# The Value of a Usability-Supporting Architectural Pattern in Software Architecture Design: A Controlled Experiment

Elspeth Golden
Carnegie Mellon University
Human-Computer Interaction Institute
Pittsburgh, PA 15213
egolden@cs.cmu.edu

Bonnie E. John
Carnegie Mellon University
Human-Computer Interaction Institute
Pittsburgh, PA 15213
bej@cs.cmu.edu

Len Bass
Carnegie Mellon University
Software Engineering Institute
Pittsburgh, PA 15213
ljb@sei.cmu.edu

## ABSTRACT

Design patterns have been claimed to facilitate modification and improve understanding in software design. A controlled experiment was performed to assess the usefulness of portions of a Usability-Supporting Architectural Pattern (USAP) in modifying the design of software architectures to support a specific usability concern. Software engineering and information technology graduate students received different subsets of a USAP supporting cancellation functionality. They then studied a software architecture design and made modifications to add the ability to cancel commands. Results showed that participants who received a usability scenario, a list of general responsibilities, and a sample solution thought of significantly more key issues than participants who saw only the scenario. Implications for software development are that usability concerns can be included at architecture design time, and that USAPs can significantly help software architects to consider responsibilities inherent from usability concerns.

## Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentations**]: User Interfaces – *evaluation/methodology, theory and methods.* D.2.11 [**Software Engineering**]: Software Architectures – *domain-specific architectures, patterns.*

## General Terms

Design, Experimentation.

## Keywords

Controlled experiment, usability, software architecture, design pattern, modification.

## 1. INTRODUCTION

Usability is an important quality attribute of interactive software systems, but it has gotten little attention in the architecture design phase of software development. Since the 1980s, usability has been treated as a subset of modifiability, that is, architects commonly separate the user interface from the functionality of the system and assume that usability issues that arise during user testing can be handled with localized modifications. As recently as 1999, usability has been labeled not "discernable" at architecture design time [12].

Unfortunately, simply separating the interface from the functionality does not support all usability concerns. In fact, many usability concerns reach deeply into a system's architecture design [4]. Therefore, when usability is not considered early in the design process and support for them is not designed into the architecture, usability problems found in user testing often require extensive and costly re-architecting of software systems. When this happens, projects often cannot afford the additional cost and ship products that are not as usable as they could be.

Our research on the relationship between usability and software architecture has lead to the development of Usability-Supporting Architecture Patterns (USAPs), each of which addresses a usability concern that is not addressed by separation alone [3]. Each USAP consists of

- an architecturally sensitive usability scenario,
- a list of general responsibilities derived from forces inherent in the task and environment, human capabilities and desires, and software state, which must be considered by any software implementation for which the usability scenario is relevant, and
- a sample solution implemented in a larger separation-based design pattern such as J2EE MVC.

An early version of USAPs were applied in the design of the architecture of the MERBoard, a wall-sized tool that supports shoulder-to-shoulder collaboration of the engineers and scientists on NASA's Mars Exploration Rover mission [1]. Tutorials on the application of USAPs to software architecture design have also been given at the ICSE and CHI conferences [6, 9, 10].

These patterns could prove of great benefit to software architects responsible for developing large-scale software systems, and to the enterprises of all kinds for which these software systems are developed. Software architects may find it beneficial to have specific sets of responsibilities through which to address usability issues at architectural design time, both to avoid errors of omission, and to facilitate analysis of tradeoffs. However, USAPs are quite detailed and complex, which might make them difficult for software architects to apply to their own design problems. Since software architects are already burdened with complex tasks and extensive methodologies, it is important to determine whether all parts of a USAP are useful or necessary before widely disseminating USAPs to these professionals.

In this paper, we describe a controlled experiment designed to assess the value of the different parts of a USAP in modifying a software architecture design. We investigate a single USAP that supports an important usability concern: canceling a long-running command, and asked software engineers to apply it to the redesign of an architecture that had not originally considered the ability to cancel. The experiment measured whether the architectural solutions produced as a result of using all three components of a USAP more fully supported the needs of a usable cancellation facility than those produced by using certain subsets of the USAP components.

## 2. DESCRIPTION OF THE EXPERIMENT

### 2.1 Participants

18 computer science graduate students at the Carnegie Mellon West Coast campus participated in the experiment. All 18 participants had completed work for a master's degree in software engineering and/or information technology, and were working toward an additional practice-based Professional Development Certificate. The 15 male and 3 female participants ranged in age from 23 to 30. Fifteen of the participants averaged 25.7 months of industrial programming experience, with a range from 6 to 48 months; the other 3 had no programming experience in industry. Fourteen of the participants averaged 15.3 months of software design in industry, with a range of 4 to 36 months; the other 4 participants had no industrial experience in software design. Participants reported currently spending between 5 and 50 hours per week programming, with an average of 22.9 hours per week, and from 0 to 30 hours per week on software design, with an average of 11.4 hours per week.

### 2.2 Experiment Design and Materials

We used a between-subjects design with participants randomly assigned to one of three experimental conditions. Participants in each condition received a different version of a "Training Document," and all participants received the same architecture redesign task. In creating the materials for this experiment, all instructional and task materials were evaluated by eight academic and industry software architecture experts for correctness and completeness with respect to software architecture.

The Training Document received by participants in the first condition contained only a usability scenario describing circumstances under which a user might need to cancel an active command [Fig. 1]. This scenario is similar to what would typically be contained in a report that a usability expert would submit to a development organization, recommending that cancellation capability be added to an application.

**Usability Scenario: Canceling a Command**

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.

**Figure 1. Usability Scenario in Training Document**

Participants in the second condition received a Training Document that contained the same usability scenario, plus a list of general responsibilities that should be considered in any software design implementation of a cancel command [Table1]. This list of responsibilities was derived from an analysis of forces generated by characteristics of the task and environment, the desires and capabilities of the user, and the state of the software itself [11]. Since this was a list of general responsibilities designed to be considered in any implementation of cancel functionality, the Training Document stipulated that not all responsibilities might apply to the solution of any specific problem.

