Reconciling Requirements and Architectures with the CBSP Approach in an iPhone App Project

Harald Vogl Klaus Lehner
Catalysts GmbH
Ottensheimer Straße 27
4040 Linz, Austria
{harald.vogl | klaus.lehner}@catalysts.cc

Paul Grünbacher Alexander Egyed
Systems Engineering & Automation
Johannes Kepler University Linz
4040 Linz, Austria
{paul.gruenbacher | alexander.egyed}@jku.at

Abstract. **There are only few methods available that help refining software requirements to software architectures. An example is the CBSP (Component-Bus-System-Property) approach that uses general architectural concerns to classify and refine requirements and to capture architectural trade-off issues and options. This paper reports about experiences of applying CBSP in an industrial project in the area of mobile applications. We illustrate CBSP using project examples. We discuss how the approach was tailored and present extensions we developed. In particular, we show how CBSP can be used together with the Goal-Question-Metric approach to guide architectural decisions. We close our paper with a discussion of lessons learned during this project.**

Keywords-requirements; architecture; Component-Bus-System-Property (CBSP) approach; traceability; rationale capture

I. INTRODUCTION

Understanding the relationship between software requirements and a software system's architecture is a big challenge in industrial practice. Part of the challenge stems from the fact that requirements and architectures use different terms and concepts to capture the elements relevant to each [1]. It has been pointed out that crafting a system's requirements and its architecture should be done in a concurrent manner by interleaving their development [2][3]. For instance, Nuseibeh has presented the Twin Peaks model as an adaptation of Boehm's spiral life-cycle model [2]. It emphasizes the equal importance of both requirements and architectures and suggests progressively developing more detailed requirements and architectural specifications in a concurrent manner.

Despite some advances and reported experiences [4] there are still only few methods available that emphasize the relationship between software requirements and an architecture satisfying those requirements [5]. For the most part, these approaches focus on capturing rationale when developing requirements and the architecture. The importance of capturing rationale was already emphasized in Perry and Wolf's model of software architecture comprising elements, form, and rationale [6]. The Archium approach by Jansen and Bosch [7] proposed to capture design decisions when developing or evolving a software architecture.

Other approaches emphasize the relationship between requirements and architecture: the ATAM technique [8] supports the evaluation of architectures and architectural decision alternatives in light of quality attribute requirements. Brandozzi and Perry [9] use the term architecture prescription language for their extension of the KAOS requirements specification language towards architectural dimensions. Compared to these rather heavyweight approaches, CBSP (Component-Bus-System-Property) [1][10] is a simple technique that aims at reconciling requirements and architectures by capturing early decisions about the architecture.

The fundamental idea behind CBSP is that software requirements may contain explicit or implicit information relevant to a software system's architecture. CBSP aims at identifying and eliciting this relevant information. The approach includes activities for classifying requirements to understand in what way a requirement might influence the basic artifacts of architectures – that is its components, bus (connectors), and configuration. CBSP also recognizes the importance of detecting conflicts among stakeholders early (for example by recording disagreement among multiple architects in identifying architectural buildings blocks in response to architectural relevance). It also provides guidance for selecting appropriate architectural styles.

The CBSP taxonomy covers a set of general architectural concerns derived from existing software architecture research [11]. These dimensions are applied to systematically classify and refine requirements and to capture architectural trade-off issues and options (e.g., the impact of a connector's throughput on scalability). Each requirement is assessed for its relevance to the system architecture's components, connectors (buses), topology of the system or a particular subsystem, and their properties. Thus, each derived CBSP artifact captures an architectural concern and represents an early architectural decision or option.

In this paper we present experiences of using the CBSP method in a project developing a mobile app for task management software. We discuss the project background and requirements. We then illustrate CBSP based on concrete system requirements. We discuss experiences of applying the approach and explain why and how we deviated from the original approach. Finally, we discuss lessons learned.

