Pattern-Based Architecture Reviews

Harrison, Neil B.; Avgeriou, Paris

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

[Link to publication in University of Groningen/UMCG research database](#)

Pattern-Based Architecture Reviews

Neil B. Harrison, Utah Valley University

Paris Avgeriou, University of Groningen

// A lightweight architecture review process can help achieve systemwide quality attributes, offering an alternative to heavyweight architecture reviews. //

QUALITY ATTRIBUTES DESCRIBE a system's usability, maintainability, performance, and reliability (though not its functionality). They can drive customer satisfaction and differentiate one product from another.

Quality attributes are systemwide, so architecture has a huge impact on them. Paul Clements and his colleagues stated, "Modifiability, performance, security, availability, reliability—all of these are precast once the architecture is laid down. No amount of tuning or clever implementation tricks will wring any of these qualities out of a poorly ar-

chitected system."[1] Unfortunately, this also means these qualities can't be fully verified until the system is basically complete and ready for system-level verification. Nevertheless, it's important to identify relevant quality issues prior to system testing.

Architecture reviews are a possible solution: they find potential problems,[2–4] particularly those related to quality attributes. However, despite their demonstrated benefits, many projects are unable or unwilling to use them. These projects tend to be characterized by

- short schedules, possibly including repeated development episodes with very short cycles;
- tight deadlines, leaving little or no time for activities other than production;
- neglected documentation, especially internal documentation such as architecture documents;
- frequently changing technological or user requirements; and
- small teams.

These characteristics can lead to a focus on producing merely "working" software or "getting the product out the door"—other activities are lower priority. For lack of a better term, we describe these projects as *production-focused*. Many such projects (though not all) follow practices found in agile and lean software development methodologies.[5–7]

We've developed a lightweight architecture review process suitable for production-focused projects. It identifies architecture patterns and examines their effects on quality attributes. We used it to review nine projects; it not only uncovered important architectural issues but also improved the development team's understanding of the architecture.

Architecture Reviews and Production-Focused Projects

Many software architecture review practices examine quality attributes in architectures in depth.[8] However, they have key incompatibilities with production-focused projects, including the following:

- *Manpower.* Production-focused projects generally have only enough

resources to simply write the software. The approximate cost of an ATAM-based architecture evaluation for even a small project is 32 staff days.[1] The average cost of an architecture review in AT&T is 70 staff days.[2]

- *Price.* Published architecture review methods are generally expensive.
- *Architectural documentation.* Even though many architecture review methods base much of their analysis on it,[1] production-focused projects have little architectural documentation—a widely recognized problem.[9]
- *Requirements.* Architecture review methods require detailed requirements specification and corresponding stability—a process that takes two to six weeks.[3] Extensive preparation hinders a review from being held in response to changing requirements.

These incompatibilities lead project managers to not review their architecture, forgoing the inherent benefits. However, a lightweight review process that addresses these incompatibilities still gives projects some of the benefits of architecture review.

Pattern-Based Architecture Reviews

A pattern-based architecture review (PBAR) is a lightweight evaluation method based on software architecture patterns, which are generalized solutions to recurring design problems. This review method provides a proven approach on using the pattern solution, including the consequences of applying the solution.[10] Although the most well-known software patterns are object-oriented design patterns, we're more concerned with those that deal with a system's architecture.[11]

Architecture patterns focus on the entire software system's design and contain its high-level modular decom-

position.[11–13] Applying a given architecture pattern can make it easier or harder to implement certain quality attributes. For example, the *layers* pattern divides the system into distinct layers so that each one provides a set of services to the layer above and uses the services of the one below.[11] This structure supports fault tolerance in that you can use layers to implement transactions for easy roll-back in the case of failure. However, this pattern requires requests to pass through multiple layers, which can hurt performance.

PBAR leverages patterns' relationships with quality attributes to create a review that's compatible with production-focused projects. It addresses the key incompatibilities between these projects and traditional architecture reviews:

- *PBAR requires only a small amount of time and effort.* This makes it more compatible with small projects that focus on writing production code.
- *PBAR doesn't require architecture documentation.* Instead, it finds the architecture patterns in use and leverages any existing documentation to make inferences about how quality attributes will be implemented within the context of those patterns.
- *Production-focused projects accommodate changing requirements.* PBAR has a short preparation time, a short review, and can return feedback to a project within one or two days. This allows it to

be used on short notice in response to changing requirements.

The essential elements of the review are the same as in heavyweight architecture reviews, but are simpler and more focused.

