A Comparison of Requirements Specification Methods from a Software Architecture Perspective

Len Bass
John Bergey
Paul Clements
Paulo Merson
Ipek Ozkaya
Raghvinder Sangwan

August 2006

TECHNICAL REPORT
CMU/SEI-2006-TR-013
ESC-TR-2006-013

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

A Comparison of Requirements Specification Methods from a Software Architecture Perspective

CMU/SEI-2006-TR-013
ESC-TR-2006-013

Len Bass
John Bergey
Paul Clements
Paulo Merson
Ipek Ozkaya
Raghvinder Sangwan

August 2006

Software Architecture Technology Initiative

Unlimited distribution subject to the copyright.

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2006 Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

Abstract .... **vii**

1 Introduction **1**

2 Identifying Architecturally Significant Requirements **3**

3 Evaluation Criteria **5**
- 3.1 Criteria Related to How the Method Fits into the Development Process 5
  - 3.1.1 Support for Discovering Architecturally Significant Requirements 5
  - 3.1.2 Derivation from the Business and Mission Goals for the System 5
  - 3.1.3 Checking for Consistency 5
  - 3.1.4 Support for Testing 5
- 3.2 Criteria Related to the Artifacts Produced by the Method 6
  - 3.2.1 Expressiveness 6
  - 3.2.2 Ease of Organizing Architecturally Significant Requirements 6
  - 3.2.3 Support for Variability 6
- 3.3 Criteria Related to the Ease of Use of the Method 7
  - 3.3.1 Skill Level Necessary to Carry Out the Method 7
  - 3.3.2 Tooling 7

4 Evaluation of Methods for Capturing Requirements **9**
- 4.1 Requirements Specification Using Natural Language 9
  - 4.1.1 Artifact(s) Produced 9
  - 4.1.2 Process 10
  - 4.1.3 Evaluation 11
- 4.2 Use Case Analysis 12
  - 4.2.1 Artifact(s) Produced 13
  - 4.2.2 Process 14
  - 4.2.3 Evaluation 15
- 4.3 The Quality Attribute Workshop 17
  - 4.3.1 Artifact(s) Produced 17
  - 4.3.2 Process 18

4.3.3 Evaluation ... 21
4.4 Global Analysis ... 23
4.4.1 Artifact(s) Produced ... 23
4.4.2 Process ... 23
4.4.3 Evaluation ... 23
4.5 O'Brien's Approach ... 25
4.5.1 Artifact(s) Produced ... 25
4.5.2 Process ... 26
4.5.3 Evaluation ... 26

5 Related Work ... 29

6 Conclusions and Future Work ... 33

References ... 35

List of Figures

Figure 1: Partial Use Case Diagram for Online Course Management System...... 14

List of Tables

Table 1: Evaluation of the Natural Language Method 11

Table 2: Evaluation of the Use Case Analysis Method 15

Table 3: Evaluation of the QAW Method 21

Table 4: Evaluation of the Global Analysis Method 24

Table 5: Evaluation of O'Brien's Approach 26

Abstract

One of the key challenges to producing high-quality software architecture is identifying and understanding the software's architecturally significant requirements. These requirements are the ones that have the most far-reaching effect on the architecture. In this report, five methods for the elicitation and expression of requirements are evaluated with respect to their ability to capture architecturally significant requirements. The methods evaluated are requirements specification using natural language, use case analysis, the Quality Attribute Workshop (developed by the Carnegie Mellon® Software Engineering Institute), global analysis, and an approach developed by Fergus O'Brien. These methods were chosen because they are in widespread use or emphasize the capture of architecturally significant requirements.

Three problems must be solved to systematically transform business and mission goals into architecturally significant requirements: (1) the requirements must be expressed in a form that provides the information necessary for design; (2) the elicitation of the requirements must capture architecturally significant requirements; and (3) the business and mission goals must provide systematic input for elicitation process. The primary finding from the evaluation of these methods is that there are promising solutions to the first two problems. However, there is no method for systematically considering the business and mission goals in the requirements elicitation.

1 Introduction

Software systems are created to satisfy business and mission goals. To ensure that the system satisfies these goals, you must ensure that the various activities involved in the creation of the system (requirements engineering, architecture design, and implementation) conform to the business and mission goals of the system. In this report, we evaluate five methods to determine how they identify, elicit, and record the types of requirements that are important for architectural design. These types of requirements are referred to as *architecturally significant requirements* in this report. The term comes from the *Software Architecture Review and Assessment (SARA) Report* [Dominick 02]. These requirements are called *architecturally significant use cases* by the Rational Unified Process (RUP) [IBM 06] and *architectural drivers* by the Carnegie Mellon® Software Engineering Institute (SEI) [Bass 03].

In this report, we evaluate five methods for the expression of architecturally significant requirements:

1. requirements expression using natural language
2. use case analysis
3. the SEI Quality Attribute Workshop (QAW)
4. global analysis
5. an approach developed by Fergus O'Brien, which we will call *O'Brien's approach*

We chose these five methods because they are in widespread use or emphasize the capture of architecturally significant requirements.

Three problems must be solved to capture architecturally significant requirements: (1) the requirements must be expressed in a form that provides the information necessary for design; (2) the elicitation of the requirements must capture architecturally significant requirements; and (3) the business and mission goals must provide systematic input for elicitation process. The primary finding from our evaluation of these methods is that there are promising solutions to the first two problems (i.e., the expressiveness of requirements and the elicitation of architecturally significant requirements), but there is no method for systematically considering the business and mission goals in the elicitation.

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

This report consists of six sections:

1. This introduction contains some background information and a brief summary of our evaluation.
2. Section 2 describes what we mean by architecturally significant requirements.
3. Section 3 introduces the evaluation criteria that we used to examine the five methods evaluated in this report.
4. Section 4 describes the methods that we evaluated and compares them based on the evaluation criteria in Section 3.
5. Section 5 briefly describes related work.
6. Section 6 presents our conclusions and outlines our future work.

2 Identifying Architecturally Significant Requirements

Determining which requirements have architectural significance is currently a matter of experience and judgment. For any particular system, architecturally significant requirements must be coupled to the business and mission goals for the system. In this section, we discuss some factors that may cause a requirement to have a significant impact on an architecture. When we look at some of the methods for capturing requirements in Section 3, we will identify other factors.

If functionality is the only concern in designing an architecture, any structure will do. This conclusion stems from the work of Parnas more than 30 years ago [Parnas 72], and it means that some properties of the system(s) must be architecturally significant in addition to the functionality of the system. Architecturally significant requirements often arise from

- quality attributes. Security, performance, reliability, modifiability, and so forth are all quality attributes that might affect the structure of the system. Each quality attribute has a collection of structural techniques (i.e., architectural tactics [Bass 03, Ch. 5]) that are used to achieve it.
- volume of functionality. Any particular function will not have an impact on structure, but large collections of similar functionality will. One example of a structural impact is breaking up the functionality so it can be implemented in a timely fashion. Another example is identifying commonality within the functionality to reduce implementation and maintenance time.
- architecting for a family of related systems. When requirements are being gathered for a collection of similar systems such as in a software product line [Clements 01], the commonalities and variations in those systems may be architecturally significant.
- choice of technologies. There may be a requirement to use a particular technology for a system, such as .NET or J2EE (Java 2 Enterprise Edition). These requirements may be architecturally significant, since many other decisions will be constrained by the use of particular technologies.
- deployment and operations. Requirements that describe an anticipated deployment strategy and how the system will be operated may be architecturally significant, since structural entities may be required to support deployment or operational considerations.