978-1-4577-0924-1/11/$26.00 © 2011 IEEE

Figure 1. The *taskmind* app on the iPhone. The left screen shows notifications and status information about the involved tasks as well as navigation support. The middle screen shows details of one task. The right screen shows an example of a change history and notes of one task.

II. TASKMIND IPHONE APP PROJECT: BACKGROUND AND REQUIREMENTS

Organizations use the team productivity software *taskmind* (http://www.taskmind.net/) to increase the understanding of team members regarding work processes; to ensure that project members are aware about the progress of collaborative tasks and their contributions; to improve the assignment of tasks; and to balance the distribution of tasks in teams. At the surface, *taskmind* manages tasks and appointments with due dates and priorities. However, the system has distinctive features that foster self-organization for ad-hoc tasks. For instance, it supports the collaboration of knowledge workers with features for delegating tasks including the management of confirmations. It also provides features to distribute tasks to avoid overloading individual team members. Users can chat about tasks with collaborators. *taskmind's* real-time collaboration features support ad-hoc workflows and the task chain mechanism helps reducing the duration of tasks. The taskmind product is currently used by about 6,900 users in 3,000 projects for managing 85,000 tasks and appointments. The size of the code base is about 230 KLOC.

The *taskmind* app project was initiated to develop an iPhone, iPad and iPod touch client for *taskmind* to bring the productivity gains of tightly collocated teams to mobile workers [12]. The *taskmind* app user interface is shown in Figure 1. Mobile knowledge workers use the app on smartphones both in an online and offline mode. The app connects the mobile users to co-workers in the office to support larger teams in getting work done. The code size of the taskmind app is about 30 KLOC in addition to the existing architecture.

As discussed the CBSP technique supports the refinement of *existing* requirements. The approach does not assume a specific requirements elicitation method. It relies however on the quality of the initial requirements that serve as a starting point. In this project detailed requirements were elicited in several stages with widely used and proven RE techniques. In the first stage the analyst created *persona profiles* [13] to characterize the key types of users. A persona is a fictitious character that represents a large group of users. The analyst then used the persona profiles to guide the elicitation of product requirements. Scenario-based requirements engineering is one of the most effective ways to eclicit and document requirements. The analyst thus used *scenarios* to define typical interactions of the personas with *taskmind*. Key scenarios involve the viewing of task details, delegating of tasks to other users, creating notes, setting up ad-hoc workflows, and creating tasks offline. The analyst finally used *scenario walkthroughs* [14] to elicit 29 detailed (F)unctional and 25 (N)on-functional requirements. These requirements were documented using the Volere template [15]. An abbreviated list of selected requirements is shown here:

F1. Login with existing account
F2. Register new user account
F3. Retrieve news and daily journal
F27. View news and daily journal in offline mode
N7. Synchronization rate 20 seconds
N8. App startup time maximum 2 seconds
N9. Quick display of tasks and appointments
N13. App must support up to 10,000 data objects in online mode
N15. Minimize amount of data transfer
N16. Minimize energy consumption
N21. Ensure portability of app to Android, Windows Phone and Blackberry OS
N24. Avoid data inconsistencies (e.g., between online and offline mode)

III. CBSP IN ACTION

The CBSP approach was applied by Catalysts. The role of the academic researchers was to provide training and suggestions on applying CBSP if requested. E.g., the researchers explained the CBSP concepts to ensure their proper use when recasting and refining the requirements.

A. From Requirements to CBSP Elements

We illustrate the six CBSP dimensions with examples from the iPhone app project. The six dimensions are based on basic architectural constructs [11].

