Leveraging Architecture Patterns to Satisfy Quality Attributes

Neil B. Harrison and Paris Avgeriou

Department of Mathematics and Computing Science, University of Groningen, Groningen, The Netherlands
harrisne@uvsc.edu, paris@cs.rug.nl

Abstract. Architectural design has been characterized as making a series of decisions that have system-wide impact. These decisions have side effects which can have significant impact on the system. However, the impact may be first understood much later; when the system architecture is difficult to change. Architecture patterns can help architects understand the impact of the architectural decisions at the time these decisions are made, because patterns contain information about consequences and context of the pattern usage. However, this information has been of limited use because it is not presented consistently or systematically. We discuss the current limitations of patterns on evaluating their impact on quality attributes, and propose integrating the information of patterns' impact on quality attributes in order to increase the usefulness of architecture patterns.

Keywords: Software Architecture, Architecture Patterns, Quality Attributes.

1 Introduction

One of the most challenging aspects of software design is creating a system that provides the quality attributes needed by the users. Quality attributes are characteristics of the system that are non-functional in nature. Typical quality attributes include reliability, usability, and security.

Because quality attributes are system-wide, their implementation must also be system-wide: satisfaction of a quality attribute requirement cannot be partitioned into a single module or subsystem. Thus, a system-level vision of the system is required in order to ensure that the system can satisfy its quality attributes. One of the primary purposes of the architecture of a system is to create a system design to satisfy the quality attributes. Wrong architectural choices can cost significant time and effort in later development, or may cause the system to fail to meet its quality attribute goals.

Designing an architecture so that it achieves its quality attribute requirements is one of the most demanding tasks an architect faces [3]. One reason is that the architect needs a great deal of knowledge about the quality attributes and about approaches to implementing systems that satisfy them. Yet there are many quality attributes; the ISO 9126 standard lists six primary and 21 secondary quality attributes [14]. In addition, quality attributes often interact – changes to the system often have repercussions on quality attributes elsewhere. Broad knowledge about how to manage tradeoffs among arbitrary quality attributes does not yet exist [2]. Requirements may not be sufficiently specific and are often a moving target. Finally, the consequences of decisions made are often overlooked [4]. As a result, architectural rework is common.

Architecture patterns are a viable approach for architectural partitioning, and have a well-understood impact on quality attributes [20]. However their application has been rather limited due to a number of factors [11]. We propose the systematic use of architecture patterns to help the architect satisfy quality attributes, and thus reduce the risk of later rework. We demonstrate that patterns can help the architect understand the impact architectural decisions that might be overlooked. We explore why patterns have been limited in large-scale industrial application. As an initial step to overcome this limitation we have analyzed several architecture patterns with respect to their impact to key quality attributes, as a means to leverage the knowledge in the patterns.

2 Architectural Decisions

The process of architectural design has been characterized as making a series of decisions that have system-wide impact. Most architectural decisions have multiple consequences, or as Jansen and Bosch put it, result in additional requirements to be satisfied by the architecture, which need to be addressed by additional decisions [15]. Some are intended, while others are side effects of the decision.

Some of the most significant consequences of decisions are those that impact the quality attributes of the system. Garlan calls them key requirements [10]. This impact may be the intent of the decision; for example, one may choose to use a role-based access control model in order to satisfy a security quality attribute. Other impacts may be side effects of different decisions. For example, the architect may adopt a layered architecture approach, which decomposes the system into a hierarchy of partitions, each providing services to and consuming from its adjacent partitions. A side effect of a layered architecture is that security measures can be easily implemented.

2.1 Unforseen Consequences

One of the key challenges in dealing with consequences is the vast amount of knowledge required to understand their impact on all the quality attributes. Bachmann et al note that the list of quality attributes in the ISO 9126 standard is incomplete, and that one must understand the impact on even the undocumented quality attributes [2]. Tyree et al note that traditional architecture methods do not focus on the rationale for an architectural decision and the options considered [21]. Kruchten notes that the reasoning behind a decision is tacit knowledge, essential for the solution, but not documented [19]. The result is that consequences of decisions may be overlooked.

