To appear in the Proceedings of EICS 2009, July 14-17, 2009, Pittsburgh, PA, USA

A Responsibility-Based Pattern Language for Usability-Supporting Architectural Patterns

Bonnie E. John
HCI Institute
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA, 15217
+1-412-268-7182
bej@cs.cmu.edu

Len Bass
Software Engineering Institute
Carnegie Mellon University
4500 Fifth Ave.
Pittsburgh, PA, 15217
+1-412-268-6763
ljb@sei.cmu.edu

Elspeth Golden
HCI Institute
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA, 15217
1+412-268-2000
egolden@cmu.edu

Pia Stoll
ABB Corp Research
Forskargränd 6
SE 72178
Västerås, Sweden
+46 21 32 30 00
pia.stoll@se.abb.com

ABSTRACT
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. In laboratory studies, the Cancellation USAP was shown to significantly improve the quality of architecture designs for supporting the ability to cancel a long-running command, sparking interest from a large industrial organization to develop new USAPs and apply them to their product line architecture design. The challenges of delivering the architectural information contained in USAPs to practicing software architects led to the development of a pattern language for USAPs based on software responsibilities and a web-based tool for evaluating an architecture with respect to those patterns.

Categories and Subject Descriptors
H.5.2 User Interfaces; D.2.11 Software Architectures.

General Terms
Design, Human Factors.

Keywords
Usability, software architecture.

1. INTRODUCTION
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 [14]. A score of patterns were originally proposed [5], loosely connected by software tactics that they might share (e.g., encapsulation of function, data indirection, preemptive scheduling), but this relationship was 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 [8].

As originally conceived to emulate patterns in the Alexander style [2], a USAP had six types of information. We illustrate the types with information from the Cancellation USAP [15].

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 1 and 2).

Making all six parts complete and internally consistent is a time-consuming process and for several years the original authors succeeded only in creating one full exemplar, 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.

The Cancellation USAP was shown to significantly improve the quality of architecture designs for supporting the ability to cancel

EICS'09, July 14–17, 2009, Pittsburgh, PA, USA.

a long-running command in laboratory studies [12, 13]. This success was encouraging, but did not address issues with designing an architecture that had to consider more than one USAP at a time. Multiple USAPs had been used in a real development setting, but that intervention required heavy involvement from the researchers who developed the USAPs because they were not sufficiently fleshed out for the architect to use them on his own [1]. Thus, prior uses of USAPs suffered from two defects. First, the industrial uses all involved the researchers who developed USAPs and this clearly does not scale up. Second, the laboratory experiments were paper based and the participants omitted important responsibilities of the USAPs, leaving additional room for quality improvement.

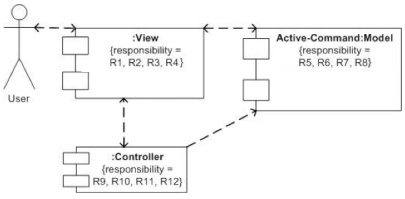

Figure 1 UML-style component diagram of the MVC reference architecture before considering the ability to cancel a long running command. Original responsibilities of MVC are numbered R1, R2, etc. and refer to prose descriptions of the responsibilities accompanying the diagram.

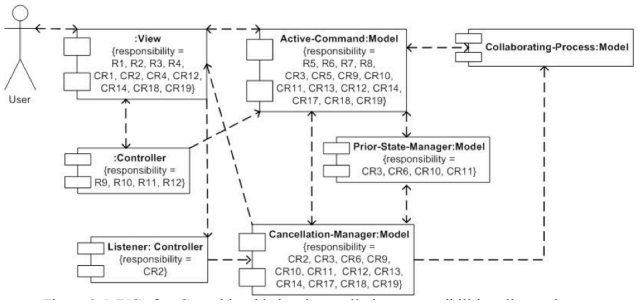

Figure 2 UML-style component diagram of MVC revised to include the responsibilities necessary to provide the ability to cancel a long running command. New cancellation responsibilities are numbered CR1, CR2, etc., are assigned to both old and new components, and refer to prose descriptions of the responsibilities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

2. A COLLABORATION BETWEEN INDUSTRY AND ACADEMIA

A large global company, ABB, was beginning the development of a new product line. Prior to the collaboration reported in this paper, the project team in the business unit developing the new product line of systems, together with an ABB research team, had done a use case analysis, performed a Quality Attribute Workshop to collect non-functional requirements from prioritized scenarios [4], used the Influencing Factors method [18] and conducted the first step of the Product Line Architecture development approach [7] with the identification of commonalities and variation points. 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 starting 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. Usability had been prioritized as one of three most important software qualities for the new system during the Quality Attribute Workshop.

The best method to support usability concerns through software architecture has been the subject of some investigation over the past years. In addition to the work on USAPs [5, 12, 13, 14, 15], Folmer and his colleagues [9, 10] and Juristo and her colleagues [16] have investigated the relationship between software architecture and usability. None of this work has gained widespread industry acceptance probably because all of the results reported require the direct involvement of the researchers. Thus, our goal in this project 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 that could scale to support global development efforts.