C elements describe or involve an individual component or subsystem in an architecture. The approach distinguishes between data components (C_d) and processing components (C_p). For example, the requirement "The mobile user should be able to change the details of a task" involves processing and affects a processing component *C_p: GUI Support for Editing Tasks* when translated into CBSP. However, CBSP encourages the architect to also consider implicit information in requirements. In this example, the architect may assume that there must be some data for tasks and thus define *C_d: Data for tasks.* While this requirement does not reveal what kind of data is meant to be edited, other requirements might provide this information. The architect thus not only creates new CBSP elements but also explores how they might relate to existing CBSP elements. If, for example, there is already a CBSP element describing the data for tasks and deadlines then the C_d element would not be created. Instead the architect would link the C_p element with the existing C_d element.

B elements describe or imply communication (as in bus, middleware, or connector). For example, the requirement "Changed tasks must be stored in a data file immediately for offline support" might yield the need for a communication channel. At this point it might be impossible to tell whether a simple file streamer or a more elaborate middleware infrastructure is needed. However, using CBSP the architecture can prescribe an interaction between the GUI component handling the table and some persistence component. It might well be that the GUI component and persistence component are the same in the final architecture. However, CBSP is meant to be flexible in not forcing a lingo that limits the architectural possibilities later.

S elements describe system-wide features or features pertinent to a large subset of the system's components and connectors. For example, the requirement "The user should be allowed to select and filter tasks by different sort criteria" could be captured as a system level property *S: the system should separate data from visualization* in CBSP.

CP elements describe or imply non-functional properties that affect data or processing components. For example, the requirement "the user should be able to visualize both local and remote data, both without noticeable latency" affects the property of a GUI processing component (perhaps the same one identified above) and results in *CP: The GUI should treat local and remote data alike*. Yet again, we notice that this requirement also implies another CBSP element: a property relevant for a bus as is discussed next.

BP elements describe or imply non-functional properties affecting communication. The above requirement also affects the communication between GUI and remote data. *BP: The communication between the GUI and the remote data must allow real-time performance.* Indeed, it is not at all obvious whether this requirement pertains to the same communication identified earlier about access to local storage (see B elements). A fact that should lead to further CBSP elements such as *B: Required interaction between the GUI and remote storage*. Perhaps later requirements clarify that these two communications are the same or distinct. If this is not clarified later then this may require addition requirements solicitation.

SP elements describe or imply system-wide properties. For example, security or uptime requirements pertain to the entire system and often cannot be attributed to single components, communications, or even subsystems. An example from the *taskmind* iPhone app is *SP: all data must be securely transferred and stored* as to ensure privacy.

B. Iteratively Refining the CBSP Model

CBSP purposely does not prescribe terminology beyond the simple constructs discussed above to leave architects with the utmost freedom in expressing what to model at what level of detail. However, CBSP provides additional constructs to define traceability and to eliminate redundancies among CBSP elements and between CBSP elements and requirements.

CBSP elements can be quite redundant due to overlapping requirements. For example, above we have seen that different requirements reveal facts about task data (leading to C_d elements above). Instead of restating the same or similar CBSP elements, CBSP elements are consolidated – usually by merging them – to increase clarity. Furthermore, CBSP also maintains mappings between its elements and the requirements that spawned them. This is particularly useful if requirements change and an engineer needs to assess the impact of the change on components, connectors, systems, or their properties. And, this is also useful for maintaining the rationale why CBSP elements exist – perhaps if the purpose and role of CBSP elements need to be re-assessed later.

Figure 2 depicts the subset of the *taskmind* requirements from Section II mapped to their corresponding CBSP elements. The CBSP elements are shown in the middle, surrounded by functional requirements on the left and non-functional requirements on the right. The figure reflects the final state of these elements, after all redundancies have been eliminated. It is evident that many CBSP elements are mapped to several requirements, implying that different requirements either contributed to the same CBSP elements or to different aspects of those elements.

CBSP elements may also be linked logically. For example, a property of a component (CP) might be related to a component (C). In Figure 2, such relationships are implied by logically grouping C, B, or S elements with their corresponding CP, BP, and SP elements. The grouping mechanism proved to be sufficient for *taskmind* though we are not necessarily arguing that the present CBSP visualization is adequate always. For example, a CP element could well relate to multiple C elements.

