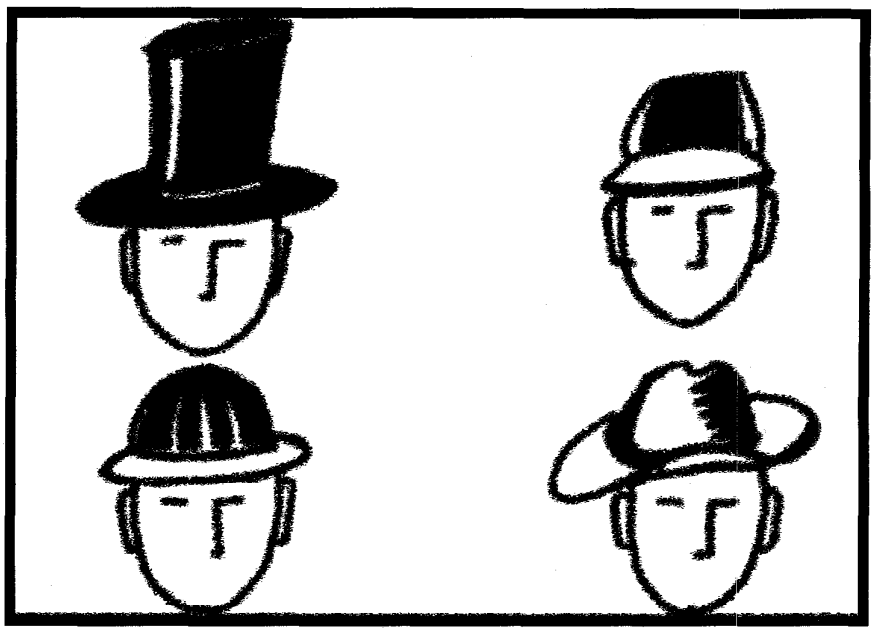
Scenario-Based Analysis of Software Architecture

Despite advances in clarifying high-level design needs, analyzing a system's ability to meet desired quality criteria is still difficult. The authors propose using scenarios to make analysis more straightforward. In their case study report, they analyze lessons learned with this approach.



RICK KAZMAN
University of Waterloo

GREGORY ABOWD
Georgia Institute of Technology

LEN BASS AND PAUL CLEMENTS
Software Engineering Institute,
Carnegie Mellon University

Analyzing a proposed software system to determine how well it meets quality criteria is a difficult process. Analysts may lack both a common understanding of high-level design and a fundamental understanding of quality attributes. Even with the recent focus on software architecture and the resulting clarification of some high-level design issues, there is still no satisfactory way to express and analyze context-dependent quality attributes like safety and portability.

To address this problem, we have developed the Software Architecture Analysis Method, an approach that uses scenarios to gain information about a system's ability to meet desired quality attributes. Scenarios—brief narratives of expected or anticipated system uses from both user and developer views—provide a look at how the system satisfies quality attributes in various use contexts.

0740-7459/96/$05.00 © 1996 IEEE

ARCHITECTURAL ANALYSIS: CRITICAL TOOL

Developers often resist contemplating architectural changes. Why should they bother when they are already under pressure to deliver existing features on time and within budget?

Unfortunately, the span of the development life cycle in which the system is created and delivered is not where most of the total system cost lies. The greatest cost is generally in system maintenance and the addition of new features. We believe, if done early on, architectural analysis can reduce that cost by revealing a design's implications. This, in turn can lead to an early detection of errors and to the most predictable and cost-effective modifications to the system over its life cycle. Moreover, architectural analysis can be done inexpensively, without detailed knowledge of the system.

Issues in software architecture are, by and large, not new. They date back at least to 1968 when Edsger Dijkstra pointed out that it pays to consider how to structure a program, not just how to make it compute the correct answer.[1] People who build large computer-based systems have been analyzing the allocation of function onto configurations for a long time.

Why focus on architecture? There are several reasons to consider an architectural view. First, architecture is often the first design artifact that represents decisions on how requirements of all types are to be achieved. As the manifestation of early design decisions, it represents design decisions that are hardest to change[2] and hence most deserving of careful consideration.

Second, an architecture is also the key artifact in successfully engineering a product line. Product line engineering is the disciplined, structured development of a family of similar systems with less effort, expense, and risk than would be incurred if each system were developed independently.[3]

Finally, the 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.