After some investigation into the interplay of usability and software architecture the ABB research team, supporting the ABB project team, decided to try using usability-supporting architectural patterns (USAPs) in a collaboration with the authors at Carnegie Mellon University (CMU). Their decision was based on the fact that USAPs use generic usability scenarios and from these construct generic software architecture responsibilities. By working this way, ABB expected to address some of the major usability concerns early in the software design phase without having an actual user interface design in place.

The collaboration started with several teleconferences and a two-day workshop where the ABB research team together with the CMU research team presented the USAP approach and the ABB project team presented their new product line system to be developed. The researchers presented 19 general usability scenarios possibly relevant to this domain. These were:

- Progress feedback
- Warning/status/alert feedback
- Undo
- Canceling commands
- User profile
- Help
- Command aggregation
- Action for multiple objects
- Workflow model
- Different views of data
- Keyboard shortcuts
- Reuse of information
- Maintaining compatibility with other systems
- Navigating within a single view
- Recovering from failure
- Identity management
- Comprehensive search
- Supporting internationalization
- Working at the user's pace

The ABB project team was asked to prioritize the general usability scenarios and they decided to focus on two – User Profile and Warning/status/alert feedback (which ABB called "Alarms and Events"). We quickly realized that User Profile should be separated into two USAPs for ABB: the normal HCI sense of User Profile, where different users have different permissions and preferences, and Environment Configuration, where the software has to be configured for different physical

operating environments. The research teams decided to elaborate these three USAPs separately; two were developed by the CMU research team and one by the ABB research team. There were two reasons for this division of labor: first, we expected faster results by working in parallel and second, we expected to transfer knowledge of the USAP development process to the ABCB research team. The ABB research team worked independently but was coached by the CMU team to understand the type and level of information contained in a USAP. Both teams acted as reviewers for each other's drafts and revisions of the USAPs.

Three major results came from this collaboration, detailed in the next three sections of this paper. Two research results were (1) a reformatting of the USAP content and (2) the development of a pattern language. An applied result occurred in a very positive experience by ABB software architects who used the research results to evaluate a preliminary architecture design.

3. REFORMATTING THE USAP

Before actually applying these new USAPs to the product line system project's design we asked software architects from a different business unit for feedback on the draft USAP produced by the ABB researchers. These architects appreciated the concept but had three negative reactions to aspects of our implementation of the concept.

1. The architects did not like the UML-style sample solution. They felt pressured to use the overarching pattern on which the sample solution was based, e.g. MVC. If they were already using 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.
2. There were too many responsibilities in the USAPs. The first draft of the Alarm and Event USAP had, by itself, 79 responsibilities. The software architects thought that examining all responsibilities from three USAPs would take far too long during architecture design.
3. The architects questioned whether it would be possible to integrate three (or more) different USAPs within a single architecture design. They imagined having three different UML sample solutions lying on their desks and by they could not figure out how these would be integrated in practice.

In retrospect, all of these criticisms were 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" (p. 276, [2]) 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.*
>
> *(pp xiii-xiv, [3]).*

Although a particular UML diagram *can* be drawn, doing so necessitates depicting and arranging components other than those that are essential to the pattern and thereby impose themselves on the architecture designer. Our motivation for including a sample UML solution in the original formulation of USAPs was to conform to standard pattern templates. Most pattern templates call for some sort of example or multiple examples, often expressed in diagrams. But in the presence of negative user feedback, we went back to the laboratory results and found that there seemed to be no difference in the quality of the finished architecture design and the amount of time the participants spent studying the UML-style diagrams. 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, diagrams like the ones in Figure 1 and Figure 2 were not developed for the three USAPs for ABB. Instead, each responsibility was accompanied by some implementation suggestions expressed in architecture-neutral prose. For example, Figure 2 assigns cancellation responsibility 1 (CR1) to MVC's View. Responsibility CR1 is "A button, menu item, keyboard shortcut and/or other means must be provided, by which the user may cancel the active command." In the new format, without UML-style diagrams, CR1 would 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 is architecture-neutral since other architectures may render information to the user in a component not called View, nor designed to be functionally identical to MVC's View. We hypothesized that this architecture-neutral expression would be more palatable to industry designers working with their own architectural styles. As will be seen in a later section reporting on the use of our materials in practice, this hypothesis seems to have been correct.

Removing the UML-style diagrams focus the patterns on the responsibilities, rather than the structural relationship that supports the implementation of those responsibilities. Although we had previously considered USAPs to be software architecture patterns in the flavor of [6], as opposed to usability patterns such as in [19], we now believe they may be something new. 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 to these requirements can 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 patterns typically found in the software engineering and HCI literature. Alternatively, perhaps it is only the absence of specific examples that makes USAPs different from these other patterns and changes the way practitioners use them. SE and HCI designers have been known to use the sample solutions directly and produce designs very similar in structure to those exemplars. Our architecture designers were working under constraints that prevented easy analogy from the sample solution and the sample solution itself is complex enough to take substantial time to understand. Thus, our architects preferred to be presented with the information at the higher level of abstraction and then instantiate it only in their own context. Either way, 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 designers; understanding its precise place in the research/methods landscape is left for future work.