The resulting CBSP model was derived from 29 functional requirements and 25 non-functional requirements. It comprises 12 C (6 C_d, 6 C_p), 25 CP, 3 B, 6 BP, 3 S, and 2 SP elements.

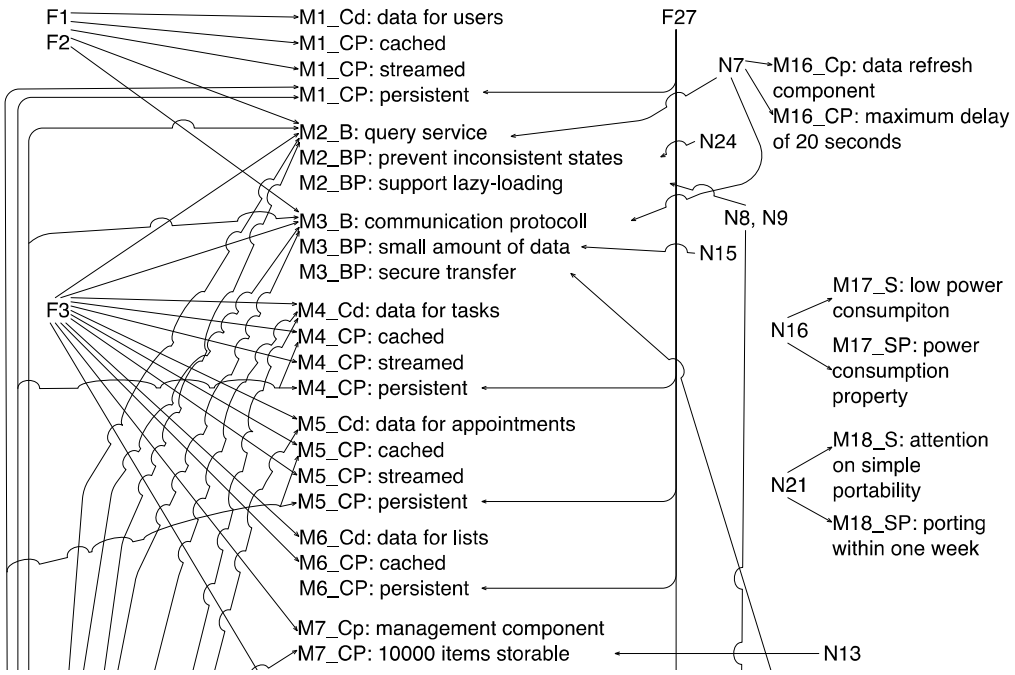

Figure 2. Partial CBSP Model of the taskmind iPhone app.

The ability to link CBSP elements also makes CBSP suitable for finding missing requirements. For example, certain CBSP elements can stand alone, such as GUI, whereas others indicate missing information that needs to be enriched. The element *C_d: Data for tasks* is a good example of that. It is clear that something is missing as the kind of task data still needs to be defined (e.g., the attributes and relationships to other kinds of data). If this does not become clear through other requirements the architect has identified an incomplete requirement.

C. Understanding Architectural Trade-Offs

The original CBSP approach required the architect to identify how CBSP elements fit different architectural styles. In context of the *taskmind* app project, it was however not straightforward to assess the suitability of styles as suggested in [1] because of a range of restrictive non-functional requirements. While in previous work it was suggested to define the pro's and con's of architectural choices ad-hoc, this turned out to be too complex for the *taskmind* app. The architect thus first assessed important goals and metrics for gauging the suitability of architectural choices.

This activity was guided by using the Goal-Question-Metric approach [16] for key architectural decisions (e.g., regarding startup of the app within 2 seconds). The measurement goal was typically the comparison of different architectural options that were derived by analyzing the CBSP elements. Table I shows an excerpt of the GQM model that was defined based on taskmind's CBSP elements An example is the selection of the data serialization mechanism used for transmitting data between server and client and for storing information at the client device offline. In both cases CP elements suggest a small amount of data and fast processing. Several metrics were used for assessing different architectural options.

