Early-Stage Software Design for Usability

Elspeth Golden

May 2010
CMU-HCII-10-103

*Submitted in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy in Human-Computer Interaction*

Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

*Submitted in partial fulfillment of
the requirements of the Program for
Interdisciplinary Education Research (PIER)*

Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This work was funded in part by the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298, by ABB Inc., and by the Institute of Education Sciences, U.S. Department of Education, through Grant R305B040063 to Carnegie Mellon University. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and should not be interpreted as representing the official opinions, either expressed or implied, of the funding agencies or the U.S. Government.

Keywords

Usability; software architecture; design tools and techniques; curriculum design

Abstract

In spite of the goodwill and best efforts of software engineers and usability professionals, systems continue to be built and released with glaring usability flaws that are costly and difficult to fix after the system has been built. Although user interface (UI) designers, be they usability or design experts, communicate usability requirements to software development teams, seemingly obvious usability features often fail to be implemented as expected. The impact of usability issues becomes increasingly severe in all kinds of software as computer use continues to rise in the home, in the workplace, and in education. If, as seems likely, software developers intend to build what UI designers specify and simply do not know how to interpret the ramifications that usability requirements have for the deep structure of the software (i.e. the "software architecture"), something is needed to help to bridge the gap between UI designers and software engineers to produce software architecture solutions that successfully address usability requirements. Usability-Supporting Architectural Patterns (USAPs) achieve this goal by embedding usability concepts in materials that can be used procedurally to guide software engineers during the complex task of software architecture design.

In order for usability to be a first-class citizen among software quality attributes, usability design must be made cost-effective for development organizations. To achieve this goal, usability needs to be addressed early in the design process in ways that enable it to be successfully incorporated into software architecture designs and software engineering implementations. Preventing late-stage changes in complex software systems by addressing architecturally-sensitive usability concerns during the architecture design phase is a victory for developers and users alike. This is the goal that this dissertation addresses.

Addressing usability early in the software development process is a non-trivial problem. The usability dicta provided by guidelines, heuristics, and UI design patterns may not give software designers all the information they require to completely and correctly implement basic usability features. I present work to construct and evaluate an approach to the problems of explicitly drawing and communicating the connection between usability concerns and software architecture design, and a tool to deliver that approach to software engineers for use in software architecture design practice. I also discuss a possible extension of the research concepts embodied in the tool to the domain of education research.

Acknowledgements

There are so many people to thank.

First of all I would like to thank my advisors, Bonnie John and Len Bass, who have been the best academic mentors a student could hope to have. Between Bonnie's indefatigable pursuit of better answers and Len's unflappable humor in the face of the vicissitudes of research, they have provided a tremendous wealth of opportunities for learning what it means to be a researcher and a teacher. For this I am grateful, and even more grateful for what they have shared about the individual nature of living well.

I would also like to thank my other committee members, Robin Jeffries and Sharon Carver, each of whom contributed meaningful insights during the writing process that changed the course of the final story, and improved it in ways I had not anticipated.

Many people provided practical support and helpful advice across the course of the work, including Mary Scott, Brandy Renduels, Audrey Russo, Rob Adams, Sandy Esch, Andy Ko, Laura Dabbish, Howard Seltman, John Pane, Nuno Nunes, and Dave Roberts. Special thanks go to David Klahr and the PIER program; in addition to funding a good deal of my graduate work, PIER expanded my horizons in directions I would not otherwise have experienced as a student in HCI.

For a thoroughly enjoyable research collaboration, and for their delightful hospitality, I am grateful to Pia Stoll, Fredrik Alfredsson, Sara Lövemark, and the software architects who participated in the user tests at ABB in Västerås.

So many wonderful friends have made CMU a home during these years: Amy, Marty, Lisa A., Carson and Nathan, Becky, Ido and Ofira, Matt, Anna and Mike, Ben, Tara and Kevin. Wherever the four winds may scatter us, we'll always have Pittsburgh, Pennsylvania!

Nor could I have made it without the love and enthusiastic support of a far-flung home team of family and friends. Jomama, Burton Bear, Thad, Tracey, Samantha, Nicholas – not a large family, but you each radiate enough cherishing to make up for a dozen. Pattie, Rhett & Heather, Brad, Diana, William & Heather, Lisa C., Susan, Kwame, Janna, Scott & Kristin, Brandon, Karen, and so many more – you are an international village of awesomeness.

Finally, for engaged nurturing and silliness above and beyond the call, a paean of gratitude rings out to Eelia and Noel, who tended me from afar as if I were an errant houseplant and kept me laughing throughout the erratic process of finishing this work. Semper amicitia, semper esoterica, je vous aime!

To Bamboo, Bubu, Irene, and Britt, with love for all time.
Save me a kiss and a dance, my darlings.

Table of Contents

List of Figures

Chapter 1. Introduction

1.1 Overview

In spite of the goodwill and best efforts of software engineers and usability professionals, systems continue to be built and released with glaring usability flaws that are costly and difficult to fix after the system has been built. Although user interface (UI) designers, be they usability or design experts, communicate usability requirements to software development teams, seemingly obvious usability features often fail to be implemented as expected. The impact of usability issues becomes increasingly severe in all kinds of software as computer use continues to rise in the home, in the workplace, and in education. If, as seems likely, software developers intend to build what UI designers specify and simply do not know how to interpret the ramifications that usability requirements have for the deep structure of the software (i.e., the "*software architecture*"), something is needed to help to bridge the gap between UI designers and software engineers to produce software architecture solutions that successfully address usability requirements. As described herein, Usability-Supporting Architectural Patterns (USAPs) achieve this goal by embedding usability concepts in materials that can be used procedurally to guide software engineers during the complex task of software architecture design.

In order for usability to be a first-class citizen among software quality attributes, usability design must be made cost-effective for development organizations. To achieve this goal, usability needs to be addressed early in the design process in ways that enable it to be successfully incorporated into software architecture designs and software engineering implementations. Usability issues in software architecture impact several key constituencies. End users care because they have to live with usability flaws that have been left in systems when it becomes too costly to fix them after the system has been built. This is also relevant in the context of educational technology when students and teachers must spend valuable learning time working around usability flaws in the software being used for educational purposes. Software engineers care about usability because they have to deal with the headache of trying to fix the problems too late in the development cycle. Usability professionals care because they are responsible for creating usable systems. Software development organizations care because user satisfaction and late-stage changes affect the bottom line. Preventing late-stage changes in complex software systems by addressing architecturally-sensitive usability concerns during the architecture design phase is a victory for developers and users alike. This is the goal that my dissertation addresses.

Addressing usability early in the software development process is a non-trivial problem. The usability dicta provided by guidelines, heuristics, and UI design patterns may not give software designers all the information they require to completely and correctly implement basic usability features. Perhaps decisions made early in the software development process about the deep structure of the software have precluded incorporating usability concerns. Perhaps HCI researchers who package the wisdom trust too much that it will be possible for the software engineers who implement the system to overcome the burden of earlier decisions, trusting that "many usability problems have fairly obvious fixes as soon as they have been identified" (Nielsen, 1994). I have two related hypotheses that address the problem of incorporating usability into the engineering of software systems.

> Hypothesis 1: Explicitly linking specific usability concerns to architectural decisions early in the software design process will enable software engineers to include better support for specific usability features into the design of software systems.

> Hypothesis 2: If a technique can be shown to successfully support Hypothesis 1, and packaged in a form that is useful and usable for software engineers, they will be likely to use this technique in software design.

In this dissertation, I present work to construct and evaluate an approach to the problems of explicitly drawing and communicating the connection between usability concerns and software architecture design, and delivering that approach to software engineers.

1.2 Approach

Usability is an important quality of interactive software systems that has not been well addressed in the context of software architecture design. The International Standard Organization's Guidance on Usability (ISO 9241-11) defines usability as the "extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use." Software engineers may not have the necessary knowledge of human psychology to effectively address such criteria. Available heuristics for usability, e.g., Neilsen (1994), are only expressed from the user's point of view and provide no guidance for architecture design. In this and the following chapters, I describe an approach to early-stage software design for usability through Usability-Supporting Architecture Patterns (USAPs) and their application and design framework, Architecture Pattern Language for Usability Support (A-PLUS). I will show how, through an iterative process of testing and revising, USAPs have been embodied in a software tool that makes them useful and usable as a performance aid for software engineers designing software architectures to support usability concerns.

Printed materials or software tools that support independent learning in this organizational context are generally classed as performance aids (Foley, 1972). If performance aids are shown to facilitate the development of professional skills in the context of software engineering (e.g., by helping software engineers to consider the ramifications of user-centered requirements), then such performance aids could by extrapolation be useful for professional development in other domains as well. Therefore I also discuss how a wider interpretation of this research could result in its application to education research.

Learning to support usability requirements is a desirable goal for professional software developers, given the increasing importance of usability as a desirable quality for software in all manner of applications. However, like professionals in many disciplines, software engineers and architects have little time for additional training to support new goals. Interrupting a career for full-time education is generally out of the question. Some software development professionals may participate in short-term courses on a specific topic in engineering, but the bulk of learning takes place in an independent context, through books or software tools (Kazman 2003). This type of learning is not subject to formal assessment; the only measure is the ability of the software developer to satisfactorily meet the needs of the organization through software design and implementation. Performance aids are an appropriate learning tool in a professional context where there is insufficient time to go back to school to learn new techniques, and where assessments are weak or nonexistent. The success of USAPs as a performance aid for applying a new technique without formal training has implications for professional development education in fields other than software engineering, including teacher professional development.

A critical step toward providing useful and usable resource materials to support a technique is to ensure that the technique is of value to the target audience, in this case, software developers. An early version of USAPs was applied to several software designs, among them a large commercial information system (Bass & John, 2003) and a wall-sized tool that supports collaboration of the engineers and scientists on NASA's Mars Exploration Rover mission (Adams, John, & Bass, 2005). Those experiences showed that USAPs as originally conceived may be valuable, but possibly too detailed and overly complex, which might make them difficult for software developers to apply to their own problems without the advice of the researchers who developed the USAPs. This lead me to examine different representations of USAPs under controlled experimental conditions to determine which parts of a USAP were useful, usable, and necessary before attempting wider dissemination of USAPs as a tool for software development teams.

This work has been made possible by prior work in diverse disciplines. In the next chapter (Chapter 2), I describe how the previous work in several fields has formed a basis for the work described in the rest of this dissertation. Chapter 2 further details the concept, early forms, and early applications of USAPs that preceded the work discussed in Chapters 3 through 7. Figure 1-1 shows a roadmap for Chapters 3 through 7, which are explained in more detail below.

In Chapter 3, I evaluate the effectiveness of Usability-Supporting Architecture Patterns (USAPs) in helping software engineers to address a specific usability concern when designing the architecture for a software system. I present a pair of studies in which software engineering graduate students apply a USAP to the task of modifying a software architecture design to address a specific usability concern. Using a system of counting consideration of software responsibilities, I show that a USAP is an improvement over providing a usability scenario alone, both in terms of coverage of responsibilities and quality of solution. I investigate the value of different portions of a USAP in the design task, and validate the usefulness and usability of USAPs, applied without researcher intervention, as an approach to improving the consideration of specified usability concerns in software architecture design. The work described in this chapter supports the first hypothesis.

In Chapter 4, I describe the process of creating a new representation of USAPs based on the effectiveness finding of the two studies. Commonalities discovered between several USAPs during this process lead to the creation of Architecture Pattern Language for Usability Support (A-PLUS), a pattern language for generating responsibilities and reusing commonalities content across multiple USAPs. The chapter describes an improved representation of USAPs that enables compression and re-use of responsibilities to support the concurrent use of multiple USAPs in software architecture design practice. The intent of this chapter is to describe A-PLUS as a change in representation of USAPs in the context of how they will be used to support software architecture design, not to teach how to create USAPs using the A-PLUS pattern language. The work described in this chapter was a group effort, and supports the first hypothesis.

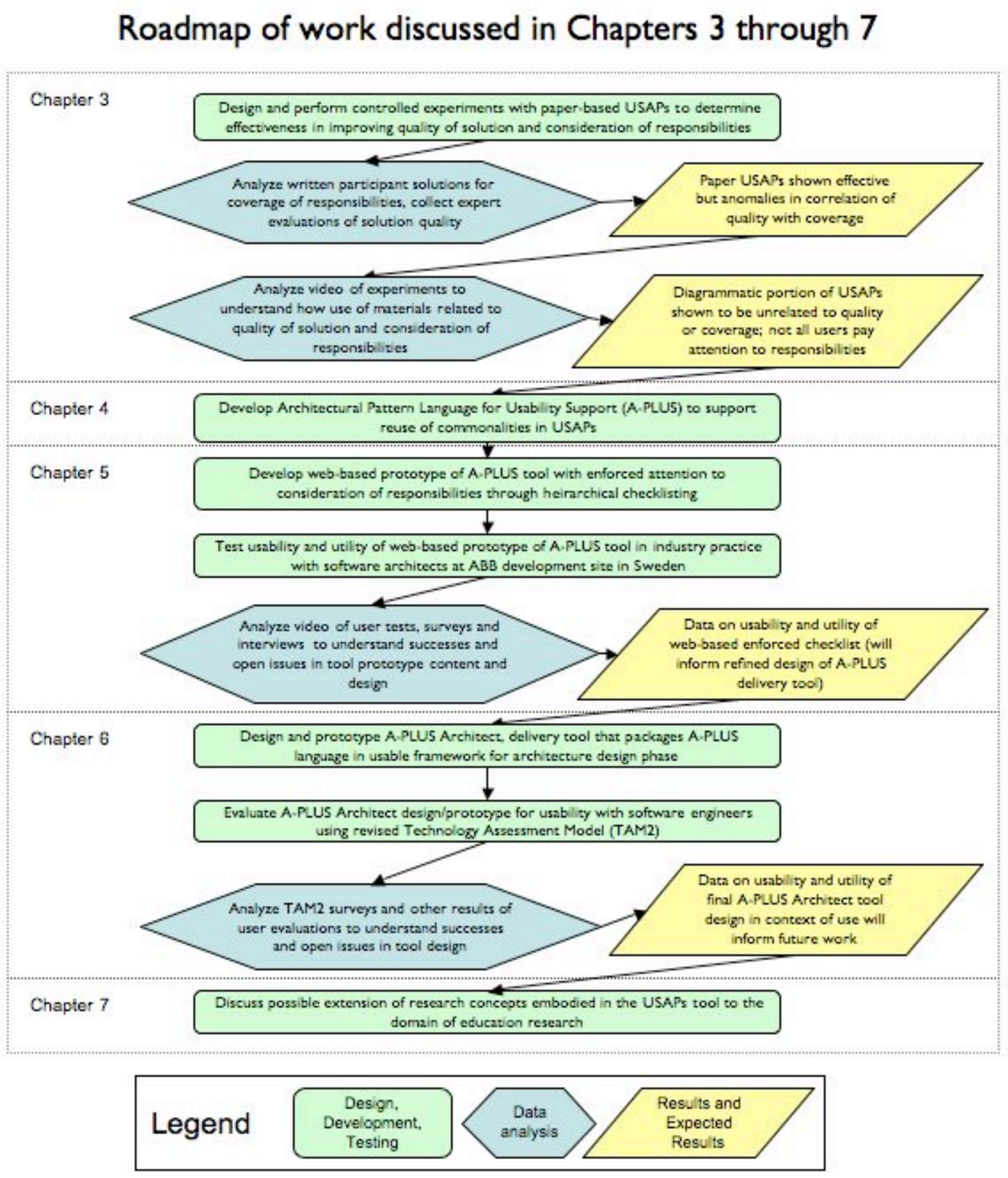

Roadmap of work discussed in Chapters 3 through 7
Chapter 3
Design and perform controlled experiments with paper-based USAPs to determine effectiveness in improving quality of solution and consideration of responsibilities
Analyze written participant solutions for coverage of responsibilities, collect expert evaluations of solution quality
Paper USAPs shown effective but anomalies in correlation of quality with coverage
Analyze video of experiments to understand how use of materials related to quality of solution and consideration of responsibilities
Diagrammatic portion of USAPs shown to be unrelated to quality or coverage; not all users pay attention to responsibilities
Chapter 4
Develop Architectural Pattern Language for Usability Support (A-PLUS) to support reuse of commonalities in USAPs
Chapter 5
Develop web-based prototype of A-PLUS tool with enforced attention to consideration of responsibilities through heirarchical checklisting
Test usability and utility of web-based prototype of A-PLUS tool in industry practice with software architects at ABB development site in Sweden
Analyze video of user tests, surveys and interviews to understand successes and open issues in tool prototype content and design
Data on usability and utility of web-based enforced checklist (will inform refined design of A-PLUS delivery tool)
Chapter 6
Design and prototype A-PLUS Architect, delivery tool that packages A-PLUS language in usable framework for architecture design phase
Evaluate A-PLUS Architect design/prototype for usability with software engineers using revised Technology Assessment Model (TAM2)
Analyze TAM2 surveys and other results of user evaluations to understand successes and open issues in tool design
Data on usability and utility of final A-PLUS Architect tool design in context of use will inform future work
Chapter 7
Discuss possible extension of research concepts embodied in the USAPs tool to the domain of education research
Legend
Design, Development, Testing
Data analysis
Results and Expected Results

Figure 1-1. Roadmap of Work Discussed in Chapters 3-7

In Chapter 5, I describe a browser-based tool, A-PLUS Architect, that was intended to enable software architects to apply the A-PLUS form of USAPs in the architecture design process. The A-PLUS pattern language described in Chapter 4 was an improved representation of USAPs in many aspects, but was not directly usable by software architects. This chapter describes my design for a further improved representation of USAPs, A-PLUS Architect, which embodies the A-PLUS representation of USAPs in a tool that enables multiple USAPs to be combined in a single solution in a form that is both useful and usable for software architects. I then discuss user tests in which software architects in industry used the tool to evaluate proposed and existing architecture designs for software systems. Finally, I describe an iteration on the design of A-PLUS Architect, based on the results of the user tests. This chapter supports both the first and second hypotheses.

In Chapter 6, I describe work to assess the feasibility of getting A-PLUS Architect accepted into work practice in industry. I describe a design-stage evaluation of this tool for perceived usefulness and ease of use. The improvements on the design of the A-PLUS Architect tool, and the work to evaluate software architects' perceptions of the tool in its redesigned form, support the second hypothesis.

In Chapter 7, I discuss a possible extension of the research concepts embodied in the USAPs tool to the domain of education research. While the domain of USAPs is software engineering and HCI, a logical analogy can be drawn from software architecture to educational systems, and from user interface design to curriculum design. Through this analogy, I propose extending the USAPs concept to improve the integration of existing knowledge in curriculum design, research-based instructional guidance and interventions, and the needs for resources from the educational infrastructure.

In Chapter 8, I conclude by discussing the contributions of the work in this thesis to the fields of HCI, software engineering, and education research, and then suggest open questions for future research.

Chapter 2. Prior and Related Work

Solving the problem of bringing together usability and software architecture is an inherently interdisciplinary endeavor. In seeking to solve the problem of incorporating usability into the engineering of software systems, I have drawn on appropriate related work that falls into several categories seated in diverse disciplines, which I will describe in this chapter. Figure 1 below shows the relationship of prior and related research described in this chapter to the work that will be shown in Chapters 3 through 7.

The prior work I discuss in this chapter is shown in the ovals in Figure 2-1. At the top of Figure 1, work in usability guidance comes primarily from the field of human-computer interaction. Software quality attributes are related to software architecture, part of the field of software engineering. Work on usability guidance and software architecture incorporates elements from both these fields. To design a usable system for software architects (upper center of Figure 2-1) I also drew on prior research on checklists to inform my efforts to apply usability knowledge to software architecture design. Checklists have been studied in many fields, including psychology, human-computer interaction, software engineering, and education. To assess the likelihood that software architects would use the system (lower center of Figure 2-1), I used an instrument from the field of information systems, the Technology Assessment Model. To extrapolate a framework for extended the work on usability and software architecture to the education research domain (bottom of Figure 2-1) I drew on prior research in curriculum design. This thesis draw together diverse elements from these varied disciplines to create a solution based on a single theme: making usability knowledge useful for, and usable by, software architects.

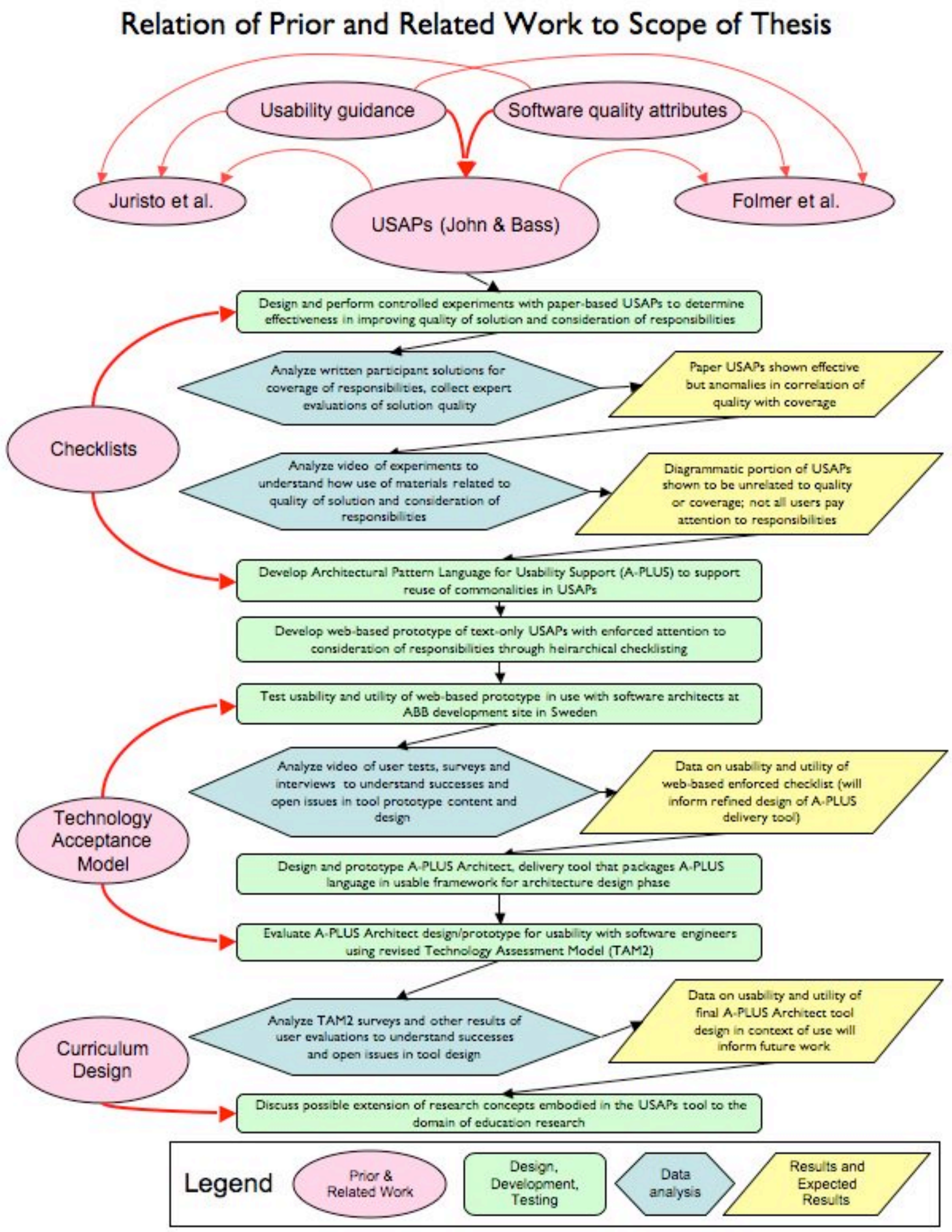

Relation of Prior and Related Work to Scope of Thesis
Usability guidance
Software quality attributes
Juristo et al.
USAPs (John & Bass)
Folmer et al.
Design and perform controlled experiments with paper-based USAPs to determine effectiveness in improving quality of solution and consideration of responsibilities
Analyze written participant solutions for coverage of responsibilities, collect expert evaluations of solution quality
Paper USAPs shown effective but anomalies in correlation of quality with coverage
Checklists
Analyze video of experiments to understand how use of materials related to quality of solution and consideration of responsibilities
Diagrammatic portion of USAPs shown to be unrelated to quality or coverage; not all users pay attention to responsibilities
Develop Architectural Pattern Language for Usability Support (A-PLUS) to support reuse of commonalities in USAPs
Develop web-based prototype of text-only USAPs with enforced attention to consideration of responsibilities through heirarchical checklisting
Test usability and utility of web-based prototype in use with software architects at ABB development site in Sweden
Technology Acceptance Model
Analyze video of user tests, surveys and interviews to understand successes and open issues in tool prototype content and design
Data on usability and utility of web-based enforced checklist (will inform refined design of A-PLUS delivery tool)
Design and prototype A-PLUS Architect, delivery tool that packages A-PLUS language in usable framework for architecture design phase
Evaluate A-PLUS Architect design/prototype for usability with software engineers using revised Technology Assessment Model (TAM2)
Curriculum Design
Analyze TAM2 surveys and other results of user evaluations to understand successes and open issues in tool design
Data on usability and utility of final A-PLUS Architect tool design in context of use will inform future work
Discuss possible extension of research concepts embodied in the USAPs tool to the domain of education research
Legend
Prior & Related Work
Design, Development, Testing
Data analysis
Results and Expected Results

Figure 2-1. Relationship of Prior and Related Work to Scope of Thesis

2.1 Usability and Software Architecture

2.1.1 Software Quality Attributes

To gain some insight into how software engineers and architects perceive usability, let us briefly examine software quality attributes as a category. From the perspective of the field of software engineering, usability is only one of many qualities that a software system may possess, qualities being "the totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs" (ISO/IEC 9126, 2001). Quality characteristics in this context are defined as "attributes of a software product by which its quality is described and evaluated," each of which "may be refined into multiple levels of sub-characteristics" (ISO/IEC 9126, 2001). Stakeholders in the development of software systems may desire the software to have quality characteristics that are not necessarily easily described or directly addressed by the functional requirements of the system, which describe the system's capabilities, services, and behaviors (Bass, Clements, & Kazman, 1998; Bass, Clements, & Kazman, 2003). These qualities are referred to as *quality attributes* of a software system.

The International Standard ISO/IEC 9126 (2001) identifies six main software quality characteristics, each with a number of subsidiary attributes:

- functionality (suitability, accurateness, interoperability, compliance, security),
- reliability (maturity, fault tolerance, recoverability),
- usability (understandability, learnability, operability),
- efficiency (time behavior, resource behavior),
- maintainability (analyzability, changeability, stability, testability), and
- portability (adaptability, installability, conformance, replaceability).

Descriptions of many software quality attributes and their applications may be found in Barbacci, Klein, Longstaff, & Weinstock (1995), and Bass et al. (1998, 2003). Quality attributes identified by Bass et al. (2003, pp. 78-93) are:

- availability (how frequently or for how long a system may acceptably fail),
- modifiability (how difficult or costly a system is to change),
- performance (how long a system takes to respond to an event),

- security (how well a system can prevent unauthorized usage),
- testability (how easily a system can be made to demonstrated its faults through testing), and
- usability (how easily and effectively a user can perform a task).

The lists of identified software quality attributes are continually evolving.

A software system must be designed to meet its functional requirements, but quality attributes desired by the business must also be designed for early in the development life cycle, as changing the system later to accommodate them can be both costly and time-consuming. Quality attributes may also necessitate tradeoffs in the design decision process. For example, performance and security are sometimes at odd with one another, requiring explicit decisions in favor of one or the other to be made in a software system's design. Decisions as to which quality attributes are most important for a software system are driven by business needs. These decisions are reached during the requirements gathering phase of the software life cycle, very early in the development process. Once both functional requirements and quality attributes have been defined, the architecture of a software system may begin to be designed. For an interactive software system, i.e., one with human users, usability is a key quality attribute. Once the usability requirements of an interactive system have been determined, those requirements along with others will be considered in designing the architecture of that system.

Since usability is a quality attribute that must be designed for, but that software engineers might not know much about (Kazman, Gunaratne, & Jerome, 2003), we next turn to the human-computer interaction literature to see what guidance it can provide to software engineers in helping them to design for usability.

2.1.2 Usability Guidance

The field of human-computer interaction has tried in various ways to package usability knowledge so as to guide software design and implementation. Heuristics, guidelines, and UI design patterns are three such types of attempts. All of these are packaged in such a way as to be used primarily by those who design and develop user interfaces, and all are valid and valuable ways to address usability issues in user interface design. However, they fall somewhat short of providing sufficiently detailed functional guidance for the software engineer to apply them with full benefit in system design and implementation.

Heuristics such as Nielsen's (Nielsen & Mack, 1994) are used to evaluate a user interface design to determine whether the design meets a limited number of overarching usability criteria. Nielsen defines heuristics as "general rules that seem to describe common properties of usable interfaces," and describes heuristic evaluation as a "discount usability engineering method, … cheap, fast, and easy to use." Although he claims that "many usability problems have fairly obvious fixes as soon as they have been identified," he admits that the method "does not provide a systematic way to generate fixes" to usability problems once they have been identified through heuristic evaluation. For example, a heuristic like

> "*User control and freedom*: Users often choose system functions by mistake and will need a clearly marked 'emergency exit' to leave the unwanted state without having to go though an extended dialogue. Support undo and redo." (Nielsen & Mack, 1994, p. 30)

provides guidance about the kind of mistake a user is likely to make, and general advice as to how a user interface can be designed to minimize the negative consequences. During a heuristic evaluation of a user interface, this heuristic would uncover any absence of such functionality. However, the heuristic does not suggest a way to address the technical design implications of "undo" and "redo" in the context of a complex software system. Solutions rely instead on the engineering team's knowledge of "established usability principles" and ability to incorporate them in a redesign.

Early example of guidelines were those laid out by Engel and Granda (Engel & Granda, 1975) and Smith and Mosier (Smith & Mosier, 1984). An extensive current example may be found in the National Cancer Institute's web usability guidelines (Koyani, Bailey, & Nall, 2003), which are numerous and supported by research-based evidence. Guidelines provide more explicit functional descriptions of usability-related software development considerations for certain types of system functions, e.g., those related to entering or displaying data. However, when functional suggestions are included they are stated in general terms. Here is a sample NCI guideline titled “Warn of ‘Time Outs’”:

> “Guideline: Let users know if a page is programmed to ‘time out,’ and warn users before time expires so they can request additional time.
>
> Comments: Some pages are designed to ‘time out’ automatically (usually because of security reasons). Pages that require users to use them within a fixed amought of time can present particular challenges to users who read or make entries slowly.” (Koyani, Bailey, & Nall, 2003, p. 14)

While this guideline does provide some usability knowledge about the need for user feedback, the knowledge is only detailed at the level of the user interface. It does not offer software engineers practical insight into how to implement the suggestions in the context of an interactive software system, e.g., identifying processes that must be timed, regulating authorized access, etc. Since many of usability issues also have architectural implications, such guidelines do not go far enough in providing usability guidance for software engineers.

User interface design patterns (Van Duyne, Landay, & Hong, 2002; Tidwell, 2004) are likewise aimed primarily at user interface and interaction designers, not software engineers. The internet-suitable design patterns delineated by Van Duyne and his colleagues are suggested for “Web design professionals, such as interaction designers, usability engineers, information architects, and visual designers,” although they do add that others, including software developers, might benefit by becoming more aware of the “customer-centered design philosophy” embodied in the patterns. Tidwell’s user interface patterns “are intended to be read by people who have some knowledge of UI design concepts and terminology,” including “whitespace” and “branding.” These are design concepts commonly understood by professional user interface designers, but are not terms of art in the field of software engineering.

UI design patterns do make some functional suggestions for steps a software system should provide to successfully interact with a user. Here is an excerpt from Tidwell's "cancelability" UI design pattern, for canceling a long-running command:

> "*What*: Provide a way to instantly cancel a time-consuming operation, with no side effects.
>
> *Use when*: A time-consuming operation interrupts the UI, or runs in the background, for longer than two seconds or so – such as when you print a file, query a database, or load a large file. Alternatively, the user is engaged in an activity that literally or apparently shuts out most other interactions with the system, such as when working with a modal dialog box.
>
> *How*: … If you really do need to cancel it, here's how to do it. Put a cancel button directly on the interface… or wherever the results of the operation appear. Label it with the word Stop or Cancel, and/or put an internationally recognizable stop icon on it: a red octagon, a red circle with a horizontal bar, or an X. When the user clicks or presses the Cancel button, cancel the operation immediately. If you wait too long, for more than a second or two, the user may doubt that the cancel actually worked… Tell the user that the cancel worked – halt the progress indicator, and show a status message on the interface, for instance. Multiple parallel operations present a challenge. How does the user cancel a particular one and not others? … If the actions are presented as a list or a set of panels, you might consider providing a separate cancel button for each action, to avoid ambiguity." (Tidwell, 2006, p. 151)

As you can see, this pattern communicates usability knowledge about appropriate timing and user feedback for canceling a long-running command in a software system. It provides more explicit advice in terms of providing for user interaction than usability guidelines do. However, it stops short of considering related issues that the user may not directly see, but which will affect the user's experience, e.g., having the software system listen for the Cancel button, or providing appropriate user feedback if the system fails to cancel the command. In other words, UI design patterns still rely on the software engineer to understand and generate or recall unstated deeper implications of user interface design elements. Their guidance is helpful but not sufficient for software engineers who are designing the functionality of complex interactive systems.

2.1.3 Usability & Software Architecture Patterns

During the 1980s, separation of the user interface from the core functionality of many software systems became common. Although usability is acknowledged to be an important quality of interactive software systems, it has been treated as a subset of modifiability. Software architects and engineers commonly assume that usability issues that arise during user testing can be handled with localized modifications. As recently as 1999, usability has been labeled as a quality attribute that could not be meaningfully addressed in the context of software architecture design (Bass et al., 1999). Usability concerns were relegated to the user interface (UI) design phase of the software life cycle, which was and is routinely performed long after the architecture of a software system has been chosen and detailed.

Unfortunately, simply separating the interface from the functionality does not support all usability concerns. In fact, many usability concerns reach deeply into the architecture of a software system. This can have costly repercussions when usability is not considered early in the design process and support for usability concerns are not designed into the architecture of the system. Usability problems found in user testing can require extensive and costly re-architecting of software systems. When this happens, projects either are burdened with extra costs and delays, or, as often happens, cannot afford the additional costs and therefore ship products with known usability issues that could have been prevented if they had been considered earlier in the software life cycle.

To address this problem, research on the relationship between usability and software architecture by John & Bass (2000, 2001, 2004) has led to the development of USAPs, each of which addresses a usability concern that is not addressed by separating interface from functionality. The USAPs approach will be described in detail in following chapters. Other approaches to the problem have been proposed by Folmer et al. (2003, 2004), and by Juristo et al. (2001, 2003, 2007), which will be discussed in the following sections.

In 2000, Bass and John (Bass & John, 2000) suggested the development of collection of Attribute-Based Architectural Styles (ABASs) for aspects of usability that might be affected by changes in a system's software architecture. An ABAS is a structured description of a measurable quality attribute, a particular architectural style, and the relevant qualitative and quantitative analysis techniques (Klein et al., 1999). The sample ABAS they proposed was to be "used to reason about whether a proposed software architecture will facilitate users being able to cancel their last operation." The eight-page ABAS included four main ideas: a problem statement, stimulus/response measures, an architectural style, and an analysis section describing "how to reason about a solution in terms of … measurable responses."

In 2001, Bass et al. further refined their work on ABASs and the connection between usability and software architecture by identifying twenty-six general scenarios that defined architecturally sensitive aspects of usability (Bass, John, & Kates, 2001) and describing an architectural pattern that would address each scenario. They also introduced a hierarchy of benefits each scenario would provide to a user of the system, since

> "[a]ssuming that the functionality needed by a system's users is correctly identified and specified, the usability of such a system can still be seriously compromised by architectural decisions that hinder or even prevent the required benefits. In extreme cases, the resulting system can become virtually unusable."

Each usability scenario covered one or more positions in the hierarchy of usability benefits. These combinations of architecturally-sensitive usability scenarios, benefit hierarchies, and architectural solutions were later named Usability-Supporting Architectural Patterns, or USAPs (John, Bass, Sanchez-Segura, & Adams, 2004).

The main work described in the remainder of this thesis examines the formulation and application of USAPs for addressing specific usability concerns during software architecture design. Before describing USAPs, however, I will discuss two other approaches that have been suggested to investigating the relationship between usability and software architecture, as well as contributing factors from other fields as shown in Figure A above.

2.1.4 Juristo et al.'s Approach

Both this approach and the approach in section 2.1.5 below were proposed by groups of researchers within the STATUS (SofTware Architectures That support USability) project, an initiative of the European Union.

The first of these groups proposed to support usability in software architecture by "decompos[ing] usability into levels of abstraction that are progressively closer to software architecture." (Juristo, Lopez, Moreno, & Sanchez-Segura, 2003) This approach suggested first breaking down the concept of usability into constituent attributes – learnability, efficiency of use, reliability, and satisfaction – and then mapping relationships between these attributes, usability "properties" phrased as simple guidelines, and user interface patterns that might support those properties. They then attempted to map relationships between user interface patterns and traditional software architecture patterns. The approach suggested a workflow where the software architect would trace the relationship back from the desired usability attribute (e.g., satisfaction) through the associated usability property (provide explicit user control) and a user interface pattern to support that property (undo) to know to use a software architecture pattern (undoer) that would provide underlying functionality to the user interface pattern.

They later proposed a set of guidelines to help software engineering teams determine what usability functionality would be required by stakeholders in a software system. (Juristo, Moreno, & Sanchez-Segura, 2007; Juristo, Moreno, Sanchez-Segura, & Baranauskas, 2007). These guidelines were intended for use during the requirements-gathering process to assist the elicitation of functional usability requirements. While both these guidelines and the mapping approach above describe processes by which user interface patterns could be mapped to software architecture patterns to improve usability, it has not been shown how these approaches would function in an actual software architecture design task.

2.1.5 Folmer et al.'s Approach

The second approach to emerge from the STATUS project took a similar tactic as far as connecting usability attributes, usability properties, and user interface patterns. In their work, they replaced the term "user interface pattern" with "architecturally sensitive usability pattern," which they defined as "a technique or mechanism that should be applied to the design of the architecture of a software system in order to address a need identified by a usability property at the requirements stage (or an iteration thereof)." (Folmer, van Gurp, & Bosch, 2003)

This group's next step, similar to the work of Juristo et al., was to propose a method of assessing usability needs during the requirements-gathering phase of the software life cycle. Their method focused on "usage scenarios," defined as "interactions between independent entities[:]… the user (as a stakeholder)… the context in which the user operates (as part of the environment)… [and] the tasks that a user can perform (as part of the system)." (Folmer, van Gurp, & Bosch, 2004) They suggested using use case maps to capture design decisions regarding usage scenarios and software architecture. They also suggested a framework to map the problem domain (Folmer & Bosch, 2004; Folmer & Bosch, 2008).

As with the work of Juristo et al., there is no operationalized process by which the software architect can proceed from the "architecturally sensitive usability pattern" to a software architecture pattern. As will be shown in the remainder of this thesis, my work in USAPs fills this gap by providing a detailed, useful, and usable process for supporting the quality attribute of usability in the software architecture design process.

2.2 Contributing Factors from Other Fields

As seen in Figure 2-1 above, material from several additional fields of research was drawn in to support the work in this thesis. I describe these diverse subjects below.

2.2.1 Checklists

Because part of the intent of USAPs with respect to software architects was to help them remember to attend to numerous considerations within a predefined task, the use of lists and checklists informed both the design and content of the USAPs materials described in Chapter 3 and the web-based tool prototype described in Chapter 5. To support this work I looked in diverse areas for circumstances where checklists might have been used to help people perform tasks they had not performed before, but for which they might be otherwise qualified, since the design tasks the software architects perform often call upon existing knowledge but are not routine.

In the fields of human factors and aviation psychology, Degani and Wiener have done important work on the use of checklists as memory aids (Degani &Wiener, 1990; Degani &Wiener, 1993). Their research addressed use of checklists by cockpit crews in the performance of both normal and non-normal procedures, showing checklists to be effective as memory aids. In these studies, the cockpit crews were performing tasks for which they had been trained. In both normal (e.g., preparing for takeoff) and abnormal procedures the cockpit crews were performing series of tasks on which they had been trained previously. In normal procedures, "the checklist is intended to achieve the following objectives:

1. Aid the pilot in recalling the process of configuring the plane.
2. Provide a standard foundation for verifying aircraft configuration that will defeat any reduction in the flight crew's psychological and physical condition.
3. Provide convenient sequences for motor movements and eye fixations along the cockpit panels.
4. Provide a sequential framework to meet internal and external cockpit

operational requirements.

5. Allow mutual supervision (cross checking) among crew members.
6. Enhance a team (crew) concept for configuring the plane by keeping all crew members "in the loop."
7. Dictate the duties of each crew member in order to facilitate optimum crew coordination as well as logical distribution of cockpit workload.
8. Serve as a quality control tool by flight management and government regulators over the pilots in the process of configuring the plane for the flight." (Degani & Wiener, 1990, p. 7)

Although some general objectives of using checklists in normal procedures apply to the tasks we will consider in Chapters 3 and 5 – helping with recall, allowing mutual supervision, enhancing quality control – the function of checklists in abnormal procedures, defined as "emergencies and/or malfunctions of the aircraft systems," is more relevant to our case. During abnormal procedures, the checklist "serves to

1. Act as memory guide.
2. Ensure that all critical actions are taken.
3. Reduce variability between pilots.
4. Enhance coordination during high workload and stressful conditions." (Degani & Wiener, 1990, p. 8)

In both normal and abnormal circumstances, checklists were found to be an effective memory aid. Checklists have proven so effective in aviation that their use in medical procedures is now being studied and advocated (Gawande, 2009). However, a "design weakness" that Degani & Wiener noted in the "traditional (paper) checklist" was that "if the individual… chooses not to use the checklist for any reason, no one can force him to use it. (Degani & Wiener, 1990, p. 60" Additionally, none of this research examined the use of checklists to perform tasks that had never before been performed by the people using the checklists.

In the field of software engineering, Porter et al. examined the use of checklists as one of several fault detection methods in software requirements inspections (Porter & Votta, 1994; Porter, Votta, & Basili, 1995). In their work, they compared three methods of manually inspecting software designs for faults: ad hoc inspection using a general taxonomy of faults, checklist-based inspection using a subset of the faults in the ad hoc taxonomy, and scenario-based inspection using groups of scenarios representing subsets of the checklist items. In experiments with both students and software professionals, their findings were not especially favorable for checklists. As a technique, checklists provided no better fault detection than ad hoc inspection, and both of these techniques were inferior to scenario-based fault detection, which provided an improvement in the fault detection rate from 35% to 51% in student participants and from 21% to 38% in the professional participants (Porter & Votta, 1998). Although it may seem at first glance that we should take these results as an indication to throw out a checklist-based approach, Chapter 3 will show an approach uses a combination of scenarios and checklists, for which Porter & Votta's results may presage a good result.

2.2.2 Technology Acceptance Model

One goal of my research is to produce knowledge that can be used successfully in nonacademic settings. USAPs were conceived with the ultimate goal of being used by software architects in professional practice. However, it is one thing to create a new technique or tool for software development but quite another to get that technique or tool accepted into practice in professional use. For a way to measure the likelihood of technology acceptance for new technologies, the field of information systems has developed the Technology Assessment model, which was used during the user tests described in Chapter 5 and the design evaluation of Chapter 6.

The Technology Acceptance Model, or TAM, is an adaptation of an intention model from social psychology, Fishbein and Ajzen's theory of reasoned action, specifically designed to explain computer usage behavior. (Davis, Bagozzi, & Warshaw, 1989) TAM was designed to explain the causal links between "perceived usefulness and perceived ease of use" on the one hand, and "users' attitudes, intentions and actual computer adoption behavior" on the other. In the model, perceived usefulness is defined as "the prospective user's subjective probability that using a specific application system will increase his or her job performance within an organizational context. Perceived ease of use is defined as "the degree to which the prospective user expects the target system to be free of effort." The model posits that these two factors will affect a user's attitude toward using a system and behavioral intention to use the system, which will in turn predict actual system use. TAM was empirically validated through a number of studies by its originator, Davis, and others (Davis, 1989; Davis et al., 1989; Adams, Nelson, & Todd, 1992).

Survey items from the Technology Acceptance Model used in this work were from the TAM2 (Venkatesh & Davis, 2000; Venkatesh, Morris, Davis, & Davis, 2003). The TAM2 is a validated survey instrument for predictive assessment of use of information systems. Items in the survey instrument measure the perceived usability and usefulness of a software system (or other product) once the system has been created. The instrument is easy to administer and measures attitudes toward the new system, not task performance. Perceived usability and usefulness are critical to new methods and tools in a professional development environment because users, unless they are forced, will not use software unless they perceive it as both helpful and reasonably easy to use.

2.2.3 Curriculum Design

Chapter 7 proposes a hypothetical framework for extending my work on usability and software architecture to the education research domain. Research in curriculum design, from the field of education research, provides the underpinning for this extension.

Curriculum may be described as "the knowledge and skills in subject matter areas that teachers teach and students are supposed to learn…. [It] generally consists of a scope or breadth of content in a given subject area and a sequence for learning…. Standards… typically outline the goals of learning, whereas curriculum sets forth the more specific means to be used to achieve those ends." (Pellegrino, 2006) Curriculum may then be understood to be the tactics used to achieve the strategic goals already set out in standards. This is a narrow but not uncommon definition of curriculum.

Much debate in education research surrounds the questions of what kinds of learning environments best foster education, and how to create those environments. Four general types of learning environments are described with the labels learner-centered, knowledge-centered, assessment-centered, and community-centered environments. Learner-centered environments focus on the "knowledge, skills, attitudes, and beliefs that learners bring to the educational setting," including cultural and language practices. Knowledge-centered environments are concerned with helping students come to terms with "well-organized bodies of knowledge that support planning and strategic thinking" by "learning in ways that lead to understanding and subsequent transfer." Assessment-centered learning environments focus on providing "opportunities for feedback and revision" and making certain that what is assessed is "congruent with one's learning goals." Community-centered learning environments, a more recent coinage, recognize and attempt to measure "norms for people learning from one another and continually attempting to improve." (Bransford, Brown, & Cocking, 2000)

Curriculum design does not exist independent of learning environment. Chapter 7 lays the groundwork for extending the USAPs method to the education domain by drawing an analogy between software architecture and the educational system (as learning environment), and user interface design and curriculum design.

In the following chapters, I will show how these seemingly disparate elements combine (as shown in Figure 2-1) to inform an approach to solving the problem of making usability knowledge useful to, and usable by, software architects, and how the solution could be generalized to apply to the field of education research.

Chapter 3. Empirical Studies

This chapter describes work to evaluate the effectiveness of a single Usability-Supporting Architecture Pattern (USAP) through controlled experiments in a laboratory setting. Previous work on USAPs, as discussed in the last chapter, stopped short of experimental validation of the form of the USAPs that had been developed by Bass and John, a logical step before investing the extensive time and effort required to fully develop multiple USAPs. I studied the effectiveness of a single USAP in helping software engineers to address a specific usability concern when designing the architecture for a software system. In the studies described in this chapter, software engineering graduate students applied a USAP to the task of modifying a software architecture design to address a specific usability concern. I will describe how I used a system of counting the consideration of software responsibilities in a software architecture design to show that a USAP is an improvement over providing a usability scenario alone, both in terms of coverage of responsibilities and quality of solution. I will also show the results of video analysis through which I examined the value of different portions of a USAP in the design task.

3.1 Experiment 1 – Consideration of Responsibilities

To examine whether all parts of a USAP were useful, usable, and necessary, I designed and carried out a controlled experiment to assess the value of the different parts of a USAP in modifying a software architecture design. The USAP chosen for the experiment supports an important usability concern: canceling a long-running command. The experimental protocol asked software engineers to apply this USAP to the task of redesigning the architecture for a software system 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. Much of the work described in this chapter has already been published in (Golden, John,

& Bass, 2005a) and (Golden, John, & Bass, 2005b).

3.1.1 Experimental Design

The experiment 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 software architecture experts for correctness and completeness with respect to software architecture, prior to conducting the experiment. Four of these experts were from academia and four from industry. In addition to the instructional and task materials they were asked to critique and approve a potential solution to the redesign task. All stimuli from the experiment can be found in Appendices B through G, with excerpts shown in this chapter.

The Training Document received by participants in the first condition, scenario-only (S), contained only a usability scenario describing circumstances under which a user might need to cancel an active command. This scenario was a single paragraph of prose, similar to the language that a usability expert might typically use to recommend that cancellation capability be added to an application (Figure 3-1; Appendix A). This may be compared with Nielsen's heuristic on User Control and Freedom: "Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo." (Neilsen & Mack, 1994, p. 30)

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 3-1. Usability Scenario in Training Document

The Training Document for the second experimental condition, scenario-plus-general-responsibilities (S+GR), consisted of the usability scenario and a list of 19 general responsibilities that should be considered when designing any software implementation of a cancel command (Figure 3-2). This list 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 (John, Bass, Sanchez-Segura, & Adams, 2004) and vetted by the panel of architecture experts mentioned above. Since this was a list of general responsibilities designed to be considered in any system for which cancel functionality was a requirement, the S+GR Training Document stipulated that not all responsibilities might be applicable to solving any given design problem.

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.

Figure 3-2. Sample of Responsibilities of Cancel

The Training Document for the third experimental condition, scenario-plus-general-responsibilities-plus-sample-solution (S+GR+SS), included the scenario, the list of general responsibilities, and also implementation suggestions in the form of a sample solution for adding cancellation to a software architecture design based on the Model-View-Controller (MVC) architectural pattern (Sun Microsystems, 2002). The sample solution contained “before” and “after” Component Diagrams of the MVC architecture, “before” meaning a design that did not support cancellation (Figure 3-3), and “after” meaning a design that did explicitly support cancellation (Figure 3-4). A set of numbered responsibilities displayed in the components in the “before” diagram corresponded to a numbered list of the responsibilities of each of the MVC components [Appendix C, Fig. 1]. Numbered responsibilities displayed in the “after” diagram additionally allocated the general cancellation responsibilities (CRs) in the Training Document to the MVC components, and added several new components to fulfill the cancellation responsibilities [Appendix C, Fig. 2]. The sample solution also included a UML Sequence Diagram of how the revised MVC architecture worked when supporting cancellation. Although the given sample solution is not the only possible arrangement of components and responsibilities to support cancellation, the panel of eight expert architects agreed that it was a satisfactory solution.

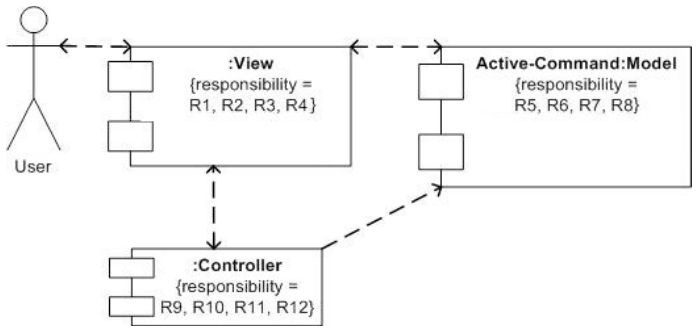


Figure 3-3. MVC Model before Adding Cancellation.

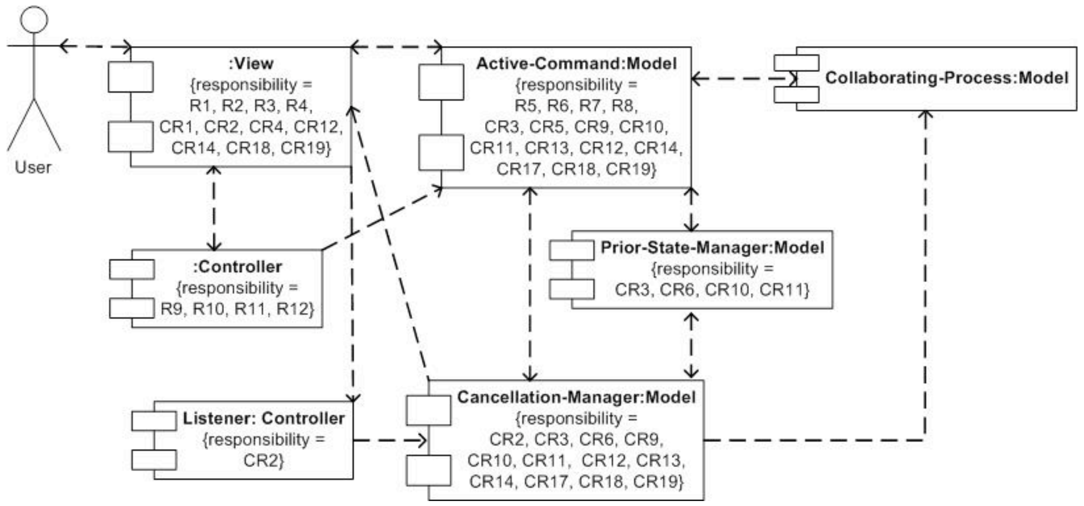


Figure 3-4. MVC Model after Adding Cancellation.

The three Training Documents varied in length accordingly as they differed in content. The S Training Document was a single paragraph, the S+GR Training Document was three pages of prose and the S+GR+SS Training Document was eight pages of both prose and software architecture diagrams.

Each participant was given the task of modifying an existing architectural design. The original design had no support for cancellation of an active command; the modified design they were asked to create was required to support cancellation. For this task we chose the architectural design for Plug-in Architecture for Mobile Devices (PAMD), a plug-in controller for the Palm OS4. PAMD had been used as a sample system in a software architecture design and analysis course at the Software Engineering Institute (Eguiluz, Govi, Kim, & Sia, 2002). This architecture design was chosen because (1) PAMD was simple enough that a participant already trained in software architecture could understand it in a relatively short period of time, (2) since the Palm OS4 was a single-threaded architecture, adding the ability to cancel a long-running command was a nontrivial task, and (3) PAMD was sufficiently different from MVC that participants who received the MVC-based sample solution had to extrapolate and generalize in order to create a specific solution for adding cancel to PAMD.

The Task Instructions included seven elements [found in Appendix F]:

- a text-based general description of the PAMD architecture;
- a text-based example scenario of how PAMD works (without considering cancellation);
- a numbered list of responsibilities of the PAMD components for normal operation;
- a numbered list of Component Interaction Steps detailing the run-time operation of PAMD while calling a plug-in;
- a Component Interaction Diagram, showing the components and connectors involved in the PAMD architecture;
- a Sequence Diagram of PAMD run-time component interaction while calling a plug-in;
- a final page instructing the participant to add the ability to cancel a plug-in to the PAMD architecture design.

For assessment purposes we designed an Answer Packet wherein the participants could easily and efficiently record their redesign. Since participants might have different levels of competency with any specific computer-based tool, the Answer Packet was paper-based. Since software architecture designs generally include architectural diagrams as well as textual information, the Answer Packet contained a Component Interaction Diagram, a Sequence Diagram, and a list of Component Interaction Steps, each with sufficient white space for the participants to insert their designs. The participants were also given several blank sheets of paper as part of the Answer Packet 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 to the diagrams were removed from the Answer Packet. Since the user's request for cancellation would always appear in the Sequence Diagram after the command was invoked, the Sequence Diagram in the Answer Packet showed execution steps up through invoking the command to be cancelled but left the following steps to the participant's discretion.

The participants were instructed to use these diagrams as the basis for their designs, and to add any components, responsibilities, or steps as needed to express their ideas for supporting cancellation. They were explicitly asked to make the diagrams in their solution consistent with each other at a detailed level, just as the diagrams in the PAMD architectural design were internally consistent.

3.1.2 Participants

Eighteen computer science graduate students participated in the original 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 (these included the 3 with no programming experience in industry). At the time of the experiment, participants reported 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.

3.1.3 Procedure

Participants were randomly assigned to one of the three experimental conditions, and participated in individual sessions with no time limit. Participants were told that they were participating in a study about fixing a usability problem in a software architecture design. They were informed that they would be given a handout to read, describing a usability scenario relevant to system architecture, then read through a description of a system architecture design, and lastly be asked to modify that architecture design to meet the requirements of the usability scenario.

Participants were then asked to read the appropriate Training Document for their experimental condition, and state when they had read and understood it. This ensured that all participants read the Training Document. 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. Participants were given the Answer Packet after the reading of the Task Instructions. Each participant was allowed unlimited time in which to complete the redesign task. After completion, the participant explained the details of the solution to the experimenter to disambiguate any handwriting or diagrammatic issues.

All of the written materials – Training Document, Task Instructions, and Answer Packet – were collected as evidence. Video recordings of each participant performing the experimental task were also collected.

3.1.4 Measures

Similarly to the method used in (Prechelt, Unger, & Schmidt, 1997; Prechelt, Unger-Lamprecht, Philippsen, & Tichy, 2002) for counting the degree to which requirements were fulfilled within subtasks of a programming maintenance task, I developed a scoring system that counted the union of all cancellation responsibilities that were found to have been considered in any elements of the participant solution. That is, I 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 indicated a narrower consideration of cancellation responsibilities; a broader consideration produced a higher number of cancellation responsibilities in the participant solution. Several measures of time on task were also collected, including time to read each of the documents and time to perform the architecture design modification task. Time spent in performing the architecture design modification task began when the participant commenced work on the Answer Packet, and ended when he/she declared completion of the design modification task. Time spent on each the documents began when the participant began reading, and ended when he/she declared completion and comprehension of the reading.

3.1.5 Results

3.1.5.1 Consideration of Responsibilities

Responsibilities considered by participants in the three experimental conditions are shown in Figure 3-5 below. Statistically significant improvement was attained for the full USAP (S+GR+SS) over scenario-alone (S) on the quantitative metric described above ($F(2,15) = 5.26$, $p < .05$). Although no other results were statistically significant, the scenario plus list (S+GR) condition trended to be better than the scenario-only condition.

Participants whose Training Document included 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. Those 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. Those who received the USAP scenario, the 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.

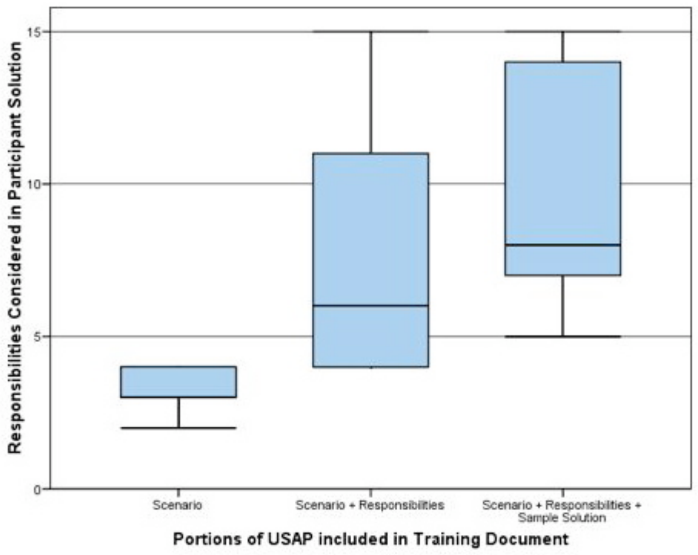


Figure 3-5. Responsibilities Considered by USAP Subset

Although no other results were statistically significant, the scenario plus list (S+GR) condition trended to be better than the scenario-only condition. Applying Pearson's correlation between groups showed no significant correlation between the number of responsibilities considered and any factor (e.g., age, gender, programming or software design experience in industry) other than the experimental condition.

3.1.5.2 Time Factors

Two measures of time were collected: time on Training Document and time on task.

Time on Training Document

The first measure, time on Training Document, measured time to use the Training Document describing the Cancel USAP, beginning when the participant started to read the version of the Training Document appropriate to the experimental condition, and ending when he/she declared that he/she has finished reading. Although, as described in Section 3.1.3, all participants also read a Task Instructions document, the experimental protocol controlled the time spent on Task Instructions so as to be the same for every participant (15 ± 2 minutes) by requiring the experiment to read the Task Instructions aloud to the participant, so time spent on Task Instructions was not an experimental measure.

Time spent reading and comprehending materials is an important factor in designing any performance aid software design. Indeed, part of the motivation for performing this experiment was to determine whether it was necessary to provide MVC examples in the Training Document, since additional material takes longer for users to learn, as well as for designers to create.

