

# Scenario-Based Analysis of Software Architecture

*Rick Kazman*

*Department of Computer Science, University of Waterloo  
Waterloo, Ontario*

*Gregory Abowd*

*College of Computing, Georgia Institute of Technology  
Atlanta, Georgia*

*Len Bass, Paul Clements*

*Software Engineering Institute, Carnegie Mellon University  
Pittsburgh, Pennsylvania*

**To appear in *IEEE Software*, November 1996**

**Abstract:** Software architecture is one of the most important tools for designing and understanding a system, whether that system is in preliminary design, active deployment, or maintenance. Scenarios are important tools for exercising an architecture in order to gain information about a system's fitness with respect to a set of desired quality attributes. This paper presents an experiential case study illustrating the methodological use of scenarios to gain architecture-level understanding and predictive insight into large, real-world systems in various domains. A structured method for scenario-based architectural analysis is presented, using scenarios to analyze architectures with respect to achieving quality attributes. Finally, lessons and morals are presented, drawn from the growing body of experience in applying scenario-based architectural analysis techniques.

**Keywords:** Software Architecture; Software Analysis Methods; Software Quality, Software Architecture Analysis, Applications of Scenarios

## 1 Introduction

Analysis of a proposed software system to determine the extent to which it meets desired quality criteria is desirable. Some of the reasons why such analysis is difficult include a lack of common understanding of high level design and a lack of fundamental understanding of many of the quality attributes. With the recent surge of interest in software architecture,<sup>1</sup> some of the issues involved in the high level design of software systems are being clarified. Our goal in this paper is to show how to exploit software architectural concepts to analyze<sup>2</sup> complex software systems for quality attributes. We compensate for the lack of fundamental understanding about how to express these attributes by using scenarios to capture essential actions involving the system under analysis.

---

1. See, for example, the April, 1995, special issue of *IEEE Transactions on Software Engineering* devoted to software architecture.

2. We distinguish between analysis and evaluation. Analysis is, according to Webster's New Collegiate dictionary, "1. separation of a whole into its component parts 2. an examination of a complex, its elements, and their relations." This definition is at the heart of what we consider important in examining software architectures. Evaluation pre-supposes a particular value scale or system, which in our world is not always forthcoming.

We will review our experiences with scenario-based analysis of architectural descriptions of software systems. Scenarios are brief narratives of expected or anticipated use of a system from both development and end-user viewpoints. A structured method employing scenarios to analyze architectures is the Software Architecture Analysis Method (SAAM). SAAM will be described in Section 2. Experience with SAAM and SAAM-related techniques will be recounted in Section 3. Section 4 will explore lessons learned.

We begin with a discussion of the relationship among software architecture, quality attributes, and scenarios.

## 1.1 Software architecture

Software architecture describes a high-level configuration of components that compose the system, and the connections that coordinate the activities of those components. We say *software* architecture here, but it is quite often the case that such high-level configurations describe functionality that will ultimately be performed by either software or hardware components. We also say *a* high-level configuration rather than *the* high-level configuration, because a system can be composed of more than type of component; each decomposition will therefore have its own configuration. For instance, a system may be composed of a set of modules in the sense of Parnas [16], and also a set of cooperating sequential processes, each of which resides in one or more modules. Both viewpoints are valid, and both are architectural in nature. But they carry different information. From the process viewpoint we can describe the interaction of the system during execution, in terms of how and when processes become active or dormant, pass or share data, or otherwise synchronize. From the module viewpoint we can describe the interaction of the teams responsible for building the modules, in terms of the information they are: allowed to share, required to share (interfaces), or prohibited from sharing (implementation secrets). The process viewpoint has implications for performance; the module viewpoint has implications for maintainability.

Software architecture manifests its usefulness in the life cycle in the following ways:

- An architecture is often the first artifact in a design that represents decisions on how requirements of all types are to be achieved. As the manifestation of early design decisions, the architecture represents those design decisions that are hardest to change [15] and hence are deserving of the most careful consideration.
- An architecture is the key artifact in achieving successful product line engineering, the disciplined structured development of a family of similar systems with less effort, expense, and risk than developing each system independently [14].
- Architecture is usually the first artifact to be examined when a programmer (particularly a maintenance programmer) unfamiliar with the system begins to work on it.

Our emphasis on analysis of software architectures is compatible with the belief that understanding of the implications of a design leads to early detection of errors, and to the most predictable and cost-effective modifications to the system over its entire life cycle.

## 1.2 Quality Attributes

We are interested in evaluating architectures to determine their fitness with respect to certain properties or *qualities* of the resulting system, such as modifiability or security. However, it is too difficult to analyze an architecture based on these abstract qualities which are too vague and provide very little procedural support for evaluating an architecture.

As an example of vagueness, suppose we can change the colors in a user interface by changing a resource file which is read in at run-time, but changing the fonts used in the interface requires a re-compilation. Is

this system modifiable or not? The answer is, perhaps, yes with respect to changing colors, but no with respect to changing fonts. And whether the design is acceptable or not depends on predictions of actual usage: if the user interface is modifiable in a way that is important to its owner, then we can say that the system is *appropriately* modifiable. This notion of appropriateness applies to all quality factors.