TABLE I. EXCERPT OF GOAL-QUESTION-METRIC MODEL TO ASSESS ARCHITECTURAL OPTIONS BASED ON THE CBSP ELEMENT M3_BP: SMALL AMOUNT OF DATA.

Goals	Questions	Metric
Purpose: Compare *Issue:* Amount of data *Object:* Different data formats for persisting of data objects *Viewpoint:* Software architect	Differences with respect to the amount of data?	Data that needs to be transferred (in bytes).
Purpose: Compare *Issue:* Performance *Object:* Different data formats for persisting of data objects *Viewpoint:* Software architect	Differences with respect to serialization and deserialization time?	Time for (de-) serializing objects (in ms).

For the client-server communication a portable data format is required due to different programming languages on the server and client. The considered options were text-based formats like XML or JSON or the portable binary format Hessian. For the offline mode, all the data must be stored on the client device. Therefore additional programming language specific data formats can be used. Table II shows different serialization options compared with respect to the amount of data and the time needed for serializing and de-serializing 500 task objects on an iPhone 3GS. The table includes a custom binary format that was specifically defined for the *taskmind* iPhone app where no metadata is stored (and can therefore only be used on the

client for the offline repository). For example, a requirement in the project was the startup of the app within a few seconds. Early tests showed using the text formats and the binary Hessian format was too slow. Therefore the custom binary format was implemented to satisfy the requirement as shown in Table II.

TABLE II. THE CUSTOM FORMAT REDUCES THE AMOUNT OF DATA THAT NEEDS TO BE TRANSFERRED AND THE TIME FOR (DE-)SERIALIZATION.

Data format	Bytes transferred	Serialize (Run 1/2/3)	Deserialize (Run 1/2/3)
Custom	360,504	114.8 116.4 135.9	209.5 209.5 184.2
Hessian	950,696	1171.6 1,150.1 1,158.4	4,284.4 4,280.2 4,254.9
XML	1,250,081	3,778.2 3,819.1 3,796.7	6,128.8 6,101.8 6,098.6
JSON	766,001	2,015.1 1,959.7 1,992.5	4,619.5 4,328.4 4,797.9

It is clear, that such architectural choices cannot be defined ad-hoc by architects. In this example, the architectural decision is important and should be retained and documented – for example, to understand later why the custom serialization was chosen.

D. From CBSP Elements to Architectures

Once architectural choices are enumerated and analyzed, the question of how to encode CBSP elements in architectural elements emerges. In the most straightforward case, a CBSP element directly becomes an architectural element. However, as the *taskmind* project has shown, architectural elements typically cover multiple CBSP elements.

TABLE III. MAPPING OF CBSP ELEMENTS TO COMPONENTS OF THE ARCHITECTURE (EXCERPT).

CBSP elements	Architecture component
M1_Cd: data for users, M4_Cd: data for tasks, M5_Cd: data for appointments, M6_Cd: data for lists, M9_Cd: data for projects, M10_Cd: data for tags, M12_Cd: data for change history and notes	Data Objects
M7_Cp: management component, M8_S: strict separation of data and visualization	Proxies
M11_Cp: task state transition component	State Machine
M2_B: query service	Services
M2_BP: lazy loading, M3_BP: small amount of data	Input Handler, Output Handler
M3_B: communication protocol	Communication Protocol
M13_Cp: Switch between offline and online mode, M14_Cp: Offline component, M15_B: Connection between offline component and file system	Offline Management
M16_Cp: data refresh component	Auto-Update
M17_S: low power consumption, M18_S: attention on simple portability	System wide feature affecting all components'

Table III shows some examples that illustrate this observation. For instance, a central *Data Objects* component has been implemented that manages all different data elements in the system. Similarly, the *Offline Management* component provides required capabilities that again cover multiple CBSP elements.