Participants in the third condition received the scenario, the list of general responsibilities, and a sample solution for adding cancellation to a software architecture design, based on J2EE MVC. The sample solution contained a "before and after" Component Diagrams of the J2EE-MVC architecture without considering cancellation (Figure 2) and with considering cancellation (Figure 3). The numbers in the components in Figure 2 correspond to a numbered list of the responsibilities allocated to the J2EE MVC components. Figure 3 allocated the cancellation responsibilities (CRs) in the numbered list to the J2EE-MVC components and added new components as required to fulfill the cancellation responsibilities. Although the example solution is not the only arrangement of components and responsibilities that would support cancellation, it is a reasonable solution, as judged by our eight external expert architects.

Thus, the Training Document for the first condition consisted of a single paragraph, the general cancellation scenario. The Training Document for the second condition was three printed pages of prose: the scenario and the list of cancellation responsibilities. The Training Document for the third condition was eight pages of both prose and software architecture diagrams that related to the prose.

**Table 1. General Responsibilities of Cancel**

| | |
|---|---|
| CR1 | A button, menu item, keyboard shortcut and/or other means must be provided, by which the user may cancel the active command. |
| CR2 | The system must always listen for the cancel command or changes in the system environment. |
| CR3 | The system must always gather information (state, resource usage, actions, etc.) that allow for recovery of the state of the system prior to the execution of the current command. |
| CR4 | The system must acknowledge receipt of the cancellation command appropriately within 150 ms. Acknowledgement must be appropriate to the manner in which the command was issued  i. For example, if the user pressed a cancel button, changing the color of the button will be seen. ii. If the user used a keyboard shortcut, flashing the menu that contains that command might be appropriate. |
| CR5 | If the command itself is able to cancel itself directly at the time of cancellation, the command must respond by canceling itself (i.e., it must fulfill responsibilities CR9-CR19 below (e.g., an object-oriented system would have a cancel method in each object)). |
| CR6 | If the command itself is not able to cancel itself directly at the time of cancellation, an active portion of the system must ask the infrastructure to cancel the command, or must fulfill responsibility CR7 below. |
| CR7 | If the command itself is not able to cancel itself directly at the time of cancellation, the infrastructure itself must provide a means to request the cancellation of the application (e.g., task manager on Windows, force quit on MacOS), or must fulfill responsibility CR6 above. |
| CR8 | If either CR6 or CR7 are fulfilled, then the infrastructure must also have the ability to cancel the active command with whatever help is available from the active portion of the application (i.e., it must fulfill Responsibilities CR9-CR19 below). |
| CR9 | If the command has invoked any 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 this information, possibly treat it as a cancellation.). The information given to collaborating processes may include the request for cancellation, the progress of cancellation, and/or the completion of cancellation. |
| CR10 | If the system is capable of rolling back all changes to the state prior to execution of the command, the system state must be restored to its state prior to execution of the command. |
| CR11 | If the system is not capable of rolling back some of the changes made during the operation of the command prior to cancellation, the system must be restored to a state as close to the state prior to execution of the command as possible. |
| CR12 | If the system is not capable of rolling back some of the changes made during the operation of the command prior to cancellation, the system must inform the user of the difference, if any, between the prior state and the restored state. |
| CR13 | The system must free all resources that it can that were consumed to process the command. |
| CR14 | If some resources has been irrevocably consumed and cannot be restored, the system must inform the user of the partially-restored resources in a manner that they can see it. |
| CR15 | If the command takes longer than 1 second to cancel, control must be returned to the user, if appropriate to the task. |
| CR16 | If control cannot be returned to the user, the system must inform the user of this fact (and ideally, why control cannot be returned). |
| CR17 | The system must estimate the time it will take to cancel within 20%. |
| CR18 | The system must inform the user of this estimate.  i. If the estimate is between 1 and 10 seconds, changing the cursor shape is sufficient.  ii. If the estimate is more than 10 seconds, and time estimate is within 20%, then a progress indicator is better.   iii. If estimate is more than 10 seconds but cannot be estimated accurately, provide other form of feedback to the user. |
| CR19 | Once the cancellation has finished, the system must provide feedback to the user that cancellation is finished (e.g., if cursor was changed to busy indicator, change it back to normal; if progress bar was displayed was displayed, remove it; if dialog box was provided, close it). |

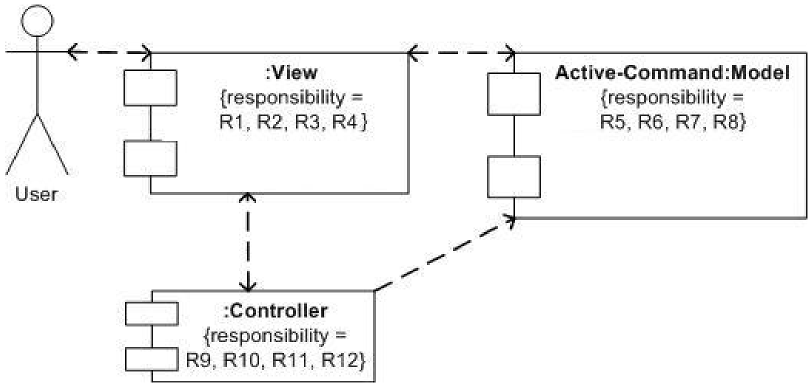

User
:View
{responsibility = R1, R2, R3, R4}
Active-Command:Model
{responsibility = R5, R6, R7, R8}
:Controller
{responsibility = R9, R10, R11, R12}

