn't matter why the user wants to stop; he or she could have made a mistake, the system could be unresponsive, or the environment could have changed.
Conditions of the Situation: A user is working in a system where the software has long-running commands, i.e., more than one second. The cancellation command can be explicitly issued by the user, or through some sensing of the environment (e.g., a child's hand in a power car window).
Potential Usability Benefits: *A. Increases individual user effectiveness* *A.1 Expedites routine performance* *A.1.2 Reduces the impact of routine user errors (slips)* by allowing users to revoke accidental commands and return to their task faster than waiting for the erroneous command to complete. *A.2 Improves non-routine performance* *A.2.1 Supports problem-solving* by allowing users to apply commands and explore without fear, because they can always abort their actions. *A.3 Reduces the impact of user errors caused by lack of knowledge (mistakes)* *A.3.2 Accommodates mistakes* by allowing users to abort commands they invoke through lack of knowledge and return to their task faster than waiting for the erroneous command to complete. *B. Reduces the impact of system errors* *B.2 Tolerates system* errors by allowing users to abort commands that aren't working properly (for example, a user cancels a download because the network is jammed). *C. Increases user confidence and comfort* by allowing users to perform without fear because they can always abort their actions.

Sections of the pattern are the heart of this paper's contribution to the research in usability and software architecture. Previous research jumped from a general scenario, like that in our **Situation** section, directly to a short list of general responsibilities and an architectural solution [2,3,5] or to detailed design solution [6] using the expertise

of the authors. Considering the forces is a step forward in codifying the human-computer interaction and software engineering expertise that was tacit in the previous work. Making tacit knowledge explicit provides a rationale for design recommendation, increases the understanding of the software engineers who use these patterns to inform their design, and provides a basis for deciding to include or exclude any specific aspect of the solution.

The **Problem** is defined by the system of forces stemming from the task and environment, recurring human desires and relevant capabilities, and the state of the software itself. These forces are arranged in columns and rows, a portion of which is shown in Table 3 for Cancelling Commands. Each row of conflicting or converging forces is resolved by a responsibility of the software, presented in the rightmost column of Table 3. These responsibilities constitute a **General Solution** to the problem.

The first row in the **Problem** and **General Solution** records the major forces that motivate the general usability situation. In our example, the facts that networks and other environmental systems beyond the software are sometimes unresponsive, that humans make mistakes or change their minds but do not want to wait to get back to their tasks, and that the software itself is sometimes unresponsive dictate that the software provide a means to cancel a command. The subsequent rows list other forces that come into play to dictate more specific responsibilities of the software. Some forces are qualitative and some are quantitative. For example, the middle of Table 3 shows a quantified human capability force that produces a performance responsibility; the software must acknowledge the reception of a cancel command within 150 ms and in a manner that will be perceived by the user [2]. These forces encapsulate decades of human performance research and provide specific performance and UI design guidance in a form that is usable and understandable by software designers.

In some rows, the forces converge and the responsibility fulfills the needs of the different sources of force. For example, in the second row of Table 3, both the environment and the human are unpredictable in their need for the cancellation function. The responsibilities that derives from these needs, that the system always be listening for the cancellation request and that is always be collecting the necessary data to perform a cancellation, solve both these compatible forces. Sometimes the forces conflict, as in part of the last row of Table 3, where the user wants the command to stop but the software is unresponsive. The responsibility must then resolve these opposing forces, in this case, going outside the software being designed to the system in which it runs.

Process of Creating the Problem and General Solution. Our process of creating the entries in the **Problem** and **General Solution** columns begins by examining prior research in usability and software architecture.

Table 3. Portion of the Problem and General Solution for Cancelling Commands.

Problem			General solution
Forces exerted by the environment & task.	**Forces exerted by human desires and capabilities.**	**Forces exerted by the state of the software.**	**General responsibilities of the software.**
Networks are sometimes unresponsive. Sometimes changes in the environment require the system to terminate	Users slip or make mistakes, or explore commands and then change their minds, but do not want to wait for the command to complete.	Software is sometimes unresponsive	Must provide a means to cancel a command
No one can predict when the environment will change	No one can predict when the users will want to cancel commands		Must always listen for the cancel command or environmental changes. Must be always listening and gathering the actions related to the command being invoked.
	User needs to know that the command was received within 150 msec, or they will try again. The user can be assumed to be looking at the cancel button, if this is how they canceled the command People can see changes in color and intensity in their peripheral vision as well as in their fovea.		Must acknowledge the command within 150 msec. Acknowledgement must be appropriate to the manner in which the command was issued. For example, if the user pressed a cancel button, changing the color of the button will be seen. If the user used a keyboard shortcut, flashing the menu that contains that command could be detected in peripheral vision.