4. THE EMERGENCE OF A PATTERN LANGUAGE FOR USAPS

> *...the possibility of language is latent in the fact that patterns are not isolated. But it comes out, in its full force, when we experience the desire to make something... [This desire] puts structure on the patterns, and makes languages out of them.*
>
> *(p. 309, [2])*

In the collaboration between ABB and CMU, 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 [1]). 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"(p. 320, [2])*, 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,

> *It is essential to distinguish those patterns which are the principal components of any given pattern, from those which lie still further down.*
>
> *(p. 321, [2])*

When we compared the three USAPs chosen by the requirements team, User Profile and Environment Configuration (fleshed out by CMU) and Alarms and Events (fleshed out by ABB), we observed that both the 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.*
>
> *(xliii-xliv, [3])*

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.

5. A PATTERN LANGUAGE

As mentioned previously, each of our original USAPs is a pattern of responsibilities and each responsibility is a pattern of implementation suggestions. **Figure 3** shows these 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**. 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." (p. 312, [2])* 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 in the Introduction, 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.

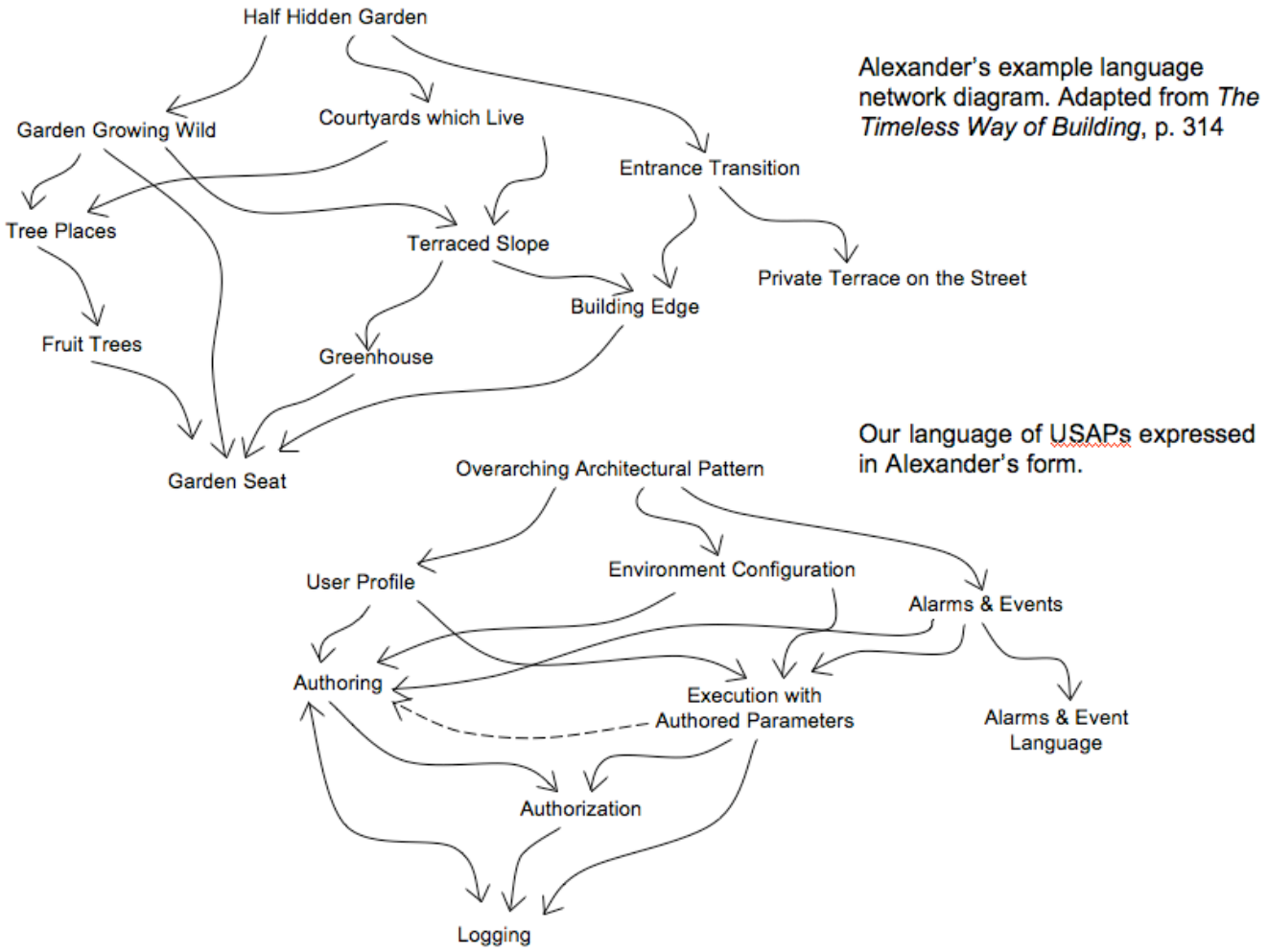

Figure 3 Our language of three USAPs is comparable to a portion of Alexander's network diagram of his language.