Overlooking issues is a significant problem in architecture. In a study of architecture evaluations, Bass et al [4] report that most risks discovered during an evaluation arise from the lack of an activity, not from incorrect performance of an activity. Categories of risks are dominated by oversight, including overlooking consequences of decisions. Many of the overlooked consequences are associated with quality attributes. Their top risk themes included availability, performance, security, and modifiability.

Missing the impact on quality attributes at architecture time has an additional liability. Because quality attributes are system-wide capabilities, they generally cannot be fully tested until system testing [7]. Consequences that are overlooked are often not found until this time, and are expensive to fix.

3 Architecture Patterns

Patterns are solutions to recurring problems. A software pattern describes a problem and the context of the problem, and an associated generic solution to the problem. The best known software patterns describe solutions to object-oriented design problems [9], but patterns have been used in many aspects of software design, coding, and development. Patterns have been written for software architecture, and can be used in numerous software architecture methods [3] [5] [8] [15] [20].

Patterns have been shown to be a useful and potentially important vehicle for capturing some of the most significant architectural decisions [11]. One of the biggest difficulties of documenting architectural decisions is the capturing of rationale and expected consequences of a decision. This is where patterns are particularly strong, because the consequences of using the architecture pattern are part of the pattern.

The result of applying a pattern is usually documented as “consequences” or “resulting context” and is generally labeled as positive (“benefits”) or negative (“liabilities”). Each benefit and liability is described in some detail.

The payoff of using patterns can be great. When an architect uses a pattern, he or she can read the pattern documentation to learn about the side effects of the pattern. This reduces the chance of the architect failing to consider important consequences. This relieves the architect of the burden of being expert in all the quality attributes.

An important advantage of pattern-based architecting is that it is an integral part of most current architecture methods. It fits into the step of ADD that selects architectural patterns and tactics to satisfy the drivers (see [3] for further details.) The Siemens’ Views method [13] and the Rational Unified Process 4+1 Views [17] [18], use various strategies, such as patterns, to resolve issues identified in the views.

3.1 Limitations of Patterns in Identifying Consequences

The use of patterns in identifying and dealing with consequences is, however, currently significantly limited. The chief limitation is that patterns’ information on consequences is incomplete, not searchable or cross-referenced, and in general not as easy to use as it should be. Furthermore, it is difficult to learn about pattern interactions: how patterns may jointly impact quality attributes. These are the difficulties we focus on in this work.

Another difficulty is that pattern consequences are most often qualitative, not quantitative. Some quantification of architecture patterns’ impact on quality attributes has been done using a graded scale [20]. This is insufficient, since an architect needs to have rigorous analysis results of quality attributes to make informed decisions.

Even qualitative information is problematic: consequences are of different strengths but no such comparative information is given. We begin to address this in this work.

Another issue is that patterns contain proven, but general solutions. Architecture is concerned with specific, but tentative decisions. As such, the pattern use must be tailored to the specific system – the architect must evaluate the consequences of a pattern in the context of its proposed use. Several architecture patterns, particularly those in Buschmann et al [8], include common variants of the patterns that provide more specific solutions. However, the variants have not been extensively documented, and have little information on consequences. So the user is left to determine whether the consequences of a pattern still apply to a pattern variant under consideration.

An important source of unforeseen consequences is the interaction of multiple decisions. Multiple patterns may have overlapping consequences, or patterns and decisions not based on patterns may have overlapping consequences.

4 Analysis of the Impact of Patterns on QAs

In order for patterns to become a truly powerful architecture tool, it must be possible to find which patterns impact certain quality attributes, compare and contrast their impacts, and discover their interactions. To this end, we are analyzing the impact of patterns on quality attributes, and organizing this analysis in a way that is accessible and informative. This work is a companion to quantifying the impact of patterns on quality attributes: it adds a qualitative dimension by examining the nature of how a pattern impacts a particular quality attribute; not just how much.