**Figure 2. J2EE-MVC before Cancel is added**

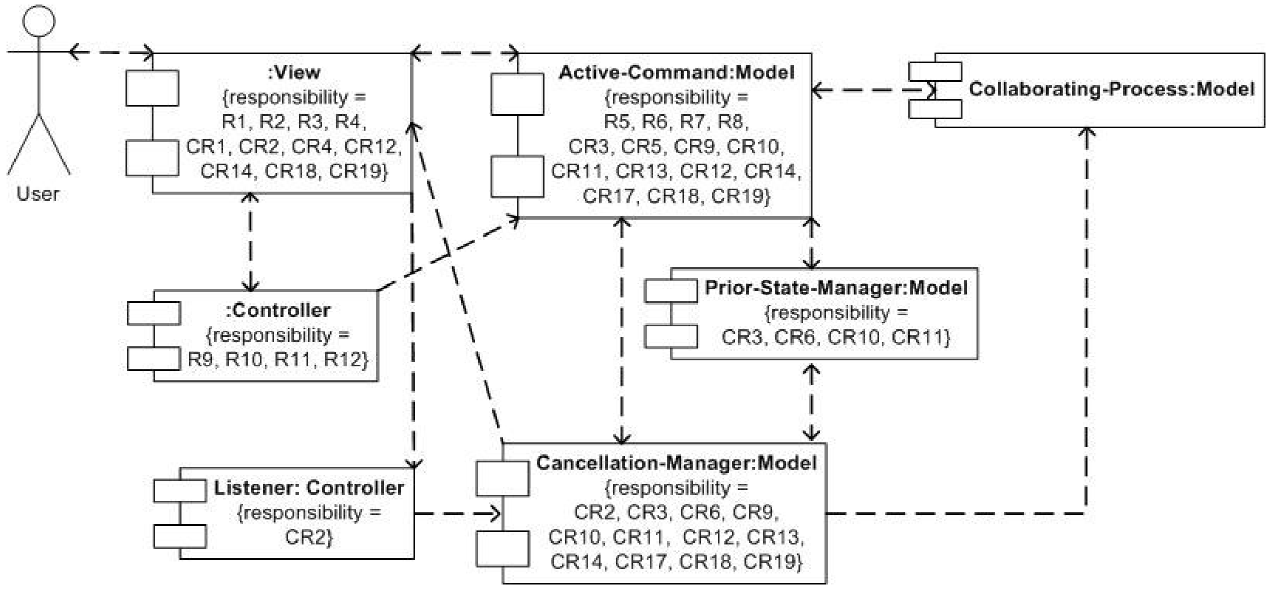

User
:View
{responsibility = R1, R2, R3, R4, CR1, CR2, CR4, CR12, CR14, CR18, CR19}
Active-Command:Model
{responsibility = R5, R6, R7, R8, CR3, CR5, CR9, CR10, CR11, CR13, CR12, CR14, CR17, CR18, CR19}
Collaborating-Process:Model
:Controller
{responsibility = R9, R10, R11, R12}
Prior-State-Manager:Model
{responsibility = CR3, CR6, CR10, CR11}
Listener: Controller
{responsibility = CR2}
Cancellation-Manager:Model
{responsibility = CR2, CR3, CR6, CR9, CR10, CR11, CR12, CR13, CR14, CR17, CR18, CR19}

**Figure 3. J2EE-MVC after Cancel is added**

The software architecture redesign task given to all participants was to redesign an existing architecture that was not designed to support cancellation, so that it did support cancellation. For this task, we chose the Plug-in Architecture for Mobile Devices (PAMD), an architecture designed by students in the Master of Software Engineering program at Carnegie Mellon University in 2002 for a plug-in controller for the Palm OS4. PAMD has also been used in a software architecture design and analysis course at the Software Engineering Institute at CMU [8]. This architecture design was chosen for several reasons. First, PAMD is simple enough that a participant already trained in software architecture can understand it in a relatively short period of time. Additionally, since the Palm OS4 is a single-threaded architecture, adding the ability to cancel a long-running command is a nontrivial task. PAMD was also sufficiently different from J2EE MVC that the participants who received the J2EE MVC-based sample solution had to extrapolate and generalize in order to create a specific solution for adding cancel to PAMD. That is, the sample solution required deep understanding of the concepts rather than simply giving the solution away.

The Task Instructions included seven elements:

(1) a general description of the PAMD architecture;
(2) an example scenario of how PAMD works;
(3) a numbered list of responsibilities of the PAMD components for normal operation;
(4) a numbered list of Component Interaction Steps detailing the run-time operation of PAMD while calling a plug-in;
(5) a Component Interaction Diagram, showing the components and connectors involved in the PAMD architecture, and assigning numbered responsibilities from element (3), and numbered steps from element (4), to each component [Fig. 4];
(6) a Sequence Diagram of PAMD run-time component interaction while calling a plug-in, utilizing the numbered

steps from element (4) and the components from element (5) [Fig. 5];

(7) a final page instructing the participant to add the ability to cancel a plug-in to the PAMD architecture design. The final page instructed participants to only modify the architecture to address cancellation, without considering or preserving other usability concerns or quality attributes. They were instructed to indicate their changes by modifying diagrams and written materials on the Answer Paper provided.

We designed an Answer Paper where the participants could easily and efficiently record the information relevant to their redesign. Because we didn't want the participants to waste time drawing boxes and moving them around, or to struggle with computer-based tools they might not be familiar with, the Answer Paper was paper-based and contained a Component Interaction Diagram, a Sequence Diagram, and a list of Component Interaction Steps, all with sufficient white space for the participants to insert their designs. The participants were also given several blank sheets of paper to use as they wished. The Component Interaction Diagram and the Component Interaction Steps were identical to those provided in the Task Instructions, except that the assignments of numbered PAMD responsibilities and run-time steps were removed from the Answer Paper. The Sequence Diagram in the Answer Paper showed unnumbered execution steps only up through calling a plug-in [Fig. 6], instead of continuing through the completion of normal plug-in termination as in the Task Instructions, because the user's request for cancellation would appear in the Sequence Diagram after the plug-in was called. The participants were instructed to use these diagrams as the bases for their designs, to add any components, responsibilities, or steps as needed to express their ideas for supporting cancellation. They were asked to make the diagrams correspond with each other, just as the PAMD diagrams corresponded with each other.

## 2.3 Procedure

Participants were randomly assigned to one of the three experimental conditions, and run in individual sessions. Participants were allowed unlimited time to complete the experiment, which resulted in sessions lasting between one and three hours. In an introduction to the experiment, the participants were told that they were participating in a study about fixing one kind of usability problem in a specific software architecture design. They were informed that they would be given a handout to read, describing a usability scenario relevant to system architecture, that we would then read through a description of a system architecture needing the information described in the handout, and finally, that they would be asked to understand and modify the sample software architecture to meet the requirements of the usability scenario.