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 described in the Introduction). The conditions under which they are applicable are defined (Part 2) and their forces enumerated (Part 4). Their benefit to the end-users can be analyzed and estimated (Part 3). They can be prioritized by a project team. Now, however, given the pattern language, instead of having a complete list of responsibilities of their own (Part 5), they will be completed by the Foundational USAPs they use. However, they may specialize the responsibilities of the Foundational USAPs and they may have responsibilities of their own.

For example, all of the End-User USAPs that we present, 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

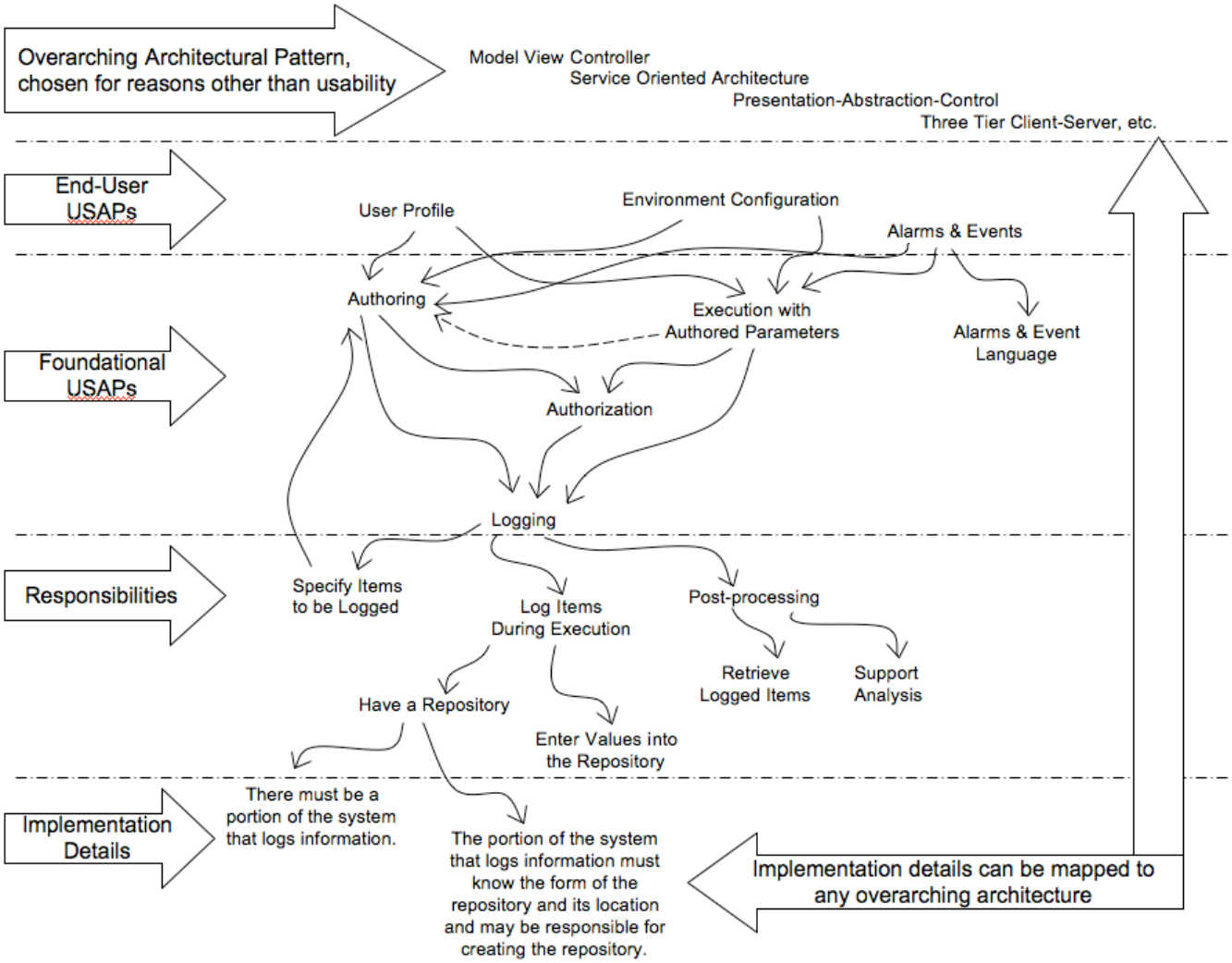

Figure 4 The multiple scales of our pattern language, a partial depiction.