3 Evaluation Criteria

To evaluate and compare various methods for requirements elicitation and specification, we used the set of criteria described in Sections 3.1-3.3. These criteria are divided into the following categories:

- criteria related to how the method fits into the development process
- criteria related to the artifacts produced by the method
- criteria related to how easy the method is to use

When choosing these criteria, we took into account the types of requirements (identified in Section 2) that might be architecturally significant.

3.1 Criteria Related to How the Method Fits into the Development Process

3.1.1 Support for Discovering Architecturally Significant Requirements

Discovering how the various methods support the discovery of architecturally significant requirements is the goal of this report. Does the method provide explicit support for discovering architecturally significant requirements?

3.1.2 Derivation from the Business and Mission Goals for the System

Requirements represent business goals. The method for elicitation should help ensure that the architecturally significant requirements provide support for the business and mission goals.

3.1.3 Checking for Consistency

Given the number of requirements for even medium-sized systems, there are bound to be inconsistencies. Does the method's formalism for the expression of requirements provide support for checking the consistency of the architecturally significant requirements?

3.1.4 Support for Testing

When the system is constructed, it must be tested to determine whether requirements are satisfied. Are the architecturally significant requirements that are generated by the method well suited for the testing process? Can they act as test specifications?

3.2 Criteria Related to the Artifacts Produced by the Method

3.2.1 Expressiveness

Expressing requirements—especially those that are architecturally significant—requires a range of vocabulary. For example, a performance requirement may be expressed in terms of latency and throughput, a modifiability requirement may be expressed in terms of time to make a change, an availability requirement may be expressed in terms of percentage of up-time, and so forth. Furthermore, requirements may vary depending on the context of the system at a specific time. For example, the system may attempt to recover for the first three failures but then shut down after the fourth.

Because a range of vocabulary is required, the method's form for expressing requirements must be flexible. Total flexibility, however, provides no guidance to the requirements analyst. Without this guidance, it becomes too easy to specify requirements that are vague and untestable, such as "The system shall be modifiable." There must be a specified form of expressing requirements that includes the testing criteria.

In summary, the method's form for expressing requirements should be sufficiently expressive to allow for any quality attribute requirement that might be architecturally significant, should allow for the specification of context, and should discourage vague requirements.

3.2.2 Ease of Organizing Architecturally Significant Requirements

Within an enumeration of requirements, those that are architecturally significant should be easy for the architect to find. If, for example, the architect wants to focus on performance and modifiability requirements, the method should make them easy to find. Those requirements that have both modifiability and performance aspects may also be the focus of attention.

The method's formalism for the expression of requirements should facilitate the collection of requirements into categories. In addition, a particular requirement should be able to exist in multiple categories simultaneously. In other words, the form of expressing the requirements should facilitate searching for a variety of different criteria and organizing the requirements based on a variety of different criteria.

3.2.3 Support for Variability

To what extent does the method provide support for eliciting and expressing the variability requirements for a collection of systems? Frequently, requirements are specified for a collection of systems—either variants of the same system or incrementally delivered versions of a single system. Does the method or its expression formalism provide support for capturing and expressing variability?

3.3 Criteria Related to the Method's Ease of Use

3.3.1 Skill Level Necessary to Carry Out the Method

What special skills should those carrying out the method possess? What type of special training must they have to perform the method effectively?

3.3.2 Tooling

What tool support exists for the method? Are there commercial versions of the tool support? How specific is the tool for the method? Does the tool provide support for ease of organizing, traceability, prioritizing requirements, and checking for inconsistencies?

The availability of tool support may not necessarily be an indicator of the method's effectiveness; rather, the level of tool support may be due to the popularity of the method or the marketing prowess of the tool vendors. However, if specialized tool support is available, the method or formalism may be easier to adopt.

4 Evaluation of Methods for Capturing Requirements

In this section, we briefly describe and evaluate five different methods for eliciting and specifying requirements:

1. requirements specification using natural language
2. use case analysis
3. the QAW
4. global analysis
5. O'Brien's approach

First, we first describe each method and identify its artifact and process aspects. Then we provide an evaluation of the method based on the criteria described in Section 3.

4.1 Requirements Specification Using Natural Language

Requirements are frequently specified using natural language. This method is especially common in the U.S. Department of Defense (DoD). A primary benefit of using natural language to specify requirements is that they are understandable by a variety of stakeholders.

4.1.1 Artifact(s) Produced

Natural language requirements have been characterized as "contract-style requirements lists" [Wikipedia 06]. A strength is that they provide a checklist of requirements. Some weaknesses are that the list can run to hundreds of pages, the requirements are hard to check for completeness and consistency, and the requirements may not be described in a useful form because the expression is so free-form.

Most descriptions of natural language requirements provide guidelines regarding their use. For example, "Write complete sentences that have proper grammar, spelling, and punctuation. Keep sentences and paragraphs short and direct" [Wiegers 03].

Other guidelines specify how to organize the requirements and the actual form of the requirements. For example, in some specifications, "shall" is used to indicate what is required of the system to be developed, while "will" is used to indicate what is expected of the system's environment.

The general form of these guidelines is shown below:

<entity> ***shall*** (or *will*) <textual description describing specific requirement>

where <entity> may be the system or subsystem, the software, a quality attribute, the contractor, the development activity (e.g., testing or configuration management), some condition, another actor, or an element of one of these.

Other guidelines involve labeling the requirements and using categories such as "performance" to help organize the requirements. Guidelines also exist for numbering the requirements in a hierarchical fashion to identify different levels of detail for the requirements.

4.1.2 Process

This form of requirements specification is the traditional means for specifying system and software requirements in the DoD. In this environment, the specification serves as a communication vehicle between the DoD agency acquiring the system and the organization building the software under contract.

DoD standards mandate that the requirements be specified in a document called a computer software configuration item (CSCI) with a fixed table of contents including the following sections and subsections:

- applicable documents
- engineering requirements
  - external interface requirements
  - capability requirements
  - internal interfaces
  - data element requirements
  - adaptation requirements
  - sizing and timing requirements
  - safety requirements
  - security requirements
  - design constraints
  - software quality factors
  - human performance/human engineering requirements
  - requirements traceability
- qualification requirements
- preparation for delivery

- notes
- appendices

4.1.3 Evaluation

Table 1 summarizes the results of our evaluation of the natural language method using the criteria described in Section 3.

Table 1: Evaluation of the Natural Language Method

Criteria	Evaluation of Method
Criteria related to how the method fits into the development process	
Support for discovering architecturally significant requirements	The organization of the CSCI isolates some of the types of requirements that may be architecturally significant. Performance, safety, and security requirements are identified separately. Other types of architecturally significant requirements are not identified explicitly.
Derivation from the business and mission goals for the system	The method provides no support for deriving architecturally significant requirements from business and mission goals.
Checking for consistency	Requirements statements using natural language are prone to • be poorly structured, which leads to confusion and misinterpretation • include ambiguities, inaccuracies, and inconsistencies [Wilson 98] • use words that have dual meanings and lack context for proper interpretation • be written at varying levels of granularity across system elements • use a large number of “shalls” and “wills,” which makes reading difficult • reflect an unstructured view of the desired operation of the system These characteristics make checking for consistency painstaking and problematic.
Support for testing	Although ensuring requirements testability is a recommended practice [IEEE 98], there is no inherent mechanism for ensuring that the requirements are testable. Many natural language requirements are not crafted carefully enough to be testable. As a result, it is common practice to establish a team to work with the testers to determine what really needs to be tested.