Table 3. Portion of the Problem and General Solution for Cancelling Commands (continued).

Problem				General solution
Forces exerted by the environment & task.	**Forces exerted by human desires and capabilities.**		**Forces exerted by the state of the software.**	**General responsibilities of the software.**
	User wants the command to stop	EITHER	The command itself is responsive	The command should cancel itself regardless of the state of the environment
		OR	The command itself is not responsive or has not yet been invoked	An active portion of the system must ask the infrastructure to cancel the command, or The infrastructure itself must provide a means to kill the application (e.g., task manager on Windows, force quit on MacOS) (These requirements are independent of the state of the environment.)
Collaborating processes may prevent the command from canceling promptly			The command has invoked collaborating processes	The collaborating processes must be informed of the cancellation of the invoking command (these processes have their own responsibilities that they must perform in response to being informed).

From the previously documented scenarios we can read, or infer, forces from the task and environment or human desires and capabilities, and sometimes from the state of the software itself. From previously enumerated responsibilities, we uncover tacit assumptions about the forces they are resolving. From prior solutions, additional general responsibilities can sometimes be retrieved. We list all these forces in the appropriate columns and the responsibilities that resolve them.

This preliminary table then becomes the framework for further discussion around what we call *considerations*. Considerations are recurring forces, or variations in

forces, that cut across multiple scenarios. The considerations we have found to be useful involve issues of feedback to the user, time, initiative, and scope.

With any interactive system, there is always a consideration of feedback to the user. The user wants to be informed of the state of the software to make best use of their time, to know what to do next, perform sanity checks, trouble-shoot and the like. There are several types of feedback in almost every pattern: acknowledgement of the user's action, feedback on the progress of software actions, and feedback on the results of software actions. The need for each of these types of feedback is forces in the human needs and capability column. In Table 3, this consideration shows up in the third row.

The time consideration involves forward-looking, current, and backward-looking issues of time. One forward-looking consideration is the issue of persistence. Does the pattern involve any objects that must persist over time? If so, there are often issues of storing those objects, naming them, finding them later, editing them, etc. (This consideration can also be thought of as a need for authoring facilities). A current time issue is whether the pattern involves a process that will be operating concurrently with human actions. If so, how will the human's actions be synchronized at an effective time for both the software and the human? An example of a backward-looking time consideration occurs in the cancelling command pattern (not included in the portion of the pattern in Table 3). What state should the software roll back to? In most applications the answer is clearly "the state before the last command was issued." However, in systems of collaborating applications or with consumable resources, the answer becomes less clear. An extreme example of this consideration for a system-level undo facility can be found in the examination of system administrators by Brown and Patterson [7].

The initiative consideration involves which entity can control the interaction with the software being designed. In the cancelling commands pattern, initiative comes from several places. One normally thinks of a cancel command being deliberately instigated by the user. However, it is also possible that the environment can change, initiating the equivalent of a cancel command to the software. For example, the software that controls an automobile window lifter should stop the window rising if the driver presses a button (user's initiative), or if a child's hand is about to be trapped (system's initiative).

The scope consideration asks whether a problem is confined to the software being designed or concerns other aspects of the larger system. In the cancelling commands example, a larger scope is evident in the last two rows in Table 3 when considering responsibilities when the software is unresponsive and when there are collaborating processes.

Thus, the combination of mining prior research in usability and software architecture and asking the questions associated with considerations, allow the definition of the forces and responsibilities that resolve them. The general responsibilities constitute a general solution to the problem created by the forces. Some pattern advocates would eschew our process of defining responsibilities because the solution is generated, not recognized as an accepted good design used repeatedly in practice. We believe that these general responsibilities have value nonetheless because (1) they serve as requirements for any specific solution, and (2)

many of the usability problems we have examined are not consistently served in practice as yet, so no widely accepted solution is available.