Resources and Planning
The reviewer should have expertise in architecture, architecture patterns, quality attributes, and a general knowledge of the domain. The reviewer should come from outside the team, in order to provide a fresh perspective on the system's architectural design—this task is more of an audit than an internal review.

Scheduling the Review
All developers, as well as other interested stakeholders, should be invited to a review that should be scheduled early in development, once the system's basic structure is known. Participants don't need formal preparation, but the reviewer should study any architecture and requirements documentation available, such as user stories or use cases.

The Review Meeting and Follow Up
The review is a face-to-face meeting during which the following steps should be iteratively executed:

> The pattern-based architecture reviews method leverages patterns' relationships with quality attributes.

1. Identify the system's most important quality attributes and discuss them. Go through the user stories and walk through scenarios that are relevant to quality attributes.
2. Discuss the system's architecture (even draw it on a whiteboard).
3. Identify the architecture patterns

used. (The reviewer does this, but other participants who know architecture patterns can help.) The main technique is to match the system's structure to the patterns' structure. You want to find established patterns rather than new ones because the impact on quality attributes is already understood for established architecture patterns.

4. Examine the architecture and qual-

> You can hold a PBAR as soon as a walking skeleton is implemented, unlike a traditional review.

ity attributes together to determine each pattern's effects on the system's quality attributes. Review past scenarios, implementations, and where in the architecture the implementation occurs. Use existing pattern documentation to look for matches (and mismatches) between the patterns and quality attributes.

5. Identify and discuss quality attribute issues, including quality attributes not addressed or adequately satisfied, patterns not used that might be useful, or potential conflicts between patterns used and quality attributes. For example, a layered architecture is often incompatible with a high-performance requirement.

After the review, the reviewer should provide a summary for the entire team. This should go quickly, as most issues will have already surfaced during the review meeting itself. (Our meetings have all lasted well under an hour.)

Reviews and Production-Focused Practices

Table 1 shows typical practices of production-focused projects and how

both PBAR and traditional heavy-weight reviews accommodate them. These practices are also found in many agile and lean methodologies. Note that not all production-focused projects follow agile methodologies, and conversely, not all agile projects are production-focused.

Frequent Releases

To increase flexibility, projects can have frequent internal or external releases. An architecture review should fit into this time: both the planning and the review itself should be short. Because participants don't need to prepare, PBAR can be flexibly scheduled. Its short duration is only a minor disruption in even a very short release cycle.

Changes for User Needs

Comprehensive architecture reviews are based on requirements specifications (generally written). But because requirements often change, the review's utility is reduced. PBAR focuses on quality attributes, which are likely to be more stable than functionality requirements.

Lightweight Documentation

Traditional reviews tend to be based on comprehensive architecture documentation, but it can simply be too much work for a project to produce it. PBAR is a lighter-weight alternative in these cases.

Walking Skeleton

A walking skeleton is an early end-to-end implementation of the architecture, often used as prototyping to help prove

architectural concepts. An ideal time for an architecture review is at the completion of a walking skeleton. Because of the small preparation time and effort needed, you can hold a PBAR as soon as a walking skeleton is implemented, unlike a traditional review, which needs considerable planning and upfront work.

Experiences with PBAR

We used PBAR on nine projects. Although roughly half were student software engineering capstone projects, all were real projects with real customers. Of these reviews, six were highly successful, one was partially successful, and two were unsuccessful. The partial success and two failures have helped us refine the process. Table 2 summarizes projects and the results; the "Major issues" column includes significant incompatibilities between the architecture and important quality attributes.

Most projects followed the bulk of the practices described earlier. All had high developer communication and high informal communication with the customer. Most had little or no architecture documentation, and didn't document or even use architecture patterns. All had frequent integrations, and a few had frequent releases. Most projects considered changing user needs and managed a flexible prioritized list of features. A few explicitly created walking skeletons.

Participants were positive about the review and its results; some were downright enthusiastic. Their feedback revealed four main benefits from the reviews:

- *Basic quality attribute issues.* The PBARs uncovered, on average, nearly four issues per project, one of which was major. In one case, the architecture used the *layers* pattern, but to improve performance, it offered a way to bypass the layers—a separate path through the system.

Common practices of production-focused projects and architecture reviews.