Table 1: Evaluation of the Natural Language Method (cont.)

Criteria	Evaluation of Method
Criteria related to the artifacts produced by the method	
Expressiveness	Natural language requirements are expressive enough to completely specify architecturally significant requirements with an arbitrary vocabulary. Natural language does not discourage requirements that are too vague, since it does not provide any guidance regarding the content of the requirements' textual descriptions.
Ease of organizing architecturally significant requirements	Because the requirements can be arbitrarily annotated, categories of requirements could be labeled, which would simplify organizing the architecturally significant requirements. Therefore, while this method provides no support for organization, it does not hinder it either, particularly the organization of requirements for particular purposes.
Support for variability	Natural language specification does not provide support for variability.
Criteria related to the method's ease of use	
Skill level necessary to carry out the method	Although no special skill is required to write a natural language requirement, extraordinary skill is required to make the requirements statement meaningful and the collection of statements comprehensive. Establishing complete and correct requirements is extremely difficult and still more of an art than a science. The individuals involved should include subject matter experts, requirements engineers, and key stakeholders of the system. An important aspect is establishing a core requirements team with an expert facilitator as the team leader.
Tooling	A variety of tools exist to support natural language requirements. They typically have features supporting traceability, organization, and modification of the requirements.

4.2 Use Case Analysis

The building blocks of use case analysis are actors, use cases, and use case scenarios. The output of the method is a model. A use case model is the complete set of diagrams, descriptions of actors, use cases, use case scenarios, and other supplemental information needed to represent the use cases. Since use case analysis primarily involves organizing the way that external entities interact with the system, it tends to give high priority to eliciting a system's functional requirements. Other requirements are often captured with supplemental documentation, for which use case analysis does not provide much guidance.

4.2.1 Artifact(s) Produced

The starting point for applying use case analysis is defining the system boundary and the actors that interact with the system. An actor is an external entity that interacts with the system for the purpose of achieving a goal [Cockburn 00]. Actors can be of different types. An actor may initiate a behavior or act as an external server, receiver, and facilitator of behavior [Armour 00].

Identifying the actors of the system lends itself to the identification of use cases that are of value to the actors. A use case is a description of sequences of actions between an actor and the system that result in value to the actor. A description of a use case often follows an agreed-upon template and, at a minimum, should specify the actor involved, the flow of events, the preconditions that should occur before the execution of the use case, and the postconditions that should occur after the execution of the use case [Armour 00, Leffingwell 00]. The scope of a given use case is a single goal that the actor tries to achieve. For example, in an online course management system, “submit assignment” and “check grade” would be examples of use cases.

A use case, in its sequences of actions, describes abstract and general behavior, but it does not describe what happens when a specific actor performs a specific action with specific values. Capturing the specific executions of a use case is referred to as a use case scenario or, sometimes, as a use case instance. For example, in the online course management system, instructors and teaching assistants may have different controls as actors. Instantiating the “edit student properties” use case may limit the properties to which a teaching assistant has access, as opposed to an instructor. Such variations would be revealed when generating scenarios during the use case analysis. Requirements are captured within the sequences of actions and scenarios.

The collection of use cases and actors is often represented by a Unified Modeling Language (UML) use case diagram, but such a diagram is not always necessary. Within a use case diagram, further structuring between use cases (such as generalization, include, and extend relationships) can also be represented. A generalization is similar to a parent/child relationship in the object-oriented programming context. Child, or specific, use cases share structural behavior with the general use case. The latter is referred to as an abstract use case. The “includes” relationship is used when common behavior within use cases can be factored out. The “extends” relationship is used when one use case adds functionality to another. Figure 1 shows a partial use case diagram for the online course management system to illustrate these relationships.

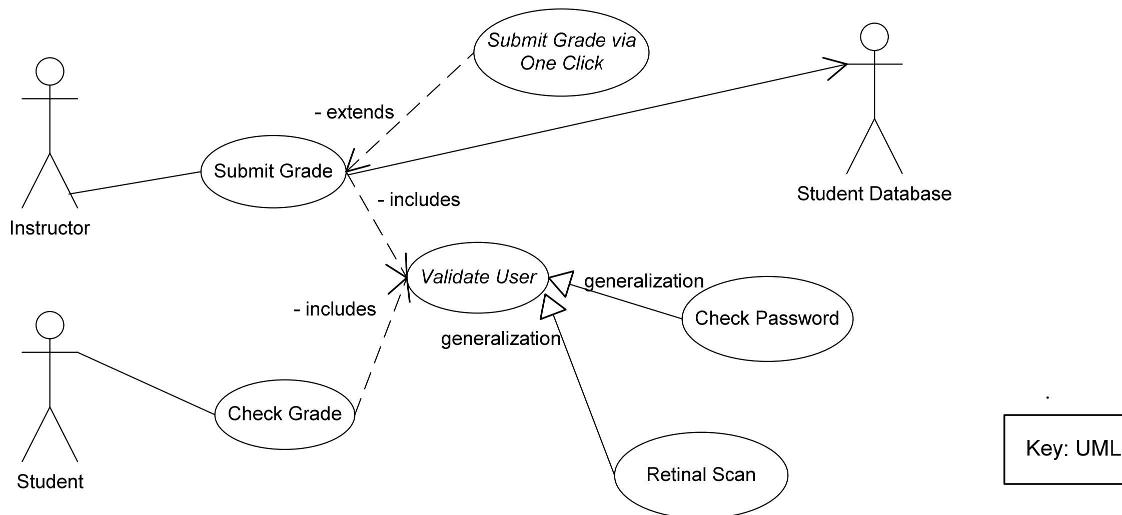

Figure 1: Partial Use Case Diagram for Online Course Management System

The relationship mechanisms allow the further organization of requirements as part of a use case analysis. In Figure 1, since both “check grade” and “submit grade” require validating the user, “validate user” is factored out with an “includes” relationship. For the purpose of demonstration, we assume that the system can validate the user either by checking for a password or performing a retinal scan, which share some commonality and are, therefore, modeled as generalization relationships from the “validate user” use case. Assume that the instructor may submit a grade either by going through the user interface and specifying each student one by one or by submitting grades via a specially formatted file. In Figure 1, submitting a grade with a given file is depicted as extending the initial behavior of submitting a grade. In this example, the “validate user” and “submit grade via one click” use cases are not directly instantiated by actors; therefore, they are abstract use cases.

4.2.2 Process

Use case analysis is used both inside and outside of the Rational Unified Process (RUP). There isn’t one overarching process for conducting a use case analysis. The process may start with conducting a domain analysis, identifying the actors, finding the use cases that the actors will conduct, and then describing each use case with sequences of actions and scenarios. Alternatively, a bottom-up approach may be used by first elaborating the scenarios. Process variations may exist depending on the overall software development process selected. Jacobsen, Booch, and Rumbaugh provide a detailed analysis of conducting use case analysis in the RUP [Jacobsen 99].

Berenbach describes an approach using a use case model that shows high-level product features in the context of a model-driven requirements engineering process [Berenbach 04]. A broad and shallow approach is taken when creating such a model; rather than elaborating on the features, all desired features of the product under development are captured first, forming

a feature specification for the product. To obtain a comprehensive set of features, a system context diagram that shows all system actors (external entities that interact with the system) is used to capture features from the perspective of each individual actor.

Use case analysis begins by first prioritizing the product features based on release planning. Initial releases of the product give higher priority to high-risk and architecturally significant product features.