Architectural views. Software architecture describes "a high-level configuration of system components and the connections that coordinate component activities." There are several important aspects of this definition. First, even though we say "software" architecture, quite often high-level configurations describe functionality that will ultimately be performed by either software or hardware. Second, we say *a* high-level configuration rather than *the* high-level configuration because a system can comprise more than one type of component. Each decomposition will therefore have its own configuration.

A system may, for example, comprise a set of modules (in the sense David Parnas uses[4]) as well as a set of cooperating sequential processes, each of which resides in one or more modules. Both the process and module viewpoints are valid, and both are architectural in nature. However, they carry different information. From the process viewpoint, you can describe system interaction during execution in terms of how and when processes become active or dormant, pass or share data, or otherwise synchronize. From the module viewpoint, you 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.

The point is that different views support the analysis of different architectural qualities. When analyzing software architecture, you must be able to both represent an architecture in multiple ways and to understand which view is most important.

REFERENCES

1. E. Dijkstra, "The Structure of the 'T.H.E.' Multiprogramming System," Comm. ACM, Aug. 1968, pp. 453-457.
2. D. Parnas, "On the Design and Development of Program Families," IEEE Trans. Software Eng., Jan. 1976, pp. 1-9.
3. "The Domain-Specific Software Architecture Program," E. Mettala and M. Graham, eds., Tech. Report CMU/SEI-92-SR-9, Software Engineering Institute, Carnegie Mellon Univ., Pittsburgh, 1992.
4. D. Parnas, "On the Criteria for Decomposing Systems into Modules," Comm. ACM, Dec. 1972, pp. 1053-1058.

Although we developed SAAM for application early in design, we validated it in an analysis of several existing industrial systems. In this article, we describe one such application, a case study analyzing a commercial revision-control system based on RCS,[1] and review lessons learned.

WHY SCENARIOS?

Analysts can use a description of software system architecture to evaluate the system's ability to exhibit certain quality attributes. Software architecture describes a high-level configuration of system components and the connections that coordinate component activities. Different architectural views, such as module or process, provide different kinds of information, as the box above describes. A good understanding of system design at the architectural level makes it easier to detect design errors early on and easier to modify the system later.

However, evaluating an architecture to determine how the resulting system will satisfy properties or qualities such as modifiability or security is difficult. Quality attributes tend to be vague and provide little procedural support for evaluating an architecture. There are no simple (scalar) "universal" measurements for attributes such as safety or portability. Rather, there are only context-dependent measures, meaningful solely in specific circumstances. Safety benchmarks are a fine example. If there was 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.

Scenarios offer a way to review these vague qualities in more specific circumstances by capturing system use contexts. Scenarios have been widely used to compare design alternatives, but they have not been applied as we use them in SAAM. By using scenarios to express particular instances of each quality attribute a customer deems important, developers can analyze the architecture with respect to how well or how easily it satisfies the constraints imposed by each scenario.

Scenarios differ widely in breadth and scope. Our scenarios are one-sen-

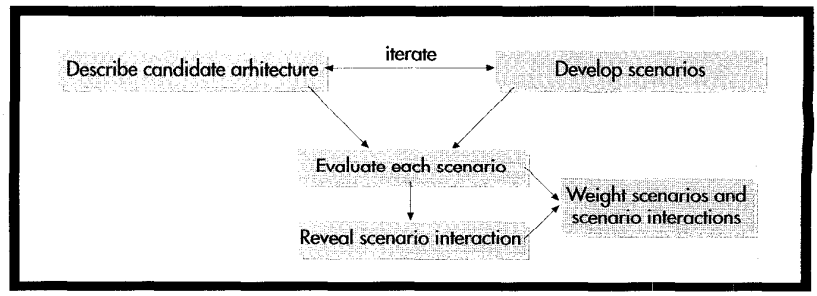


***Figure 1**: The five steps of SAAM and their dependencies.*

tence descriptions, probably closer to vignettes. We do, however, have an extensive scope, including scenarios for all roles that involve the system. We include not only the common role of system operator, but also the roles of system designer and modifier, system administrator, and others, depending on the domain. We believe this wide scope is important because design decisions may be made to accommodate any of these roles.