Since the three Training Documents were very different in length (one paragraph, three pages of prose, or eight pages of prose and diagrams), we hypothesized that there would be significant differences in the time the participants needed to read and comprehend the materials. Participants in the S condition took an average of 1.3 minutes to read the Training Document, with a range from 1 to 3 minutes. Participants in the S+GR condition took an average of 8 minutes, ranging from 5 to 11, while participants in the S+GR+SS condition averaged 16.33 minutes, and ranged from 7 to 29. As expected, one-way ANOVA between the three experimental conditions supported this hypothesis, showing a significant main effect of USAP subset provided on time required to read and comprehend the Training Documents ($F(2,15) = 14.95$, $p < .001$). Pairwise comparisons made using Tukey's HSD indicated significant mean difference between average time required to read and comprehend the Training Documents in the S and S+GR Training Documents ($p < 0.01$), the S+GR and the S+GR+SS conditions ($p < 0.05$) and between the S and S+GR+SS conditions ($p < 0.01$) (Figure 3-6).

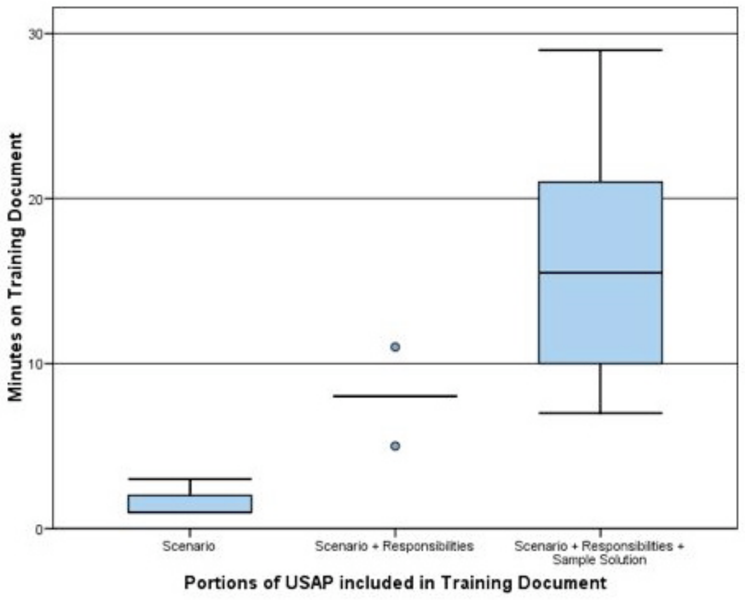


Figure 3-6. Time on Training Document by USAP Subset

Time on Task

Participants in the S condition took an average of 79.7 minutes to perform the design modification task, with a range from 47 to 124 minutes. Participants in the S+GR condition took an average of 88.2 minutes, ranging from 64 to 112, while those in the S+GR+SS condition averaged 88.8 minutes, and ranged from 39 to 138. One-way ANOVA between the three experimental conditions in the first experiment revealed no significant difference in average time required to perform the task, using the three training documents ($F(2,15) = .21$, $p < .10$) (Figure 3-7).

Summary of Experiment 1

The results of this experiment therefore showed gratifyingly significant improvement between the S and S+GR+SS conditions. There was also a strong trend toward improvement between the S and S+GR conditions, but not at a statistically significant level. However, power calculations suggested that a greater *n*, i.e., more participants in each condition, would yield statistically significant difference between the S and S+GR conditions. This would be an even more exciting result, because it would show that sufficient improvement over the usability scenario alone might be achieved through a text-based pattern in the form of a scenario and set of general responsibilities, even absent a UML-style sample solution. If sample solutions were not necessary, it would be far quicker and easier to create more new USAPs. Perhaps more importantly, it would also take the software engineers less time to understand USAPs, potentially lowering the bar to adoption in practice. Accordingly, I decided to try to find a matched set of participants with whom to replicate the experiment and try for more power with a larger *n*.

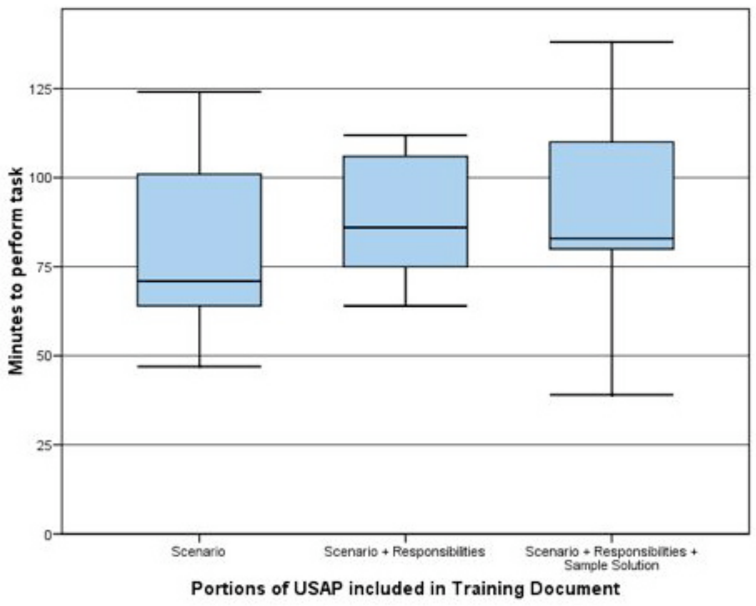


Figure 3-7. Time on Task by USAP Subset

3.2 Experiment 2 (replication)

For the replication experiment, the same experimental design and procedures materials were used as in the first experiment.

3.2.1 Participants

Fifteen students participated in our replication experiment. The 9 male and 6 female participants ranged in age from 22 to 28, with an average age of 23.4. Six averaged 8 months each of industrial programming experience, with a range from 4 to 18 months; the other 9 had no programming experience in industry. Four of the participants averaged 13.72 months of software design in industry, with a range of 3 to 18 months; the other 11 participants had no industrial experience in software design (these included the 9 with no programming experience in industry). Participants reported spending between 0 and 30 hours per week programming (average=18.33 hrs/week), and from 0 to 24 hours per week on software design (average=7.9 hrs/week).

3.2.2 Results of Replication Experiment

3.2.2.1 Consideration of Responsibilities

Using the scoring system developed for the first experiment, I measured the number of cancellation responsibilities considered in each participant's redesign. Participants in the scenario-only (S) condition considered an average of 3.8 cancellation responsibilities in their solutions, with a range from 2 to 4. Participants in the scenario plus general responsibilities (S+GR) condition considered an average of 7.8 responsibilities, with a range from 4 to 16, while participants in the full USAP (S+GR+SS) condition ranged from 6 to 14 responsibilities, with an average of 8.6. Replicating the results of the first experiment, one-way ANOVA between the three experimental conditions showed a significant main effect of the USAP subset given to participants on the number of cancellation responsibilities considered ($F(2,14) = 4.33$, $p < .05$) (Figure 3-8).

3.2.2.2 Time Factors

Time on Training Document

As in the first experiment, there were significant differences in the time the participants needed to read and comprehend the materials. Participants in the S condition took an average of 1.4 minutes to read the Training Document, with a range from 1 to 2 minutes. Participants in the S+GR condition took an average of 9.8 minutes, ranging from 8 to 13, while participants in the S+GR+SS condition averaged 20 minutes, and ranged from 11 to 36. One-way ANOVA between the three experimental conditions showed a significant main effect of USAP subset provided on time required to read and comprehend the instructional materials (F(2, 14), $p < 0.01$). Pairwise comparisons made using Tukey's HSD indicated significant mean difference between average time required to read and comprehend the Training Documents in the S+GR and the S+GR+SS conditions ($p = < 0.05$) and between the S and S+GR+SS conditions ($p < 0.01$), but the difference between time for the S and S+GR Training Documents was not statistically significant (Figure 3-9).

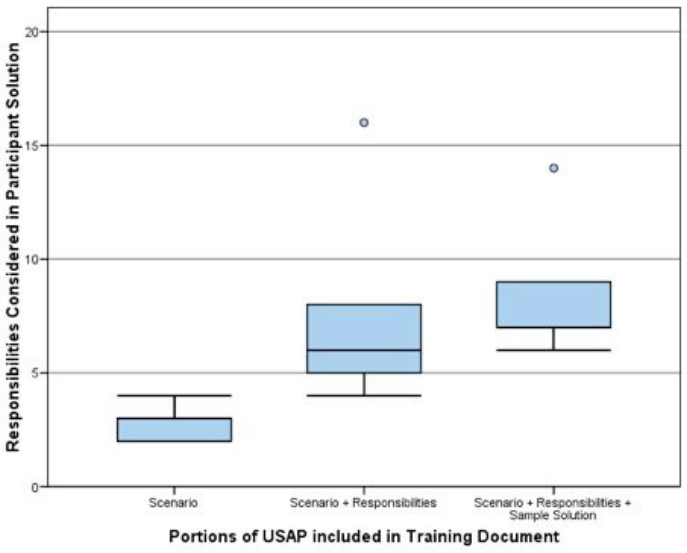


Figure 3-8. Responsibilities Considered by USAP Subset

Time on Task

In the second experiment, participants in the S condition took an average of 81.4 minutes to perform the design modification task, with a range from 51 to 118 minutes. Participants in the S+GR condition took an average of 102.4 minutes, ranging from 53 to 174, while those in the S+GR+SS condition averaged 76 minutes, and ranged from 51 to 107. As in the first experiment, one-way ANOVA between the three experimental conditions in the first experiment revealed no significant difference in average time required to perform the task, using the three training documents (F(2, 14), p = 0.44) (Figure 3-10).

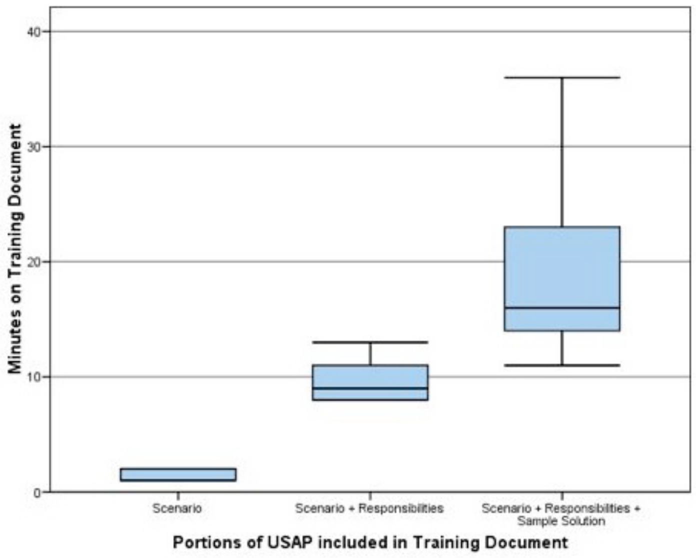


Figure 3-9. Time on Training Document by USAP Subset

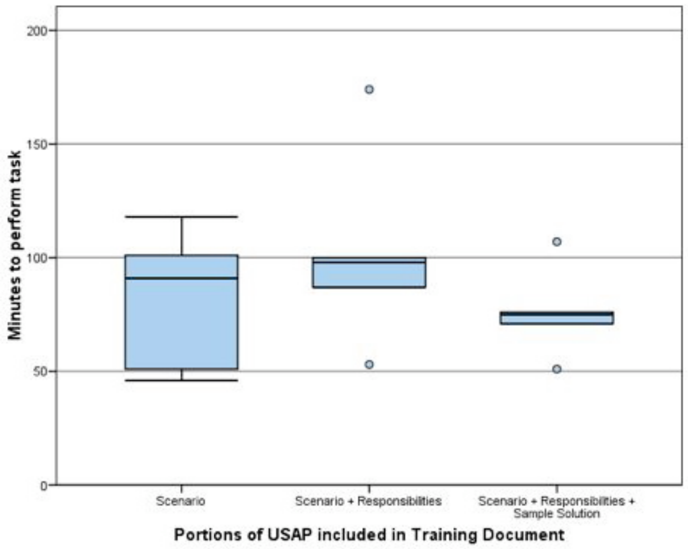


Figure 3-10. Time on Task by USAP Subset

3.3 Combining Results from Experiments 1 and 2

To investigate the contributions of each part of a USAP further, I wanted to combine the results of both experiments. The materials and procedure were identical in both instances, and the results were similar. I performed a fresh analysis of results from both experiments to determine whether combining the results was justified. Mean comparisons of participant demographics between the two experiments yielded no significant differences between groups in age or in hours per week currently spent on programming or design. Although there were statistically significant differences in industry experience in both programming (average=21 months in the first experiment v. 5 months in the second experiment) and software design (average=12 months in the first experiment v. 2 months in the second experiment), neither of these independent variables was statistically associated with performance on the design modification task and were therefore not an obstacle to combining results. There was also no significant difference between the dependent measures of responsibilities considered, time on instruction, and time on task, for each instructional condition. Thus, we were able to combine the participant groups into a single analysis, the results of which are described below.

3.3.1 Consideration of Responsibilities: Combined

In measuring consideration of responsibilities, the additional statistical power afforded by combining experiments revealed a different pattern of results than either experiment alone. As shown in Figure 11, participants in the S condition considered fewer cancellation responsibilities than either of the other conditions, averaging just 3 responsibilities, with a range from 2 to 4. Participants in the S+GR condition considered an average of 7.73 responsibilities, with a range from 4 to 16, while S+GR+SS considered an average of 9.09 responsibilities, with a range from 5 to 15. Again, one-way ANOVA between the three experimental conditions showed a significant main effect of the full USAP given to participants on the number of cancellation responsibilities considered ($F(2,32) = 10.5$, $p < .01$).

However, when analyzing the combined data, pairwise comparisons made using Tukey's HSD indicated significant mean difference between giving the scenario alone (S) and giving the list of general responsibilities (S+GR or S+GR+SS, both $p < .01$), while adding the sample solution did not significantly improve performance (i.e., the difference between S+GR and S+GR+SS was not significant) (Figure 3-11). This result suggested that future USAPs might not require sample solutions to be useful for software design, cutting down the time required to develop USAPs. It should be noted that an alternate explanation is that a single sample solution was insufficient, but that several examples might have proved helpful. However, since creating examples greatly increase the development time for USAPs, it is worth exploring whether the simpler version can prove useful first.

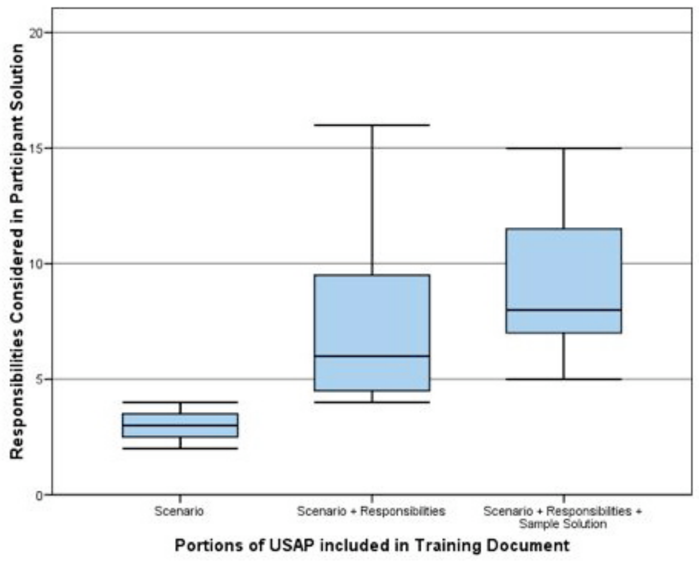


Figure 3-11. Responsibilities Considered by USAP Subset

3.3.2 Time Factors: Combined

In terms of time on task, the scenario took less time to read and comprehend (average of 1.45 min) than the scenario plus general responsibilities (average of 8.82 min), which took less time than the full USAP (S+GR+SS, average of 18.00 min) (Figure 3-12). One-way ANOVA again showed a significant main effect, but pairwise comparisons using Tukey's HSD showed that each of the conditions were significantly different at $p < 0.01$. This result makes the previous result even more interesting; not only did checklists take less time to develop than checklists plus sample solutions (saving time for researchers and USAP designers), but they also took half the time to read and comprehend, saving time for every software developer who used them. By contrast, the longer time needed to read and comprehend the full USAP might pose more of a hurdle to adoption in practice.

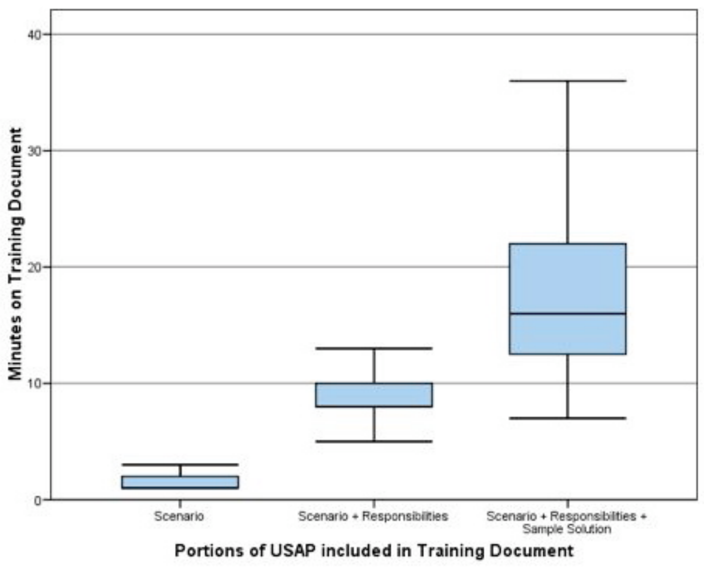


Figure 3-12. Time on Training Document by USAP subset

Participants in the S condition took an average of 80.5 minutes to perform the design modification task, with a range from 46 to 124 minutes. Participants in the S+GR condition took an average of 94.6 minutes, ranging from 53 to 174; participants in the S+GR+SS condition averaged 83 minutes, ranging from 39 to 138. As in the single experiments, one-way ANOVA between the three experimental conditions revealed no significant difference in average time to perform the design modification task, using the S, S+GR, or S+GR+SS training documents (F(2,32), $p < 0.49$) (Figure 3-13). The average time to redesign the architecture was not dependent on the training materials given to the software developers, yet those who had more complete training materials produced a better redesign. This leads one to wonder how participants used the materials they were given during the redesign task, a question that will be addressed in some detail in section 3.5 below.

In the combined experiment with a larger number of participants per condition, we could investigate beyond simple ANOVA to find patterns in performance and time on task per condition. Linear regression comparing individual time on task with individual consideration of cancellation responsibilities is shown in Figure 3-14. This analysis showed a significant association between minutes on task and number of cancellation responsibilities considered (F = 6.897, $p < 0.05$).

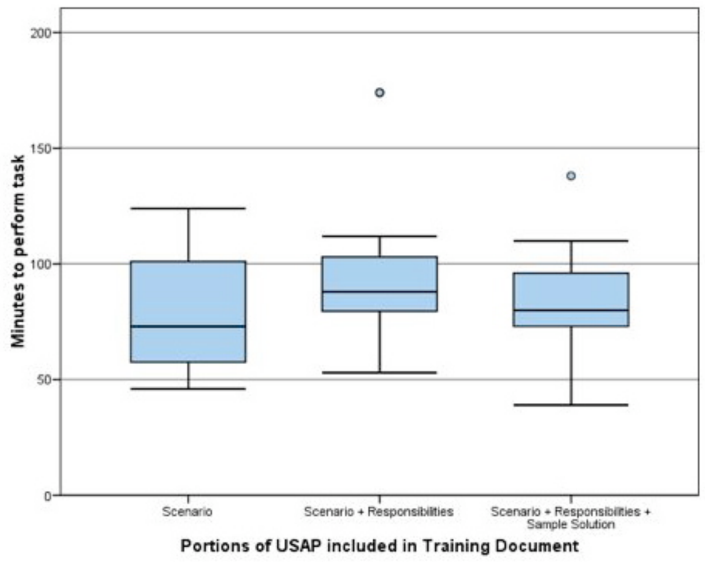


Figure 3-13. Time on Task by USAP subset

Closer examination of the association between time on task and consideration of responsibilities revealed differences between conditions, and showed that the association is only significant in the S+GR+SS condition (F = 8.98, $p < 0.05$). An R^2 value of 0.5 in this condition indicates that 50% of the overall variation in number of cancellation responsibilities considered can be explained by the time spent on the design modification task. Far less of the variation can be thus explained in the S+GR condition, and virtually none in the S condition. Figure 3-14 speaks directly to the question we hoped to investigate with our experiments and analyses: whether traditionally trained software developers can design software to satisfy usability requirements without detailed usability guidance. With only a brief usability scenario as guideline, participants considered very few cancellation responsibilities in their design solutions no matter how long they spent on the design task. On the other hand, with additional guidance, spending longer on the design task produced better performance, indicating that the additional guidance was helpful over time.

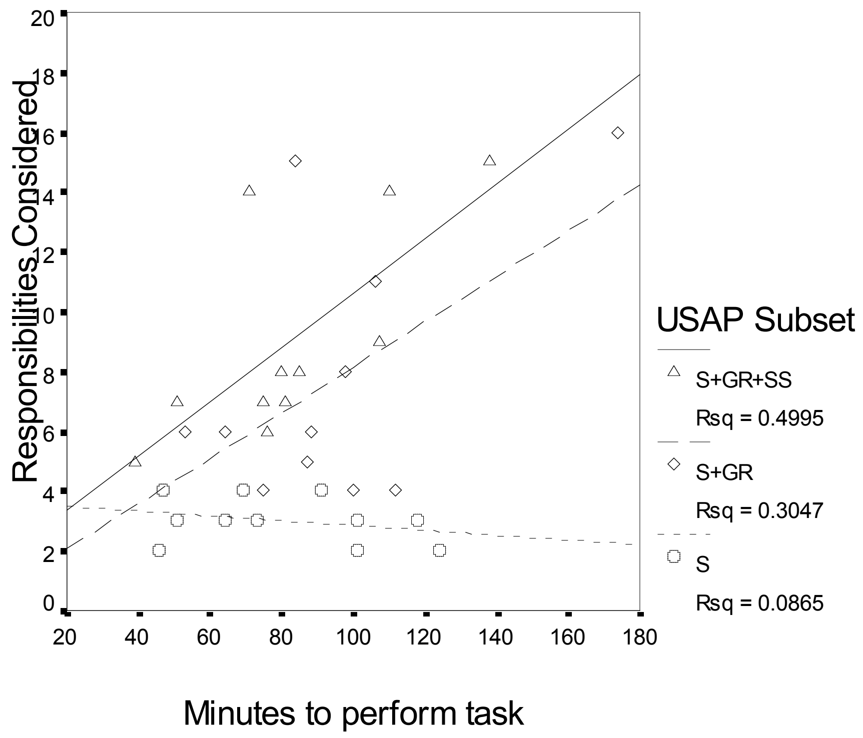


Figure 3-14. Responsibilities Considered by Time on Task

3.3.3 Not All Responsibilities Are Equally Obvious

We have seen in the preceding sections that participants in the S+GR and S+GR+SS conditions considered more responsibilities in their design solutions. It is natural to inquire as to which responsibilities most benefited from inclusion in the USAP, as not all responsibilities appear to be equally obvious. In this section we examine which responsibilities associated with the specified usability concern, canceling a command, benefited from the detail of the list-based and example-based materials. Many cancellation responsibilities were never considered in the solutions of participants who received only the general scenario, and several responsibilities were considered by fewer than 10% of participants overall. Figure 3-15 below shows a matrix of individual responsibilities considered in the solutions of all experimental participants.

Figure 3-15 shows the distribution of cancellation responsibilities considered in participant solutions. Along the horizontal axis, all the responsibilities are arrayed from those most rarely considered (on the left) to those most frequently considered (on the right). The vertical axis arrays the participant solutions from those that considered the fewest responsibilities to that considered the most responsibilities. Several interesting features are revealed by this figure.

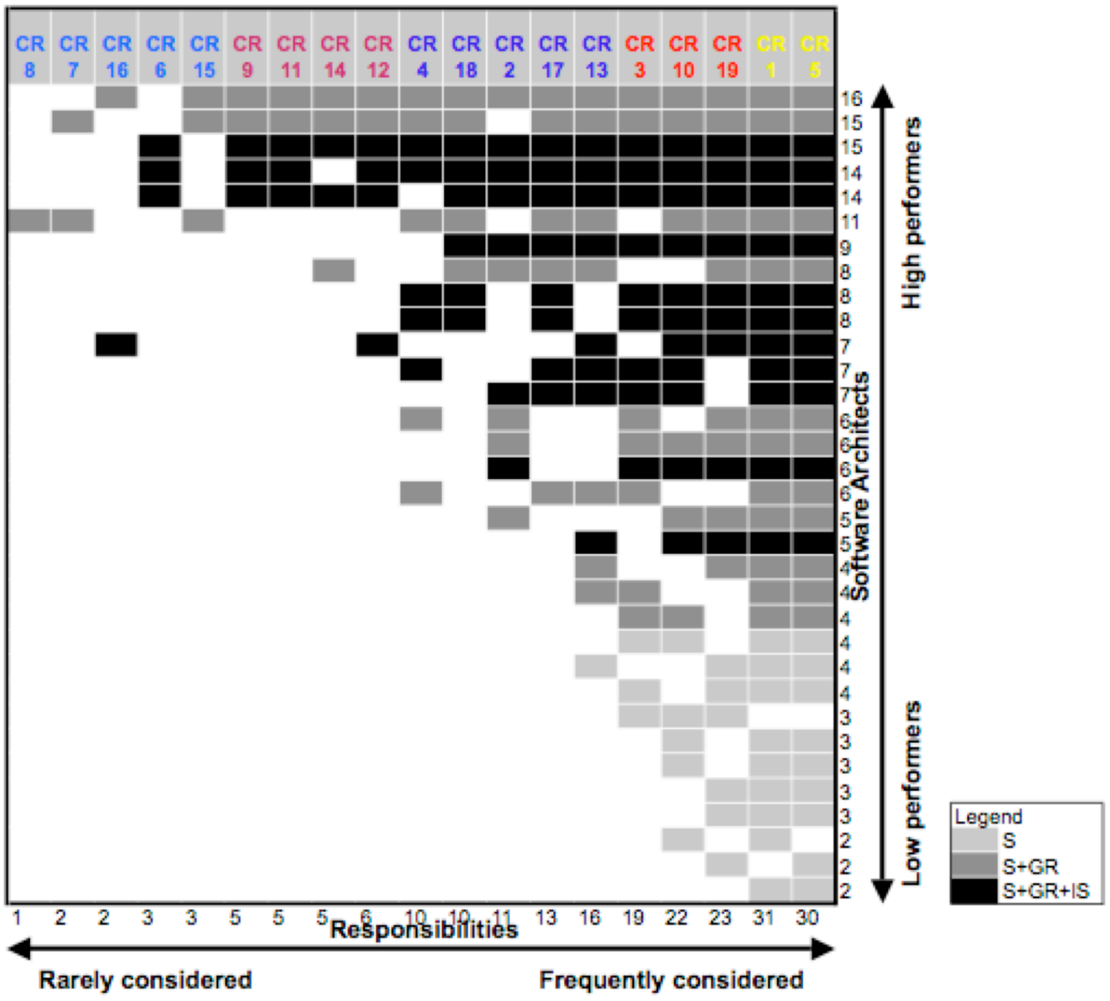


Figure 3-15. Cancellation responsibilities considered by individuals

First we will look at overall patterns in the data by experimental condition. Let us begin by looking at the participants in the S (scenario-only) condition (show in light grey). No S participant outperformed any participant in either the S+GR or S+GR+SS conditions, and no scenario-only participant identified more than four of a possible nineteen responsibilities. Furthermore, the six responsibilities identified by S participants (CR1, CR3, CR5, CR10, CR13, CR19) were also the most frequently considered by participants in the other two conditions. Since the scenario was typical of the guidance commonly provided to software engineers by usability professionals, this indicates that the additional guidance provided by the other parts of the USAP helped participants to think of responsibilities they would not otherwise have considered.

Next, observe performance in the S+GR vs. S+GR+SS conditions. Several S+GR participants outperformed S+GR+SS participants; in fact, two of the three highest performers had only the S+GR Training Document. So the sample solution is no more helpful across the board than not having one. There is also a correspondence in four of the five least-considered responsibilities (considered by fewer than 10% of participants) between these two experimental conditions; we address this below.

In terms of which responsibilities were addressed most or least frequently, only two responsibilities were identified by more than 90% of participants. These responsibilities (CR1, CR5, labeled at the top in yellow) addressed providing a means for a user to initiate a cancellation (e.g., a button), and providing a means within the software for cancellation to occur when invoked (e.g., a method). [The complete text of all the responsibilities provided in the Training Documents for the S+GR and S+GR+SS conditions may be found in Appendix B.] These two responsibilities were fairly simple to address from a software architecture standpoint since each involved only a single component in the architecture, which may explain why they were apparently obvious enough to be identified by the majority of participants without the aid of lists of explicit cancellation responsibilities.

Three responsibilities (CR3, CR10, CR19, labeled at the top in red) were identified by 50-70% of participants. Five responsibilities (CR2, CR4, CR13, CR17, CR18, labeled at the top in dark blue) were identified by only 25-50% of participants. As with the two most commonly identified responsibilities, a solution to address CR4 involved only one component in the software architecture. However, CR4 also involved knowing how quickly a human user would perceive visual feedback; to identify this independently would have required knowledge which software engineers might not have, but which usability specialists do have. Four responsibilities (CR9, CR11, CR12, CR14, labeled at the top in maroon) were identified by merely 10-25% of participants. It is worth noting that all nine responsibilities considered by 25% or fewer of participants were phrased as conditionals, which the participants may therefore have regarded as less important than responsibilities phrased as always required.

Five responsibilities (CR6, CR7, CR8, CR15, CR16, labeled at the top in light blue) were identified by fewer than 10% of participants. Among these five responsibilities there are clear differences between the S+GR and S+GR+SS conditions, and most of these may be attributable to features of their respective Training Documents. One of these responsibilities (CR6) was only considered by participants in the S+GR+SS condition. Three of these responsibilities (CR7, CR8, CR15) were only considered by participants in the S+GR condition. The Training Documents for both conditions contained the text of these responsibilities, but only the S+GR+SS condition had the sample solution, in which responsibilities were assigned to components of the sample solution; in the sample solution, these three responsibilities were, for various reasons explained in the solution, deliberately not assigned to any components. Participants who had the sample solution to examine while creating their own solutions may have transferred the assignment of responsibilities to their solutions more literally than I had hoped, despite my deliberate choice of different types of software systems for the sample solution and the experimental task.

3.4 Analysis of Quality – Experiment 1

Up to this point, we have focused entirely on the consideration of responsibilities in participant solutions. However, in addition to counting the responsibilities considered in a redesign, the quality of the resulting software architecture design was also of concern. There were two good reasons to evaluate quality. First, consideration of responsibilities alone was not useful unless using USAPs also helped software architects to produce a good architectural solution. Second, expert evaluators are expensive and difficult to coordinate, to say nothing of getting experts to agree on an opinion. I therefore wanted to see whether coverage of responsibilities could be reasonably used as a discount method for evaluating quality of solution. The analysis presented in this section was performed between the first and second experiments.

3.4.1 Measures

I arranged for additional analysis of the participant solutions from the first experiment to assess the quality of the software architectures in the solutions. This assessment allowed me to examine correlations between the cancellation responsibilities considered and the overall quality of the software architecture design. The qualitative analysis took place before the replication experiment, so the results only reflect the quality of the participant solutions in the first experiment.

The same panel of eight software architecture experts who had reviewed the experimental materials and canonical solution prior to the experiment evaluated the quality of the participant solutions. The canonical solution was not used as a definitive "answer sheet" by the evaluators, although the process of creating it had assured that the evaluators were calibrated as to what constituted a reasonable solution. Participant solutions were anonymized to normalize handwriting and drawing issues, and to remove any indication as to which Training Document the participant might have received, and then randomly distributed to these evaluators. Evaluators were asked to rate the overall quality of each participant solution on a scale from 1 to 7, where a rating of 1 indicated that the "participant solution is a substantially poor architectural solution for adding cancellation," a rating of 4 was an "adequate architectural solution", and 7 was a "substantially good architectural solution." Finally, the evaluators were invited to add written comments on the overall quality of the participant solution.

3.4.2 Results

3.4.2.1 Quality of Solution

One-way ANOVA showed a significant main effect of the USAP subset given to participants on the quality of the solution ($F(2,14) = 8.64$, $p < .01$). Pairwise comparisons made using Tukey's HSD indicated significant mean difference between giving the scenario alone (S) and giving the full USAP (S+GR+SS), while mean difference between other pairs of conditions was not significant (Figure 3-16). These results mirrored the results of the coverage metric in the first experiment. Paired-sample t-tests did not show any significant effect of time on task on the quality of the solution, and Pearson's correlation between groups did not find any significant correlation between the quality of the solution and any factor other than the experimental condition, such as age, gender, or industrial experience in programming or software design.

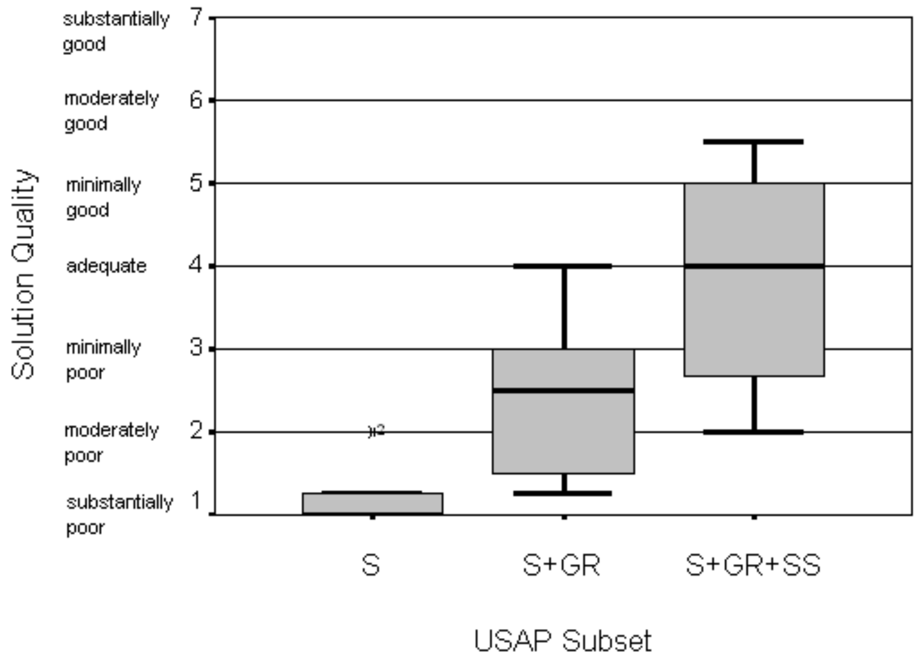


Figure 3-16. Solution Quality by USAP Subset

3.4.2.2 Comparing Quality of Solution with Consideration of Responsibilities

After assessing the quality of the participant solutions I compared the experts' judgments of solution quality with our earlier count of responsibilities considered. One goal of this analysis was to investigate convergent construct validity, that is, whether these different measures of the quality of a solution were measuring analogous effects of the Training Condition. If so, our relatively easily collected and analyzed coverage metric might be used as a discount metric for quality in future, saving the work and time of engaging and managing expert evaluators.

To compare the two measures I first adjusted the coverage metric for scalar consonance with the quality measure, using the formula [Adjusted CR Count] = [CR Count] * 7 /20. This adjustment was performed to allow easier visual comparison of the two outcome measures. Fig. 3-17 shows a side-by-side comparison of the mean solution quality and adjusted cancellation responsibilities count measures for the three different experimental conditions.

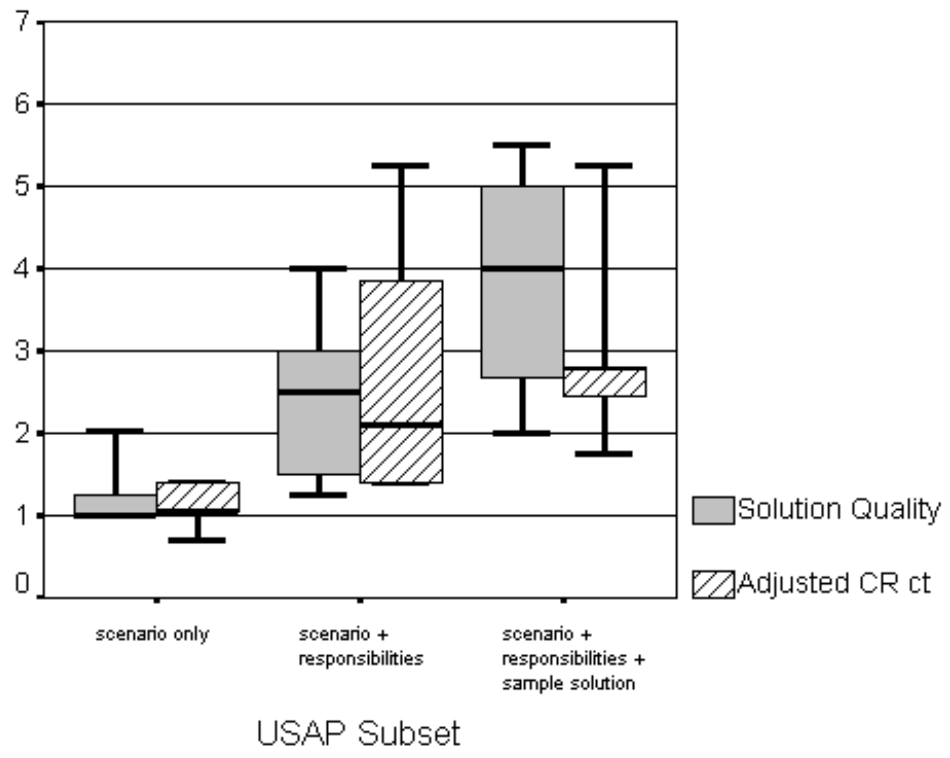


Figure 3-17. Solution Quality vs. Adjusted Count of Responsibilities by USAP Subset

The quality evaluation showed a significant positive relationship between the count of cancellation responsibilities and the overall quality of the solution. Pearson's correlation between groups found significant correlation between the number of responsibilities considered by participants and the quality of solution as rated by expert evaluators (F(1,15), r = .775, p < .01). We performed a linear regression to determine whether the solution quality measure could be predicted from our coverage metric. Linear regression of the adjusted count of responsibilities against solution quality also showed significant association between the count of responsibilities and quality of the solution (F = 22.56, p < .01). This was an important result because it had been unclear whether simple consideration of cancellation responsibilities would correlate with overall quality of solution. The results were encouraging, accounting for 60% of the variance in the quality judgments.

In an attempt to understand whether the unaccounted for variance is noise in the data or some systematic variation, we plot Predicted vs. Actual Solution Quality in Fig. 3-18 below. Each point is a single participant, ordered by condition (S, S+GR, then S+GR+SS) and by ascending actual solution quality within condition. [Only five cases are included in the third condition, as one quality rating had to be dropped due to errors in anonymizing the participant solution before submitting it to the expert evaluators.] The figure shows an apparent tendency of the model toward centrality. That is, the model more often appears to over-predict when the solution quality is low (condition S), and under-predict when the solution quality is high (condition S+GR+SS). Although we do not have sufficient data to analyze this statistically, we can hypothesize why this might have occurred. One hypothesis is that experts may value some cancellation responsibilities more highly than others in arriving at their quality ratings, while the coverage metric weights all cancellation responsibilities equally. This could result in high quality ratings for solutions that consider responsibilities that are more highly valued by experts, even when coverage is not especially high, and lower quality ratings for solutions that leave out expert-valued responsibilities. Additionally, in the S+GR+SS condition, there were two cases in which the difference between coverage and quality was markedly greater than any difference in the S or S+GR conditions.

Looking at all the evidence considered thus far, the matched pair of experiments had provided evidence that a USAP, in whole or in part, was helpful to software engineers in designing a more thoroughly considered and higher quality architectural solution, before any detailed design took place. Software architecture design is like designing a course, curriculum or assessment in that there is no "right" answer to the design problem. Results in my experiments showed that the USAP was in fact helpful at a statistically significant level in both the measures of performance we assessed: coverage of considerations and quality of solution. However, even the full USAP only improved the quality of solution from poor to average (the midpoint of the quality scale). This fell short of the desired result: improvement from poor to good (the high end of the quality scale).

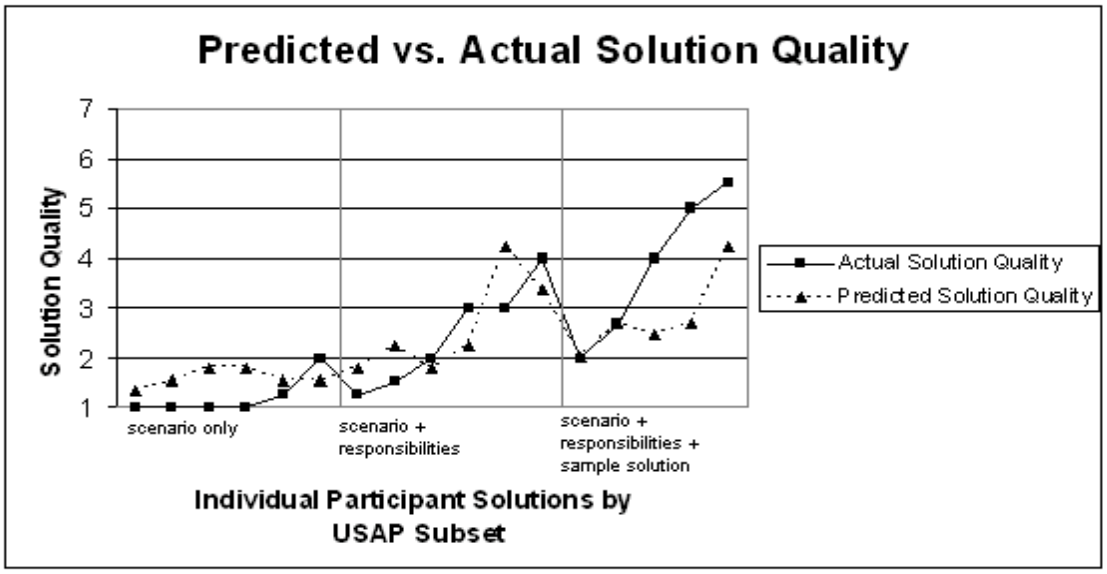


Figure 3-18. Predicted vs. Actual Solution Quality

Furthermore, measures of coverage of responsibilities seemed to be more strongly correlated with quality of solution when the participants had the text-only portion of the USAP to work with than when they had a full USAP. Further analysis was done to explore why quality did not match coverage more closely when the full USAP was provided. The experiments had provided video data of the participants using the materials describing the USAP, and it was to this data that we turned for further detailed explanation.

3.5 Exploratory Analysis of Video Data

During the first experiment, in addition to the written solutions to the software architecture design modification task performed by the participants, I had collected video of their task performance. Video data for the replication experiment was unavailable due to technical difficulties I encountered at the time of the replication. Analysis of the video data from the first experiment provides some interesting insights into possible reasons for our quantitative and qualitative results, with implications for the design and presentation of further USAPs.

In this phase, I examined video of participants in the first experiment in an attempt determine which specific parts of the original representation of the USAP were associated with high and low performance in the architecture redesign task. As described in the last section I had examined correlations between coverage of cancellation responsibilities, and the overall quality of the software architecture solution, with an eye to determining whether coverage could be used as a discount method for expert evaluations of quality (Golden et al. ISESE 2005). We had found that coverage predicted quality of solution well in the first and second experimental conditions, but not as well in the third condition, where participants received the full USAP as their Training Document. I hoped to find a better understanding of the cases where quality of solution did not correspond closely with consideration of responsibilities by examining the video of task performance by the five participants in the full USAP condition for whom we had both consideration of responsibility and quality scores (Figure 3-19).

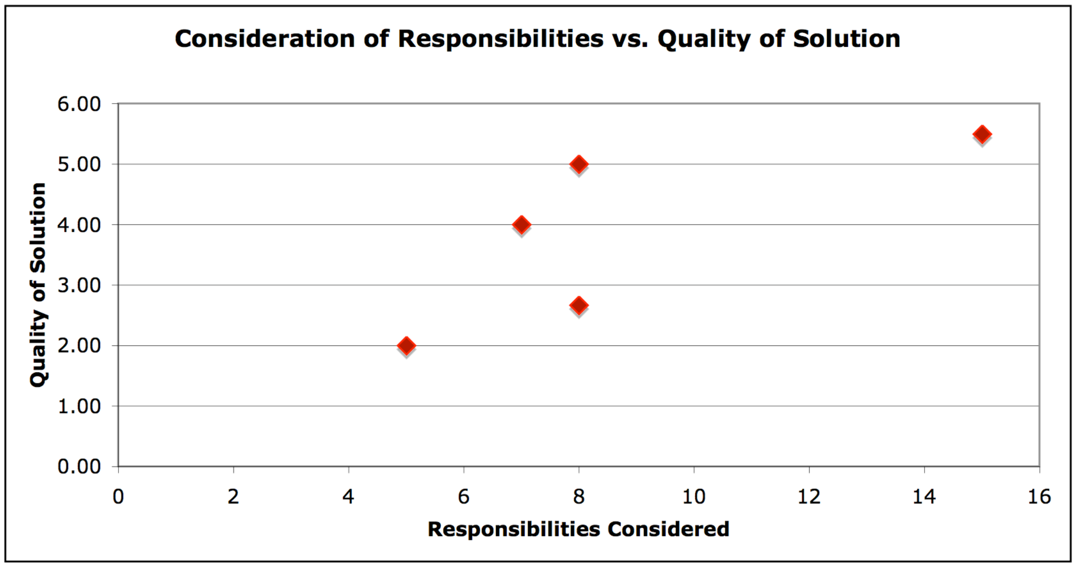


Figure 3-19. Coverage vs. Quality in video of S+GR+SS participants

It was among these participants that there was the widest range of quality scores. I was also interested in the detailed differences in usage of USAP materials between high performance and low performance. I looked at several ways in which usage of the full USAP might have affected the participants' task performance.

3.5.1 Measures of Activity Identified in Video Data Analysis

During the experiments I videotaped the table in front of each participant. Participants were requested to keep experimental materials on the table at all times. No video was taken of participants' faces. Experimental materials consisted of three multi-page printed documents, each printed on one side of the paper, and each stapled at the upper left hand corner: a seven-page set of Task Instructions, an eight-page Answer Paper (five of the pages were blank), and a Training Document which varied in length according to the experimental condition. The Training Document describing the full USAP (S+GR+SS) had eight pages. The eight pages of this Training Document [Appendix G] were divided as shown in Figure 3-20.

Page	Type	Content
1	Text	Scenario; Responsibilities (1-4a)
2	Text	Responsibilities (4b-13)
3	Text	Responsibilities (14-19)
4	Diagram and text	C&C diagram of MVC architecture without cancel; Assignment of MVC Responsibilities to MVC-without-cancel
5	Diagram and text	C&C diagram of MVC architecture with cancel; Assignment of Cancellation Responsibilities to MVC-with-cancel
6	Text	Assignment of Cancellation Responsibilities to MVC-without-cancel
7	Diagram	Sequence diagram of MVC architecture with cancel
8	Text	Description of steps in sequence diagram of MVC-with-cancel

Figure 3-20. Content of individual pages of Training Document

Participants were allowed to write on any of the documents, and to separate pages from any document if they so desired. Since I had neither video of the participants' faces nor eye-tracking data, I could not say for certain which pages they were looking at, so it is not possible to speak definitively in terms of "attention" to a given page. Instead, since the video showed the experimental materials at all times, I used the term "activity" to refer to the state of each page of each document. To glean the richest possible information from the video, I developed a coding system in which I measured each participant's use of the experimental materials at one-second intervals for the full length of the task. I observed several states of activity in which a given page might be at any point during the task. In the coding system I organized these states of activity into four categories, in ascending order of activity possibly indicating attention (Figure 3-21). Activity in this case means that the pages were visible, and the participants could see them and/or were writing on them.

State of printed or participant-written material on page	**Page active or not active**	**Possibility/likelihood of direct attention**
Not visible	Page *not* active	Direct attention not possible.
Partly or entirely visible	Page active	Direct attention possible.
Partly or entirely visible, and the participant's hand is hovering over or resting on some portion of the page	Page active	Direct attention possible, perhaps more likely than if no corresponding hand activity.
Partly or entirely visible, and the participant is writing or drawing on the page	Page active	Direct attention possible and almost a certainty.

Figure 3-21. Levels of page activity in Training Document

3.5.2 Results

As I will detail below, results showed that the graphical representation portions of the first embodiment of the USAP did not appear to improve performance in the design task, but several textual portions of the USAP may have done so.

3.5.2.1 Significance of Different Individual Pages of the Training Document

Because there was high variance between participants as to how long they had taken to complete the task, I framed the data analysis in terms of percentage of task time spent on various activities, rather than absolute time.

Because I was interested in which parts of the full USAP might have affected task performance, I looked at how usage of different pages of the Training Document, such as text describing responsibilities, or diagrams of the sample solution, was related to differences in coverage of responsibilities and/or quality of solution. Figure 3-22 shows how task performance corresponded with activity in individual pages in the Training Document for the S+GR+SS participants.

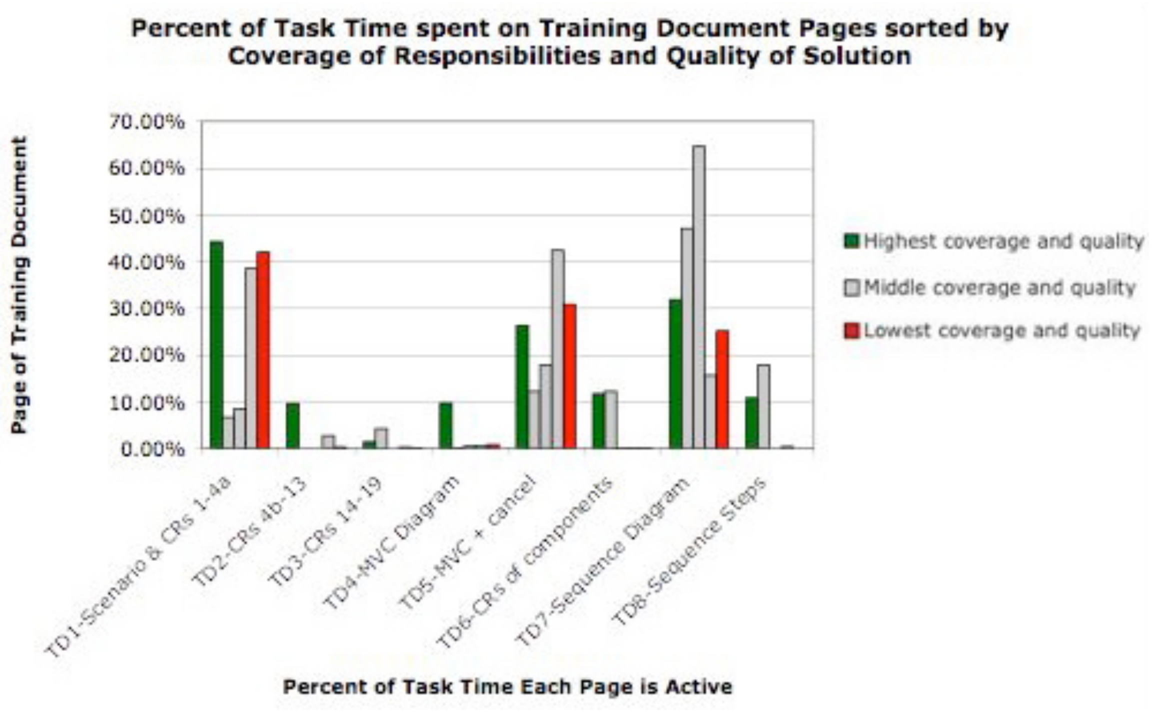


Figure 3-22. Task Time on Training Document pages by Participants

Several features emerged from examining this data. First, the highest performer in both coverage of responsibilities and quality of solution spent more equal task time on each of the different pages of the Training Document than any other participant. By contrast, the lowest performer in both coverage of responsibilities and quality of solution used only three pages of the Training document: the first page, containing the scenario and a handful of responsibilities, and the two diagrams. This suggested that skipping text-based portions of the USAP might severely reduce the usefulness of the USAP.

Second, all five participants spent task time on the diagrams. This indicated that if we provided diagrams as part of the USAPs, people would look at them. However, more task time spent looking at the diagrams did not correspond with better task performance. Taken together with the additional time required before the task to read and comprehend the Training Document that included the diagrammatic sample solution, this suggested that not only might including the sample solution in a USAP be a hurdle to adoption for time-pressured professionals, it might not benefit them in their work if they paid more attention to the diagrams than to the accompanying text!

3.6 Motivation for Next Steps

Using a full USAP increased the quality of solution that participants created 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 create a solution of significantly higher quality, as well as 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. In the combined results of the first and second experiments, the coverage metric showed significant improvement in using the list of responsibilities over the scenario alone. Since the quality and coverage metrics were well aligned in the first experiment, it may reasonably be posited that quality for the combined experiments would also have shown significant improvement for the list of responsibilities over the scenario alone. Taken altogether, the USAP for canceling a command could already be considered a valuable tool for modifying software architecture designs to address a specific usability concern.

More work was clearly necessary, however, since the USAP brought these participants up to an average quality of "adequate," the midpoint of the Likert scale the expert evaluators used to measure quality. This result fell short of the desired goal: to bring the average software designer up to the high end of the quality scale. In particular, the high variability in quality of solutions produced using the full USAP indicated that the sample solutions might benefit from an improved format, and possibly an improved delivery mechanism.

Several design tradeoffs were involved in deciding whether to include diagrammatic sample solutions in the next version of USAPs. The increased quality of design solutions had to be weighed against the increased time to read and understand the Training Document. Although there were fewer expert evaluations of quality of solution than of coverage of responsibilities, correlation was shown between coverage and quality, and coverage had not been significantly improved by the inclusion of the diagrammatic portion of the USAP. Missed responsibilities in the solutions might also have been due to inattention to the text of the responsibilities in the Training Document, as seen in the video data where some participants who had diagrams devoted more of their time to looking at those, and less time to looking at the responsibilities, a division of attention that did not appear to benefit their performance.

Since text-only lists of USAP responsibilities had already produced statistically significant improvement over scenarios alone in the first phase, with quality closely correlated with coverage of responsibilities, I decided to focus more heavily on improving the text-only portion of the USAPs in the next phase, with the goal of determining whether performance could be even more improved if attention to the textual portions were enforced in a checklist style. Educational research and human factors research have both shown checklists to be useful, and educational research also speaks to the benefits of guided attention. Further, a computer-based checklist has been shown to be less error-prone than a paper-based one. Therefore I decided to develop a computer-based tool for verified checklist-style usage of USAPs, which will be described in detail in the next chapters.

3.7 Summary

In this chapter I described my work to investigate the value of paper-based Usability-Supporting Architecture Patterns (USAPs), applied without researcher intervention, as an approach to improving the consideration of specified usability concerns in software architecture design. In a pair of controlled experiments in a laboratory environment, I validated the efficacy of USAPs in a software architecture design task. Software engineering graduate students achieved significantly better results using a paper-based version of USAPs to modify a software architecture design to include specific usability concerns than they achieved using only a usability scenario. This scenario was similar to the recommendations usability experts commonly give to software engineers in professional practice. Even using only the scenario and the list of responsibilities from a USAP allowed the experimental participants to achieve significantly better consideration of responsibilities and quality of solution than they were able to achieve with the scenario alone (Golden, John, & Bass, 2005a; Golden, John, & Bass, 2005b).

Through these controlled experiments, I demonstrated that Usability-Supporting Architecture Patterns (USAPs) are a useful technique for incorporating usability concerns into the software architecture design of interactive systems. Using these patterns helped software engineers to design software architecture that considers specific usability concerns they would not otherwise notice, and to produce higher quality software architecture designs than the type of usability scenarios that are currently used in software development practice.

However, there was room for improvement. The full USAP in the controlled experiments raised the quality of solutions from the bottom to the midpoint of the quality scale, but fell short of the desired goal of raising quality to a level of excellence. Detailed analysis of the experimental results suggested directions for improvement: enforcing attention to responsibilities because better performers attended to them, and omitting diagrammatic examples since these took time to use and create and did not improve task performance.

Chapter 4. From Individual USAPs to the A-PLUS Pattern Language

In this chapter, I describe the creation of a new representation of USAPs based on the effectiveness findings of the empirical studies in Chapter 3. The identification of tactics in (Bass, John, & Kates, 2001) was an earlier attempt at identifying commonalities between USAPs. This endeavor resurfaced when commonalities discovered between several USAPs lead to the creation of Architecture Pattern Language for Usability Support (A-PLUS), a pattern language for generating responsibilities and reusing commonalities content across multiple USAPs. I describe A-PLUS in this chapter.[1] Though the creation of the pattern language lies outside the direct scope of this thesis, its structure is motivated by directly relevant in earlier chapters and provides the motivation for directly relevant work in subsequent chapters.

4.1 Revised Approach: Content and Format of USAPs

4.1.1 Content and Format of USAPs Thus Far

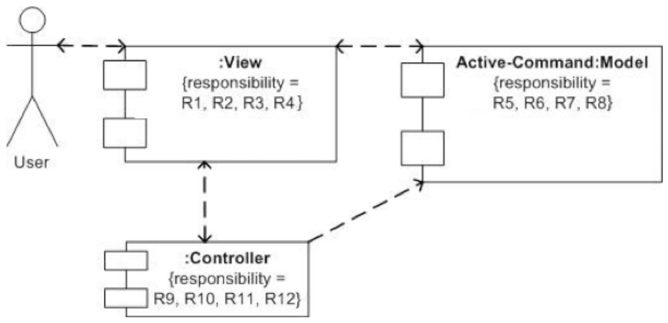
As described in Chapter 3, usability-supporting architectural patterns (USAPs) were developed as a way to explicitly connect the needs of architecturally-sensitive usability concerns to the design of software architecture (John & Bass, 2001). More than twenty patterns were originally proposed (Bass, John, & Kates, 2001). These patterns were loosely connected by software tactics that they might share (e.g., encapsulation of function, data indirection, preemptive scheduling), but the relationships between the patterns were not developed sufficiently to be of benefit in subsequent research or practice. Thus, the original USAPs were a pattern "catalogue", as opposed to a pattern language (Dearden & Finley, 2006).

[1] The material in this chapter was published in (Stoll, John, Bass, & Golden, 2008; John, Bass, Golden, & Stoll, 2009) and adapted to fit this chapter.

As originally conceived to emulate patterns in the Alexander style (Alexander, 1979), a USAP had six types of information. We illustrate the types with information from the Cancellation USAP (John, Bass, Sanchez-Segura, & Adams, 2004).

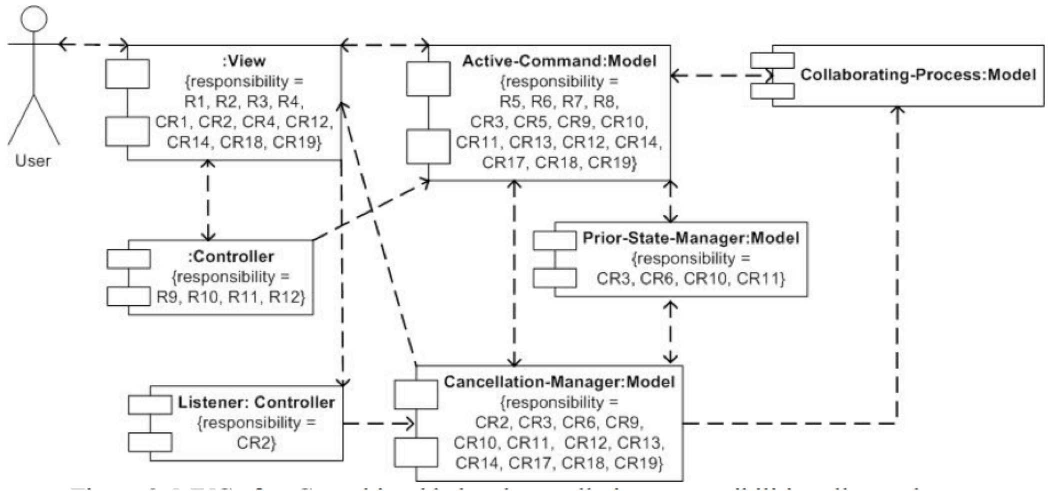
1. A brief scenario that describes the situation that the USAP is intended to solve. For example, "The user issues a command then changes his or her mind, wanting to stop the operation and return the software to its pre-operation state."
2. A description of the conditions under which the USAP is relevant. For example, "A user is working in a system where the software has long-running commands, i.e., more than one second."
3. A characterization of the user benefits from implementing the USAP. For example, "Cancel 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."
4. A description of the forces that impact the solution. For example, "No one can predict when the users will want to cancel commands"
5. An implementation-independent description of the solution, i.e., responsibilities of the software. For example, one implication of the force given above is the responsibility that "The software must always listen for the cancel command."
6. A sample solution using UML-style diagrams. These diagrams were intended to be illustrative, not prescriptive, and are, by necessity, in terms of an overarching architectural pattern (e.g., MVC). Both component and sequence diagrams were provided, showing MVC before and after incorporating the USAP's responsibilities. Responsibilities were assigned to each component and represented in the UML-style diagrams (Figures 4-1 and 4-2).