Each feature for a given release is elaborated into concrete use cases, each of which represents the system functions required to implement the given feature. Each system function is further described using scenarios of how the system will be used by its actors. These functions are then used to generate a requirements specification for the system. This specification is only for the given release, and subsequent iterations will add requirements for future releases.

4.2.3 Evaluation

Table 2 summarizes the results of our evaluation of the use case analysis method using the criteria described in Section 3.

Table 2: Evaluation of the Use Case Analysis Method

Criteria	Evaluation of Method
Criteria related to how the method fits into the development process	
Support for discovering architecturally significant requirements	If a use case scenario is annotated as a type of requirement that could be architecturally significant, the annotations could be searched to discover the particular types of requirements. However, use cases are not typically annotated in this way.
Derivation from the business and mission goals for the system	Berenbach's method supports deriving variability from the business goals, but it does not support any other type of derivation. Other methods do not support any form of derivation.
Checking for consistency	Use case modeling is a scenario-based technique. Knowing when to stop generating scenarios and how to achieve consistency is one of the frequently discussed challenges in use case modeling [Cockburn 00]. Similarly, achieving consistency becomes challenging because, as the number of scenarios increases, variations create opportunities for inconsistencies that are hard to track.
Support for testing	One of the advantages of use case scenarios is that they can be used for functional testing and generating test cases.

Table 2: Evaluation of the Use Case Analysis Method (cont.)

<table>
<tr><th>Criteria</th><th>Evaluation of Method</th></tr>
<tr><td colspan="2">Criteria related to the artifacts produced by the method</td></tr>
<tr><td>Expressiveness</td><td>Each use case is described using sequences of events and scenarios that capture the interaction of the actor with the system. When describing a scenario, the quality attributes and other requirements that may be architecturally significant could be captured as special requirements [Larman 05]. Since use cases or groups of use cases map to system-level functions, using this technique to capture architecturally significant requirements may be difficult [Garlan 00, Kazman 04]. Consider the following example:

An improved, commercially available discrete event generator is available for the system, and the system permits engineers to remove the old discrete event generator and incorporate the new one in less than two person-weeks.

While it is possible to express this requirement as a desirable use case of the discrete event generator, elaborating the requirement into sequences of actions and their corresponding special requirements may be awkward. Moreover, use cases best capture runtime behavior as requirements, whereas this example is clearly a design-time requirement.

Even though some architecturally significant requirements could be expressed as use cases, in practice, they are not generally included. They may also be expressed as annotations to a use case diagram, but then they would be expressed using natural language.</td></tr>
<tr><td>Ease of organizing architecturally significant requirements</td><td>Architecturally significant requirements are not an explicitly identified portion of use cases or scenarios. They may appear as fragments within the sequences of actions in describing a particular step (e.g., “The online management course system validates the password that the user enters in 0.1 seconds”). However, architecturally significant requirements do not normally appear in use case scenarios. If they do, they are not easily accessible.</td></tr>
<tr><td>Support for variability</td><td>There are examples of using use cases to capture product line requirements where variability is critical [Gooma 05]. Although there are no tools available to support specifying variability requirements in a use case analysis, variability can be captured using scenarios, “extend” relationships, and techniques for identifying generalization relationships.</td></tr>
</table>

Table 2: Evaluation of the Use Case Analysis Method (cont.)

Criteria	Evaluation of Method
Criteria related to the method's ease of use	
Skill level necessary to carry out the method	Use case analysis is ideally performed by a requirements engineer during a requirements workshop with key system stakeholders (including subject matter experts and architects). The requirements engineer is an expert facilitator and elicits requirements, trying to identify special requirements for the use cases being modeled. The skills required to conduct the method effectively range from understanding object-oriented development and UML to having enough experience to know when further scenario elicitation will no longer add value and new requirements.
Tooling	Use cases can be diagramed in UML, and there are many UML modeling tools available on the market. These tools help in generating use case diagrams and interaction diagrams (such as sequence and collaboration) where the interactions can be modeled. Generating these diagrams alone does not create a use case model. Word processors support combining these diagrams with supplemental descriptions.

4.3 The SEI Quality Attribute Workshop

The Quality Attribute Workshop (QAW) is a method for eliciting quality attribute requirements [Barbacci 03]. The output of the method is a prioritized set of quality attribute requirements that could be architecturally significant. The QAW is based on two main premises:

1. Concrete quality attribute requirements can be described in the form of quality attribute scenarios.
2. Stakeholders are the best sources of the different perspectives that are manifested in quality attribute requirements.

The QAW does not help to elicit architecturally significant requirements that are not quality attribute requirements.

4.3.1 Artifact(s) Produced

Quality attribute goals, by themselves, are not definitive enough for either design or evaluation. They must be made more concrete. Using modifiability as an example, if a system can be adapted easily to have different user interfaces but is dependent on a particular operating system, is it modifiable? The answer depends on which modifications to the system are expected over its lifetime. That is, the abstract quality goal of modifiability must be made concrete.

System-specific scenarios can be used to describe more clearly the quality attributes that are important to the system and to identify the desired quality attribute responses. Scenarios are short stories that describe an interaction with the system that exercises a particular quality attribute. For example, a scenario makes the modifiability requirement above more explicit as follows:

> An improved, commercially available discrete event generator is available for the system, and the system permits engineers to remove the old discrete event generator and incorporate the new one in less than two person-weeks.

To be useful, a scenario must, at the very least, have a clear stimulus and response. The stimulus is the part of the scenario that describes an agent or factor that causes the system to react. In the example above, the stimulus is "An improved, commercially available discrete event generator is available for the system." The response is the system's reaction to the stimulus. In the example above, the response is "…the system permits engineers to remove the old discrete event generator and incorporate the new one in less than two person-weeks."

An important part of the response is a clear response measure. Simply stating "Modify the system to incorporate the new discrete event generator" would not describe how well the architecture accommodated the modification. Given enough time and money, any modification is possible. However, in this scenario, a response measure of "two person-weeks" is imposed, forcing the architect to ensure that the system is modifiable with respect to very specific and measurable criteria.

One proposal, although not in the current QAW method, is to use a utility tree [Clements 02] to organize the scenarios that are derived during a QAW.

4.3.2 Process

A computer system is intended to support a myriad of business goals emanating from the different perspectives that various stakeholders have for the system. The business manager may view a system as a means of consolidating or expanding particular markets. The operating manager may view a system as a means of making a particular process more efficient. The maintenance team may view a system as something new to be understood. Perspectives such as these are rarely written down explicitly, and yet they could potentially have an impact on the system design.

The QAW method is designed to identify the concerns of the different stakeholders. The more distinct perspectives that are expressed, the more likely it is that the concerns implicit in the perspectives will be addressed.

The QAW Method

A QAW is a one- or two-day facilitated meeting of the stakeholders of a particular system. The output of the QAW is a set of prioritized quality attribute scenarios and associated information.

The QAW involves the following steps.

Step 1: QAW Presentation and Introductions

In this step, the QAW facilitator describes the motivation for the QAW and explains each step of the method. The purpose of this step is to ensure that all attendees understand the QAW process and meet all of the other attendees. It is also an opportunity for the facilitator to get to know the stakeholders in attendance.

Step 2: Business/Mission Presentation

After Step 1, a representative of the stakeholder community presents the business and mission drivers for the system. The stakeholder representing the business and mission concerns (typically a manager or management representative) spends about one hour presenting

- the system's business/mission context
- high-level functional requirements, constraints, and quality attribute requirements

The purpose of this step is for the facilitator to understand the context for the system under discussion and to begin creating a utility tree of the important quality attributes.