Specific Solution. The specific solution is derived from the general responsibilities and the forces that come from prior design decisions. Usability-supporting architectural patterns differ from other architecture patterns in that they are neither overarching nor localized. Patterns such as client-server, layers, pipe and filter, and blackboard [8] tend to dominate the architecture of the systems in which they are used. It may be that they only dominate a portion of the system but in this case, they are usually encapsulated within a defined context and dominate that context. Other patterns such as publish-subscriber, forward-receiver, and proxy [8] are local in how they relate to the remainder of the architecture. They may impose conditions on components with which they interact but these conditions do not seriously impact the actions of the components.

Usability-supporting architectural patterns are not going to be overarching. One does not design a system, for example, around the support for cancelling commands. The support for this usability feature must be fit into whatever overarching system designs decisions are made to facilitate the core functionality and other quality attributes of the system. Usability-supporting architectural patterns are also not local, by definition. They involve multiple portions of the architecture almost regardless of what the initial design decisions have been made. Cancel, for example, ranges from a requirement to listen for user input (at all times), to freeing resources, to knowing about and informing collaborators of the cancellation request. All these responsibilities involve different portions of the architecture.

When presenting a specific solution, then, there are two choices – neither completely satisfactory.

1. Present the solution independent of prior design decisions. That is, convert the general responsibilities into a set of components and assign the responsibilities to them, without regard for any setting. A specific solution in this form does not provide good guidance for architects who will come to the usability supporting architectural patterns after having made a number of overarching design decisions. For example, if the J2EE-MVC pattern is used as the overarching pattern, then a listener for the cancel command is decoupled from the presentation of feedback to indicate acknowledgement of the command. If the PAC pattern is used, then a listener would be part of the presentation and would also be responsible for feedback.
2. Present the solution in the context of assumed prior design decisions. That is, assume an overarching pattern such as J2EE-MVC or PAC and ensure that the specific solution conforms to the constraints introduced by this decision. This increases the utility of the specific solution for those who are implementing within the J2EE-MVC context but decreases the utility for those implementing within some other context.

We have tried both solutions when we have presented earlier versions of this material, without finding a completely satisfactory solution. However, common practice in interactive system development currently uses some form of separation of the interface from the functionality. Therefore demonstrating the interplay of general responsibilities with a separation-based overarching architecture is a necessity to

make contact to current practice. Given the popularity of J2EE-MVC, we present our specific solution in that context.

For our cancel example, the forces caused by a prior design decision to use J2EE-MVC govern the assignment of function to the model objects, the view objects, or to the control objects (Figure 1). Any new responsibilities added by the usability problem must adhere to the typical assignments in J2EE-MVC. Thus, responsibilities that interact with the user must reside in the view, responsibilities that map user gestures to model updates or define application behavior or select views must reside in controller objects, and responsibilities that store state or respond to state queries must reside in models.

Table 4. Row of specific solution that concerns the general responsibility of always listening for the cancel command or environmental changes

Specific Solution			
Responsibilities of general solution. i.e., requirements	**Forces exerted by prior design decisions**	**Allocation of responsibilities to specific components**	**Rationale**
Must always listen for the cancel command or environmental changes.	In J2EE-MVC, user gestures are recognized by a controller J2EE-MVC is neutral about how to deal with environmental sensors	Listener component is a controller. It must • run on an independent thread from any model. • receive user gestures that are intended to invoke cancel. • receive environmental change notification that require a cancel.	Since the command being cancelled may be blocked and preempting the Listener, the Listener is assigned to a thread distinct from the one used by the command. Since J2EE-MVC is neutral with respect to environmental sensors, we chose to listen for the environmental sensors in the same controller that listens for user gestures that request cancellation (the Listener)

Table 4 shows a small portion of the **Specific Solution** for cancelling commands in J2EE-MVC, resolving the general responsibilities with the prior design decisions. For easy reading, the general responsibilities, i.e., requirements of the specific solution are repeated in the first column of the Specific Solution. In Table 4, we've chosen to illustrate the responsibility of always listening for the cancel command or environmental changes that signal the need for cancellation. This general responsibility was the first responsibility in the second row of Table 3. The next column contains those forces exerted by the prior design decisions that apply to the general responsibility in the same row. The fact that J2EE-MVC controllers recognize user gestures is one such force. That J2EE-MVC does not mention environmental sensors is listed as a force, but its inclusion simply records that J2EE-MVC does not exert a force on this point. The third column resolves these forces by further specifying the general responsibilities and allocating them to specific components in the overarching architecture. In this case, a new controller entitled the Listener is

assigned the specific responsibilities that fulfil the general responsibility. The last column provides additional rational for this allocation, for example, that since J2EE-MVC does not specify a component for environmental sensors, we chose to use the same controller as that listening for user requests to cancel.