Making all six parts complete and internally consistent is a time-consuming process and for several years only one exemplar was fully completed, the Cancellation USAP for canceling long-running commands. With only one USAP fully fleshed out, there was no opportunity to discover structure that might lead to a pattern language.



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

Figure 4-1. UML-style component diagram of the MVC reference architecture before considering the ability to cancel a long running command. Original responsibilities of MVC are numbered R1, R2, etc. and refer to prose descriptions of the responsibilities accompanyi



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 4-2. UML-style component diagram of MVC revised to include the responsibilities necessary to provide the ability to cancel a long running command. New cancellation responsibilities are numbered CR1, CR2, etc., are assigned to both old and new components, and refer to prose descriptions of the responsibilities in the Cancellation USAP.

The form of USAPs used in the controlled experiments discussed in the previous chapter included three of the sections of information listed above: the general usability scenario, the list of general responsibilities, and the sample solution including UML-style diagrams applying the general scenario to an MVC architecture. However, as described in the last chapter, participants in the experiments had achieved statistically significant improvement in both consideration of responsibilities and quality of architectural solution using just the list of general responsibilities in addition to the usability scenario, and detailed examination of task performance by experimental participants had revealed no significant gain from using the sample solution diagrams, although much time was spent in using them.

4.2 Moving from Controlled Experiments to Industry Practice

Following the two controlled experiments, we began a research collaboration with ABB, a global company specializing in power and automation systems. A project team at the ABB research and development facility in Sweden was in the process of developing a new product line of process control and robotics systems. Still in the early stages of requirements gathering and system design, a project team in the ABB business unit, developing a new product line of systems, together with a ABB research team, had done a use case analysis, performed a Quality Attribute Workshop to collect non-functional requirements from prioritized scenarios (Barbacci et al., 2003) and taken steps to identify commonalities and variation points in the products they intended to bring together under a new system architecture. Thus, from the requirements collection and analysis perspective, the project team was well prepared when they began to outline the architecture.

The software architects had just begun sketching the architecture and had not yet written any code. Their implementation plan started with the backbone of the product line system, the core functionality, which would support all the variation points for the products. The project's stakeholders had prioritized usability as one of three top important software qualities for the new architecture during the Quality Attribute Workshop. One of the challenges they identified was that they would have to incorporate usability requirements into the core architecture early without having either a designed user interface, a sample user, or a complete list of future products that would be built atop the core functionality. They had thus encountered a common problem in software development: that the user interface was to be developed individually for each product and each product would use common core parts of the system. How to support usability early when the user interfaces were to be implemented last?

In seeking to address this problem, the ABB research team had investigated the work on USAPs as well as other approaches (Juristo, Moreno, & Sanchez-Segura, 2007; Juristo, Moreno, Sanchez-Segura, & Baranauskas, 2007; Folmer, van Gurp, & Bosch, 2003; Folmer, van Gurp, & Bosch, 2004) discussed in Chapter 2. After some investigation, ABB decided to try USAPs in a collaborative effort with the Usability & Software Architecture project at Carnegie Mellon University (CMU). The choice was based on the fact that USAPs use general usability scenarios and from these construct general software architecture responsibilities. The goal of the collaboration was to deliver appropriate knowledge concerning usability and software architecture to ABB's software architects in a format and at a time that would benefit their design, and in a form that could later scale to worldwide development efforts.

The collaborative work began with several teleconferences and a two-day workshop in which the Carnegie Mellon research team – Bonnie John, Len Bass, and I - presented the USAP approach and the ABB project team presented their new product line system to be developed. The ABB team explained the general domain for their product line system and presented an overview of how the product line system's products would interact with engineers, operators etc. The CMU team presented 19 general usability scenarios possibly relevant to ABB's domain [Appendix I]. These scenarios were:

1. Progress feedback
2. Warning/status/alert feedback
3. Undo
4. Canceling commands
5. User profile
6. Help
7. Command aggregation
8. Action for multiple objects
9. Workflow model
10. Different views of data
11. Keyboard shortcuts
12. Reuse of information

13. Maintaining compatibility with other systems
14. Navigating within a single view
15. Recovering from failure
16. Identity management
17. Comprehensive search
18. Supporting internationalization
19. Working at the user's pace

From these scenarios, ABB chose two for the collaboration to develop into USAPs: User Profile, and Warning/status/alert feedback, which ABB called "Alarms and Events." It was decided that the CMU team would develop the content addressing User Profile, and coach the ABB team in developing USAP content addressing warning/status/alert feedback. We quickly realized that to accommodate the needs of ABB's product line, User Profile would need to be separated into two USAPs: the normal sense of User Profile, where permissions and preferences vary for different users, and Environment Configuration, where software must accommodate different physical operating environments. The teams initially worked independently to develop content, with our team coaching the ABB team to understand the nature and level of information contained in a USAP.

Several major results began with this collaboration, which will be discussed in this chapter and the next. The reformatting of the USAP content and the development of a pattern language for USAPs will form the rest of this chapter. In Chapter 5 I will describe the design and development of a tool for the application of the new format of USAPs, and the experience of ABB software architects using the research results to evaluate a preliminary software architecture design.

4.3 Revising the Format of USAPs

While developing content for the new USAPs, we asked software architects in a different business unit at ABB for feedback on a draft of the USAP produced by the ABB researchers. The software architects appreciated the concept of USAPs, but expressed three negative reactions to aspects of the implementation.

First, software architects at ABB expressed dislike for the UML-style sample solution that formed a part of each USAP. The architects felt that they were being pressured to use the overarching pattern on which the sample solution was based, i.e., MVC. If their architecture design used a different overarching pattern, such as SOA or a pattern derived from a legacy system, the sample solution seemed to them to be an unwanted recommendation to redesign their system completely.

Second, the software architects felt that USAPs contained too many responsibilities. The first draft of the Alarms and Events USAP alone included 79 responsibilities. The software architects were apprehensive that reviewing such extensive responsibilities in three USAPs would take them far too long.

Third, the software architects questioned whether it would be practically possible to integrate three (or more) different USAPs within a single architecture design. They imagined having three different UML-style sample solutions lying on their desks and could not figure out how those would be integrated in practice.

Examining the literature in retrospect, one can find all of these criticisms anticipated in Christopher Alexander's writings about patterns, pattern languages, and their use. Despite Alexander's dictum "If you can't draw it, it isn't a pattern," (Alexander, 1979, p. 267) actually supplying a specific UML diagram violates another closely held patterns belief

> *…we have tried to write each solution in a way which imposes nothing on you. It contains only those essentials which cannot be avoided if you really want to solve the problem (Alexander, Ishikawa, & Silverstein, 1977, p. xiii).*

Although a UML diagram can be drawn and included in a pattern, doing so may require components, or arrangements of components, other than those that are essential to the pattern. Unnecessary components or arrangements may thereby be imposed on the architecture designer. A sample UML solution had been included in the original formulation of USAPs in the interest of conforming to standard pattern templates. Most pattern templates call for some sort of example or multiple examples, often expressed in diagrams.

After receiving negative user feedback from software architects at ABB, however, I reexamined the results from the laboratory experiments and found that although the participants spent a lot of time studying the UML-style diagrams it made no apparent difference in the quality of the finished architecture design. These forces (resistance from professionals and lack of clear benefit in the lab studies), led us to replace the UML sample solution with textual descriptions of implementation details that expressed structural and behavioral parts of a solution.

Therefore, UML-style diagrams like the ones in Figures 4-1 and 4-2 were not developed for the three USAPs for ABB. In place of the diagrams, each responsibility was accompanied by general implementation suggestions expressed in architecture-neutral prose. For example, in the earlier formulation of USAPs, Figure 4-2 assigns cancellation responsibility 1 (CR1) to the View component of the MVC architecture. Responsibility CR1 states, "A button, menu item, keyboard shortcut and/or other means must be provided, by which the user may cancel the active command" [Appendix B]. In the new format, without UML-style diagrams, CR1 would instead have an implementation suggestion like the following. "That part of the architecture that renders information to the user should provide a button, menu item, keyboard shortcut and/or other means by which the user may cancel the active command." These two representations are informationally equivalent, since the MVC's View is "that part of the architecture that renders information to the user." However, the prose version is architecture-neutral, since architectural styles other than MVC may furnish information to the user in a component not called View, and which is not designed to be functionally identical to MVC's View. It seemed reasonable to conjecture that an architecture-neutral expression would be more palatable to professional software architects working with a variety of architectural styles. As will be seen in Chapter 5, this conjecture seems to have been correct.

Removing the UML-style diagrams made the responsibilities the focus of the patterns, rather than the structural relationship that supports the implementation of those responsibilities. Although up until this point USAPs had been described as software architecture patterns in the flavor of (Buschmann, Meunier, Rohnert, & Sommerlad, 1996), rather than usability patterns such as in (Tidwell, 2006), it is now reasonable to consider that they may be neither, but something new altogether. Each USAP is a pattern of responsibilities; each responsibility is a pattern of implementation suggestions. The emphasis is on software responsibilities, which can also be thought of as requirements on the architecture. Solutions which accommodate these requirements may be implemented in many different ways, in keeping with the spirit of Alexander's patterns for built spaces, but patterns of responsibilities seem to be different from the more specific architecture or usability patterns typically found in the software engineering and HCI literature.

Alternatively, it is possible that only the absence of specific examples differentiates USAPs from these other types of patterns and changes the way practitioners use them. Software engineers and HCI designers have been known to use sample solutions directly and produce designs very similar in structure to those exemplars. ABB's software architects were working under constraints that prevented easy analogy from the sample solution, and the sample solution for a USAP was complex enough to take substantial time to understand. These architects preferred to examine the information at the higher level of abstraction provided by the responsibilities and general implementation suggestions, and then consider how to implement it only in their own context. Whatever the reasons, this format was driven by laboratory data and real-world feedback, seems to be consistent with a patterns philosophy, and seems to work well for software architects, as will be shown in Chapter 5.

In exploring this different delivery format, further examination of the textual portion of USAPs led to the development of a pattern language for USAPs. Previously USAPs had been treated as discrete scenarios with discrete sets of attendant responsibilities. The pattern language recognizes common foundational issues that apply across multiple usability scenarios. This language will be discussed in the next section.

4.4 Emergence of a Pattern Language for USAPS

In our collaboration with ABB, the goal was to aid in designing a real-world product line architecture that would support important usability concerns, i.e., we had the desire to make something. This was the first large-scale industrial application of fully developed USAPs (as opposed to the more conceptual discussions of multiple usability concerns that guided the architecture design in (Adams, Bass, & John, 2005)). In addition, for the first time, there were multiple detailed USAPs developed by different groups. Alexander predicted that this situation would foster the emergence the structure that makes languages out of patterns, i.e., a network of patterns at different scales.

As mentioned previously, developers in a different unit of ABB criticized the approach for having too many responsibilities and an unclear process for combining multiple USAPs. They could not see the patterns as leading to a whole, coherent architecture design. These criticisms were anticipated in Alexander's writings about pattern languages in his concepts of "principal components" and "compression". He warned that "*There must not be too many patterns underneath a given pattern"(Alexander, 1977, p. 320)*, giving the example that 20 or 30 would be too many, but that 5 allows the pattern to be imagined coherently. He proclaimed that to create a language that does not have too many patterns at any one level, since "it is essential to distinguish those patterns which are the principal components of any given pattern, from those which lie still further down." (Alexander, 1977, p. 321)

When we compared the three USAPs chosen by the ABB requirements team, User Profile and Environment Configuration (fleshed out by CMU) and Alarms and Events (fleshed out by ABB), we observed that both the research teams had independently grouped their responsibilities into similar categories, with a handful of elements at each level. These elements seemed to be Alexander's principal components of the level above. These elements, and their hierarchical relationships, were the beginning of a network of patterns, a language, which was further developed in the course of the project.

In addition to imposing the structure of the language, we also saw these similarities as an opportunity for what Alexander calls “compression”.

> *Every building, every room, every garden is better, when all the patterns which it needs are compressed as far as it is possible for them to be. The building will be cheaper, and the meanings in it will be deeper.*
>
> *It is essential then, once you have learned to use the language, that you pay attention to the possibility of compressing the many patterns which you have put together, in the smallest possible space (Alexander, Ishikawa, & Silverstein, pp. xliii-xliv).*

In building architecture, compression refers to fitting multiple patterns into the same space. In the context of software architecture design, compressing patterns may be related to sharing concepts, structure, and eventually code, in the design and implementation so that the system achieves the important qualities specified for the system. Just as a building will be less expensive and its meaning deeper with compression of patterns into space, we expect a software system will be less expensive to build and maintain, and its structure more deeply understood by more of the development team if the team shares concepts, structure and code.

As will be evident in the next section, the emergence of a pattern language, with multiple scales, attention to principal components, and with the concept of compression guiding our choice of format, addresses the initial criticisms posed by developers.

4.5 A-PLUS: an Architecture Pattern Language for Usability Support

As described above, the CMU research team initially developed sets of general responsibilities for the User Profile and Environment Configuration USAPs, while the

ABB team independently developed a set of general responsibilities for the Alarms & Events USAP. When we then compared the three USAPs prepared by our two different teams, we discovered common conceptual elements in all three. Both teams had independently grouped responsibilities into similar categories, with multiple elements in each grouping. This led to a restructuring of the three USAPs into a pattern language, Architecture Pattern Language for Usability Support (A-PLUS), that defines the relationships between USAPs in terms of reusable sets of responsibilities.

As mentioned previously, each of our original USAPs is a pattern of responsibilities and each responsibility is a pattern of implementation suggestions. Figure 4-3 shows a portion of language in comparison to a portion of Alexander's language. With the discovery of intermediate levels of structure, two additional concepts arose, which we call End-User USAPs and Foundational USAPs, and these are delineated in Figure 4-4. Our pattern connects to other patterns in the literature (e.g., MVC, SOA, PAC, etc.) and could be further decomposed (which is beyond the scope of this research). In short, our language is made up of the following elements.

- End-User USAPs, which are patterns of Foundation USAPs and, if necessary, a few responsibilities specific to each End-Use USAP.
- Foundational USAPs, which are patterns of responsibilities, and, occasionally, other Foundational USAPS.
- Responsibilities, which are patterns of implementation suggestions (components, communication, and behavior of the software) and, occasionally, other responsibilities.

Alexander tells us that "Each pattern, then, depends both on the smaller patters it contains, and on the larger patterns within which it is contained." (Alexander, 1979) Thus it is with the elements in our language. Foundational USAPs do not stand alone, in that the functionality they provide is not obviously related to usability issues. Since they do not touch the end-user's needs directly, their benefit to the end-user cannot be estimated, a cost-benefit analysis cannot be done, and thus, they cannot be prioritized independently from the End-User USAPs they complete. Of the original six-part representation of a

USAP described in Section 4.1.1, Foundational USAPs have only two, a description of the forces that impact the solution, and the responsibilities. To operationalize the relationship between Foundational USAPs and End-User (or other Foundational) USAPs that use them, we specify aspects of Foundational USAPs, as follows.

- Purpose: A general statement of the purpose of the Foundational USAP.
- Justification: Why for this Foundational USAP is important. (This includes the description of forces that impact the solution.)
- Parameters needed by this USAP: Parameters necessary to make the Foundational USAP specific to referring USAPs.

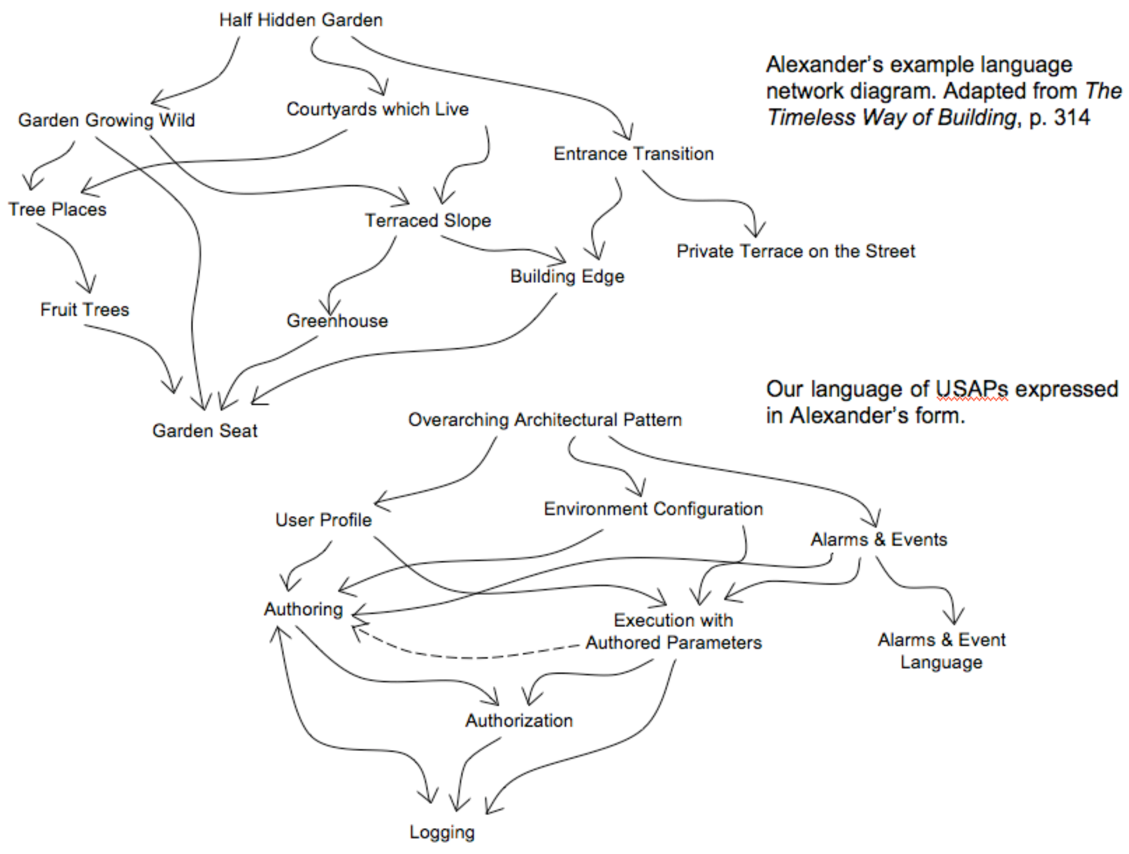

Half Hidden Garden
Alexander's example language network diagram. Adapted from *The Timeless Way of Building*, p. 314
Garden Growing Wild
Courtyards which Live
Entrance Transition
Tree Places
Terraced Slope
Private Terrace on the Street
Building Edge
Fruit Trees
Greenhouse
Garden Seat
Overarching Architectural Pattern
Our language of USAPs expressed in Alexander's form.
User Profile
Environment Configuration
Alarms & Events
Authoring
Execution with Authored Parameters
Alarms & Event Language
Authorization
Logging

Figure 4-3. Our language of three USAPs is comparable to a portion of Alexander's network diagram of his language.

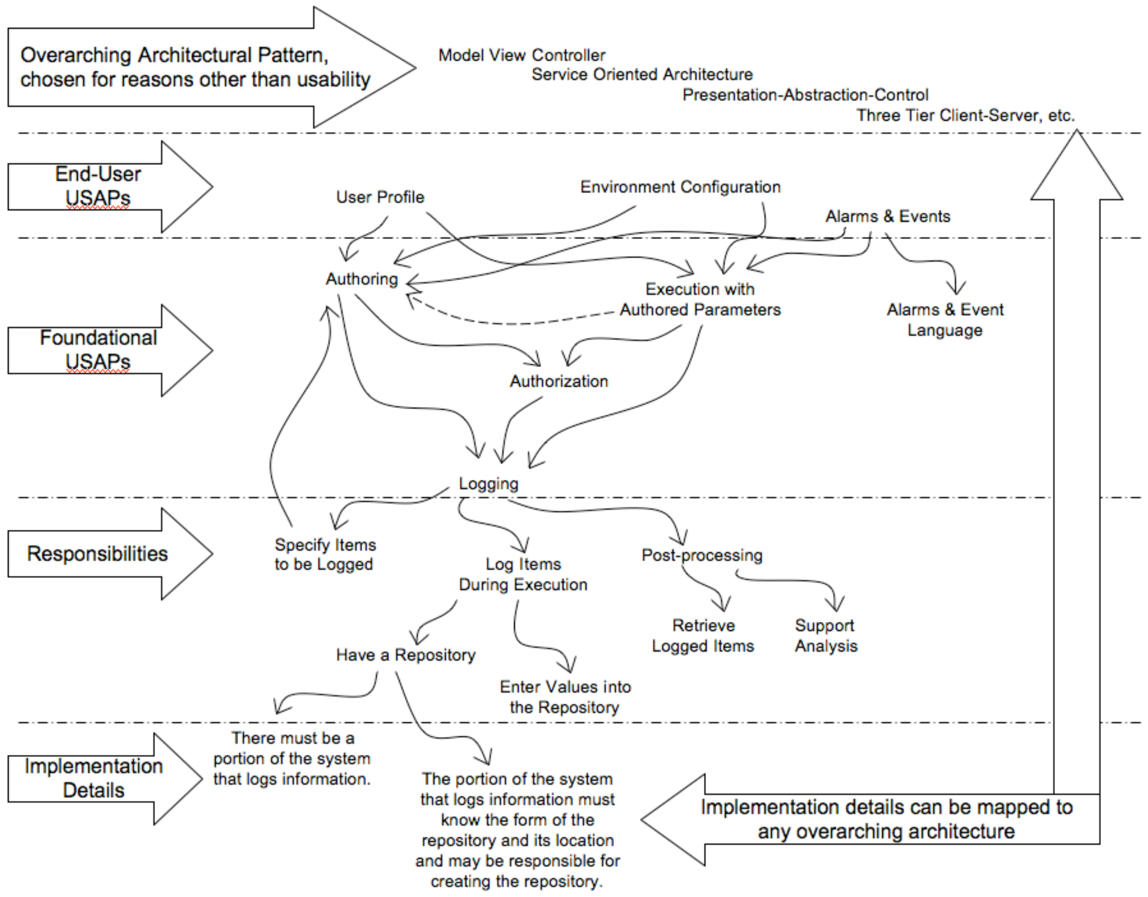

Overarching Architectural Pattern, chosen for reasons other than usability
Model View Controller
Service Oriented Architecture
Presentation-Abstraction-Control
Three Tier Client-Server, etc.
End-User USAPs
User Profile
Environment Configuration
Alarms & Events
Foundational USAPs
Authoring
Execution with Authored Parameters
Alarms & Event Language
Authorization
Logging
Responsibilities
Specify Items to be Logged
Log Items During Execution
Post-processing
Retrieve Logged Items
Support Analysis
Have a Repository
Enter Values into the Repository
Implementation Details
There must be a portion of the system that logs information.
The portion of the system that logs information must know the form of the repository and its location and may be responsible for creating the repository.
Implementation details can be mapped to any overarching architecture

Figure 4-4. The multiple scales of the A-PLUS pattern language, a partial depiction.

End-User USAPs, however, retain the properties of the original USAPs. They relate directly to usability concerns of end-users and this connection is easily summarized in a short scenario (Part 1). The conditions under which they are applicable are defined as assumptions (Part 2) and the forces from which they are derived can be enumerated (Part 4). Their benefit to the end-users can be analyzed and estimated (Part 3). As an example, these portions of a single End-User USAP for Environment Configuration are shown in Figure 4-5.

Now, however, given the pattern language, instead of having an independently-generated list of responsibilities of their own (Part 5), each End-User USAP will be completed by the Foundational USAPs it uses. However, each End-User USAP may specialize the responsibilities of the Foundational USAPs and may include additional responsibilities of its own.

Part 1	Scenario	An organization wants to supply the same software system to different hardware environments containing different collections of sensors and actuators. A configuration description of the sensors and actuators will allow the system to operate correctly in its environment.
Part 2	Conditions for applicability (assumptions)	1. There is at least one user who is authorized to author configuration descriptions. 2. The syntax and semantics for the concepts that are included in the configuration description are defined and known to the development team. 3. The protocol for saving the configuration description is defined and known to the development team. 4. Defaults exist for the specifiable parameters. 5. A template exists for authors to use when creating a new configuration description (i.e., the names and definitions of specifiable parameters and their defaults, with optional format).
Part 3	Benefits to end-users	Environment configuration prevents mistakes by tailoring the interface to present only information relevant to the current environment.
Part 4	Forces	For a software system to be configurable for different environments, actions of the system must be parameterized and the parameter values have to be available at execution time. The values of the parameters must be specified, this configuration description has to be associated with its environment, and the configuration description has to be persistent across sessions.

Figure 4-5. Non-responsibility portions of End-User USAP for Environment Configuration

For example, all of the fully-developed End-User USAPs – User Profile, Environment Configuration, and Alarms and Events – have an authoring portion and an execution portion. Specifically, a system that includes user profiles must have a way to create and maintain (i.e., author) those profiles and the mechanisms necessary to execute as specified in those profiles, a system that reports alarms and events must have a way to author the conditions under which these messages are triggered and a mechanism for displaying them. Thus, those End-User USAPs use the Authoring Foundational USAP and the Execution with Authored Parameters Foundational USAP. These Foundational USAPs, in turn, are patterns of responsibilities and may also use other Foundational USAPs (e.g., Authorization and Logging). The responsibilities in the Foundational USAPs are parameterized, so that values can be passed to them by the USAPs that use them. At the most detailed scale, implementation suggestions are patterns of components, communication and behavior that complete each responsibility. These can be realized in any overarching architectural pattern already chosen, usually for reasons other than usability.

In addition to defining the values of parameters, End-User USAPs explicitly elaborate assumptions about decisions the development team must make prior to implementing the responsibilities. For example, in the Alarms and Events End-User USAP, it is assumed that the development team will have defined the syntax and semantics for the conditions that will trigger alarms, events or alerts. This is a task of the development team on which the implementation suggestions of many of the responsibilities ultimately depend. End-User USAPs may also have additional specialized responsibilities beyond those of the Foundational USAPs they use. For example, the Alarms and Events End-User USAP has an additional responsibility that the system must have the ability to translate the names/ids of externally generated signals (e.g., from a sensor) into the defined concepts. Both the assumptions and additional responsibilities will vary among the different End-User USAPs. A complete listing of the language may be found in Appendix J.

More rigorously, there are two types of relationships between the patterns in Figure 4-3: `uses` and `depends-on`. When a USAP `uses` another one, it passes values to that USAP, which specialize the responsibilities that comprise the used USAP. If there are any conditionals in the responsibilities of the used USAP, the using USAP defines the values of those conditionals as well. The `uses` relationship is typically between End-User USAPs and Foundational USAPs. However, some Foundational USAPs `use` other Foundational USAPs under certain circumstances. There is only one `depends-on` relationship in our language, implying a temporal relationship between Authoring Foundational USAP and the Execution with Authored Parameters Foundational USAP (dashed line in the lower third of Figure 4-3), i.e., the system cannot execute with authored parameters unless those parameters have first been authored. Finally, the double-headed arrow between Authoring and Logging reflects the possibility that the items being logged might have to be authored and the possibility that the identity of the author of some items may be logged.

Foundational USAPs are completed by one or two levels of responsibilities under them. Authorization has four principal components: Identification (with five lower-level responsibilities), Authentication (with three), Permissions (with two), and Log-Off. Authoring has five principal components: Create (with four lower-level responsibilities), Save, Modify (with three), Delete (with three) and Exit the authoring system. Execution with Authored Parameters has two principal components: Access Specification (with seven lower-level responsibilities) and Use specified parameters (with two). Logging has three principal components: Specify the items to be logged, Log during execution (with two lower-level responsibilities) and Post-processing (with two). These single-digit principal components and lower-level responsibilities are in line with groups in Alexander's pattern language and compare favorably to the flat list of twenty-one responsibilities of the Cancel USAP that seemed to be too much for the experiments' participants to absorb in one sitting (John, Bass, Sanchez-Segura, & Adams, 2004).

They are also far fewer than the seventy-nine responsibilities in our first draft that elicited the negative reactions from practitioners. Since the goal of the collaboration was to support the design of a real-world architecture in an industry setting, it seemed a step in the right direction to address the practitioners' objections before asking them to use the results of our work in their project. Having developed what seemed to be a usable set of Foundational and End-User USAPs, the next step was to embody the A-PLUS language in a form that the practitioners could use in the software architecture design process. In the next chapter, I describe the design and development of a tool to deliver A-PLUS to software architects for practical use in design, and the results of user testing of the tool in professional practice.

4.6 Summary

This chapter describes the first part of research to further evolve USAPs for use by professional software architects in an industry setting, and a major representational change in form of USAPs that resulted. The research described in this chapter was performed in collaboration with corporate researchers at ABB, a global company specializing in power and automation systems. The ABB researchers were involved in plans to design a new product line architecture to integrate several existing products for which they had identified usability as a key quality attribute. Based on the research literature, the ABB researchers had selected USAPs as the most promising approach to address usability concerns early in the software architecture design phase of the product line architecture.

In addition to allowing us to investigate USAPs in an industry setting, the ABB research project was an opportunity to try using multiple USAPs in a single project. We began to develop responsibilities for each of three usability scenarios that we had not previously developed into USAPs, working in two separate teams, one at Carnegie Mellon and one at ABB. Three responsibilities resulted in lists of more than 100 responsibilities – an unwieldy number for use in practice – but on comparing the lists many commonalities became apparent.

We abstracted these commonalities to create a pattern language: Architecture Pattern Language for Usability Support (A-PLUS). Through compression the A-PLUS language allowed three USAPs to be represented in only 33 responsibilities. A-PLUS was a representational change in the description of USAPs in which the usability scenarios, each with a set of assumptions, usability benefits, and any specialized responsibilities, became End-User USAPs, while the abstractions of the commonalities, including the responsibilities and text-based implementation suggestions, became Foundational USAPs. The language combined End-User USAPs and Foundational USAPs into lists of responsibilities for software architects to review, with the capability to review the same responsibility for multiple usability scenarios to which it may apply.

The A-PLUS pattern language allows multiple USAPs with common conceptual elements to be defined in terms of reusable sets of responsibilities. Software architects can use A-PLUS to apply patterns of related responsibilities to multiple related usability scenarios. Experts in usability and software engineering can use A-PLUS to author usability scenarios and lists of responsibilities that are common to the usability scenarios. The pattern language enables compression and re-use of responsibilities to support their use in software architecture design practice. The language is couched in terms that allow for programmatic combination of the scenarios and the responsibilities to support scalability and automated integration with a software tool to embody the language. The work described in this chapter arose from collaborative work between me, Bonnie John, and Len Bass at Carnegie Mellon, and Pia Stoll, Fredrik Alfredsson, and Sara Lövemark at ABB Corporate Research, and was truly a group effort.

Chapter 5. Design for a Tool-Based Approach

5.1 Motivating the Design

In Chapter 4, I discussed the design and creation of the A-PLUS pattern language for describing USAPs. In this chapter, I describe the design and user testing of a web-based prototype for a tool, A-PLUS Architect, to deliver the USAPs created with the A-PLUS pattern language to software architects. I explain how the tool is situated in the software development process, and briefly touch on how other A-PLUS tools might be developed to support the creation and selection of more USAPs. I then describe the user tests of a prototype version of A-PLUS Architect that I performed with software architects at ABB, and the results of those tests. Finally I describe an iteration on the design of the tool, motivated by the user test results.

5.2 Revised Approach: Content and Format of USAPs

As discussed in Chapter 3, the form of USAPs used in the controlled experiments had included three sections of information: the general usability scenario, the list of general responsibilities, and the sample solution including UML-style diagrams applying the general scenario to an MVC architecture. However, participants in the experiments had achieved significant improvement in both consideration of responsibilities and quality of architectural solution using just the list of general responsibilities in addition to the usability scenario, and detailed examination of task performance by experimental participants had revealed no significant gain from using the sample solution diagrams. Furthermore, as described in Chapter 4, software designers at ABB expressed dislike for the UML-style sample solution when shown early drafts of the USAPs. They felt that they were being pressured to use the overarching pattern on which the sample solution was based, i.e., MVC. If they used another overarching pattern such as SOA or a pattern derived from a legacy system, the sample solution seemed to be an unwanted recommendation to totally redesign their system.

The use of the sample UML solution in the original formulation of USAPs was intended to conform somewhat to standard software architecture pattern templates, which typically call for some sort of graphical example or multiple examples. The fact that these templates cause resistance when dealing with existing (or partially existing) designs is interesting but investigating its cause is outside of the scope of this thesis. However this, along with my findings that the UML-style diagrams did not help participants in the laboratory experiments, led to replacement of the UML sample solution with textual descriptions of implementation details that expressed structural and behavioral parts of a solution. As will be shown later, each *general responsibility* would therefore to be delivered with implementation details, instead of delivering a single UML sample solution for the entire USAP.

5.3 A-PLUS in the Software Development Process

It became clear, once we began to reinterpret USAPs in light of the A-PLUS pattern language, that software tools might support several diverse yet related functions relative to USAPs in the software development process. A-PLUS tools were conceived as a suite of three tools, two of which lie outside the scope of this thesis. Nevertheless, a conceptual understanding of all three may explain some of the limitations placed in the design of A-PLUS Architect, which will be discussed in detail in this chapter and the next.

The toolsuite as originally conceived would include one tool for creating additional USAPs in the A-PLUS language, a second tool for choosing which USAPs to apply within a given software development project, and a third tool to assist software architects and engineers in applying the USAPs chosen through the second tool. Each of these tools would have different users and a different place in the software development process (Figure 5-1).

The A-PLUS pattern language (Chapter 4) is complex and demanding for anyone seeking to create additional USAPs. Creation of USAPs requires the collaborative efforts of experts in usability and experts in software architecture to create the content. New USAPs would likely require modifications to the pattern language as well. The process is time-consuming; the three related USAPs developed in Chapter 4 took nearly three months of effort on the part of multiple experts. However, some of that expert time was spent in developing the structure of the USAPs, making certain the parameters passed by the End-User USAPs were those expected by the corresponding Foundational USAPs, and other tasks to ensure that the resulting USAPs were complete and structurally consistent. While it was possible and relatively practical to create the initial three USAPs that way as part of developing the A-PLUS pattern language, the task of creating further USAPs would be far easier for content experts if a software tool could assist by providing a linguistic structure within which to work. Such a software tool would also check for completeness and structural consistency. As envisioned, this tool – A-PLUS Author – would allow researchers and other interested experts to spend valuable time on developing the content that should belong to any new USAP while taking care of the mechanics of creation and display.

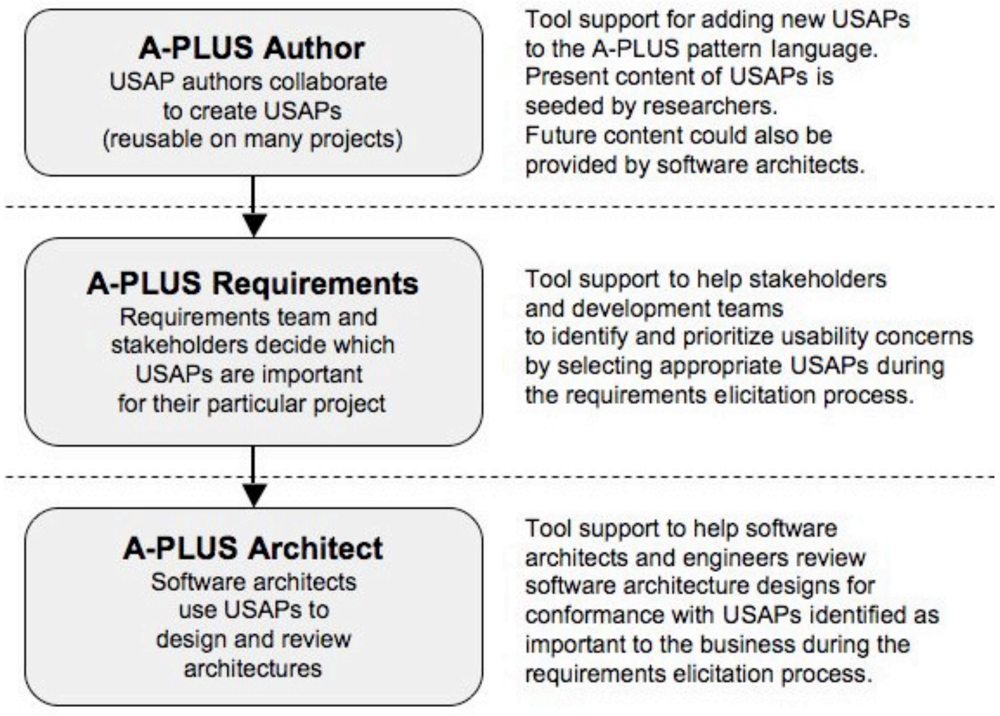

A-PLUS Author
USAP authors collaborate to create USAPs (reusable on many projects)
Tool support for adding new USAPs to the A-PLUS pattern language. Present content of USAPs is seeded by researchers. Future content could also be provided by software architects.
A-PLUS Requirements
Requirements team and stakeholders decide which USAPs are important for their particular project
Tool support to help stakeholders and development teams to identify and prioritize usability concerns by selecting appropriate USAPs during the requirements elicitation process.
A-PLUS Architect
Software architects use USAPs to design and review architectures
Tool support to help software architects and engineers review software architecture designs for conformance with USAPs identified as important to the business during the requirements elicitation process.

Figure 5-1. Software tools as conceived in A-PLUS toolsuite. A-PLUS Architect is the focus of this thesis; A-PLUS Author and A-PLUS Requirements are future work to complete the vision.

The output of A-PLUS Author would be additional USAPs, which could then be used by software architects and engineers in their design practice. However, it is not only the software architects and engineers who decide which usability scenarios are most relevant to the business needs of a product or product line. That decision belongs to a wider group of stakeholders. A second software tool is envisioned to aid and capture the process of selecting USAPs, much as ABB had selected three usability scenarios relevant to the needs of their product line. This tool would situate the selection of appropriate USAPs in the requirements-gathering phase of the software development process: A-PLUS Requirements. As envisioned, A-PLUS Requirements would record the decisions of the organization regarding their choices of USAPs for different projects, and save those choices as projects that could be imported into the third tool, A-PLUS Architect. The design and implementation of A-PLUS Requirements and A-PLUS Author are outside the scope of this thesis.

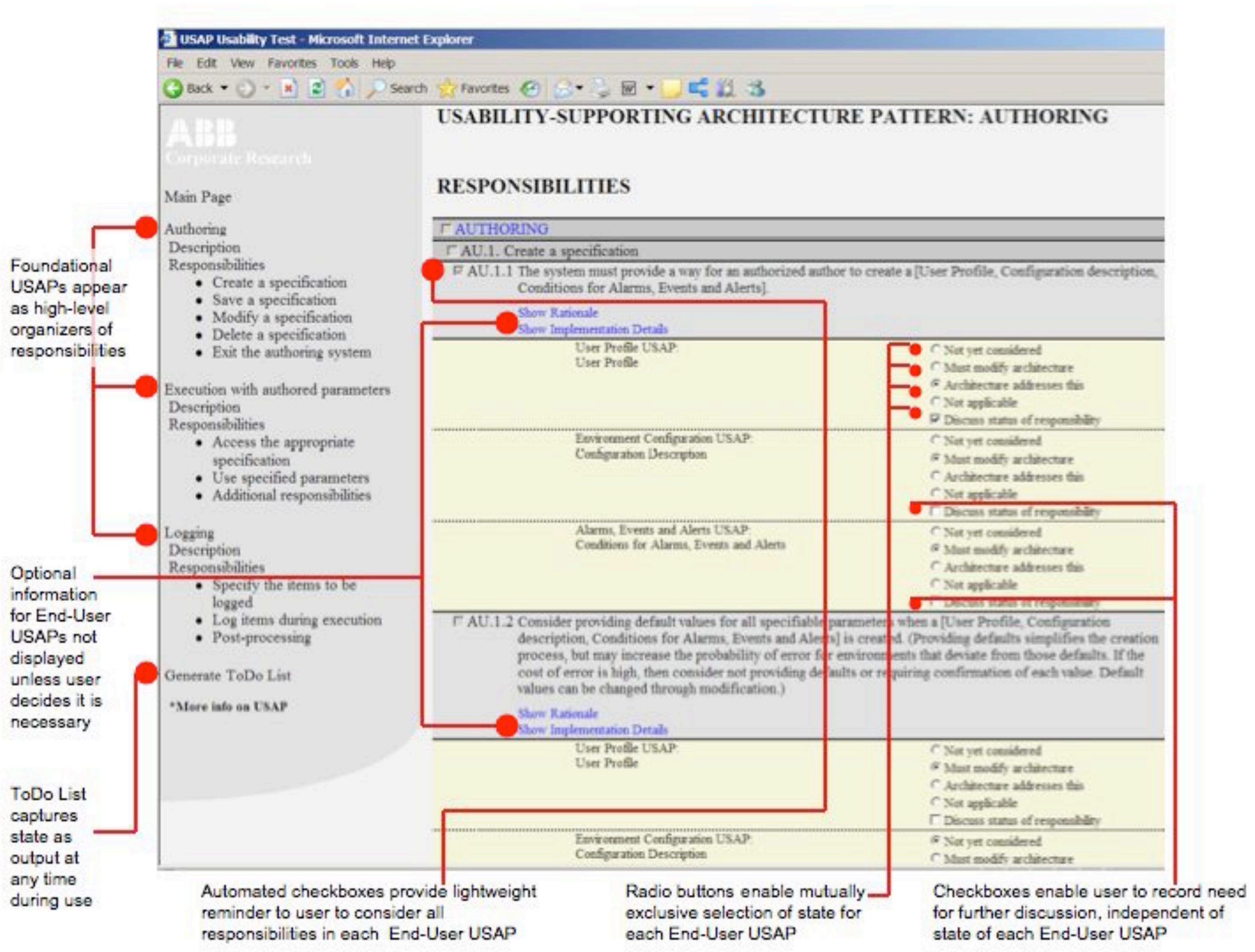
When software architects came to design the architecture of a software system with the help of A-PLUS Architect, they would simply open the appropriate project shared with A-PLUS Requirements. Since the USAPs relevant to the requirements of the project would already have been selected, opening the project in A-PLUS Architect would generate content automatically for the End-User USAPs chosen earlier in A-PLUS Requirements. For the purposes of my prototype design and implementation of A-PLUS Architect, I assumed that the three End-User USAPs for User Profile, Environment Configuration, and Alarms & Events had already been selected. This assumption underlies the A-PLUS Architect discussed in the rest of this chapter and the next.

5.4 Design and Prototype of A-PLUS Architect

Several potential roadblocks to widespread application of USAPs in software development practice were the need for software architects to have direct involvement with researchers to use USAPs in the development process, negative reactions to diagrammatic examples using a particular overarching architectural pattern (MVC), and a possibly overwhelming amount of information delivered to the software architect. Data from the empirical studies (Chapter 3) and the A-PLUS pattern language (Chapter 4) laid the groundwork for addressing these problems.

In the controlled experiments with the early form of USAPs (Chapter 3), the USAPs themselves had been delivered as a paper document. One phenomenon I had encountered in examining the results of those experiments was that many experimental participants apparently failed to consider each responsibility in the list of general responsibilities, even when they had the list available and had been instructed to consider them. Analysis of video data from the experiments, described in Chapter 3, also suggested that more attention to the lists of responsibilities might lead to better architectural solutions, both in terms of quality and of completeness. In the experiments I had not instructed participants specifically to use the lists as checklists, i.e., there was no enforcing function. Also, the lists of responsibilities in the Training Documents for the experiments did not have any explicit affordances (e.g., checkboxes) that identify them as checklists. One challenge was to encourage the designers to consider all the general responsibilities in any given USAP.

The challenge to encourage the designers to consider all responsibilities was met by transferring the USAPs into a web-based tool (Stoll et al. 2008). To encourage attention to all responsibilities in the USAP, I chose to design a web-based tool that presented responsibilities in an interactive checklist. The goals of this tool were to help the designers to actively consider all responsibilities, to be easy to use with minimal learning and with no participation by researchers, and to make the USAPs relatively easy to understand. There is also an overarching goal of USAPs which is inherent in the tool: to bridge the gap between usability requirements from a set of general usability scenarios to software architecture requirements in the form of responsibilities. I designed the tool prototype at CMU, with extensive, iterative input from the CMU team and ABB. It was implemented in HTML and Javascript by our research partner at ABB. An overview of the tool is shown in Figure 5-2; details will be described below.


USAP Usability Test - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
ABB Corporate Research
Foundational USAPs appear as high-level organizers of responsibilities
Optional information for End-User USAPs not displayed unless user decides it is necessary
ToDo List captures state as output at any time during use
Automated checkboxes provide lightweight reminder to user to consider all responsibilities in each End-User USAP
Radio buttons enable mutually exclusive selection of state for each End-User USAP
Checkboxes enable user to record need for further discussion, independent of state of each End-User USAP

Main Page

Authoring
Description
Responsibilities
- Create a specification
- Save a specification
- Modify a specification
- Delete a specification
- Exit the authoring system

Execution with authored parameters
Description
Responsibilities
- Access the appropriate specification
- Use specified parameters
- Additional responsibilities

Logging
Description
Responsibilities
- Specify the items to be logged
- Log items during execution
- Post-processing

Generate ToDo List

*More info on USAP

USABILITY-SUPPORTING ARCHITECTURE PATTERN: AUTHORING

RESPONSIBILITIES

☐ AUTHORING

☐ AU.1. Create a specification

☑ AU.1.1 The system must provide a way for an authorized author to create a [User Profile, Configuration description, Conditions for Alarms, Events and Alerts].

Show Rationale
Show Implementation Details

User Profile USAP:
User Profile
○ Not yet considered
○ Must modify architecture
● Architecture addresses this
○ Not applicable
☑ Discuss status of responsibility

Environment Configuration USAP:
Configuration Description
○ Not yet considered
● Must modify architecture
○ Architecture addresses this
○ Not applicable
☐ Discuss status of responsibility

Alarms, Events and Alerts USAP:
Conditions for Alarms, Events and Alerts
○ Not yet considered
● Must modify architecture
○ Architecture addresses this
○ Not applicable
☐ Discuss status of responsibility

☐ AU.1.2 Consider providing default values for all specifiable parameters when a [User Profile, Configuration description, Conditions for Alarms, Events and Alerts] is created. (Providing defaults simplifies the creation process, but may increase the probability of error for environments that deviate from those defaults. If the cost of error is high, then consider not providing defaults or requiring confirmation of each value. Default values can be changed through modification.)

Show Rationale
Show Implementation Details

User Profile USAP:
User Profile
○ Not yet considered
● Must modify architecture
○ Architecture addresses this
○ Not applicable
☐ Discuss status of responsibility

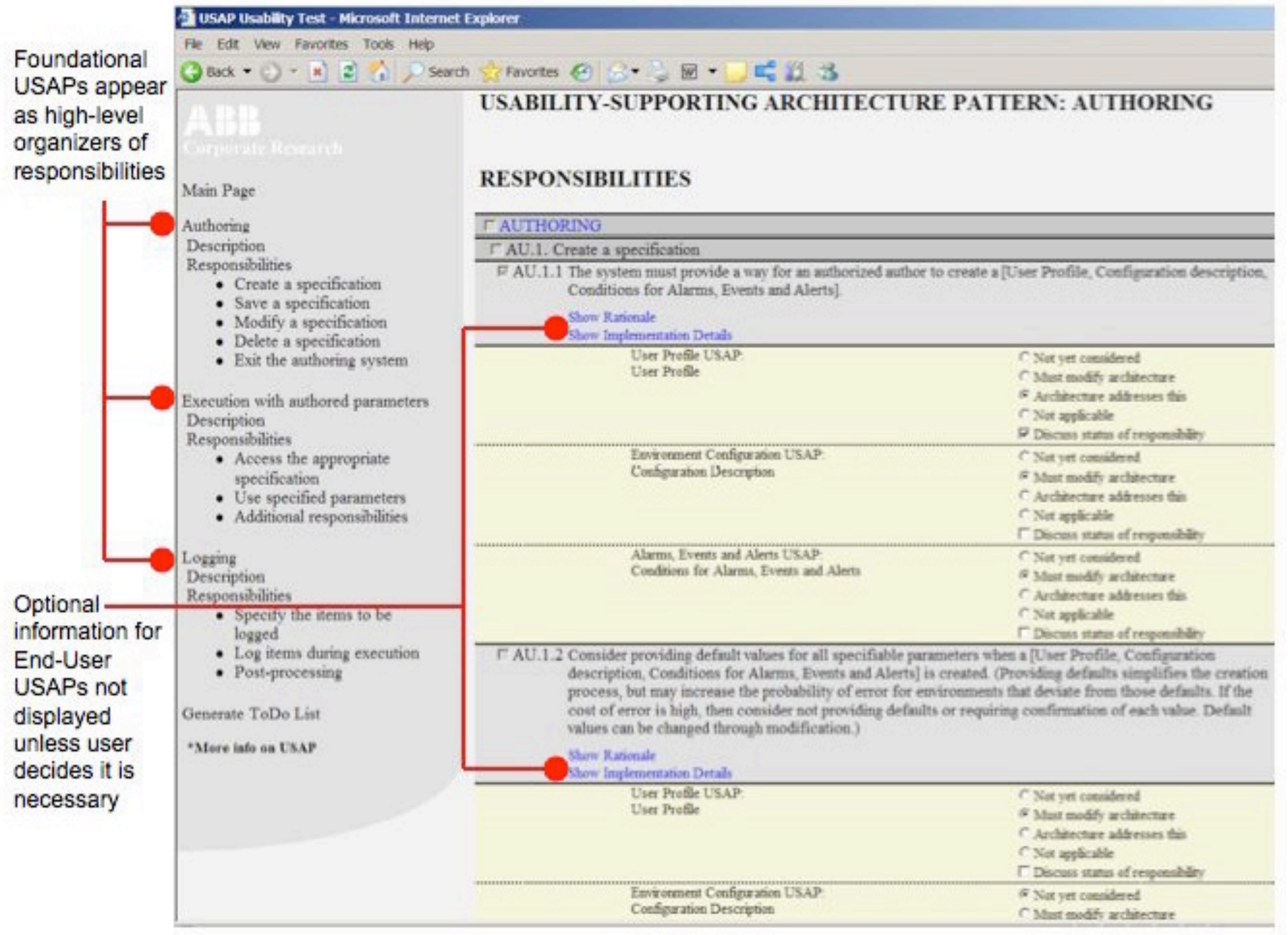
Environment Configuration USAP:
Configuration Description
● Not yet considered
○ Must modify architecture

Figure 5-2. Overview of Elements in Tool Interface

Although USAPs as described in the A-PLUS pattern language contain a great deal of information, the entire layout of the USAP delivery tool was consciously made as simple and direct as possible to support the goals of ease-of-understanding and ease-of-use. The A-PLUS language's internal concept of dividing USAPs into Foundational USAPs and End-User USAPs was hidden from the user as extraneous information that was not needed by users in order to use the tool. To support the tool's goals, the concept is instead expressed by presenting the Foundational USAPs hierarchy in the left content menu, and End-User USAPs as contextual information for considering the individual responsibilities, without using the words "Foundational" or "End-User". As Figure 5-3 shows, the responsibilities are arranged in a hierarchy that reflects the internal structure within the Foundational USAPs.

In designing the tool, I had to decide how to present optional additional information that was available for each responsibility. The A-PLUS language included rationales for the inclusion of each responsibility, and text-based implementation suggestions for each responsibility, some of which were lengthy. Each responsibility could be considered without this additional information, and it was unknown how much the software architects would use it. Additionally, I was concerned that too much additional prose on the page would hamper comprehension of the responsibilities. Since I could not precisely anticipate the needs of the software architects, I decided to allow them to choose whether to view the optional information on a by-request basis, displaying it only when the user chose to display it by clicking a link, e.g., "Show rationale" for a responsibility. The optional text could be hidden again by clicking a link, e.g., "Hide rationale" (Figure 5-3).

As well as simplifying the presentation, this design decision lent itself well to the substitution of text-based implementation suggestions for the diagrammatic examples used in the controlled experiments. I expected that the text-based implementation suggestions, presented thus, would induce fewer negative reactions than the generic example UML-style diagrams. When using the tool in-house in industry, the software architects could interpret the text-based implementation suggestions in the context of whatever architectural style they used in their project. My hope was that this would both ease their understanding of the examples and increase the possibility of re-using the implementation suggestions.


Foundational USAPs appear as high-level organizers of responsibilities
Optional information for End-User USAPs not displayed unless user decides it is necessary
USAP Usability Test - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
ABB Corporate Research

Main Page

Authoring
Description
Responsibilities
- Create a specification
- Save a specification
- Modify a specification
- Delete a specification
- Exit the authoring system

Execution with authored parameters
Description
Responsibilities
- Access the appropriate specification
- Use specified parameters
- Additional responsibilities

Logging
Description
Responsibilities
- Specify the items to be logged
- Log items during execution
- Post-processing

Generate ToDo List

*More info on USAP

USABILITY-SUPPORTING ARCHITECTURE PATTERN: AUTHORING

RESPONSIBILITIES

AUTHORING

AU.1. Create a specification

AU.1.1 The system must provide a way for an authorized author to create a [User Profile, Configuration description, Conditions for Alarms, Events and Alerts].

Show Rationale
Show Implementation Details

User Profile USAP: User Profile	Not yet considered Must modify architecture [✓ Architecture addresses this] Not applicable [✓ Discuss status of responsibility]
Environment Configuration USAP: Configuration Description	Not yet considered [✓ Must modify architecture] Architecture addresses this Not applicable Discuss status of responsibility
Alarms, Events and Alerts USAP: Conditions for Alarms, Events and Alerts	Not yet considered [✓ Must modify architecture] Architecture addresses this Not applicable Discuss status of responsibility

AU.1.2 Consider providing default values for all specifiable parameters when a [User Profile, Configuration description, Conditions for Alarms, Events and Alerts] is created. (Providing defaults simplifies the creation process, but may increase the probability of error for environments that deviate from those defaults. If the cost of error is high, then consider not providing defaults or requiring confirmation of each value. Default values can be changed through modification.)

Show Rationale
Show Implementation Details

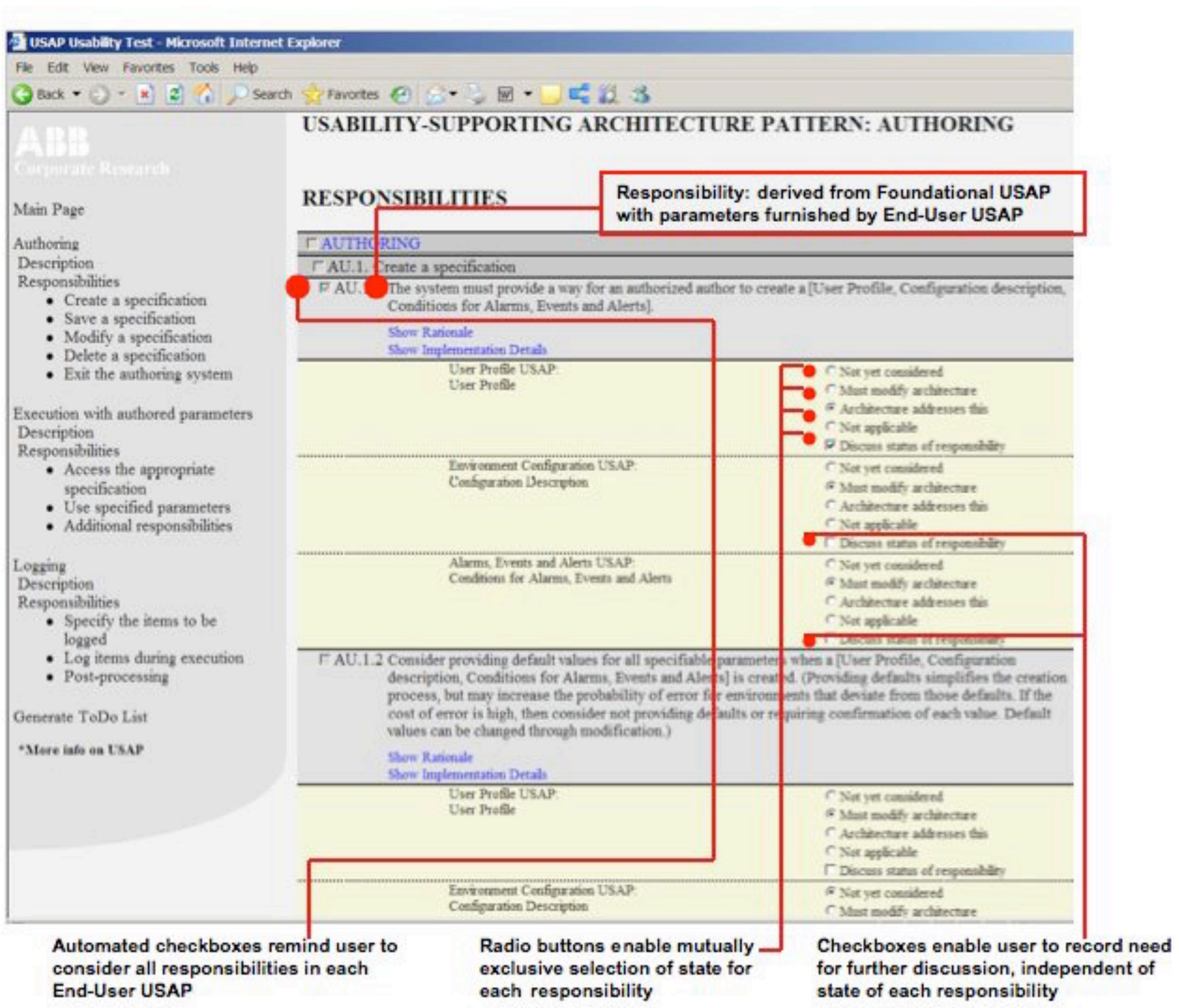
User Profile USAP: User Profile	Not yet considered [✓ Must modify architecture] Architecture addresses this Not applicable Discuss status of responsibility
Environment Configuration USAP: Configuration Description	[✓ Not yet considered] Must modify architecture

Figure 5-3. Displaying Only Necessary Information

In the main content area each Foundational USAP's responsibilities are displayed with the parameters furnished by the prioritized End-User USAPs; Alarm & Events, User Profile, and Environment Configuration. Each responsibility has a checkbox where an internal state is set to automatically "check" when the software architect has set the states of the radio-buttons of each End-User USAP related to the responsibility. The checkboxes enforce the hierarchical structure by automatically checking off a higher-level box when all its children have been checked off, and conversely, not allowing a higher-level box to be checked when any of its children remain unchecked. These checkboxes cannot be changed directly by the software architect, since that would defeat the purpose of automatically informing the architect that input is still required on responsibilities that have not yet been considered (Figure 5-4). I hoped that this reminder to consider each and every responsibility would increase the coverage of responsibilities, which is correlated with the quality of the architecture solution (Golden, John, & Bass, 2005a).

The radio-buttons are set by the software architect and reflects the architecture design's state in relation to the responsibility. The architect was intended to read each responsibility and determine whether that responsibility was already accounted for in the architecture, not applicable to the system being designed, or required the architecture to be modified to fulfill the responsibility. The choice of radio buttons reflects that the states are mutually exclusive; a responsibility cannot be in more than one of these states at a time. The default state of each responsibility is "Not yet considered," indicating that the software architect has not yet made an active choice. The other available states for a responsibility are, "Architecture addresses this", "Must modify architecture" and "Not applicable". The assumption is that the software architect will select one of these three states after reading the responsibility text thoroughly (Figure 5-4).

A checkbox that can be set by the software architect appears below each set of four radio buttons. This checkbox allows the software architect to record whether the state of the responsibility merits further discussion, irrespective of the state selected in the radio buttons. This feature arose from discussions with the ABB research team, wherein they posited a reasonable situation in which the software architects had in fact considered a responsibility, but wanted to consult with colleagues before they could confidently make a final choice as to how it was reflected in their design (Figure 5-4).