Step 3: Architectural Plan Presentation

At this point in the workshop, a technical stakeholder presents the system architectural plans as they are currently known. Information in this presentation may include

- plans and strategies for how key business/mission requirements will be satisfied
- key technical requirements and constraints—such as mandated operating systems, hardware, middleware, and standards—that will drive architectural decisions
- existing context diagrams, high-level system diagrams, and other written descriptions

The purpose of this step is twofold:

1. The attendees should understand what technical decisions have been made already.
2. The facilitators will use this information to produce the list of architectural drivers in Step 4.

Step 4: Identification of Architectural Drivers

The facilitator enumerates a list of key architectural drivers and asks the stakeholders for clarifications, additions, deletions, and corrections. This step requires architectural skill on the part of the facilitators. The purpose of this step is to provide some focus for the scenario brainstorming in Step 5.

Step 5: Scenario Brainstorming

Stakeholders generate scenarios through a brainstorming process. Each stakeholder expresses a scenario representing his or her concerns with respect to the system in round-robin fashion. During a nominal QAW, at least two round-robin passes are made so that each stakeholder can contribute at least two scenarios. The facilitator ensures that at least one representative scenario exists for each architectural driver listed in Step 4.

Facilitators must remember that there are three general types of scenarios and ensure that each one is covered during the QAW. The three types of scenarios are

1. use case scenarios, which involve anticipated uses of the system
2. growth scenarios, which involve anticipated changes to the system
3. exploratory scenarios, which involve unanticipated stresses to the system that can include uses or changes

The purpose of this step is to express the stakeholders' concerns. These concerns are cross referenced to architectural drivers to ensure that requirements important for architectural design are included in the final list of scenarios.

Step 6: Scenario Consolidation

After scenario brainstorming, similar scenarios are consolidated when reasonable. To consolidate the scenarios, the facilitator asks stakeholders to identify those scenarios that are very similar in content. These scenarios are merged, as long as the people who proposed them agree to the merging and feel that their scenarios will not be diluted in the process. The purpose of this step is to prevent a "dilution" of votes during the prioritization of scenarios.

Step 7: Scenario Prioritization

The scenarios are prioritized through a voting scheme that treats all of the stakeholders as equals. The purpose of this step is to determine which scenarios are viewed as important by the stakeholders.

Step 8: Scenario Refinement

After the prioritization, the top four or five scenarios are refined in more detail. The facilitator further elaborates each top scenario and documents the results of the following activities:

- Further clarify the scenario by clearly describing the following:
  - stimulus: the condition that affects the system
  - response: the activity that results from the stimulus
  - source of stimulus: the entity that generated the stimulus
  - environment: the condition under which the stimulus occurred
  - artifact stimulated: the artifact that was stimulated
  - response measure: the measure by which the system's response will be evaluated
- Describe the business/mission goals that are affected by the scenario.
- Describe the relevant quality attributes associated with the scenario.
- Allow the stakeholders to pose questions and raise any concerns regarding the scenario.

4.3.3 Evaluation

Table 3 summarizes the results of our evaluation of the QAW method using the criteria described in Section 3.

Table 3: Evaluation of the QAW Method

Criteria	**Evaluation of Method**
Criteria related to how the method fits into the development process	
Support for discovering architecturally significant requirements	The QAW provides support for discovering architecturally significant requirements that are also quality attribute requirements. It has a process for prioritizing quality attribute requirements.
Derivation from the business and mission goals for the system	The QAW maintains traceability back to the business and mission goals, but it does not have a process to derive architecturally significant requirements based on these goals.
Checking for consistency	The QAW checks the scenarios generated against the business goals and against the architectural drivers. There is no check for consistency of functionality or against other potential architecturally significant requirements.
Support for testing	The quality attribute scenarios generated by the QAW are suitable for use as test cases.

Table 3: Evaluation of the QAW Method (cont.)

Criteria	Evaluation of Method
Criteria related to the artifacts produced by the method	
Expressiveness	Quality attribute scenarios allow the use of any vocabulary in expressing a quality attribute requirement and ensure that the requirement is explicit in both the desired response and the measure used for that response. The terms *stimulus* and *response* may cause confusion when applied to some quality attributes such as modifiability. When using quality attribute scenarios, the quality attribute requirements are expressed at a concrete level. There are no means for expressing requirements at a more abstract level.
Ease of organizing architecturally significant requirements	For the quality attribute scenarios that are refined, a link to the relevant quality attributes is maintained. For those that are not refined, no such link is maintained within the method. The linking of the scenarios to the relevant quality attribute provides one means of organizing those architecturally significant requirements that are also quality attribute requirements. Another means of organizing the quality attribute requirements is through the response measure. All requirements that have the same type of response measure can be grouped together easily. (For example, group all requirements that have "time to change" as a response measure.) Other types of organization methods (e.g., identifying the quality attribute requirements associated with a particular piece of functionality) are not supported by the QAW method or by the use of quality attribute scenarios.
Support for variability	The quality attribute scenarios can be used to express variability requirements. (For example, a requirement stating that "a new product incorporating features x, y, and z can be created from the core assets within three person-months" fits into the quality attribute scenario template.)
Criteria related to the method's ease of use	
Skill level necessary to carry out the method	In order to identify the architectural drivers, the facilitator must have architectural skills in addition to facilitation skills.
Tooling	No specialized tool support is available for use within a QAW.

4.4 Global Analysis

The purpose of the global analysis method is to identify and analyze the factors that have a global influence on the architecture [Hofmeister 00]. These factors fall into three categories: technological, organizational, and product. The target hardware and software platforms, standards to be followed, and development frameworks to be used are examples of technological factors. Organizational factors include the project's schedule and budget, software process in use, and developers' skill set. Product factors are the requirements specific to the product being developed and include functional features as well as quality attribute requirements.

4.4.1 Artifact(s) Produced

The artifacts produced by a global analysis are

- factor tables: There is one factor table for each category of factors. A factor table has three columns: (1) a description of the factor, (2) the aspects of the factor that can change, and (3) a description of the impact that the factor or the changes to the factor have on the architecture.
- issue cards: Issues are identified during the analysis and recorded on issue cards. Each card contains the description of a design issue that results from factors and their changeability. A common example of an issue is that it may not be possible to implement all requirements on schedule. Another example is that changes in the target runtime platform may affect the ability of the system components to function. The issue cards also contain a list of influencing factors and, most importantly, a discussion of strategies to address the issue.

4.4.2 Process

Global analysis complements risk analysis and requirements analysis, which can be performed using other techniques. It consists of two phases: analyze factors and develop strategies. Each phase has three steps: (1) identify and describe factors, (2) characterize the changeability and flexibility of factors, and (3) analyze the impact of factors. The global analysis steps and their artifacts are revisited as needed as the design activities proceed.

4.4.3 Evaluation

Table 4 summarizes the results of our evaluation of the global analysis method using the criteria described in Section 3.