Participants were then given the appropriate Training Document for their experimental condition, and asked to read and understand it. After reading their Training Document, participants were given the Task Instructions. To minimize variation in time and comprehension level during this portion of the experiment, the experimenter read the Task Instructions aloud, while the participant read along silently, and interrupted with any questions. During the reading of the final page of the Task Instructions, participants were given the Answer Packet. Each participant was allowed unlimited time in which to complete the redesign task. After completing their solution, the participant was asked to explain the details of the solution to the experimenter to disambiguate any hand-writing or diagrammatic difficulties.

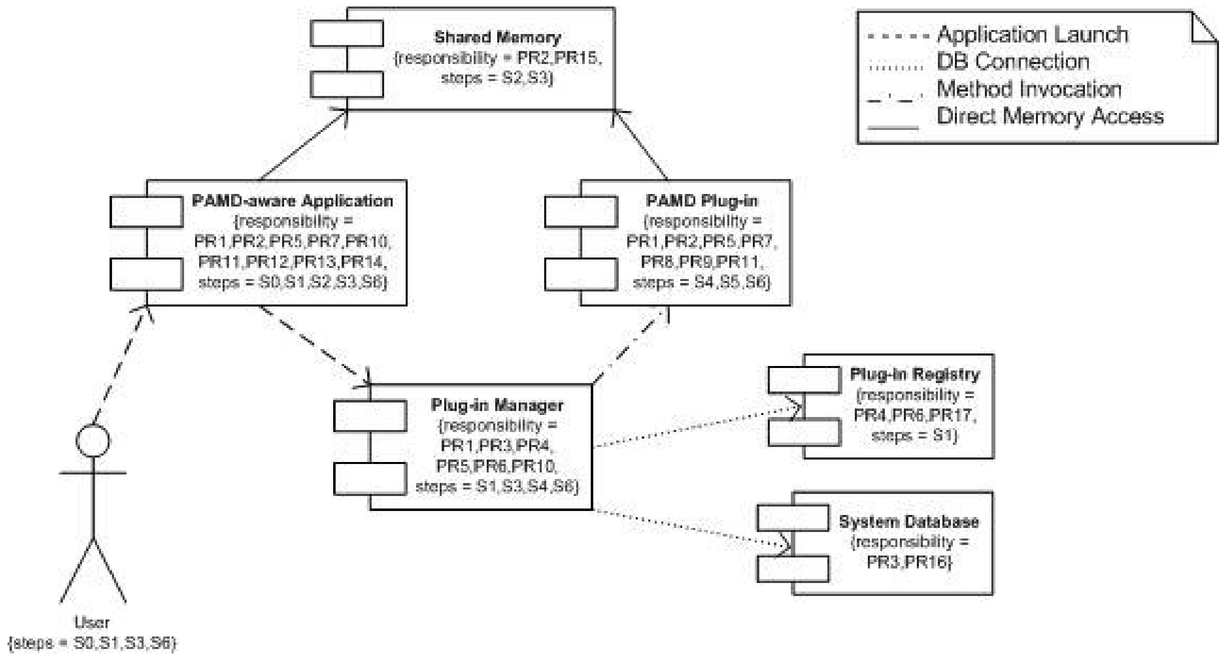

Shared Memory
{responsibility = PR2,PR15, steps = S2,S3}
Application Launch
DB Connection
Method Invocation
Direct Memory Access
PAMD-aware Application
{responsibility = PR1,PR2,PR5,PR7,PR10, PR11,PR12,PR13,PR14, steps = S0,S1,S2,S3,S6}
PAMD Plug-in
{responsibility = PR1,PR2,PR5,PR7, PR8,PR9,PR11, steps = S4,S5,S6}
Plug-in Registry
{responsibility = PR4,PR6,PR17, steps = S1}
Plug-in Manager
{responsibility = PR1,PR3,PR4, PR5,PR6,PR10, steps = S1,S3,S4,S6}
System Database
{responsibility = PR3,PR16}
User
{steps = S0,S1,S3,S6}