USAP Usability Test - Microsoft Internet Explorer

USABILITY-SUPPORTING ARCHITECTURE PATTERN: AUTHORING

Main Page

Authoring
Description
Responsibilities
- Create a specification
- Save a specification
- Modify a specification
- Delete a specification
- Exit the authoring system

Execution with authored parameters
Description
Responsibilities
- Access the appropriate specification
- Use specified parameters
- Additional responsibilities

Logging
Description
Responsibilities
- Specify the items to be logged
- Log items during execution
- Post-processing

Generate ToDo List

*More info on USAP

RESPONSIBILITIES

AUTHORING

AU.1. Create a specification

AU.1.1 The system must provide a way for an authorized author to create a [User Profile, Configuration description, Conditions for Alarms, Events and Alerts].

Show Rationale
Show Implementation Details

User Profile USAP: User Profile — Not yet considered; Must modify architecture; [✓ Architecture addresses this]; Not applicable; [✓ Discuss status of responsibility]

Environment Configuration USAP: Configuration Description — Not yet considered; [✓ Must modify architecture]; Architecture addresses this; Not applicable; Discuss status of responsibility

Alarms, Events and Alerts USAP: Conditions for Alarms, Events and Alerts — Not yet considered; [✓ Must modify architecture]; Architecture addresses this; Not applicable; Discuss status of responsibility

AU.1.2 Consider providing default values for all specifiable parameters when a [User Profile, Configuration description, Conditions for Alarms, Events and Alerts] is created. (Providing defaults simplifies the creation process, but may increase the probability of error for environments that deviate from those defaults. If the cost of error is high, then consider not providing defaults or requiring confirmation of each value. Default values can be changed through modification.)

Show Rationale
Show Implementation Details

User Profile USAP: User Profile — Not yet considered; [✓ Must modify architecture]; Architecture addresses this; Not applicable; Discuss status of responsibility

Environment Configuration USAP: Configuration Description — [✓ Not yet considered]; Must modify architecture

Responsibility: derived from Foundational USAP with parameters furnished by End-User USAP
Automated checkboxes remind user to consider all responsibilities in each End-User USAP
Radio buttons enable mutually exclusive selection of state for each responsibility
Checkboxes enable user to record need for further discussion, independent of state of each responsibility

Figure 5-4. Indicating States of Responsibilities in End-User USAPs

A separate main page described the concept of USAPs, and provided instructions on how to use the USAP delivery tool [Appendix L]. The states of the radio-buttons and checkboxes were persistent as long as the web-tool remained open, enabling the user to go back and forth in the tool without losing data.

The output of the prototype was a "to do" List. This is a list of the responsibilities that either have not yet been considered, that require a modification of the architecture, or that need to be discussed further. The "to do" list appeared on a page in which each responsibility appearing on the "to do" list was accompanied by an editable textbox in which the user could enter notes before printing the list. In the prototype, the "to do" list could not be saved, persisting only so long as the browser session continued. This required the user to remember to print the list before ending the browser session. However, so long as the browser session continued, the states of the responsibilities could be changed as many times as the user desired, and those changes were reflected in the "to do" list. A link allowed the software architects to generate and print the "to do" list as at any point during the browser session. Since the delivery tool was a prototype we did not take it to the level of a full-fledged content management tool with a database as the backbone. We waited for user tests to indicate to us if this would be a good direction for next steps before expending the effort.

Before proceeding to the user tests, however, I will discuss in detail how the design of A-PLUS Architect addressed the problem of combining multiple USAPs in a single presentation format.

5.4.1 Combining Multiple USAPs in a Single Presentation Format

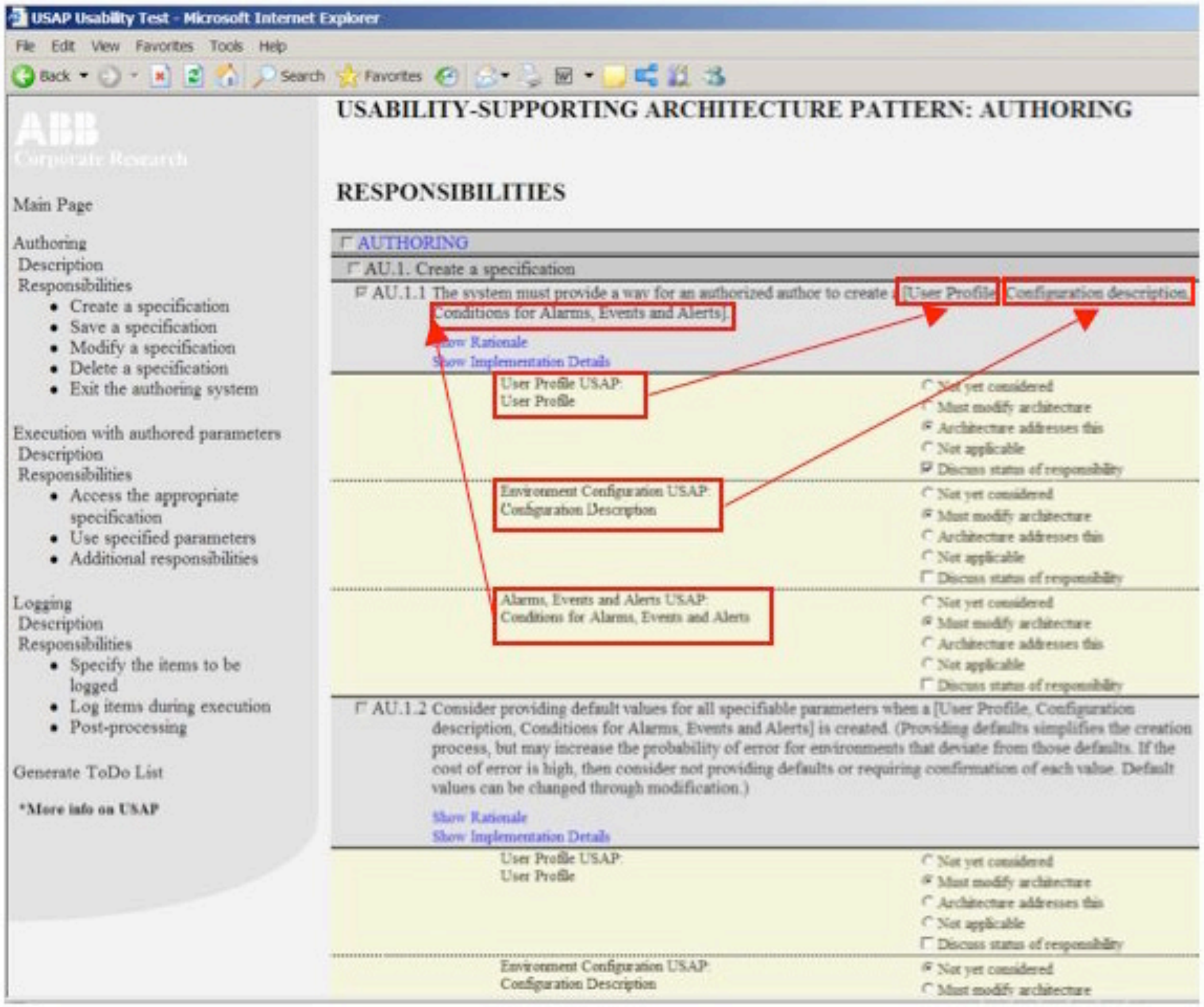
One motivation for developing both the A-PLUS pattern language and the A-PLUS Architect tool was that the ABB software architects could not imagine how they might incorporate multiple USAPs in the same architectural design. Another motivation was the opportunity to reuse the principles underlying a single USAP to develop multiple related USAPs, and the consequent ability to show software architects similar opportunities in their design solutions. The solution was to simplify the delivery of USAPs when multiple USAPs were deemed relevant to a particular system.

Recall that the Foundational USAPs are parameterized and each End User USAP provides a string that is used to replace the parameter. For instance, consider a responsibility from the Authoring Foundational USAP: “The system must provide a way for an authorized user to create a SPECIFICATION.” Accordingly, when three End User USAPs are relevant to the system under design, such as User Profile, Alarms & Events, and Environment Configuration, the three responsibilities would be displayed to the user as “The system must provide a way for an authorized user to create a [User Profile, Alarm, Event & Display rules, and Configuration description] specification.

This presentation satisfied two goals and introduced one problem. Presenting three responsibilities as one reduced the amount of information displayed to the architect since every Foundational USAP responsibility is displayed only once, albeit with multiple pieces of information. This presentation also indicated to the architect the similarity of these three responsibilities and the reuse possibilities of fulfilling them through a single piece of parameterized code.

The problem introduced by this form of the presentation was that the radio buttons then became ambiguous. Did the entry “architecture addresses this” mean that all of the three responsibilities had been addressed or that only some of them had been? I resolved this ambiguity by repeating the radio buttons three times, once for each occurrence of the responsibility. Thus, the three responsibilities were combined into one textual description of the responsibility but three occurrences of the radio buttons. This is shown in Figure 5-5, which illustrates the presentation of responsibilities for multiple related USAPs in the web-based tool prototype.

The name of each of the three USAPs chosen for delivery was enumerated under each responsibility. The three sets of radio buttons indicated that the software architect must respond to each responsibility in the context of each USAP. This was designed to encourage the user to consider each responsibility's applicability for each individual USAP, while allowing them to consider multiple USAPs simultaneously with a minimum of effort in reaching an understanding of any given responsibility.



USAP Usability Test - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
ABB Corporate Research

Main Page

Authoring
Description
Responsibilities
- Create a specification
- Save a specification
- Modify a specification
- Delete a specification
- Exit the authoring system

Execution with authored parameters
Description
Responsibilities
- Access the appropriate specification
- Use specified parameters
- Additional responsibilities

Logging
Description
Responsibilities
- Specify the items to be logged
- Log items during execution
- Post-processing

Generate ToDo List

*More info on USAP

USABILITY-SUPPORTING ARCHITECTURE PATTERN: AUTHORING

RESPONSIBILITIES

☐ AUTHORING

☐ AU.1. Create a specification

☑ AU.1.1 The system must provide a way for an authorized author to create a [User Profile, Configuration description, Conditions for Alarms, Events and Alerts].
Show Rationale
Show Implementation Details

User Profile USAP: User Profile	Not yet considered [✓ Must modify architecture] Architecture addresses this Not applicable; ☑ Discuss status of responsibility
Environment Configuration USAP: Configuration Description	Not yet considered [✓ Must modify architecture] Architecture addresses this Not applicable; ☐ Discuss status of responsibility
Alarms, Events and Alerts USAP: Conditions for Alarms, Events and Alerts	Not yet considered [✓ Must modify architecture] Architecture addresses this Not applicable; ☐ Discuss status of responsibility

☐ AU.1.2 Consider providing default values for all specifiable parameters when a [User Profile, Configuration description, Conditions for Alarms, Events and Alerts] is created. (Providing defaults simplifies the creation process, but may increase the probability of error for environments that deviate from those defaults. If the cost of error is high, then consider not providing defaults or requiring confirmation of each value. Default values can be changed through modification.)
Show Rationale
Show Implementation Details

User Profile USAP: User Profile	Not yet considered [✓ Must modify architecture] Architecture addresses this Not applicable; ☐ Discuss status of responsibility
Environment Configuration USAP: Configuration Description	[✓ Not yet considered] Must modify architecture

Figure 5-5. Combining Responsibilities for Multiple Related USAPs.

With the responsibilities for the three related USAPs combined in this manner in the tool prototype, the next step was to proceed to user testing, which I discuss in Section 5.5.

5.5 User Testing in Practice

In this section I discuss user testing of the web-based prototype of the delivery tool for USAPs with software architects at ABB, with respect to usability and usefulness. I tested the delivery tool using the text-only portion of three new USAPs. The user testing in this phase took place at the ABB software development site in Västerås, Sweden.

Two days of user testing were performed. On the first day, a single software architect used the tool prototype to review the architecture of a mature software system (ten years old and in its fifth version). Although the tool prototype was designed primarily to assist architects in the early phases of designing a new software system, it is also possible to use it to uncover usability gaps in the designs of existing systems. On the second day, two software architects from the product line system project used the tool prototype together to review their preliminary architecture design. All the architects in the user tests used the tool prototype to review their designs while I observed and video recorded their process. They then filled out a 26-item survey, with questions derived from the Technology Assessment Model (TAM2) (Venkatesh & Davis, 2000) to assess their perceptions of the usefulness and usability of the tool prototype [Appendix N]. Finally, they participated in semi-structured interviews to uncover information that was not captured by the survey instrument. The instructions given to these architects can be found in Appendix M.

Before the use of the A-PLUS Architect prototype discussed here, the research team (ABB and CMU) had discussed whether we should include all of the four foundational USAPs in the web tool but decided to simplify the tests by omitting the Authorization foundational USAP. We expected this reduction to make the number of responsibilities tractable for a single day of testing. The architects at ABB were only available for one day each of testing, and we were concerned that the list of responsibilities might otherwise be too long for them to get through in a single day. Omitting responsibilities generated from the foundational USAPs for Authorization, and also Logging as it related to Authoring, was a logical division of the responsibilities and left what appeared to be a tractable and coherent set of 31 responsibilities for the architects to consider. [See Appendix K for a full list of responsibilities included in the user tests.]

One software architect working on a mature system in its fifth version used the tool to check whether the USAP tool might be general enough to be used by all ABB software developing businesses, even for systems that were not at the initial stage of architectural design, and also to further test the utility and usability of the tool. The two software architects from the product line system project used the USAP delivery tool at a time when they had completed a preliminary architecture design. One of these two architects was senior and had created most of the preliminary design. The second architect had recently joined the project but had previously been an experienced software architect at an automobile company.

The single architect with the mature system used the tool prototype in one session lasting two and a half hours interrupted by a fifteen-minute break. He began by reviewing the "Main Page" of the A-PLUS Architect tool prototype, which briefly described the USAPs embodied in the tool and gave instructions as to how to use the tool [Appendix M]. He examined each responsibility while thinking aloud, considering all the responsibilities at an average of about five minutes per responsibility. However, he did not make use of all the optional materials provided with the responsibilities. This architect viewed implementation suggestions for 13 of the 31 responsibilities, and rationales for 19 of the 31. It may come as little surprise that he chose to skip the implementation suggestions for 18 of the 31 responsibilities, considering that the system he was reviewing was in its tenth year and fifth version of implementation.

The two architects from the product line system project used the tool prototype in one session lasting six working hours broken up by a one-hour lunch break and two fifteen-minute coffee breaks. As with the single architect, they began by reviewing the "Main Page" of the A-PLUS Architect tool prototype. They examined and discussed each responsibility, made notes on paper as appropriate, and decided what response to make to that responsibility. In the six hours of work they completed consideration of all of the responsibilities for each of the USAPs. They averaged about 12 minutes per responsibility. As part of reviewing the responsibilities, these architects chose to view a vast preponderance of the optional materials. They clicked links to view the optional implementation suggestions for 29 of the 31 responsibilities. The two responsibilities for which they did not choose to review the implementation suggestions were the last two of three responsibilities in the Authoring Foundation USAP subsection on deleting a specification. The architects decided while reviewing the first responsibility in this subsection that deleting a specification was not applicable to their system, and did not examine the details of the second and third responsibilities in that subsection. The architects also clicked links to view the optional rationale for 28 of the 31 responsibilities, omitting the two responsibilities mentioned above, and a third responsibility related only to the execution of the system in displaying alarm state transitions in a timely fashion.

5.5.1 Performance Results

The architects achieved positive results using the tool prototype to review their software architecture designs during the user tests. The architect reviewing the mature architecture discovered one issue that had never been considered before, even though the USAPs had not been prioritized with his system in mind and was not designed to be used with a mature architecture. The architects reviewing the preliminary design of the new product line architecture identified nineteen "Must modify architecture" responses to the thirty-one responsibilities presented in the tool during the course of the one-day user test. A To-Do list was generated by the architects at the end of their user test. The architects chose to include the implementation suggestions for the responsibilities in their To-Do list, which they planned to review later when they addressed the usability issues they had identified in their preliminary architecture design.

During their use of the tool, the architects identified 14 issues that needed further consideration. Over the next several weeks, the architects considered these fourteen issues and their actual impact. The architects' judgment as to the resolution of each of the issues is detailed below.

Issue 1.	*Cost Saving: - would have been done anyway*
Issue 2.	*Cost Saving: - 1week*
Issue 3.	*Cost Saving: - 1week*
Issue 4.	*Cost Saving: - would have been done anyway*
Issue 5.	*Cost Saving: - very uncertain of value*
Issue 6.	*Cost Saving: - very uncertain of value*
Issue 7.	*Cost Saving: - very uncertain of value*
Issue 8.	*Cost Saving: - 1 week*
Issue 9.	*Cost Saving: - very uncertain of value*
Issue 10.	*Cost Saving: - would have been done anyway*
Issue 11.	*Cost Saving: - very uncertain of value*
Issue 12.	*Cost Saving: - 2 weeks, could be more if this idea is fully exploited*
Issue 13.	*Cost Saving: - very uncertain of value*
Issue 14.	*Cost Saving: - very uncertain of value*

For the issues where the architect felt secure in providing a value, 5 weeks were saved. Note the uncertainty of the architect with respect to many of the other issues. In the worst case, this uncertainty translates to no additional savings but, likely, there were additional savings beyond that estimated initially. This cost savings of five weeks represents a return-on-investment (ROI) of 16.67-to-1, given six hours of work (0.75 days) for two software architects invested to save twenty-five eight-hour days of future work. Note that this ROI derives from a first use of a prototype version of A-PLUS Architect during a first user test. The savings does not include the time the researchers have invested in producing the USAPs but Alarms and Events and user profiles are common usability scenarios. These USAPs are reusable across many projects and thus the investment to produce them will get amortized across multiple projects.

5.5.2 User Reactions

In general the architects felt that the USAP delivery tool was quite helpful. As one indication that they considered it useful, they asked for a copy of the tool so that they could have it available as they worked through their to-do list after the user tests. In semi-structured interviews immediately after the user tests they provided some insights regarding their impressions of the tool.

The main goal ABB had sought to achieve when applying the USAP technique was to incorporate usability support early in the design process in order to build in the support in the core architecture. By building in usability support early in the architecture, ABB expected to avoid late and costly redesign after the users have tested an actual version of the product line systems products. Some of the comments the architects made both during and immediately after the user tests reflected directly on the goal of early architectural usability support[2]:

Software Architect 1 (single architect, mature system): I think the tool would be used probably after any of the requirements but before doing the design, you will use this as a checklist of your…first design ideas or something, to make sure you… covered all this stuff. That would be very useful.

Software Architect 2 (paired architects, new system): We have discussed lots of internal stuff in the system but this gave us some picture of what the user is going to see.

Software Architect 3: And that is things that we were not going to get that input until very late in the design process, if we hadn't used this tool now. So it was good

[2] All the architects in the ABB user tests spoke English as a second language.

to have these points of view come in this early. I think we have identified at least a couple of new subsystems.

Software Architect 2: Yes. And some shortcomings of the previous design.

In contrast to the negative reaction to UML diagrams of a sample solution, as the paired software architects examined the responsibilities, they nearly always examined and discussed the implementation suggestions. One of their suggestions for improvement of the tool was that the implementation suggestions would be automatically included in the to-do list so that they would be available for future use, indicating that they saw these suggestions as useful instead of intrusive.

As the tool functioned during the test, the possibility of adding implementation details manually to each responsibility was perceived as tedious, since the software architects wanted to view all implementation details in the generated to-do list and were therefore forced to click “Add implementation details” for each responsibility.

The results of the TAM2 survey instrument were positive but not generalizable because of the very small sample size (n = 2, plus one incomplete instrument). I was able to apply TAM2 instrument items more broadly to assess the next iteration of the tool’s design; this will be discussed in the next chapter. Several important usability issues with the tool were also revealed by the user test, some of which I had anticipated and some not; these will also be discussed in the next section.

5.5.3 Design Observations

In order to identify issues in the design of the tool prototype that might make it difficult to understand or problematic to use, I performed a critical incident analysis of the video recordings of the user tests. I identified both positive and negative critical incidents based on types of statements users made in reaction to the user interface (UI) of the tool prototype (Figure 5-6). In order to qualify as critical, an incident had to last for at least one minute, beginning when the user first expressed either positive or negative reaction, and ending when the user either moved on to something else or found a solution to a problem. User interface incidents falling within the scheme but lasting less than a minute were recorded but regarded as non-critical.

Positive Reaction Types: User said something like…	**Negative Reaction Types: User said something like…**
That is good	That is bad OR that is not good
That works well	That does not work well
I can do <x> here	I want to <x> but I can't…
This is easy to use	This is hard to use

Figure 5-6. UI incident scheme for user statements

Two critical UI incidents and four non-critical UI incidents occurred during the course of the user tests. All of them took place during testing with the paired users (U2 and U3). The UI incidents are listed in Figure 5-7 below.

Incident	**Evidence**	**Explanation**	**Possible solution and/or tradeoffs**
Negative aspect: Cannot add comments until ToDo list is generated.	U2: "Hey, could I add comments? Where do I add my comments?" Said while viewing and discussing a responsibility. U2 referred back to tool's instructions, which say "The ToDo list has a place to record comments for any responsibility you wish to annotate. Annotate the list as desired, and then print it." U2 then recorded notes on paper.	Comments fields only occur on the ToDo list, not where the user is reviewing each responsibility. U2 wrote four pages of notes on paper while reviewing responsibilities during six hours of user testing.	Possible solution: provide functionality to add comments with each set of radio buttons. Tradeoffs: screen real estate; question as to how to save/display comments in ToDo list.
Negative aspect: Cannot add End-User USAPs to a visible responsibility.	U2 and U3 reviewed a responsibility that only used one End-User USAP (Alarms & Events). U3: "I would like to have the opportunity to choose Configuration Description [the parameter from the Environment Configuration End-User USAP] here." U2: "Yeah, I agree. It's not only a matter of…" U3: "It's not just Alarms & Events."	U2 and U3 wanted flexibility to discuss a responsibility for End-User USAPs in a case where the authors had not included that responsibility in that End-User USAP. The UI did not support this.	Possible solution: See discussion below the figure.

Figure 5-7. User Interface Incidents during User Tests

Incident	Evidence	Explanation	Possible solution and/or tradeoffs
Positive aspect: Heirarchical checklist is checked automatically.	U2 read the tool's instructions aloud: "When you have checked off the individual items under a responsibility, the checkbox to the left of the responsibility will be checked automatically to indicate that you have completed that section of the checklist." They responded: U2: "That could be good, this checklist." U3: "Yeah… we can see if we have missed something in the model."	U2 liked the idea of the checklist. U3 saw the automatic checking-off of the responsibilities hierarchy as potentially useful in avoiding errors of omission.	Tradeoff: New users may be confused by inability to manually check hierarchical checkboxes.
Positive aspect: ToDo list is generated as output.	U2 read the tool's instructions aloud: "When you have finished reviewing all the responsibilities in the checklist, click the "Generate ToDo List" link in the left menu to see a list of the current status of the responsibilities. This list will show all responsibilities that have yet to be addressed, required architectural changes, and/or need to be discussed further." U2 responded: "Hey, look here, we won't have to modify our model during the task because we will have a to-do list as output. That's good."	U2 liked the idea of having a To-Do list to take away from using the tool. Users may not want to perform changes in their architecture designs while using the tool. They may prefer to address them later with software architecture design or discussion.	Tradeoff: ToDo is a static snapshot, user may not think to generate new ToDo list if status changes on responsibilities.
Negative aspect: Hide/Show Rationale Hide/Show Implementation Details links change text when used.	U3 clicked the "Show Rationale" link in a responsibility. The text of the link changed to "Hide Rationale" and the rationale information appeared beneath the "Hide Rationale" link. U3: "I have a comment there about this Hide and Show Rationale. I would like to see, when I click Show, I would like to have the text beneath Show… It's a bit confusing." Implementation Details are displayed/hidden in the same way as Rationale in the UI.	User was confused by the way optional Rationale text was displayed / hidden using "Show Rationale" link when rationale text is hidden, and "Hide Rationale" link replacing "Show Rationale" link when rationale text is shown.	Possible solution: Use collapsible panels to indicate whether Rationale and Implementation Details are shown or hidden. In order to make this behavior clear to the users, an arrow pointing to the right (collapsed state) or down (expanded state) can be used. Tradeoff: Less experienced users may be confused.

Figure 5-7(cont). User Interface Incidents during User Tests

Incident	Evidence	Explanation	Possible solution and/or tradeoffs
Negative aspect: Implementation Details must be added to ToDo list one at a time.	U2 and U3 viewed the ToDo list they had generated. A link labeled "Add Implementation Details" appeared next to each responsibility displayed in the list. U2: "What is this Add Implementation Details?" U3: "We would like to get all implementation details for everything in the ToDo list, just to sort of provide a bigger context." U2: "We should have a…" U3: "Yeah, add *all* implementation details." U2 then clicked the "Add Implementation Details" link for each responsibility displayed (10 links in 38 seconds).	User wanted an easier and/or faster way to include Implementation Details for all responsibilities on ToDo List. Implementation Details can be shown on the ToDo list one responsibility at a time, by clicking a separate link on the ToDo list for each item on the list.	Possible solution: add one-click functionality to allow user to add Implementation Details to all ToDo list items at once. Tradeoff: User may add information she does not need to the ToDo list.

Figure 5-7(cont). User Interface Incidents during User Tests

The most interesting incident in Figure 5-7 was one in which users encountered the first specialized responsibility (EX 3.1, Appendix I) in the list of responsibilities, i.e., one that only addressed the End-User USAP for Alarms and Events. Upon reading this responsibility and noticing that it was only applied to Alarms and Events, one user remarked that he would like to be able to consider it for User Profile as well. This was not possible in the prototype tool, and the users moved on to discuss the responsibility in the context of Alarms and Events.

Although it is impossible to know precisely why this occurred, several possible explanations are worth considering. One possible explanation is that the users were experiencing functional fixedness (Duncker, 1945, cited in Functional fixedness, n.d., Examples in Research section, para. 1). By the time the users arrived at the responsibility in question, they had used the tool for several hours and had encountered only responsibilities with three parameters, so that became their new tradition. If this was the case, then the first time they encountered a responsibility with only a single parameter their expectations were confounded and they simply responded by expressing a desire to handle this responsibility exactly as they had handled those that came before.

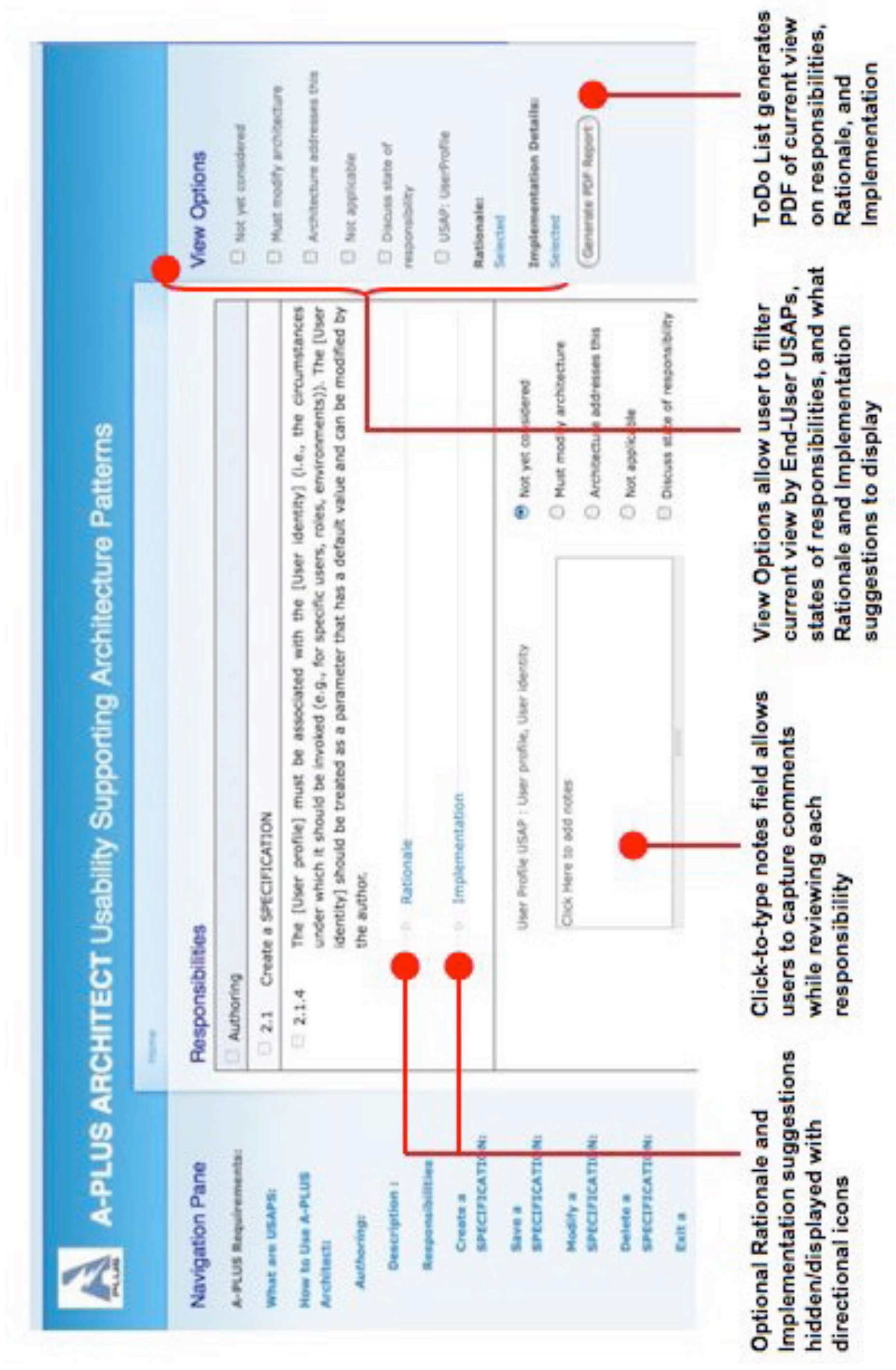
Another possibility is that we should have written the responsibility differently, since perhaps it could have been phrased in a way that made it clear to users why it was only relevant to the Alarms and Events USAP, and not to the User Profile USAP. Training might also have mitigated this situation. The instructions the users read before using the tool did not include any explicit statement that the number of End-User USAPs addressed in a single responsibility could vary, so perhaps they were not sufficiently prepared for what they could expect in using the tool. Yet another possibility is that the user is correct and this responsibility actually does apply to more than Alarm and Events and we should have included this responsibility in more than that single End-User USAP. This possibility opens up a question for further research: how might the users of A-PLUS Architect contribute to its content. This will be discussed more in Chapter 8.

When combined with comments made during the post-test interviews as to what they did and did not like about using the tool, several issues emerged that formed a list of requirements for redesign. I incorporated these issues in a redesign of the delivery tool: A-PLUS Architect.

5.6 A-PLUS Architect Redesign

Based on the results of user tests of the tool prototype in the previous phase, the design of A-PLUS Architect retained use of automated hierarchical checkboxes in the user interface to encourage attention to each responsibility through automated hierarchical checkboxes. The software architects in the user tests obtained good results and commented favorably on the usefulness of the checklist in forcing them to pay attention to all the responsibilities in each USAP (Figure 5-8).

In response to the critical incident regarding inability to record comments in the tool until the To-Do list was generated, a note-taking field was added to each USAP in the tool. During the user tests, the software architects had recorded some notes on paper when they found that the tool had no mechanism for capturing process notes. In the revised design, editable text Notes fields were added so that users could capture their thought processes while evaluating individual responsibilities.



A-PLUS ARCHITECT Usability Supporting Architecture Patterns

Home

Navigation Pane

A-PLUS Requirements:
What are USAPS:
How to Use A-PLUS Architect:
Authoring:
Description :
Responsibilities
Create a SPECIFICATION:
Save a SPECIFICATION:
Modify a SPECIFICATION:
Delete a SPECIFICATION:
Exit a

Responsibilities

☐ Authoring
☐ 2.1 Create a SPECIFICATION
☐ 2.1.4 The [User profile] must be associated with the [User identity] (i.e., the circumstances under which it should be invoked (e.g., for specific users, roles, environments)). The [User identity] should be treated as a parameter that has a default value and can be modified by the author.

▷ Rationale
▷ Implementation

User Profile USAP : User profile, User identity

Click Here to add notes

● Not yet considered
○ Must modify architecture
○ Architecture addresses this
○ Not applicable
☐ Discuss state of responsibility

View Options

☐ Not yet considered
☐ Must modify architecture
☐ Architecture addresses this
☐ Not applicable
☐ Discuss state of responsibility
☐ USAP: UserProfile

Rationale:
Selected

Implementation Details:
Selected

Generate PDF Report

Optional Rationale and Implementation suggestions hidden/displayed with directional icons

Click-to-type notes field allows users to capture comments while reviewing each responsibility

View Options allow user to filter current view by End-User USAPs, states of responsibilities, and what Rationale and Implementation suggestions to display

ToDo List generates PDF of current view on responsibilities, Rationale, and Implementation

Figure 5-8. Design Iteration on UI of A-PLUS Architect

To address any user confusion regarding the mechanisms for hiding and showing optional rationale and implementation suggestions, links that changed text when changed were replaced with collapsible panels. These icons indicated whether additional text was either available to view, or available to hide. In order to make this behavior clear to the users, an arrow pointing to the right (collapsed state) or down (expanded state) was used. Users were given options to include rationale and/or implementation details in the To-Do list, and to view any of the USAPs separately, or all of them together.

The To-Do list had elicited both positive and negative responses in the user tests: the users responded positively to the concept, but negatively to part of the implementation. In exploring solutions to the problem of users wanting a simple way to show all implementation details at once on the To-Do list, I realized that simple redesigning the To-Do list UI as a *view* on the entire set of USAPs could add a great deal of flexibility for the user. Like the To-Do list in the first UI, a view could be saved or output as a snapshot at any time during the review process. However, unlike the original To-Do list, which was conceived as a report, the view-based To-Do list precisely matched whatever the user chose to view in the tool at the time that the To-Do list was created. A user could select to filter their view of the tool at any time, e.g., to view only the responsibilities for one End-User USAP at a time, to show Rationales, etc., and the output of the To-Do list would capture that view. With the To-Do list captured as a view of the checklist, the new Notes fields by the individual USAPs were also shown in the To-Do list unless the user selected otherwise in the view. Redesigning the To-Do list as a view on the current state of the USAPs at any time introduced additional flexibility for the end users.

Some visual design changes were made as well, including elements of the color palette, fonts, and use of white space to reduce clutter and make it easier to read large amounts of text.

One significant structural improvement that was included in the design of A-PLUS Architect did not arise from user testing, but was necessary to move from prototype to full-fledged design tool. The browser-based tool prototype was implemented only in

HTML and JavaScript, had no persistence beyond a single browser session, and no way to store the output beyond printing a To-Do list. A-PLUS Architect was designed to use a content management system to support the content of the foundational and end-user USAPs, and to be supported by a database backbone. It is intended to be a web-based application, suitable for use on multiple browsers and operating systems, so as to be usable by software architects on a variety of systems.

Implementation of the A-PLUS Architect application is beyond the scope of this thesis. I did work with a team of graduate software engineering students who attempted to implement the revised tool design, but their full implementation never achieved an acceptable level of functionality or performance. Nor was the implementation necessary in order to evaluate the design of A-PLUS Architect for perceived usefulness and usability in a broader scope, as will be discussed in the next chapter.

5.7 Summary

The A-PLUS pattern language discussed in Chapter 4 produced a representation of USAPs that separated End-User USAPs from Foundational USAPs and allowed more uniform and reusable authoring of USAPs. For software architects to use the information in A-PLUS, End-User USAPs and Foundational USAPs must be combined to form lists of responsibilities. In chapter I described a tool, A-PLUS Architect, that I designed to embody and combine the elements of the A-PLUS pattern language and deliver them to software architects for use in architecture design practice. I designed this browser-based tool simultaneously with the group development of the A-PLUS language.

I designed A-PLUS Architect to allow the software architects to review their software architecture designs against the checklist of responsibilities included in one or more USAPs, determine if each responsibility was addressed, and if it was not, decide whether to alter the architecture design. I hid the A-PLUS language's internal concept of dividing USAPs into Foundational USAPs and End-User USAPs from the user as extraneous information that was not needed by users in order to use the tool. I used automated checkboxes to enforce the hierarchical structure of the sets of responsibilities and to remind the architects to consider every responsibility included in each USAP. I chose to display the text-based implementation suggestions and rationales for each responsibility as optional information that users could view on demand. A prototype of my tool design was implemented by the researchers at ABB.

I conducted user testing of the A-PLUS Architect prototype with software architects at ABB's corporate development site. Two paired architects working on the new product line system design used the prototype for a single day. They uncovered fourteen major usability issues in their preliminary software architecture design. The lead software architect reviewed the list of issues after the user tests and estimated that the single day of user testing by two software architects had saved twenty-five days of development time further down the road, for a return on investment (ROI) of 25:2.

Following the positive results of the user tests, I iterated on the design of A-PLUS Architect. Given the predominantly smooth interaction observed during the user test and confirmed by comments by the users, I retained the simplicity of the hierarchical checklist format. However, the redesign addressed the few usability issues that were uncovered in the user tests, e.g., allowing user to record process notes, and some known issues that had been deferred until after the first prototype was tested for usefulness, e.g., data persistence. The evaluation of the redesign will be discussed in the next chapter.

Chapter 6. Further steps for A-PLUS Architect

6.1 Evaluating Perceived Usability and Usefulness

In this chapter I discuss the evaluation of a revised design of the A-PLUS Architect tool for perceived usability and perceived usefulness. In order for A-PLUS Architect to be practical for use in the field, a software engineer must be able to use it to apply USAPs independently in the software architecture design process. Although I have shown USAPs to be useful in laboratory studies and industry user tests, a significant problem that remains is to get them accepted into work practice by professional software developers. To this end I used the Technology Assessment Model to assess the perceptions of software architects as to whether the redesigned A-Plus Architect tool would be useful and usable. The question to be answered was whether this tool was something software architects could see themselves using to address usability issues in their software architecture designs, provided that those usability issues were deemed important by the project stakeholders of the software being designed.

6.1.1 Measuring User Acceptance

Although implementing the iterated design for A-PLUS Architect lay outside the scope of my thesis, I wanted to assess the likelihood that the intended end users of the tool, software engineers and architects, would accept the tool for professional practice if it were implemented and made available to them. In this work, user acceptance was measured by intent to use rather than by actual usage. This approach was taken by (Hu, Chau, Sheng, & Tam, 1999) in evaluation the potential of acceptance for then-new telemedicine technology by medical professionals.

Data were collected using an online survey instrument, with items drawn from the Technology Acceptance Model, or TAM. The Technology Acceptance Model, as described in Chapter 2, is designed to explain the causal links between "perceived usefulness and perceived ease of use" on the one hand, and "users' attitudes, intentions and actual computer adoption behavior" on the other.

6.1.2 Participants

Recruiting participants for meaningful testing of software architecture design tools and methods is difficult because software architecture design is an expert discipline within the field of software engineering. Such experts are difficult to corral, costly to hire, and often rather busy. On the other hand, testing the same tools and methods with more readily available software engineering students could lead to questions about whether the results were generalizable to experts in practice. Given these constraints, I elected to recruit participants for my evaluations from several possible pools of professional software engineers and software architects for remote evaluations. Participants were recruited via email and web postings to professional organizations.

Of 133 people who accessed the online survey, 43 participants completed the survey instrument. All 43 participants attested to be over 18 years of age. Participants reported having worked an average of 15.94 years in professional software development, with a range of 4 to 33 years.

To determine participants' level of design experience we asked whether they had performed several design tasks of varying complexity. All participants reported some design experience: 88% of participants reported having designed the overall structure of a system, 100% reported having designed the structure of a subsystem within a system, 93% reported having designed the structure of a component within a subsystem, and 81% reported having designed a class hierarchy within a component.

Participants were also asked to report on circumstances under which they had studied software architecture. Fifty-three percent of participants reported having studied software architecture in university at the undergraduate level, 33% in graduate school, 63% in professional development courses/seminars, 74% through on the job training, and 63% of participants reported having studied software architecture independently, these options being non-exclusive. Additionally, 5% reported that they had not studied software architecture at all.

Participants were also asked to report their job titles. The large number of different job titles is indicative of the wide variety of job positions held by software development professionals whose responsibilities include designing software architecture. A table of these titles and their frequency appears in Figure 6-3, from which you can see the diversity of nomenclature.

Job Title	**# of Respondents**
Application Architect	3
Architect	2
Consultant	2
CTO	1
Data Processing Supervisor	1
Director of Senior Architecture	1
Enterprise Architect	1
Integration Architect	1
IT Architect	2
IT Consultant	1
Lead Technical Architect	1
Principal Architect	1
Principal Software Development Engineer	1
Principal Solutions Consultant	1
Product Development Manager	1
Programmer Analyst	1
Scientific System Developer	1
Senior Software Consultant	1
Senior Software Engineer	3
Senior Systems Analyst	1
Senior Technology Architect	1
Software Architect	5
Software Consultant	1
Software Designer/Architect	1
Software Developer	1
Software Developer/Architect	1
Technical Architect	1
Technical Manager Software Architecture	1
UK Head of Architecture and Analysis	1
VP of Product Development	1

Figure 6-1. Job Titles of Survey Respondents

6.1.3 Method

The evaluation used items drawn from the Technology Acceptance Model (TAM) (Davis, 1989; Davis, Bagozzi, & Warshaw, 1989; Venkatesh & Davis, 2000; Venkatesh, Morris, Davis, & Davis, 2003), which I will describe first. The TAM is a validated survey instrument for predictive assessment of use of a software system, measuring the perceived usability and usefulness of a software system (or other product) once the system has been created. It is easy to administer and measures attitudes toward the new system, not task performance. Perceived usability and usefulness are critical to new methods and tools in a professional development environment because users, unless they are forced, will not use software unless they perceive it as both helpful and reasonably easy to use.

Twenty-one items measured responses in four categories: perceived usefulness, perceived ease of use, intention to use, and attitude. Different versions of the TAM have included a variety of items. In choosing these items, I followed an approach used by (Hu, Chau, Sheng, & Tam, 1999), who evaluated the potential of acceptance for then-new telemedicine technology by medical professionals when actual usage of the technology could not be measured. The TAM items included in the survey are shown in Figure 6-4 below.

I made the revised design of A-PLUS Architect, including changes described in Section 1.2, available to online participants for them to examine on their own (Appendix M). After reviewing the design, participants were asked to fill out the TAM survey items.

Participants were also asked to respond to a brief set of items regarding their professional experience in the field of software architecture. Using the online survey had the advantage of allowing me to collect data from a greater number of participants than might have been available to participate in a focus group, for reasons of time, convenience, or location. On the other hand, there was the disadvantage of not being able to ask participants why they provided the answers they did.

Perceived Usefulness (PU)
PU1: Using A-PLUS Architect can enable me to complete software architecture design work more quickly.
PU2: Using A-PLUS Architect cannot improve my software architecture designs.
PU3: Using A-PLUS Architect can increase my productivity in software architecture design work.
PU4: Using A-PLUS Architect cannot enhance my effectiveness at software architecture design work.
PU5: Using A-PLUS Architect can make my software architecture design work easier.
PU6: I would find A-PLUS architect not useful in my software architecture design work.
Perceived Ease of Use (PEU)
PEU1: Learning to use A-PLUS Architect would not be easy for me.
PEU2: I would find it easy to get A-PLUS Architect to do what I need it to do in my software architecture design work.
PEU3: My interaction with A-PLUS Architect would be clear and understandable.
PEU4: I would find A-PLUS Architect inflexible to interact with.
PEU5: It would not be easy for me to become skillful in using A-PLUS Architect.
PEU6: I would find A-PLUS Architect easy to use.
Attitude (ATT)
ATT1: Using A-PLUS Architect in software architecture design is a good idea.
ATT2: Using A-PLUS Architect in software architecture design would be unpleasant.
ATT3: Using A-PLUS Architect would be beneficial to my software architecture design work.
Intention to Use (ITU)
ITU1: I intend to use A-PLUS Architect in my software architecture design work when it becomes available to me.
ITU2: I intend to use A-PLUS Architect to review my software architecture designs as often as needed.
ITU3: I intend not to use A-PLUS Architect in my software architecture design work routinely.
ITU4: Whenever possible, I intend not to use A-PLUS Architect in my software architecture design work.
ITU5: To the extent possible, I would use A-PLUS Architect to review different software architecture designs.
ITU6: To the extent possible, I would use A-PLUS Architect in my software architecture design work frequently.

Figure 6-2. TAM items used in A-PLUS Architect Survey

6.1.4 Results

6.1.4.1 Results of TAM Questions

Survey items were measured on a 7-point scale, anchored at 1 (strongly disagree) and 7 (strongly agree), with a value of 4 representing neutrality (neither agree nor disagree).

Half of the questions had been negated and the questions randomized to reduce potential monotonous response effects. Responses to the negated items were inverted on the 7-point scale during analysis to allow assessment of convergent validity among the items in each category. Figure 6-5 below shows descriptive statistics for each of the 21 items in four categories. Values for the negative valence items were inverted prior to statistical analysis.

Category & Item	**Mean**	**Standard Deviation**	**Cronbach's Alpha**
Perceived usefulness (PU)			0.87
PU1	4.30	1.63	
PU2	5.05	1.80	
PU3	4.95	1.29	
PU4	5.33	1.49	
PU5	4.88	1.24	
PU6	5.00	1.63	
Perceived Ease of Use (PEU)			0.85
PEU1	5.65	1.46	
PEU2	4.53	1.50	
PEU3	5.37	1.22	
PEU4	4.77	1.48	
PEU5	5.40	1.26	
PEU6	5.28	1.35	
Attitude (ATT)			0.83
ATT1	5.47	1.50	
ATT2	5.16	1.70	
ATT3	5.16	1.33	
Intention to Use			0.93
ITU1	4.84	1.73	
ITU2	4.44	1.58	
ITU3	4.56	1.84	
ITU4	5.26	1.63	
ITU5	4.84	1.36	
ITU6	4.70	1.60	

Figure 6-3. Descriptive Statistics for TAM Survey

Mean values for perceived usefulness ranged from 4.30 to 5.33, with a combined average of 4.92. Mean values for perceived ease of use were slightly higher, ranging from 4.52 to 5.65 with a combined average of 5.17. Mean values for attitude were the highest of the four constructs, with a range from 5.16 to 5.47 and a combined average of 5.26. Mean values for intention to use were comparable to those for perceived usefulness, having a range from 4.44 to 5.26 and a combined average of 4.92.

At this writing, there are no published norms for what the absolute results mean for technology adoption. The model for combining the components and their effect on intention to use is evolving and stabilizing and a correlation with technology adopted has been obtained in four longitudinal studies (Venkatesh & Davis, 2000), but the absolute responses are not reported for those studies. A later set of longitudinal studies (Venkatesh & Bala, 2008, p. 287), however, do report their absolute results for TAM items administered after initial training on new systems but before those systems had been used. In these studies, means were reported at 4.14 for perceived usefulness, 3.98 for perceived ease of use, and 4.10 for behavioral intention (a renaming of intention to use). No measurement of attitude was collected in these studies. Their results were smaller than those in my survey, and in all cases the new technology was adopted by the users. Their findings were that perceived usefulness was the strongest predictor of behavioral intention at all points of measurement (p. 290), and that behavioral intention was a significant predictor of use at all points of measurements (p. 291). This suggests that my survey's high results for perceived usefulness would translated to a likelihood that software architects would be willing to try A-PLUS Architect.

6.1.4.2 Examination of the TAM

To contribute to the ongoing endeavor of evolving the TAM, we examined the internal consistency and explanatory power, as have previous researchers. Internal consistency in terms of reliability was evaluated using Cronbach's alpha. As shown in Figure 6-5, the values were at or above 0.82. This result is similar to results found in other TAM studies (Venkatesh & Davis, 2000, p. 201; Moon & Kim, 2001, p. 224; Hu, Chau, Sheng, & Tam, 1999, p. 101). These strong results for reliability and internal consistency indicate that TAM survey items for each construct are well correlated.

I examined the explanatory power of the model for the individual constructs included, using the resulting R^2 for each dependent construct (Figure 6-6). The R^2 values indicate the extent to which the overall variation in an outcome can be explained by an input variable. The data supported the causal paths postulated by TAM. Perceived ease of use had a significant direct positive effect on both perceived usefulness and attitude, with path coefficients of 0.74 and 0.96 respectively. These coefficients mean that for every unit increment in perceived ease of use, we could expect perceived usefulness to increase by 0.74, and attitude to increase by 0.96. Perceived usefulness, in turn, had a significant direct effect on both attitude and intention to use, both with path coefficients of 0.96. Attitude also had a direct significant effect on intention to use, with a path coefficient of 0.90. All of these were significant at a level of $p < 0.0001$.

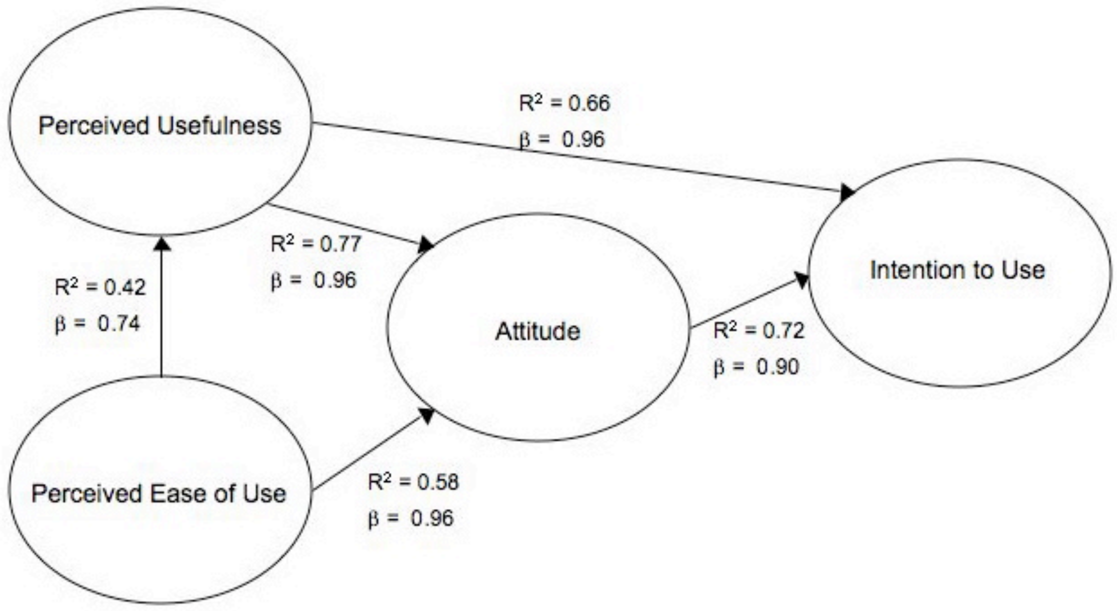


Figure 6-4. Results of the model

It is worth noting that Hu's use of the TAM with physicians found no significant influence of perceived ease of use on altitude or perceived usefulness. They hypothesized that "TAM may not be appropriate for user populations who have considerably above-average general competence and intellectual capacity or have constant and reliable access to assistance in operating technology. The explanatory power of TAM, particularly the perceived ease of use factor, may weaken as the competency of the users increases.” (p. 106). Our results belie this hypothesis. The experienced software engineers who responded to our survey are almost certainly generally competent, have high intellectual capacity, and are especially technically competent, but perceived ease of use significantly influenced their perception of usefulness and attitude. Perhaps it is precisely their competence and their extensive experience with computer applications (especially with frustratingly un-usable applications!) that increases the influence of perceived ease of use on these factors. This points to the need for more research to unearth the situations under which TAM is appropriate.

6.1.5 Discussion

Assessing the factors related to likelihood that new technology will be accepted could be very helpful in educational software as well, especially when it is not practical to immediately assess learning from new tools used by teachers in their classrooms. The use of the TAM survey instrument may therefore also be applicable to new technologies in education research. The significance of this statement will become apparent in Chapter 7 below.

On average the responses to the revised design of A-PLUS Architect were entirely positive as indicated by the TAM items. However, stronger positive results would have been a better indication of the likelihood that A-PLUS Architect would be accepted into professional practice in the form taken by the revised design. The most encouraging results of the survey instrument were those for attitude and perceived usefulness, while results for perceived ease of use and intention to use were slightly lower. From this it seems that survey respondents might have been less satisfied that the tool would be easy to use than that it would be useful and beneficial. If this were the case, then increasing their confidence in perceived ease of use might contribute to a greater likelihood that they would be willing to use the tool if it were made available to them. User acceptance has long been a problem in information technology, but with the use of a model like the TAM, we can surmise that a mildly positive result would correspond to some willingness trying the tool. And if users were to achieve positive results with A-PLUS Architect as did the architects at ABB, they would be motivated to use it again where relevant.

Although the work on USAPs has been confined to a highly specialized subdomain of software engineering, it is possible to extrapolate some of the understanding we have reached to benefit research in other domains. In Chapter 7, we will examine the possible application of some of the research concepts from the work on USAPs to the domain of education research.

6.2 Summary

I showed the iterated design of A-PLUS Architect online to professional software architects and used a survey instrument, the Technology Assessment Model, to measure their perceptions of the redesigned tool's usefulness and usability, attitudes toward the tool, and whether they would intend to use the tool if it became available to them.

Forty-three software architects examined the design and completed the survey. On average the responses to the revised design of A-PLUS Architect were entirely positive as indicated by the TAM items. In the TAM model, perceived ease of use, perceived usefulness, and attitude are predictive of intention to use. The strongest results of the survey instrument were those for attitude and perceived usefulness, while results for perceived ease of use and intention to use were slightly lower.

User acceptance is often a problem in information technology, but from the results of this model we can surmise that a positive result would correspond to willingness trying the tool. And if users did achieve positive results with A-PLUS Architect, as did the architects at ABB, they would be motivated to use it again where relevant.

Chapter 7. Extension to the Education Research Domain

7.1 Reaching Across Domains

In the preceding chapters I discussed a method for addressing usability issues in software architecture design through USAPs, as well as a software tool to apply that method in practice. Although this method and tool were developed specifically for use to bridge the domains of human-computer interaction (HCI) and software engineering, the embodied concept of using checklists of responsibilities developed in collaboration by experts in multiple domains is not, in principle, limited to the domains I have shown. In this chapter, I speculatively extend research concepts embodied in my work on USAPs to the domain of education research.

7.1.1 Solving for the Software Engineering Domain

The USAPs method works by helping software architects analyze software architecture designs using a checklist of usability responsibilities to determine whether the software architecture design includes support for each responsibility. Using USAPs, software architects decide whether each responsibility is applicable to the software system they are designing. Their decisions are based on their knowledge of the conditions and constraints under which the software will be used. If the architects decide that a responsibility is applicable to their software system, they then determine whether the design of the system provides adequate support for the fulfillment of that responsibility. This method is embodied in a software tool, A-PLUS Architect. The resulting outputs of analyzing a software architecture design using A-PLUS Architect are lists of which usability responsibilities are already supported by the architecture design, and which will require changes to the architecture before they can be supported. The software architects also have the option to receive lists of suggestions for functional elements a software architecture design should include in order to fulfill each responsibility on these lists.

As will be discussed in Section 7.2.2, existing curriculum design templates are deficient in dealing with the full range of conditions and constraints that must be satisfied by the educational infrastructure in order to achieve the learning goals expressed in curriculum standards. Further, even where different aspects of curricula are considered in related frameworks (e.g., California Department of Education 2000, California Department of Education 2010), they do not provide explicit linking between these aspects, making it difficult for decision-makers to understand the implications of choices in one aspect on the efficacy of another. In this chapter I propose an enhanced curriculum design analysis method, capable of providing more complete descriptions of the conditions and constraints that must be satisfied by the educational infrastructure in curriculum designs. The goal of the proposed method is to apply the structure of USAPs as expressed in the A-PLUS language, and the tool embodiment of that language in A-PLUS Architect, to the task of analyzing curriculum design within the larger context of the educational infrastructure.

This chapter is intended to provide the basis for a future research proposal to the Institute of Education Sciences (IES) in the U.S. Department of Education, entitled "Addressing Important Issues In Education Through Analogy From Another Domain: A Method and Tool for Curriculum Design Review."

7.1.2 The Problem Space in Education Research

Much educational research focuses on improvements and interventions in curriculum. Within the definitional structure of the IES, research on curriculum standards falls into the area of education policy, finance, and systems. "Through the Policy/Finance program, the Institute supports research to improve student learning and achievement through the implementation of systemic programs and broad policies that affect large numbers of schools within a district, state or the nation. Systemic programs and policies may seek to impact student outcomes by attempting to change the behavior of large numbers of students (e.g., offering material incentives for improved academic and behavioral outcomes). More often, systemic programs and policies work indirectly to impact student outcomes through changing how large numbers of schools or districts carry out instruction and the functions that support or evaluate instruction. For example, district and state curriculum standards and assessments directly impact what is taught (Institute of Education Sciences, 2010, p. 34)." The project described in this chapter would come under the further IES definition of a Development/Innovation project, which is intended "to support development of and innovation in education interventions – curricula, instructional approaches, technology, policies, and programs…. Development/Innovation applications are about development and not about demonstrations of the efficacy of the intervention (p. 54)."

Despite the perceived impact of curriculum standards on student outcomes, curriculum design is often so constrained by the rest of the educational system that many potential ideas may not even be considered because curriculum designers accept (implicitly or explicitly) the many constraints of the system. Curriculum, in the form of content standards, is designed and redesigned independently of and after the deep structure of the educational system: funding of school districts, professional education requirements for teachers, even assessments of student learning. The deep structure of the educational system constrains the structure of the curriculum, and educational researchers seeking to improve curriculum design rarely have opportunities to suggest or implement changes to the school system.

7.1.3 Structural Similarities of the Problem Domains

There are many structural similarities between curriculum design in education and user interface design in software engineering. Software development practices in the 1980s separated the user interface from the core functionality of software systems. This separation had the effect of constraining user interface design to rely on what core functionality was provided by the system. In practice, this is much like the constraints placed on curriculum design, that make it easier for designers to change the curriculum than to change the deep structure of educational systems. In the separated model in software architecture, the user interface, like curriculum design, is often developed after the rest of the system is already in place and must utilize whatever functionality the deep structure of the system can provide as best it can. If the user interface has needs that are not supported by the core functionality, meeting those needs may require changes or redesign of the core; and if the changes are too great, and the expenses thus too high, those changes may be rejected regardless of whether severe disadvantages result to the end user. Likewise, excellent curriculum design may not have a chance to reach the student if the education system considering its adoption does not support its requirements.

Curriculum designers have many commonalities with user interface designers. Both user interfaces and curricula are constrained by the systems in which they must function. Tradeoffs must be considered and made to satisfy the availability of time and resources. In curriculum design, for instance, there might be a strong learning-based reason for wanting to combine some math and science classes; this integration could have impact on a host of educational system constraints, e.g., teacher training, facilities management, technology, etc. Both curriculum designers and user interface designers may have expertise in some areas, but not in others, and may therefore need to enlist the assistance of experts in other domains to fulfill their design requirements.

Some of the similarities in the structure of these two problem domains are shown in Figure 7-1 below.

7.2 Transferring the USAPs Method to Education Research

7.2.1 USAPs Content for the Software Engineering Domain

In the preceding chapters, I described work to address this problem in the software engineering domain. I described a method for addressing usability issues (which the user will encounter through the user interface) early in the design process, when the deep structures of the software architecture are being designed. Part of that work included creating a language to enumerate issues that need to be addressed in the UI design (Chapter 4) and expressing that language in a template so that software architects could use it to identify areas where the software architecture might need to be modified to support those UI design issues (Chapter 5). Developing the content for that template required extensive participation from experts in software architecture and usability.