Table 4: Evaluation of the Global Analysis Method

Criteria	Evaluation of Method
Criteria related to how the method fits into the development process	
Support for discovering architecturally significant requirements	Global analysis provides tables of common organizational, technological, and product features. These tables are a starting point for discovering architecturally significant requirements.
Derivation from the business and mission goals for the system	Global analysis provides no assistance for deriving architecturally significant requirements from the business goals.
Checking for consistency	The global analysis method does not prescribe consistency checks. Dividing the influencing factors into three groups and suggesting several categories within each group may help architects cover all influencing factors, but no verification step is enforced.
Support for testing	The factor tables collect the influencing factors in an organized way. Although factor tables do not provide enough detail to map directly to test cases, they can help to define tests and validations, not only for functional features, but also for other types of requirements. Because the factor tables describe what can change for a given factor, they also provide insight into exploratory test conditions. Paulish has suggested that issue cards produced in global analysis provide valuable input to test planning [Paulish 01]. The issue cards identify the key issues related to functionality, so that test planners can concentrate the testing effort on those issues.
Criteria related to the artifacts produced by the method	
Expressiveness	Global analysis recognizes the importance of architecturally significant requirements, which are categorized and listed in the factor tables. However, the method does not prescribe structure or well-formed rules for specifying factors.
Ease of organizing architecturally significant requirements	The way that global analysis organizes the architecturally significant requirements makes it easy to browse and find information. The requirements (factors) are distributed in three groups: technological factors, organizational factors, and product factors. Within each group, the architect should list the factors separated into categories. Typical categories of product factors are, for example, functional features, user interface, performance, and dependability.
Support for variability	Although factor tables describe the intended or forecasted changeability for each factor, global analysis does not include specific provisions for handling multiple systems simultaneously.

Table 4: Evaluation of the Global Analysis Method (cont.)

Criteria	Evaluation of Method
Criteria related to the method's ease of use	
Skill level necessary to carry out the method	The team or individual performing the global analysis method should (1) have good knowledge of the organization and details of the project resources and constraints, so that the team or individual is able to elicit organizational factors; (2) be conversant in the technology and standards involved, so that the team or individual is able to explore the technological factors; and (3) have good design skills and quality attribute knowledge to analyze the product factors.
Tooling	No tool specialized for global analysis is publicly available. Text editors could be adapted through templates to support global analysis.

4.5 O'Brien's Approach

Fergus O'Brien has outlined an approach for strongly linking architectural decisions to measurable quality attributes that become apparent when business goals are captured explicitly [O'Brien 04].

4.5.1 Artifact(s) Produced

This approach is intended to cover the whole life cycle—from business case to deployment and decommissioning of the system. In the requirements phase, six artifacts are identified, although the forms and details of these artifacts are not provided. The six artifacts are

1. business case: a justification for the system in terms understandable to the board of directors
2. functional requirements: a specification of the functional requirements of the system
3. quality attribute requirements: a specification of the quality attribute requirements of the system
4. use cases: a use case model, as described earlier, for the functional aspects of the system
5. service-level agreement: a description of the level of service (in quality attribute terms) that is to be expected from the system. This agreement is human readable and is intended to describe the various stakeholders' understanding of the system being constructed.
6. executable service-level agreement: a machine-executable agreement in which the elements of the agreement can be determined in real time by examining the executing system

4.5.2 Process

The basic feature of this approach is that a business case for a system is captured, which leads to the identification of functional and quality attribute requirements. Both types of requirements inform the architecture of the system and are monitored for compliance throughout the development and production life cycle.

The key characteristics of this approach, compared to more traditional life-cycle models, include the following:

- explicit capture of a business case for the system. The purpose of this business case is to inform technical decisions, not just financial ones. Hence, its audience includes the system's architect(s).
- explicit derivation of quality attribute requirements from the business case
- careful crafting of measurable quantities (which O'Brien calls *metrics*) for each of the quality attributes, instead of vague descriptions that are difficult to test
- "derivation" of the architecture from the requirements, primarily the quality attribute requirements. O'Brien doesn't mean automated derivation but rather a principled design approach for systematically examining the requirements and applying suitable design techniques (which we would call *tactics* [Bass 03, Ch. 5]) to develop the architecture. This process is similar in spirit to the SEI Attribute-Driven Design method [Bass 03, Ch. 7].

4.5.3 Evaluation

Table 5 summarizes the results of our evaluation of O'Brien's approach using the criteria described in Section 3.

Table 5: Evaluation of O'Brien's Approach

Criteria	**Evaluation of Method**
Criteria related to how the method fits into the development process	
Support for discovering architecturally significant requirements	The method describes interacting with the board of directors to discover dominant quality attribute requirements; these requirements are then regarded as the architecturally significant ones.
Derivation from the business and mission goals for the system	The interaction with the board members is intended to identify their business goals for a system, and these goals are then translated into quality attribute requirements. This process is described as very time and labor intensive.

Table 5: Evaluation of O'Brien's Approach (cont.)

Criteria	Evaluation of Method
Criteria related to how the method fits into the development process	
Checking for consistency	The best example of a consistency rule in O'Brien's approach is the requirement that each quality attribute should be accompanied by a metric that defines it quantitatively.
Support for testing	Monitoring the coverage of functional and quality attribute requirements is an integral part of this approach and can be expected to make testing more straightforward. This monitoring constitutes indirect support for testing by making the resulting system easier to test. O'Brien suggests using the cleanroom approach to generate test cases for the use cases. Cleanroom statistical testing covers functionality, but it is also the starting point for monitoring nonfunctional requirements. O'Brien's approach advocates the use of an automated monitoring framework to ensure that the system meets performance, reliability, and maintainability requirements. The monitoring framework is implemented separately and used continuously during development and production.
Criteria related to the artifacts produced by the method	
Expressiveness	Any quality attribute that can be measured can be expressed. Although use cases are mentioned as a way to capture functional requirements, no particular expressive form is given for quality attribute requirements. We assume that quality attribute scenarios, or even more formal languages, could be used if desired. The assumption is that all architecturally significant requirements are quality attribute requirements, so there is no discussion of other architecturally significant requirements.
Ease of organizing architecturally significant requirements	The method does not prescribe how quality attribute requirements are to be organized or formatted. However, the intent is to examine the requirements systematically and explicitly, so it is reasonable to assume that they should be cataloged in one place, arranged by quality attribute name, and easily searched and used.
Support for variability	No support for variability was explicitly mentioned.
Criteria related to the method's ease of use	
Skill level necessary to carry out the method	The approach involves taking the type of business case that exists in boardrooms and translating that business case into quality attribute requirements. Doing this requires a specialized skill. Once the quality attribute requirements are identified, the remainder of the approach can be performed by a skilled architect or developer.
Tooling	No tools specific to the method were mentioned.

5 Related Work

Work related to this study generally falls into two categories: (1) work associated with the relationship between requirements and architecture or (2) work associated with representing requirements that are usually architecturally significant. An example of the latter category is the paper by Paech and colleagues, who give an elegant argument for intertwining functional and nonfunctional (we would say "quality attribute") requirements with architecture; in fact, they build a compelling case claiming that the failure to connect requirements with architecture resulted in the infamous Ariane 5 disaster [Paech 03].

Work that has attempted to constructively connect requirements with architecture includes the 2001 and 2003 workshops titled *From Software Requirements to Architectures* (STRAW), which were held at the International Conference on Software Engineering (ICSE) [Castro 01, Berry 03].

Connecting requirements to architecture can be viewed as a special case of connecting a system's problem space and its solution space. When the problem is viewed in that light, other work becomes relevant.

Lucia Rapanotti and colleagues [Rapanotti 04] have drawn a bridge between problem space and solution space by extending Michael Jackson's concept of problem frames [Jackson 01]. Rapanotti's approach is to create *architectural frames* (Aframes), which are styles that provide solutions to problems that are members of a corresponding problem frame. For example, a (linear) Pipe-and-Filter Transformation Aframe represents the class of transformation problems whose solution is to be provided through the Pipe-and-Filter architectural style. Similarly, Nuseibeh has also emphasized developing requirements and architectures simultaneously by using problem frames [Nuseibeh 01]. In this work, he suggests a process model called Twin Peaks. In this model, the goal is to create a relationship between requirements, architecture, and design at a pattern level. For example, rigid requirements, manifested by problem frames, can limit the candidate architecture and design choices, manifested by architectural styles and design patterns.