The lesson is that at the present time and for the foreseeable future, there are no simple (scalar) “universal” measurements for attributes such as safety or portability. Rather, there are only context-dependent measures, meaningful only in the presence of specific circumstances of execution or development. Safety benchmarks are a fine example. If there were a universal measurement of safety, benchmarks would be unnecessary. As it is, a benchmark represents data about a system executing with particular inputs in a particular environment, and we use them as benchmarks.

While we may wish for better understanding and more universal expression of quality attributes, for now we must recognize the role played by specifying a particular operational context for a system. To represent contexts, we use scenarios.

### 1.3 Scenarios

Scenarios have been widely used and documented as a technique during requirements elicitation, especially with respect to the operator of the system ([4], [7]). They have also been widely used during design as a method of comparing design alternatives. Experience also shows that programmers use them to understand an already-built system, by asking how the system responds (component by component) to a particular input or operational situation. Scenarios have not, however, been used as a tool for analysis of quality, our primary utilization of them. We use scenarios to express the particular instances of each quality attribute important to the customer of a system. We then analyze the architecture under consideration with respect to how well or how easily it satisfies the constraints imposed by each scenario.

Scenarios differ widely in breadth and scope. Our use of scenarios is as a brief description of some anticipated or desired use of a system. At this point in our work, scenarios are typically one sentence long and could more appropriately be called vignettes.

We emphasize the use of scenarios appropriate to all roles involving a system. The operator role is one widely considered but we also have roles for the system designer and modifier, the system administrator, and others, depending on the domain. It is important when analyzing a system that all roles relevant to that system be considered since design decisions may be made to accommodate any of the roles.

The process of choosing scenarios for analysis forces designers to consider the future uses of, and changes to, the system. Thus we believe that architectural analysis cannot give precise measures or metrics of fitness. Such measures would need to be couched in terms of qualities (e.g. “how modifiable is this architecture?”) and such questions are typically of little value. What we really want to know is: “how will this architecture accommodate the following change?” or “how will this architecture accommodate a change of the following class?”, and we use architectural analysis to guide our inspection of the architecture, focussing attention on potential trouble spots.

Of course, not all scenarios speak to architecture-level issues. For example, a portability scenario might have architectural implications (such as determining how machine dependencies should be isolated) but it may also depend upon code-level or hardware-level factors, such as byte ordering. Furthermore, some scenarios simply cannot be evaluated using architectural information. For example, if a developer’s scenario was to ensure that no module had more than 250 lines of code, this constraint could not be either checked or ensured by architecture-level analysis.

## 2 A method for scenario-based architectural analysis

A particular method for doing a scenario-based architectural analysis is SAAM (Software Architecture Analysis Method). SAAM was originally developed to enable comparison of competing architectural solutions [11]. As a result of our experience with architectural analysis, the prescribed steps of SAAM have evolved. Not all of our experience with architectural analysis has strictly followed the method prescribed by SAAM, nor has it always been the case that we were comparing competing candidate architectures. Nevertheless, in all cases scenarios were used as the foundation for illuminating the properties of an architecture, and from this body of experience a stable set of activities and dependencies between those activities has emerged, which we call SAAM. SAAM therefore may be considered a canonical method for scenario-based architecture analysis of computer-based systems; particular analysis efforts may be carried out using a subset or variation of SAAM as appropriate.

Figure 1 shows the steps of SAAM and the dependency relationships between those stages. The steps of

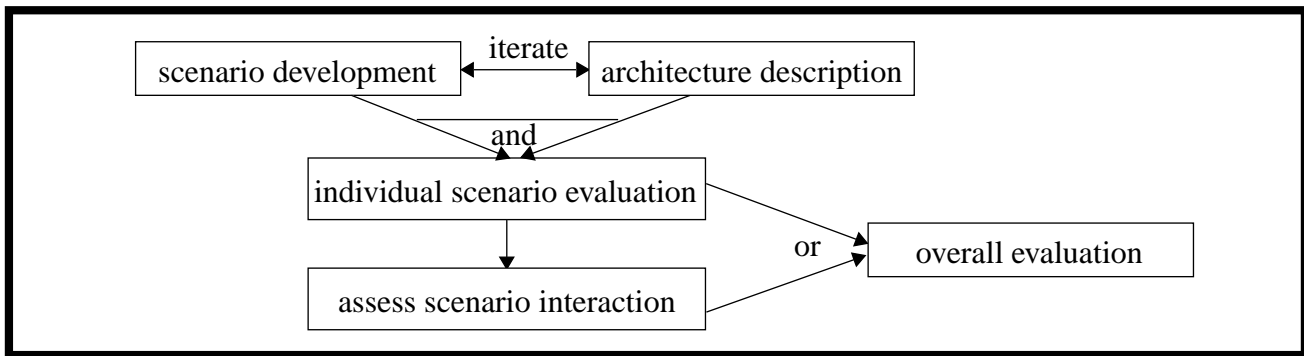


Figure 1: Activities and dependencies in scenario-based analysis

SAAM, and the products of each, are:

1. **Describe candidate architecture.** The candidate architecture or architectures should be described in a syntactic architectural notation that is well-understood by the parties involved in the analysis. These architectural descriptions need to indicate the system's computation and data components, as well as all component relationships, sometimes called connectors.
2. **Develop scenarios.** Develop task scenarios that illustrate the kinds of activities the system must support and the kinds of changes which it is anticipated will be made to the system over time. In developing these scenarios, it is important to capture all important uses of a system. Thus scenarios will represent tasks relevant to different roles such as: end user/customer, marketing, system administrator, maintainer, and developer.
3. **Perform scenario evaluations.** For each scenario, determine whether the architecture can be executed directly, or whether a change would be required to execute it (in which case we call the scenario *indirect*). For each indirect scenario, list the changes to the architecture that are necessary for it to support the scenario and estimate the cost of performing the change. A modification to the architecture means that either a new component or connection is introduced or an existing component or connection requires a change in its specification. By the end of this stage, there should be a summary table which lists all scenarios (direct and indirect). For each indirect scenario the impact, or set of changes, that scenario has on the architecture should be described. In our experience, it is sufficient to list the existing components and connections that must be altered and the new components and connections that must be introduced, although our method allows for more sophisticated cost functions. A tabular summary is especially useful when comparing alternative architectural candidates because it provides an

easy way to determine which architecture better supports a collection of scenarios.

4. **Reveal scenario interaction.** Different indirect scenarios may necessitate changes to the same components or connections. In such a case we say that the scenarios *interact* in that component or connector. Determining scenario interaction is a process of identifying scenarios that affect a common set of components. Scenario interaction measures the extent to which the architecture supports an appropriate separation of concerns. For each component determine the scenarios which affect it. SAAM favors the architecture with the fewest scenario conflicts.<sup>3</sup>
5. **Overall evaluation.** Finally, weight each scenario and the scenario interactions in terms of their relative importance and use that weighting to determine an overall ranking. This is a subjective process, involving all of the stake-holders in the system. The weighting chosen will reflect the relative importance of the quality factors that the scenarios manifest.

## 2.1 Notes on the method

As discussed in Section 1.1, a software architecture may have more than one representation. There is an appreciable amount of ongoing research into languages and representations for these static configurations, but no clearly superior notation has yet emerged. For our purposes, we have tended to use very simplistic architectural primitives in our case studies and have not found these simple representations too limiting. A typical representation will distinguish between components that are active (transform data) and passive (store data) and also depict data (passing information between components) and control (one component enabling another component to perform its function) connections. This simple lexicon provides a reasonable static representation of the architecture. Accompanying this static representation of the architecture is a description of how the system behaves over time, or a more dynamic representation of the architecture. This can take the form of a natural language specification of the overall behavior or some other more formal and structured specification.

Steps 1 and 2 are highly interdependent. Deciding the appropriate level of granularity for an architecture will depend on the kinds of scenarios you wish to evaluate (though not all scenarios are appropriate, such as a code size scenario). Determining a reasonable set of scenarios also depends on the kinds of activities you expect the system to be able to perform, and that is reflected in the architecture. One important relationship between steps 1 and 2 is the role that direct scenarios play in helping to understand an architectural description. Direct scenarios can help to determine static architectural connections and can aid in the formulation of more structured descriptions of the dynamic behavior of an architecture.

Rather than offering a single architectural metric, SAAM produces a collection of small metrics (per-scenario analyses). Given this set of mini-metrics, SAAM can be used (and in fact was developed with the intent to) compare competing architectures on a per-scenario basis. It is left to the users of SAAM to determine which scenarios are most important to them, in order to resolve cases where the candidates out-score each other on different scenarios. Overall evaluation can only be derived in the context of organizational requirements.

## 3 Validation of the method

Although SAAM is intended to be applied early in the design, in order to validate it we used it to analyze a number of existing systems. These applications were industrial in nature, and decidedly non-academic:

---

3. This assumes that the scenarios are inherently different in nature. We will return to this point in Section 4.3.

Participants based subsequent development or procurement actions on the outcome of the analyses. Applications include:

1. **Global information system** — A company was contemplating the purchase of a system as the infrastructure to support applications development for multimedia communication with unlimited conferencing. The purchasing company wanted some assurance that the architecture of the system they purchased was going to provide for the generic satellite-based multi-user applications they anticipated developing in the near and long term. As a result of the analysis, the company decided to not purchase the system, avoiding an investment of tens of millions of dollars.
2. **Air traffic control** — This was an investigation of a complex, real-time system against a set of proposed changes to that system. The purpose of the evaluation was to determine whether future development on this system was justified. The change scenarios were intended to represent appropriate manifestations of the abstract qualities of performance and availability. The result of this evaluation was a decision to proceed with the proposed changes [2].
3. **WRCS** — This case study was an analysis of a commercial version control/configuration management tool. This analysis covers all activities of SAAM and shows all artifacts of a SAAM evaluation that can be produced. The result of the analysis was that significant problems were discovered with the product's architecture, with respect to the scenarios considered. This case study appears in the next section.

We have also conducted a number of other industrial and academic case studies in scenario-based analysis as SAAM was maturing in areas such as user interface development environments [11], internet information systems [13], key word in context (KWIC) systems [3], embedded audio systems, and visual debuggers.

## 3.1 The WRCS System

### 3.1.1 System context/purpose

In this section we will discuss the application of SAAM to a commercially available revision control system, based upon RCS [18], which we will call WRCS. WRCS provides the functionality to allow developers of projects the ability to create archives, compare files, check files in and out, create releases, back up to old versions of files, and so on. "Project" in this context means any group of related files that, when linked together appropriately, form a finished product. For example these files might be source code for a computer program, text for a book, or digitized audio and video for the creation of a video clip. WRCS keeps track of changes made to these files as they evolve over time. It provides capabilities for multiple users to work on the same project within their own private work areas, allowing each developer to modify and test the system in isolation, without disturbing other developers' work and without corrupting the primary copy of the system. Managerial functions, such as production of reports, are also provided. WRCS's functionality has been integrated with several program development environments, and can be accessed through these tools, or through WRCS's own graphical user interface.

Certain details have been slightly modified to protect proprietary interests.

### 3.1.2 Applying the steps of SAAM

#### Develop Scenarios/Describe Candidate Architecture

For any evaluation to take place we require an architectural representation of the product with a well-specified semantic interpretation (principally what it means to be a component or a connector). Creating an architectural description proved to be one of the most difficult tasks in evaluating WRCS. At the start of this project there was *no* architectural description of the product, and so we needed to devise a way of

eliciting this information.

This information had to be analyzed and grouped in a way that it would aid in the construction of an architectural diagram. Our sources of information were limited: they consisted of interviews with some of the members of the development team, the product's documentation, and the product itself. In particular, we had no access to the source code or the product's specifications. This is appropriate in that software architecture is supposed to concern itself with a level of abstraction above code. In essence, our task was reverse engineering: to create a design document out of a finished product.

The product's architectural description was arrived at iteratively. At each stage we studied the product's existing description, the product itself (executables and libraries), and its documentation, and devised a new set of questions. The answers to the questions in each stage helped us to clarify the current description. Each new stage allowed us to obtain more insight on the product and motivate new questions to be asked in order to arrive at the next stage. Since we didn't have any previous representation we chose to start with a gross listing of the modules along with their basic relationships, and from there iterate, adding structure as we went. The process of eliciting scenarios also helped to clarify the architecture, as we shall see in the next section.

It took three iterations to obtain a representation which was satisfactory for architectural evaluation. This representation is shown in Figure 2.

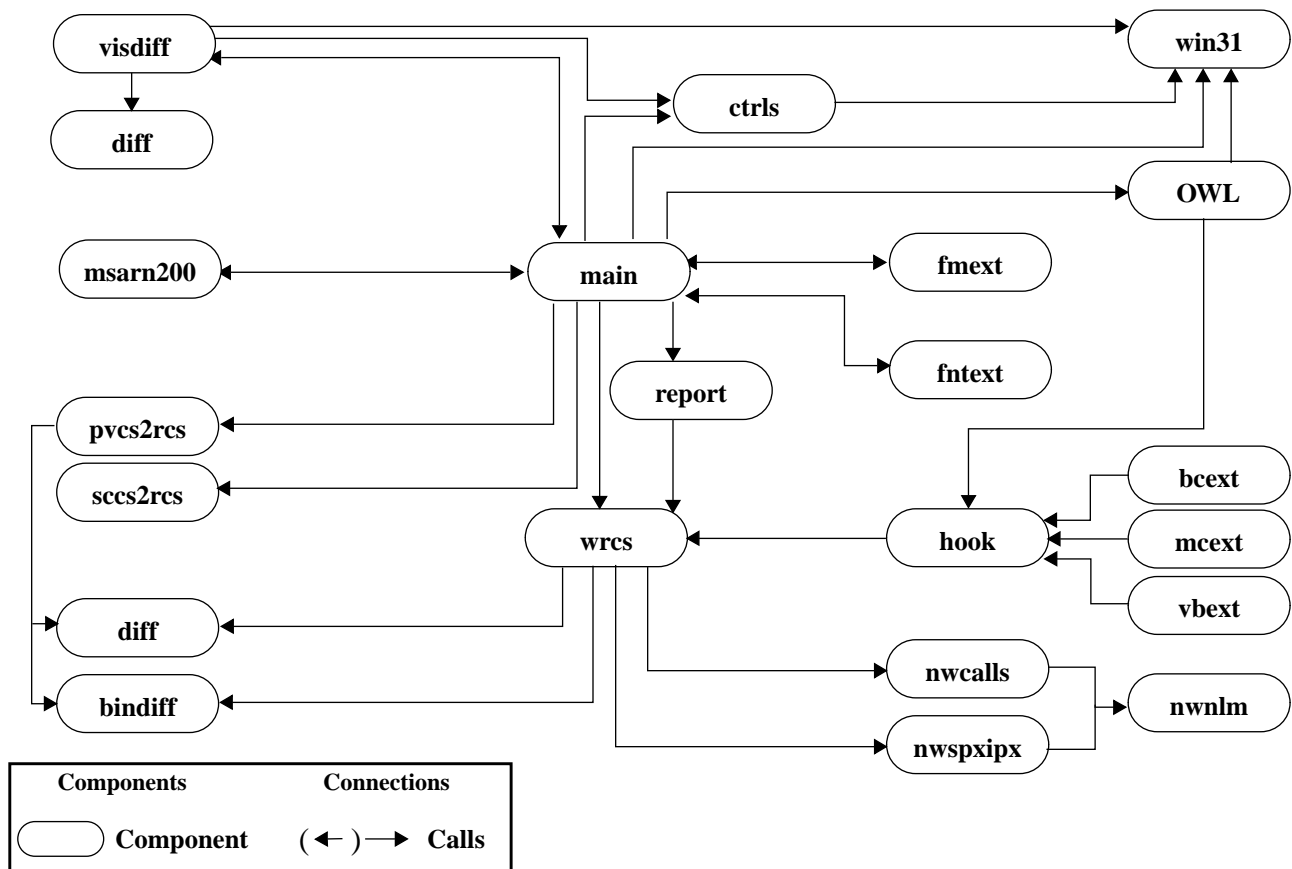


Figure 2: Architectural Representation of WRCS

During the process of describing the architecture, scenarios were continually developed that represented the various stakeholder roles in the system. For WRCS these roles were: users, developers, maintainers, and system administrators. Scenario enumeration is simply a particular form of requirements elicitation and analysis [1]. These scenarios were developed in discussion with *all* the stake-holders in the system, in

order to try to characterize all current and projected uses of the system. The scenarios formed the basis for all further architectural evaluation.

The tasks which we present here are a subset of the tasks which were elicited from the WRCS domain expert. In total we studied 15 tasks, 6 of which are presented here. A complete evaluation of a complex system would involve dozens of scenarios [7].

#### User:

1. *Compare binary file representations.* Compare binary files generated by other products. For example, FrameMaker files are stored in a binary representation. But when we are comparing two versions of a FrameMaker file we want to see our editing changes in a human-readable form, and not the changes to the binary codes stored in the files.
2. *Configure the product's toolbar.* Change the icons and actions associated with a button in the toolbar.

#### Maintainer:

3. *Port to another operating system.*
4. *Make minor modifications to the user interface.* Add a menu item, change the look and feel of a dialog box.

#### Administrator:

5. *Change access permissions for a project.*
6. *Integrate with a new development environment.* Attach for example to Symantec C++.

#### Perform Scenario Evaluations

Once the scenarios have been created, we then need to classify them as direct (i.e. those that can be satisfied by executing the system being developed) or indirect (i.e. those which require a change to some of the components or connections within the architecture). The direct/indirect classification is a first indication of the fitness of an architecture with respect to satisfying a set of scenarios. For example, looking at scenario 2 above, if one can reconfigure a product's toolbar within the product, then we say that this is a direct scenario with respect to WRCS's architecture. If one needs to modify the architecture to achieve this change then the task is indirect, and so the architecture is less desirable with respect to the feature. At this stage, we also want to estimate the difficult of the change (say, in terms of person-hours required, or lines of code impacted). One might simply modify an ascii resource file and re-start the product, in which case the architectural implications of this indirect scenario are minimal. One might need to change an internal table and re-compile, in which case the implications of scenario 2 are moderate. Or one might need to dramatically re-structure the user interface code, in which case the implications are considerable.

We indicate the nature of the scenarios, and which of WRCS's modules they affect in Table 1.

Scenario	Description	Direct/ Indirect	Changes
1	Compare new binary file representations	Indirect	This will require modifications to <b>diff</b> (to make the comparison) and <b>visdiff</b> (to display the results of the comparison).
2	Configure the product's toolbar	Direct	

**Table 1: Scenario Evaluations for WRCS**



Scenario	Description	Direct/ Indirect	Changes
3	Port to another operating system	Indirect	All components that call <b>win31</b> must be modified; specifically: <b>main</b> , <b>visdiff</b> , and <b>ctrls</b> . If the target operating system does not support OWL then either OWL needs to be ported, or all components that call OWL, specifically: <b>main</b> and <b>hook</b> . If the new operating system is not supported by Novell's software then <b>wrcs</b> will have to be modified to work with a new networking environment
4	Make minor modifications to the user interface	Indirect	This will require changes to one or more of those components which call the win31 API, specifically: <b>main</b> , <b>diff</b> and <b>ctrls</b> .
5	Change access permissions for a project	Direct	
6	Integrate with a new development environment	Indirect	This requires changes to <b>hook</b> , as well as the addition of a module along the lines of <b>bcext</b> , <b>mcext</b> , and <b>cbext</b> , which connects the new development environment to <b>hook</b>

**Table 1: Scenario Evaluations for WRCS**

### Reveal Scenario Interactions

When two or more indirect task scenarios necessitate changes to some component of a system, they are said to *interact*. Scenario interaction is an important consideration because it exposes the allocation of functionality to the product's design. In a very explicit way it is capable of showing which modules of the system are involved in tasks of different nature. High scenario interaction reveals a poor isolation of functionality in a particular component of a design, giving a clear guideline on where to focus the designer's subsequent attention. As we shall show in Section 4.3, the amount of scenario interaction is related to metrics such as structural complexity [8], coupling, and cohesion [9], and so it is likely to be strongly correlated with number of defects in the final product.

Table 2 shows the number of changes required in each module of the system. In this table we are taking into account *all* the relevant scenarios elicited in the WRCS analysis, not just the 6 presented in section above. Since each of these scenarios imposes a single change to the architecture, the number of changes per module indicates the level of indirect scenario interactions for the module.