Software engineering	**Education**
Software architecture. • mostly legacy • mostly accrued, not designed • more intractable to change the longer the system has been in place	Educational system. • mostly legacy • mostly accrued, not designed • more intractable to change the longer the system has been in place
User interface (UI) design. • after design of the rest of the software system • severely constrained • not generally allowed to make changes to the software architecture	Curriculum design. • after design of the rest of the educational system • severely constrained • not generally allowed to make changes to the educational system
User interface. • designed and redesigned independently of and often after the deep structure of the software architecture	Curriculum. • designed and redesigned independently of and often after the deep structure of the educational system
Constraints. • hardware environment • intended users • legacy software systems • time, costs, standards, etc.	Constraints. • educational system infrastructure • availability and training of teachers • legacy materials and technology • time, costs, standards, etc.
Experts needed to get it right. • experts in usability • experts in software architecture	Experts needed to get it right. • experts in curriculum design • experts in educational systems

Figure 7-1. Structure of Problem Domain

Through the work described in the preceding chapters, experts in the domain of software architecture and usability have access to:

- Re-occurring usability scenarios that have benefit to the user;

- For each scenario, conditions under which the scenario may be applicable;
- For each scenario, an enumeration of activities that the software architecture may have to perform to achieve that benefit;
- For each activity, suggestions for implementation in the software architecture;
- For each activity, a rationale for why this activity supports its scenario;
- Suggestions for enumerating these activities (i.e., considerations such as time, scope, initiative, and feedback)
- A checklist of responsibilities that address usability concerns arising from each scenario, all of which must be considered when designing software architecture.

I have verified that use of the method results in improved software architecture design, both in experiments (Chapter 3) and in the field (Chapter 5). I have also developed an effective delivery mechanism for the method, as embodied in the A-PLUS Architect tool (Chapter 5).

7.2.2 Curriculum Design Methodology

The question remains, how could this method be transferred productively to the domain of education? Using the methodology we have developed, education experts could have a method for enumerating issues that need to be addressed in curriculum design and identifying changes that need be made to the education system to support those curriculum design issues. Many curriculum standards, as they exist today, address content separately from the instructional materials, technology, teacher training, and other environmental variables that affect the ability of an educational system to educate students, i.e., the context in which learning will take place. Examples may be found at (California Board of Education, 2010; New York State Education Department, 2009). The separation of content standards and educational system constraints within the development of curriculum and instruction standards makes it a complex task to identify whether changes in how the curriculum is implemented will necessitate changes to the educational environment.

Despite the fact that most K-12 education is constrained by curriculum standards, some educational design researchers take the position that curriculum design based on content standards is the wrong place to begin. The backward design model proposed by Wiggins and McTighe suggests that educators should first identify the desired learning results or goals for students, then determine what constitutes acceptable evidence of those learning results, and finally plan learning experiences and instruction to achieve those results (Wiggins & McTighe, 2005, p. 18). Their approach to curriculum design uses templates that begin with established goals such as content standards but that also require educators to consider what understandings are desirable, what misunderstandings can be predicted, what critical knowledge and skills students will acquire, and the implications of that knowledge for further learning and understanding. Only after considering these issues are educators encouraged to progress to devising performance tasks and learning activities that will help students to achieve the desired learning goals. In this model, termed Understanding by Design (UbD), curriculum standards and environmental factors become design considerations (p. 34) that inform the design of instructional units, with each unit resulting in new knowledge and skills that can serve to scaffold students' further understanding of the subject matter. Since curriculum standards are at the same level of consideration with environmental factors in this model, both act as constraints on the design, but suggestions for change to either one are not part of the output of the UbD design method.

Other research addresses the notion of design research to include environmental factors in creating a learning environment within a local classroom context. Bielaczyc (2006) suggested a Social Infrastructure Framework for use in design research at this localized context, taking into account not only the content being taught and the tools being used to teach it, but the cultural beliefs of students and teachers, the practices through which learning activities are organized, the socio-techno-spatial relations governing locations of computers and data and the extent to which students work in collaboration, and the level of interaction with the "outside world" (Bielaczyc, 2002, pp. 314-15). Although this framework focuses on social factors associated with using a "technology-based tool" in a local classroom setting, Bielaczyc states that research is needed "to go beyond understanding the impact of a *given* classroom social infrastructure on the integration of a technology-based tool and begin to systematically identify and analyze the aspects of social infrastructure that are amenable to design" (2006, p. 303).

By using a method and a software tool similar to the one described in Chapters 5 and 6 above, in which software architects were able to review software architecture designs systematically against a set of usability responsibilities, many environmental factors related to specific curriculum needs could be addressed in design research at multiple levels using a single tool. Curriculum designers could review content and instruction standards against the environmental constraints of an educational system, at the state, district or school level, to identify changes that might need to be made in that system to support delivery of the standard curriculum. Enumerating needs for resources to support teaching is not a novel idea. The "Doing What Works" website (U.S. Department of Education, 2010) offers planning templates for educators to "carry out comprehensive needs assessment and planning" in preparation for working with state education agencies, school districts and schools to get the resources they need to implement research-based best practices. At each of the three levels of the education system, these templates enumerate areas of support the system should provide: leadership, setting standards and expectations, recommending and providing research-based curricula, instruction and assessments, quality of staff, planning and accountability. Additional areas are included at different levels: fiscal adequacy at the state level, family and community engagement at the district and school level, and a safe and supporting learning environment at the school level. Educators using these templates are directed to note whether support exists in each of these areas, potential areas for development, and what next steps might be taken at the appropriate level of the education system to address each area using research included in the list of "Doing What Works" resources.

More detailed recommendations for instruction in specific areas of curriculum are provided in the Institute of Education Sciences (IES) "Practice Guides" through the "What Works Clearinghouse" (Institute of Education Sciences, 2010b). The goal of the Practice Guides is to provide evidence-based suggestions for screening students for difficulties in different instructional areas (e.g., mathematics), provide interventions for struggling students, and suggest ways to monitor student responses to those interventions. The IES Practice Guides offer concrete implementation steps for interventions in a number of curriculum areas: literacy (K-12), mathematics (K-8), and early childhood education (pre-K). They also address strategies for special student populations – English language learners and students with learning disabilities – and socio-educational issues of character education and dropout prevention. Although the Practice Guides do not address instructional practices in other basic areas like science education or social sciences, they do cover identifying and addressing student difficulties in K-8 mathematics and K-12 reading at a formidable level of detail. Each Practice Guide includes a checklist for general steps to implement the recommendations therein, as well as research-based evidence for why the suggested recommendations are included. Unlike the Doing What Works materials, the IES Practice Guides do not make recommendations for general classroom instruction in mathematics or reading; instead they focus on identifying struggling students and helping them succeed.

Some of the principles embodied in the Doing What Works templates and the IES Practice Guides align with what I am proposing. The technique of considering whether the various levels of the education system provide adequate support for instructional practices, and noting areas in which additional support may be needed, is similar to how software architects used A-PLUS Architect to review usability responsibilities. However, the three Doing What Works templates for working with state education agencies, school districts and schools differ from USAPs and their A-PLUS embodiment in three important ways. First, the Doing What Works templates do not share information with one another; each template must be filled in separately by hand, with no assurance of completeness or internal consistency, and no linking of the consequences that a choice (or no choice) at one level will have on the other levels. Second, the Doing What Works templates address application of research-based teaching practices, but they are not directly linked to the content of standards in the different areas of curricula (e.g., math, science, literacy), i.e., they are more general in their approach than USAPs. Third, they are not directly linked to the resources that would suggest specific research-based steps to address; instead, the educator who identifies an area for further development must access the research and make the connections.

The IES Practice Guides detail more specific implementation suggestions for interventions in some areas of curriculum, similar to and indeed more prescriptively detailed than USAPs' implementation suggestions. The Practice Guides also provide general checklists to help educators track their progress and trace interventions back to the research that justifies using them, which is similar to the rationale provided for the inclusion of each responsibility in a USAP. However, as with the Doing What Works templates, each Practice Guide must be accessed and connections made by each instructor between that instructor's needs and the Practice Guide's suggested practices. In addition, while the Practice Guides for mathematics and reading generally suggest that schools consider the availability and allocation of resources, they do not provide a detailed mechanism for enumerating resource needs. The solution path I propose in the next section could directly link curriculum standards to responsibilities for needs assessment and planning, providing a method and tool that would improve the current state of needs assessment and planning for educators.

7.2.3 A Solution Path for Transfer

A successful solution path for applying USAPs to user interface design and software systems, and for embodying that solution path in a web-based software tool, has already been described. The elements for a corresponding solution path in the domains of curriculum design and education systems are delineated in Figure 7-2 below.

It is important to note that the content for the checklists described in Figure 7-2(16) must be a collaborative effort on the part of experts in curriculum design and educational system design, and that it is a time-consuming process. For example, experts in usability and software architecture took weeks of person-hours over an eight-month period to develop the content for checklists of responsibilities to support three usability scenarios (Chapter 4). However, once the content has been developed, it can be re-used by an unlimited number of practitioners, and the practitioner can readily use the checklist tool (Chapter 5).

Software Engineering & HCI	**Education**
As experts in the domain of usability and software architecture, we have:	Experts in the domain of curriculum design and education systems could have:
1. Recurring usability scenarios (benefits are to the end-user) e.g., Canceling A Command	2. Recurring curriculum design scenarios (benefits are to the student) e.g., Scientific Investigation and Experimentation
3. Conditions under which the scenario may be applicable, e.g., • type of software system • environment in which system will function	4. Conditions under which the scenario may be applicable, e.g., • age of student • grade level • class size • learning domain
5. Enumeration of activities that the software architecture may have to perform to achieve that benefit, e.g., • when software has to record information • when software has to provide user feedback • how fast software has to work	6. Enumeration of elements that the educational system may have to include to achieve that benefit, e.g., • teaching time • teacher training • instructional goals • assessments • instructional materials • technology
7. Suggestions for implementation in the software architecture	8. Suggestions for implementation in the educational system
9. Rationale for why each activity supports its scenario	10. Rationale for why each element supports its scenario
11. Suggestions for enumerating activities, e.g., • time • scope • initiative • user feedback	12. Suggestions for enumerating these elements, e.g., • time • scope • division of responsibility • mechanism for feedback

Figure 7-2. Similar Solution Paths Across Disparate Domains

Software Engineering & HCI	**Education**
13. Sample re-occuring usability scenario: Canceling an Active 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.	14. Sample re-occurring curriculum design scenario: Scientific Investigation and Experimentation. Scientific progress is made by asking meaningful questions and conducting careful investigations. As a basis for understanding this concept and addressing content in the curriculum, students should develop their own questions and perform appropriate investigations.
15. Output • checklist of responsibilities that a software architecture must fulfill to support specific user interface concerns • software architects use this checklist to design software architecture that can support usability concerns • user interface design teams could use this checklist to review proposed user interface designs to determine whether they are feasible within the existing software architecture design	16. Output • checklist of responsibilities that an education system must fulfill to support specific curriculum design concerns • curriculum designers can use this checklist to make suggestions for educational system design or changes that can support curricular needs • curriculum designers could use this checklist to review proposed curricula to determine whether they are feasible within the existing educational system

Figure 7-2(cont). Similar Solution Paths Across Disparate Domains

7.2.4 Example Scenarios

To illustrate how analogous patterns could be created in education research, I will show three possible examples of recurring learning scenarios that could be used to create patterns in education, and some sample responsibilities that might apply for each scenario. For each of these scenarios (Figure 7-2, 14), users of the method would review each of the responsibilities and determine whether appropriate resources are available to support that responsibility in the context of the education system. My sample scenarios are drawn from state standards for K-12 curricula. Because these patterns would address the relationship between curriculum design and educational infrastructure, the actual responsibilities for such scenarios would have to be created in collaborative work between curriculum designers and other education experts, not by one person alone. Creating detailed content for the three End-User and four Foundation USAPs enumerated in the A-PLUS language (Chapter 4) required months of extensive collaborative work between usability and software architects. Creating analogous content for Curriculum-Supporting Educational Patterns (CSEPs) will also probably require an extensive collaborative process, although some transfer many turn out to be possible from existing sets of standards, practice guides, and curriculum design frameworks.

Example 1: Scientific Investigation and Experimentation
Scenario: Scientific progress is made by asking meaningful questions and conducting careful investigations. As a basis for understanding this concept and addressing content in the curriculum, students should develop their own questions and perform appropriate investigations (California Department of Education, 2000, p. 52).

Example 2: Reading Fluency and Systematic Vocabulary Development
Scenario: As the English language learner recognizes and produces the sounds of English, the student is simultaneously building vocabulary. Learning new labels for concepts, objects, and actions is a key building block for the integration of the language. The pathways in the English-Language Development (ELD) standards lead to the achievement of fluent oral and silent reading. Those pathways are created by building vocabulary and are demonstrated through actions and spoken words, phrases, and sentences and by transferring this understanding to reading (California Department of Education, 2002, p. 36).

Example 3: Symbolic Reasoning in Algebra
Scenario: Symbolic reasoning and calculations with symbols are central in algebra. Through the study of algebra, a student develops an understanding of the symbolic language of mathematics and the sciences. Students should use and know simple aspects of a logical argument (California Department of Education, 1999, pp. 38-40).

It is difficult to estimate how many such CSEPs might exist for curriculum and education systems development. In enumerating and creating USAPs, we have only begun to map the solution space for the software engineering domain with three completed USAPs in the A-PLUS language and a potential list of approximately two dozen more that focus only on a single-user at a desktop, perhaps analogous to the work of a single student in an individual learning activity. There are potentially more USAPs specific to group activities (such as social networks or collaborative work environments) which might be analogous to shared learning environments, and to mobile computing devices (like smart phones or Apple's iPad) which might be analogous to educational situations outside the classroom.

In addition to USAPs, which explore the relationship between usability and software architecture, more patterns could be developed in software engineering to address each of many other quality attributes (e.g., security, learnability), and also the tradeoffs between supporting conflicting quality attributes. Similarly, sets of patterns for educators could cover a variety of subjects (e.g., mathematics, science, reading, social sciences) for different grade levels and educational settings. In practice, user testing with curriculum designers and educators will be needed to determine the properly usable level of decomposition for using these patterns in education. Just as End-User USAPs share Foundational USAPs, End-User CSEPs for science, mathematics, reading and other areas of curriculum could share Foundational CSEPs that abstracted issues common to multiple subjects, e.g., learning subject-related vocabulary.

7.2.5 Challenges

7.2.5.1 Design Challenges

This method has the potential to be useful to creators of curricula at state, school district, and individual school levels, as well as in advocating for education research at the federal level. Standards boards at the state level and curriculum committees of school districts or charter schools could benefit, as could teachers constructing lesson plans. All of these educators are tasked with reconciling content requirements with complex factors of the educational environment. Commercial curriculum developers are another potential type of user. All of these kinds of users may respond differently to the patterns or the tool embodiment than the software architects who used A-PLUS Architect did. The parameterized responsibilities of the A-PLUS language may be easier for software engineers to understand and use than for educators. Although we know the software architects were able to use A-PLUS without researcher intervention, it will be necessary to conduct tests with education users to find whether they, too, can use the analogous patterns independently.

One interesting difference between using this method in software engineering and using it in education is the difference in user groups. In software engineering, USAPs were designed for use by software architects, but not UI designers, to anticipate the usability needs of UI designs. In the case of USAPs, expertise in usability was needed to enumerate the usability concerns, and expertise in software architecture was needed to couch the responsibilities and implementation suggestions in the professional language of software engineering. In education, the patterns would be used by curriculum designers and educators developing curricula to anticipate and communicate needs for resources to support the curricula they are developing. The scenarios and responsibilities would need to be understandable to educators and curriculum designers, and the resulting lists of needs phrased in an understandable way for these as well as administrators at state, district, and school levels. A great challenge for this extension is to achieve collaboration between the appropriate education experts to develop the education patterns in language that is understandable for users in different parts of the education domain.

Some taxonomies in education design may be difficult to represent in the current level of complexity of USAPs. As will be discussed in the next chapter, USAPs as currently written do not include a mechanism for evaluating design tradeoffs between competing needs. Educational design for the needs of different learning environments may want to include support for considering tradeoffs between competing factors in detailed curricula to support a learning scenario, or to weigh different curriculum strategies for the same scenario with different balances between the four learning environments. This capacity for examining tradeoffs might be especially valuable for those modifying existing curricula to fit changes in the learning environment, e.g., supporting Scientific Experimentation and Investigation in an environment currently heavily knowledge-centered but seeking to integrate more formative assessments and feedback, become more sensitive to students' existing knowledge structures, or foster feelings of community in the classroom.

Another risk to the feasibility of implementation is that curriculum developers may not perceive the process as difficult to manage without the checklists, so they may need to have the method's usefulness proven before they will be willing to try it. To address this risk, part of the task of extending this method to the education domain will involve working with curriculum developers to identify their perceived areas of difficulty. Surveying curriculum developers to determine whether any curricula they developed were not accepted or adopted, and why that acceptance or adoption was denied, could indicate what scenarios the education patterns should cover first.

7.2.5.2 Evaluation Challenges

An additional challenge lies in evaluating the impact of the method on student outcomes. In the work in software development, I was able to assess coverage and quality of architecture support for the usability concern and the software architects in the field were able to estimate the time they had saved by each need for change they uncovered while reviewing their software architecture design against several relevant USAPs. It is unclear whether standards boards, curriculum committees, individual teachers or commercial curriculum developers would be equally able to estimate their time savings, or other impacts of using the method on their design process. Although outcome evaluation is not strictly within the IES mandate for a Development/Innovation research project, some metric of success would be desirable in education research, as it is in software engineering research, to ease barriers to adoption in practice.

However, if we cannot measure the direct impact of the method and tool on student learning outcomes, we can measure its usefulness and usability for curriculum designers and educators and then gauge the likelihood of its acceptance in practice, using some of the same methods we used to measure those constructs for USAPs as embodied in A-PLUS Architect. In this model, a limited number of education patterns could be developed and embodied in a tool similar to A-PLUS Architect. User testing of this tool could be performed initially with a small number of curriculum designers and/or educators to see how education users would interact with both the tool and the content. In user testing of A-PLUS Architect, video data captured during the user tests allowed me to identify usability issues in the tool and understand how the users used the tool, enabling me to respond to users' needs in a redesign (Chapter 6); the same process would be both necessary and beneficial to making the technique work for education users. In addition to testing and refinement of in small numbers of user tests, TAM survey items could be used to allow the tool to be reviewed by larger numbers of education experts for perceived usability and usefulness. The results of TAM survey items would help to discover whether improvement was needed before getting larger numbers of users to try the education patterns, as well as to suggest possible directions for improvement.

7.3 Summary

For education researchers, this analogical approach offers a preliminary framework for applying my research in usability and software architecture to a different domain. The USAPs method has been shown to be helpful in the software engineering domain when suitable content is created in collaborative work between usability and software architecture experts. With experts in education collaborating to create the content for a similar curriculum design tool, this same method could be used to create patterns of responsibilities for specific issues in curriculum design that would include recommendations as needed for elements that reach beyond content standards, into the deep structure of the educational system. Existing curriculum design templates are deficient in integrating curriculum standards, research-based instructional guidance and interventions, and detailed needs for resources at the state, district, and local level. An enhanced template embodied in a software tool is proposed, capable of providing better integration of existing knowledge in curriculum design, research-based instructional guidance and interventions, and needs for resources that must be supported by the educational infrastructure. This tool would be useful to educators and other curriculum designers in their design work and provide clear and consistent enumerations of the resources needed to support curricula in practice.

In the next chapter, I will conclude by discussing the contributions of the work in this thesis to the fields of HCI, software engineering, and education research, and then suggest open questions for future research.

Chapter 8. Contributions, Open Questions, and Conclusions

8.1 Research Contributions

The work presented in Chapters 3, 4, 5, and 6 represents three phases of investigation of a single research problem: how to address architecturally-sensitive usability concerns early in the development process of complex software systems so as to prevent the need for costly late-stage changes in software architecture designs.

I posited two hypotheses in response to the research problem. The first hypothesis was that explicitly linking usability concerns to implementation decisions early in the software design process would enable software engineers to design better for basic usability features. Work in Chapters 3, 4, and 5 supports the first hypothesis. The second and related hypothesis was that if a technique could be shown to successfully support the first hypothesis, and packaged in a form that was useful and usable for software engineers, they would be likely to use this technique in practice. The second hypothesis is supported by Chapters 5 and 6.

The work described in this dissertation makes several contributions to HCI and software engineering. The effectiveness of USAPs in supporting the design of software architecture for interactive systems has been demonstrated in the controlled experiments in Chapter 3. A pattern language to represent USAPs has been developed that reuses shared responsibilities between usability scenarios and allows for multiple USAPs to be compressed into a shorter form for easier use (Chapter 4). An enhanced method and tool have been created and refined for the design of software architecture for interactive systems based on USAPs, and their effectiveness confirmed in supporting the use of USAPs in software architecture design (Chapters 5 and 6). This section reviews the several phases of this research and summarizes open questions raised by this work that may provide guidance to future researchers.

8.1.1 Validating Usability-Supporting Architecture Patterns

In Chapter 3, I investigated the value of Usability-Supporting Architecture Patterns (USAPs), applied without researcher intervention, as an approach to improving the consideration of specified usability concerns in software architecture design. Before the creation of USAPs (Bass & John, 2000), usability knowledge in HCI had not been packaged in a way that explicitly addressed the relationship between usability and software architecture. USAPs were designed to achieve better usability of systems through making more informed early software design decisions by identifying aspects of usability that are architecturally significant and then describing each aspect with a small scenario, a list of important software responsibilities, and possible implementation suggestions to satisfy the scenario.

USAPs seemed like a plausible approach to integrating usability into software architecture design. There was a successful intervention in the software architecture design process of the NASA MERboard -- a shared whiteboard project to support scientific planning for the Mars Exploration Rovers -- but that was a single case that required researcher intervention (Adams, John, & Bass, 2005). Participants in a conference tutorial on USAPs responded enthusiastically, but that was anecdotal evidence. There was no experimental validation of the efficacy of USAPs, and they had not been applied without researcher intervention.

Through controlled experiments in a laboratory setting, I validated the efficacy of USAPs in a software architecture design task. Software engineering graduate students achieved significantly better results in a software architecture design task using a paper-based version of USAPs without researcher intervention than they did using only a usability scenario, similar to the recommendations usability experts give to software engineers in professional practice. Even using only the scenario and the list of responsibilities from a USAP allowed them to achieve significantly better consideration of responsibilities and quality of solution than they could achieve on their own (Golden, John, & Bass, 2005a; Golden, John, & Bass, 2005b).

Through these controlled experiments, I demonstrated that Usability-Supporting Architecture Patterns (USAPs) are a useful technique for incorporating usability concerns into the software architecture design of interactive systems. Using these patterns can help software engineers to design software architecture that considers specific usability concerns they would not otherwise notice, and to produce higher quality software architecture designs than the type of usability scenarios that are currently used in software development practice.

8.1.2 Architecture Pattern Language for Usability Support

I had successfully demonstrated the usefulness of a USAP in an architecture design task in controlled experiments in a laboratory setting. Two open questions that followed were whether USAPs would be useful for professional software architects in an industry setting, and how multiple USAPs could be incorporated in a single software architecture design. An opportunity arose to investigate these questions in collaboration with corporate researchers at ABB, a global company specializing in power and automation systems. The ABB researchers were involved in plans to design a new product line architecture to integrate several existing products that measured ongoing forces in real time in manufacturing equipment and large engines. Prior to our involvement, the ABB business unit had decided on usability as a key quality attribute for their new product line. Based on our previous publications, the researchers had selected USAPs from among the research literature as the most promising approach to including usability in software architecture design.

The ABB research project afforded an opportunity to try using multiple USAPs in a single project. Three usability scenarios that we had not previously developed into USAPs were chosen by the ABB business unit as important for their needs, and we began to develop responsibilities for each of these in two separate teams, one at Carnegie Mellon and one at ABB. These three scenarios resulted in more than 100 responsibilities – an unwieldy number for use in practice – but on comparing the lists many commonalities became apparent. We abstracted these commonalities to create a pattern language: Architecture Pattern Language for Usability Support (A-PLUS). As represented in the A-PLUS language, only 33 responsibilities were needed to address the three USAPs. A-PLUS was a new way to describe USAPs in which the usability scenarios became End-User USAPs, while the abstractions of the commonalities, including the responsibilities and text-based implementation suggestions, became Foundational USAPs. The language combined End-User USAPs and Foundational USAPs into lists of responsibilities for software architects to review, with the capability to review the same responsibility for multiple usability scenarios to which it may apply.

In the Architecture Pattern Language for Usability Support (A-PLUS), we created a pattern language that allows multiple USAPs with common conceptual elements to be defined in terms of reusable sets of responsibilities. Software architects can use A-PLUS to apply patterns of responsibilities that experts in usability and software engineering have identified and defined for one usability scenario to multiple related usability scenarios, and they can be applied by an unlimited number of software developers to their software development projects. Experts in usability and software engineering can use A-PLUS to author usability scenarios and lists of responsibilities that are common to the usability scenarios. The pattern language enables compression and re-use of responsibilities to support their use in software architecture design practice. The language is couched in terms that allow for programmatic combination of the scenarios and the responsibilities to support scalability and automated integration with a software tool to embody the language. The A-PLUS pattern language arose from collaborative work between me, Bonnie John, and Len Bass at Carnegie Mellon, and Pia Stoll, Fredrik Alfredsson, and Sara Lövemark at ABB Corporate Research. It was truly a group effort and is a collective contribution.

8.1.3 A-PLUS Architect

The A-PLUS pattern language produces a representation of USAPs that separates End-User USAPs from Foundational USAPs and allows for more uniform and reusable authoring of further USAPs. The elements in this representation can be combined to form lists of responsibilities and implementation suggestions for software architects to use in architecture design practice. End-User USAPs and Foundational USAPs are not, however, usable for software architects until they have been combined to form lists of responsibilities. The A-PLUS pattern language is designed to be used in conjunction with a software tool that embodies the language and presents it in a form that software architects can use. Simultaneously with the development of the A-PLUS language, I designed a browser-based tool, called A-PLUS Architect, to embody the language and deliver it to software architects.

The purpose of A-PLUS Architect was to allow the software architects to review their software architecture designs against the checklist of responsibilities included in one or more USAPs, determine if each responsibility was addressed, and if it was not, decide whether to alter the architecture design. The A-PLUS language's internal concept of dividing USAPs into Foundational USAPs and End-User USAPs was hidden from the user as extraneous information that was not needed by users in order to use the tool. Automated checkboxes enforced the hierarchical structure of the sets of responsibilities and provided a reminder to the architects to consider every responsibility included in each USAP. Lists of text-based implementation suggestions for each responsibility were included as optional information that users could view on demand. A prototype of my tool design was implemented by the researchers at ABB.

I conducted user tests of the A-PLUS Architect prototype with software architects at ABB's corporate development site. Two paired architects working on the new product line system design used the prototype for six hours in a single day. They uncovered fourteen major usability issues in their preliminary software architecture design while using the A-PLUS Architect prototype. The lead software architect reviewed the list of issues after the user tests and estimated that the single day of user testing with two software architects (i.e., two person-days) had saved twenty-five days of development time further down the road, for a return on investment (ROI) of 25:2. The software architects also made many favorable and helpful comments about the user interface.

Following the positive results of the user tests, I iterated on the design of A-PLUS Architect. Given the predominantly smooth interaction observed during the user test and confirmed by comments by the users, I retained the simplicity of the hierarchical checklist format. However, the redesign addressed the few usability issues that were uncovered in the user tests, e.g., allowing users to record process notes, and some that were known issues that had been deferred until after the first prototype was tested for usefulness, e.g., data persistence. I showed the iterated design of A-PLUS Architect online to professional software architects and used a survey instrument, the Technology Assessment Model, to measure their perceptions of the redesigned tool's usefulness and usability, and whether they would intend to use the tool if it became available to them. Forty-three software architects examined the design and completed the survey. On average, the responses to the revised design of A-PLUS Architect were entirely positive as indicated by the TAM items. In the TAM model, perceived ease of use, perceived usefulness, and attitude are predictive of intention to use. The strongest results of the survey instrument were those for attitude and perceived usefulness, while results for perceived ease of use and intention to use were slightly lower. User acceptance is often a problem in information technology; but from the results of this model, we can surmise that a mildly positive result would correspond to some willingness trying the tool. And if users did achieve positive results with A-PLUS Architect as did the architects at ABB, they would be motivated to use it again where relevant.

I have shown A-PLUS Architect to be a useful and usable prototype tool that embodies this technique. I have designed and demonstrated the usability and usefulness of this prototype tool. I have produced a second iteration of the design of the tool and assessed the likelihood of its acceptance by professional software architects. A-PLUS Architect enables software architects to apply USAPs to their software architecture designs without researcher intervention, and thereby identify specific areas where their designs need to accommodate the usability concerns that have been identified as important for their software projects.

8.1.4 Extending USAPs Conceptually into Education Research

As an outcome of the work on incorporating usability concerns into software architecture design, this thesis also makes a contribution to education research. I have described an analogy between user interface design and software architecture in the software engineering domain, and curriculum design and educational system infrastructure in the education domain. This analogy is a first step toward applying the method and tool described herein for HCI and software architecture to the education problem of curriculum design in the context of educational systems.

I have shown that the USAPs method is helpful for bridging the gap between usability and software architecture in the software engineering domain, and that it can be beneficially applied to software architecture design with suitable content created in collaborative work between usability and software architecture experts. With experts in education collaborating to create patterns for an analogous method and tool, the USAPs method could be used to create patterns of responsibilities for specific issues in curriculum design that would include recommendations as needed for elements that reach beyond content standards, into the deep structure of the educational system. These patterns developed specifically for the education domain could be termed Curriculum-Supporting Educational Patterns (CSEPs).

Existing curriculum design templates are deficient in integrating curriculum standards, research-based instructional guidance and interventions, and detailed needs for resources at the state, district, and local level. I have proposed an enhanced template embodied in a pattern language and a software tool, capable of providing better integration of existing knowledge in curriculum design, research-based instructional guidance and interventions, and the needs for resources that must be supported by the educational infrastructure. This pattern language and tool could be useful to educators and other curriculum designers in their design work and would help to provide clear and consistent enumerations of the resources needed to support curricula in practice.

8.2 Open Questions

The work in this thesis answers many questions, as described above, but raises more if the work I have discussed is to be taken to its full potential. In this section, I discuss open questions raised, both for future research and future production.

A-PLUS Architect has been shown to be useful and usable in user tests of the prototype, and software architects have reacted positively to a description of an iterated design. Further research on USAPs, the A-PLUS language, and conceptual extensions could proceed in several different directions discussed here. I also discuss some production questions of proceeding to take A-PLUS Architect from a tool prototype and a design iteration to best practices use in software development.

8.2.1 Research Questions

8.2.1.1 Usefulness of Diagrams in Solution Suggestions for USAPs

One interesting set of findings in the controlled experiments described in Chapter 3 concerned the effect of including or omitting diagrammatic solution suggestions in the USAPs participants used to help them redesign software architecture. In those studies, participants who had solution suggestions including UML diagrams did not perform significantly better than those without them, either in coverage of responsibilities or in quality of solution. A detailed analysis of video recordings of five participants performing the design task revealed that all who had diagrams in a sample solution looked at the diagrams, but we know from analyzing their task performance that the diagrams did not help those participants achieve better coverage of responsibilities or higher quality of solution. The open question is, why did the diagrams not help those participants who had them?

This question is important because software architecture patterns as presently constructed are largely expressed with diagrams. If, as my research on USAPs suggests, patterns consisting entirely of written responsibilities, rationales for including each responsibility in the pattern, and implementation suggestions for designing software architecture to address the responsibility, and lacking any diagrams whatsoever, are useful and usable for software architects in the design process, then why did the diagrams not improve their performance even more than the responsibilities alone?

One possible answer to this question is that a single example expressed in diagrams, as provided in the empirical studies, was not sufficient to impart the additional understanding that users would gain from having multiple examples. Investigation into this question would explore whether multiple sample solutions expressed in diagrams could produce the significant performance gain over responsibilities alone that the single diagrammatic solution failed to produce.

A second possibility is that the level of the particular USAP used in the controlled experiments, Canceling an Active Command, may not have been well matched to the level of the UML representations we used in the experimental materials. Research to investigate this question would examine the impact of different levels of solution suggestion diagrams to USAPs to determine whether a kind of pictorial representation should be added to improve their usefulness.

8.2.1.2 Implications of Creating More USAPs

At present the A-PLUS pattern language, described in Chapter 4, includes three End-User USAPs and four Foundational USAPs. The End-User USAPs are derived from usability scenarios: User Profiles, Environment Configuration, and Alarms and Events. The Foundational USAPs were abstracted from commonalities discovered between the End-User USAPs when we developed the language, and fell into four categories: Authorization, Authoring, Execution with Authored Parameters, and Logging. The list of potential End-User USAPS is therefore the same as the list of potential usability scenarios. This list is not complete but those we have enumerated currently numbers roughly twenty (Appendix I). An open research question is what the potential number of Foundational USAPs might be. If the body of USAPs were enlarged, how would the proportion of End-User and Foundational USAPs change?

The proportion of Foundational USAPs to End-User USAPs and the number of potential Foundational USAPs yet to be discovered and developed are questions of interest because the responsibilities included in End-User USAPs come directly from the Foundational USAPs. Simply put, adding more Foundational USAPs would mean adding more responsibilities for software architects to review and understand, extending the time required to use the method in software architecture design. Since the architects at ABB estimated their cost savings from the user tests in time, we know that time is valuable to them, so an important question in adding more Foundational USAPs would be how changes in time overhead for users would affect the benefit they received from using USAPs.

The number of Foundational USAPs could also affect the time and effort needed to create more End-User USAPs. Foundational USAPs are the building blocks that End-User USAPs use to compile the majority of responsibilities required to support their scenarios. As such, if a small number of Foundational USAPs sufficed to support most End-User USAPs, their re-use would make it much easier for authors to create new End-User USAPs. On the other hand, the process of creating an End-User USAP involves determining which Foundational USAPs include commonalities shared by the End-User USAP, i.e., usability scenario, and then reviewing each of the individual responsibilities in each Foundational USAP used by the scenario to understand how to use directly or specialize each responsibility for the usability scenario in question, so more Foundational USAPs could mean more responsibilities for USAP authors to review during the creation process, adding time and complexity to an already lengthy and complex process.

8.2.1.3 Scalability of the USAPs Approach

More questions of scale arise when we consider applying the USAPs approach to include patterns for more software qualities than usability alone. USAPs are Usability-Supporting Architecture Patterns, designed to help software architects address usability concerns in software architecture design. Usability is only one of many software qualities for which such patterns could be developed. As discussed in Chapter 2, various taxonomies enumerate a wide range of other software quality characteristics. In addition to usability, software engineering recognizes qualities of availability, modifiability, performance, security, testability, reliability, efficiency, portability, and many others (International Standards Organization, 2001; Bass et al., 2003, pp. 78-93). The number is inexact since various taxonomies categorize different qualities as overarching, overlapping, and subsidiary. Each of the many quality attributes could have its own set of Supporting Architecture Patterns, just as usability has USAPs. The open question is whether the approach to using USAPs in software architecture design would scale up when more qualities and more patterns are added. What is the relationship between the number of patterns and their effectiveness in practice?

Software architects at ABB took six hours to review three USAPs in user tests (Chapter 5). The number of possible USAPs is at least twenty. We do not yet know how many other Quality-Supporting Architecture Patterns (QSAPs) there might be, but it is easy to see how the number could easily climb into the hundreds if several software qualities were addressed. A much larger number of patterns might make it impractical for software architects to use the same approach unless they could choose up front which patterns to target. Since time was shown to be valuable to the architects at ABB, we know that an important question of scaling up to address more software quality attributes, as with the question of the total number of Foundational USAPs to End-User USAPs, is the increase in time overhead for users.

Adding patterns for more quality attributes also raises the question of how the A-PLUS language could or should be expanded to include support for design tradeoffs between quality attributes with potentially conflicting responsibilities, e.g., usability and security, or security and performance. Tradeoffs between quality attributes would be a new category of information added to the A-PLUS language. Research to address including tradeoffs in USAPs would be a complex problem requiring collaborative input between experts in whatever quality attributes were being considered as well as experts in software architecture.

8.2.1.4 Modification of USAPs by Users

In the course of the work described in this thesis, USAPs have gone through three representational changes. The first representational change was from the earliest versions that required researcher intervention to a paper-based form that represented a USAP as a usability scenario, a list of responsibilities, and a sample solution with UML diagrams, and enabled USAPs to be used without researcher intervention. The second representational change was from the paper-based USAPs to the A-PLUS pattern language with its separation of End-User USAPs, consolidation of Foundational USAPs, and text-based implementation suggestions. The third representational change from the pattern language to its embodiment in the A-PLUS Architect tool, which hides the separation of Foundational and End-User USAPs while using it to create the structure of responsibilities, and which encourages attention to responsibilities through hierarchical checklists.

Throughout this process, the content of USAPs was determined entirely by a small number of researchers. An open question is how, and how much, we can or should shift control during the further evolution of USAPs into the hands of software architects. Should users be allowed to add more responsibilities to USAPs? What would be the impact of their additions on the A-PLUS language?

When we authored the three End-User USAPs and four Foundational USAPs in the A-PLUS pattern language, usability and software architecture experts collaborated to review each individual responsibility in each of the Foundational USAPs and then decided whether it applied to each End-User USAP that might use it. There is no guarantee that users would follow the same rigorous process, so they might add misconceptions that would cause design errors. On the other hand, if users find in practice that they want to add more responsibilities, in effect either extending a Foundational USAP, it may indicate that the authors of USAPs have missed something. One approach that would allow users to make desired changes without compromising the integrity of the A-PLUS language, which mitigates the risk of propagating misconceptions forward, would be to have a gating committee of experts who would review suggested changes to USAPs for approval to add them to the A-PLUS language. In addition to the logistical problems posed by creating a reviewing body, however, an open question remains as to whether the quality of the A-PLUS language is best served by strict researcher control or by allowing for the flexibility of natural evolution.

8.2.1.4 USAPs and a Tool for Curriculum Design

Just as the A-PLUS language was created to support USAPs in software architecture design, a pattern language of learning scenarios and responsibilities, Curriculum-Supporting Educational Patterns (CSEPs), could be developed to support curriculum design in the education domain, fulfilling the promise of the research concepts proposed in Chapter 7. Such a pattern language would require collaboration among experts in curriculum design, education systems design, and education research. Experts in these fields would be needed for collaborative creation of patterns of responsibilities, which would be disseminated using a CSEPs tool similar to the A-PLUS Architect for using USAPs. In theory it would be possible to use the A-PLUS Architect tool design with different domain-specific content. However, the tool has only been tested with one type of users – software architects – and different usability results might occur with a different kind of users. Therefore it would be necessary to prototype, test, and redesign the tool as needed for use by educators and curriculum developers. Creating a language of CSEPs that is useful and usable for the education community, and developing a tool for their successful dissemination in education and curriculum design practice, is a non-trivial set of research problems.

8.2.2 Production questions

8.2.2.1 Adding USAPs to the A-PLUS Language

In addition to open research questions, the work in my dissertation leaves open the question of production for the next steps to be taken with the products of the completed research. As a next step for the A-PLUS pattern language, more End-User USAPs could be developed that share commonalities already addressed by the existing Foundational USAPs. More Foundational USAPs might be developed, or the existing Foundational USAPs extended, as needed to accommodate conceptual commonalities in new End-User USAPs yet to be developed. The creation of new Foundational USAPs and/or End-User USAPs is a question of production as well as of research. The work would require collaboration between experts in usability and software engineering.

8.2.2.2 Integration with Existing Processes

One clear direction would be to further improve, develop, and disseminate A-PLUS Architect as a tool for use in professional software development practice. However, dissemination might be more successful if A-PLUS Architect were incorporated into existing practices. Organizations already have tools that they use in the requirements process. Software engineers have tools they use in the architecture design process. Incorporating A-PLUS Architect and A-PLUS Requirements into existing software development processes would lower barriers to acceptance and provide a larger body of active users whose experiences could suggest productive avenues for specific improvements.

8.2.2.3 A-PLUS Author and A-PLUS Requirements

Perhaps the most positive step toward lowering the barriers to developing further USAPs and integrating them into professional practice is to develop the two other software tools envisioned in an A-PLUS suite of tools. One of the additional tool concepts is A-PLUS Requirements, which would be used to facilitate incorporating USAPs into software development practice at the organizational level. A-PLUS Requirements would helps software development organizations to identify the USAPs that are most relevant to their projects by selecting relevant usability scenarios from among the existing body of USAPs. A-PLUS Requirements, like A-PLUS Architect, should be developed in a way that is compatible with inclusion in existing requirements processes and the developers or development organizations that use them.

Another tool concept is for A-PLUS Author, a tool to assist in the authoring of more USAPs in the A-PLUS pattern language. The A-PLUS language thus far was created laboriously using Microsoft Word, an approach which is not scalable. Developing A-PLUS Author would be a positive step for further expansion of the A-PLUS pattern language. The A-PLUS pattern language has been created and demonstrated for four Foundational USAPs: Authorization, Authoring, Execution with Authored Parameters, and Logging, and three End-User USAPs with conceptual commonalities: User Profiles, Environment Configuration, and Alarm and Events. The process of creating these elements took six months, a production barrier that might prove insuperable to future authors.

Developing A-PLUS Author could make collaboration between usability and software architecture experts easier, lowering the barriers to creating additional USAPs. A-PLUS Author could also make it easier for experts to develop responsibility-based architectural patterns to investigate relationships between other software quality attributes (e.g., security) and software architecture. Developing these patterns would require collaborative work between experts in those other quality attributes and software architecture, but the same tool could be used to facilitate the collaborative work as with USAPs.

8.3 Conclusions

Taken together, the work described in this dissertation has laid a foundation for the integration of usability knowledge into best practice in software engineering development, and the extension of an integrative method into curriculum design in education.

Usability-Supporting Architecture Patterns have undergone many changes in representation in the decade since they were first conceived. The first representations were useful but required researcher intervention to be usable. The paper-based representations used in experiments were useful and usable without researcher intervention, but they contained possibly unnecessary elements and were not designed in a way that supported combining them. The A-PLUS language representation included fewer elements and compressed and redistributed information in a way that was more scalable, but the patterns expressed in this language were not directly usable by the intended user. The A-PLUS Architect tool representation supported combining multiple USAPs in a single solution and rendered the A-PLUS pattern language usable and useful for software architects. The A-PLUS language and A-PLUS tool representations embody principles that are sufficiently developed to allow the concepts to be extended to other software qualities and other research domains. Many questions remain as to how the current representations may evolve in the hands of more end-users and researchers, both in software engineering and education research.

The natural evolution of a successful method is that it grows beyond the boundaries originally conceived by its creators. If the USAPs approach is to be successful in the long term, more patterns will need to be created by more experts interested in their creation and their use. Future research should be directed toward

- encouraging and assessing the impact of evolution in the A-PLUS language over time,
- incorporating useful contributions as they arise, and
- investigating effective mechanisms to apply and disseminate more and different types of content, created by more and different types of authors collaborating across a variety of knowledge domains.

References

1. Adams, D. A., Nelson, R. R., & Todd, P. A. (1992). Perceived Usefulness, Ease of Use, and Usage of Information Technology: A Replication. *MIS Quarterly, 16*(2), 227-247.

2. Adams, R. J., John, B. E., & Bass, L. (2005). Applying general usability scenarios to the design of the software architecture of a collaborative workspace. In A. Seffah, J. Gulliksen & M. Desmarais (Eds.), *Human-Centered Software Engineering — Integrating Usability in the Software Development Lifecycle* (Vol. 8): Springer Netherlands.

3. Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction.*: Oxford University Press.

4. Alexander, C. (1979). *A Timeless Way of Building*: Oxford University Press.

5. Barbacci, M. R., Ellison, R., Lattanze, A. J., Stafford, J. A., Weinstock, C. B., & Wood, W. G. (2003). *Quality Attribute Workshops (QAWs), Third Edition*: Carnegie Mellon University / Software Engineering Institute, Technical Report CMU/SEI-2003-TR-016).

6. Barbacci, M. R., Klein, M. H., Longstaff, T. A., & Weinstock, C. B. (1995). *Quality Attributes*: Carnegie Mellon University / Software Engineering Institute, Technical Report CMU/SEI-95-TR-021).

7. Bass, L., & John, B. E. (2000). *Achieving usability through software architectural styles.* Paper presented at the CHI '00 extended abstracts on Human factors in computing systems, The Hague, The Netherlands.

8. Bass, L., & John, B. E. (2003). Linking usability to software architecture patterns through general scenarios. *Journal of Systems and Software, 66*(3), 187-197.

9. Bass, L., & John, B. E. (2001). Supporting Usability Through Software Architecture. *Computer, 34,* , October 2001, 113-115.

10. Bass, L., John, B. E., & Kates, J. (2001). *Achieving Usability Through Software Architecture*. Pittsburgh, PA, USA: Carnegie Mellon University/Software Engineering Institute, Technical Report No. CMU/SEI-TR-2001-005.

11. Bass, L., John, B. E., Juristo, N., & Sanchez-Segura, M.-I. (2004). Usability and Software Architecture, *26th International Conference on Software Engineering, ICSE 2004*. Edinburgh, Scotland.

12. Bass, L., Clements, P., & Kazman, R. (1999). *Software Architecture in Practice*: Addison-Wesley.

13. Bass, L., Clements, P., & Kazman, R. (2003). *Software Architecture in Practice* (2nd ed.): Addison-Wesley.

14. Bhavnani, S. K., Peck, F. A., & Reif, F. (2008). Strategy-Based Instruction: Lessons Learned in Teaching the Effective and Efficient Use of Computer

Applications. *ACM Transactions on Computer-Human Interaction (TOCHI), 15*(1).

15. Bielaczyc, K. (2006). Designing Social Infrastructure: Critical Issues in Creating Learning Environments With Technology. *The Journal of the Learning Sciences, 15*(3), 301-329.

16. Bransford, J. D., Brown, A. L., & Cocking, R. R. (Eds.). (2000). *How People Learn: Brain, Mind, Experience, and School*. Washington, DC, USA: National Academy Press.

17. Buschmann, F., Meunier, R., Rohnert, H., & Sommerlad, P. (1996). *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*: Wiley.

18. California Department of Education (2010, February 05). Curriculum Frameworks & Instructional Materials. Retrieved 3/25/10, from http://www.cde.ca.gov/ci/cr/cf/index.asp

19. California Department of Education (2002). *English-Language Development Standards for California Public Schools*, *Kindergarten through Grade Twelve*, California Department of Education.

20. California Department of Education (1999). *Mathematics Content Standards for California Public Schools*, *Kindergarten through Grade Twelve*, California Department of Education.

21. California Department of Education (2000). *Science Content Standards for California Public Schools, Kindergarten Through Grade Twelve*, California Department of Education.

22. Clements, P., & Northrop, L. (2001). *Software Product Lines: Practices and Patterns*: Addison Wesley.

23. Davis, F. D. (1989). "Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology." *MIS Quarterly* 13(3): 319-340.

24. Davis, F. D., & Bagozzi R. P. (1989). "User Acceptance of Computer Technology: A Comparison of Two Theoretical Models." *Management Science* 35(8): 982-1003.

25. Davis, F. D., Bagozzi, R. P., & Warshaw, P. R. (1989). User Acceptance of Computer Technology: A Comparison of Two Theoretical Models. *Management Science, 35*(8), 982-1003.

26. Dearden, A., & Finley, J. (2006). Pattern Languages in HCI: A critical review. *Human-Computer Interaction, 21*(1), 49-102.

27. Degani, A., & Wiener, E. L. (1990). *Human Factors of Flight-Deck Checklists: The Normal Checklist*: National Aeronautics and Space Administration, NASA Contractor Report 177549.

28. Degani, A., & Wiener, E. L. (1993). Cockpit Checklists: Concepts, Design, and Use. *Human Factors, 35*(2), 28-43.

29. Duncker, K. (1945) On problem solving. Psychological Monographs, 58(5). (Whole No. 270)

30. Engel, S. E., & Granda, R. E. (1975). *Guidelines for Man/Display Interfaces*. (IBM Technical Report TR00-2720)

31. Folmer, E., & Bosch, J. (2003). *Usability patterns in Software Architecture.* Paper presented at the 10th International Conference on Human-Computer Interaction, Crete.

32. Folmer, E., & Bosch, J. (2004). Architecting for usability: a survey. *Journal of Systems and Software 70*(1-2), 61-78.

33. Folmer, E., & Bosch, J. (2008). Experiences with Software Architecture analysis of Usability *International Journal of Information Technology and Web Engineering 3*(4), 1-29.

34. Folmer, E., Gurp, J. v., & Bosch, J. (2003). A Framework for capturing the Relationship between Usability and Software Architecture. *Software Process: Improvement and Practice, 8*(2), 67-87.

35. Folmer, E., van Gurp, J., & Bosch, J. (2004). *Scenario-Based Assessment of Software Architecture Usability.* Paper presented at the 9th IFIP Working Conference on Engineering for Human-Computer Interaction, Jul 2004, Hamburg, Germany.

36. Foley, J. P., Jr. (1972). *Task Analysis for Job Performance Aids and Related Training.* Paper presented at the Conference on Uses of Task Analysis in the Bell Systems, Hopewell, NJ.

37. Functional fixedness. (n.d.) In Wikipedia. Retrieved May 7, 2010, from http://en.wikipedia.org/wiki/Functional_fixedness.

38. Gawande, A. (2009). *The Checklist Manifesto*. New York, NY: Metropolitan Books.

39. Golden, E., John, B. E., & Bass, L. (2005). *The value of a usability-supporting architectural pattern in software architecture design: a controlled experiment.* Paper presented at the 27th International Conference on Software Engineering (ICSE 2005), St. Louis, MO, USA.

40. Golden, E., John, B. E., & Bass, L. (2005). *Quality vs. quantity: comparing evaluation methods in a usability-focused software architecture modification task.* Paper presented at the International Symposium on Empirical Software Engineering (ISESE 2005), Noosa Heads, Queensland, Australia.

41. Halawi, L., & McCarthy, R. (2007). Measuring Faculty Perceptions of Blackboard Using The Technology Acceptance Model. *Issues in Information Systems, VIII*(2), 160-165.

42. Halawi, L., & McCarthy, R. (2008). Measuring Students Perceptions of Blackboard Using The Technology Acceptance Model: A PLS Approach. *Issues in Information Systems, IX*(2), 95-102.

43. Institute of Education Sciences (2010). *REQUEST FOR APPLICATIONS: Education Research Grants,* CFDA Number: 84.305A.

44. Institute of Education Sciences (2010). What Work Clearinghouse. Retrieved 4/30/10, from http://ies.ed.gov/ncee/wwc/

45. International Standard ISO/IEC 9126. *Software engineering - Product quality*. International Organization for Standardization / International Electrotechnical Commission, Geneva, 2001.

46. John, B. E., & Bass, L. (2001). Usability and software architecture. *Behaviour & Information Technology, 20*(5), 329-338.

47. John, B. E., Bass, L., Sanchez-Segura, M.-I., & Adams, R. J. (2004). *Bringing Usability Concerns to the Design of Software Architecture.* Paper presented at the 9th IFIP Working Conference on Engineering for Human-Computer Interaction and the 11th International Workshop on Design, Specification and Verification of Interactive Systems, July 11-13, 2004, Hamburg, Germany.

48. Juristo, N., Windl, H., & Constantine, L. (2001). Introducing Usability. *IEEE Software,* 20-21.

49. Juristo, N., Lopez, M., Moreno, A., & Sanchez-Segura, M.-I. (2003). *Improving software usability through architectural patterns.* Paper presented at the ICSE 2003 Workshop on Bridging the Gaps Between Software Engineering and Human-Computer Interaction, Portland, Oregon, USA.

50. Juristo, N., Moreno, A., & Sanchez-Segura, M.-I. (2007). Guidelines for Eliciting Usability Functionalities. *IEEE Transactions on Software Engineering, 33*(11), 744-758.

51. Juristo, N., Moreno, A., Sanchez-Segura, M.-I., & Baranauskas, M. C. C. (2007). *A Glass Box Design: Making the Impact of Usability on Software Development Visible.* Paper presented at INTERACT 2007.

52. Kan, M., & Cheng, D. (2007, August 20-22, 2007). *An Empirical Research on Online Infomediary Based on Extension of the Technology Acceptance Model (TAM2).* Paper presented at the 14th International Conference on Management Science & Engineering, Harbin, China.

53. Kazman, R., Gunaratne, J., & Jerome, B. (2003). Why Can't Software Engineers and HCI Practitioners Work Together? In C. Stephanidis & L. Erlbaum (Eds.), *Human-Computer Interaction Theory and Practice*, Elsevier.

54. Klein, M. H., Kazman, R., Bass, L., Carriere, S. J., Barbacci, M. R. & H.F. Lipson (1999). *Attribute-Based Architecture Styles*. Kluwer.

55. Moon, J.W., & Kim, Y.G. (2001). *Extending the TAM for a World-Wide-Web Context*. Journal of Information & Management, 38, pp. 217-230.

56. New York State Education Department, (2009, October 1). New York State Learning Standards and Core Curriculum. Retrieved March 25, 2010, from http://www.emsc.nysed.gov/ciai/cores.html#MST.

57. Nielsen, J. (1994). Heuristic evaluation. In Usability Inspection Methods (pp. 25-62). New York, NY, USA: John Wiley & Sons, Inc.

58. Pellegrino, J. W. (2006). Rethinking and Redesigning Curriculum, Instruction, and Assessment: What Contemporary Research and Theory Suggests (Commissioned Paper): National Center on Education and the Economy for the New Commission on the Skills of the American Workforceo. Document Number)

59. Porter, A. A., & Votta, L. G. (1994). *An experiment to assess different defect detection methods for software requirements inspections.* Paper presented at the 16th International Conference on Software Engineering.

60. Porter, A. A., & Votta, L. G. (1998). Comparing Detection Methods For Software Requirements Inspections: A Replication Using Professional Subjects *Empirical Software Engineering, 3*(4), 355-379.

61. Porter, A. A., Votta, L. G., & Basili, V. R. (1995). Comparing detection methods for software requirements inspections: a replicated experiment. *IEEE Transactions on Software Engineering, 21*(6), 563-575.

62. Prechelt, L., Unger, B., & Schmidt, D. C. (1997). *Replication of the First Controlled Experiment on the Usefulness of Design Patterns: Detailed Description and Evaluation* (Technical Report No. wucs-97-34). St. Louis, MO: Washington University.

63. Prechelt, L., Unger-Lamprecht, B., Philippsen, M., & Tichy, W. F. (2002). Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Transactions on Software Engineering, 28*(6), 595-606.

64. National Cancer Institute. Research-Based Web Design & Usability Guidelines. from http://www.usability.gov/guidelines/.

65. Rubin, J. (1994). *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*: John Wiley & Sons.

66. Smith, S. L., & Mosier, J. N. (1984). *Guidelines for Designing User Interface Software* Hanscom Air Force Base, MA: USAF Electronic Systems Division, Technical Report ESD-TR-86-278).

67. Stoll, P., John, B. E., Bass, L., & Golden, E. (2008). *Preparing Usability Supporting Architectural Patterns for Industrial Use.* International Workshop on the Interplay between Usability Evaluation and Software Development, Pisa, Italy.

68. Stoll, P., Wall, A., & Norstrom, C. (2008). *Guiding Architectural Decisions with the Influencing Factors Method.* Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA), Vancouver, Canada.

69. Sun Microsystems.(2002) *Java BluePrints: Model-View-Controller.* Retrieved from http://java.sun.com/blueprints/patterns/MVC-detailed.html.

70. Tidwell, J. (2006). *Designing Interfaces: Patterns for Effective Interation Design*. Sebastopol, CA: O'Reilly Media.

71. Tidwell, J. (2004). UI Patterns and Techniques. from http://time-tripper.com/uipatterns/.

72. U.S. Department of Education (2010). Doing What Works. Retrieved 4/30/10, from http://dww.ed.gov.

73. Van Duyne, D. K., Landay, J. A., & Hong, J. I. (2002). *The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience*. Boston, MA: Addison-Wesley.

74. Venkatesh, V., & Davis, F.D. (2000). "A Theoretical Extension of the Technology Acceptance Model: Four Longitudinal Field Studies." *Management Science* 46(2): 186-204.

75. Venkatesh, V., & Morris, M.G. (2003). "User acceptance of information technology: Toward a unified view." *MIS Quarterly* 27(3): 425-478.

76. Wiggins, G. & J. McTighe. Understanding by Design, 2nd edition, Association for Supervision and Curriculum Development, 2005.

77. Wu, M.-Y., Chou, H.-P., Weng, Y.-C., & Huang, Y.-H. (2008, December 9-12, 2008). *A Study of Web 2.0 Website Usage Behavior Using TAM 2.* Paper presented at the IEEE Asia-Pacific Services Computing Conference, Yilan, Taiwan.

Appendices

Appendix A: Usability Scenario of Cancel Included in Training Document for Empirical Studies.

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.

Appendix B: General Responsibilities of Cancel Included in Training Document for Empirical Studies.

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. 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).

Appendix C: "Before" and "After" Diagrams from Cancel Sample Solution Included in Training Document for Empirical Studies.

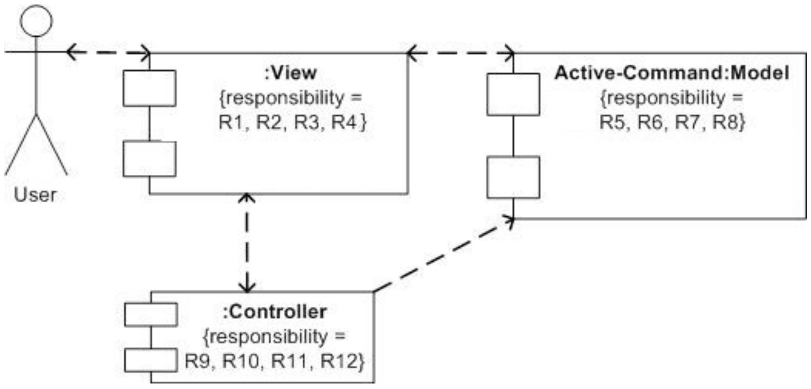


Figure 1. MVC before Cancel is added and cancellation responsibilities allocated

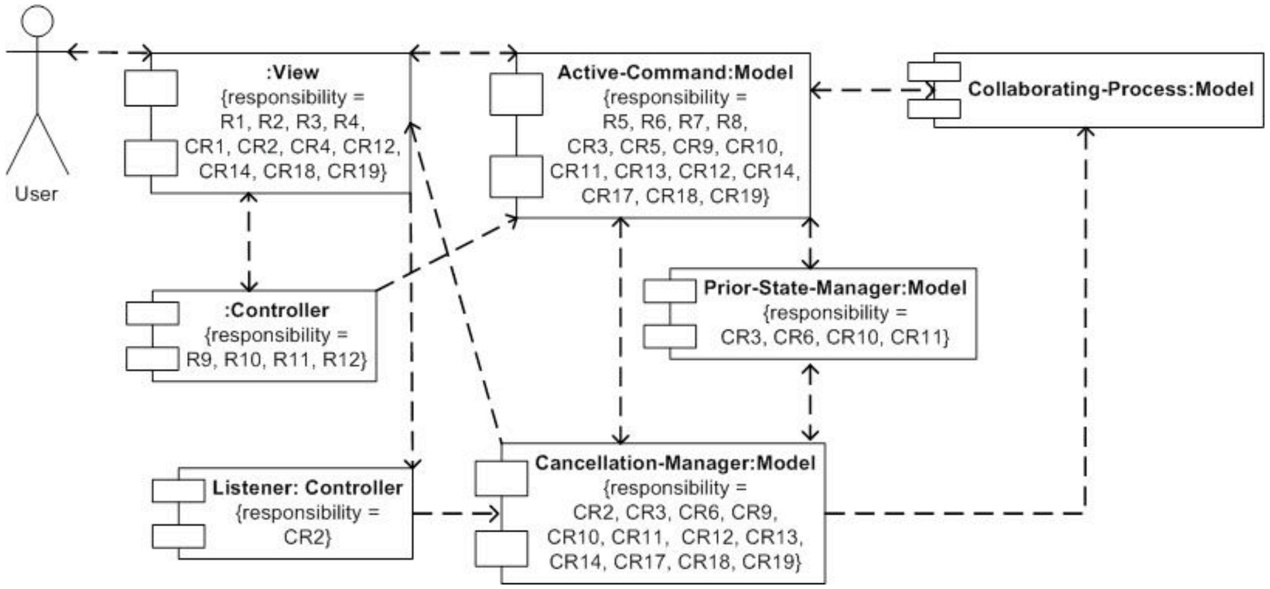


Figure 2. MVC after Cancel is added and cancellation responsibilities allocated

Appendix D: Content of Training Documents Given to Participants in Empirical Studies.

Training Document

The following is an architecturally-sensitive usability scenario. Usability scenarios are considered to be architecturally-sensitive if it is difficult to add the scenario to a system after the architecture has been designed.

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.

Appendix E: Content of Training Document Given to Scenario-plus-General-Responsibilities Participants in Empirical Studies.