These approaches are somewhat reminiscent of the work in attribute-based architectural styles (ABASs) by Klein and colleagues [Klein 99]. ABASs are architectural styles that are carefully annotated to denote the quality attributes they provide, thus offering another bridge between problem and solution spaces.

Some people have worked to solve a specific instance of the requirement-to-architecture problem by working in specific problem areas or specific solution areas. Corradini and colleagues have provided a way to relate state-based software architecture specifications to

high-level functional specifications [Corradini 06]. Lapouchnian and colleagues use goal-oriented requirements models to formally generate feature, statechart, and component-connector models of a system's architecture [Lapouchnian 05]. Sparkman and colleagues have described a method to produce multi-agent architectures from a specification language that is intended to be used for expressing agent requirements [Sparkman 02]. Grünbacher and colleagues have created intermediate, architecture-aware requirements models by applying a taxonomy of architectural dimensions to requirements [Grünbacher 04]. This process involves extracting the component, bus (connector), and system, as well as their respective properties from requirement statements. Finally, Stephenson and McDermid have described a technique for assessing individual requirements for issues that may indicate uncertainty on the part of the customer [Stephenson 05]. This assessment includes steps to identify information that would be expected to resolve the issue, the resulting requirements change, and the derived flexibility requirements (architecturally provided) that would guard against the resulting change. The technique provides information that enables a designer to make an immediate architectural decision while waiting for formal confirmation of the customer's intent.

A finer grained approach tying architectural design decisions with qualities in the problem domain is represented by architectural tactics. A tactic is a fine-grained design decision that has a positive impact on some quality attribute. For example, ping/echo is a tactic that might be chosen to help provide fault detection in a system whose requirements include high availability. Deferred binding is a tactic to promote modifiability, and resource management (such as scheduling) is a tactic to increase performance. Quality attributes are specified using quality attribute scenarios [Bass 03].

Baniassad and colleagues discuss a requirements approach that feeds into architecture in an aspect-oriented setting [Baniassad 06].

The FURPS+ model is a categorization scheme for requirements [Grady 92]. It divides requirements into the following categories: functionality, usability, reliability, performance, and supportability. The "+" in the acronym refers to different types of constraints that might be imposed. It does not discriminate among the requirements to identify ones that are architecturally significant.

Some requirement specification techniques aim to improve upon natural language specifications (e.g., planguage [Gilb 05]). Planguage, referring to the combination of planning and language, is a language that provides parameters for structuring natural language specifications instead of just providing a textual description. The goal of this method is to recognize each requirement as a unique instance and provide additional information to allow for specifying the requirement's quality measures, cost, schedule, priority, and other attributes using the parameters as tags. Example parameter tags in planguage are stakeholders, owner, scale, goal, wish, ideal, version, author, status, and design proposal.

Finally, Alan Davis's excellent bibliography on requirements engineering is worth reviewing from time to time to see if any new architecture-related work has been published [Davis 06].

6 Conclusions and Future Work

We identified three problems that motivated our evaluation. In this section, we restate the problems and summarize our conclusions based on the evaluation.

1. **Requirements must be expressed in a form that provides the information necessary for design.**

   The quality attribute scenarios from the QAW explicitly identify the information necessary for design. Designers need to know what environmental assumptions are made in the requirements and must differentiate among the sources of a stimulus. The QAW method is more explicit than the other methods about requiring the different types of information needed by the designer.

2. **The elicitation of the requirements must capture architecturally significant requirements.**

   We identified many different techniques for capturing architecturally significant requirements during requirements elicitation:

   - Global analysis factors provide a list that includes candidates for architecturally significant requirements.
   - The QAW identifies and uses business goals, as determined by the stakeholders, to prioritize quality attribute scenarios.
   - Berenbach's refinement of use case analysis identifies variability.

3. **The business and mission goals must provide systematic input for the elicitation process.**

   O'Brien's approach is the only method that addresses business goals explicitly. The other methods rely on the stakeholders to translate business and mission goals into requirements.

The major unsolved problem is how to link business and mission goals to architecturally significant requirements. O'Brien's approach is time and labor intensive, and it requires the active involvement of an organization's board members. Additional work is needed to determine if there is a more systematic and less labor-intensive process for identifying architecturally significant requirements from business and mission goals.

References

URLs are valid as of the publication date of this document.

[Armour 00] Armour, F. & Miller, G. *Advanced Use Case Modeling: Software Systems*. Boston, MA: Addison-Wesley, 2000.

[Baniassad 06] Baniassad, E.; Clements, P.; Araujo, J.; Moreira, A.; Rashid, A.; & Tekinerdogan, B. "Discovering Early Aspects." *IEEE Software 23*, 1 (January/February 2006): 61-70.

[Barbacci 03] Barbacci, M.; Ellison, R.; Lattanze, A.; Stafford, J.; Weinstock, C.; & Wood, W. *Quality Attribute Workshops (QAWs), Third Edition* (CMU/SEI-2003-TR-016, ADA418428). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports /03tr016.html.

[Bass 03] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*, *Second Edition*. Boston, MA: Addison-Wesley, 2003.

[Berenbach 04] Berenbach, B. "The Evaluation of Large, Complex UML Analysis and Design Models," 232-241. *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004).* Edinburgh, Scotland, UK, May 23-28, 2004. Los Alamitos, CA: IEEE Computer Society, 2004.

[Berry 03] Berry, D.; Kazman, R.; & Wieringa, R. "Second International Workshop on SofTware Requirements to Architectures (STRAW '03)," 797-798. *Proceedings of the 25th International Conference on Software Engineering*. Portland, OR, May 3-10, 2003. Los Alamitos, CA: IEEE Press, 2003.

[Castro 01] Castro, J. & Kramer, J. "From Software Requirements to Architectures" (workshop summary), 764-765. *Proceedings of the 23rd International Conference on Software Engineering*. Toronto, Ontario, Canada, May 12-19, 2001. Los Alamitos, CA: IEEE Computer Society Press, 2001.

[Clements 01] Clements, P. & Northrop, L. *Software Product Lines: Practices and Patterns*. Boston, MA: Addison-Wesley, 2001.

[Clements 02] Clements, P.; Kazman, R.; & Klein, M. *Evaluating Software Architectures: Methods and Case Studies*. Boston, MA: Addison-Wesley, 2002.

[Cockburn 00] Cockburn, A. *Writing Effective Use Cases*. Boston, MA: Addison-Wesley, 2000.

[Corradini 06] Corradini, F.; Inverardi, P.; & Wolf, A. "On Relating Functional Specifications to Architectural Specifications: A Case Study." *Science of Computer Programming 59*, 3 (February 2006): 171-208.

[Davis 06] Davis, Alan. *Requirements Bibliography*. http://web.uccs.edu/adavis/UCCS/reqbib-abcd.htm#A (2006).

[Dominick 02] Dominick, L.; Hilliard, R.; Kahane, E.; Kazman, R.; Kruchten. P.; Kozaczynski, W.; Obbink, H.; Postema, H.; Ran, A.; & Tracz, W. *Software Architecture Review and Assessment (SARA) Report, Version 1.0*. http://philippe.kruchten.com/architecture/SARAv1.pdf (2002).