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.

More rigorously, there are two types of relationships between the patterns in **Figure 4**: `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 `use` relationship is typically between End-User USAPs and Foundational USAPs. However, some Foundational USAPs `use` other Foundational USAPs under certain circumstances. The `depends-on` relationship (dashed line in **Figure 4**) implies a temporal relationship. The only `depends-on` relationship in our language is between Authoring Foundational USAP and the Execution with Authored Parameters Foundational USAP, 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 4 principal components: Identification (with 5 lower-level responsibilities), Authentication (with 3), Permissions (with 2), and Log-Off. Authoring has 5 principal components: Create (with 4 lower-level responsibilities), Save, Modify (with 3), Delete (with 3) and Exit the authoring system. Execution with Authored Parameters has 2 principal components: Access Specification (with 7 lower-level responsibilities) and Use specified parameters (with 2). Logging has 3 principal components: Specify the items to be logged, Log during execution (with 2 lower-level responsibilities) and Post-processing (with 2). 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 21 responsibilities of the Cancel USAP that seemed to be too much for the experiments' participants to absorb in one sitting [13]. They are far fewer than the 79 responsibilities in our first draft that elicited the negative reactions from practitioners.

6. A TOOL FOR USING THIS PATTERN LANGUAGE

To bring the power of this pattern language to the design and evaluation of a software architecture, we built an HTML-based prototype of a tool for software architects (Figure 5) [15]. The complexity of the pattern language has been simplified, consistent with the lessons learned from our laboratory experiments, but the relationships in the pattern language dictate the content that is presented to the architects. Just as Alexander describes a "tick mark" (i.e., checklist) procedure for deciding which patterns are included in a project's pattern language [*p. xxxviii-xxxix A Pattern Language]*, our tool suite uses checklists at several scales to allow the system developers to tailor the patterns to their specific project. First, we envision a checklist of End-User USAPs that can be selected and prioritized by a team of stakeholders, and their initial decisions propagate to this presentation to the software architect. Figure 5 is what would result if the User Profile, Environment Configuration, and Alarms and Events End-User USAPs had been selected as important by the stakeholders.

Figure 5 shows the screen after the architect has selected the Authoring USAP to work on from the navigation pane on the left, and the Create a Specification principal component of Authoring appears at the top of the screen. The responsibility to Create a Specification, is first. The Authoring Foundational USAP itself would express this responsibility with a parameter, SPECIFICATION, as follows

The system must provide a way for an authorized author to create a SPECIFICATION.

However, the tool fills in that parameter so the architect sees the three specific specifications he or she needs to be able to create: the user profile itself for the User Profile USAP, a configuration description for the Environment Configuration USAP, and the conditions for alarms, events and alerts for the Alarms and Events USAP. The fact that all three are expressed in the same responsibility emphasizes to the architect that there can be commonality of concept, structure and perhaps even code, in the architecture that fulfills this responsibility. However, the architect can decide individually whether the architecture already addresses the responsibility or needs to be modified to do so for each of the three USAPs. He or she can also decide that this responsibility is not applicable to their system for one or more of the USAPs, again, analogous to Alexander's tick marks at every level of the pattern language.

In deciding whether, or how, to design the architecture to support this responsibility, the next level of elements in the pattern language are also at the architect's disposal. The Implementation Details can be shown after the statement of the responsibility (Figure 5 shows it expanded). This architecture-neutral prose suggests which portions of the system should be involved, how they should communicate, and how they should behave to implement the responsibility. The tool also fills in any parameters at this level (passed through from the higher-level), as shown.

By using the word "portion of the system" instead of a visual description in the form of a UML-style diagram the architect can project the words onto her/his design and verify that the portion exists or, if not, design a new part in the solution corresponding to the "portion of the system" and its described activities.

At this writing, we have translated the responsibilities of Authoring, Execution with Authored Parameters, and Logging with respect to Execution with Authored Parameters, to the pattern language and the prototype tool. We can estimate the reduction in materials presented to software architects over our original pattern catalogue approach by counting degree of reuse in these three areas. Without the reuse inherent in the pattern language, Authoring, Execution with Authored Parameters, and Logging for all three end-user USAPs contains 83 responsibilities; 26 for User Profile, 26 for Environment Configuration, 31 for Alarms and Events. (Note: Alarms and Events has fewer than the originally drafted 79 because some were eliminated in revision, and, to a lesser extent, because a few responsibilities derived from the Authorization and Logging foundational USAPs have not yet been translated to the pattern language.) With the pattern language, a total of 31 responsibilities cover all three end-user USAPs; 26 are shared by all three, with 8 slight specializations for User Profile and Environment Configuration, and 5 are specific to Alarms and Events. The 63% reduction, from 83 to 31, is the result of reusing 84% of the responsibilities. Whether this level of reuse will continue as we expand the pattern language is a question for future research.

6.1 Using the USAP pattern language tool

We tested the prototype USAP pattern language tool with ABB's software architects from the project that requested these three high-priority USAPs.