**Figure 4. Component Interaction Diagram in Task Instructions**

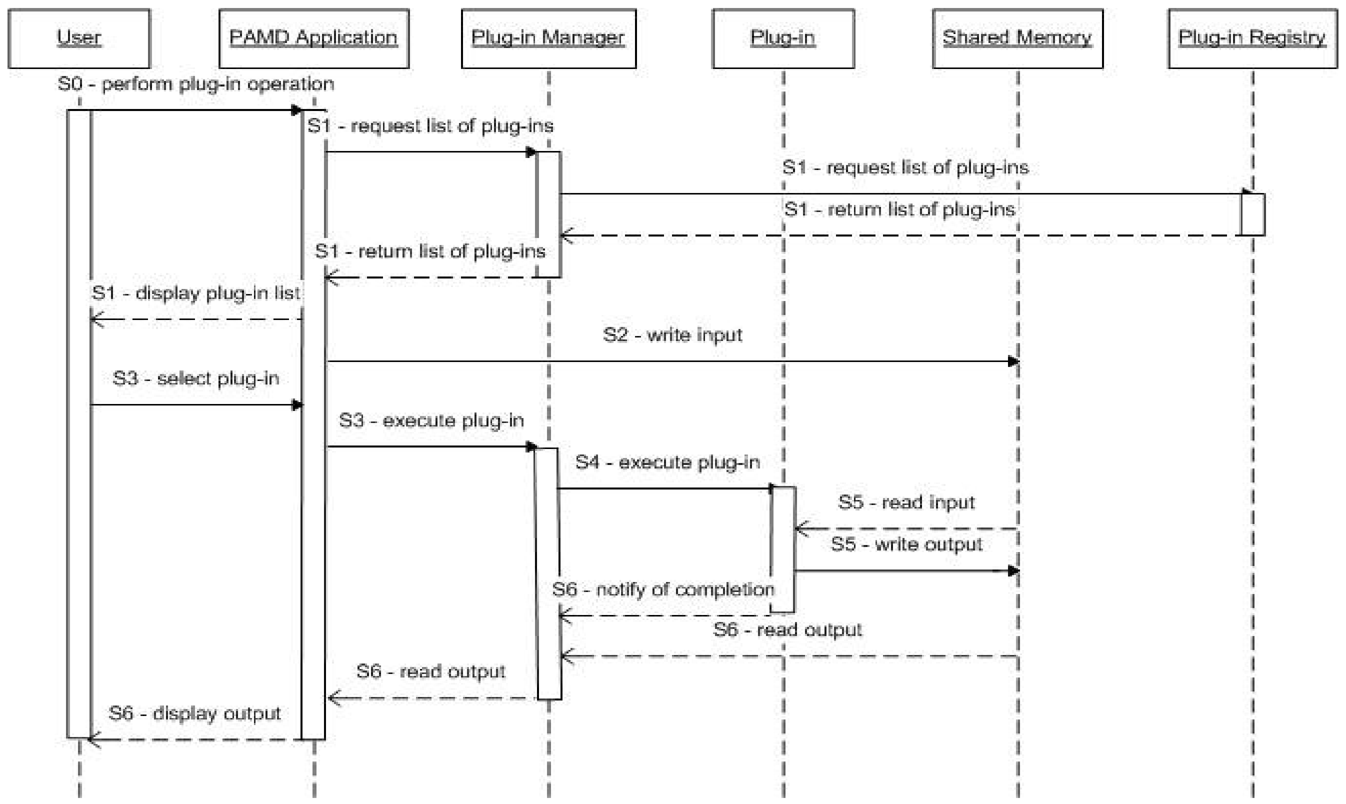
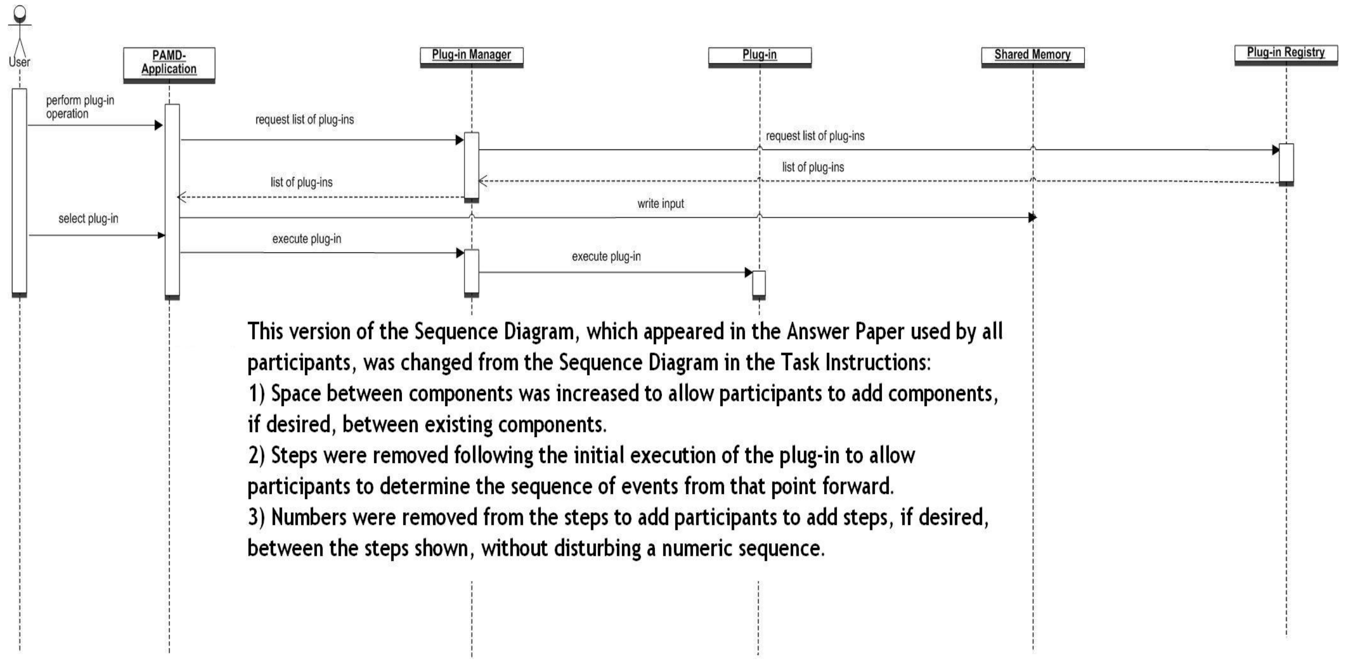


**Figure 5. Sequence Diagram in Task Instructions**



This version of the Sequence Diagram, which appeared in the Answer Paper used by all participants, was changed from the Sequence Diagram in the Task Instructions:
1) Space between components was increased to allow participants to add components, if desired, between existing components.
2) Steps were removed following the initial execution of the plug-in to allow participants to determine the sequence of events from that point forward.
3) Numbers were removed from the steps to add participants to add steps, if desired, between the steps shown, without disturbing a numeric sequence.

**Figure 6. Sequence Diagram in Participant Answer Paper**

## 3. RESULTS

Two dependent variables will be presented here: time on task, and cancellation responsibilities considered in the participant solution. Additional analysis related to the quality of the participant solutions is ongoing and will be discussed in Section 5.

The time to complete the redesign task varied from 39 to 138 minutes, with an average of 86 minutes [Fig. 7]. A one-way ANOVA revealed no statistical main effect of condition on this measure.

Similarly to the method used in Prechelt et al. for counting the degree of requirements fulfillment within subtasks of a programming maintenance task [14], we developed a scoring system that counted the union of all cancellation responsibilities found to have been considered in elements of the participant solution. That is, we counted a responsibility as having been considered if it appeared in the Component Interaction Diagram, or the Sequence Diagram, or the Component Interaction Steps, or in any list of Additional Responsibilities added by the participants. Each specific cancellation responsibility that appeared in the participant solution was counted only once, irrespective of the number of solution elements in which it appeared. A lower number of cancellation responsibilities appearing in a participant solution indicates a narrower consideration of cancellation responsibilities; a broader consideration of cancellation responsibilities produces a higher number of such responsibilities in the participant solution.