Scenarios also force designers to consider the future uses of, and changes to, the system. Instead of asking to what degree a system can be modified, designers must ask how the architecture will accommodate a particular change in the system, or how the architecture will accommodate a certain class of changes. Architectural analysis cannot give precise fitness measures. Scenarios, in effect, channel architectural analysis into an inspection of the architecture and focus attention on potential trouble spots.

SAAM ACTIVITIES

We initially developed SAAM to help us compare architectural solutions,[2] but the method evolved to a set of steps that provide a structured scenario-based architectural analysis. Not all our experience with architectural analysis has strictly followed these steps, but in all applications of SAAM, we used scenarios as the foundation for illuminating architectural properties. From these applications emerged a stable set of activities and interdependencies.

Figure 1 shows these activities and the dependencies among them. A subset of these activities or some variation of SAAM may be sufficient for some analyses.

Describe the candidate architecture. For any analysis, you must first have an architectural representation of the product with a well-specified semantic interpretation that describes what it means to be a component or a connector. The description should be in a syntactic architectural notation that all parties involved in the analysis can understand. The architectural descriptions should indicate the system's computation and data components, as well as all component relationships, sometimes called connectors, and their interfaces. Moreover, a software architecture may have more than one representation. We have used simplistic architectural primitives in our case studies and have not found them too limiting. A typical representation will distinguish between components that are active (transform data) and passive (store data) and also depict data connections (passing information between components) and control connections (one component enabling another to perform its function).

This simple lexicon provides a reasonable *static* representation of the architecture. You also need a description of how the system behaves over time, a *dynamic* representation of the architecture. This can be a natural-language specification of the overall behavior or some other more formal and structured specification.

Develop scenarios. Task scenarios should illustrate the kinds of activities the system must support and the kinds of changes analysts expect the system to undergo over time. In developing these scenarios, you should capture all important system uses. Scenarios should represent tasks relevant to different roles, or system stakeholders, such as end user/customer, marketer, system administrator, maintainer, and developer.

Evaluate each scenario. For each scenario, you should determine if the architecture can execute it directly, or if a change is required to execute it. If no architectural change is required, the scenario is *direct*. Otherwise, it is *indirect*. We are interested primarily in indirect scenarios because they represent the extra functions the architecture must satisfy; direct scenarios represent functions the architecture already has. Direct scenarios are interesting only insofar as they indicate a component's potential complexity.

For each indirect scenario, you should list the changes necessary for the architecture to execute the scenario and estimate the cost of performing the change in terms of, say, person-hours required or lines of code affected. Modifying the architecture means introducing a new component or connection or changing the specification of an existing component or connection. By the end of this stage, you should have a summary table that lists all direct and indirect scenarios. For each indirect scenario, you should describe the impact (set of changes) the scenario has on the architecture. We have found it sufficient to list the existing components and connections that must be altered, a rough cost measure of those alterations, and the new components and connections that must be introduced and their cost. SAAM does allow for more sophisticated cost functions if they are needed. A tabular summary is especially useful when comparing alternative architectural candidates because it lets you easily determine which architecture better supports a collection of scenarios.

Reveal scenario interaction. When two or more indirect scenarios necessitate changes to some component, they are said to interact. Revealing this scenario

interaction is important because you are exposing how the product's functionality is allocated in the design. Scenario interaction shows you which modules are involved in which tasks. A high degree of interaction may mean that the functionality of a particular component is poorly isolated. The designer can then see and correct this trouble spot. As we show later, the amount of scenario interaction is related to metrics such as structural complexity, coupling, and cohesion and so is likely to be strongly correlated with the number of defects in the final product.

The amount of scenario interaction is related to metrics such as structural complexity, coupling, and cohesion.

When determining scenario interaction, you identify scenarios that affect a common set of components. Scenario interaction measures the extent to which the architecture supports an appropriate separation of concerns.

Weight scenarios and scenario interactions. Once you have determined the scenarios, mapped them onto the structural description, and revealed all scenario interactions, the scenario implications should be clear. You should now weight each scenario and scenario interaction in terms of relative importance and use that weighting to determine an overall ranking. This is a subjective process that involves all stakeholders in the system. The weighting reflects the relative importance of the quality factors that the scenarios manifest.