We began by selecting a standard definition of quality attributes to be used in the study. We used the ISO quality model [14], which contains functionality, reliability, usability, efficiency, maintainability, and portability. We initially confined ourselves to the primary attributes, with the exception of functionality, where we selected the security sub-attribute. We added a property, implementability, as a measure of the difficulty of implementing the pattern.

We then selected the best-known architecture patterns, those from Buschmann et al [8]. We used the consequences in the book for our analysis of consequences. While the book gives several variants of the patterns, we limited this analysis to the “pure” form of each pattern – the variants will be investigated in our future work.

In the analysis of the consequences, we designated strengths as “strength” or “key strength,” and liabilities as either “liability” or “key liability,” based on the importance of the impact. If the impact on the quality attribute might be sufficient reason by itself to use or avoid the pattern, it was designated as “key.” This differentiation supports architectural reasoning: used in the context of a project’s architectural drivers, a key strength tends to enable fulfillment of an architectural driver, while key liability will severely hinder or perhaps prevent its fulfillment. We differentiated normal versus key impacts based on the severity described in the documentation. Where it was unclear, consequences were weighed against each other, and judgment was applied. Not every pattern had both key strengths and liabilities.

At least two to three sentences are needed to express each impact fully. Because of space limitations, we abbreviated the impacts to just a short sentence.

Table 1. Patterns' Impact on Usability, Security, Maintainability and Efficiency

	Usability	**Security**	**Maintainability**	**Efficiency**
Layers	*Neutral*	*Key Strength:* Supports layers of access.	*Key Strength:* Separate modification and testing of layers, and supports reusability	*Liability:* Propagation of calls through layers can be inefficient
Pipes and Filters	*Liability:* Generally not interactive	*Liability:* Each filter needs its own security	*Strength:* Can modify or add filters separately	*Strength:* If one can exploit parallel processing *Liability:* Time and space to copy data
Blackboard	*Neutral*	*Liability:* Independent agents may be vulnerable	*Key Strength:* extendable *Key Liability:* Difficult to test	*Liability:* Hard to support parallelism
Model View Controller	*Key Strength:* Synchronized views	*Neutral*	*Liability:* Coupling of views and controllers to model	*Liability:* Inefficiency of data access in view
Presentation Abstraction Control	*Strength:* Semantic separationo	*Neutral*	*Key Strength:* Separation of concerns	*Key Liability:* High overhead among agents
Microkernel	*Neutral*	*Neutral*	*Key Strength:* Very flexible, extensible	*Key Liability:* High overhead
Reflection	*Neutral*	*Neutral*	*Key Strength:* No explicit modification of source code	*Liability:* Meta-object protocols often inefficient
Broker	*Strength:* Location Transparency	*Strength:* Supports access control	*Strength:* Components easily changed	*Neutral:* Some communication overhead

4.1 Implications of Analysis

A few patterns have conflicting impacts on a quality attribute. The Blackboard pattern has both a positive and negative impact on maintainability, and efficiency is both a strength and a liability in the Pipes and Filters pattern. This shows the complex nature of quality attributes: the categories above should be broken down in more detail (see future work.) However, they also indicate that a pattern can have complex consequences. In these cases, the designer must consider multiple different impacts.

The context of the application affects the importance of the consequences. For example, the efficiency strength of Pipes and Filters to exploit parallel processing may not be achievable in some single thread systems. This also highlights how best to use the information: one uses the information as a starting point for more in-depth analysis and design. This is particularly true for the liabilities, as illustrated below.