[Garlan 00] Garlan, D. "Software Architecture and Object-Oriented Systems." *Proceeding of the IPSJ Object-Oriented Symposium 2000*. Tokyo, Japan, August 30-September 1, 2000. Tokyo. Japan: Information Processing Society of Japan, 2000. http://acme.able.cs.cmu.edu/pubs/show.php?id=155.

[Gilb 05] Gilb, Tom. *Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage*. Burlington, MA: Butterworth-Heinemann, 2005.

[Gooma 05] Gooma, H. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Boston, MA: Addison-Wesley, 2005.

[Grady 92] Grady, R. *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

[Grünbacher 04] Grünbacher, Paul; Egyed, Alexander; & Medvidovic, Nenad. "Reconciling Software Requirements and Architectures with Intermediate Models." *Springer Journal of Software and System Modeling 3*, 3 (August 2004): 235–253.

[Hofmeister 00] Hofmeister, C.; Nord, R.; & Soni, D. *Applied Software Architecture*. Boston, MA: Addison-Wesley, 2000.

[IBM 06] IBM. *Rational Method Composer*. http://www-306.ibm.com /software/awdtools/rmc/index.html (2006).

[IEEE 98] Institute of Electrical and Electronics Engineers (IEEE). *Recommended Practice for Software Requirements Specifications* (IEEE Std. 830-1998). New York, NY: IEEE, 1998.

[Jackson 01] Jackson, M. *Problem Frames: Analyzing and Structuring Software Development Problems*. Boston, MA: Addison-Wesley, 2001.

[Jacobsen 99] Jacobsen I.; Booch G.; & Rumbaugh J. *The Unified Software Development Process*. Boston, MA: Addison-Wesley, 1999.

[Kazman 04] Kazman, R.; Kruchten, P.; Nord, R.; & Tomayko, J. *Integrating Software-Architecture-Centric Methods into the Rational Unified Process* (CMU/SEI-2004-TR-011). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents/04.reports /04tr011.html.

[Klein 99] Klein, M.; Kazman, R.; Bass, L.; Carriere, S. J.; Barbacci, M.; & Lipson, H. "Attribute-Based Architectural Styles," 225-243. *Proceedings of the First Working IFIP Conference on Software Architecture*. San Antonio, TX, February 22-24, 1999. Norwell, MA: Kluwer Academic Publishers, 1999.

[Lapouchnian 05] Lapouchnian, A.; Liaskos, S.; Mylopoulos, J.; & Yu, Y. "Towards Requirements-Driven Autonomic Systems Design," 1-7. *Proceedings of the 2005 Workshop on the Design and Evolution of Autonomic Application Software*, International Conference on Software Engineering. St. Louis, MO, May 21, 2005. New York,

NY: Association of Computing Machinery (ACM) Press, 2005.

[Larman 05] Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design, Third Edition*. Upper Saddle River, NJ: Prentice-Hall, 2005.

[Leffingwell 00] Leffingwell, D. & Widrig, D. *Managing Software Requirements: A Unified Approach*. Reading, MA: Addison-Wesley, 2000.

[Nuseibeh 01] Nuseibeh, B. "Weaving Together Requirements and Architectures." *IEEE Computer 34*, 3 (March 2001): 115-119.

[O'Brien 04] O'Brien, F. *The Engineering of Software Quality*. French Forest, New South Wales, Australia: Pearson SprintPrint, 2004.

[Paech 03] Paech, B.; Dutoit, A.; Kerkow, D.; & von Knethen, A. "Functional Requirements, Non-Functional Requirements, and Architecture Should Not Be Separated," 102-107. *Proceedings of the International Workshop on Requirements Engineering*: *Foundations for Software Quality*. Essen, Germany, September 9-10, 2002. Amsterdam: Elsevier B.V., 2003.

[Parnas 72] Parnas, David. "On the Criteria to be Used in Decomposing Systems into Modules." *Communications of the ACM 15*, 12 (December 1972): 1053–1058.

[Paulish 01] Paulish, D. *Architecture-Centric Software Project Management*. Boston, MA: Addison-Wesley, 2001.

[Rapanotti 04] Rapanotti, Lucia; Hall, Jon G.; Jackson, Michael; & Nuseibeh, Bashar. "Architecture-Driven Problem Decomposition," 80-89. *Proceedings of the 12th IEEE International Requirements Engineering Conference (RE'04)*. Kyoto, Japan, September 6-10, 2004. Los Alamitos, CA: IEEE Computer Society Press, 2004.

[Sparkman 02] Sparkman, C.; DeLoach, S.; & Self, A. "Automated Derivation of Complex Agent Architectures from Analysis Specifications," 278-296. *Agent-Oriented Software Engineering II, Second International Workshop (AOSE 2001): Revised Papers and Invited Contributions*. Montreal, Quebec, Canada, May 29, 2001. Lecture Notes in Computer Science, Volume 2222. Berlin, Germany: Springer-Verlag, 2002.

[Stephenson 05] Stephenson, M. & McDermid, J. “Deriving Architectural Flexibility Requirements in Safety-Critical Systems.” *IEEE Proceedings on Software 152*, 4 (August 2005): 143-152.

[Wiegers 03] Wiegers, Karl. *Software Requirements: Practical Techniques for Gathering and Managing Requirements Throughout the Product Development Cycle*. Redmond, WA: Microsoft Press, 2003.

[Wikipedia 06] Wikipedia. *Requirement Analysis*. http://en.wikipedia.org/wiki/Requirements_analysis (2006).

[Wilson 98] Wilson, William; Rosenberg, Linda; & Hyatt, Lawrence. *Automated Quality Analysis of Natural Language Requirement Specifications*. Greenbelt, MD: Software Assurance Technology Center (SATC), NASA Goddard Space Flight Center (GSFC), 1998.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY
(Leave Blank)

2. REPORT DATE
August 2006

3. REPORT TYPE AND DATES COVERED
Final

4. TITLE AND SUBTITLE
A Comparison of Requirements Specification Methods from a Software Architecture Perspective

5. FUNDING NUMBERS
FA8721-05-C-0003

6. AUTHOR(S)
Len Bass, John Bergey, Paul Clements, Paulo Merson, Ipek Ozkaya, Raghvinder Sangwan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-2006-TR-013

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
ESC-TR-2006-013

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified/Unlimited, DTIC, NTIS

12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

One of the key challenges to producing high-quality software architecture is identifying and understanding the software's architecturally significant requirements. These requirements are the ones that have the most far-reaching effect on the architecture. In this report, five methods for the elicitation and expression of requirements are evaluated with respect to their ability to capture architecturally significant requirements. The methods evaluated are requirements specification using natural language, use case analysis, the Quality Attribute Workshop (developed by the Carnegie Mellon® Software Engineering Institute), global analysis, and an approach developed by Fergus O'Brien. These methods were chosen because they are in widespread use or emphasize the capture of architecturally significant requirements.

Three problems must be solved to systematically transform business and mission goals into architecturally significant requirements: (1) the requirements must be expressed in a form that provides the information necessary for design; (2) the elicitation of the requirements must capture architecturally significant requirements; and (3) the business and mission goals must provide systematic input for elicitation process. The primary finding from the evaluation of these methods is that there are promising solutions to the first two problems. However, there is no method for systematically considering the business and mission goals in the requirements elicitation.

14. SUBJECT TERMS
architecturally significant requirements, natural language, Quality Attribute Workshop, QAW, requirements elicitation, use case analysis

15. NUMBER OF PAGES
52

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL