

Using Scenarios in Architecture Evaluations

Rick Kazman



When we analyze software architectures, we always want to do so with respect to an explicit or assumed set of quality attributes: modifiability, reusability, performance, and so forth. Most software quality attributes are, however, too complex and amorphous to be evaluated on a simple scale, in spite of our persistence in describing them that way.

Consider the following:

- Suppose that a system can accommodate a new computing platform merely by being re-compiled, but that same system requires a manual change to dozens of programs in order to accommodate a new data storage layout. Do we say that this system is or is not modifiable?
- Suppose that the user interface to a system is carefully thought out so that a novice user can exercise the system with a minimum of training, but the experienced user finds it so tedious as to be inhibiting. Do we say that this system is usable or not?

The point, of course, is that quality attributes do not exist in isolation, but rather only have meaning within a context. A system is modifiable (or not) with respect to certain classes of changes, secure (or not) with respect to specific threats, usable (or not) with respect to specific user classes, efficient (or not) with respect to specific resources, and so forth. Statements of the form “This system is highly maintainable” are, in my opinion, without operational meaning.

This notion of context-based evaluation of quality attributes has led us to adopt scenarios as the descriptive means of specifying and evaluating quality attributes. For example, the Software Architecture Analysis Method [1] is a scenario-based method for evaluating architectures; it provides a means to characterize how well a particular architectural design responds to the demands placed on it by a particular set of scenarios, where a scenario is a specified sequence of steps involving the use or modification of the system. It is thus easy to imagine a set of scenarios that would test what we normally call modifiability (by proposing a set of specific changes to be made to the system), security (by proposing a specific set of threat actions), performance (by proposing a specific set of usage profiles), etc.

A particular scenario actually serves as a representative for many different scenarios. For example, the scenario “change the background color on all windows to blue” is essentially equivalent to the scenario “change the window border decorations on all windows.” We use the clustering of scenarios as one of our evaluation criteria. As a consequence, judgment needs to be exercised as to whether the clustered scenarios

represent variations on a similar theme or whether they are substantially different. In the first case, the clustering is a good thing; in the second, it is bad. In other words, if a group of scenarios are determined to be similar, and they all affect the same component or components in an architecture (i.e., they cluster), we deem that to be a good thing, because it means that the system's functionality has been modularized in a way that properly reflects the modification tasks. If, on the other hand, a group of similar scenarios affect many different components throughout an architecture, we deem that to be bad.

As an aid to creating and organizing scenarios, we appeal to the concept of roles related to the system. Examples of roles include: the person responsible for upgrading the software—the end user; the person responsible for managing the data repositories used by the system—the system administrator; the person responsible for modifying the runtime functions of the system—the developer; the person responsible for approving new requirements for the system, etc. The concept of roles matches the difference between runtime qualities and non-runtime qualities—that is, those qualities that are a function of the system's execution (such as performance), and those that reflect operations performed offline in a development environment (such as modifiability). Scenarios of the former variety would be performed by roles such as the end user; the latter by developers or maintainers.

In a perfect world, the quality requirements for a system would be completely and unambiguously specified in a requirements document that is completed before the architecture begins. In reality requirements documents are not written, or are written poorly, or are not finished when it is time to begin the architecture. So the first job of an architecture evaluation is to elicit the specific quality goals against which the architecture will be judged.

If all of these goals are specifically, unambiguously articulated, that's wonderful. Otherwise, we ask the stakeholders to help us write them down. The mechanism we use is the scenario. A scenario is a short statement describing an interaction of one of the stakeholders with the system. A user would describe using the system to perform some task. A maintenance stakeholder would describe making a change to the system, such as upgrading the operating system in a particular way or adding a specific new function. A developer's scenario might talk about using the architecture to build the system or predict its performance. A customer's scenario might describe the architecture reused for a second product in a product line, or might assert that the system is buildable given certain resources.