Production-focused practice	PBAR	Traditional reviews
Frequent releases[5–7]	Can be scheduled between early releases; a short review-feedback cycle fits well in small release windows	Not practical between releases; long planning-review-feedback time can cut across releases
Changes for user needs[5,7]	Focuses on quality attributes (which are more stable than functional requirements); allows features to change	Requires stability of requirements, including functional requirements
Lightweight documentation[5,6]	Requires no special documentation; leverages knowledge in patterns about architecture-QA issues	Encourages extensive architecture documentation; may require some to be written for review
Walking skeleton[6]	Can be scheduled in response to walking skeleton being implemented	Requires calendar-based scheduling due to need for extensive planning

Pattern-based architecture reviews.

System	Size	Project phase	Project description	No. of issues found	No. of major issues	No. of major issues resolved	Effort (in staff hours [reviewer/team])
A	Large	Implementation	Streaming data manipulation and analysis	3	1	0	5 (5/0)
B	Medium	Architecture	Computer-controlled process control	4	1	0	11 (6/5)
C	Small	Postrelease	Embedded GPS platform application	2	0	0	6 (4/2)
D	Small	Early implementation	Web-based time-tracking system	7	1	1	8 (3.5/4.5)
E	Small	Early implementation	Distributed subscription management system	3	2	1	9.5 (3.5/6)
F	Small	Early implementation	E-commerce inventory management system	3	1	1	8 (3.5/4.5)
G	Small	Early implementation	Android phone application	3	1	1	7.5 (3.5/4)
H	Small	Early implementation	Web-based game platform	5	0	0	7.5 (3.5/4)
I	Small	Early architecture	Web-based business process support system	0	0	0	4 (2/2)

In another case, the review revealed that the user interface design (based on existing software) was arcane and difficult to extend.

- *Team understanding about architecture.* Two comments were, "The review helped everyone see the whole picture," and, "The review helped clarify and unify the vision of the system."

- *Team understanding about quality attribute requirements.* For example, one team knew the system was reliable but needed further clarification. During the review, we determined that the system didn't need to be up continuously, but it did need to handle certain failure cases.

- *Team members knowing more about software architecture itself.* Through the PBAR process, teams

learned about architecture patterns and their relationship with quality attributes.

Obviously, these benefits must be weighed against the costs, but fortunately the cost was very low—the total effort for all participants was under two staff days. The short preparation time lets teams use the reviews in reaction to changing requirements. Although none

full participation and early in the development cycle, yet not so early that requirements aren't yet understood. Finally, a person with strong expertise in both architecture and architecture patterns must conduct them.

A Detailed Example

In order to illustrate the PBAR process and its benefits, we take one of the reviews and describe it in more detail.

> PBAR [is] useful even when the architecture documentation is entirely nonexistent and requirements are only sparsely documented.

of the reviews we did were held specifically in response to changing requirements, in some cases they were scheduled on short notice (a week or less). The only real complaint we heard was about the timing of the review—most were done while development was well along, and several participants wished the review had happened earlier.

If the participants found the reviews useful and acted on the issues identified, we succeeded. In six out of nine cases, this was true. Possible factors contributing to the three failures include the following:

- The issues identified had already been acted upon.
- We didn't receive confirmation of results, possibly because the review was done offline.
- The review wasn't completed, possibly because the reviewer was a novice architect. (In one particular case, requirements hadn't yet been established with the user, making it impossible to review the architecture against the requirements.)

Unsuccessful reviews teach us that reviews must be done with the team's

The project we studied was a student capstone project, so the students had no time for a lengthy project review. The small team of three developers followed no particular methodology, with few written requirements and no written architecture documentation. An additional challenge was that the project was an Android application, and the Android software development kit was very new at the time and under constant change; this affected feature development and implementation.

We began the review by discussing the functional and quality attribute requirements. We walked through scenarios to help us understand the four most important quality attributes, which were usability, security, reliability (fault tolerance), and extensibility. This was especially helpful for exploring fault tolerance. We then discussed the architecture and drew it on a whiteboard, using boxes and lines to represent components and connectors. A team member took notes, so at the end of the review, the team had some architecture documentation. We identified two architecture patterns: *peer-to-peer* and *shared repository*.[13]

We identified three issues with the

quality attributes, one of which was significant, and discussed ways to resolve the issues, identifying three measures the team could implement to do so. Len Bass and his colleagues call these *tactics*.[4] We annotated the architecture diagram with notes about where these tactics would be implemented, thus giving the team a "map" of how to implement them. The review took less than two hours.

The team noted specific benefits to the review, such as

- producing some architecture documentation;
- increasing their understanding of the architecture;
- increasing their understanding about the project's quality attribute requirements; and
- identifying some issues with proposed solutions.

This experience demonstrated that PBAR is useful even when the architecture documentation is entirely nonexistent and requirements are only sparsely documented.