Module	# of Changes
wrcs	7
main	4
hook	4
visdiff	3
ctrls	2
report, diff, bindiff, pvcs2rcs, sccs2rcs, nwcalls, nwspixpx, nwnlm	1 each

**Table 2: Scenario interactions by module for WRCS**

One visualization technique we have used to highlight scenario interactions is a fish-eye view. In Figure 2 the WRCS architecture is presented with module size made proportional to the number of interacting scenarios which affect it. This figure shows where the scenario interactions lie, and the relative scale of the interactions. It can be seen clearly that the component with most scenario interaction is **wrcs**. It is within

this component that most of the future development effort will be concentrated. Modules **main**, **visdiff**, and **hook** also suffer from high scenario interaction, and **ctrls** has a small amount of scenario interaction.

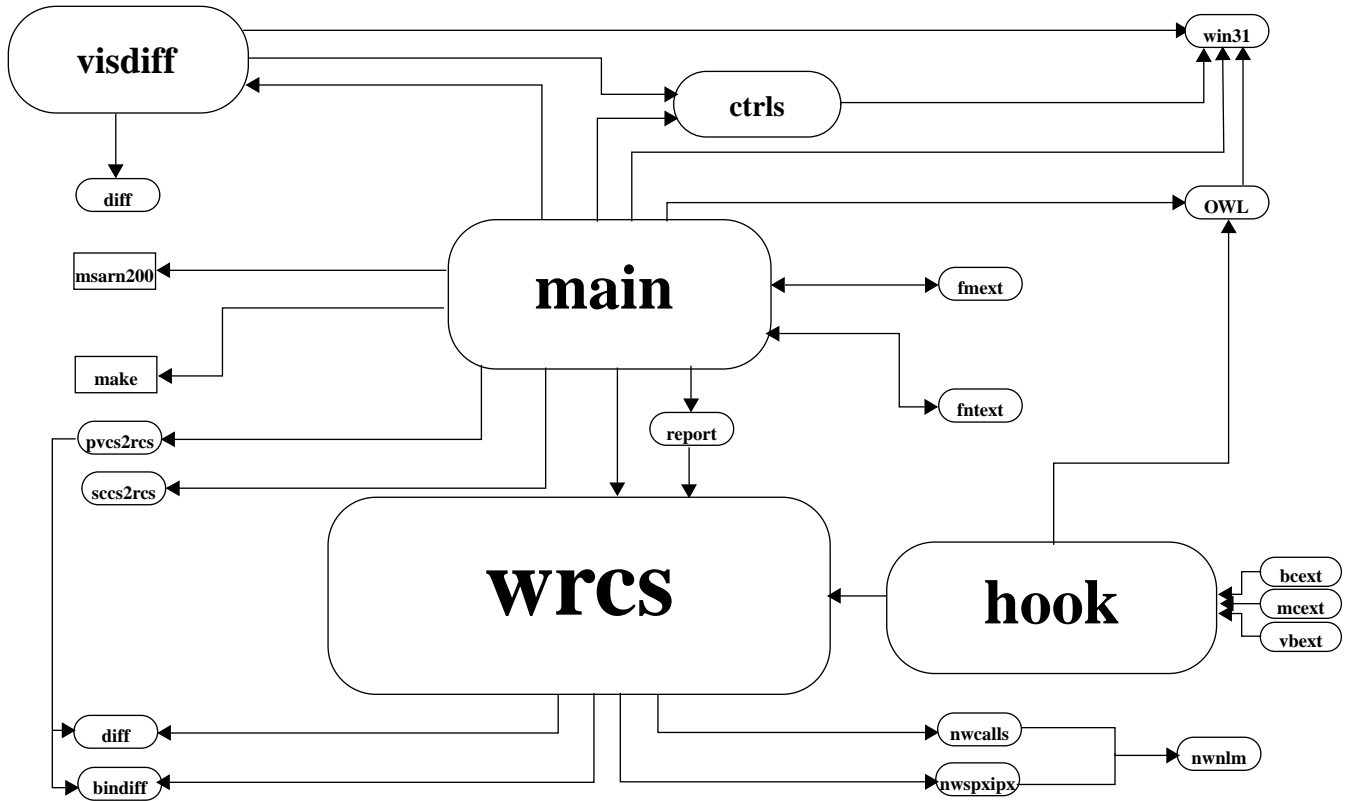


Figure 3: Fish-eye Representation of Scenario Interactions in WRCS

This information immediately calls attention to the most architecturally significant features of the system, as it currently exists, and guides designers and developers in their allocation of time and effort. It has proven to be a highly effective device for communication among team members in the WRCS case study.

### Overall Evaluation

Once the scenarios have been determined, mapped onto the structural description, and all scenario interactions have been determined, the extent of the implications of the scenarios is made manifest. All that remains to be done is to prioritize the scenarios which have been identified as potentially problematic, in order to arrive at an overall evaluation of the architecture.

#### 3.1.3 What was the result?

The WRCS analysis identified a number of severe limitations in achieving the extra-functional qualities of portability and modifiability. A major redesign of the system was recommended. Having gone through an analysis procedure such as SAAM before implementation would have substantially contributed to avoiding the problems which the WRCS developers now face.