Yet, the architecture of the *taskmind* system already existed partially because the iPhone app was a lightweight version of a desktop application and the goal was to ensure maximal reuse with minimal changes to the existing architecture.

IV. DISCUSSION AND LESSONS LEARNED

Use of CBSP dimensions. Applying the 6 architectural dimensions in a lightweight fashion provided good guidance and led to the systematic analysis of requirements for architectural concerns. The effort for building the CBSP model (without the GQM extensions) was about two person days. This is rather low compared to the effort of defining the requirements which took about a person week. The process also helped identifying missing requirements (e.g., if a certain architectural dimension was not addressed). This is confirmed by the fact that almost 50% of the requirements were non-functional many of which were found by explicitly analyzing the properties of CBSP elements and going back and forth between requirements and architectural issues. Additionally, refining or generalizing a requirement also required consulting the system's customers for additional context and information. As such, while it would undoubtedly be very useful, it is unrealistic to expect that formal rules could be provided for transforming a requirement into more specific or general CBSP elements. The intermediate model facilitates the mapping of requirements to architectures. Furthermore, the intermediate CBSP model eases capturing and maintaining arbitrary complex relationships between requirements and architectural model elements, as well as among CBSP model elements.

Using CBSP in the presence of an existing architecture. The original work on CBSP primarily addressed the use of CBSP in case no architecture is available. In the *taskmind* app project the process was different as the server architecture was already in place when the project started. However, in some cases a careful analysis of the CBSP elements also led to changes of the original architecture. Examples are the on-demand loading of task information to optimize startup time on the mobile app (*M15_BP: Support of lazy-loading of offline stored data*) or the lazy and on demand loading of notes triggered only when user scrolls down to optimize response time. Also, data in the offline cache are only checked for server-side updates when retrieved to be displayed locally (cf. *M2_BP: Support of lazy-loading of remote elements*). CBSP turned out to be quite suitable for modifying an existing architecture which was not created using CBSP. However, the latter aspect could be a problem for later maintenance.

Understanding architectural options. Deriving architectural styles from CBSP elements is not straightforward. Our basic observation is that 'one size does

not fit all' – at least not when it comes to understanding the pro's and con's of architectural choices. We also found that evaluating architectural choices is hard unless there are clearly defined goals, questions, and metrics. In contrast to the published papers on CBSP we thus combined GQM and CBSP to evaluate important architectural choices in a few specific cases as shown in the examples. Our observation is that CBSP and GQM are a good fit, in particular GQM provided a better and more meaningful structure for describing the CP, BP, and SP properties and assessing their satisfiability.

Flexibility with respect the choice of requirements elicitation method. While we used the WinWin approach in earlier projects to elicit and negotiate requirements we applied CBSP with other requirements and architecture definition techniques in the *taskmind* project. The use of personas and scenario walkthroughs provided a good input to the CBSP step. We also learned that CBSP helps with assessing and completing the requirements as already discussed.

CBSP for single architects. The original CBSP was geared towards the idea that different architects contribute to the creation of CBSP elements and their subsequent refinement to architectures. This was motivated by the problem that requirements are informal and requirement problems (conflicts or incompleteness) can be identified better if multiple architects are queried as to their architectural relevance. In context of the *taskmind* project, the pragmatics of limited resources and people precluded the use of multiple architects. While doing so may weaken the chances of detecting problems early, the single architect had no problem in following the CBSP approach.

CBSP tool support and visualization. We did not use specific tool support which turned out as a problem as the CBSP model was rather complex. The lack of tool support was thus seen as the biggest downside, particularly when it came to the ability to visualize CBSP elements and their many kinds of relationships. Tool support would also be required for voting on architectural relevance of requirements if multiple architects are involved. This was not the case in this project as was discussed above. Furthermore, the taskmind app is continuously updated to meet additional customer requirements such as voice tasks; attachments; urgencies; contact management. Maintaining the CBSP model is however cumbersome and demonstrates the need for proper tool support.