Participants who were given only the USAP scenario considered between 2 and 4 cancellation responsibilities in their solutions out of a possible 19, with an average of 3.17. Those who received the USAP scenario and list of 19 general responsibilities averaged 7.7 cancellation responsibilities in their solution, with a range from 4 to 15. Participants who received the USAP scenario, list of 19 general responsibilities, and a sample solution, considered from 5 to 15 cancellation responsibilities in their solutions, with an average of 9.5. Responsibilities considered for all groups are shown in Table 2 and Figure 8.

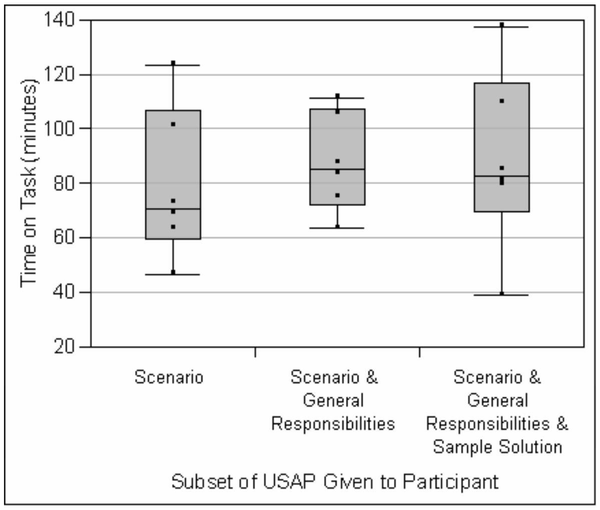


**Figure 7. Time on Task by USAP Subset**

**Table 2. Cancellation Responsibilities Considered**

| Cancellation Responsibilities Considered | Scenario | Scenario & General Responsibilities | Scenario & General Responsibilities & Sample Solution |
|---|---|---|---|
| Mean | 3.17 | 7.67 | 9.50 |
| Std Dev | 0.75 | 4.41 | 4.04 |
| Lowest | 2 | 4 | 5 |
| Highest | 4. | 15 | 15 |

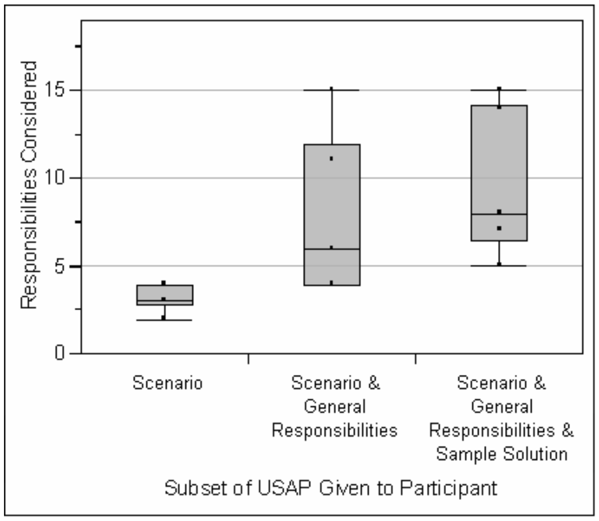


**Figure 8. Responsibilities Considered by USAP Subset**

Analysis of variance showed a significant main effect of the USAP subset given to participants on the number of cancellation responsibilities considered ($F(2,15) = 5.26$, $p < .05$). Pairwise comparisons made using Tukey's HSD indicated significant mean difference between giving the scenario alone (Scenario) and giving the full USAP (Scenario & General Responsibilities & Sample Solution), while mean difference between other pairs of conditions was not significant.

Analysis of variance between groups showed that time on task had no significant effect on number of cancellation responsibilities considered ($F(2,15) = .21$, $p < .10$). Pearson's correlation between groups did not find significant correlation between the number of responsibilities considered by participants, and any factor other than the experimental condition, such as industrial experience in programming or software design, age, or gender. We are continuing further analysis of our data, and any other findings of interest will be reported in future work.

## 4. DISCUSSION

The experiment showed a strong main effect of providing the full USAP on the number of responsibilities participants considered in redesigning a software architecture to add the ability to cancel a command. Participants who received only the cancellation scenario considered, on average, only a third of the responsibilities considered by participants who were given the full

USAP. This indicates that the full USAP provided significant help to the participants in remembering the responsibilities they needed to consider when modifying software to add command cancellation functionality.

The absence of significant correlation between time on task and task performance between conditions indicates that the gains in responsibilities considered were made without an associated speed/accuracy tradeoff. In many cases, giving additional information causes people to spend more time on a task, and then the gains in performance cannot be unambiguously attributed to the materials themselves in light of the additional time on task. However, our results are clean: the additional information in the full USAP increases performance with no additional time. This is especially gratifying because, as discussed in the introduction, the full USAP is quite detailed and complex. The additional information in the sample solution and its relationship to the cancellation responsibilities, might have been so difficult to apply that it might have increased time while increasing coverage. However, administering the full USAP materials allowed the participants to create a far more complete solution than administering a general cancellation scenario, in nearly the same amount of time.

The lack of significant interaction effects of moderator variables implies that the USAP was valuable to people with varied levels of experience. Factors such as self-reported experience (e.g., programming, software design) or familiarity with related technologies (e.g. component diagrams, design patterns) did not interact with the experimental conditions, indicating that the usefulness of a USAP was not tied to whether the participant was a novice or an expert programmer and/or software designer. However, because we have only a small number of participants in each condition, this indication is suggestive and not conclusive. To have confidence in this result would require many more experimental participants.