Training Document

The following is an architecturally-sensitive usability scenario. Usability scenarios are considered to be architecturally-sensitive if it is difficult to add the scenario to a system after the architecture has been designed.

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.

Checklist of Cancellation Responsibilities

CR = Cancellation Responsibility.

CR1 – CR19 enumerate the responsibilities that should be fulfilled by any implementation of a cancel command. These are intended to be general responsibilities, covering a range of conditions, so many of them contain conditionals (if statements). Not all the responsibilities are relevant to every possible scenario. In a given scenario, the task and/or the system environment may make some cancellation responsibilities relevant, and some irrelevant, for that scenario.

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.

a. For example, if the user pressed a cancel button, changing the color of the button will be seen.

b. 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.

a. If the estimate is between 1 and 10 seconds, changing the cursor shape is sufficient.
b. If the estimate is more than 10 seconds, and time estimate is within 20%, then a progress indicator is better.
c. 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).

Appendix G: Content of Training Document given to Scenario-plus-General-Responsibilities-plus-Implementation-Suggestions Participants in Empirical Studies

Training Document

The following is an architecturally-sensitive usability scenario. Usability scenarios are considered to be architecturally-sensitive if it is difficult to add the scenario to a system after the architecture has been designed.

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.

Checklist of Cancellation Responsibilities

CR = Cancellation Responsibility.

CR1 – CR19 enumerate the responsibilities that should be fulfilled by any implementation of a cancel command. These are intended to be general responsibilities, covering a range of conditions, so many of them contain conditionals (if statements). Not all the responsibilities are relevant to every possible scenario. In a given scenario, the task and/or the system environment may make some cancellation responsibilities relevant, and some irrelevant, for that scenario.

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.

   a. For example, if the user pressed a cancel button, changing the color of the button will be seen.

b. 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.

a. If the estimate is between 1 and 10 seconds, changing the cursor shape is sufficient.
b. If the estimate is more than 10 seconds, and time estimate is within 20%, then a progress indicator is better.
c. 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).

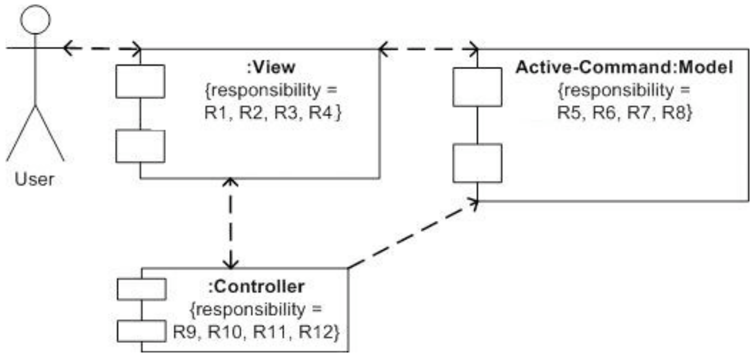
Sample Solution

In this sample solution, we will show you an example of how the Cancellation Responsibilities checklist above could be used. Our example will use the J2EE-MVC architecture. First we will show you a context, i.e., an architecture without the ability to cancel an active process. Then we will show you a sample solution, i.e., the same architecture after cancellation has been added, using the Cancellation Responsibilities checklist.

We'll use a non-critical task for the example. This implies that

- The user can have control while the cancellation is happening
- The user need not acknowledge the results of the cancellation

J2EE MVC architecture - Component Interaction Diagram


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

Assignment of Responsibilities of Components to J2EE-MVC without Cancellation:

View (Type: View)

R1. The view must render the models.
R2. The view must request updates from models.
R3. The view must send user gestures to the controller.
R4. The view must allow the controller to select views.

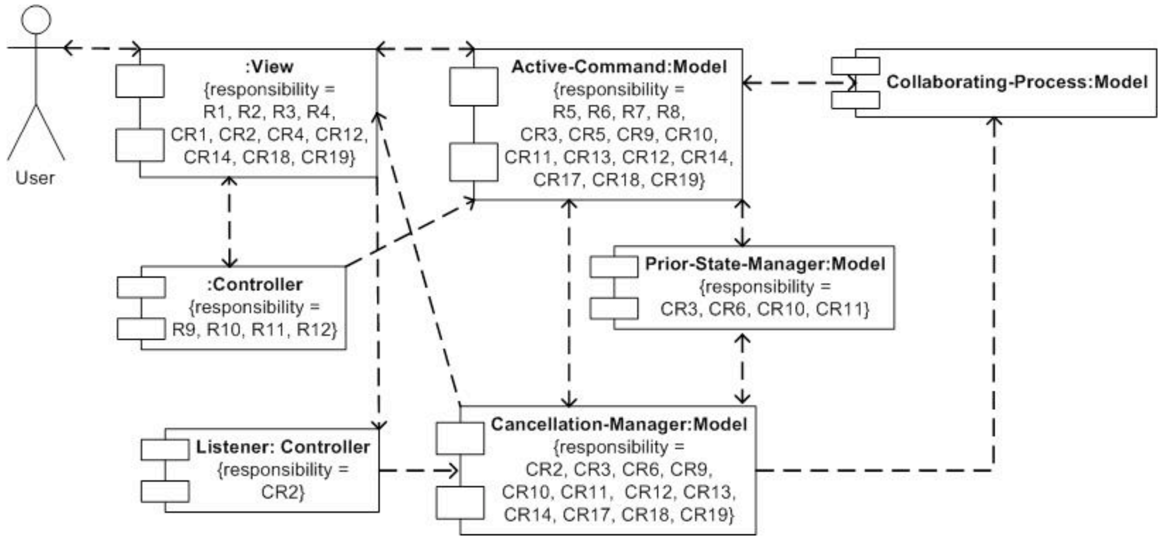
Active-Command (Type: Model)

R5. The model must encapsulate application state.
R6. The model must respond to state queries.
R7. The model must expose application functionality.
R8. The model must notify views of changes.

Controller (Type: Controller)

R9. The controller must define application behavior.
R10. The controller must map user actions to model updates.
R11. The controller must select a view for response.
R12. There must be a controller for each functionality.

Component Interaction Diagram for a Solution to Adding Cancel



Assignment of our Checklist of Responsibilities to new Components in J2EE-MVC with Cancellation

Listener (Type: Controller)
- Must always listen for the cancel command or environmental changes (CR2)

Cancellation Manager (Type: Model)
- Always listen and gather information (CR2, CR3)
- If the Active Command is not responding, handle the cancellation (CR6, CR9, CR10, CR11)
- Free resources (CR13)
- Estimate time to cancel (CR17)
- Inform the user of Progress of the cancellation (CR12, CR14, CR18, CR19)

Prior State Manager (Type: Model)
- 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 (CR3)
- If the Active Command is not responding (CR6), work with the Cancellation Manager to restore the system back to its state prior to execution of the command (CR10) or as close as possible to that state (CR11)

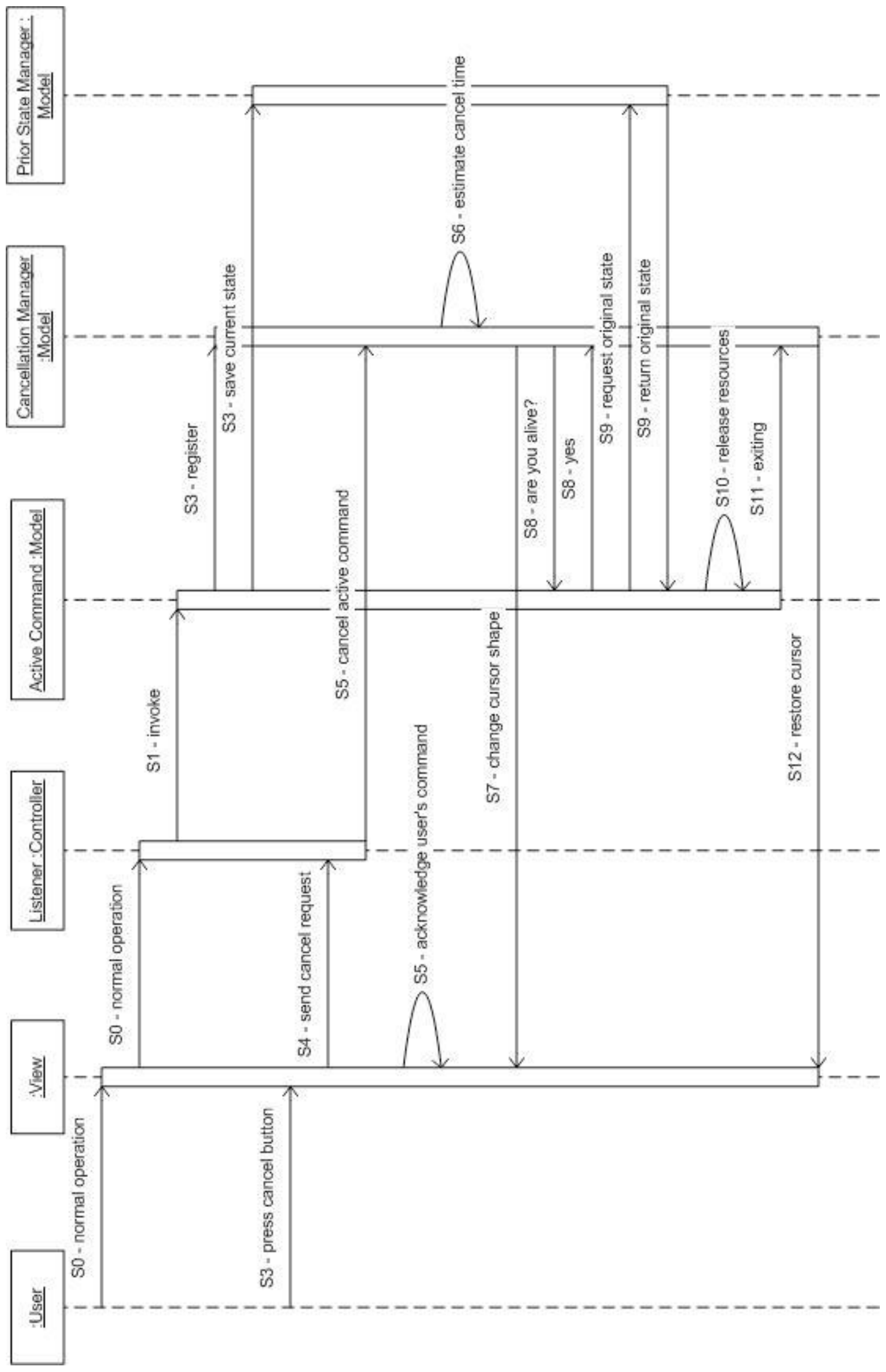
Assignment of our Checklist of Responsibilities to existing Components in J2EE-MVC with Cancellation

View (Type: View)
- Provide a button, menu item, keyboard shortcut and/or other means to cancel the active command (CR1)
- Must always listen for the cancel command or environmental changes (CR2)
- Provide feedback to the user about the progress of the cancellation (CR4, CR12, CR14, CR18, CR19)

Active Command (Type: Model)
- Always gather information (CR3)
- Handle the cancellation by terminating processes, and restoring state and resources (CR5, CR9, CR10, CR11, CR13)
- Provide appropriate feedback to the user (CR12, CR14, CR17, CR18, CR19)

Responsibilities not assigned or shown in our diagrams, and reasons why they are not assigned or shown
- Our diagram does not show the infrastructure (e.g., operating system) in which the application runs, therefore responsibilities assigned to the infrastructure (CR7, CR8) are not assigned.
- J2EE-MVC implicitly returns control to the user during cancellation, so CR15 is not assigned.
- We are not considering a “critical task” where the progress and results of the cancellation must affect user behavior, therefore CR16 is not assigned.

Sequence diagram: Component interaction during cancellation of active command



Component Interaction Steps (these correspond with the diagrams above) during successful cancellation of active command

S0. User performs normal operation in View, which passes the operation to Controller.

S1. Controller invokes Active Command.

S2. Active-command registers command with Cancellation Manager, and saves current state to Prior State Manager (CR3).

S3. User presses cancel button in View (CR1).

S4. View sends cancel request to Listener (CR1, CR2).

S5. Listener requests Cancellation Manager to cancel active command, and acknowledges user's command (CR2, CR4).

S6. Cancellation Manager estimates cancel time between 1 and 10 seconds (CR17).

S7. Cancellation Manager directs View to change cursor shape (CR18, busy cursor needed).

S8. Cancellation Manager verifies that Active Command is alive (CR5).

S9. Active Command requests original state from Prior State Manager, which returns original state to Active Command (CR10, CR11, CR12).

S10. Active Command releases resources consumed to process the command (CR13, CR14).

S11. Active Command exits, passing control to Cancellation Manager (CR19).

S12. Cancellation Manager restores original cursor to View, providing feedback to User that cancellation is finished (CR19).

Appendix F: Content of Task Instructions given to Participants in Empirical Studies

Usability & Software Architecture Training TASK INSTRUCTIONS

Some background: How Plug-ins Work

A plug-in is a separate module of code that behaves like an extension to the application that invokes it. Plug-ins enable the features of an existing application to be extended beyond its original design. For example, plug-ins are frequently used in Web browsers to increase the flexibility of the browser by handling one or more data types (through Multipurpose Internet Mail Extensions [MIMEs]), thereby extending the capabilities to a wide range of interactive and multimedia capabilities. The details of the plug-in mechanisms may be different from each other due to the various platforms and semantics. In desktop operating systems (e.g. Windows), a browser (e.g. IE) stores a list of plug-ins in a plug-in directory that acts like a registry. When the browser encounters data of a particular MIME data type, it searches its registry for a matching plug-in that can handle that data type. Once located, the plug-in is loaded into memory and initialized, and an instance of it is created. The life cycle of the plug-in is controlled by the browser that invokes it. When the browser window is closed, the plug-in instance is deleted, and the code is unloaded from memory.

The architectural design: Plug-In Architecture for Mobile Devices (PAMD)

PAMD is an architecture design for the exchange of data and control between applications operating on a device using Palm OS 4. PAMD details the architecture, syntax, and semantic behavior of applications to support application-level data exchange.

Example Scenario: How PAMD Works

There is no easy and standard way for applications to communicate inside a PDA running Palm OS 4, which uses a single-threaded architecture. PAMD was designed to help a user facilitate this communication. The following end-to-end scenario explains how PAMD can help achieve this and assumes that PAMD-aware applications and the plug-ins used are already installed in the user's device.

1) A user launches the PAMD-aware browser application.
2) The user surfs to an online movie trailer website in the browser application.
3) The user decides to view a Shockwave trailer for a movie in the browser application, and clicks on the appropriate link to select the Shockwave trailer. The browser informs the user that there are no Shockwave file viewing capabilities (as determined by the MIME data type), he decides to see if any plug-ins already installed on the device can be used to view Shockwave files.
4) The user selects the Plug-Ins menu item to locate the available plug-in services installed in the Palm device.
5) PAMD displays a list of all the plug-ins available for the browser application to use. In this case, plug-ins such as Shockwave, RealPlayer, and Java Console are displayed.
6) The user selects the Shockwave plug-in.
7) PAMD sends the link for the movie trailer to the Shockwave plug-in.
8) The Shockwave window is launched, and the movie trailer begins to load in that window. A progress indicator shows the progress of the download.

9) When the movie trailer is completely loaded, the Shockwave plug-in plays the movie trailer.

If the user wishes to cancel the operation of the plug-in, for example if the download is taking too long, he/she might do so during step 8 of this scenario.

PAMD Requirements

Here are the overall requirements of PAMD:

- Any application that complies with the PAMD framework will access PAMD plug-ins to extend its capabilities, and any plug-in that complies with PAMD shall be used by any PAMD-aware application.
- PAMD uses bi-directional communication between applications and plug-ins. PAMD-aware applications shall be able to pass and receive complex information to and from plug-ins.

Checklist of Responsibilities of PAMD Components

The PAMD architecture is composed of three active entities: a PAMD plug-in manager, a PAMD-aware application, and a PAMD plug-in, and three repositories: a system database, a plug-in registry, and shared memory.

PR = PAMD Responsibility (e.g. PR1 = PAMD Responsibility 1)

PR1. PAMD-aware applications and PAMD plug-ins must interact via the plug-in manager. When the application sends a request for services to the plug-in manager, the plug-in manager delegates the request to a plug-in. The assumption behind this process is that applications and plug-ins should know what kind of data types they exchange.

PR2. PAMD-aware applications and PAMD plug-ins must exchange data through direct access to the shared memory.

PR3. To maintain up-to-date PAMD plug-in information, the plug-in manager must query the system database.

PR4. The plug-in manager must cache this information in a private database called a plug-in registry, for faster use.

PR5. The plug-in manager must coordinate communication between plug-ins and applications. When an application wants to execute a plug-in, it requests the plug-in manager to execute the plug-in on its behalf. At this point, the plug-in manager calls the specified plug-in and passes control to it. Once the plug-in executes, the plug-in manager regains control and then returns the result and control back to the calling application.

PR6. The plug-in manager must maintain the plug-in registry, where information about each installed plug-in, such as the data it can handle, is stored. The PAMD system uses that information to determine the state of availability of every registered plug-in. The type of information that can be processed by a given plug-in serves as selection criteria for matching plug-ins and applications.

PR7. The PAMD plug-in must provide a service to applications.

PR8. The PAMD plug-in does not store any information about which applications will use it.

PR9. The PAMD plug-in does not need to be aware of PAMD as long as it conforms to the PAMD specification.

PR10. The PAMD-aware application must be aware of the availability of the plug-in manager.

PR11. The PAMD-aware application must request and receive services from PAMD plug-ins.

PR12. The PAMD-aware application must be able to request and receive services through the provided PAMD APIs and abstract data types (ADTs).

PR13. The PAMD-aware application must know about plug-ins in regards to PAMD.

PR14. The PAMD-aware application must not assume that a given plug-in is present.

PR15. The shared memory must store data, and must be accessible by the PAMD-aware application and by the PAMD Plug-in.

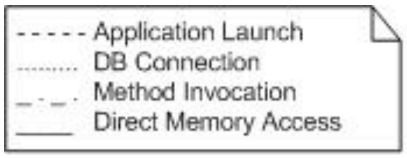
PR16. The system database must store data about registered plug-ins, and must be accessible by the Plug-in Manager.

PR17. The Plug-in Registry must store data about registered plug-ins in cache memory, and must be accessible by the Plug-in Manager.

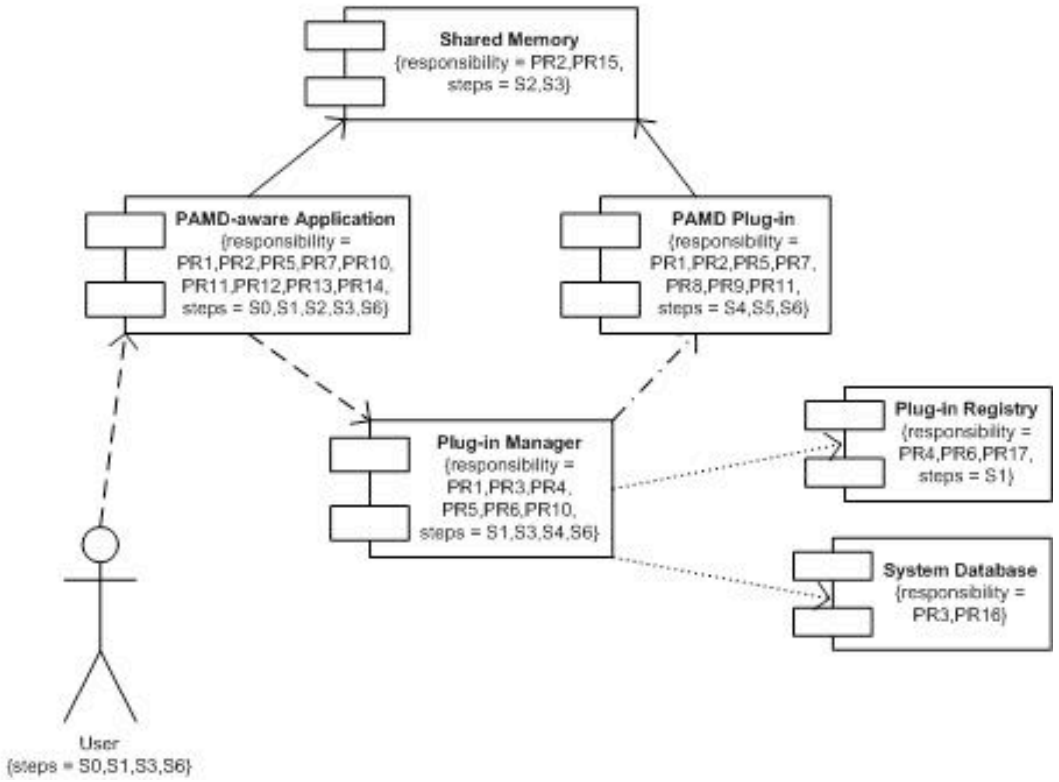
For the sake of simplicity, the following architectural diagrams show only one PAMD-aware application, and one plug-in which acts as a service provider. The **Component Interaction Diagram** below describes the basic components and connectors in PAMD.

Component Interaction Diagram

In this diagram, PR = PAMD Responsibility, and S = Step. PR numbers (PR1, PR2, etc.) in this diagram correspond with the PAMD Responsibilities in the checklist on pages 2 and 3.
S numbers (S1, S2, etc.) correspond with the Component Interaction Steps listed on page 6 of these instructions.



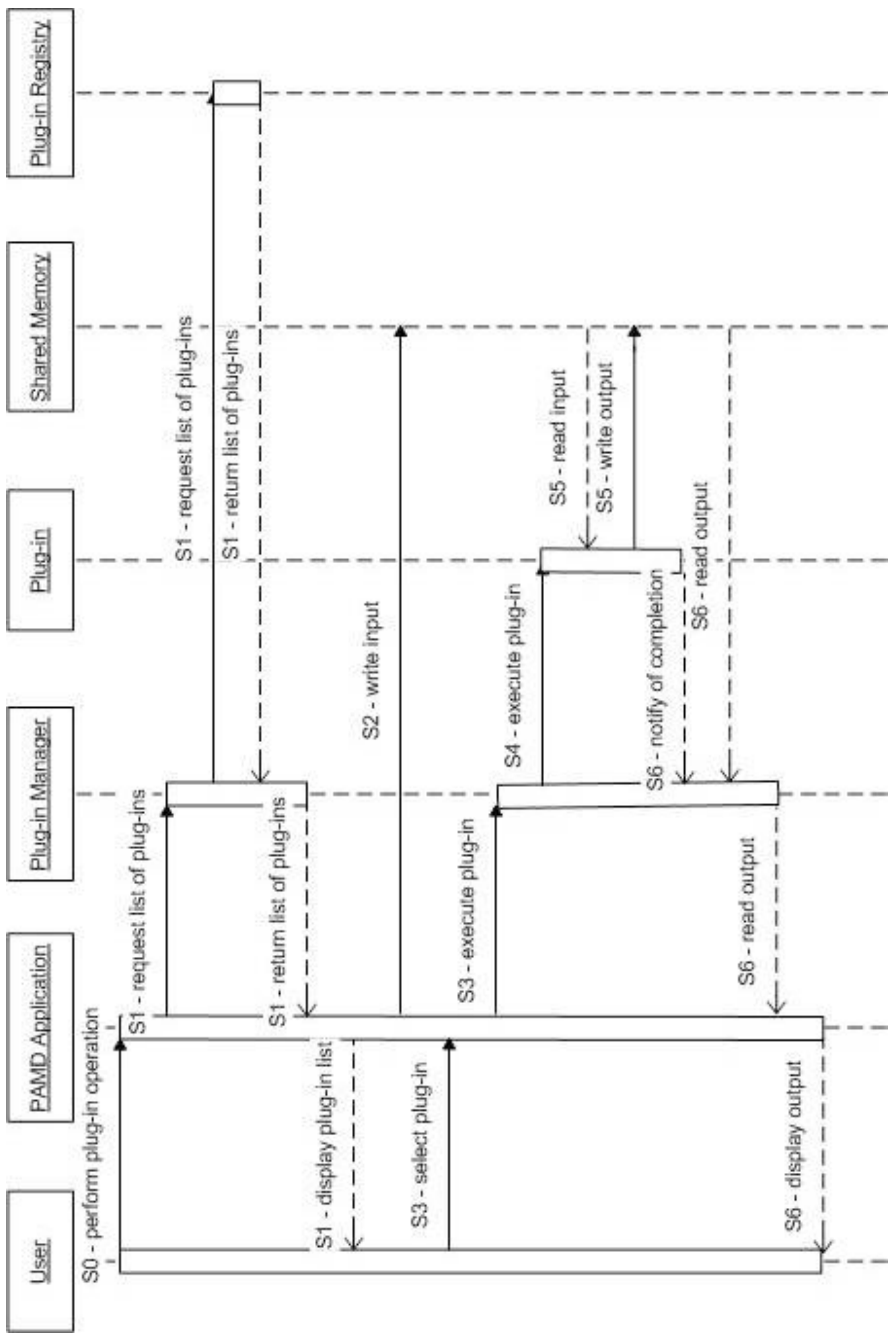
- - - - - Application Launch
.... DB Connection
_ . _ . Method Invocation
_____ Direct Memory Access



Shared Memory {responsibility = PR2,PR15, steps = S2,S3}
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 Manager {responsibility = PR1,PR3,PR4, PR5,PR6,PR10, steps = S1,S3,S4,S6}
Plug-in Registry {responsibility = PR4,PR6,PR17, steps = S1}
System Database {responsibility = PR3,PR16}
User {steps = S0,S1,S3,S6}

Sequence Diagram: Component Interaction During Plug-In Execution - Calling A Plug-In

In this diagram, S = Step. S numbers (S1, S2, etc.) correspond with the Component Interaction Steps listed on page 6 of these instructions.



Component Interaction Steps

These steps correspond with the Component Interaction Diagram on page 4 of this document, and the Sequence Diagram on page 5 of this document.

S = Step (e.g. S1 = Step 1)

Assumption: The needed plug-in is already registered.

The plug-in manager searches the system database to detect a newly installed plug-in. If one is found, the plug-in manager registers it by asking it for its name and description, and the data types it supports. (Note: this occurs before the sequence diagram starts.)

S0. User performs plug-in operation.

User selects the plug-ins menu in the PAMD-aware application.

S1. A list of compatible plug-ins is requested.

A PAMD-aware application requests a list of compatible plug-ins that can process the input and output data types handled by the application. Upon receiving this request, the plug-in manager searches the plug-in registry and returns the list of compatible plug-ins to the application.

S2. The application copies its data into shared memory.

This allows the plug-in to access required input data.

S3. The execution of the plug-in is requested.

A PAMD-aware application requests the plug-in manager that a specified plug-in be executed.

S4. The plug-in is called.

The plug-in manager calls the plug-in to perform its service and yields the control thread to it.

S5. Needed data is accessed.

If a plug-in needs to access the input data during execution, it accesses it from shared memory. At the end of its execution, the plug-in can save its output data in shared memory.

S6. After the plug-in finishes execution, control and the result are returned to the plug-in manager. The plug-in manager then returns the result and control to the calling application.

Task

Using the materials in the Training Document, and the information we just went over, add the ability to cancel a plug-in to the PAMD architectural design.

- **Only modify the architecture to address cancellation.** Do not attempt to modify the architecture to address any other concerns, just cancellation. Only the concerns described in the Training Document are relevant to this task. This also means that you do not need to preserve any other existing concerns (e.g. extensibility or performance) in considering your solution.

- **Be as complete as possible.** The diagrams and checklists of responsibilities currently correspond with each other, that is, the components, numbered steps and responsibilities match across all diagrams and written lists. Be sure that your modified diagrams and checklists of responsibilities also correspond with each other, including any changes to static elements, sequential operations, and responsibilities of the elements during the operation.

- **Indicate changes by modifying the diagrams and written steps on the Answer Paper (the same diagrams and steps shown above)**:

  - *Component Interaction Diagram*
  - *Sequence Diagram: Component Interaction During Plug-In Execution*
  - *Component Interaction Steps*

  Please modify the diagrams on the Answer Paper. In adding cancel to the PAMD architectural design, you should add new components, connectors, component interaction steps and/or responsibilities, to the diagrams, lists of steps, and/or checklists of responsibilities, as needed.

  We have provided extra Answer Paper for your convenience. The number of pages provided is not indicative of how many pages your answer might require.

  If you add new *connectors*, use the notation shown in the Legend on the Component Interaction Diagram.

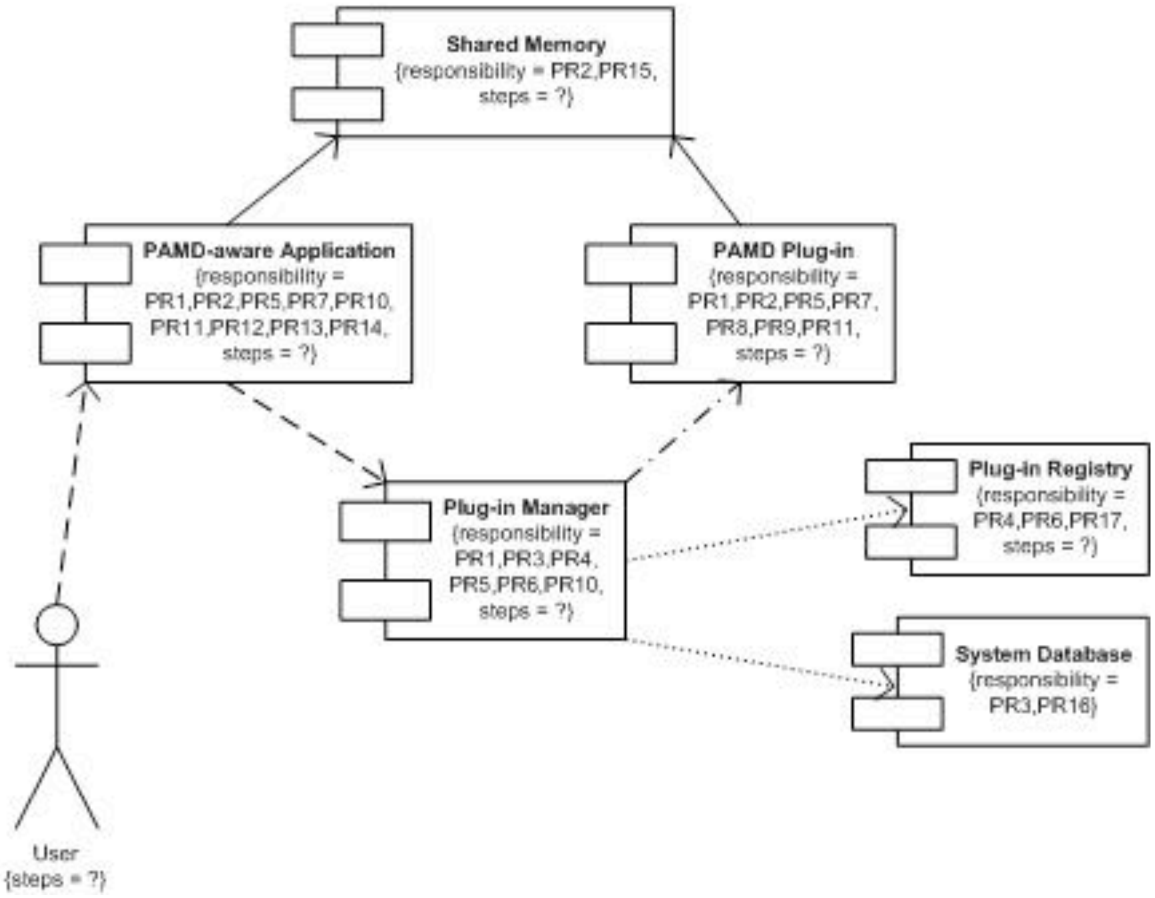
  If you add new *components*, use the notation shown in the Component Interaction Diagram: draw a box for each new component, list its responsibilities and steps inside the box (you may go outside the box if you don't have enough space to write).

  If you create any new *responsibilities* for any components, write them in a checklist and number them. Then assign the responsibilities you created by writing their numbers inside or next to the corresponding components in the Component Interaction Diagram.

Appendix G: Content of Answer Paper Given to Participants in Empirical Studies.

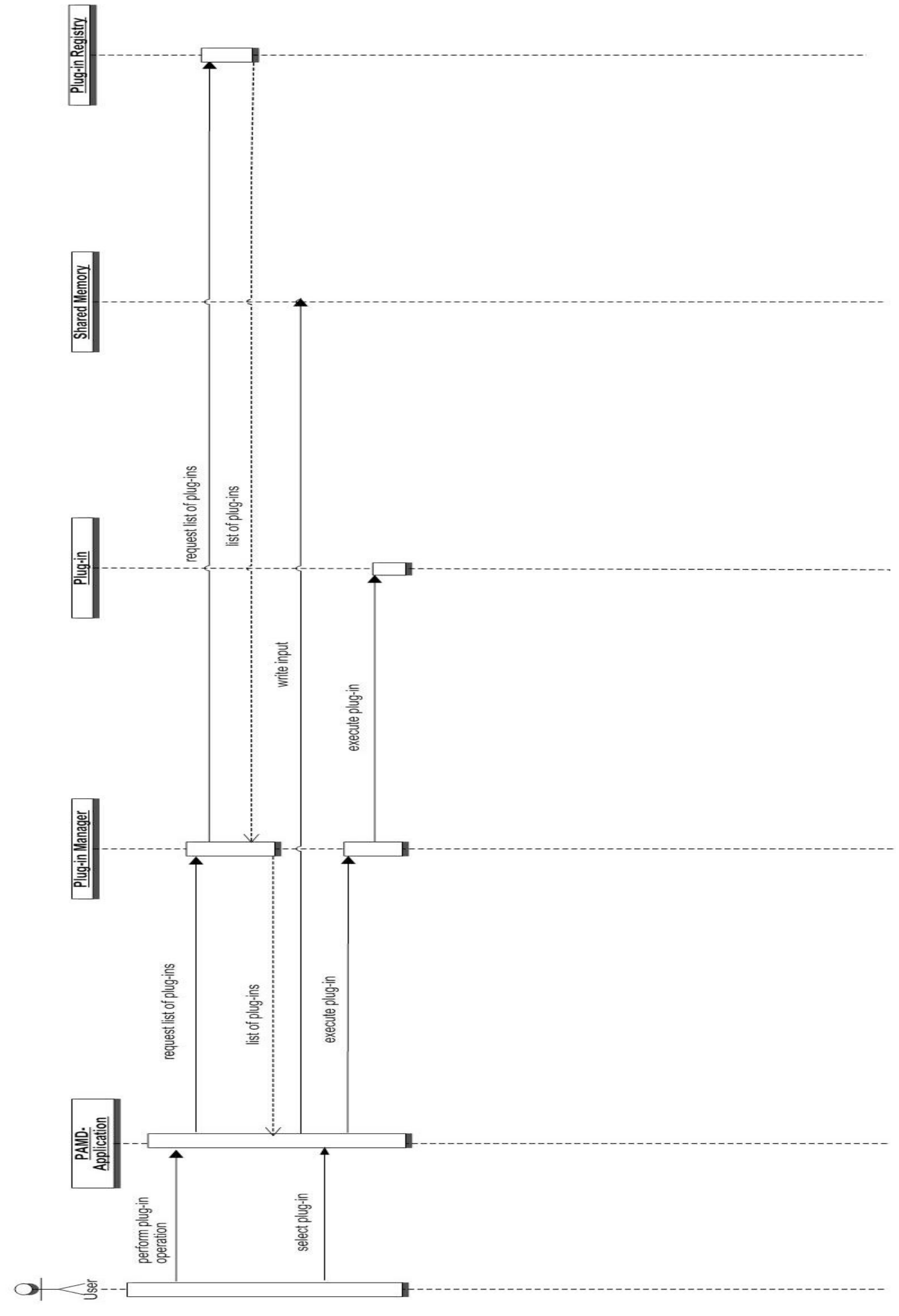
Component Interaction Diagram

- - - - - Application Launch
.... DB Connection
_ . _ . Method Invocation
_____ Direct Memory Access


Shared Memory
{responsibility = PR2,PR15, steps = ?}
PAMD-aware Application
{responsibility = PR1,PR2,PR5,PR7,PR10, PR11,PR12,PR13,PR14, steps = ?}
PAMD Plug-in
{responsibility = PR1,PR2,PR5,PR7, PR8,PR9,PR11, steps = ?}
Plug-in Manager
{responsibility = PR1,PR3,PR4, PR5,PR6,PR10, steps = ?}
Plug-in Registry
{responsibility = PR4,PR6,PR17, steps = ?}
System Database
{responsibility = PR3,PR16}
User
{steps = ?}

Sequence Diagram: Component Interaction During Plug-In Execution – Calling A Plug-In

After the plug-in begins to execute (the last step shown below), the user decides to cancel. Please modify the sequence diagram to show the sequence after the ability to cancel has been added, and the user decides to cancel during the execution of the plug-in. Extend the instantiation of the given components, and add new components and steps as needed. Please remember to number all steps.



Component Interaction Steps

These steps correspond with the Component Interaction Diagram, and the Sequence Diagram. The steps currently have no numbers, because you may add steps as needed at any point in this list. When you have finished adding steps, please number all the steps, including the existing ones and any you have added. Make sure these step numbers appear in your finished Component Interaction Diagram and Sequence Diagram.

S = Step (e.g., S1 = Step 1)

Assumption: The needed plug-in is already registered.

The plug-in manager searches the system database to detect a newly installed plug-in. If one is found, the plug-in manager registers it by asking it for its name and description, and the data types it supports. (Note: this occurs before the sequence diagram starts.)

S7. User performs plug-in operation.

User selects the plug-ins menu in the PAMD-aware application.

S8. A list of compatible plug-ins is requested.

A PAMD-aware application requests a list of compatible plug-ins that can process the input and output data types handled by the application. Upon receiving this request, the plug-in manager searches the plug-in registry and returns the list of compatible plug-ins to the application.

S9. The application copies its data into shared memory.

This allows the plug-in to access required input data.

S10. The execution of the plug-in is requested.

A PAMD-aware application requests the plug-in manager that a specified plug-in be executed.

S11. The plug-in is called.

The plug-in manager calls the plug-in to perform its service and yields the control thread to it.

S12. Needed data is accessed.

If a plug-in needs to access the input data during execution, it accesses it from shared memory. At the end of its execution, the plug-in can save its output data in shared memory.

S13. After the plug-in finishes execution, control and the result are returned to the plug-in manager. The plug-in manager then returns the result and control to the calling application.

Appendix H: Coding Plan for Experimental Solutions in Empirical Studies.

USAPs Experiment Participant Data Coding Plan for Written Participant Solutions

(In this plan, the term "Template" refers to individual participant solution coding template ScoredData-PIDTemplate.xls)

1. **PORTIONS OF PARTICIPANT SOLUTION TO IGNORE.** If any diagram or text in the Participant Solution is marked by the Participant as indicated in the instructions of this step, that part of the Participant Solution will be ignored.
    a. If any diagram or text in the Participant Solution is marked with the word "IGNORE" by the Participant, that part of the Participant Solution will be ignored. [This rule was added during coding of PID 19.]
    b. If any diagram or text in the Participant Solution is crossed with a single solid line, "X," or wavy line or "squiggle," that part of the Participant Solution will be ignored. [This rule was added during coding of PID 19.]
    c. If any diagram or text in the Participant Solution is marked with the word "ROUGH" by the Participant, that part of the Participant Solution will be ignored. [This rule was added during coding of PID 22.]
    d. If any preprinted page of the Answer Paper has not been written on or marked in any way by the Participant, that part of the Participant Solution will be ignored. [This rule was added during coding of PID 22.]
    e. If any preprinted step of the Component Interaction Steps has no number or notation attached to it by the Participant, that part of the Participant Solution will be ignored. [This rule was added during coding of PID 25.]
    f. If any diagram or text in the Participant Solution is crossed out with a series of lines or "crosshatching," that part of the Participant Solution will be ignored. [This rule was added during coding of PID 29.]

g. If any diagram or text in the Participant Solution is marked with the word "worksheet" by the Participant, that part of the Participant Solution will be ignored. [This rule was added during coding of PID 38.]

h. Using colored highlighter, mark off each area to be ignored under the rules above on the Participant Solution, until there are no unmarked areas to be ignored remaining on the Participant Solution. [This step was added following the scoring of PID27, and applied retroactively to the previously scored solutions.]

2. **ADDITIONAL RESPONSIBILITIES.** Find any Additional Responsibilities written in Participant Solution. Three types of entries will be classified as Additional Responsibilities.

   a. If the participant wrote any CRs (CR1, CR2, etc.) in the Component Diagram, these will be counted as Additional Responsibilities.

      i. Number each referenced CR with the corresponding AR number (CR1 = AR1, etc.).

      ii. Enter the new AR number in Column D of the Template, in the same row as the corresponding CR number. The possible numbers are AR1 through AR19.

      iii. AR1 through AR19 will not require rewriting the text of the original CR in column E of the Template.

      iv. Using colored highlighter, mark off each CR on the Participant Solution after transferring to the Template, until there are no unmarked CRs remaining on the Participant Solution. [This step was added following the scoring of PID27, and applied retroactively to the previously scored solutions.

   b. If the participant wrote any new PRs (PR18, PR19, etc.) anywhere on the answer paper, these will be counted as Additional Responsibilities.

      i. Number each new PR with an AR number beginning with AR20 (PR18 = AR20, PR19 = PR21, etc.). [anonymization only]

ii. Enter the new AR number(s) in Column D of the Template, beginning with the row immediately following the row containing CR19. [anonymization only]

iii. Enter the text of each new PR in Column E of the Template, immediately to the right of the AR designation corresponding to that text. [anonymization only]

iv. Where applicable, move each new AR and its corresponding text to the row of the CR (in Column A) whose responsibility the new AR satisfies (keeping the AR number and the AR text in Columns D and E of the same row). If there is more than one AR corresponding to any CR, add multiple rows to the CR so that there is one row (from Column D through the end of the individual column components) for each AR within the CR. [anonymization only]

v. Correspondence of PRs/ARs to CRs:

    1. If the text of the AR speaks to the provision of a user interface for invoking cancellation, the AR will be considered to correspond to CR1.

        a. This can be indicated by the inclusion of the words “cancel” or “cancellation,” AND “user” or “users,” OR “button” or “menu”, OR optionally “interface.” [This rule added during coding of PID22.]

    2. If the text of the AR speaks to the estimation of time required for cancellation, the AR will be considered to correspond to CR17.

        a. This can be indicated by the inclusion of the words “estimate,” “estimated,” or “estimating,” AND “time.” [This rule added during coding of PID 22.]

    3. If the text of the AR speaks to the notification of the user of time estimation required for cancellation, the AR will be considered to correspond to CR18.

    a. This can be indicated by the inclusion of the words “notify” or “display” or “inform” AND “estimate” or “estimated” or “calculated” AND “time” or “progress.” [This rule added during coding of PID 22.]

4. If the text of the AR speaks to the freeing of resources, the AR will be considered to correspond to CR13.
    a. This can be indicated by the inclusion of the words “removes,” “releases,” “free” or “clear” or “unload” or “dispose” AND “resource” or “resources,” or “memory.” [This rule added during coding of PID 22.]
    b. This can also be indicated by the inclusion of the words “delete” or “deletes,” “erase” or “erased,” and “list” or “information.” [This rule added during coding of PID34.]
5. If the text of the AR speaks to listening for cancellation, the AR will be considered to correspond to CR2.
    a. This can be indicated by the inclusion of the words “listen” or “listens” or “listener” and “cancel” or “cancellation” or “users.” [This rule added during coding of PID 28.]
6. If the text of the AR speaks to the provision of an internal system mechanism for cancellation, but not to the provision of a UI, the AR will be considered to correspond to CR5.
    a. This can be indicated by the inclusion of the words “terminate” or “terminates,” or “cancel” or “cancels,” or “kill” or “kills,” or “halts” or “abort” AND “plug-in” or “application” or “plug-in manager” or “sequence,” but NOT “user.”

7. If the text of the AR speaks to gathering of state information for restoration of state in the event of cancellation, the AR will be considered to correspond to CR3.
    a. This can be indicated by the inclusion of the words "save" or "saving," or "store" or "storing," or "gather" or "gathering," or "collect" or "collecting," or "write" or "written," AND "state," "usage," "resource usage," or "actions," [This rule added during coding of PID 23.]
    b. This can also be indicated by the inclusion of the words "maintain" or "maintaining," or "update" or "updating" AND "a list" or "details" or "log" of specific actions. [This rule added during coding of PID34.]
8. If the text of the AR speaks to restoration of state in the event of cancellation (but not to gathering of information related to state), the AR will be considered to correspond to CR10.
    a. This can be indicated by the inclusion of the words "restore" or "restored," or "return" or "display" or "regress" AND "state" or "data" or "values" AND "after cancellation" or "after canceling" or "after … cancelled" or step takes place after cancellation. [This rule added during coding of PID 39.]
9. If the text of the AR speaks to the mechanism of communication between two or more components, but not to any other issue of functionality besides communication, and if none of the components mentioned is the User, that AR should not be deemed to be a cancellation responsibility.

10. If the text of the AR speaks to notification to the user that cancellation is complete, the AR will be considered to correspond to CR19.
    a. This can be indicated by the inclusion of the words "notify" or "notification" or "acknowledge" AND "user" AND "cancelled," OR {"cancel" or "abort" AND "complete" or "completed" or "finished" or "restored"} but NOT "estimate" or "estimation."
11. If the text of the AR speaks to notification to the user that cancellation has been requested, the AR will be considered to correspond to CR4.
    a. This can be indicated by the inclusion of the words "notify" or "notification" or "acknowledge" AND "user" AND "cancel" or "abort" or "cancelled," but NOT "estimate" or "estimation."
12. If the text of the AR speaks to notification to the user that control cannot be returned to the user, the AR will be considered to correspond to CR16.
    a. This can be indicated by the inclusion of the words "notify" or "display" AND "error message" immediately following any description of returning control to user.
13. If the text of the AR speaks to notification to the user that resources cannot be released, the AR will be considered to correspond to CR14.
    a. This can be indicated by the inclusion of the words "notify" or "display" AND "error message" immediately following any description of releasing resources.

vi. Using colored highlighter, mark off each PR on the Participant Solution after transferring to the Template, until there are no unmarked PRs remaining on the Participant Solution. [This step was added following the scoring of PID27, and applied retroactively to the previously scored solutions.

c. If the participant wrote any new "Functionalities" anywhere on the answer paper, these will be counted as Additional Responsibilities. [This set was added during the scoring of PID33.]
    i. These will be identified by the inclusion of the word "functionalities" by the participant.
    ii. Follow same procedure as for substep b above.
d. If the participant rewrote the contents of P1 through P17 on the Answer Paper, these shall not be counted as new responsibilities.

3. **ADDITION OF COMPONENTS**.
    a. If the Participant has added any Components to the Component Diagram, add a column to the Template for each Component added by the Participant. Such columns are to inserted immediately to the right-most existing component in the Template, and the header cell to be included in the merged cells that begin with Column M in Row 1 of the Template. Additional columns should be added until there is an individual column for each preprinted component, and each participant-added component, in the Participant Solution.
4. **CODING OF WRITTEN COMPONENT INTERACTION STEPS**.
    a. If preprinted Component Interaction Steps in the Participant Solution are numbered by the Participant, those steps are to be handled as follows:
        i. Only steps relevant to Cancellation Responsibilities will be counted. Steps that relate only to the execution of the PAMD system previous to cancel, and that do not speak to listening for cancellation, gathering of state information, or provision of UI for cancellation, should not be entered in the Template.
        ii. Using colored highlighter, mark off each numbered, preprinted Component Interaction Step on the Participant Solution after transferring to the Template, until there are no unmarked,

numbered, preprinted Component Interaction Steps remaining on the Participant Solution. Steps so crossed out should also be crossed out where referenced in the Sequence Diagram and the Component Interaction Diagram, leaving only steps added or altered by the Participant. [This step was added following the scoring of PID27, and applied retroactively to the previously scored solutions.]

b. If any preprinted Component Interaction Steps in the Participant Solution are **not** numbered by the Participant, those steps are to be ignored.
c. If any Component Interaction Steps in the Participant Solution are added by the participant, those steps are to be handled as follows:
    i. Only steps relevant to Cancellation Responsibilities will be counted. Steps that relate only to the execution of the PAMD system previous to cancel, and that do not speak to listening for cancellation, gathering of state information, or provision of UI for cancellation, should not counted.
    ii. Language of the specific steps should be handled according to the rules set forth for Additional Responsibilities above.
    iii. Using colored highlighter, mark off each handwritten Component Interaction Step on the Participant Solution after transferring to the Template, until there are no unmarked Component Interaction Steps remaining on the Participant Solution. [This step was added following the scoring of PID27, and applied retroactively to the previously scored solutions.]

5. **CODING OF SEQUENCE DIAGRAM STEPS**.
    a. If any preprinted Sequence Diagram steps in the Participant Solution are numbered by the Participant, those steps are to be handled as follows:
    b. If any preprinted Component Interaction Steps in the Participant Solution are **crossed out** by the Participant, those steps are to be ignored.
    c. If more than one Sequence Diagram is provided by the Participant, the union of all Sequence Diagrams provided by the Participant will be

included in the Participant Solution. [This rule was added during the coding of PID 36.]

d. If any Sequence Diagram steps in the Participant Solution are added by the participant, those steps are to be handled as per the rules for Written Component Interaction Steps.

Appendix I: List of Usability Scenarios Offered to ABB

1. Progress Feedback – for commands that take between 2-10 sec, the shape of the cursor should change. For commands that take longer than 10 sec, the user should see the amount of work completed, the amount remaining, and, possibly, the time remaining. The user should also have the ability to cancel long running commands.

2. Warning/status/alert feedback – Status or alert indicators should be displayed when the appropriate conditions are satisfied. Some of the issues associated with they type of feedback are: how are "appropriate conditions" specified, is the feedback presented under the initiative of the system or does the user have to request it, is the feedback blocking or can other input proceed, how is the feedback dismissed from the screen – time based or user input.

3. Undo – commands issued by the user should be able to be undone after the command completes. The command issued immediately prior to the one that is undone should then be available for being undone, and so forth.

4. Cancel – long running commands should be able to be canceled by the user at any point prior to the completion of the command. The system should be reset to the extent possible to its state prior to the issuance of the command.

5. User profile – each user should be able to set various parameters that control the presentation. Customizations such as special icons, special terms, font and size of characters should all be settable by the user.

6. Help – user should be able to receive on line help at any point in a session. The help should be customized to the context.

7. Command aggregation – the user should be able to invoke a file with a collection of commands in it and have the individual commands executed. Some issues are: how

are the parameters of the individual commands specified, how is the collection created, how is it named and stored.

8. Action for multiple objects (aggregation of data). The user should be able to execute commands on collections of objects as well as individual objects. Some issues are: how are the collections specified, named, and stored.

9. Workflow model. Users should have support in sequencing their commands. That is, if one command must be executed prior to executing a second command, the system should disable the second command until the first one is executed. Help should provide a list of available commands and dependencies among commands.

10. Different views – Users should be able to look at data through different views. For example, a tabular view or a graphical view of numeric data should be available. Some of the issues are: how are different views invoked, the data seen in each view should be consistent and updated together, commands available in one view should be available in all views (where this makes sense).

11. Shortcuts (key and tasks) – The user should be able to identify special keys and abbreviations for commonly used commands.

12. Reuse of information. Information already in the computer should not have to be re-input by the users. Mechanisms for achieving this may be cut and paste or the propagation of data from one portion of the system to another.

13. Maintaining compatibility with other systems (leveraging human knowledge). System upgrades should not change the user interface for features that existed in the old system.

14. Navigating within a single view (e.g., scrolling, overview+detail, panning, zooming, outliner, etc.). Users should have the ability to move around a view using techniques

such as increasing font size, zooming, panning, scrolling. While performing these movements, the system should keep the user oriented as to their current context through showing an overview of the whole view.

15. Recovering from failure. When the system, the processor, or the network fails, the user should not lose any work in progress.

16. Identity management (retrieving forgotten password). Password protected systems should provide means to inform users of forgotten passwords. These means may involve mechanisms outside of the scope of the current system – e.g., sending mail with a temporary password.

17. Comprehensive searching. The user should have the ability to search the system to locate any kind of information managed by the system. This includes documents, data, file stores, project management, etc.

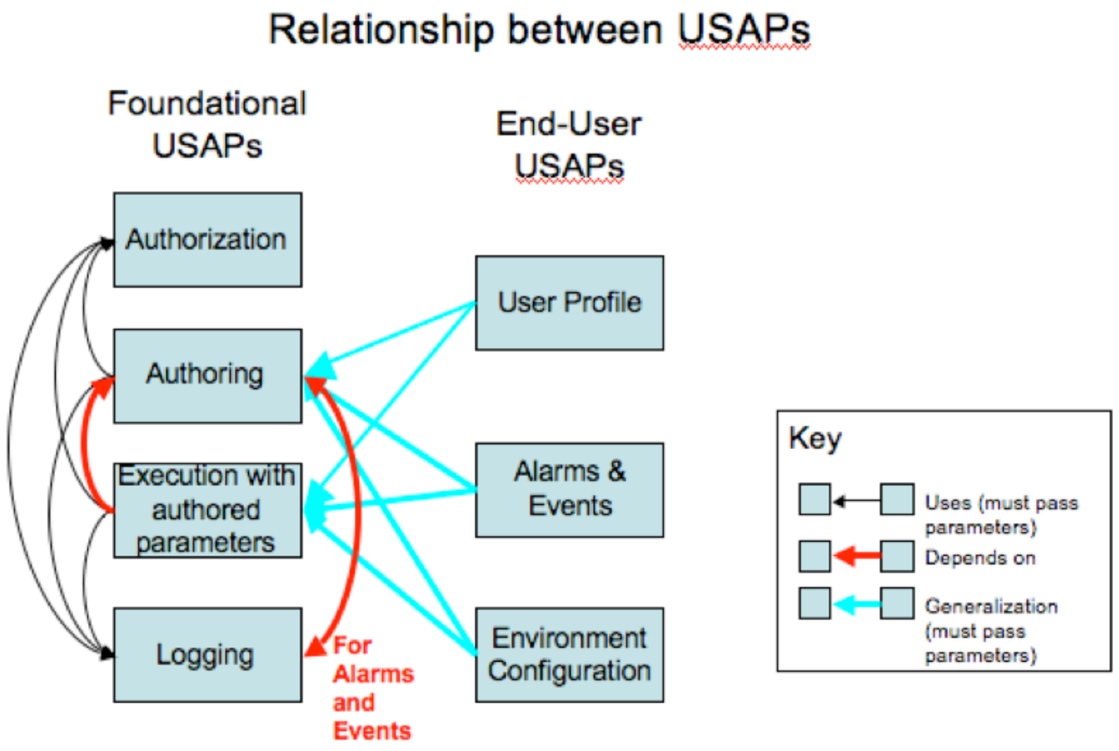
18. Supporting international use. The user should have the ability to use the system in a language and with a display format that is familiar to them.

19. Working at the user's pace. Users can only read and respond at human scale. The system should not flash messages, scroll, or otherwise present output too fast for the user to read and react. Similarly, users expect output to appear in a timely fashion and systems should operate at a speed appropriate to the task they are performing.

Appendix J: Pattern Language for ABB Usability Supporting Architectural Patterns.

Assumptions about USAPs

- There are intra-responsibility assumptions (e.g., Communication between involved portions of the system is assumed).
- There are inter-responsibility, intra-USAP assumptions (e.g., things done by one responsibility are accessible to other responsibilities in the USAP)
- There are inter-USAP assumptions (e.g., that the protocol for saving an authored specification is defined and known to the developers of both the authoring system and the execution system). These are "in" the red arrows in the diagram below and we just put in a "shared assumptions" section into the front-matter of each USAP.
- There are assumptions about the execution environment for systems informed by USAPs (i.e., code developed unconnected to USAPs and code developed by considering USAPs). For example, USAPs assume that there is a portion of the system that displays information to a user.


Relationship between USAPs
Foundational USAPs
End-User USAPs
Authorization
Authoring
Execution with authored parameters
Logging
User Profile
Alarms & Events
Environment Configuration
For Alarms and Events
Key
Uses (must pass parameters)
Depends on
Generalization (must pass parameters)

1. Foundational USAPs

For each Foundational USAP, provide:

Purpose:
- A general statement of the purpose of the foundational USAP.

Justification:
- Justification for this foundational USAP

Glossary:
- Definitions of terms introduced in this foundation USAP

Parameters needed by this USAP:
- These parameters are used to make the foundational USAP specific to referring USAPs.

(NOTE: Our expectation is that the following information will ultimately be generated automatically from the USAPs that use and are used by this USAP either directly or indirectly, otherwise there will be a maintenance headache.)

For each Foundational USAP used by this USAP (if any), provide:

Parameters furnished to the foundational USAP:
- The parameter values furnished to the foundational USAP used by this USAP.

For each End-User USAP that refers to this foundational USAP, provide

End-User USAP interpretation:
- The parameter values furnished to this foundational USAP by the referring end-user USAP.

1.1 Authorization

Purpose:
The Authorization Foundational USAP's purpose is to identify and authenticate users (human or other systems) of the system.

Justification:
Users must be authorized when security or personalization is important.

Glossary:
- To be completed after user testing

Parameters needed by this USAP:
- USER: Who the users are, i.e., the role they play. These can be human and/or other systems.
- ACTIVITY: What activities authorization will authorize (i.e., permissions).

Foundational USAPs used by this USAP:
None.

USAPs that use this Foundational USAP:

In the User Profile End-User USAP, there are two paths to this USAP, with two sets of parameters.

Through the Authoring Foundational USAP:

- USER: Author.
  ACTIVITY: Author user profile.

Through the Execution with Authored Parameters Foundational USAP:

- USER: End user.
  ACTIVITY: Execute the system with parameter values from user profile.

In the Environment Configuration End-User USAP, there are two paths to this USAP, with two sets of parameters.

Through the Authoring Foundational USAP:

- USER: Author.
- ACTIVITY: Author configuration description.

If necessary environment information must be entered by an authorized user, then through the Execution with Authored Parameters Foundational USAP:

- USER: End user.
- ACTIVITY: Execute the system with parameter values from configuration description.

In the Alarms and Events End-User USAP, there are two paths to this USAP.
Through the Authoring Foundational USAP:

- USER: Author.
- ACTIVITY: Author alarm and event rules and displays.

Through the Execution with Authored Parameters Foundational USAP:

- USER: End user.
- ACTIVITY: Author alarm and event rules and displays.

1.1.1 Identification

1.1.1.1 The system has to provide a way for the USER to input his, her or its identity. (For a human user, this input could be user typing a login ID, running a finger over a fingerprint reader, or standing in front of a camera for face recognition, etc. For a system user, this input could be an IP address, a previously specified identity, etc.)

1.1.1.1.1 Rationale

Software has no way of recognizing a USER without explicit input.
USERs want to access the system.

The environment contains multiple potential users only some of whom are allowed to use the system.

1.1.1.1.2 Implementing this responsibility

The USERs are involved because they must take explicit action.

The portion of the system that receives user input is involved, because users' explicit action must be handled by the system.

There must be a portion of the system that processes the input.

1.1.1.2 The system should inform the USER and/or the system administrator of the results of the identification. Typically, a successful identification is indicated by allowing the USER to proceed. (If logging of identification results is desired use Logging Foundational USAP with parameter CONTEXT=Identification.)

1.1.1.2.1 Rationale

USERs want to know if their identification succeeded so they can proceed.

USERs want to know if their identification failed because they cannot proceed without it.

System administrators might want to know of a failed identification because it might be an indication of unauthorized users attempting to access the system.

The environment contains multiple potential users, some of whom might be malicious.

1.1.1.2.2 Implementing this responsibility

In the event of failing to be identified, the portion of the system that does the identification must provide information about the failure.

If informing the system administrator: In the event of a failure, a portion of the system must have a mechanism to inform the system administrator. This may be quite complicated (e.g., sending email) and is beyond the scope of this USAP.

For informing the USER: In the event of a failure, the portion of the system that renders information to the user should display information about the failure. Often this feedback will wait for authentication information to be input and the feedback will be of the form that this userID/password are not known to the system.

In the event of a successful identification, the portion of the system that renders information to the user should indicate success in some way (e.g., simply letting the USER proceed).

The portion of the system that handles input from the user should provide a UI to dismiss any unwanted information.

1.1.1.3 The system must remember the USER's identity for the duration of the session.

1.1.1.3.1 Rationale

The software will use this information later (e.g., in determining authorization).

The USER only wants to enter this information once per session.