Recall that in the paper and pencil laboratory tests, the participants did not seem to consider all the responsibilities in their designs. We addressed this problem by implementing an interactive, hierarchical checklist, for the software architect to work through. In the center pane of the page, each responsibility is shown, with the parameters filled in from the end-user USAPs. Below the text of the responsibility are links that can be expanded to show the rationale for the responsibility (the forces that impact the solution) and suggested implementation details (the lower-level pattern that completes this responsibility). Below that are the names of each end-user USAP that uses that responsibility and a set of radio-buttons that reflect the architecture's state in relation to the responsibility in the context of the end-user USAP: "Not yet considered" (the default state set by the tool), "Architecture addresses this", "Must modify architecture" and "Not applicable". The intended workflow is for the architect to read the responsibility and, optionally, its rationale and implementation details, then consider whether the architecture needs to be changed to address this responsibility for each of the end-user USAPs his or her team has decided is high priority. The architect selects the appropriate state for each of the end-user USAPs by clicking a radio button. When all of the end-user USAPs related to a given responsibility are changed to a state other than "not yet considered", the checkbox next to the responsibility is checked off automatically. When all of the responsibilities in a set in the hierarchy are checked off,

the checkbox next to their parent is checked off automatically. If the architect wishes to discuss a responsibility with the remainder of the design team or other stakeholders, a check-box "Discuss this" can also be checked. The state of the discussion checkbox does not affect the automatic checking-off of the hierarchy.

Thus, the architect must click radio buttons for every responsibility and the expectation is that he or she will read the responsibility and consider the state of the architecture in order to do so. Also observe that the name of each of the three USAPs appears under the responsibility and that the architect must respond to each responsibility in the context of each USAP. It is possible that the responses will vary among the USAPs and making the different USAPs explicit will encourage the architect to consider each responsibility's applicability for each USAP. It will also remind the architect that these three USAPs share this responsibility and that they may also share architectural solutions. This is how we envision conveying the compression of multiple USAPs to architects, in response to the third complaint we got from the practicing architects when we showed them the draft responsibilities Once the designers have considered and responded to all of the responsibilities, they can generate a "to do" list. This is a list of the responsibilities that either have not yet been considered or that require a modification of the architecture and/or further discussion. The "to do" list can then be incorporated into whatever project management scheme the designers use.

Thus, we expect the architect to move fluidly between several levels of our pattern language as he or she evaluates the architecture design. They will read, and decide on the applicability of each responsibility. They will read the implementation details, the lowest-level pattern, and assign them to the portions of their overarching architecture. Thinking about the implementation details may send them back to the responsibility level to reassess their decisions. Whole portions of Foundational USAPs can be included or excluded depending on conditions specific to their design. We do not envision, however, as fluid a movement between Foundational USAPs and End-User USAPs, because

.