The size of the main effect we discovered is a strong indication of the usefulness of full USAPs. However, checklists alone are commonly accepted memory aids used in software development practice. Therefore, prior to this experiment, we had hypothesized that participants who received the list of general responsibilities would produce a solution that considered more responsibilities than the solutions produced by recipients of the scenario alone. Our results show a trend in this direction, but it did not reach statistical significance with our small number of participants (n=6 in each condition). In adding a sample solution in the third condition, we considered that positive effects gained from learning from an example [2] might be limited by difficulties in extrapolating and generalizing from an example in J2EE-MVC to a solution for the single-threaded Palm OS4. Our results indicate that the improvement gains in working with a checklist and a sample solution, taken together with the absence of a speed/accuracy tradeoff, outweigh the costs of assimilating the additional information.

Looking deeper, we can examine which responsibilities associated with canceling a command were covered by the participants who received only the scenario and which benefited from the greater detail of the other portions of the USAP. For 15 of the 19 Cancellation Responsibilities (CRs), Chi-square tests revealed no significant difference between training conditions, however, four CRs stood out as being considered by the majority of participants in each condition: CR1, CR5, and CR10 [Table 1]. On the other hand, several important cancellation responsibilities were entirely ignored by participants who received only the general scenario, and Chi-square tests produced a significant difference ($p < .05$) between these participants and participants who received a list of responsibilities or a list and a sample solution: CR4, CR13, CR 17 and CR18 [Table 1]. We will examine each of these in turn.

CR1 requires a means to be provided for the user to cancel the active command. This means could be a button, a menu item, a keyboard shortcut, or something else. All participants, except one, received credit for considering that the user must be able to instigate the cancellation process. The sole participant who failed to consider this responsibility received credit for only the general scenario. Thus, remembering to bring access to a new command out to the user seems to be easy for software architects.

CR5 requires the active command to cancel itself upon request, if it is able to do so. Again, nearly all participants did consider this responsibility, the sole exception having been in the scenario-only experimental condition. However, since adding the ability to cancel was the task the participants were given, it is not surprising that most of them received credit for considering this responsibility. On the other hand, only 4 participants considered the failure case where a command is unable to cancel itself. That is, only 4 participants got credit for either CR6, CR7, or CR8 in their solutions, which require the system and/or infrastructure to make provisions for canceling the active command if the command cannot successfully cancel itself. Provisions for these contingencies were made only by participants who received the list of responsibilities (2 participants) and those who received the list plus a sample solution (2 participants), and were never included by a participant who received only the scenario. Thus, remembering to provide for this failure case seems more difficult for software architects.

CR10 requires that system state be restored to the state that existed prior to the execution of the command. Fourteen participants did consider this responsibility in their solutions, and all who received a Training Document that included the general scenario, cancellation responsibilities, and sample solution, did in fact consider this responsibility. Restoring state is an integral part of cancellation, again extremely tied to the task given to the participants and it is not surprising that most participants considered this responsibility. However, as with CR5, few participants considered the failure condition. No participants in the scenario-only condition, and only 3 participants in the other conditions combined, considered CR11 and CR12, which require the system to make provisions for partial system rollback and user feedback, in the case that full system state restoration is not possible. Participants who received only the scenario never created solutions that provided for these contingencies. This is additional evidence that remembering to provide for failure cases can be difficult.

Finally, another responsibility that provided for failure contingency, CR16, requires the system to notify the user in the event that control cannot be returned to the user during a long cancellation process. This responsibility was not considered by a single participant.

Now we turn to the cancellation responsibilities that were considered very differently between the scenario-only participants and those who received a list of responsibilities in their Training Documents. CR4 requires the software to acknowledge receipt of the cancellation command to the user. This responsibility is taken care of by default in most new GUI builders. This may explain why it was not explicitly considered by the participants, and may also diminish the importance of the omission. However, both CR4, and CR18 below, require the integration of another important usability heuristic, visibility of system status. The failure of participants to address system visibility through the provision of a general scenario is an additional strong indication of the insufficiency of providing usability heuristics or scenarios without implementation guidelines.

CR13 requires the software to free system resources used to process the command being cancelled. Freeing system resources is an important consideration that may have serious consequences if it is overlooked. Participants in a CHI tutorial given by John et al. [11] provided anecdotes about their experiences with implementing cancellation. One participant reported that his company had implemented cancel in an application but had forgotten to release resources. The result of this omission was that every time the user pressed the cancel button, it left 500 MB of garbage on the user's hard drive, dramatically illustrating the importance of remembering that software must be responsible for freeing any resources it has consumed.

In the case of CR17, estimating the time required to cancel the command, and CR18, informing the user of the estimated time required to cancel the command, it is difficult to separate the consideration of concerns. If a software engineer did not consider estimating the time required to cancel, this information cannot be provided to the user. Alternatively, a software engineer could provide the facility to estimate the time for cancellation but forget to display that estimate to the user, and the external results would look the same. These two responsibilities, taken together, can be mapped to real world software usability flaws. For instance, in the six months preceding the writing of this paper, we observed mis-implementation of canceling a command in half a dozen popular commercial software programs. For example, the cancel button in well known diagramming and video editing applications required minutes to halt a process, during which it was impossible to tell whether cancellation was in progress. The cancel button in a major email program did put up a cancellation progress indicator, but did not halt the original process, which eventually timed out. We conclude that helping software architects remember to provide for this level of detail could make a real contribution to the usability of real-world software.

The improved consideration of many of these responsibilities in the solutions of participants who had a list of general responsibilities, clearly indicates that the USAP had a positive impact on the attention paid to these demonstrably relevant usability concerns. However, some of the responsibilities were handled inconsistently, and not included in the solutions even by participants who had received them as part of their training materials. This is an indication that there is still room for improvement in our delineation of such responsibilities for the use of software developers and architects.

## 5. CONCLUSIONS AND FUTURE WORK

Using a full USAP increased the number of responsibilities that participants considered in an architectural redesign to add cancellation to the existing architecture design for the PAMD system.

Participants who used all three parts of the cancellation USAP were able to identify and address three times as many cancellation responsibilities, on average, as participants who received only a general usability scenario, in the same amount of time, and without having more work experience or formal training prior to the task. Thus, USAP for canceling a command can already be considered a valuable tool for modifying software architecture designs to address a specific usability concern. However, more work needs to be done to increase the consistency with which software architects apply the USAPs, perhaps in the format of the USAP itself or in training provided with the USAP.