The environment contains multiple potential users only some of whom are allowed to use the system.

1.1.1.3.2 Implementing this responsibility

There must be a portion of the system that maintains the USER's identity.

1.1.1.4 The system must display the identity of the current USER.

1.1.1.4.1 Rationale

The USER wants to be sure that the system knows who he or she is.

1.1.1.4.2 Implementing this responsibility

The portion of the system that maintains the USER's identity provides the USER's identity.

The portion of the system that renders information to the user displays the USER's identity.

The portion of the system that handles input from the user should provide a UI to dismiss any unwanted information.

1.1.1.5 The system must provide a way for the USER to recover from mistakes in identification. (This could be an erroneous input by the user or the system or it could be the user forgetting his or her user ID. In the case of erroneous input, the solution could be as simple as re-entering the information, so this section will expand on the case of forgetting.)

1.1.1.5.1 Rationale

USERs sometimes forget the way to identify themselves to the system.

Software has no way of recognizing a USER without correct explicit input.

The environment contains multiple potential users only some of whom are allowed to use the system.

1.1.1.5.2 Implementing this responsibility

The USERs are involved because they must explicitly indicate that they forgot their ID.

The portion of the system that receives user input is involved, because users' explicit action must be handled by the system.

There must be a portion of the system that has a mechanism to inform the USER of the correct ID. This mechanism may be quite complicated (e.g., sending email to a previously-stored address), and is beyond the scope of this USAP.

1.1.2 Authentication

1.1.2.1 The system has to provide a way for the USER to input his, her or its authentication. (For a human user, this input could be user typing a password, running a finger over a fingerprint reader, or standing in front of a camera for face recognition, etc. For a system user, this input could be a certificate or previous authentication, etc.)

1.1.2.1.1 Rationale

Software has no way of authenticating a USER without explicit input.

USERs want to know that they are the only ones able to take action under their names.

The environment contains multiple potential users only some of whom are allowed to use the system.

The organization wants to restrict access to authorized USERs and to have USERs be accountable for their actions.

1.1.2.1.2 Implementing this responsibility

The USERs are involved because they must take explicit action.

The portion of the system that receives user input is involved, because users' explicit action must be handled by the system.

There must be a portion of the system that processes the input.

1.1.2.2 The system must perform and remember the results of the authentication. (This could be matching a password, recognizing a fingerprint, face recognition, etc. This might be a multi-stage process, e.g., first validating a user's ID and then matching it to the password.)

1.1.2.2.1 Rationale

The system is the entity that has the information necessary to perform the authentication.

1.1.2.2.2 Implementing this responsibility

The portion of the system that maintains the USER's identity provides the USER's identity.

There must be a portion of the system that performs the authentication.

The portion of the system that maintains the USER's identity must remember whether this USER has been authenticated.

1.1.2.3 The system should inform the USER and/or the system administrator of the results of the authentication. (If logging of identification results is desired use Logging Foundational USAP with parameter CONTEXT=Authentication.)

1.1.2.3.1 Rationale

USERs want to know if their authentication succeeded so they can proceed.

USERs want to know if their authentication failed because they cannot proceed without it.

System administrators might want to know of a successful authentication for audit trail purposes.

System administrators might want to know of a failed authentication because it might be an indication of unauthorized users attempting to access the system.

The environment contains multiple potential users, some of whom might be malicious.

1.1.2.3.2 Implementing this responsibility

In the event of failing to be authenticated,the portion of the system that does the authentication must provide information about the failure.

If informing the system administrator: In any event, a portion of the system must have a mechanism to inform the system administrator. This may be quite complicated (e.g., sending email) and is beyond the scope of this USAP.

For informing the USER: In the event of a failure, the portion of the system that renders information to the user should display information about the failure. In the event of a successful authentication, the portion of the system that renders information to the user should indicate success in some way (e.g., a welcome message or simply letting the user proceed).

The portion of the system that handles input from the user should provide a UI to dismiss any unwanted information.

1.1.3 Permissions

1.1.3.1 The system must permit or prohibit specific ACTIVITY dependent on who the USER is and on the ACTIVITY they are attempting to perform, i.e., the system must know (built in, stored, provided by the OS, etc.) the specific permissions for each USER and then enforce these permissions.

1.1.3.1.1 Rationale

The organization may wish to allow different USERs to have different capabilities.

The USERs want access to the ACTIVITY they have permission for.

The system has to have a concept of permissions to be able to allow or disallow an ACTIVITY.

The system has to know the mapping between USERs and ACTIVITY to grant different permissions to different users.

1.1.3.1.2 Implementing this responsibility

The portion of the system that maintains the USER's authenticated identity provides the USER's identity.

The portion of the system that does the ACTIVITY must provide the requested ACTIVITY.

There must be a portion of the system that maintains the mapping between USERs and their permitted ACTIVITY.

This latter portion of the system must use the USER's authenticated identity, the particular ACTIVITY desired and the map to determine whether the ACTIVITY is allowed.

1.1.3.2 The USER and/or the system administrator should be informed when permission is granted or denied for doing an ACTIVITY. Typically, this is done through the portion of the system that requests the ACTIVITY.

1.1.3.2.1 Rationale

USERs want to know if their operations failed because of permission.

System administrators might want to know if permission is denied because it might be an indication of users attempting to exceed their permissions.

The environment contains multiple potential users, some of whom might be malicious.

1.1.3.2.2 Implementing this responsibility

In the event of failing to be allowed to do ACTIVITY,the portion of the system that does the ACTIVITY must provide information about the failure.

If informing the system administrator: A portion of the system must have a mechanism to inform the system administrator. This may be quite complicated (e.g., sending email) and is beyond the scope of this USAP.

For informing the USER: In the event of a failure, the portion of the system that renders information to the user should display information about the failure. In the event of a successful permission, the portion of the system that does the ACTIVITY should proceed and inform the USER appropriately, typically through the results of performing the ACTIVITY.

The portion of the system that handles input from the user should provide a UI to dismiss any unwanted information.

1.1.4 Logging Off

1.1.4.1 The system must have a way for the USER to log off. (If there are requests still pending (e.g., unsaved changes), then notify the user and ask for confirmation of the log-off request. Consider some way to protect the system if the confirmation if not forthcoming in a reasonable amount of time.)

1.1.4.1.1 Rationale

Software has no way of knowing that a USER is finished without explicit input.

USERs want to protect their work.

The environment contains multiple potential users, some of whom may be malicious, so logging off prevents them from having access.

1.1.4.1.2 Implementing this responsibility

<u>If log-off is at the USER's request:</u>
The system must provide a means for the USER to indicate a desire to log-off (e.g., a button, keyboard shortcut, voice command).

The USER must indicate their desire to log off and respond to any confirmation request.

The portions of the system that do the ACTIVITY must ask for confirmation in the event of request still pending. This involves keeping track of pending requests and requesting the appropriate interactions with the USER.

There must be a portion of the system that de-authenticates the USER after receiving the log-off request and any necessary confirmations.

<u>If with a system-initiated request:</u>
There must be a portion of the system with a mechanism to initiate a log-off request. (This may be as simple as a time-out or quite complicated (e.g., shutting down the entire system), and is beyond the scope of this USAP.)

If there are pending requests, waiting for the USER to confirm is not a good idea, either the requests are aborted or the log-off is aborted.

1.2 Authoring

<u>Purpose:</u>
The Authoring Foundational USAP's purpose is to allow specification of the behavior of the system in certain ways under certain circumstances.

<u>Justification:</u>
Users want to control the behavior of the system in certain ways under certain circumstances without having to set it up every time. The system needs a specification of parameters to determine its behavior in these circumstances. Therefore the user must author a specification of parameters that will subsequently be used upon execution (see Foundational USAP Execution with Authored Parameters).

<u>Glossary:</u>

- To be completed after user testing

Parameters needed by this USAP:

- SPECIFICATION: The persistent parameter values that are authored.
- APPROPRIATENESS-INFORMATION: The circumstances under which it is appropriate to use a particular SPECIFICATION.

Shared Assumptions:
Shared with any other USAP that uses the SPECIFICATION:

- The syntax and semantics for the concepts that are included in the SPECIFICATION are defined and known to the developers of both the authoring system and the systems informed by other USAPs that share the SPECIFICATION.
- The protocol for saving the SPECIFICATION is defined and known to the developers of both the authoring system and the systems informed by other USAPs that share the SPECIFICATION.
- USAPs sharing these assumptions
    - Execution with Authored Parameters foundational USAP

Foundational USAPs used by this USAP:
Authorization Foundational USAP:

- USER: Author
- ACTIVITY: Author the SPECIFICATION

USAPs that use this Foundational USAP:

(Optional) Logging Foundational USAP

- SPECIFICATION: Logging specification.
- APPROPRIATENESS-INFORMATION: Logging state.
-

(Optional) Execution with Authored Parameters USAP

- SPECIFICATION: Execution with Authored Parameters.SPECIFICATION (this is the SPECIFICATION parameter that was passed to Execution with Authored Parameters)
- APPROPRIATENESS-INFORMATION: Execution with Authored Parameters. APPROPRIATENESS-INFORMATION
  (this is the APPROPRIATENESS-INFORMATION parameter that was passed to Execution with Authored Parameters)

User Profile End-user USAP:

- SPECIFICATION: User profile.
- APPROPRIATENESS-INFORMATION: User identity.

Environment Configuration End-User USAP:

- SPECIFICATION: Configuration description.
- APPROPRIATENESS-INFORMATION: Environment identity.

Alarms and Events End-User USAP:

- SPECIFICATION: Rules for Alarms, Events and Displays.
- APPROPRIATENESS-INFORMATION: Context of use.

1.2.1 Create a SPECIFICATION

1.2.1.1 The system must provide a way for an authorized author to create a SPECIFICATION. (See Authorization Foundational USAP with parameters USER=Author and ACTIVITY=Author the SPECIFICATION.)

1.2.1.1.1 Rationale

A SPECIFICATION doesn't exist unless it is created.

1.2.1.1.2 Implementing this responsibility

The portion of the system that renders output must render a UI that allows the parameters to be specified and displays existing values.

The portion of the system that accepts input from the user must accept parameters.

There must be a portion of the system with a mechanism to create new SPECIFICATIONs.

1.2.1.2 Consider providing default values for all specifiable parameters when a SPECIFICATION is created. (Providing defaults simplifies the creation process, but may increase the probability of error for environments that deviate from those defaults. If the cost of error is high, then consider not providing defaults or requiring confirmation of each value. Default values can be changed through modification.)

1.2.1.2.1 Rationale

System has to have *something* so defaults might be provided.

Circumstances might be very similar so defaults might capture that similarity.

Authors may want to be efficient and defaults may save authoring time.

Authors may only be interested in changing specific aspects of the SPECIFICATION, so having defaults for the rest of it is useful.

1.2.1.2.2 Implementing this responsibility

The portion of the system that creates a new SPECIFICATION must assign defaults.

1.2.1.3 The SPECIFICATION must be given an identifier. The identifier should be treated as a parameter that has a default value and can be modified by the user. (One mechanism for assigning this identification might be a "Save as".)

1.2.1.3.1 Rationale

Circumstances might be very similar so values in one SPECIFICATION may transfer to other circumstances.

Authors may want to be efficient and may want to begin the specification process with a similar SPECIFICATION.

The authoring system must have an identifier to distinguish one SPECIFICATION from another.

1.2.1.3.2 Implementing this responsibility

<u>If the identifier is provided by a author:</u>
The portion of the system that renders output must render a UI that allows the author to provide an identifier and display it.

The portion of the system that accepts input from the user must accept the identifier.

There must be a portion of the system that manages the authoring process.

The portion of the system that manages the authoring process must associate the SPECIFICATION with the identifier.

<u>If the identifier is provided automatically by the system:</u>

The portion of the system that creates the SPECIFICATION generates an identifier (e.g., MSWord automatically generates a name for a new document).

The portion of the system that manages the authoring process must associate the SPECIFICATION with the identifier.

1.2.1.4 The SPECIFICATION must be associated with the APPROPRIATENESS-INFORMATION (i.e., the circumstances under which it should be invoked (e.g., for specific users, roles, environments)). The APPROPRIATENESS-INFORMATION should be treated as a parameter that has a default value and can be modified by the author.

1.2.1.4.1 Rationale

The SPECIFICATION is only appropriate for use under certain circumstances.

System can only use a SPECIFICATION if it is associated with a circumstance.

Users want the system to work for them the way they want it to work when they want it to work that way.

1.2.1.4.2 Implementing this responsibility

<u>If the association is provided by an author:</u>
The portion of the system that renders output must render a UI that allows the authoring of APPROPRIATENESS-INFORMATION and display existing values.

The portion of the system that accepts input from the user must accept the APPROPRIATENESS-INFORMATION.

The portion of the system that manages the authoring process must associate the SPECIFICATION with the APPROPRIATENESS-INFORMATION.

<u>If the association is provided automatically by the system when a new user or role is created:</u>
The portion of the system that creates the SPECIFICATION generates an association.

The portion of the system that manages the authoring process associates the SPECIFICATION with the circumstances.

1.2.2 Save a SPECIFICATION.

1.2.2.1 The system must provide a means for an authorized author to save and/or export the SPECIFICATION (e.g., by autosave or by author request). (See Authorization Foundational USAP with parameters USER=Author and ACTIVITY=Author the SPECIFICATION.) If other systems are going to use the SPECIFICATION, then use a format that can be used by the other systems. (If logging of authoring results is desired use Logging Foundational USAP with parameter CONTEXT=Authoring, SPECIFICATION=SPECIFICATION.)

1.2.2.1.1 Rationale

Authors want to be efficient (i.e., input information into the SPECIFICATION only once).

The system can only remember things if they are persistent from session to session.

The software may need to share a SPECIFICATION with other software.

1.2.2.1.2 Implementing this responsibility

If the initiation of the save was automatic:
That portion of the system that manages the authoring process performs the initiation.

That portion of the system that manages the authoring process stores and/or exports the SPECIFICATION.

If the initiation of the save was at the author's request:
The portion of the system that renders output must render a UI that allows the parameters needed by the system (e.g., format, location) to be input and display them.

The portion of the system that accepts input from the user must accept the parameters.

That portion of the system that manages the authoring process stores and/or exports the SPECIFICATION.

1.2.3 Modify a SPECIFICATION

1.2.3.1 Provide a way for an authorized author to retrieve a SPECIFICATION (e.g., import a previously-saved file, utilize a previously-generated data structure, or restore to default values). (See Authorization Foundational USAP with parameters USER=Author and ACTIVITY=Author the SPECIFICATION.)

1.2.3.1.1 Rationale

Authors might want different values than are currently assigned.

System has stored the information and must have a current set of data to work with.

1.2.3.1.2 Implementing this responsibility

The portion of the system that renders output must render a UI that allows the author to request a retrieval of a SPECIFICATION.

The portion of the system that accepts input from the user must accept this request.

That portion of the system that manages the authoring process retrieves the SPECIFICATION.

1.2.3.2 Display the current parameter values for the SPECIFICATION (including identifier and circumstances).

1.2.3.2.1 Rationale

Authors want to see what they are editing.

1.2.3.2.2 Implementing this responsibility

That portion of the system that manages the authoring process must provide the values.

The portion of the system that renders output must render a UI that displays the values.

The portion of the system that handles input from the user should provide a UI to dismiss any unwanted information.

1.2.3.3 The system must provide a ways for an authorized author to change the parameter values. The syntax and semantics of the values specified should conform to the assumptions of the execution environment of the system. (More details: Best practice is to constrain the author to such conformation (e.g., choose from drop-down list, provide a slider for a range of values). If the author's choices are not constrained, then the system should check the syntax and semantics and provide feedback if either are unreasonable.) (See Authorization Foundational USAP with parameters USER=Author and ACTIVITY=Author the SPECIFICATION.)

1.2.3.3.1 Rationale

Authors might want different values than are currently assigned.

1.2.3.3.2 Implementing this responsibility

The portion of the system that renders output must render a UI that allows values to be changed.

The portion of the system that accepts input from the user must accept new values.

That portion of the system that manages the authoring process replaces the current values with the new values.

1.2.4 Delete a SPECIFICATION

1.2.4.1 Provide a way for an authorized author to tentatively remove a SPECIFICATION from the system (e.g., analogous to dragging a file to the trash). The system might require a confirmation from the author prior to performing this action. (See Authorization Foundational USAP with parameters USER=Author and ACTIVITY=Author the SPECIFICATION.) (If logging of authoring deletions is desired use Logging Foundational USAP with parameter CONTEXT=Authoring, SPECIFICATION=SPECIFICATION.)

1.2.4.1.1 Rationale

Authors might accidentally delete a SPECIFICATION and want to restore it.

The organization doesn't want extraneous SPECIFICATIONs on the system.

The system has to keep it around in case it has to be restored.

1.2.4.1.2 Implementing this responsibility

The portion of the system that renders output must render a UI that allows the author to tentatively delete a SPECIFICATION.

The portion of the system that accepts input from the user must accept this request.

That portion of the system that manages the authoring process saves the SPECIFICATION in case has to be restored.

The portion of the system that renders output must indicate that the SPECIFICATION has been (tentatively) deleted.

1.2.4.2 Retrieve a tentatively-deleted SPECIFICATION from the system (e.g., analogous to dragging a file out of the trash)

1.2.4.2.1 Rationale

Authors might want to restore an accidentally deleted SPECIFICATION. The system has kept it around so that it can be restored.

1.2.4.2.2 Implementing this responsibility

The portion of the system that renders output must render a UI that allows the author to restore a tentatively deleted SPECIFICATION.

The portion of the system that accepts input from the user must accept this request.

That portion of the system that manages the authoring process must restore the SPECIFICATION.

The portion of the system that renders output must indicate that the SPECIFICATION has been restored.

1.2.4.3 Provide a way for an authorized author to permanently remove a SPECIFICATION from the system (e.g., analogous to emptying the trash). The system should require a confirmation from the author prior to performing this action. (See Authorization Foundational USAP with parameters USER=Author and ACTIVITY=Author the SPECIFICATION.)

1.2.4.3.1 Rationale

Authors want to delete a SPECIFICATION that is no longer relevant to their needs.

The organization doesn't want extraneous SPECIFICATIONs on the system.

The system has limited resources.

1.2.4.3.2 Implementing this responsibility

The portion of the system that renders output must render a UI that allows the author to permanently delete a SPECIFICATION.

The portion of the system that accepts input from the user must accept this request.

That portion of the system that manages the authoring process permanently deletes the SPECIFICATION.

The portion of the system that renders output must indicate that the SPECIFICATION has been deleted.

1.2.5 Exit the Authoring System

1.2.5.1 The system must have a way for the author to exit the authoring system for the SPECIFICATION. (If there are requests still pending (e.g., unsaved changes), then notify the author and ask for confirmation of the exit request.)

1.2.5.1.1 Rationale

Software has no way of knowing that an author is finished without explicit input.

Authors want to exit when they are done.

1.2.5.1.2 Implementing this responsibility

The system must provide a means for the author to indicate a desire to exit (e.g., a button, keyboard shortcut, voice command).

The authors must indicate their desire to exit and respond to any confirmation request.

The portions of the system that do the authoring activities must ask for confirmation in the event of request still pending. This involves keeping track of pending requests and requesting the appropriate interactions with the author.

1.3 Execution with Authored Parameters

<u>Purpose:</u>

- The Execution with Authored Parameters USAP's purpose is to allow a system to use a specification of parameters to determine its behavior in the areas in which the parameters apply.

<u>Justification:</u>

- Users want to control the behavior of the computer in certain ways under certain circumstances that they have previously specified (see the Authoring foundational USAP).

<u>Glossary:</u>

- To be completed after user testing

<u>Parameters needed by this USAP:</u>

- SPECIFICATION: The persistent parameter values that have been previously specified.
- APPROPRIATENESS-INFORMATION: The information necessary to locate the appropriate SPECIFICATION.

<u>Shared Assumptions:</u>

Shared with any other USAP that uses the SPECIFICATION:

- The syntax and semantics for the concepts that are included in the SPECIFICATION are defined and known to the developers of both the authoring system and the systems informed by other USAPs that share the SPECIFICATION.
- The protocol for saving is defined and known to the developers of both the authoring system and the systems informed by the Execution with Authored Parameters USAP.
- USAPs sharing this assumption
    - Authoring Foundational USAP

<u>Foundational USAPs used by this USAP:</u>

Authorization Foundational USAP:

- USER: End user
- ACTIVITY: Execute the system with parameter values from SPECIFICATION

(Optionally) Authoring Foundational USAP:

- SPECIFICATION: SPECIFICATION.
- APPROPRIATENESS-INFORMATION: APPROPRIATENESS-INFORMATION.

USAPs that use this Foundational USAP:
User Profile End-user USAP:

- SPECIFICATION: User profile.
- APPROPRIATENESS-INFORMATION: User identity

Environment Configuration End-User USAP:

- SPECIFICATION: Configuration description.
- APPROPRIATENESS-INFORMATION: Environment identity

Alarms and Events End-User USAP:

- SPECIFICATION: Rules for Alarms, Events and Displays.
- APPROPRIATENESS-INFORMATION: Context of use

1.3.1 Access the appropriate SPECIFICATION

1.3.1.1 Retrieve APPROPRIATENESS-INFORMATION (which will allow determination of appropriate SPECIFICATION). (If APPROPRIATENESS-INFORMATION depends on the user then the user must be authorized, see Authorization USAP with parameters USER=End user and ACTIVITY=Execute the system with parameter values from SPECIFICATION.)

1.3.1.1.1 Rationale

The system needs a SPECIFICATION in order to execute appropriately.

System has no way to determine appropriateness of a SPECIFICATION without APPROPRIATENESS-INFORMATION. Sometimes this information may have to come from a user, sometimes it can be inferred from the environment.

Users and organizations want the system to execute appropriately.

1.3.1.1.2 Implementing this responsibility

There must be a portion of the system that knows how to retrieve APPROPRIATENESS-INFORMATION. For example, APPROPRIATENESS-INFORMATION may be maintained in a fixed location within a system or within the file structure of the system.

1.3.1.2 The system must retrieve the appropriate SPECIFICATION.

1.3.1.2.1 Rationale

The system needs a SPECIFICATION in order to execute appropriately.

Users and organizations want the system to execute appropriately.

1.3.1.2.2 Implementing this responsibility

The portion of the system that will use the specified parameters must retrieve the SPECIFICATION.

1.3.1.3 The system should inform the user and/or the system administrator of the results of attempting to retrieve the appropriate SPECIFICATION. There are three cases: find zero, find one, find many. (Typically, finding zero generates an error message, finding one is indicated by allowing the user to proceed, finding many is indicated by listing their identifiers and allowing the user to view the contents of the SPECIFICATION.)

1.3.1.3.1 Rationale

Users want to know if the appropriate SPECIFICATION has been located so they can proceed.

Users want to know if the appropriate SPECIFICATION has not been located because they cannot proceed without it.

Users want to know if more than one appropriate SPECIFICATION has been located because they can help resolve the ambiguity.

System administrators might want to know if the appropriate SPECIFICATION has not been located or if there are many, because it might be an indication of system error.

1.3.1.3.2 Implementing this responsibility

In the event of failing to be located or finding more than one appropriate SPECIFICATION:
The portion of the system that does the locating must provide information about the failure or the identities of each SPECIFICATION.

If informing the system administrator: In the event of a failure or locating more than one appropriate SPECIFICATION, a portion of the system must have a mechanism to inform the system administrator. This may be quite complicated (e.g., sending email) and is beyond the scope of this USAP.

For informing the user:

In the event of a failure: the portion of the system that renders information to the user should display information about the failure.

In the event of a successfully locating an appropriate SPECIFICATION: The portion of the system that renders information to the user displays should indicate success in some way (e.g., progress feedback saying it is retrieving the SPECIFICATION).

In the event of finding many: The portion of the system that renders information to the user should indicate the identifier of each SPECIFICATION.

The portion of the system that renders information to the user should provide a UI for allowing the user to resolve the ambiguity.

The portion of the system that handles input from the user should allow the user to select one of the located SPECIFICATIONs or request to view the contents of one or more of the located SPECIFICATIONs.

The portion of the system that handles input from the user should provide a UI to dismiss any unwanted information.

1.3.1.4 The system must check that the retrieved SPECIFICATION is valid for use.

1.3.1.4.1 Rationale

System cannot use an invalid SPECIFICATION.

1.3.1.4.2 Implementing this responsibility

The portion of the system that will use the specified parameters must check the SPECIFICATION to make sure it is valid (e.g., it could be empty, corrupt, or incomplete).

1.3.1.5 The system should inform the user and/or the system administrator of the results of validating the SPECIFICATION.

1.3.1.5.1 Rationale

Users want to know if the SPECIFICATION is valid so they can proceed.
Users want to know if the SPECIFICATION is not valid because they cannot proceed with an invalid SPECIFICATION.
System administrators might want to know if the SPECIFICATION is not valid, because it might be an indication of system error.

1.3.1.5.2 Implementing this responsibility

In the event of an invalid SPECIFICATION: The portion of the system that does the validation must provide information about the failure.

If informing the system administrator: In the event of a failure, a portion of the system must have a mechanism to inform the system administrator. This may be quite complicated (e.g., sending email) and is beyond the scope of this USAP.

For informing the user:

In the event of an invalid SPECIFICATION: The portion of the system that renders information to the user should display information about the failure.

In the event of a valid SPECIFICATION: The portion of the system that renders information to the user displays should indicate success in some way. This is typically indicated by allowing the user to proceed.

The portion of the system that handles input from the user should provide a UI to dismiss any unwanted information.

1.3.1.6 In the event of a missing, invalid or partially invalid SPECIFICATION, the system should provide the user with options for exiting the system or fixing the problem e.g., offering defaults for values or offering to direct the user to the authoring interface. (If directing the user to the authoring interface, use the Authoring Foundational USAP with parameter SPECIFICATION= SPECIFICATION, APPROPRIATENESS-INFORMATION= APPROPRIATENESS-INFORMATION).

1.3.1.6.1 Rationale

Even though the portion of the system that creates a new SPECIFICATION provides defaults, a SPECIFICATION might have been corrupted.

A system cannot work without a valid SPECIFICATION.
The user wants to use the system, so if the system can’t be used for lack of a valid SPECIFICATION then the user wants to fix it.

1.3.1.6.2 Implementing this responsibility

The portion of the system that renders information to the user should display information about the options.

The portion of the system that handles input from the user should allow the user to select one of the options.

The portion of the system that validates the SPECIFICATION should generate defaults for unspecified or invalid parameters. These defaults should be consistent with the defaults generated by the portion of the system that creates a new SPECIFICATION.

The portion of the system that validates the SPECIFICATION should act on the option selected by the user.

1.3.1.7 The system must provide a means for displaying and dismissing the content of a SPECIFICATION. (This is so they can decide between multiple SPECIFICATIONs or check that the current SPECIFICATION has correct values.) Optionally, when displaying, the system could offer the user access to the authoring interface to modify parameter values. If directing the user to the authoring interface, use the Authoring Foundational USAP with parameter SPECIFICATION= SPECIFICATION, APPROPRIATENESS-INFORMATION= APPROPRIATENESS-INFORMATION).

1.3.1.7.1 Rationale

Users might want to check the parameter values in the SPECIFICATION.

Users might want to modify the parameter values in the SPECIFICATION.

1.3.1.7.2 Implementing this responsibility

The portion of the system that retrieves the SPECIFICATION must provide the parameters and their values.

The portion of the system that renders information to the user should display the parameters and their values.

The portion of the system that handles input from the user should provide a UI to request the display and dismiss any unwanted information.

<u>Optional connection to the authoring system for users with authoring permission:</u>

The portion of the system that handles input from the user should provide a UI to invoke the authoring system (with parameter SPECIFICATION).

The portion of the system that retrieves the SPECIFICATION should invoke the authoring system (with parameter SPECIFICATION).

1.3.2 Use specified parameters

1.3.2.1 The system must apply the specified parameters as necessary for execution. That is, the items specified must be treated as parameters by the system code (i.e., not hard-coded anywhere) and the values must be taken from the SPECIFICATION. (If logging of execution with specified parameters is desired use Logging Foundational USAP with parameters CONTEXT=Execution, SPECIFICATION=SPECIFICATION.)

1.3.2.1.1 Rationale

The entire point of this foundational USAP is that users want to control the behavior of the computer in certain ways under certain circumstances that they have previously specified. Therefore, the system must use the specified parameters.

1.3.2.1.2 Implementing this responsibility

There must be a portion of the system where the parameters that have been specified have some effect.

This portion of the system must treat the parameters as variables rather than as hard-coded values. It must assign the specified parameter values to these variables.

1.3.2.2 The system must provide a UI for the SPECIFICATION to accept operator inputs as necessary. (If logging of operator input is desired use Logging Foundational USAP with parameters CONTEXT=Execution, SPECIFICATION=SPECIFICATION.)

1.3.2.2.1 Rationale

Some of the actions of the SPECIFICATION may require operator input.

1.3.2.2.2 Implementing this responsibility

The portion of the system that renders information to the user should display the request/opportunity for operator input.

The portion of the system that handles input from the user should provide a UI to provide operator input.

There must be a portion of the system that receives and interprets operator input.

1.4 Logging

Purpose:

- The Logging foundational USAP's purpose is to retain and examine selected information generated during execution.

Justification:

- Some information known only during execution needs to be retained either for debugging or audit-trail purposes.

Glossary:

- To be completed after user testing

Parameters needed by this USAP:

- CONTEXT: The activities that generate the events that are logged.

Foundational USAPs used by this USAP:

(Optionally) Authoring Foundational USAP:

- SPECIFICATION: Logging specification.
- APPROPRIATENESS-INFORMATION: Logging state.

  Shared Assumptions:

  Shared with any other USAP that uses the Logging Specification:

  - The development team has defined the syntax and semantics for the concepts that are included in the Logging Specification.
  - The protocol for saving the Logging Specification is defined and known to the developers of both the authoring system and the systems informed by other USAPs that share the Logging Specification.
  - USAPs sharing these assumptions
    - Authoring Foundational USAP
    - Execution with Authored Parameters Foundational USAP

(Optionally) Execution with Authored Parameters Foundational USAP:

- SPECIFICATION: Logging specification.
- APPROPRIATENESS-INFORMATION: Logging state.

  Shared Assumptions:

  Shared with any other USAP that uses the Logging Specification:

  - The development team has defined the syntax and semantics for the concepts that are included in the Logging Specification.
  - The protocol for saving the Logging Specification is defined and known to the developers of both the authoring system and the systems informed by other USAPs that share the Logging Specification.
  - USAPs sharing these assumptions
    - Authoring Foundational USAP
    - Execution with Authored Parameters Foundational USAP

USAPs that use this Foundational USAP:

Authorization foundational USAP has two potential contexts that might produce events to be logged

- CONTEXT: Identification.
- CONTEXT: Authentication.

Authoring foundational USAP
- CONTEXT: Authoring.

Execution with Authored Parameters foundational USAP
- CONTEXT: Execution.

1.4.1 Specify the items to be logged.

1.4.1.1 This could be done either during development (in which case, this is beyond the scope of this USAP) or during or after deployment (in which case, use the Authoring Foundational USAP with parameter SPECIFICATION=Logging specification, APPROPRIATENESS-INFORMATION=Logging state). Ensure that sufficient parameters of the SPECIFICATION are specified for logging so that subsequent analysis is meaningful (e.g., CONTEXT, parameter name, and time stamp). Consider prototyping and testing log information and analysis to ensure sufficiency.

1.4.1.1.1 Rationale

Software must know the information to be logged.

The values in the repository are going to be examined at a later time and these values must be able to be uniquely identified with sufficient information to be useful.

1.4.1.1.2 Implementing this responsibility

<u>If done during development:</u>
Implementation is beyond the scope of this USAP.

<u>If done during or after deployment:</u>
Use the Authoring Foundational USAP with parameters SPECIFICATION=Logging specification and APPROPRIATENESS-INFORMATION=Logging state.

1.4.2 Log items during execution

1.4.2.1 Have a repository in which to store logged items relevant to the SPECIFICATION. This repository could be bounded in size, e.g., circular buffer, or unbounded, e.g., disk file.

1.4.2.1.1 Rationale

The system must have a place to put logged information.

1.4.2.1.2 Implementing this responsibility

There must be a portion of the system that logs information.

The portion of the system that logs information must know the form of the repository and its location and may be responsible for creating the repository.

1.4.2.2 Enter values relevant to the SPECIFICATION into the repository as specified. (If a Logging specification has been Authored, then use the Execution with Authored Parameters Foundational USAP with parameters SPECIFICATION=Logging specification and APPROPRIATENESS-INFORMATION=Logging state, to access the appropriate specification and use it to enter the values into the repository.)

1.4.2.2.1 Rationale

Users need the logged values for debugging or audit trail purposes.

Stored information must persist long enough for analysis to be undertaken.

1.4.2.2.2 Implementing this responsibility

The portion of the system that logs information enters the particular values into the repository.

Attention should be paid to performance considerations since this code may be executed many times.

1.4.3 Post-processing

1.4.3.1 Retrieve items relevant to the SPECIFICATION from the repository. This is typically done some time after the information has been logged, e.g., during the analysis of an anomaly.

1.4.3.1.1 Rationale

Users need information to analyze past events.
The information they need has been stored in the repository and must be retrieved.

1.4.3.1.2 Implementing this responsibility

There must be a portion of the system that knows how to get information out of the repository and does so.

1.4.3.2 Support analysis of retrieved items relevant to the SPECIFICATION by a log analyst (a special type of user).

1.4.3.2.1 Rationale

Users need support to analyze past events.

1.4.3.2.2 Implementing this responsibility

This may be quite complicated (e.g., graphical display, manipulation, mathematical modeling, debugging) and is beyond the scope of this USAP.

2. End-User USAPs

For each End-User USAP, we expect to include

Scenario:

- A story from the end-user's perspective showing the purpose of the end-user USAP.

Usability Benefits:

- Justification for this end-user USAP in terms of usability benefits potentially achieved by implementing this USAP.

For each Foundational USAP used by this USAP (if any), provide:

Parameters furnished to the foundational USAP:

- The parameter values furnished to the foundational USAP used by this USAP.

2.1 Environment Configuration

Scenario:

- An organization wants to supply the same software system to different hardware environments containing different collections of sensors and actuators. A configuration description of the sensors and actuators will allow the system to operate correctly in its environment.

Overview:

- For a software system to be configurable for different environments, actions of the system must be parameterized and the parameter values have to be available at execution time. The values of the parameters must be specified, this configuration description has to be associated with its environment, and the configuration description has to be persistent across sessions.

Glossary:

- To be completed after user testing

Usability Benefits:

- Environment configuration prevents mistakes by tailoring the interface to present only information relevant to the current environment.

Assumptions:

6. There is at least one user who is authorized to author configuration descriptions.

7. The syntax and semantics for the concepts that are included in the configuration description are defined and known to the development team.
8. The protocol for saving the configuration description is defined and known to the development team.
9. Defaults exist for the specifiable parameters.
10. A template exists for authors to use when creating a new configuration description (i.e., the names and definitions of specifiable parameters and their defaults, with optional format).

<u>Foundational USAPs used by this USAP:</u>
Authoring Foundational USAP:
- SPECIFICATION: Configuration description.
- APPROPRIATENESS-INFORMATION: Environment identity

Execution with Authored Parameters Foundational USAP:
- SPECIFICATION: Configuration description.
- APPROPRIATENESS-INFORMATION: Environment identity

2.1.1 Author Configuration Description

Use Authoring Foundational USAP with SPECIFICATION= Configuration description, APPROPRIATENESS-INFORMATION= Environment identity.

Additional responsibilities beyond those inherited from foundational USAPs
None.

Specializations are as follows.
None.

2.1.2 Execute with Authored Configuration Description

Use Execution with Authored Parameters Foundational USAP with SPECIFICATION= Configuration description, APPROPRIATENESS-INFORMATION= Environment identity.

Additional responsibilities beyond those inherited from foundational USAPs
None.

Specializations are as follows.

1.3.1.1 Retrieve APPROPRIATENESS-INFORMATION (which will allow determination of appropriate SPECIFICATION) (If APPROPRIATENESS-INFORMATION depends on the user then the user must be authorized. See Authorization USAP with parameters USER=End user and ACTIVITY=Execute the system with parameter values from SPECIFICATION.) **For the Environment Configuration End-User USAP, the**

APPROPRIATENESS-INFORMATION=environment identity, so there is no need for authorization. In many cases, APPROPRIATENESS-INFORMATION may not need to be retrieved at all because the location of the configuration description is built-in (e.g., config.dat at a known location). If so, this responsibility is not applicable.

1.3.1.1.1 The system must check that the retrieved SPECIFICATION is valid for use.
For the Environment Configuration End-User USAP, the validity depends on the consistency between the Configuration Description and the physical reality of execution environment. Thus, consistency involves checking hardware. For example, sensors may need to be polled to verify that they are currently present and working.

2.2 User Profile

Scenario:

- A user wishes to have the capabilities of the system personalized to reflect his or her preferences or role. The capabilities that can be personalized may include language, access to system functionality, display characteristics, account information or any preference that might vary among users or roles.

Overview:

- For a user profile to work, configurable actions of the system must be parameterized and the parameter values have to be available at execution time. The values of the parameters must be specified, this specification (the "user profile") has to be associated with the user and persist across sessions.

Glossary:

- To be completed after user testing

Usability Benefits:

- User profile accelerates error-free portion of routine performance by providing information in a familiar and individually-tailored form, providing user-defined hot-keys and allowing common operations to be easily accessible.
- User profile prevents mistakes by simplifying the interface to that which is familiar and necessary. This prevents infrequent users from making the types of mistakes caused by too many options.
- User profile accommodates mistakes by enabling the user to disallow certain options (e.g., disabling the tap-to-click option on a track pad).
- User profile increases user confidence and comfort by providing an individualized interface.

Assumptions:

1. There is at least one user who is authorized to author user profiles.

2. The syntax and semantics for the concepts that are included in the User Profile are defined and known to the development team..
3. The protocol for saving the User Profile is defined and known to the development team.
4. Defaults exist for the specifiable parameters.
5. A template exists for authors to use when creating a new user profile (i.e., the names and definitions of specifiable parameters and their defaults, with optional format).

Foundational USAPs used by this USAP:
Authoring Foundational USAP:

- SPECIFICATION: User profile
- APPROPRIATENESS-INFORMATION: User identity.

Execution with Authored Parameters Foundational USAP:

- SPECIFICATION: User profile
- APPROPRIATENESS-INFORMATION: User identity

2.2.1 Author User Profile

Use Authoring Foundational USAP with SPECIFICATION=User Profile, APPROPRIATENESS-INFORMATION=User identity.

Additional responsibilities beyond those inherited from foundational USAPs
None.

Specializations are as follows.
None.

2.2.2 Execute with Authored User Profile

Use Execution with Authored Parameters Foundational USAP with SPECIFICATION=User Profile and APPROPRIATENESS-INFORMATION=User identity.

Additional responsibilities beyond those inherited from foundational USAPs
None.

Specializations are as follows.

1.3.1.1 Retrieve APPROPRIATENESS-INFORMATION (which will allow determination of appropriate SPECIFICATION) (If APPROPRIATENESS-INFORMATION depends on the user then the user must be authorized. See Authorization USAP with parameters USER=End user and ACTIVITY=Execute the system with parameter values from SPECIFICATION.)
For the User Profile End-User USAP, the APPROPRIATENESS-

INFORMATION=user identity, so use the AUTHORIZATION foundational USAP with USER=End user and ACTIVITY=Execute the system with parameter values from user profile.

2.3 Alarms and Events

Scenario:
The user needs feedback from the system when an error occurred or a specific condition is met. The user can be the operator of the system or a superior system. The feedback can be needed for safety reasons, diagnostic, problem solving or information purposes.

Overview:
Alarm, Events and Messages Management System
EEMUA describes how Alarm & Events are important in the control of plant and machinery. Alarm and Event systems form an essential part of the operator interfaces in large modern industrial systems. They provide vital support to the operators managing these complex systems by warning them of situations that need their attention.
Alarms are signals which are annunciated to the operator typically by an audible sound, some sort of visual indication, usually flashing, and by the presentation of a message or some other identifier. An alarm will indicate a problem requiring operator attention, and is generally initiated by a process measurement passing a defined alarm setting as it approaches an undesirable or potentially unsafe value. Alarm systems help the operator;

- to maintain the plant within safe operating envelope;
- to recognize and act to avoid hazardous situations;
- to identify deviations from desired operating conditions that could lead to financial loss;
- to better understand complex process conditions. Alarms should be an important diagnostic tool, and are one of several sources that an operator uses during an upset.

The terms alarm and event are often used interchangeably and their meanings are not distinct. An alarm is an abnormal condition that requires special attention. An event may or may not be associated with a condition. For example, the transitions into the level alarm condition and the return to normal are events which are associated with conditions. However, operator actions, system configuration changes, and system errors are examples of events which are not related to specific conditions.

Alarm processing and handling
The Norwegian Petroleum Directorate(NPD) has identified alarm processing and handling concepts described in Figure 1 and used in this document;

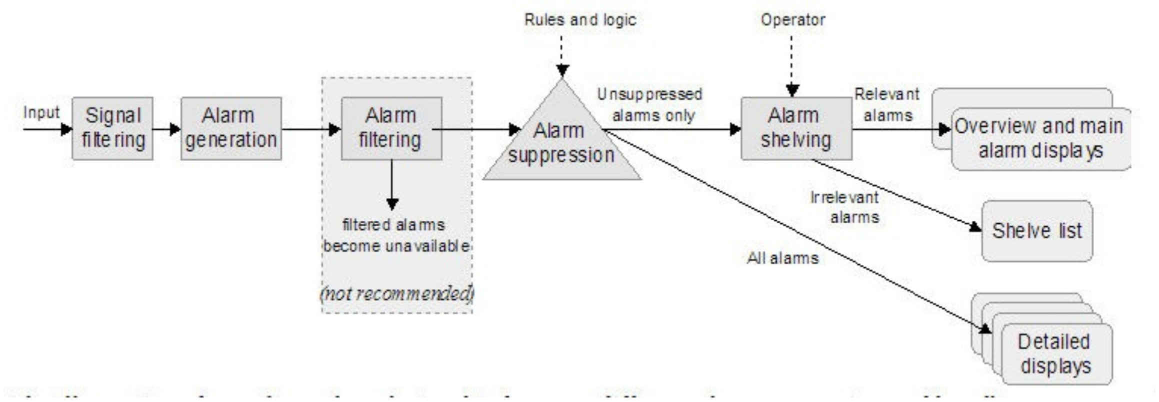


Figure 1. NPD's definition of alarm processing.

- Alarm generation means generating an alarm according to some defined rules.
- Alarm filtering means preventing an alarm signal so that it is not available for the operator in any part of the system.
- Alarm suppression means preventing an alarm from being presented in main alarm displays, e.g., overview displays, but the alarm is still available in the system at a more detailed level.
- Alarm shelving is a facility for manually removing an alarm from the main list and placing it on a shelve list, temporarily preventing the alarm from re-occurring on the main list until it is removed form the shelf. Shelving will normally be controlled by the operator, and is intended as a "last resort" for handling irrelevant nuisance alarms that have not been caught by signal filtering or alarm suppression mechanisms.

Figure 2 shows the actions which cause transitions between the states that a displayed alarm may have according to EEMUA.

Other definitions

- Alarm prioritization is a categorization of alarms based on the importance of each alarm for the operator tasks.
- Overview displays are designed to help operators get an overview of the state of the process. Overview displays include: Main alarm lists, tiles or enunciator alarm displays, as well as large screen displays showing key information.
- Selective lists show only a selection of the available alarm information, based on selection and sorting criteria specified by operators. For more definitions see Table 1.

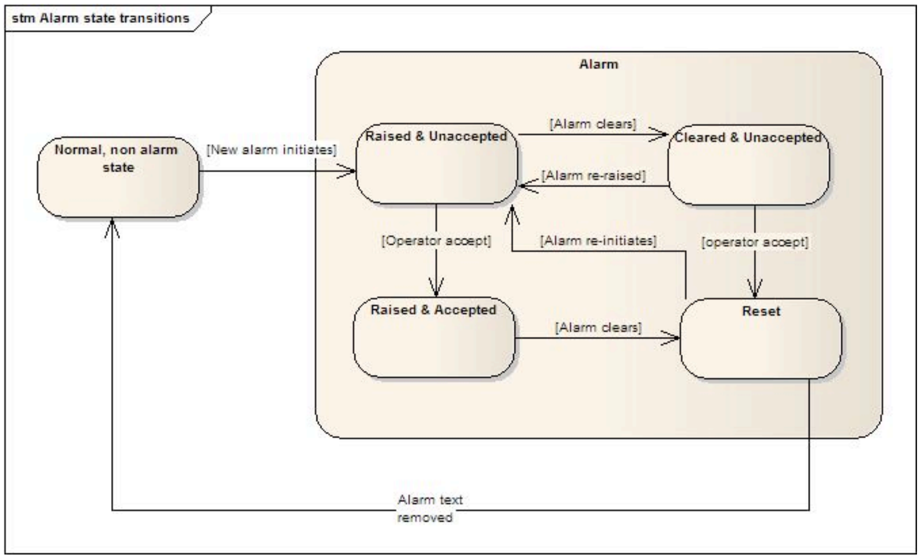


Figure 2. EEMUA's definitions of alarm state transitions.

References

[1] EEMUA 191: Alarm Systems. A Guide to Design, Management and Procurement. 1999, ISBN 0 8593 1076 0 (http://www.eemua.co.uk).
[2] Norwegian Petroleum Directorate YA-711: Principles for alarm system design, 2001 (http://www.ptil.no/regelverk/R2002/ALARM_SYSTEM_DESIGN_E.HTM).
[3] MSDN library, Windows Vista User Experience
[4] M.Hollender, Beuthel, C.: Intelligent alarming, Effective alarm management improves safety, fault diagnosis and quality control, ABB Review 1, 2007.

Glossary:

The following definitions are used in this document. <insert Pia's table)

Usability Benefits:

- Reduces impact of slips by informing the user about the deviation from normal procedure.
- Supports problem solving by helping user understand the problem and contributes to the solution of the problem.
- Facilitate learning by helping the user to understand and learn the consequences of actions
- Action confirmation feedback prevents the user from going any further with a mistake. The feedback acts as a gatekeeper, and need an acknowledgement to go ahead with the procedure.
- Feedback helps user tolerate system error by informing the user that there is a problem and possible cause of the problem.
- Being able to solve a problem with help of a system's feedback will increase user confidence by supporting the user performing his/her work. It will also build trust for the system as it always presents accurate and helpful feedback.

Assumptions:

1 There is at least one user who is authorized to author rules for alarms, events and display.
2 The syntax and semantics for the concepts that are included in the Alarms&Events specification are defined and known to the development team. These concepts may include values, logic, properties, etc. to control the behavior of alarms and events. These concepts must support the customers current alarm philosophy and the relevant standards that apply for the systems customers. See AlarmLanguage_BassJohnGolden.doc for an example inspired by the Norwegian Petroleum Directorate YA-711: Principles for alarm system design, 2001 (http://www.ptil.no/regelverk/R2002/ALARM_SYSTEM_DESIGN_E.HTM).
3 The development team has defined presentation conventions that are salient to the user, e.g, fonts, icons, colors. These conventions must differentiate alarms, events, and messages from each other.
4 The protocol for saving an Alarms&Events specification is defined and known to the development team.
5 There is no more than one Alarms&Events specification per context of use. That is, all of the rules are bundled into one specification that the system loads when it is executed in a particular context. So, for example, there may be a single specification for normal operation of the system and a different single specification for diagnosis procedures.
6 The development team has decided on an appropriate protocol for concurrent users. That is, can each user be completely autonomous (e.g., being able to dismiss or suppress alarms), is one user the "master" and all the rest can only observe, etc.
7 The development team has decided which items will be logged during execution. For example, these items might include: state transitions of an alarm, the event of failure of the alarm generation routine, the alarm generation rate,

Foundational USAPs used by this USAP:

Authoring Foundational USAP:

- SPECIFICATION: Rules for Alarms, Events and Displays
- APPROPRIATENESS-INFORMATION: Context of use.

Execution with Authored Parameters Foundational USAP:

- SPECIFICATION: Rules for Alarms, Events and Displays
- APPROPRIATENESS-INFORMATION: Context of use.

2.3.1 Author Rules for Alarms, Events and Displays

Use Authoring Foundational USAP with SPECIFICATION=Rules for Alarms, Events and Displays, APPROPRIATENESS-INFORMATION= Context of use. By "Context of use" we mean things like "in operation" "in maintenance" "in diagnosis".

Additional responsibilities beyond those inherited from foundational USAPs

None.

Specializations are as follows.

1.2.1.1 The system must provide a way for an authorized author to create a SPECIFICATION. (See Authorization Foundational USAP with parameters USER=Author and ACTIVITY=Author the SPECIFICATION.) For the Alarms and Events End-User USAP, there is an assumption that a language has been defined that describes the parameters and their interactions (Assumption 1). The specification being created must conform to this language.

1.2.2 Save a SPECIFICATION. The system must provide a means for the SPECIFICATION to be saved and/or exported (e.g., by autosave or by author request). If other systems are going to use the SPECIFICATION, then use a format that can be used by the other systems. (If logging of authoring results is desired use Logging Foundational USAP with parameter CONTEXT=Authoring.) ***For the Alarms & Events End-User USAP logging authoring results may be needed for regulatory purposes. If this is the case, then use the Logging Foundational USAP with Context=Authoring.***

> *Display the current parameter values (including identifier and circumstances).*
> ***For the Alarms & Events End-User USAP the system must not only display the current parameter settings, but also display some of the implications of some of the current parameter settings. For example, the system must display the current priority distribution of the alarms during authoring of the setting of the alarm priority. E.g., "5% of the configured alarms are high priority". [Section 2.5. #27, p. 15 Norwegian Petroleum Directorate YA-711: Principles for alarm system design, 2001 (http://www.ptil.no/regelverk/R2002/ALARM_SYSTEM_DESIGN_E.HTM).***
>
> The system must provide a ways for an authorized author to change the parameter values…
> ***For the Alarms & Events End-User USAP the authorization information to modify a rule is a property of the rule. Therefore use the Authorization Foundational USAP with parameters USER=Author and ACTIVITY=Modify a particular rule.***
>
> The system must permit or prohibit specific ACTIVITIES dependent on who the USER is and on the set of ACTIVITIES they are attempting to perform, i.e., the system must know (built in, stored, provided by the OS, etc.) the specific permissions for each USER and then enforce these permissions
> ***For the Alarms & Events End-User USAP the permission information to modify a rule is a property of a rule and the***

system must know how to retrieve the permission information from that rule.

2.3.2 Execute with Authored Rules for Alarms, Events and Displays

Use Execution with Authored Parameters Foundational USAP with SPECIFICATION=Rules for Alarms, Events and Displays and APPROPRIATENESS-INFORMATION=Context of use.

Additional responsibilities beyond those inherited from foundational USAPs

2.3.2.1 The system must permit multiple users to operate simultaneously in accordance with the protocol defined in the assumptions, e.g., each user could have the ability to have their own display filter settings.

2.3.2.1.1 Rationale

Some large systems may require more than one operator to function safely.

Each operator may want to sort the alarm list display according to different criteria depending on his/hers current task or preferences.

2.3.2.1.2 Implementing this responsibility

The portion of the system that stores system data must be shareable among multiple users.

The portion of the system that manages system data must synchronize among multiple users to avoid simultaneous update of system data.

The portion of the system that manages system data must implement a protocol that determines what the system must do in the event that two users simultaneously issue conflicting commands. Consider informing the users in the event of a conflict as a portion of the protocol.

The portion of the system that manages user-specific data must be thread safe (e.g., re-entrant)

The portion of the system that interacts with the users must be thread safe.

2.3.2.2 The system must have the ability to translate the names/ids of externally generated signals, e.g., from a sensor, into the concepts that are included in the Alarms&Events specification.

2.3.2.2.1 Rationale

The environment contains sensors that generate and actuators that respond to analog or digital signals in their own form.

The alarm and event portion of the system can only operate with logical concepts.

2.3.2.2.2 Implementing this responsibility

There should be a portion of the system (e.g., an intermediary) that sits between those portions of the system that interact directly with sensors and actuators (e.g., the device drivers) and the portion of the system that implements the alarm and event logic.

This intermediary should translate between the signals by the sensors and actuators and the logical concepts required by alarm and event rules.

2.3.2.3 The system must have the ability to broadcast a generated event so that an external system can use it. E.g., an external long-time storage system.

2.3.2.3.1 Rationale

Some events require informing external people or systems. For example, an explosion may need to call emergency responders.

2.3.2.3.2 Implementing this responsibility

The portion of the system that executes the alarm and event rules must have the capability to broadcast to appropriate external systems.

2.3.2.4 The system must have the ability to present alarm state transitions in the alarm displays within the time restrictions valid for this system.

2.3.2.4.1 Rationale

Users have limits as to how fast they can operate, i.e., perceive information, comprehend information, make decisions, and perform motor actions. This imposes a lower bound on how long information has to be displayed (visual or auditory).

Because the alarm system may supervise hazardous environments, safety regulations may require specific response times. This imposes an upper bound on the number of human actions that can be required to respond to alarms.

Any claims made for the operator action in response to alarms should be based upon sound human performance data and principles.

2.3.2.4.2 Implementing this responsibility

The portion of the system that displays information to the user should ensure that information is displayed long enough for a person to see or hear it. This requires that this portion of the system maintain timing information of how long information has been displayed and ensure that it is longer than minimal human perceptual limits.

The portion of the system that does the scheduling should schedule alarms as high-priority activities.

The portion of the system that interacts with the user during emergency situations must be designed to meet the response time requirements. This is the responsibility of the UI designers and does not impose additional architectural requirements.

2.3.2.5 The system must have sufficient persistent storage for alarms, rules and data to be saved. It may be acceptable to limit the number of events to be saved.

2.3.2.5.1 Rationale

Regulations might require long-term storage of alarms, rules and data (e.g., in the food and drug business).

An organization may have publication, notification, history, fault diagnostic needs that require persistent data.

The system may be stopped and started again and the rules and data must not get lost when this happens.

There might be a lot of data.

Storage media has cost (hardware, time to read and write, network bandwidth).

2.3.2.5.2 Implementing this responsibility

The portion of the system that manages persistent data must ensure that the most recent and the most important data is not lost. This could be done by having large persistent data stores; it could be done by overwriting older data with newer data.

The portion of the system that manages persistent data must have a protocol to determine which data gets overwritten when persistent storage is almost full.

Additional responsibilities beyond those inherited from foundational USAPs

1.3.1.1 Retrieve APPROPRIATENESS-INFORMATION (which will allow determination of appropriate SPECIFICATION) (If APPROPRIATENESS-INFORMATION depends on the user then the user must be authorized. See Authorization USAP with parameters USER=End user and ACTIVITY=Execute the system with parameter values from SPECIFICATION.)
For the Alarms & Events End-User USAP the APPROPRIATENESS-INFORMATION may include not only the Context of use, but in the case of concurrent users also who is using a display. Therefore also use the Authorization Foundational USAP with parameters USER=End-user and ACTIVITY=Execute with Rules for Alarms, Events and Displays.

1.3.2.1 The system must apply the specified parameters as necessary for execution. That is, the items specified must be treated as parameters by the system code (i.e., not hard-coded anywhere) and the values must be taken from the SPECIFICATION. (If logging of execution with specified parameters is desired use Logging Foundational USAP with parameters CONTEXT=Execution.)
For the Alarms & Events End-User USAP, the information to be logged must be in accordance with the assumptions about logged information. Therefore use the Logging Foundational USAP with parameters CONTEXT=Execution.

Other things to consider when designing alarms, events and displays:

Provide the ability to support native languages.

Provide context-sensitive help for instances of alarms, events and messages guided by their unique specification identity.

Present multiple views of the alarm displays. For example, one view could include N number of raised alarms, another could include all raised alarms since the timestamp of the oldest raised and non-cleared alarm.
Give feedback on user actions within 150 ms. Feedback must be appropriate to the manner in which the command was issued. For example, if the user pressed a button, changing the color of the button would indicate the user feedback.

Appendix K: Responsibilities used in ABB User Tests.

Authoring

AU.1. Create a Specification

AU.1.1** **The system must provide a way for an authorized author to create a [User Profile, Configuration Description, Conditions for Alarms, Events and Displays].

Rationale

A [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] doesn't exist unless it is created.

Implementing this responsibility

The portion of the system that renders output must render a UI that allows the parameters to be specified and displays existing values.

The portion of the system that accepts input from the user must accept parameters.

There must be a portion of the system with a mechanism to create new [User Profile, Configuration Description, Conditions for Alarms, Events and Displays].

AU.1.2** **Consider providing default values for all specifiable parameters when a [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] is created. (Providing defaults simplifies the creation process, but may increase the probability of error for environments that deviate from those defaults. If the cost of error is high, then consider not providing defaults or requiring confirmation of each value. Default values can be changed through modification.)

Rationale

System has to have *something* so defaults might be provided.
Circumstances might be very similar so defaults might capture that similarity.
Authors may want to be efficient and defaults may save authoring time.
Authors may only be interested in changing specific aspects of the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays], so having defaults for the rest of it is useful.

Implementing this responsibility

The portion of the system that creates a new [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] must assign defaults.

AU.1.3 The [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] must be given an identifier. The identifier should be treated as a parameter that has a default value and can be modified by the user. (One mechanism for assigning this identification might be a "Save as".)

Rationale

Circumstances might be very similar so values in one [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] may transfer to other circumstances.

Authors may want to be efficient and may want to begin the specification process with a similar [User Profile, Configuration Description, Conditions for Alarms, Events and Displays].

The authoring system must have an identifier to distinguish one [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] from another.

Implementing this responsibility

<u>If the identifier is provided by a author:</u>
The portion of the system that renders output must render a UI that allows the author to provide an identifier and display it.

The portion of the system that accepts input from the user must accept the identifier.

There must be a portion of the system that manages the authoring process.

The portion of the system that manages the authoring process must associate the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] with the identifier.

<u>If the identifier is provided automatically by the system:</u>
The portion of the system that creates the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] generates an identifier (e.g., MSWord automatically generates a name for a new document).

The portion of the system that manages the authoring process must associate the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] with the identifier.

AU.2. Save a Specification.

AU.2.1. The system must provide a means for an authorized author to save and/or export the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] (e.g., by autosave or by author request). If other systems are going to use the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays], then use a format that can be used by the other systems.

Rationale

Authors want to be efficient (i.e., input information into the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] only once).

The system can only remember things if they are persistent from session to session.

The software may need to share a [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] with other software.

Implementing this responsibility

<u>If the initiation of the save was automatic:</u>
That portion of the system that manages the authoring process performs the initiation.

That portion of the system that manages the authoring process stores and/or exports the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays].

<u>If the initiation of the save was at the author's request:</u>
The portion of the system that renders output must render a UI that allows the parameters needed by the system (e.g., format, location) to be input and display them.

The portion of the system that accepts input from the user must accept the parameters.

That portion of the system that manages the authoring process stores and/or exports the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays].

AU.3. Modify a SPECIFICATION

AU.3.1 Provide a way for an authorized author to retrieve a [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] (e.g., import a previously-saved file, utilize a previously-generated data structure, or restore to default values).

Rationale

Authors might want different values than are currently assigned.

System has stored the information and must have a current set of data to work with.

Implementing this responsibility

The portion of the system that renders output must render a UI that allows the author to request a retrieval of a [User Profile, Configuration Description, Conditions for Alarms, Events and Displays].

The portion of the system that accepts input from the user must accept this request.

That portion of the system that manages the authoring process retrieves the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays].

AU.3.2 Display the current parameter values for the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] (including identifier and circumstances).

Rationale

Authors want to see what they are editing.

Implementing this responsibility

That portion of the system that manages the authoring process must provide the values.

The portion of the system that renders output must render a UI that displays the values.

The portion of the system that handles input from the user should provide a UI to dismiss any unwanted information.

AU.3.3 The system must provide a ways for an authorized author to change the parameter values. The syntax and semantics of the values specified

should conform to the assumptions of the execution environment of the system. (More details: Best practice is to constrain the author to such conformation (e.g., choose from drop-down list, provide a slider for a range of values). If the author's choices are not constrained, then the system should check the syntax and semantics and provide feedback if either are unreasonable.)

Rationale

Authors might want different values than are currently assigned.

Implementing this responsibility

The portion of the system that renders output must render a UI that allows values to be changed.

The portion of the system that accepts input from the user must accept new values.

That portion of the system that manages the authoring process replaces the current values with the new values.