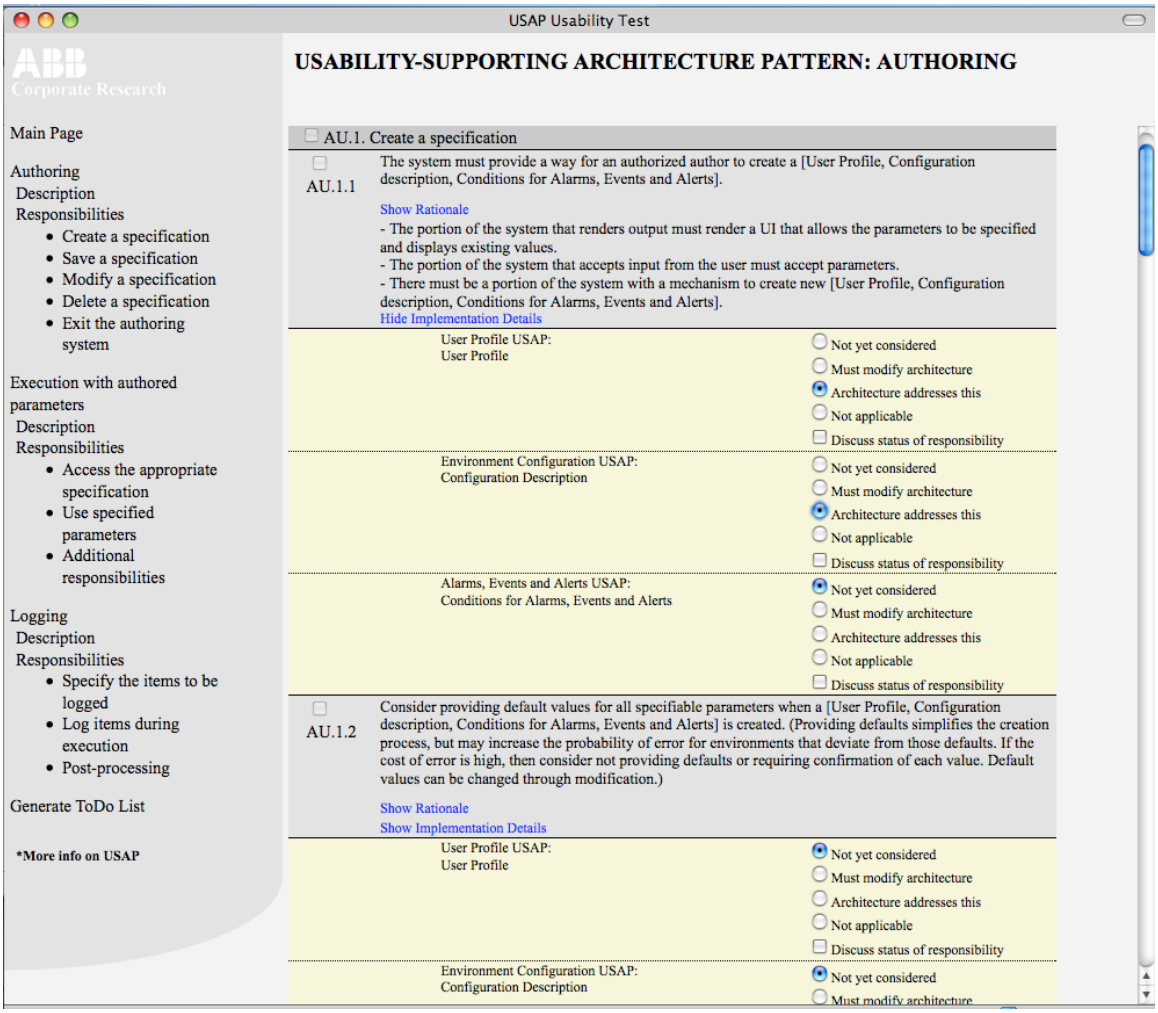

Figure 5 Screen shot of the delivery tool for the USAPs

decisions about these two levels are made by different types of people. Software architects alone can make decisions from the Foundational USAP level down, but End-User USAPs are prioritized by the team of stakeholders who represent different aspects of the company and set requirements. Information can easily flow down from End-User USAP decisions to the lower levels, but we expect the process of revisiting earlier decisions about End-User USAPs is more involved.

6.2 Results of using the USAP delivery 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 architect was senior and had created most of the preliminary design. The second architect had recently joined the project but has a solid background as software architect at an automobile company.

The two architects from the product line system project used the USAP delivery tool in one session lasting six hours interrupted by a one hour break for lunch and two fifteen-minute breaks for coffee. They examined and discussed a responsibility, made notes 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.

Overall the designers felt that the USAP delivery tool was quite helpful, as reflected by some of the things they said in a post-test interview:

Designer 1: Yeah, I, I think it's, it's a very easy way to get some kind of review of your work. You will not get the complete picture of all your work, but it will be a very good check, or at least an indication of the completeness of your system.

Designer2: 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 to have these points of view come in this early. I think we have identified at least a couple of new subsystems.

Designer 2: The most useful thing with this tool is that it guides your thoughts, and it helps you to think about the architecture that you have from different perspectives.

In contrast to the negative reaction to UML-style diagrams in previous USAPs, as the designers 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 could 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 like the informationally-equivalent UML diagrams.

In summary, the reactions of the software architects were very positive. The designers had viewed all implementation details as they considered responsibilities, indicating that they found the reformatting of USAP content helpful. They also asked for a copy of the tool so that they could have it available as they worked through their to-do list.

Perhaps more important than how the software architects reacted to the USAP delivery tool was the impact on their software architecture design. The designers of the new product line architecture identified 19 "Must modify architecture" responses to the 31 responsibilities presented in the tool. After reflection, they identified 14 major issues that would suggest possible changes to the core architecture in order to support their chosen usability scenarios, including the identification of two new subsystems. The senior designer presented to management that two architects using this tool for 6 hours saved the team 5 weeks of work, a return on investment (ROI) of about 17 to 1.

7. Conclusions

On the one hand, providing professionals with a checklist of activities they should perform is a very old concept, computerizing the checklist is not a major step, and the resulting tool is extremely simple to use. On the other hand, having a senior architect claim a 17 to 1 ROI - one day work saved five weeks - is an amazing result. One study with one estimate is not scientific evidence but this study is one of the few reports of ROI with respect to the use of *any* architectural technique. This is an existence proof that architectural knowledge can be encoded into very simple tools and still be effective. Architectural tool builders might consider simple methods to encode their knowledge rather than attempting very sophisticated tools.

Furthermore, three aspects of this work are significant.

1. A pattern language emerged from this work with patterns at multiple levels, from those that have direct benefit to users (the End-User USAPs) through several levels of patterns of responsibilities, down to architecture-neutral patterns of components, communication and behavior, expressed in prose instead of diagrams.
2. The pattern language allowed us to reuse 84% of the responsibilities, resulting in a 63% reduction in number of responsibilities that had to be presented to architects.
3. The prototype tool was well-received and the senior architect claimed that it provided a 17 to 1 ROI of architect time. (This ROI does not yet include any benefit to the users of the system from having a more usable interface.)