Scenarios guide elicitation

In addition to clarifying requirements, scenarios help to prioritize what parts of the architecture should be elicited first. The brainstormed list of scenarios is prioritized based on several criteria. Prioritization criteria might include the most important uses of the system, the most illuminating uses in terms of how much of the architecture is covered by the scenario, the most important attribute goals, and the attribute goals that are most difficult to achieve.

Scenarios realized as sequences of responsibilities

A scenario can be thought of as a structure that operationalizes the other structures. A scenario starts with a stimulus of some kind and then shows how the architecture responds to that stimulus by identifying and locating the sequence of actions or responsibilities for carrying out a response to the stimulus. Scenarios tie together functional requirements, the quality attributes associated with the functional requirements, and the various architectural structures, as were presented in my December 1998 column, [“Representing Software Architecture”](#)[3].

The process of mapping or overlaying scenarios onto the architecture’s structures is an important aspect of many types of analysis. Modifiability analysis maps change scenarios onto the other structures as part of trying to understand the transitive closure of the change and the interaction among various types of changes. Reliability analysis starts by considering various failure scenarios. Security analysis starts by considering various types of threat scenarios. Performance analysis starts by considering a scenario for which a timing requirement is central.

Eliciting scenarios in design and analysis

Although we have been discussing scenarios that are specific to a single quality attribute, even quality-specific scenarios have an impact on multiple qualities. For example, consider the scenario: “change the system to add a cache to the client.” It is not only legitimate, it is mandatory, to ask about the effect of this change on performance, security, or reliability. So one quality may be used to motivate creation of a scenario but then the impact of that scenario on other qualities must be considered.

Furthermore, the requirements come from many stakeholders. Why is this? No single stakeholder represents all the ways in which a system will be used. No single stakeholder will understand the future pressures that a system will have to withstand. Each of these concerns must be reflected by the scenarios that we collect.

We now have a complex problem however. We have multiple stakeholders, each of whom might have multiple scenarios of concern to them. They would rightly like to be reassured that the architecture satisfies all of these scenarios in an acceptable fashion. And some of these scenarios will have implications for multiple system qualities, such as maintainability, performance, security, modifiability, and availability.

We need to reflect these scenarios in the architectural structures that we document and the architectures that we build. We need to be able to understand the impacts of the scenarios on the software architecture. We further need to trace the connections from a scenario to other scenarios, to the analytic models of the architecture that we construct, and to the architecture itself. As a consequence, understanding the architecture's satisfaction of the scenario depends on having a framework that helps us to ask the right questions of the architecture.

To use scenarios appropriately, and to ensure complete coverage of their implications, we typically consider three orthogonal dimensions, as shown in Figure 1.

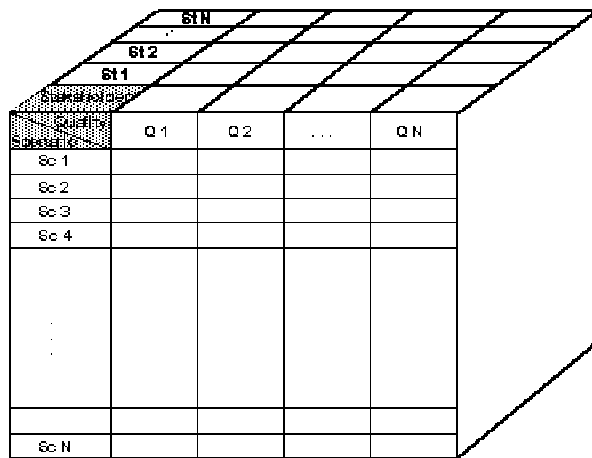


Figure 1: A Scenario Elicitation Matrix

The entries in the matrix are the specific scenarios. This characterization allows us to manage the scenarios not only for specification of requirements but also for subsequent validation of the architecture that is designed. The initial use of a quality-specific scenario might be considered during the design step, but the impact of that quality scenario on other qualities is also important during the analysis step.