In essence, you will have a collection of small metrics for comparing architectures rather than a single architectural metric. With these minimetrics (per-scenario analyses), you can use SAAM to compare competing architectures on a per-scenario basis. When candidate architectures outscore each other on different scenarios, it is up to you to determine which scenarios are most important and thus which architecture is most desirable. In other words, this step is impossible to perform without first understanding specific organizational requirements.

Dependencies. As Figure 1 shows, the first and second steps 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 the first and second steps is the role that direct scenarios play in helping you understand an architectural description. For example, they can help you determine static architectural connections and form more structured descriptions of an architecture's dynamic behavior.

Other dependencies may also exist. For example, when performing an architectural analysis of a system in its design phase, the evaluation of individual scenarios or scenario interaction may cause the designers to modify the architectural representation.

WRCS: CASE STUDY

To validate SAAM, we used it to analyze several existing systems. These applications were decidedly nonacademic. In fact, decision-makers based subsequent development or procurement actions on the outcome of our analysis. One of these applications involved a commercial version control/configuration management system based on RCS,[1] which we call WRCS here.

Because the analysis of WRCS involved all the steps of SAAM, we felt it was a good case study for evaluation. In the following description, we have slightly modified design details to protect the company's proprietary interests.

WRCS functions. WRCS lets developers of a "project" create archives, compare files, check files in and out, create releases, back up to old versions of files, and so on. "Project" means any group of related files that, when linked appropriately, form a finished product. These files might be source code, text, or digitized audio and video. WRCS tracks changes and performs managerial functions such as report production. It also lets multiple users work on the same project in their private work areas, which gives each developer the freedom to modify and test the system in isolation without disturbing other developers' work and without corrupting the primary copy of the system.

The company has integrated WRCS's functionality with several program-development environments. Users can access WRCS functions either through these tools or through WRCS's graphical user interface.

Applying SAAM. Some of the steps for this analysis may not be as complex in other analyses. As we noted earlier, organizational requirements and software development process maturity have much to do with how SAAM is applied.

Describe candidate architecture. There was no architectural description of WRCS, so we had to create one. This proved to be one of the most difficult

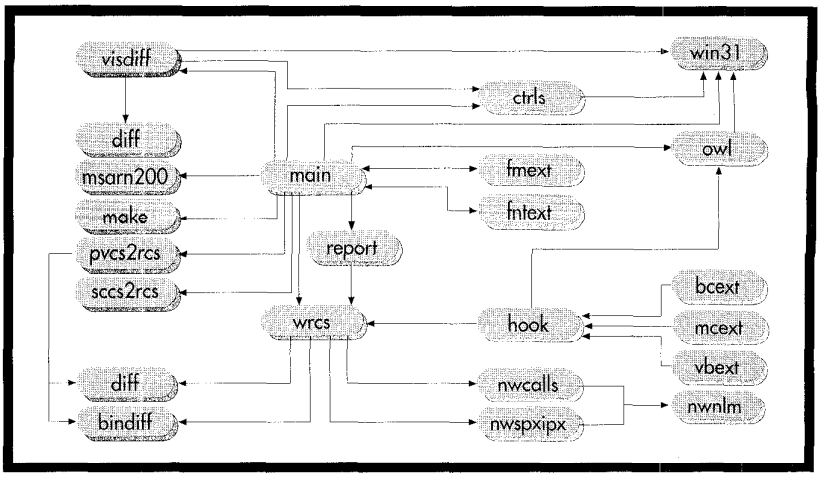


Figure 2. *Architectural representation of WRCS, in which the ovals are components (modules) and arrows are connections (calls).*

tasks in our analysis. We began by devising a way to elicit the required information. We had to group and analyze the information in a way that would help us construct an architectural diagram. Our sources were limited to the development team, the product's documentation, and the product itself in the form of executables and libraries. We had no access to the source code or the product's specifications (which was actually appropriate because software architecture is supposed to concern itself with a level of abstraction above code). In essence, our task was to reverse-engineer a finished product to get a design document. This is not unusual: Many mature software products either have no recorded architecture or the recorded architecture no longer matches the current product.

We arrived at the product's architectural description iteratively. At each stage we studied the product's existing description, the product, and the product documentation, and devised a new set of questions whose answers would help us clarify the current description. Consequently each new stage gave us more insight into the product and motivated new questions. Because we didn't have any previous representation we started with a gross listing of modules along with their basic relationships. We then iterated the representation and built the structure as we went. The process of eliciting scenarios also helped clarify the architecture, as we describe next.

Figure 2 shows the final architectural representation, which we arrived at after three iterations.

Develop scenarios. As we described earlier, the first and second steps are highly interdependent. While we were evolving an architectural description, we were continually developing scenarios that represented the various stakeholder roles in the system. For WRCS these roles were user, developer, maintainer, and system administrator.

Scenarios can be thought of as descriptions of tasks associated with system roles. We elicited a set of 15 tasks from the WRCS domain expert. The list of scenarios below reflects the more significant tasks. A complete evaluation of a more complex system than WRCS would involve dozens of scenarios.[3]

In creating scenarios, we collaborated with all the stakeholders in the system so that we could characterize all current and projected system uses of the system. The scenarios formed the basis for all further architectural evaluation.

For the user role, scenarios included

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

For the maintainer role, scenarios include

- *Port to another operating system.*
- *Modify the user interface in minor ways.* These included adding a menu item and changing the look and feel of a dialog box.

For the administrator role, scenarios included

- *Change access permissions for a project.*
- *Integrate WRCS functionality with a new development environment.* Attach it to Symantec C++, for example, so that a Symantec C++ user can check files in and out from within the Symantec development environment.
- *Port to a different network-management system.*

Evaluate each scenario. Once we created the scenarios, we classified them as direct or indirect. This was our first indication of how easily the architecture would satisfy the scenarios. Our classification was based on how much work it would take to satisfy the scenario. For example, if we could reconfigure a product's toolbar within the product (the second scenario given for the user role), we classified the scenario as direct. However, if we had to modify the architecture to achieve this change, the scenario was indirect. The more indirect scenarios, the more required changes, and the less desirable the architecture with respect to those scenarios.

In estimating the cost of indirect scenarios, we also evaluated the proposed solution. For the scenario of reconfiguring a product's toolbar, for example, the cost would be minimal if developers simply modified an ASCII resource file and restarted the product. If they opted to change an internal table and recompile, the cost would be moderate. If they had to dramatically restructure the user interface code, the

TABLE 1
RESULTS OF SCENARIO EVALUATION FOR WRCS

Scenario	Direct/Indirect	Required Changes
Compare new binary file representations	Indirect	Modify `diff` to make the comparison and `visdiff` to display the results of the comparison.
Configure the product's toolbar	Direct	—
Manage multiple projects	Indirect	Modify `wrcs`, `main`, `hook`, and `report`.
Port to another operating system	Indirect	Modify all components that call win31, specifically, `main`, `visdiff`, and `ctrls`. If the target operating system does not support `owl` then port either `owl` or all components that call `owl`, specifically `main` and `hook`. If the new operating system is not supported by Novell's software, modify WRCS to work with a new networking environment.
Modify the user interface in minor ways	Indirect	Modify one or more components that call the win31 API, specifically `main`, `diff`, and `ctrls`.
Change access permissions for a project	Direct	—
Integrate with a new development environment	Indirect	Modify `hook` and add a module along the lines of `bcext`, `mcext`, and `vbext` that connects the new development environment to `hook`.
Port to a different network-management system	Indirect	Modify `wrcs`.

TABLE 2
SCENARIO INTERACTION BY MODULE FOR WRCS

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

cost would be considerable.

Table 1 lists the scenario subset described earlier and the changes required to the architecture shown in Figure 2.

Reveal scenario interactions. Table 2 shows the number of changes required in each module. In this table we take into account all 15 scenarios elicited in the WRCS analysis, not just the subset presented earlier. Because each scenario imposes a single change to the architecture, the number of changes per module indicates the level of indirect scenario interactions for that module.

To highlight scenario interactions, we used a fish-eye view of the architecture, in which the module size is proportional to the number of interacting scenarios that affect it. Figure 3 shows this representation. This contrasts to the view in Figure 2, in which all modules are of equal size. Figure 3 shows that the `wrcs` module clearly has the most scenario interactions, so it is a natural focus for future development efforts. `Main` and `hook` also suffer from high scenario interaction, with `visdiff` and `ctrls` having slightly less.

This view shows at a glance the most architecturally significant features and where designers and developers should allocate time and effort. We found it a highly effective device for focusing communication among WRCS team members.