In addition to counting the responsibilities considered in a redesign, the quality of the software architecture design that results from such considerations is also of concern. Additional analysis is currently in progress to assess the quality of the software architectures produced by the participants. This qualitative analysis will be performed by expert software architects external to our research project. Such assessment will allow us to examine correlations between the number of cancellation responsibilities considered, which specific responsibilities were considered, and the overall quality of the software architecture solution to the problem of adding cancellation functionality to an existing architecture design.

Additionally, this study involved only a single usability-supporting architectural pattern, and its effects in modifying a single, specific, software architecture design. In our previous work we have identified more than two dozen usability scenarios that require USAPs [5]. Therefore, future investigations will seek to determine how our results can be replicated and extended across additional USAPs, as well as other types of software architectures.

Perhaps the greatest surprise, however, is not that USAPs help to produce a more complete design, but that they are necessary at all. There is an implicit assumption in the world of human-computer interaction, and perhaps more generally, that software engineers simply know how to implement whatever requirements are provided to them. While this may be true in some areas which are more traditionally included in the training of either computer scientists or software engineers, it is not true in the case of software engineering for usability. Usability issues have been seen to comprise more than 60% of requirements-related defects in some professional software development projects [16]. Our results cast further doubt on the validity of the assumption that software engineers can readily move from usability requirements to successful design implementations.

One final conclusion involves the relationship between the field of usability and software engineering. Usability specialists often advise software engineers with heuristics [13] even less detailed than the scenario given in this experiment [Fig 1.]. Software engineers are also currently trained to view user-related functionality as separate from core functionality, and to omit functionality that is seen as part of the user interface from all but the latest stages of the requirements engineering process [15]. We have come to believe that the implications of usability heuristics for software design are not obvious and should be made explicit to software designers as well as software engineering students. Indeed, this is the motivation for our work with USAPs. The results of this experiment lend strong support to our conviction.

## 6. ACKNOWLEDGEMENTS

We wish to thank Sara Kiesler, Lynn Carter and the PDC scholars at Carnegie Mellon University's West Coast Campus, Rob J. Adams, Andrew Ko, Jacob Wobbrock, our expert evaluators, and the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NASA or the U. S. Government.

## 7. REFERENCES

[1] Adams, R.J., Bass, L., and B.E. John, (in press) Applying general usability scenarios to the design of the software architecture of a collaborative workspace. In A. Seffah, J. Gulliksen and M. Desmarais (Eds.) *Human-Centered Software Engineering: Frameworks for HCI/HCD and Software Engineering Integration*, Kluwer Academic Publishers.

[2] Anderson, J.R. and J.M. Fincham, "Acquisition of Procedural Skills from Examples", *Journal of Experimental Psychology: Learning, Memory and Cognition*, 20, 1322-1340.

[3] Bass, L., Golden, E., John, B.E., Juristo, N., Moreno, A, and M-I. Sanchez-Segura, "Unraveling the Myths of Developing Usable Software", submitted to the 27th International Conference on Software Engineering, ICSE 2005, St. Louis, MO, May 15-21, 2005.

[4] Bass, L. and B.E. John, "Supporting Usability Through Software Architecture", *IEEE Computer*, 34 (10), 113-115.

[5] Bass, L., John, B.E., and J. Kates, "Achieving Usability Through Software Architecture", Carnegie Mellon University/Software Engineering Institute Technical Report No. CMU/SEI-TR-2001-005, 2001.

[6] Bass, L., John, B.E., Juristo, N. and M-I. Sanchez-Segura, "Usability and Software Architecture," tutorial materials presented at the 26th International Conference on Software Engineering, ICSE 2004, Edinburgh, Scotland, May 23-38, 2004.

[7] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. & J. Stafford, Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2003.

[8] Eguiluz, H., Govi, V., Kim, Y.J. and A. Sia, PAMD: Developing a Plug-In Architecture for Palm OS-Powered Devices Using Software Engineering, Technical Note CMU/SEI-2002-TN-020, Carnegie Mellon University, Pittsburgh, PA, 2002.

[9] John, B.E. and L. Bass., "Avoiding 'We can't change THAT!': Software Architecture and Usability", tutorial materials presented at CHI 2003, Ft. Lauderdale, FL, April 5-10, 2003.

[10] John, B.E., Bass, L., Juristo, N., and M-I. Sanchez-Segura, "Avoiding 'We can't change THAT!': Software Architecture and Usability", tutorial materials presented at CHI 2004, Vienna, Austria, April 24-29, 2004.

[11] John, B.E., Bass, L., Sanchez-Segura, M-I., and R.J. Adams, "Bringing Usability Concerns to the Design of Software Architecture", *Proceedings of the 9th IFIP Working Conference on Engineering for Human-Computer Interaction and the 11th International Workshop on Design, Specification and Verification of Interactive Systems*, Hamburg, German, July 11-13, 2004.

[12] Naveda, J. Fernando, "Teaching Architectural Design in an Undergraduate Software Curriculum", *ASEE/IEEE Frontiers in Education Conference Proceedings*, 1999, 12b1-4.

[13] Nielsen, J., Heuristic evaluation. In Nielsen, J., and Mack, R.L. (Eds.), *Usability Inspection Methods*, John Wiley & Sons, New York, NY, 1994.

[14] Prechelt, L., Unger, B. Philippsen, M. and W. Tichy, "Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance", *IEEE Transactions on Software Engineering*, 28, 6, 595 – 606, June 2002.

[15] Van der Veer, G. and H. van Vliet, "A Plea for a Poor Man's HCI Component in Software Engineering Curricula," *Proceedings of the 14th Conference on Software Engineering Education and Training*, Charlotte, NC, February 19-21, 2001.

[16] Vinter, O., "Experience-Based Approaches to Process Improvement," Proceedings of the Thirteenth International Software Quality Week, Software Research, San Francisco, CA, 2000.