After allocating all general responsibilities, all the new components and their responsibilities, and all new responsibilities assigned to old components of the overarching architecture can be collected into a specification for implementation. For example, when the remainder of the complete **Specific Solution** table (not shown) is considered, the Listener is responsible for

- always listening for a user's request to cancel,
- always listening for external sensor's request for cancellation (if any), and
- informing the Cancellation Manager (a model) of any cancellation request.

A component diagram of our specific solution is given in Figure 4. The View, Controller and Active Command (model) and Collaborating Processes (if any) are the components associated with J2EE-MVC under normal operations, without the facility to cancel commands. The results of the analysis in the complete **Specific Solution** table (not shown) added several new components. The Listener has already been described.

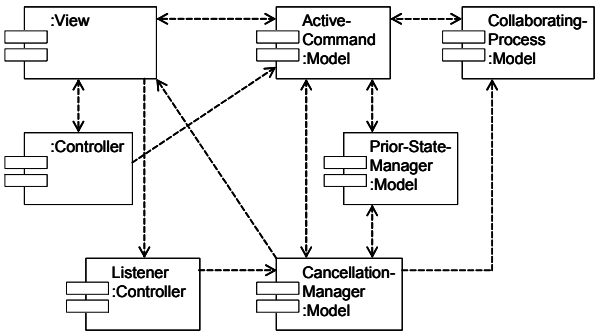

Fig. 4. Component diagram for the specific solution.

The Cancellation Manager and Prior State Manager are new models fulfilling the other general and specific responsibilities of cancelling commands. Because dynamic behaviour is important for the cancel command we also use two different sequence diagrams. The first (Figure 5) shows the sequence of normal operation with a user issuing a command to the software. This figure represents the case in which:

- The user requests a command
- The command can be cancelled

The command saved its state prior to execution using the Prior State Manager. The sequence diagram in Figure 6 represents the case in which:

- The user requests cancellation of an active command
- The current command is not blocked
- The prior state was stored
- Time of cancellation will be between 1 and 10 seconds. Change cursor shape but progress bars are not needed.

- It is not critical for the task that the cancellation be complete before another user action is taken
- All resources are properly freed by the current command.
- Original state is correctly restored.

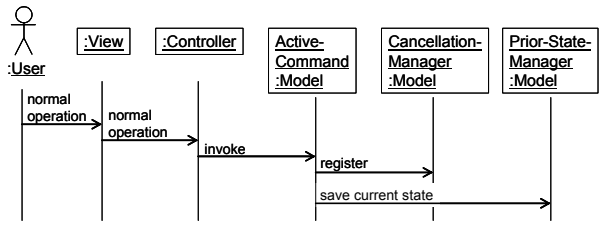

Fig. 5. Sequence diagram of normal operation, before cancel is requested.

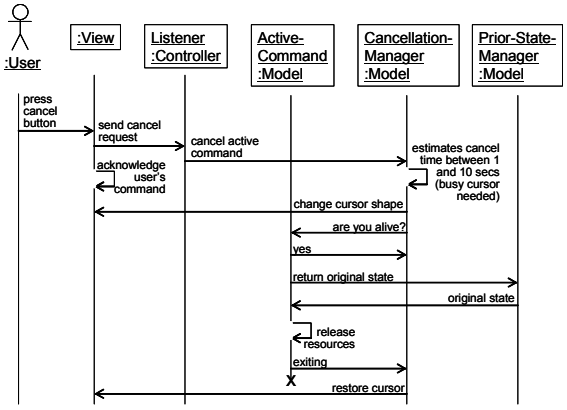

Fig. 6. Sequence diagram of canceling.