Through these experiences using PBAR, we learned some important overall lessons about how to make PBAR as successful as possible. The architecture reviewer must come from outside the project. This is the case with all types of reviews and similar to the rationale for pair programming—a separate set of eyes can detect problems that project members can't. Having a team of two reviewers is better still.

Moreover, the review should be done as early as possible once enough of the architecture is in place to hold a meaningful review. It's important to note that because of PBAR's lightweight nature, it can be done very early, even before the architecture has solidified. However, if the quality attri-

bute requirements aren't yet solidified, the review is likely to fail.

What if the architects didn't use patterns in their architecture? This was the case in most of the reviews we conducted. But because architecture patterns are almost always present,[14] the review can proceed normally, and patterns will be identified.

Alas, PBAR won't find all the issues that a traditional architecture review will. Instead, it offers a trade-off: a review process that requires little time and effort and that can work even with little architectural documentation, certainly more so on an agile project when a heavyweight review process isn't used at all. PBAR finds incompatibilities between architecture patterns used and important quality attributes (for instance, performance versus layers, or fault tolerance versus pipes and filters); it won't find obscure problems such as performance issues from complex interactions of components.

Another limitation is that the reviewers must be well versed in architecture, architecture patterns, quality attributes, and tactics. This is like traditional reviews: reviewers need similar expertise, although architecture pattern knowledge isn't as critical. The key challenge for many organizations will be finding reviewers with sufficient expertise.

Nearly all the projects that have used PBAR so far are very small, which might not involve the same demands as larger industrial projects. Although users consider this lack of experience to be a limitation, we expect that PBAR would continue to be successful. ⑨

ABOUT THE AUTHORS

NEIL HARRISON is an associate professor of computer science at Utah Valley University. His research interests include software patterns, effective organizations, and software testing. Harrison received a PhD in software engineering from the University of Groningen, the Netherlands. He's a coauthor of *Organizational Patterns of Agile Software Development* (Prentice Hall, 2004). Contact him at Neil.Harrison@uvu.edu.

PARIS AVGERIOU is a professor of software engineering at the University of Groningen, the Netherlands. His research interests concern software architecture with a strong emphasis on architecture modeling, knowledge, evolution, and patterns. Avgeriou received his PhD in software engineering from the National Technical University of Athens, Greece. Contact him at paris@cs.rug.nl.

References

1. P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2001.
2. G. Abowd et al., *Recommended Best Industrial Practice for Software Architecture Evaluation*, tech. report CMU/SEI-96-TR-025, Carnegie Mellon Univ., Software Eng. Inst., 1997.
3. J.F. Maranzano et al., "Architecture Reviews: Practice and Experience," *IEEE Software*, vol. 22, no. 2, 2005, pp. 34–43.
4. L. Bass et al., *Risk Themes Discovered through Architecture Evaluations*, tech. report CMU/SEI-2006-TR-012, Carnegie Mellon Univ., Software Eng. Inst., Sept. 2006.
5. K. Beck and K. Andrus, *Extreme Programming Explained: Embrace Change*, 2nd ed., Addison-Wesley, 2004.
6. A. Cockburn, *Crystal Clear: A Human-Powered Methodology for Small Teams*, Addison-Wesley, 2004.
7. M. Poppendieck and T. Poppendieck, *Implementing Lean Software Development: From Concept to Cash*, Addison-Wesley, 2006.
8. L. Dobrica and E. Niemelä, "A Survey on Software Architecture Analysis Methods," *IEEE Trans. Software Eng.*, vol. 28, no. 7, 2002, pp. 638–653.
9. R. Kazman and L. Bass, "Making Architecture Reviews Work in the Real World," *IEEE Software*, vol. 19, no. 1, 2002, pp. 67–73.
10. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
11. F. Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns*, vol. 1, Wiley, 1996.
12. N. Harrison, P. Avgeriou, and U. Zdun, "Using Patterns to Capture Architectural Decisions," *IEEE Software*, vol. 24, no. 4, 2007, pp. 38–45.
13. P. Avgeriou and U. Zdun, "Architectural Patterns Revisited: A Pattern Language," *Proc. 10th European Conf. Pattern Languages of Programs* (EuroPLoP 05), Butterworth-Heinemann, 2005, pp. 1003–1034.
14. N. Harrison and P. Avgeriou, "Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation," *Proc. 7th Working IEEE/IFIP Conf. Software Architecture* (WICSA 08), IEEE CS Press, 2008, pp.147–156.