Scenarios guide analysis

In the Architecture Tradeoff Analysis Method [2], we use three types of scenarios to guide and inform the analysis: use cases (these involve typical uses of the existing system

and are used for information elicitation); growth scenarios (these cover anticipated changes to the system), and exploratory scenarios (these cover extreme changes that are expected to “stress” the system). These different types of scenarios are used to probe a system from different angles, optimizing the chances of surfacing decisions at risk. Examples of each type of scenario follow.

Use cases:

1. *The ship's commander* requires his authorization before releasing certain kinds of weapons.
2. *A domain expert* wants to determine how the software will react to a radical course adjustment during weapon release (e.g., loft) in terms of meeting latency requirements?
3. *A user* wants to examine budgetary and actual data under different fiscal years without re-entering project data.
4. A user wants to have the system notify a defined list of recipients by e-mail of the existence of several data-related exception conditions, and have the system display the offending conditions in red on data screens.
5. *A tester* wants to play back data over a particular time period (e.g., last 24 hours).

Notice that each of the above use cases expresses a specific stakeholder's desires. (Scenarios normally do not include the stakeholder as we did above.) In some cases a specific attribute is also called out. For example, in the second scenario, latency is called out as being important. In other cases, the such as the following, growth scenarios are used to illuminate portions of the architecture that are relevant to the scenario:

1. Make the head-up display track several targets simultaneously.
2. Add a new message type to the system's repertoire.
3. Add collaborative planning: two planners at different sites collaborate on a plan.
4. Double the maximum number of tracks to be handled by the system.
5. Migrate to a new operating system, or a new release of the existing operating system.

The above growth scenarios represent typical anticipated future changes to the system. Each scenario also has attribute-related ramifications (other than for modifiability). For example, the following explanatory scenario will:

1. Add new feature 3-D maps.
2. Change the underlying platform to a Macintosh.

3. Reuse the software on a new generation of the aircraft.
4. Reduce the time budget for displaying changed track data by a factor of 10.
5. Improve the system's availability from 95% to 99.99%.

Each exploratory scenario stresses the system. Systems were not conceived to handle these modifications, but at some point in the future these might be realistic requirements for change.

Through the use of these scenarios, the software architecture is explored and analyzed. Use cases help in eliciting the architecture, understanding it, and analyzing its runtime qualities such as availability, security, and performance. Growth and exploratory scenarios help in understanding how the architecture will respond to future pressures, and thus aid in understanding the system's modifiability. Each is a crucial addition to gaining intellectual and managerial control over the important asset that a software architecture represents.

References

- [1] R. Kazman, G. Abowd, L. Bass, M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architectures," Proceedings of the 16th International Conference on Software Engineering, (Sorrento, Italy), May 1994, 81-90.
- [2] R. Kazman, M. Barbacci, M. Klein, S. J. Carriere, "Experience with Performing Architecture Tradeoff Analysis", Proceedings of ICSE 21, (Los Angeles, CA), May 1999, to appear.
- [3] R. Kazman, "Representing Software Architecture", *SEI Interactive*, Volume 1, Issue 3, December 1998.

About the Author

Rick Kazman is a senior member of the technical staff at the SEI, where he is a technical lead in the Architecture Tradeoff Analysis Initiative. He is also an adjunct professor at the Universities of Waterloo and Toronto. His primary research interests within software engineering are software architecture, design tools, and software visualization. He is the author of more than 50 papers and co-author of several books, including a book recently published by Addison-Wesley entitled *Software Architecture in Practice*. Kazman received a BA and MMath from the University of Waterloo, an MA from York University, and a PhD from Carnegie Mellon University.

The views expressed in this article are the author's only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University. This article is intended to stimulate further discussion about this topic.

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon University.

SM IDEAL, Interim Profile, Personal Software Process, PSP, SCE, Team Software Process, and TSP are service marks of Carnegie Mellon University.

® Capability Maturity Model, Capability Maturity Modeling, CERT Coordination Center, CERT, and CMM are registered in the U.S. Patent and Trademark Office.