V. CONCLUSIONS

In this paper we reported experiences of applying the Component-Bus-System-Property (CBSP) approach in a mobile application project. We discovered that although CBSP worked well for the purpose of the project some tailoring was necessary (e.g., dropping the voting of multiple architects). Furthermore, as part of this project we also came up with an extension to use CBSP together with the Goal-Question-Metric to guide the assessment of architectural options. CBSP worked well in the presence of an existing architecture. It is also flexible regarding the choice of the requirements elicitation method. A drawback is the current lack of tool support. We thus plan developing a CBSP rationale capture tool in future work. The tool will require features to visualize the CBSP model and navigation support to identify related requirements and architectural elements.

ACKNOWLEDGMENT

The authors would like to thank Daniela Lettner for her feedback on an earlier draft of this document.

REFERENCES

[1] P. Grünbacher, A. Egyed, N. Medvidovic, Reconciling software requirements and architectures with intermediate models. Software and System Modeling 3(3):235-253, 2004.

[2] B. Nuseibeh, "Weaving together requirements and architecture." IEEE Computer, 34(3), 2001, pages 115–119.

[3] W. Swartout and R. Balzer, "On the Inevitable Intertwining of Specification and Implementation," Comm. ACM, vol. 25, no. 7, 1982, pp. 438-440.

[4] C. L. Chen, D. Shao, and D. E. Perry. 2007. An Exploratory Case Study Using CBSP and Archium. In *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent* (SHARK-ADI '07). IEEE Computer Society, Washington, DC, USA.

[5] T. Yue, L. Briand, and Y. Labiche. A Systematic Review of Transformation Approaches between User Requirements and Analysis Models, Requirements Engineering (Springer), 2010.

[6] D. E. Perry and A. L. Wolf. "Software Architecture". August 1989. http://users.ece.utexas.edu/~perry/work/swa/

[7] A. Jansen, J. Bosch. Software Architecture as a Set of Architectural Design Decisions. 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), 2005, pages 109-120.

[8] R. Kazman, M. Barbacci, M. Klein, S.J. Carriere, S.G. Woods, Experience with Performing Architecture Tradeoff Analysis, Proceedings of the 21st Int'l conference on Software engineering (ICSE '99). ACM, New York, NY, USA, pages 54-63.

[9] M. Brandozzi, D.E. Perry, Transforming Goal-Oriented Requirement Specifications into Architecture Prescriptions, STRAW Workshop "From Software Requirements to Architectures" at ICSE, 2001.

[10] P. Grünbacher, A. Egyed, N. Medvidovic: Reconciling Software Requirements and Architectures: The CBSP Approach. Proceedings Int'l Conference on Requirements Engineering, 2001, pages 202-211.

[11] N. Medvidovic, R. N. Taylor: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Software Eng., 26(1):70-93. 2000.

[12] C. Steindl and C. Federspiel. Get to the next level with taskmind! In CEO Today, special report on Agile Software Development, 2-2010.

[13] M. Aoyama. Persona-and-Scenario Based Requirements Engineering for Software Embedded in Digital Consumer Products. IEEE Int'l Conference on Requirements Engineering, pages 85–94, 2005.

[14] N. Seyff, N. Maiden, K. Karlsen, J. Lockerbie, P. Grünbacher, F. Graf, and C. Ncube. Exploring how to use scenarios to discover requirements. Requir. Eng., 14:91–111, April 2009.

[15] S. Robertson and J. Robertson. Mastering the Requirements Process (2nd Edition). Addison-Wesley Professional, 2006.

[16] V. Basili, G. Caldiera, and D. Rombach. Goal/question/metric paradigm. In J. Marciniak, editor, Encyclopedia of Software Engineering, pages 528–532, New York, 1994, John Wiley and Sons.