5. Experience with Usability-Supporting Architectural Patterns

We have presented the cancel example (although not this pattern of forces and their link to responsibilities) to professional audiences several times (e.g., [11]). After each presentation, audience members have told anecdotes about their experiences with implementing cancellation. One professional told us about the difficulty of adding cancel after initial implementation, confirming the utility of having a set of commonly encountered usability problems that can be considered early in design. Another professional told us that his company had included the ability to cancel from the beginning, but had not completely analyzed the necessary responsibilities and each cancellation request left 500MB of data on the disk. This anecdote confirms the utility

of having a detailed checklist of general responsibilities that must be fulfilled with sufficient traceability and rationale to convince developers of their importance.

We have also applied a collection of about two dozen usability-supporting architectural patterns ([3,5], again, prior to our inclusion of forces) in several real-world development projects. As part of their normal software architecture reviews, development groups have considered such patterns as *Supporting Undo*, *Reusing Information, Working at the User's Pace, Forgotten Passwords, Operating Consistently across Views, Working in an Unfamiliar Context, Supporting International Use,* and several different types of *Feedback to the User*. Discussions of these scenarios and their associated architectural recommendations allowed these development groups to accommodate usability concerns early in the design process.

6. Conclusions

Our major conclusion is that software architects must pay attention to usability while creating their design. It is not sufficient to merely use a separation based pattern such as MVC and expect to deliver a usable system.

Furthermore, we have shown that usability problem can be considered in light of several sources of forces acting in the larger system. These forces lead to general responsibilities, i.e., requirements, for any solution to the problem. Because the solutions to these usability situations do not produce overarching patterns and yet are also not localized, additional forces are exerted by design decisions made prior to the consideration of the usability situation. Finally, we have proposed a template that captures the different forces and their sources and provides a two level solution (general and specific), as well as substantial traceability and rationale.

We visualize a collection of usability-supporting architectural patterns formatted as we have described. These could be embodied in a *Handbook of Usability for Software Architects* that could be used in whatever architecture design and review processes employed by a development team. For example, as part of an Architectural Tradeoff Analysis Method review [9], the **Usability Context** of each pattern could be examined by the stakeholders to determine its applicability to their project. The usability specialists and software architects could then work together to determine the risks associated with particular architecture decisions and whether the benefits of supporting the pattern in the context of that project exceed the costs. They could use the general responsibilities to verify that their adaptation of the specific solution satisfies all of the forces acting in their context. The raw material for the production of such a handbook is in place. About two dozen usability scenarios exist with explicit solutions, at different levels, documented by several research groups. Half a dozen of these have been augmented with forces and responsibilities using the template proposed here [4]. We believe that publication of such a handbook would make a significant contribution to improving the usability of fielded systems because the concept of forces resolved by responsibilities provides a traceability and rationale surpassing previous work.

References

1. Alexander, C., Ishikawa, S., and Silvernstein, M. *A Pattern Language*, Oxford University Press, New York, 1997.
2. Bass, L. and John, B. E. *Supporting the CANCEL Command Through Software Architecture*, CMU/SEI-2002-TN-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2002.
3. Bass, L. and John, B. E. "Linking Usability to Software Architecture Patterns through general scenarios", *Journal of System and Software*, 66, Elsevier, 2003, pp. 187-197.
4. Bass, L., John, B. E., Juristo, N., and Sanchez-Segura, M. Tutorial "Usability-Supporting Architectural Patterns" in *Proceedings of the 26th International Conference on Software Engineering*, IEEE Computer Society, May 23-28, 2004, Edinburgh, Scotland.
5. Bass, L., John, B. E. and Kates, J. Achieving Usability Through Software Architecture, CMU/SEI-TR-2001-005 Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, (2001). Available for download at http://www.sei.cmu.edu/publications/documents/01.reports/01tr005.html
6. Bosch, J. and Juristo, N. Tutorial "Designing Software Architectures for Usability" in *Proceedings of the 25th International Conference on Software Engineering*, IEEE Computer Society, May 3-10, 2003, Portland, Oregon, USA.
7. Brown, A. B. and Patterson, D. A., "Undo for Operators: Building an Undoable E-mail Store" *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, TX, June 2003.
8. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., *Pattern-Oriented Software Architecture: A System Of Patterns*, Volume 1. John Wiley & Sons Ltd., New York, 1996.
9. Clements, P., Kazman, R., and Klein, M. *Evaluating Software Architectures: Methods and Case Studies.* Addison-Wesley, Reading. MA, 2001.
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Massachusetts, 1995.
11. John, B. E. and Bass, L., Tutorial, "Avoiding "We can't change THAT!": Software Architecture and Usability" In Conference Companion of the ACM Conference on Computer-Human Interaction, 2002, 2003, 2004.
12. Sun Microsystems, Inc, "Java BluePrints, Model-View-Controller," September 2003, http://java.sun.com/blueprints/patterns/MVC-detailed.html. Copyright 2000-2003 Sun Microsystems. All rights reserved.

Discussion

[Michael Harrison] I'm not familiar with this work, so forgive the naive question. It sounds like you've got a generic notion of CANCEL and you're trying to situate that within a particular context and within a particular application. Is this correct?

> [Bonnie John] No, we're looking more at generic contingencies, conditions and forces. We're trying to say "if you look at your specific situation and these fit" then you have to take the architectural guidance into account.

[Tom Omerod] You raised the question of how you know when you're done producing one of these descriptions. For example, you've ended up with about twenty responsibilities for CANCEL alone. How do you know when you're done?

> [Bonnie John] We don't have a good answer for that question. In essence, we have to keep presenting the description to new audiences, and comparing it to new systems, and seeing if we get new insights. In the particular case of CANCEL, we've only added one responsibility in the last year so we think we may be close to done. However, the fact that there is no reliable way of telling whether you're done is quite disconcerting.

[Tom Ormerod] Maybe it would be better if you were exploring several issues in parallel, rather than just CANCEL.

> [Bonnie John] Yes, and we are. In fact we have documented six of these usability architectural issues, which is helping us to derive general patterns (as shown in the paper).

[Willem-Paul Brinkman] Does usability prescribe only one software architecture, or are only responsibilities mentioned? Because if there is only one right architectural solution, then you can simply start checking the architecture.

> [Bonnie John] No, absolutely not. This is why we particularly like having the forces and responsibilities in our descriptions --- they give insight into how to fit the solution into the rest of the system's architecture (which will necessarily vary based on many other concerns).

[Gerrit van der Veer] You are labelling parts of your solutions as patterns. This suggests that it is design knowledge that can be shared. Doesn't this imply that you need examples of each pattern, as well as counter-patterns, to provide the generic design knowledge? Is there an intention or effort to collect these (which is a huge effort)?

> [Bonnie John] Yes. We're working with Dick Gabriel at Sun, president of Hillside Group, to get better integrated with the patterns community. With the community's help we're hoping to make a collective effort to document both these kinds of patterns.

[Jurgen Ziegler] Developers may get overwhelmed with the large number of requirements, particularly since there are also many more requirements that are not usability-related. Wouldn't it help to show developers different examples of architectures that fulfil your requirements to different degrees?

> [Bonnie John] Yes, absolutely. For example, one thing we're doing is keeping track of products that don't do cancel correctly or completely, and how. We haven't documented all of these yet.

[Nick Graham] In designing an architecture you have two basic options --- either attempt to anticipate all cases, or make the architecture sufficiently resilient to change that it is possible to modify afterwards. In the first case you may end up with an architecture that's bloated by features that may never be used. In the second, you seem to be back with the original "you can't change that" problem. Where does your approach really fit in?

> [Bonnie John] We use risk assessment techniques to assess which requirements are really likely to come up. Since these requirements aren't

core to the system function (in some sense they're peripheral) we're hoping that with these checklists people can consider stuff like this early in the process. We're not trying to anticipate everything, but rather things that we know get left out. The kinds of things we're considering are general problems that recur frequently and that reach deep into the architecture.

[Michael Harrison] Have you looked at whether people are actually helped by the forces and responsibilities?

[Bonnie John] We've done one really in-depth project with this approach using a Mars Rover control board with NASA. They say that the architectural suggestions helped them, but now we're looking at the actual code and the user performance data that NASA collected to get a view beyond their subjective evaluation. (However, this was before we had the forces and responsibilities directly in our model.) We're also doing similar things with some of our tutorial participants. The data is sparse so far. We're conducting a controlled experiment to answer this question which we hope to report on at ICSE and/or CHI 2005.