AU.4. Delete a SPECIFICATION

AU.4.1 Provide a way for an authorized author to tentatively remove a [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] from the system (e.g., analogous to dragging a file to the trash). The system might require a confirmation from the author prior to performing this action.

Rationale

Authors might accidentally delete a [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] and want to restore it.

The organization doesn't want extraneous [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] on the system.

The system has to keep it around in case it has to be restored.

Implementing this responsibility

The portion of the system that renders output must render a UI that allows the author to tentatively delete a [User Profile, Configuration Description, Conditions for Alarms, Events and Displays].

The portion of the system that accepts input from the user must accept this request.

That portion of the system that manages the authoring process saves the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] in case has to be restored.

The portion of the system that renders output must indicate that the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] has been (tentatively) deleted.

AU.4.2 Retrieve a tentatively-deleted [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] from the system (e.g., analogous to dragging a file out of the trash)

Rationale

Authors might want to restore an accidentally deleted [User Profile, Configuration Description, Conditions for Alarms, Events and Displays].

The system has kept it around so that it can be restored.

Implementing this responsibility

The portion of the system that renders output must render a UI that allows the author to restore a tentatively deleted [User Profile, Configuration Description, Conditions for Alarms, Events and Displays].

The portion of the system that accepts input from the user must accept this request.

That portion of the system that manages the authoring process must restore the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays].

The portion of the system that renders output must indicate that the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] has been restored.

AU.4.3 Provide a way for an authorized author to permanently remove a [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] from the system (e.g., analogous to emptying the trash). The system should require a confirmation from the author prior to performing this action.

Rationale

Authors want to delete a [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] that is no longer relevant to their needs.

The organization doesn't want extraneous [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] on the system.

The system has limited resources.

Implementing this responsibility

The portion of the system that renders output must render a UI that allows the author to permanently delete a [User Profile, Configuration Description, Conditions for Alarms, Events and Displays].

The portion of the system that accepts input from the user must accept this request.

That portion of the system that manages the authoring process permanently deletes the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays].

The portion of the system that renders output must indicate that the [User Profile, Configuration Description, Conditions for Alarms, Events and Displays] has been deleted.

AU.5. Exit the Authoring System

AU.5.1 The system must have a way for the author to exit the authoring system. (If there are requests still pending (e.g., unsaved changes), then notify the author and ask for confirmation of the exit request.)

Rationale

Software has no way of knowing that an author is finished without explicit input.

Authors want to exit when they are done.

Implementing this responsibility

The system must provide a means for the author to indicate a desire to exit (e.g., a button, keyboard shortcut, voice command).

The authors must indicate their desire to exit and respond to any confirmation request.

The portions of the system that do the authoring activities must ask for confirmation in the event of request still pending. This involves keeping track of pending requests and requesting the appropriate interactions with the author.

Execution with Authored Parameters

EX.1. Access the Appropriate SPECIFICATION

EX.1.1 Retrieve [User identity, Environment identity, Context of use] (which will allow determination of appropriate [User profile, Configuration description, Rules for Alarms, Events and Displays]). In many cases, Environment identity may not need to be retrieved at all because the location of the configuration description is built-in (e.g., config.dat at a known location). If so, this responsibility is not applicable for Configuration Description.

Rationale

The system needs a [User profile, Configuration description, Rules for Alarms, Events and Displays] in order to execute appropriately.

System has no way to determine appropriateness of a [User profile, Configuration description, Rules for Alarms, Events and Displays] without [User identity, Environment identity, Context of use]. Sometimes this information may have to come from a user, sometimes it can be inferred from the environment.

Users and organizations want the system to execute appropriately.

Implementing this responsibility

There must be a portion of the system that knows how to retrieve [User identity, Environment identity, Context of use]. For example, [User identity, Environment identity, Context of use] may be maintained in a fixed location within a system or within the file structure of the system.

EX.1.2 The system must retrieve the appropriate a [User profile, Configuration description, Rules for Alarms, Events and Displays].

Rationale

The system needs a [User profile, Configuration description, Rules for Alarms, Events and Displays] in order to execute appropriately.

Users and organizations want the system to execute appropriately.

Implementing this responsibility

The portion of the system that will use the specified parameters must retrieve the [User profile, Configuration description, Rules for Alarms, Events and Displays].

EX.1.3 The system should inform the user and/or the system administrator of the results of attempting to retrieve the appropriate [User profile, Configuration description, Rules for Alarms, Events and Displays]. There are three cases: find zero, find one, find many. (Typically, finding zero generates an error message, finding one is indicated by allowing the user to proceed, finding many is indicated by listing their identifiers and allowing the user to view the contents of the [User profile, Configuration description, Rules for Alarms, Events and Displays].)

Rationale

Users want to know if the appropriate [User profile, Configuration description, Rules for Alarms, Events and Displays] has been located so they can proceed.

Users want to know if the appropriate [User profile, Configuration description, Rules for Alarms, Events and Displays] has not been located because they cannot proceed without it.

Users want to know if more than one appropriate [User profile, Configuration description, Rules for Alarms, Events and Displays] has been located because they can help resolve the ambiguity.

System administrators might want to know if the appropriate [User profile, Configuration description, Rules for Alarms, Events and Displays] has not been located or if there are many, because it might be an indication of system error.

Implementing this responsibility

In the event of failing to be located or finding more than one appropriate [User profile, Configuration description, Rules for Alarms, Events and Displays]:

The portion of the system that does the locating must provide information about the failure or the identities of each [User profile, Configuration description, Rules for Alarms, Events and Displays].

If informing the system administrator:
In the event of a failure or locating more than one appropriate [User profile, Configuration description, Rules for Alarms, Events and Displays], a portion of the system must have a mechanism to inform the

system administrator. This may be quite complicated (e.g., sending email) and is beyond the scope of this USAP.

For informing the user:
In the event of a failure: The portion of the system that renders information to the user should display information about the failure.

In the event of a successfully locating an appropriate [User profile, Configuration description, Rules for Alarms, Events and Displays]: The portion of the system that renders information to the user displays should indicate success in some way (e.g., progress feedback saying it is retrieving the [User profile, Configuration description, Rules for Alarms, Events and Displays]).

In the event of finding many:
The portion of the system that renders information to the user should indicate the identifier of each [User profile, Configuration description, Rules for Alarms, Events and Displays].

The portion of the system that renders information to the user should provide a UI for allowing the user to resolve the ambiguity.

The portion of the system that handles input from the user should allow the user to select one of the located [User profile, Configuration description, Rules for Alarms, Events and Displays] or request to view the contents of one or more of the located [User profile, Configuration description, Rules for Alarms, Events and Displays].

The portion of the system that handles input from the user should provide a UI to dismiss any unwanted information.

EX.1.4 The system must check that the retrieved [User profile, Configuration description, Rules for Alarms, Events and Displays] is valid for use. For Configuration description, the validity depends on the consistency between the Configuration description and the physical reality of execution environment. Thus, consistency involves checking hardware. For example, sensors may need to be polled to verify that they are currently present and working.

Rationale

System cannot use an invalid [User profile, Configuration description, Rules for Alarms, Events and Displays].

Implementing this responsibility

The portion of the system that will use the specified parameters must check the [User profile, Configuration description, Rules for Alarms, Events and Displays] to make sure it is valid (e.g., it could be empty, corrupt, or incomplete).

EX.1.5 The system should inform the user and/or the system administrator of the results of validating the [User profile, Configuration description, Rules for Alarms, Events and Displays].

Rationale

Users want to know if the [User profile, Configuration description, Rules for Alarms, Events and Displays] is valid so they can proceed.

Users want to know if the [User profile, Configuration description, Rules for Alarms, Events and Displays] is not valid because they cannot proceed with an invalid [User profile, Configuration description, Rules for Alarms, Events and Displays].

System administrators might want to know if the [User profile, Configuration description, Rules for Alarms, Events and Displays] is not valid, because it might be an indication of system error.

Implementing this responsibility

In the event of an invalid [User profile, Configuration description, Rules for Alarms, Events and Displays]: The portion of the system that does the validation must provide information about the failure.

If informing the system administrator:
In the event of a failure, a portion of the system must have a mechanism to inform the system administrator. This may be quite complicated (e.g., sending email) and is beyond the scope of this USAP.

For informing the user:
In the event of an invalid [User profile, Configuration description, Rules for Alarms, Events and Displays]:
The portion of the system that renders information to the user should display information about the failure.

In the event of a valid [User profile, Configuration description, Rules for Alarms, Events and Displays]:
The portion of the system that renders information to the user displays should indicate success in some way. This is typically indicated by allowing the user to proceed.

The portion of the system that handles input from the user should provide a UI to dismiss any unwanted information.

EX.1.6 In the event of a missing, invalid or partially invalid [User profile, Configuration description, Rules for Alarms, Events and Displays], the system should provide the user with options for exiting the system or fixing the problem e.g., offering defaults for values or offering to direct the user to the authoring interface.

Rationale

Even though the portion of the system that creates a new [User profile, Configuration description, Rules for Alarms, Events and Displays] provides defaults, a [User profile, Configuration description, Rules for Alarms, Events and Displays] might have been corrupted.

A system cannot work without a valid [User profile, Configuration description, Rules for Alarms, Events and Displays].

The user wants to use the system, so if the system can't be used for lack of a valid [User profile, Configuration description, Rules for Alarms, Events and Displays] then the user wants to fix it.

Implementing this responsibility

The portion of the system that renders information to the user should display information about the options.

The portion of the system that handles input from the user should allow the user to select one of the options.

The portion of the system that validates the [User profile, Configuration description, Rules for Alarms, Events and Displays] should generate defaults for unspecified or invalid parameters. These defaults should be consistent with the defaults generated by the portion of the system that creates a new [User profile, Configuration description, Rules for Alarms, Events and Displays].

The portion of the system that validates the [User profile, Configuration description, Rules for Alarms, Events and Displays] should act on the option selected by the user.

EX.1.7 The system must provide a means for displaying and dismissing the content of a [User profile, Configuration description, Rules for Alarms, Events and Displays]. (This is so they can decide between multiple [User profile, Configuration description, Rules for Alarms, Events and Displays]or check that the current [User profile, Configuration description, Rules for Alarms, Events and Displays]has correct values.) Optionally, when displaying, the system could offer the user access to the authoring interface to modify parameter values.

Rationale

Users might want to check the parameter values in the [User profile, Configuration description, Rules for Alarms, Events and Displays].

Users might want to modify the parameter values in the [User profile, Configuration description, Rules for Alarms, Events and Displays].

Implementing this responsibility

The portion of the system that retrieves the [User profile, Configuration description, Rules for Alarms, Events and Displays]must provide the parameters and their values.

The portion of the system that renders information to the user should display the parameters and their values.

The portion of the system that handles input from the user should provide a UI to request the display and dismiss any unwanted information.
<u>Optional connection to the authoring system for users with authoring permission:</u>

The portion of the system that handles input from the user should provide a UI to invoke the authoring system (with parameter [User profile, Configuration description, Rules for Alarms, Events and Displays]).

The portion of the system that retrieves the [User profile, Configuration description, Rules for Alarms, Events and Displays]should invoke the authoring system (with parameter [User profile, Configuration description, Rules for Alarms, Events and Displays]).

EX.2. Use Specified Parameters

EX.2.1 The system must apply the specified parameters as necessary for execution. That is, the items specified must be treated as parameters by the system code (i.e., not hard-coded anywhere) and the values must be taken from the [User profile, Configuration description, Rules for Alarms, Events and Displays]. If logging of execution with specified parameters is desired, see Logging responsibilities.

Rationale

The entire point of this foundational USAP is that users want to control the behavior of the computer in certain ways under certain circumstances that they have previously specified. Therefore, the system must use the specified parameters.

Implementing this responsibility

There must be a portion of the system where the parameters that have been specified have some effect.

This portion of the system must treat the parameters as variables rather than as hard-coded values. It must assign the specified parameter values to these variables.

EX.2.2 ***The system must provide a UI for the [User profile, Configuration description, Rules for Alarms, Events and Displays] to accept operator inputs as necessary. If logging of operator input is desired, see Logging responsibilities.***

Rationale

Some of the actions of the [User profile, Configuration description, Rules for Alarms, Events and Displays] may require operator input.

Implementing this responsibility

The portion of the system that renders information to the user should display the request/opportunity for operator input.

The portion of the system that handles input from the user should provide a UI to provide operator input.

There must be a portion of the system that receives and interprets operator input.

EX.3. Additional Responsibilities for Execution

EX.3.1 ***The system must permit multiple users to operate simultaneously in accordance with the protocol defined in the assumptions, e.g., each user could have the ability to have their own display filter settings.***

Rationale

Some large systems may require more than one operator to function safely.

Each operator may want to sort the alarm list display according to different criteria depending on his/hers current task or preferences.

Implementing this responsibility

The portion of the system that stores system data must be shareable among multiple users.

The portion of the system that manages system data must synchronize among multiple users to avoid simultaneous update of system data.

The portion of the system that manages system data must implement a protocol that determines what the system must do in the event that two users simultaneously issue conflicting commands. Consider informing the users in the event of a conflict as a portion of the protocol.

The portion of the system that manages user-specific data must be thread safe (e.g., re-entrant)

The portion of the system that interacts with the users must be thread safe.

EX.3.2 The system must have the ability to translate the names/ids of externally generated signals, e.g., from a sensor, into the concepts that are included in the Alarms&Events specification.

Rationale

The environment contains sensors that generate and actuators that respond to analog or digital signals in their own form.

The alarm and event portion of the system can only operate with logical concepts.

Implementing this responsibility

There should be a portion of the system (e.g., an intermediary) that sits between those portions of the system that interact directly with sensors and actuators (e.g., the device drivers) and the portion of the system that implements the alarm and event logic.

This intermediary should translate between the signals by the sensors and actuators and the logical concepts required by alarm and event rules.

EX.3.3 The system must have the ability to broadcast a generated event so that an external system can use it. E.g., an external long-time storage system.

Rationale

Some events require informing external people or systems. For example, an explosion may need to call emergency responders.

Implementing this responsibility

The portion of the system that executes the alarm and event rules must have the capability to broadcast to appropriate external systems.

EX.3.4 The system must have the ability to present alarm state transitions in the alarm displays within the time restrictions valid for this system.

Rationale

Users have limits as to how fast they can operate, i.e., perceive information, comprehend information, make decisions, and perform motor actions. This imposes a lower bound on how long information has to be displayed (visual or auditory).

Because the alarm system may supervise hazardous environments, safety regulations may require specific response times. This imposes an upper bound on the number of human actions that can be required to respond to alarms.

Any claims made for the operator action in response to alarms should be based upon sound human performance data and principles.

Implementing this responsibility

The portion of the system that displays information to the user should ensure that information is displayed long enough for a person to see or hear it. This requires that this portion of the system maintain timing information of how long information has been displayed and ensure that it is longer than minimal human perceptual limits.

The portion of the system that does the scheduling should schedule alarms as high-priority activities.

The portion of the system that interacts with the user during emergency situations must be designed to meet the response time requirements. This is the responsibility of the UI designers and does not impose additional architectural requirements.

EX.3.5 The system must have sufficient persistent storage for alarms, rules and data to be saved. It may be acceptable to limit the number of events to be saved.

Rationale

Regulations might require long-term storage of alarms, rules and data (e.g., in the food and drug business).

An organization may have publication, notification, history, fault diagnostic needs that require persistent data.

The system may be stopped and started again and the rules and data must not get lost when this happens.

There might be a lot of data.

Storage media has cost (hardware, time to read and write, network bandwidth).

Implementing this responsibility

The portion of the system that manages persistent data must ensure that the most recent and the most important data is not lost. This could be done by having large persistent data stores; it could be done by overwriting older data with newer data.

The portion of the system that manages persistent data must have a protocol to determine which data gets overwritten when persistent storage is almost full.

Logging

LG.1. Specify the Items to be Logged

LG.1.1 Ensure that sufficient parameters of the [User Profile, Configuration Description, Rules for Alarms, Events and Displays] are specified for logging so that subsequent analysis is meaningful (e.g., execution, parameter name, and time stamp). Consider prototyping and testing log information and analysis to ensure sufficiency. (Assumption: logging will be specified during development, not during or after deployment.)

Rationale

Software must know the information to be logged.

The values in the repository are going to be examined at a later time and these values must be able to be uniquely identified with sufficient information to be useful.

Implementing this responsibility

Implementation is beyond the scope of this USAP.

LG.2. Log Items During Execution

LG.2.1 Have a repository in which to store logged items relevant to the [User Profile, Configuration Description, Rules for Alarms, Events and Displays]. This repository could be bounded in size, e.g., circular buffer, or unbounded, e.g., disk file.

Rationale

The system must have a place to put logged information.

Implementing this responsibility

There must be a portion of the system that logs information.

The portion of the system that logs information must know the form of the repository and its location and may be responsible for creating the repository.

LG.2.2 Enter values relevant to the [User Profile, Configuration Description, Rules for Alarms, Events and Displays] into the repository as specified.

Rationale

Users need the logged values for debugging or audit trail purposes.

Stored information must persist long enough for analysis to be undertaken.

Implementing this responsibility

The portion of the system that logs information enters the particular values into the repository.

Attention should be paid to performance considerations since this code may be executed many times.

LG.3. Post-processing

LG.3.1 Retrieve items relevant to the [User Profile, Configuration Description, Rules for Alarms, Events and Displays] from the repository. This is typically done some time after the information has been logged, e.g., during the analysis of an anomaly.

Rationale

Users need information to analyze past events.

The information they need has been stored in the repository and must be retrieved.

Implementing this responsibility

There must be a portion of the system that knows how to get information out of the repository and does so.

LG.3.2 Support analysis of retrieved items relevant to the [User Profile, Configuration Description, Rules for Alarms, Events and Displays] by a log analyst (a special type of user)

Rationale

Users need support to analyze past events.

Implementing this responsibility

This may be quite complicated (e.g., graphical display, manipulation, mathematical modeling, debugging) and is beyond the scope of this USAP.

Appendix L: Introductory Material for Users in ABB User Tests.

Overview of USAPs Tool

A Usability-Supporting Architectural Pattern (USAP) provides a set of architecturally significant usability responsibilities that a software architect should consider when designing the architecture of a software system.

This USAPs tool prototype is designed to help you review usability concerns that may impact your software architecture. The prototype was designed under the assumption that you have selected to review architectural implications of User Profiles, Environment Configuration, and Alarms, Events, and Displays, for the software system on which you are already working. The USAPs tool will help you consider whether your architecture design supports usability issues for the following USAPs:

User Profile USAP

> Scenario: A user wishes to have the capabilities of the system personalized to reflect his or her preferences or role. The capabilities that can be personalized may include language, access to system functionality, display characteristics, account information or any preference that might vary among users or roles.

Environment Configuration USAP

> Scenario: An organization wants to supply the same software system to different hardware environments containing different collections of sensors and actuators. A configuration description of the sensors and actuators will allow the system to operate correctly in its environment.

Alarms, Events & Displays USAP

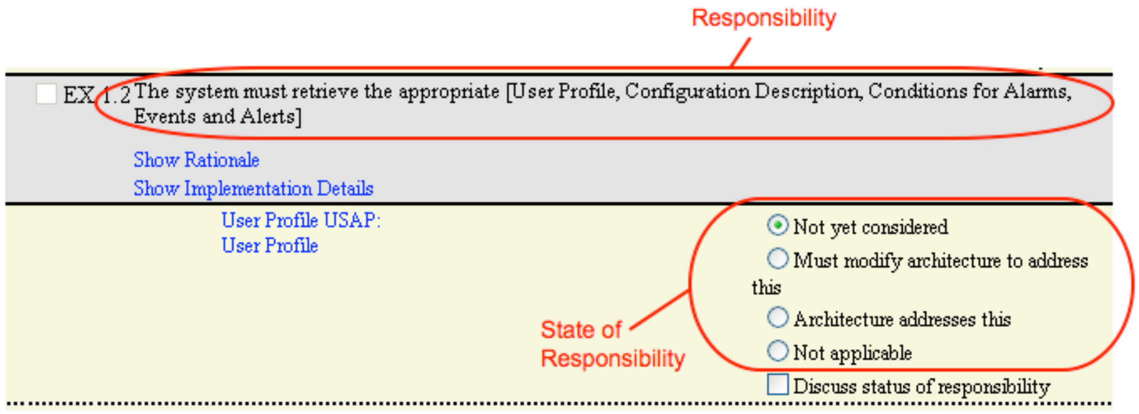
> Scenario: The user needs feedback from the system when an error occurred or a specific condition is met. The user can be the operator of the system or a superior system. The feedback can be needed for safety reasons, diagnostic, problem solving or information purposes.

How to use the USAPs tool

This tool contains a checklist of architecturally significant responsibilities you should consider when seeking to support User Profiles, Environment Configuration, and Alarms, Events and Displays in the architecture of a software system. Not all these responsibilities may be applicable to a particular system. However, each responsibility should be reviewed against the software architecture design of your system. As you review the checklist, you can generate a To-Do list for supporting usability issues in the architecture design with respect to User Profiles, Environment Configuration, and Alarms, Events and Displays.

A responsibility may be in one of four states, as shown below. By clicking the radio buttons, you set the status of the responsibility.

- "Not yet addressed" is the default state, and indicates that you have not yet considered the responsibility.
- "Not applicable" indicates that you have considered the responsibility and decided that it does not apply to the system you are designing.
- "Must modify architecture" indicates that you have considered the responsibility, found it applicable, but do not yet have functionality in the architecture design to support it.
- "Architecture addresses this" indicates that you have considered the responsibility, found it applicable, and also found that the architecture design already contains functionality to support the responsibility.



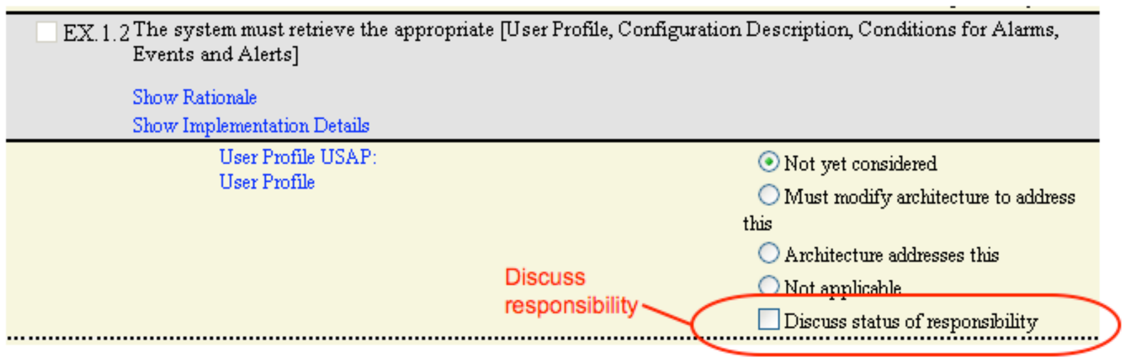
EX.1.2 The system must retrieve the appropriate [User Profile, Configuration Description, Conditions for Alarms, Events and Alerts]

Show Rationale
Show Implementation Details

User Profile USAP:
User Profile

[✓ Not yet considered]
Must modify architecture to address this
Architecture addresses this
Not applicable
Discuss status of responsibility

Additionally, if you think that discussion of a responsibility (with team members or others) is desirable, you may check the "Discuss status of this responsibility" checkbox.



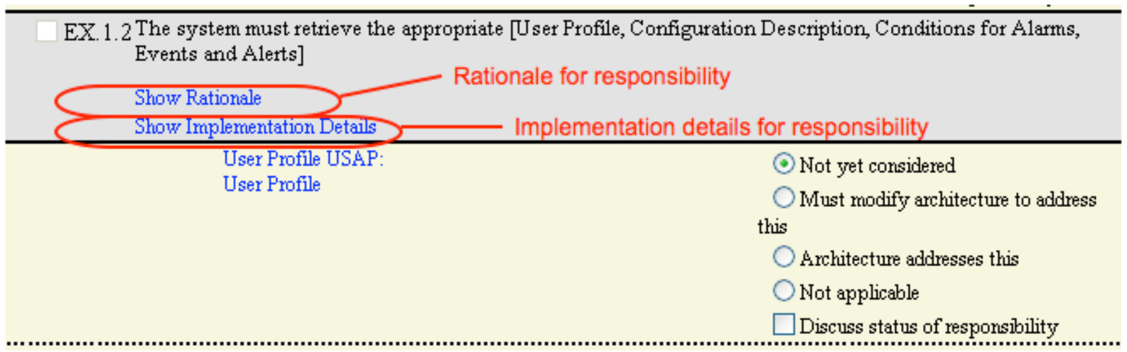
EX.1.2 The system must retrieve the appropriate [User Profile, Configuration Description, Conditions for Alarms, Events and Alerts]

Show Rationale
Show Implementation Details

User Profile USAP:
User Profile

[✓ Not yet considered]
Must modify architecture to address this
Architecture addresses this
Not applicable
Discuss status of responsibility

Each responsibility also contains links you can click for further information. Clicking "Show Rationale" or "Show Implementation Details" will show you a more detailed background of the responsibility. Clicking these links will not cause you to navigate away from the responsibility.


Rationale for responsibility
Implementation details for responsibility

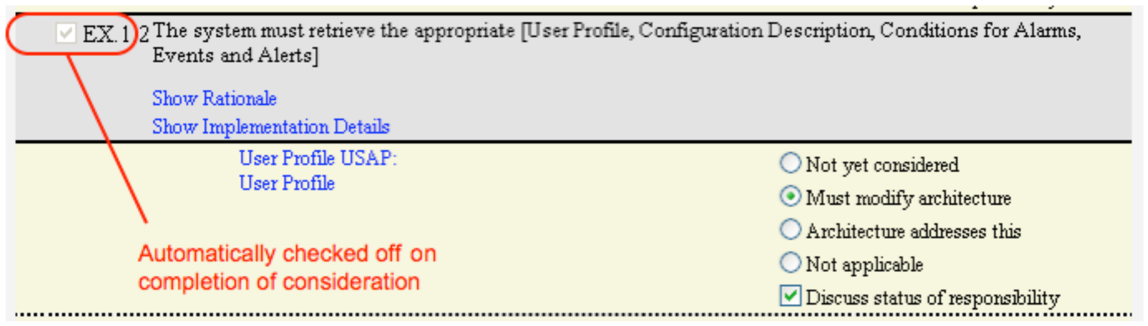
EX. 1.2 The system must retrieve the appropriate [User Profile, Configuration Description, Conditions for Alarms, Events and Alerts]

Show Rationale
Show Implementation Details

User Profile USAP:
User Profile

[✓ Not yet considered]
Must modify architecture to address this
Architecture addresses this
Not applicable
☐ Discuss status of responsibility

When you have considered a responsibility, check the appropriate radio button. Do not leave it in the default state (not yet addressed). When you have checked off the individual items under a responsibility, the checkbox to the left of the responsibility will be checked automatically to indicate that you have completed that section of the checklist.



✓ EX. 1.2 The system must retrieve the appropriate [User Profile, Configuration Description, Conditions for Alarms, Events and Alerts]

Show Rationale
Show Implementation Details

User Profile USAP:
User Profile

Not yet considered
[✓ Must modify architecture]
Architecture addresses this
Not applicable
☑ Discuss status of responsibility

Responsibilities are arranged in sets of related issues (i.e., authoring, execution with authored parameters, logging). When you have completed a set of responsibilities, the checkboxes for that set of responsibilities will also be checked automatically. This will help you know when you have reviewed all the responsibilities.

When you have finished reviewing all the responsibilities in the checklist, click the "Generate ToDo List" link in the left menu to see a list of the current status of the responsibilities. This list will show all responsibilities that have yet to be addressed, required architectural changes, and/or need to be discussed further. The ToDo list has a place to record comments for any responsibility you wish to annotate. Annotate the list as desired, and then print it. For the prototype, print to a PDF file using CutePDF as the printer name.

Appendix M: A-PLUS Architect Design Used in Online Survey.

An Introduction to A-PLUS Architect

As a software architect, you need to be able to evaluate whether your architecture designs will support usability concerns that arise from your business drivers. A-PLUS Architect is a tool to help do just that: review usability concerns that may impact your software architecture.

There are usability concerns commonly found in modern computer-based systems (e.g., the ability to cancel and undo, customizable user profiles, configurable environment, ubiquitous search, etc.) that can be specified during requirement formulation. Many of these will have impact on the software architecture design. The relationship between a usability concern and its architectural impact has been captured in research in Usability-Supporting Architectural Patterns (USAPs). A-PLUS Architect embodies the knowledge in USAPs in a tool to help ensure that the architecture design will support the necessary usability capabilities.

For example, suppose your system is required to cancel long-running processes. One responsibility of this usability concern is to release resources that the process is using as it executes. If that responsibility were not considered at design time, retrofitting an architecture to keep track of resources for every running process and release them on demand would be much more difficult than designing this in at the beginning.

For the purpose of this survey, suppose that you are evaluating the preliminary software architecture design for a system to determine whether it supports customization of the user interface (i.e., user profiles). A-PLUS Architect will help you review all the usability responsibilities necessary to support that capability. The example you see here only shows architecture design support for customizable user profiles, but the tool can support many different usability scenarios. Many responsibilities may be shared between usability concerns in more than one scenario. A-PLUS Architect can make evaluation easier by helping you review the responsibilities for multiple scenarios at the same time.

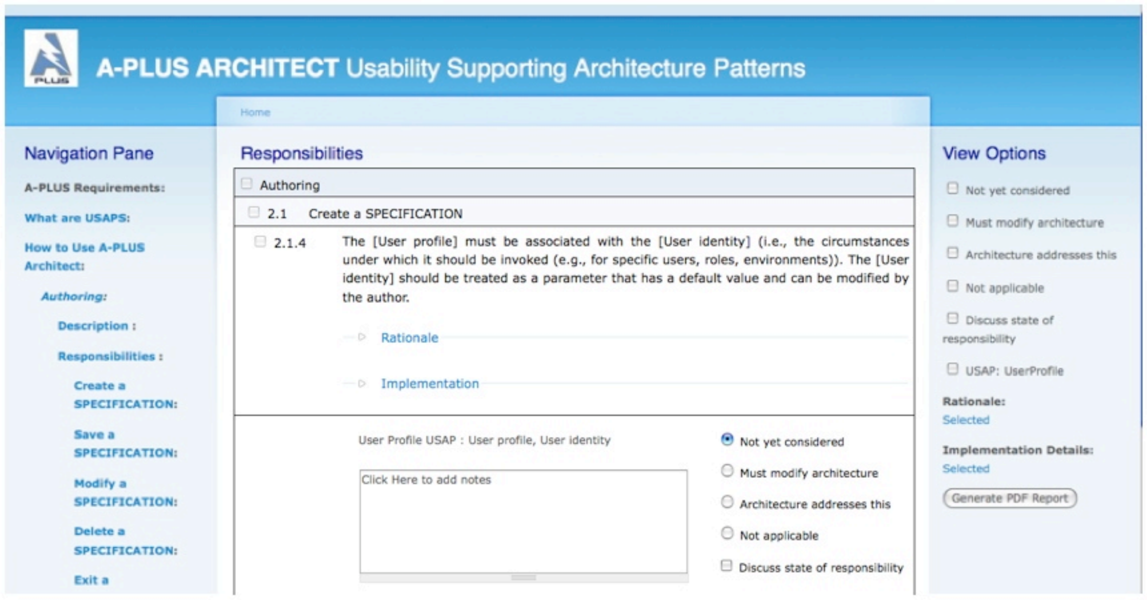
The software responsibilities in A-PLUS Architect are divided into convenient sets, each set coming from one or more USAPs. Each of these USAPs includes the software responsibilities that may arise from a single usability scenario. In the example shown below, the responsibilities that a system may have to include to provide for multiple user profiles come from this scenario:

“A user wishes to have the capabilities of the system personalized to reflect his or her preferences or role. The capabilities that can be personalized may include language, access to system functionality, display characteristics, account information or any preference that might vary among users or roles.”

You can use A-PLUS Architect to review these sets of usability responsibilities systematically. This will help you identify which responsibilities are relevant to your particular software system. If a responsibility is relevant to your system, you can decide whether design modifications will be needed, whether further discussion is warranted, and make notes about your design decisions. This systematic review will let you thoroughly evaluate the ability of your design to support usability concerns related to usability concerns before detailed design or implementation even begin. By addressing these issues up front you can avoid painful late-stage architectural changes to your system.

How to Use A-PLUS Architect

A-PLUS Architect is a browser-based tool. It contains a checklist of architecturally significant responsibilities you should consider when seeking to support different User Profiles in the architecture of a software system. Not all these responsibilities may be applicable to a particular system. However, each responsibility should be reviewed against the software architecture design of your system. As you review the checklist, you can generate a To-Do list for supporting usability issues in the architecture design with respect to User Profiles.


A-PLUS ARCHITECT Usability Supporting Architecture Patterns

Home

Navigation Pane

A-PLUS Requirements:

What are USAPS:

How to Use A-PLUS Architect:

Authoring:

Description :

Responsibilities :

Create a SPECIFICATION:

Save a SPECIFICATION:

Modify a SPECIFICATION:

Delete a SPECIFICATION:

Exit a

Responsibilities

☐ Authoring

☐ 2.1 Create a SPECIFICATION

☐ 2.1.4 The [User profile] must be associated with the [User identity] (i.e., the circumstances under which it should be invoked (e.g., for specific users, roles, environments)). The [User identity] should be treated as a parameter that has a default value and can be modified by the author.

▷ Rationale

▷ Implementation

User Profile USAP : User profile, User identity

Click Here to add notes

[✓ Not yet considered] Must modify architecture / Architecture addresses this / Not applicable

☐ Discuss state of responsibility

View Options

☐ Not yet considered

☐ Must modify architecture

☐ Architecture addresses this

☐ Not applicable

☐ Discuss state of responsibility

☐ USAP: UserProfile

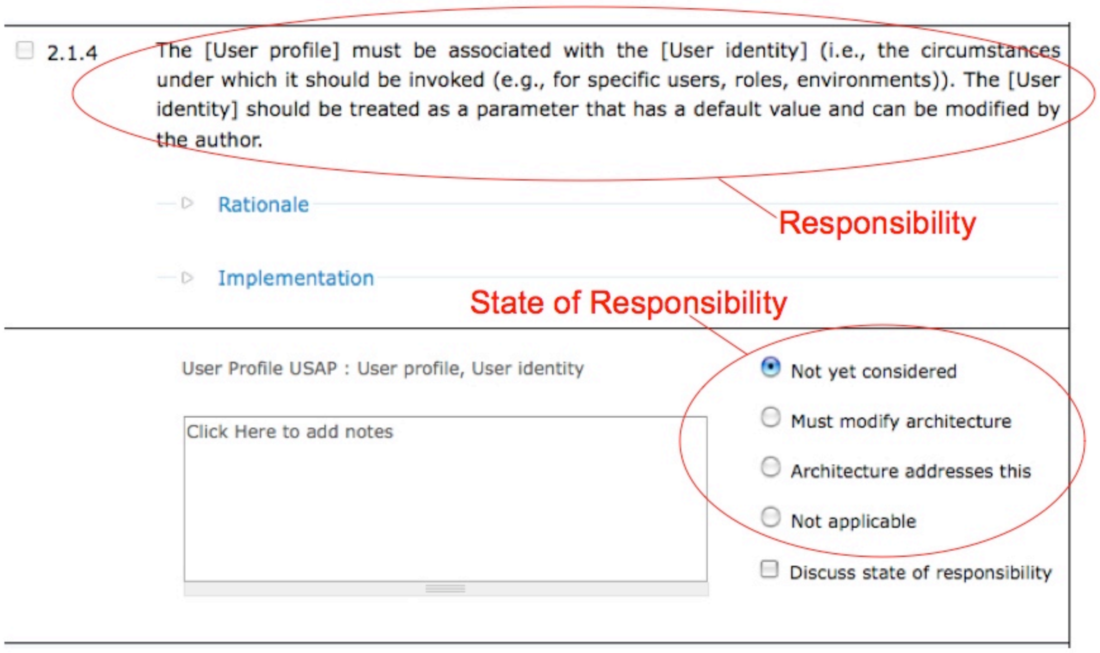
Rationale: Selected

Implementation Details: Selected

Generate PDF Report

A responsibility may be in one of four states, as shown below. By clicking the radio buttons, you set the status of the responsibility.

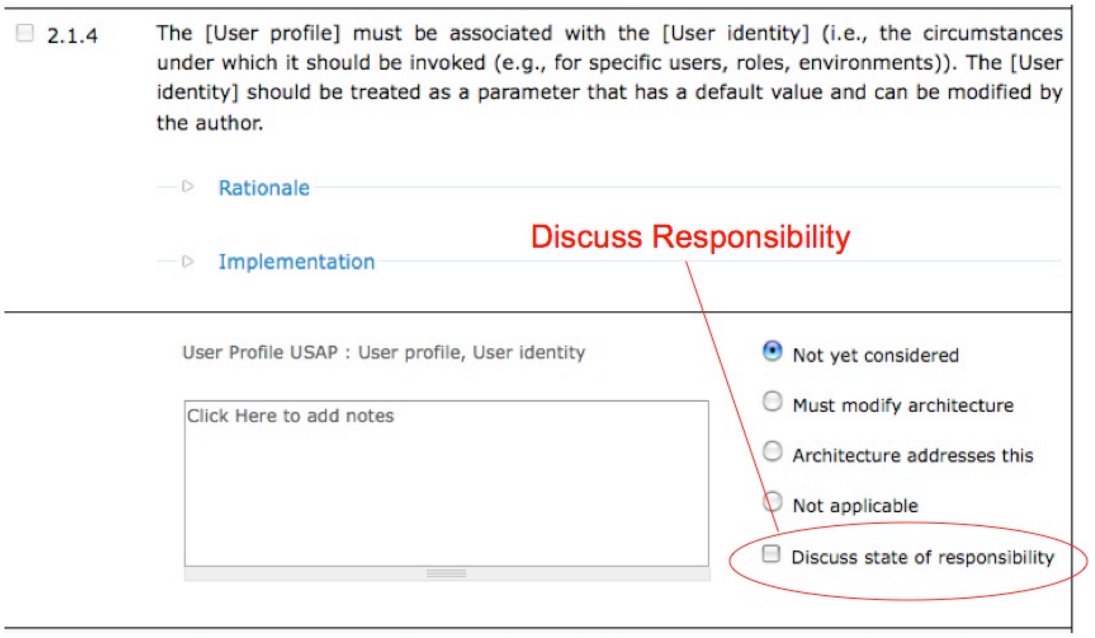
– "Not yet considered" is the default state, and indicates that you have not yet noted whether you have considered the responsibility.
– "Must modify architecture" indicates that you have considered the responsibility, found it applicable, but do not yet have functionality in the architecture design to support it.
– "Architecture addresses this" indicates that you have considered the responsibility, found it applicable, and also found that the architecture design already contains functionality to support the responsibility.
– "Not applicable" indicates that you have considered the responsibility and decided that it does not apply to the system you are designing.



2.1.4 The [User profile] must be associated with the [User identity] (i.e., the circumstances under which it should be invoked (e.g., for specific users, roles, environments)). The [User identity] should be treated as a parameter that has a default value and can be modified by the author.

User Profile USAP : User profile, User identity

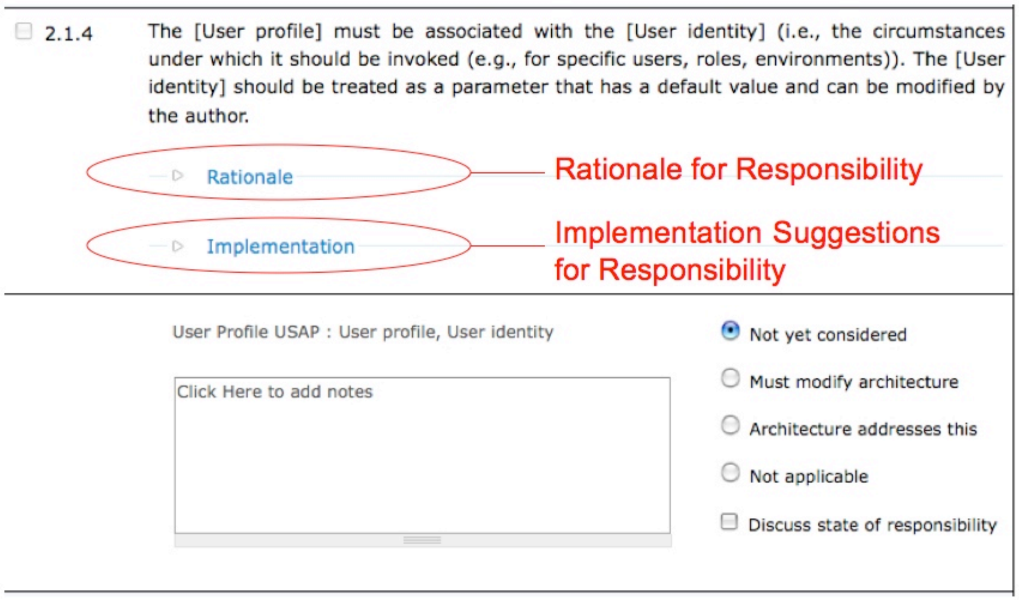
Additionally, if you think that discussion of a responsibility (with team members or others) is desirable, you may check the “Discuss status of this responsibility” checkbox.



2.1.4 The [User profile] must be associated with the [User identity] (i.e., the circumstances under which it should be invoked (e.g., for specific users, roles, environments)). The [User identity] should be treated as a parameter that has a default value and can be modified by the author.

User Profile USAP : User profile, User identity

Each responsibility also contains links you can click for further information. Expanding the "Rationale" or “Implementation" links will show you a more detailed background of the responsibility. Clicking these links will not cause you to navigate away from the responsibility.



☐ 2.1.4 The [User profile] must be associated with the [User identity] (i.e., the circumstances under which it should be invoked (e.g., for specific users, roles, environments)). The [User identity] should be treated as a parameter that has a default value and can be modified by the author.

▷ Rationale

▷ Implementation

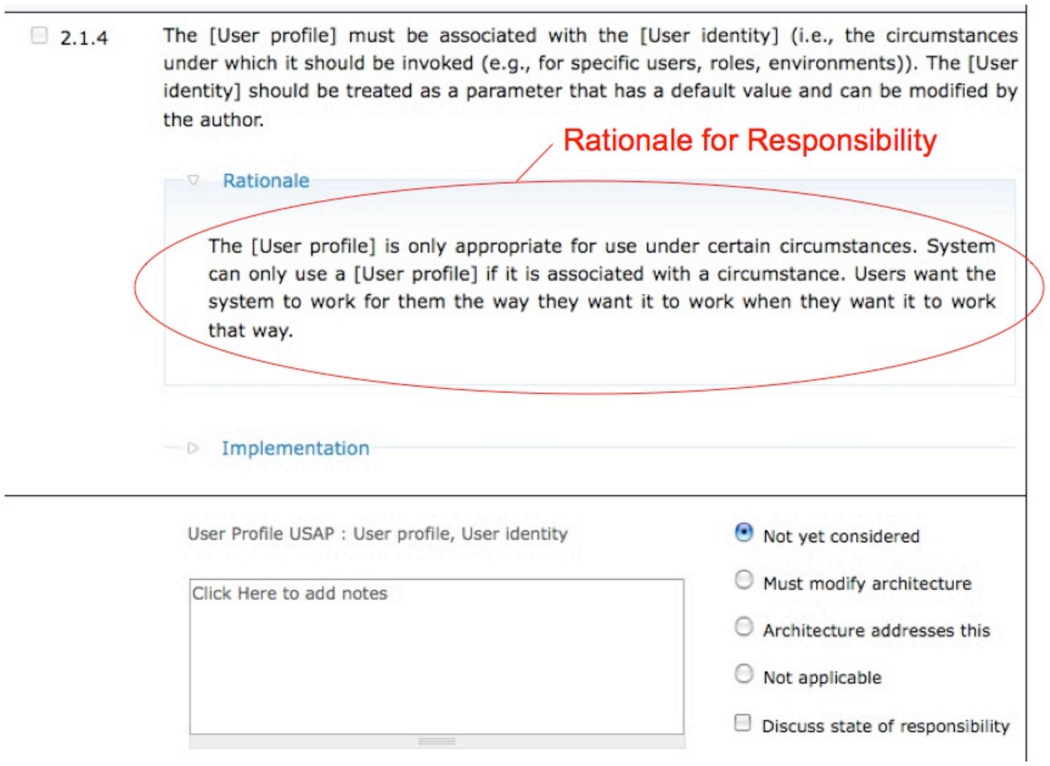
User Profile USAP : User profile, User identity

Click Here to add notes

(● Not yet considered) ○ Must modify architecture ○ Architecture addresses this ○ Not applicable ☐ Discuss state of responsibility

Rationale for Responsibility
Implementation Suggestions for Responsibility

Expanding the Rationale link for a responsibility shows the rationale behind including the responsibility when considering User Profiles.



☐ 2.1.4 The [User profile] must be associated with the [User identity] (i.e., the circumstances under which it should be invoked (e.g., for specific users, roles, environments)). The [User identity] should be treated as a parameter that has a default value and can be modified by the author.

▽ Rationale

The [User profile] is only appropriate for use under certain circumstances. System can only use a [User profile] if it is associated with a circumstance. Users want the system to work for them the way they want it to work when they want it to work that way.

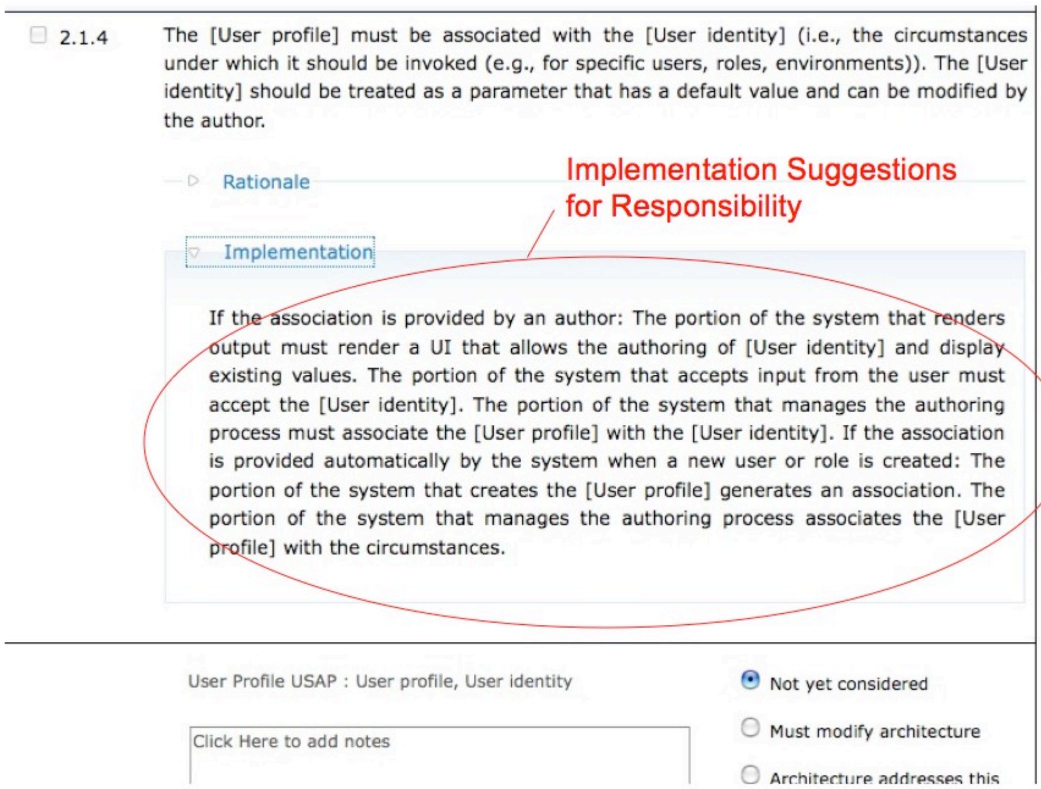
▷ Implementation

User Profile USAP : User profile, User identity

Click Here to add notes

(● Not yet considered) ○ Must modify architecture ○ Architecture addresses this ○ Not applicable ☐ Discuss state of responsibility

Expanding the Implementation link for a responsibility shows general implementation suggestions for including that responsibility in an architecture design.



☐ 2.1.4 The [User profile] must be associated with the [User identity] (i.e., the circumstances under which it should be invoked (e.g., for specific users, roles, environments)). The [User identity] should be treated as a parameter that has a default value and can be modified by the author.

▷ Rationale

▽ Implementation

If the association is provided by an author: The portion of the system that renders output must render a UI that allows the authoring of [User identity] and display existing values. The portion of the system that accepts input from the user must accept the [User identity]. The portion of the system that manages the authoring process must associate the [User profile] with the [User identity]. If the association is provided automatically by the system when a new user or role is created: The portion of the system that creates the [User profile] generates an association. The portion of the system that manages the authoring process associates the [User profile] with the circumstances.

User Profile USAP : User profile, User identity

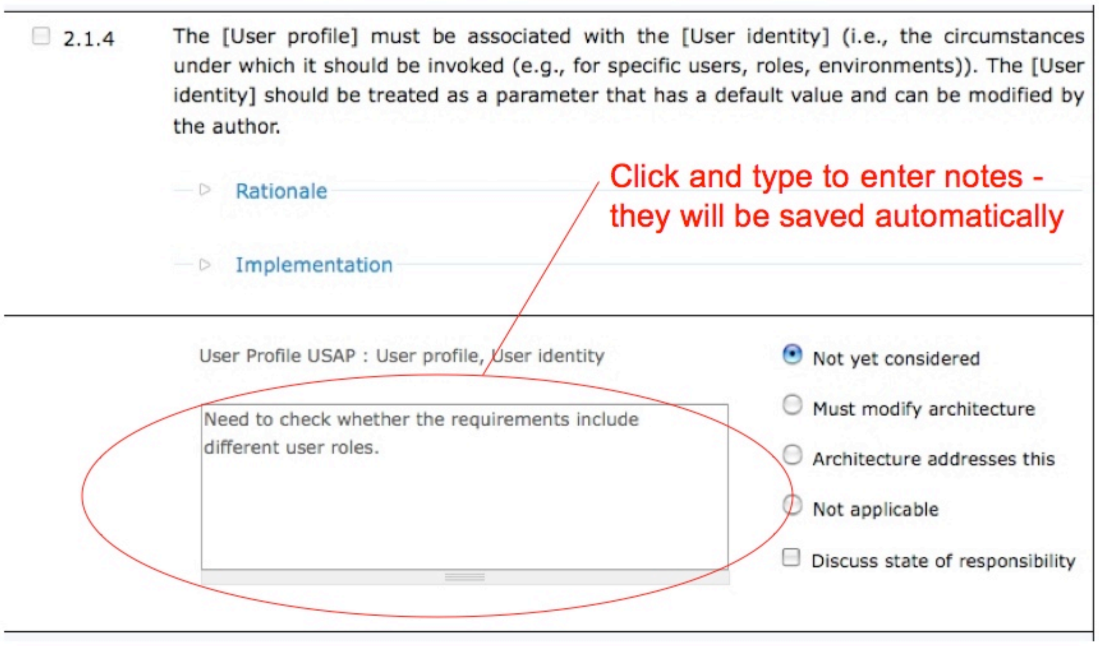
Click Here to add notes

● Not yet considered

○ Must modify architecture

○ Architecture addresses this

A responsibility may be in one of four states, as shown below. By clicking the radio buttons, you set the status of the responsibility.



☐ 2.1.4 The [User profile] must be associated with the [User identity] (i.e., the circumstances under which it should be invoked (e.g., for specific users, roles, environments)). The [User identity] should be treated as a parameter that has a default value and can be modified by the author.

▷ Rationale

▷ Implementation

User Profile USAP : User profile, User identity

Need to check whether the requirements include different user roles.

● Not yet considered

○ Must modify architecture

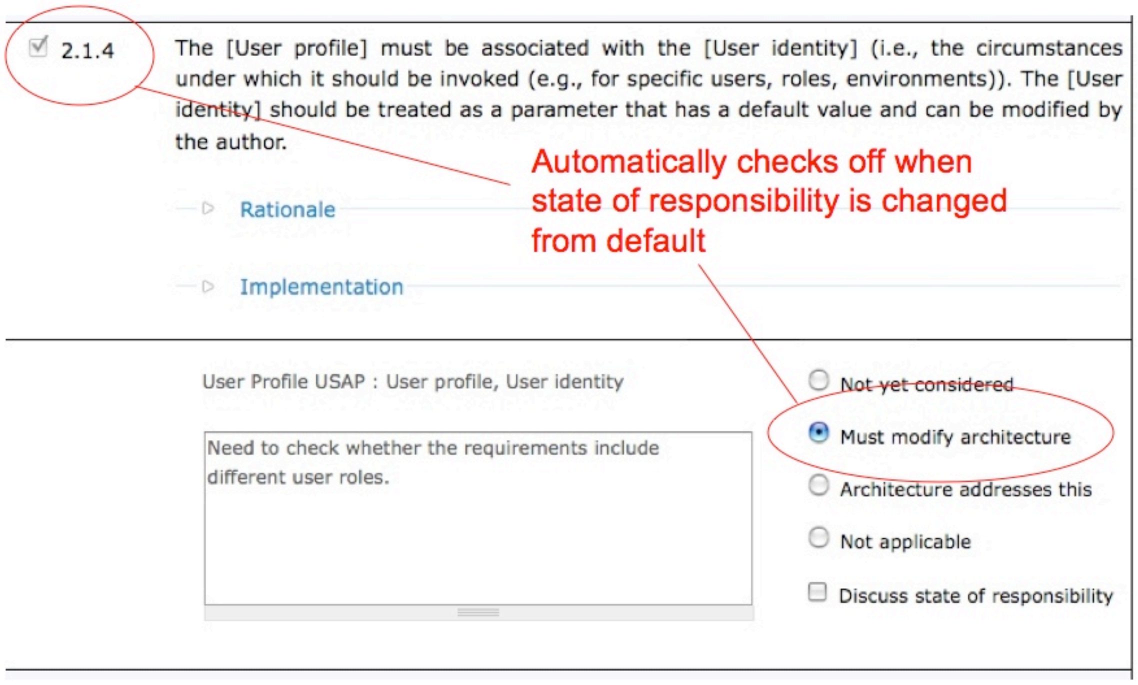
○ Architecture addresses this

○ Not applicable

☐ Discuss state of responsibility

Click and type to enter notes - they will be saved automatically

When you have considered a responsibility, check the appropriate radio button. Do not leave it in the default state ("Not yet considered"). When you have checked off the individual items under a responsibility, the checkbox to the left of the responsibility will be checked automatically to indicate that you have completed that section of the checklist.


☑ 2.1.4 The [User profile] must be associated with the [User identity] (i.e., the circumstances under which it should be invoked (e.g., for specific users, roles, environments)). The [User identity] should be treated as a parameter that has a default value and can be modified by the author.

▷ Rationale

▷ Implementation

User Profile USAP : User profile, User identity

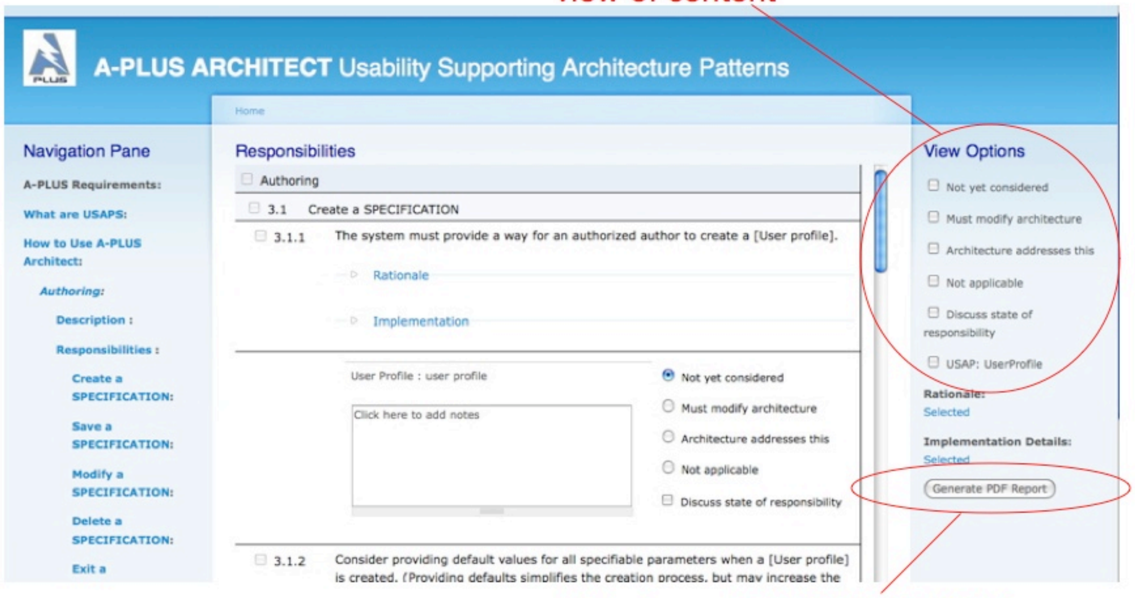
Need to check whether the requirements include different user roles.

Automatically checks off when state of responsibility is changed from default
Not yet considered
Must modify architecture
Architecture addresses this
Not applicable
Discuss state of responsibility

Responsibilities are arranged in sets of related issues (i.e., authoring, execution with authored parameters, logging). When you have completed a set of responsibilities, the checkboxes for that set of responsibilities will also be checked automatically. This will help you know when you have reviewed all the responsibilities.

When you have finished reviewing all the responsibilities in the checklist, you will have a ToDo list of the current status of all the responsibilities related to User Profile. This list can be filtered as you wish to show or hide responsibilities in any of the four states, those that need to be discussed further, the rationale and implementation details. The list will include the notes you have added to any responsibility. Filter the list as desired, and then print it using the “Generate PDF Report”.

Use checkboxes to control view of content



A-PLUS ARCHITECT Usability Supporting Architecture Patterns

Home

Navigation Pane

A-PLUS Requirements:

What are USAPS:

How to Use A-PLUS Architect:

Authoring:

Description :

Responsibilities :

Create a SPECIFICATION:

Save a SPECIFICATION:

Modify a SPECIFICATION:

Delete a SPECIFICATION:

Exit a

Responsibilities

☐ Authoring

☐ 3.1 Create a SPECIFICATION

☐ 3.1.1 The system must provide a way for an authorized author to create a [User profile].

▷ Rationale

▷ Implementation

User Profile : user profile

Click here to add notes

[✓ Not yet considered]

Must modify architecture

Architecture addresses this

Not applicable

☐ Discuss state of responsibility

☐ 3.1.2 Consider providing default values for all specifiable parameters when a [User profile] is created. (Providing defaults simplifies the creation process, but may increase the

View Options

☐ Not yet considered

☐ Must modify architecture

☐ Architecture addresses this

☐ Not applicable

☐ Discuss state of responsibility

☐ USAP: UserProfile

Rationale:
Selected

Implementation Details:
Selected

Click to create PDF ToDo list from responsibilities and comments

Appendix N: Survey Items Used in ABB User Tests.

TAM2 Measurement Items (adapted from Venkatesh & Davis, 2000)

Intention to Use
1. Assuming I have access to the tool, I intend to use it.
2. Given that I have access to the tool, I predict that I would use it.

Perceived Usefulness
1. Using the tool improves my performance in my job.
2. Using the tool in my job increases my productivity.
3. Using the tool enhances my effectiveness in my job.
4. I find the tool to be useful in my job.

Perceived Ease of Use
1. My interaction with the tool is clear and understandable.
2. Interacting with the tool does not require a lot of my mental effort.
3. I find the tool to be easy to use.
4. I find it easy to get the tool to do what I want it to do.

Subjective Norm
1. People who influence my behavior think that I should use the tool.
2. People who are important to me think that I should use the tool.

Voluntariness
1. My use of the tool is voluntary.
2. My supervisor does not require me to use the tool.
3. Although it might be helpful, using the tool is certainly not compulsory in my job.

Image
1. People in my organization who use the tool have more prestige than those who do not.
2. People in my organization who use the tool have a high profile.
3. Having the tool is a status symbol in my organization.

Job Relevance
1. In my job, usage of the tool is important.
2. In my job, usage of the tool is relevant.

Output Quality
1. The quality of the output I get from the tool is high.
2. I have no problem with the quality of the tool's output.

Result Demonstrability
1. I have no difficulty telling others about the results of using the tool.
2. I believe I could communicate to others the consequences of using the tool.

3. The results of using the tool are apparent to me.
4. I would have difficulty explaining why using the tool may or may not be beneficial.