#### 3.1.4 What did we learn?

Within the organization the evaluation itself obtained mixed results. Senior developers and managers found a very important tool in architectural analysis and plan to impose it in future developments of new products. They realized that they can identify many potential problems early in the software life cycle and at an extremely low cost. Within the WRCS team, however, this evaluation was regarded as just an aca-

ademic exercise. We attribute this to the fact that senior developers and managers have enough perspective to understand that the majority of the software development life cycle is spent in maintenance and feature enhancements. For this reason, any effort that aids in improving a product's support for extra-functional qualities is significant. However, the developers within the WRCS team did not have this broad perspective. When one is concerned with meeting the next release deadlines, or with finding a bug, there is no time for the luxury of contemplating major changes to the architecture. In the words of one senior manager, "They have features to implement!" This is why architectural analysis must be done early. Otherwise, it will never be done or, if done, it will be meaningless.

SAAM allowed an insight to the product capabilities that could not be easily achieved from inspections of code and design documents. In a very simple, straightforward and cost-effective way it exposed specific limitations of the product. Furthermore, this was accomplished with only scant knowledge of the internal workings of WRCS. As we said earlier, we had no access to the WRCS source code.

Most importantly, the process, and its frustrating lack of real usable results, has caused them to change their practice for future development. It has convinced them of the need for architectural analysis up front.

## **4 Results and Lessons**

Having now performed architectural evaluations on half a dozen small to medium sized software architectures and two large industrial systems, we have begun to see patterns emerging in the ways that architectural analysis proceeds, and in the benefits which accrue to the process.

### **4.1 SAAM is for people**

The strengths of SAAM are largely social. The process of analysis helps to focus attention on the important details of the architecture, and allows users to ignore less critical areas. The use of scenarios has proven to be an important tool for both communication among a team of developers and for communication between a development team and upper-level managers. The use of scenarios suggests where to: refine an architectural description, ask more questions, refine an analysis. It is difficult to get agreement on an "appropriate" set of scenarios; the process of doing so forces the system's stakeholders to talk and reach consensus. A collection of scenarios—particularly scenarios which have caused problems for similar systems in the past—can provide a benchmark with which to evaluate new designs.

Finally, visualization has proven to be an effective tool in communicating design problems to the stakeholders. The visualization of an architecture, emphasizing scenarios and scenario interaction focuses attention, effectively proposing areas for discussion.

### **4.2 SAAM and traditional architectural metrics**

Architectural evaluation has an interesting relationship with the more traditional design notions of coupling and cohesion. Good architectures exhibit low coupling and high cohesion in terms of some breakdown of functionality. What does this mean in terms of a SAAM analysis? Low coupling means that a single scenario doesn't affect large numbers of structural components. High cohesion means that structural components are not host to scenario interactions. The implication of this correspondence is that architectural analysis is a means of determining coupling and cohesion in a highly directed manner.

Architectural metrics such as structural complexity, as well as metrics for coupling and cohesion, have been criticized as being crude instruments of measure. SAAM improves upon these metrics by allowing

one to measure coupling and cohesion with respect to a particular scenario or set of scenarios. In this way the instruments of measures become much sharper, and hence more meaningful. For example, in the standard interpretation of coupling, if two components are coupled, they are coupled irrespective of whether they communicate once (say, for initialization) or repeatedly. Similarly, structural complexity measures (based upon data inflows and outflows from components) do not consider predicted future changes to a given part of the architecture. They simply record a part of the architecture with a high structural complexity as being “bad”. Scenarios, on the other hand, will tease cases such as these apart.

### 4.3 Determining the proper level of architectural description

As we have already said, one of the benefits of software architecture is the ability to view software from a higher level of abstraction. This means that an architectural diagram, to be useful, must choose an appropriately high level of description. However, how do the designers of the architecture know what that level should be? The simple answer is: whatever level the scenarios dictate. This is exactly what happened when we iterated through our three versions of the representation of the WRCS system.

When the architecture has been given its initial structural description, we need to map the scenarios onto the structure. In particular, for each *indirect* scenario, we need to highlight the components and connections which will be affected by the change that the scenario implies. We are primarily interested in indirect scenarios as they represent the extra-functional qualities which the architecture is to satisfy, whereas the direct scenarios represent the system’s function. Direct scenarios, and their interactions are interesting only insofar as the indicate a component’s potential complexity.

The mapping of scenarios onto the structural description serves two purposes: it guides the process of architectural evaluation, and it aids in validating scenario interaction (a difficult process without this step, as [7] describes). What does it mean to have multiple indirect scenarios that affect a single module? There are three possible meanings.

- First, the interaction could mean that the scenarios are all of the same class. That is, they could be variants of the same basic scenario. For this case, the fact that the scenarios are of the same class and cluster together in the same module can be taken to be a good sign. It means that the system’s functionality is sensibly allocated. Put another way, it means that the architecture exhibits high *cohesion* with respect to this class of scenarios.
- Second, the interaction could mean that the scenarios are of different classes and that the module can be further subdivided, but that it was not shown subdivided in the original architectural representation. Recall that we said that there is no *a priori* right level of description for architectural description, but that the scenarios would dictate the appropriate level. For example, it might be that the module is really composed of three functions, each of which deals neatly with one of the scenarios. In this case, the process of scenario-based architectural analysis has helped to refine the level at which the software architecture is presented.
- Third, the interacting scenarios could be of different classes and the module cannot be further subdivided. This case reveals a potential problem area within the architecture, since, if scenarios from different classes are affecting the same module then the architecture is not appropriately separating concerns.