Weight scenarios and scenario interactions. As we described earlier, prioritizing scenarios depends on the organizational requirements and preferences of the SAAM user. In this case, the organization was most concerned with the implications of porting WRCS to other platforms. Most of the work lay in porting the user interface. Updates of WRCS to include new customer-requested features were also driving design decisions.

Results. Our analysis identified several severe limitations in achieving portability and modifiability. We ended up recommending a major redesign of the system. Had WRCS designers gone through this analysis before the system was implemented, they might have avoided many of the problems the developers and maintainers now face.

On an organizational level, the analysis had 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. The WRCS team itself, however, regarded this evaluation as just an academic exercise. We attribute this to the broad view of the software development life cycle that senior developers and managers share. They have the perspective to understand that much of the life cycle is devoted to maintenance and feature enhancements. Any effort that aids in improving a product's support for extra-functional qualities is significant. WRCS developers were more concerned with meeting the next release deadline or with finding a bug. They tend to view the contemplation of major architectural changes as a luxury. As one senior manager put it, "They have features to implement!"

This is why architectural analysis must be done early. Otherwise, it will never be done or, if done, will be meaningless.

We also recognized that SAAM gave us insight into the product's capabilities that we could not easily get from code and document inspections. In a simple, straightforward, and cost-effective way, it exposed specific product limitations. Furthermore, we were able to do this beginning with only scant knowledge of the product's internal workings.

Perhaps most important of all, the results were frustrating to the developers. The realization of the problems architectural analysis could have avoided has made the organization change its development practice. It has convinced management that developers need architectural analysis up front.

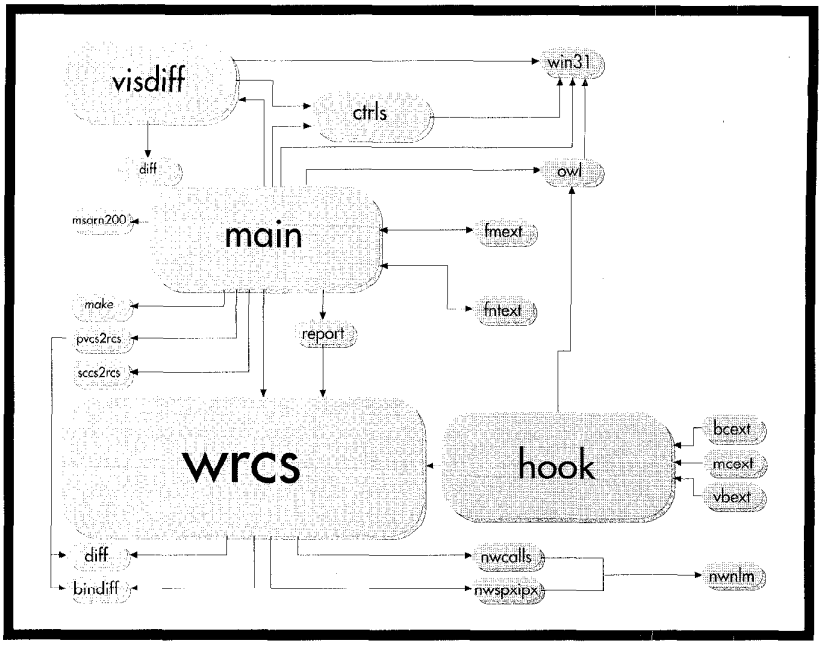


Figure 3. Fish-eye representation of WRCS. This representation helps illuminate problem modules, ones that contain a high degree of scenario interaction.

LESSONS LEARNED

We have performed architectural evaluations on a dozen small- to medium-sized software architectures and four large industrial systems. We have begun to see universal benefits to using SAAM and to recognize patterns in architectural analysis.

Enhanced communication. The use of scenarios has proven important in both team communication and communication between the team and upper-level management. It is difficult to get agreement on an appropriate set of scenarios, so the system's stakeholders must talk to each another and reach consensus. A collection of scenarios—particularly scenarios that have caused problems for similar systems—can provide a benchmark with which to evaluate new designs.

Visualization has been a powerful tool in enhancing that communication, particularly in relating design problems to the stakeholders. The visualization of an architecture, emphasizing scenarios and scenario interaction, focuses attention on more critical areas, effectively proposing topics for discussion.