On a final note, there is nothing in the USAP delivery tool that is specific to usability patterns. Just as Alexander's patterns combine as disparate entities as garages and gardens, our pattern language could extend to combine seemingly disparate quality attributes connected only because they exist in a single software system. Any quality attribute where the requirements can be expressed as a set of responsibilities, e.g. security, could likely be included in the pattern language and the tool and the interplay between security and usability concerns may be revealed. It is our hope that such research can make a living language of what we have presented here.

> *We have spent years trying to formulate this language, in hopes that when a person uses it, he will be so impressed by its power, and so joyful in its use, that he will understand again, what it means to have a living language of this kind.*
>
> *(p. xvii [3])*

8. ACKNOWLEDGMENTS

Our thanks to Fredrik Alfredsson and Sara Lövemark for their contributions to the Alarm and Event End-User USAP, and to the ABB software architects who diligently used our prototype tool in their work. We also thank the anonymous EICS2009 reviewers; their comments improved this paper immensely. This research was supported in part by funds from ABB Inc. The views and conclusions in this paper are those of the authors and should not

be interpreted as representing the official policies, either expressed or implied, of ABB.

9. REFERENCES

[1] Adams, R. J., Bass, L., & John, B. E. (2005) Applying general usability scenarios to the design of the software architecture of a collaborative workspace. In A. Seffah, J. Gulliksen and M. Desmarais (Eds.) *Human-Centered Software Engineering: Frameworks for HCI/HCD and* Software *Engineering Integration*. Kluwer Academic Publishers.

[2] Alexander, C. (1979). The Timeless Way of Building. USA: Oxford University Press. ISBN 0195034028

[3] Alexander, C., Ishikawa, S., Silverstein, M. (1977). A Pattern Language: Towns, Buildings, Construction. USA: Oxford University Press. ISBN 019501919

[4] Barbacci, M., Ellison, R., Lattance, A, Stafford, J., WeinStock, C, and Wood,, W., "Quality Attribute Workshops, 3rd Edition", Technical Report No. SEI-2003-TR.016, Pittsburgh, PA, 2003.

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

[6] Buschmann, F., Meunier, R., Rohnert, H. and Sommerlad , P., Pattern-Oriented Software Architecture Volume 1: A System of Patterns, Wiley, 1996

[7] Clements, P., Northrop, L., Software Product Lines: Practices and Patterns, Addison Wesley, 2001.

[8] Dearden, A., Finlay, J. (2006). Pattern Languages in HCI: A critical review. Human-Computer Interaction 21 (1).

[9] Folmer, E. (2005) Software Architecture Analysis of Usability, Ph.D. thesis. Department of Computer Science, University of Groningen, Groningen.

[10] Folmer, E., van Gurp, J., Bosch, J. (2003) A Framework for capturing the relationship between usability and software architecture; Software Process: Improvement and Practice, Volume 8, Issue 2. Pages 67-87. April/June 2003.

[11] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995) Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, Reading, Massachusetts, 1995.

[12] Golden, E., John, B. E., Bass, L. (2005) Quality vs. quantity: Comparing evaluation methods in a usability-focused software architecture modification task. Proceedings of the 4 th International Symposium on Empirical Software Engineering (Noosa Heads, Australia, November 17-18 2005).

[13] Golden, E, John, B. E., & Bass, L. (2005) The value of a usability-supporting architectural pattern in software architecture design: A controlled experiment. Proceedings of the 27th International Conference on Software Engineering, ICSE 2005 (St. Louis, Missouri, May, 2005).

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

[15] John, B. E., Bass, L. J., Sanchez-Segura, M-I. & Adams, R. J. (2004) Bringing usability concerns to the design of software architecture. Proceedings of The 9th IFIP Working Conference on Engineering for Human-Computer Interaction and the 11th International Workshop on Design, Specification and Verification of Interactive Systems, (Hamburg, Germany, July 11-13, 2004).

[16] Juristo, N., Moreno, A. M., Sanchez-Segura, M. (2007), Guidelines for Eliciting Usability Functionalities, IEEE Transactions on Software Engineering, Vol. 33, No. 11, November 2007, pp. 744-758.

[17] Stoll, P., John, B., Bass,L., & Golden, E. (2008) Preparing Usability Supporting Architectural Patterns for Industrial Use. In Proceedings of the International Workshop on the Interplay between Usability Evaluation and Software Development, I-USED 2008 (Pisa, Italy, September 24th, 2008)

[18] Stoll, P., Wall, A., Norström, C.: Guiding Architectural Decisions with the Influencing Factors Method. WICSA. IEEE, Vancouver (2008)

[19] Tidwell, J. (2006), Designing Interfaces: Patterns for Effective Interaction Design. O'Reilly Media: Sebastopol, CA.