### 4.4 Determining the proper set of scenarios

Given the great emphasis that SAAM places on scenarios, an interesting question is: “when has one generated a sufficient number of scenarios to adequately test the architecture”? Or, put another way: “when

should one stop generating new scenarios”? There are two possible answers to this question. The simple answer is: “when you run out of resources”. The more complex and meaningful answer is that one can stop generating scenarios when the addition of a new scenario no longer perturbs the design. In this way scenario generation is much like software testing: you cannot prove that you have a sufficient number of test cases, but you can determine a point at which the addition of new test cases is providing negligible improvement to the software.

One way of minimizing the number of scenarios needed (again, on analogy with testing), is to group scenarios into equivalence classes, as discussed earlier. However, this merely generates a new question. Given the emphasis on classes of scenarios to determine architectural cohesion, how can we determine whether scenarios are *appropriately* grouped into classes? Another way of thinking about the problem of scenario classes is: all domain experts should cluster scenarios the same way. If they do not, then they must have additional, implicit scenarios in mind, and these must be elicited.

## 5 Acknowledgments

The authors would like to acknowledge the efforts and contributions of Mauricio de Simone and Linda Northrop in the creation of this paper: This work was sponsored in part by the National Sciences and Engineering Research Council of Canada and the U. S. Department of Defense.

## 6 References

- [1] Atwood, J. *The Systems Analyst*, Hayden, 1977.
- [2] Brown, A., Carney, D., Clements, P., “A Case Study in Assessing the Maintainability of a Large, Software-Intensive System”, *Proceedings of the International Symposium on Software Engineering of Computer Based Systems*, Tucson, Az., IEEE Computer Society, March 1995.
- [3] Clements, P., Bass, L., Kazman, R., Abowd, G., “Predicting Software Quality by Architecture-Level Evaluation”, *5th International Conference on Software Quality*, (Austin, TX), October 1995, to appear.
- [4] Dardenne, A., “On the Use of Scenarios in Requirements Acquisition”, CIS-TR-93-17, Department of Computer and Information Science, University of Oregon, 1993.
- [5] Dean, T., Cordy, “A Syntactic Theory of Software Architecture”, *Transactions on Software Engineering*, 21(4), April 1995, 302-313.
- [6] Garlan, D., Shaw, M. “An Introduction to Software Architecture”. *Advances in Software Engineering and Knowledge Engineering*, Volume I, World Scientific Publishing, 1993.
- [7] Gough, P., Fodemski, F., Higgins, S., Ray, S., “Scenarios - an Industrial Case Study and Hypermedia Enhancements”, *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, York, England, March, 1995, 10-17.
- [8] Henry, S., Kafura, D. “Software Structure Metrics Based on Information Flow”, *IEEE Transactions on Software Engineering*, SE-7(5), Sept. 1981.
- [9] Heyliger, G., “Coupling”, *Encyclopedia of Software Engineering*, J. Marciniak (ed.), 220-228.
- [10] Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

- [11] Kazman, R., Bass, L., Abowd, G., Webb, M., “SAAM: A Method for Analyzing the Properties of Software Architectures”, *Proceedings of ICSE 16*, Sorrento, Italy, May 1994, 81-90.
- [12] Kazman, R., Bass, L., “Toward Deriving Software Architectures from Quality Attributes”, CMU/SEI-94-TR-10, Software Engineering Institute, Carnegie Mellon University, 1994.
- [13] Kazman, R., Bass, L., Abowd, G., and Clements, P., “An Architectural Analysis Case Study: Internet Information Systems,” *Proceedings, First International Workshop on Software-Intensive Systems*, Seattle, April 1995. (Also available as CMU-CS-TR-95-151, School of Computer Science, Carnegie Mellon University, Pittsburgh).
- [14] Mettala, E., Graham, M. (eds.), “The Domain-Specific Software Architecture Program”, CMU/SEI-92-SR-9, Software Engineering Institute, Carnegie Mellon University, 1992.
- [15] Parnas, D., “On the design and development of program families,” *IEEE Transactions on Software Engineering*, SE-2(1), 1976, 1-9.
- [16] Parnas, D., “On the criteria for decomposing systems into modules,” *Communications of the ACM*, 15(12), December 1972, 1053-1058.
- [17] Shaw, M., “Larger Scale Systems Require Higher-Level Abstractions”, *Proceedings of Fifth International Workshop on Software Specification and Design*, IEEE Computer Society, 1989, 143-146.
- [18] Tichy, W. “RCS—A System for Version Control”, *Software—Practice & Experience*, 15(7), July 1985, 637-654.
- [19] Weiss, D., Parnas, D., “Active Design Reviews: Principles and Practices,” *Proceedings, Eighth International Conference on Software Engineering*, 1985, 132-136.