Improvement of traditional 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. That is, if a single component's functionality breaks down or is modified, it is less likely to affect the functionality of other components. What does this mean in terms of a SAAM analysis?

Low coupling means that a single scenario doesn't affect many structural components. High cohesion means that structural components are not host to scenario interactions. This correspondence implies that you can use architectural analysis to determine coupling and cohesion.

Architectural metrics such as coupling and cohesion have been called crude instruments of measure. SAAM improves on these metrics by letting you measure them with respect to a particular scenario or set of scenarios. Consequently, these measures become much sharper and more meaningful. For example, in traditional coupling, two components are coupled whether they communicate regularly or only once, for example, at initialization. With SAAM, on the other hand, we consider only component coupling that is relevant to the scenario at hand, not the coupling of all components.

Similarly, we are interested in

expending effort to ensure high cohesion only if the portion of the architecture under consideration is the site of many proposed changes. Put another way, low cohesion is a problem only for parts of the architecture that have scenario interactions.

If part of the architecture has a high structural complexity, it is simply labeled "bad."

Structural complexity (based on dataflow among components) is another measure that has been criticized as crude because it does not consider predicted changes to the architecture. If part of the architecture has a high structural complexity, it is simply labeled "bad." Scenarios, on the other hand, will tease such cases apart by showing that the structural complexity is of concern only in areas of the architecture that are candidates for change.

Proper description level. Viewing software from a higher level of abstraction—that of software architecture—means that you must choose an appropriately high level of description. The level of description you choose is dictated by the scenarios. We learned this lesson as we iterated through our three versions of the architectural representation for WRCS and in our other applications of SAAM.

When the architecture has been given its initial structural description, you should map the scenarios onto the structure. In particular, for each indirect scenario, you should highlight the components and connections that will be affected by the change the scenario implies. This mapping serves three purposes: it guides architectural evaluation, it aids in validating scenario interaction (a difficult process without this step[3]), and it drives you to the appropriate level of architectural representation.

Consider a case in which multiple indirect scenarios affect a single module. There are three possibilities:

- *The scenarios are of the same class.* This would make them variants of the same basic scenario; the same kind of system change is required. Scenarios of the same class should cluster in the same module, so if this is the case, it's a good sign. It means the system's functionality is sensibly allocated. That is, the architecture has high cohesion with respect to this scenario class.
- *The scenarios are of different classes and you can further divide the module.* Remember that scenarios dictate the proper level of architectural description. In this case, the module might actually be composed of three functions, each of which deals neatly with one scenario. You are merely refining the level at which the software architecture is presented.
- *The scenarios are of different classes and you cannot further divide the module.* Here you have a potential problem area. If scenarios of different classes are affecting the same module, the architecture may not be appropriately separating concerns, and so this module is a site of scenario interaction.

Efficient scenario generation. Scenario generation is much like software testing: You cannot prove you have a sufficient number of test cases, but you can determine a point at which adding new test cases yields only negligible improvement. How much testing you do also depends on the resources available. Thus, there are two possible answers to "How many scenarios is enough to adequately exercise the architecture?" The simple one is "Stop when you run out of resources." The more complex and meaningful answer is "Stop generating scenarios when the addition of a new scenario no longer perturbs the design."

One way to minimize the scenarios needed is to group scenarios into equivalence classes. However, this merely generates a new question: "How do you know if the scenarios are appropriately grouped into classes?" You can look at this problem another way: All domain experts should cluster scenarios the same way. If they do not, they must have additional scenarios in mind, and the analysis must make them explicit.

As we developed SAAM, we conducted many industrial and academic case studies in scenario-based architectural analysis. Applications include user interface development environments,[2] Internet information systems,[4] key word in context systems,[5] embedded audio systems, object request brokers, and visual debuggers. In almost all instances, developers benefited from the results. The primary benefits were a deepened understanding of the system; enhanced communication both among the development team and among developers, managers, and customers; the ability to make high-level comparisons of competing designs and to document those comparisons; and the ability to consider and document the effects of sets of proposed system changes.

We have also applied the completed SAAM to large-scale industrial systems, including

- *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 company wanted some

assurance that the system architecture was going to provide for the generic satellite-based multiuser applications they anticipated developing in the near and long term. As a result of our analysis, the company decided not to purchase the system, avoiding an investment of tens of millions of dollars.

♦ *Air traffic control system.* This investigation involved assessing 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.[6]

Our future plans include extending the scope of this technique from primarily maintainability and modifiability measures to execution-based measures such as performance and reliability. This will let us consider principled architecture trade-off analysis, in which you evaluate competing designs with respect to a palette of scenarios that represent different quality attributes. Finally, SAAM analyses require a lot of "bookkeeping" and much of the information collected can be reused. For example, scenarios are an asset that can be reused in analyzing a family of related systems. We are currently building a tool to support the collection, analysis, and reuse of architectural information.[7]

ACKNOWLEDGMENTS

We thank Mauricio de Simone and Linda Northrop for their efforts and contributions in creating this article. This work was sponsored in part by the National Sciences and Engineering Research Council of Canada and the US Department of Defense.

REFERENCES

1. W. Tichy, "RCS—A System for Version Control," *Software-Practice & Experience*, July 1985, pp. 637-654.
2. R. Kazman et al., "SAAM: A Method for Analyzing the Properties of Software Architectures," *Proc. Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1984, pp. 81-90.
3. P. Gough et al., "Scenarios—An Industrial Case Study and Hypermedia Enhancements," *Proc. Int'l Symp. Requirements Eng.*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 10-17.
4. R. Kazman et al., "An Architectural Analysis Case Study: Internet Information Systems," Tech. Report CMU-CS-TR-95-151, Carnegie Mellon Univ., Pittsburgh, 1995.
5. P. Clements et al., "Predicting Software Quality by Architecture-Level Evaluation," in *Component-Based Engineering*, A. Brown, ed., IEEE CS Press, Los Alamitos, Calif., 1996, pp. 19-25.
6. A. Brown, D. Carney, and P. Clements, "A Case Study in Assessing the Maintainability of a Large, Software-Intensive System," *Proc. Int'l Symp. and Workshop Systems Eng. Computer-Based Systems*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 240-247.
7. R. Kazman, "Tool Support for Architectural Analysis and Design," *Joint Proc. SIGSOFT '96 Workshops*, ACM Press, New York, pp. 94-97.

Rick Kazman is a senior software engineer at the Software Engineering Institute of Carnegie Mellon University. During the work reported here he was a faculty member in computer science at the University of Waterloo. His primary research interests are software engineering (software architecture, design tools, and program visualization), human-computer interaction (interaction with 3D environments), and computational linguistics (information retrieval).

Kazman received a BA in English and an MMath in computer science from the University of Waterloo, an MA in English from York University, and a PhD in computational linguistics from Carnegie Mellon University. He is a member of the IEEE Computer Society and ACM.

Gregory Abowd is an assistant professor in computing at the Georgia Institute of Technology, where he is a member of the Software Systems Design Group and the Graphics, Visualization and Usability Center. He is also director of the institute's Multimedia Lab. His research interests include software architecture, formal methods, software engineering for interactive systems, and future computing environments.

Abowd received a BS in mathematics from the University of Notre Dame and an MSc and a DPhil in computation from the University of Oxford, where he attended as a Rhodes scholar. He is a member of the IEEE Computer Society and ACM.

Len Bass is a senior software engineer at the SEI, where he is working on user-interface design issues for wearable computers and techniques for software architectural analysis. Bass was the designer of the Serpent User Interface Management System. He organized a group that defined a user-interface software reference model that is an industry standard, and headed a group that developed a software architecture for flight training simulators that the US Air Force has adopted as a standard. He has written or edited five books and many papers on software engineering, human-computer interaction, databases, operating systems, and the theory of computation.

Bass received a BA and an MA in mathematics from the University of California at Riverside and a PhD in computer science from Purdue University. He is chair of the International Federation of Information Processing's Technical Working Group on User Interface Engineering.

Paul Clements is a senior software engineer at the SEI, where he works on the Software Architecture Technology Initiative, an effort intended to focus and facilitate the community's work on software architectures for real-world practitioners. He has also worked for the US Naval Research Laboratory, where he led the Software Cost Reduction, or A-7 project, an experiment in the software engineering of mission-critical reactive systems.

Clements received a BS in mathematics and an MS in computer science from the University of North Carolina at Chapel Hill and a PhD in computer science from the University of Texas at Austin.

Address questions about this article to Kazman at The Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890; kazman@sei.cmu.edu.