Table 2. Patterns' Impact on Reliability, Portability, and Implementability

	Reliability	Portability	Implementability
Layers	*Strength*: Supports fault tolerance and graceful undo	*Strength*: Can confine platform specifics in layers	*Liability*: Can be difficult to get the layers right
Pipes and Filters	*Key Liability*: Error handling is a problem	*Key Strength*: Filters can be combined in custom ways	*Liability*: Implementation of parallel processing can be very difficult
Blackboard	*Neutral*: Single point of failure, but can duplicate it	*Neutral*	*Key Liability*: Difficult to design effectively, high development effort
Model View Controller	*Neutral*	*Liability:* Coupling of components	*Liability:* Complex structure
Presentation Abstraction Control	*Neutral*	*Strength:* Easy distribution and porting	*Key Liability:* Complexity; difficult to get atomic semantic concepts right
Microkernel	*Strength:* Supports duplication and fault tolerance	*Key Strength:* Very easy to port to new hardware, OS, etc	*Key Liability:* Very complex design and implementation
Reflection	*Key Liability:* Protocol robustness is key to safety	*Strength:* If you can port the meta-object protocol	*Liability:* Not well supported in some languages
Broker	*Neutral:* Single point of failure mitigated by duplication	*Key Strength:* Hardware and OS details well hidden	*Strength:* Can often base functionality on existing services.

We have used this information in evaluating the architecture patterns in a few industrial systems. While this work is early, our studies indicate that such evaluations can be very useful. The process consists of identification of the patterns in the architecture, and examining their impact on the important quality attributes of the system. In one case, we reviewed an architecture which used the Pipes and Filters pattern. A key liability of this pattern is reliability; it is difficult to implement error handling. This became a drill-down point in the review, and we investigated error handling in more depth. In another case, we observed the Layers pattern in a time-critical system. In order to deal with the fact that the Layers pattern has a performance liability, the designers allowed certain functions at the lowest layers to be called from the highest layer. Such "breakages" of a pattern should be designated as areas for careful testing, because they are intentional deviations from a proven design.

Early experience suggests that it supports lightweight architecture. It adds some rigor to architecture without extensive documentation and reviews. It strikes a "middle ground" between the extremes of no architecture and highly formalized architecture.

5 Related Work

Several quality attribute centered software architecture methods take an intuitive approach, including the QASAR method [5] and the attribute driven design (ADD) method [3]. Use of architecture patterns is also intuitive, and fits well in these models. In addition, the architecture pattern quality attribute information formalizes architecture patterns and their consequences, relieving the architect of some of the burden of ferreting out the consequences of architectural decisions.

Bachmann et al describe a knowledge framework designed to help architects make specific decisions about tradeoffs that impact individual quality attributes [2]. It focuses on individual quality attributes independently, while the pattern approach focuses more on interactions among patterns and quality attributes. It might be said that the knowledge framework favors depth, while the pattern-driven approach favors breadth. In this sense, it is likely that these two research efforts are complementary.

In the general model of architecture [12], the information is useful in the *Architectural Synthesis* activity, but is most valuable in the *Architectural Evaluation* activity. Architecture evaluators can use it to help them detect risks of omission [4].

6 Future Work

We have shown an initial analysis of a few architecture patterns identified to date, namely those found in Buschmann et al [8]. We are beginning to analyze others; most are described in Avgeriou and Zdun [1]. We have also begun analyzing the subcategories of quality attributes given in the ISO quality standard [14].

A process for using this data in pattern-based architecture reviews is being developed and used to collect data.

The interaction of the consequences of patterns has not been explored in detail. We intend to study which patterns are often used together. This helps identify potentially conflicting decisions, and help them make tradeoffs about which patterns to use.

The consequences in patterns are qualitative, but some quantification is useful. It would be useful to make the rudimentary quantification of the consequences: Key Strength, Strength, Neutral, Liability, and Key Liability more detailed. Such quantification is of necessity limited, and must be carefully crafted so as not to give the false impression that a numerical score of a pattern can replace analysis.

Pattern variants have rich potential. Variants of patterns should be investigated to understand in more detail how individual variants affect the impact of generic architecture patterns on the quality attributes. Different pattern variants have somewhat different strengths and liabilities. This information can be used to help the architect choose among different variants of patterns.

A table such as the ones above can show only very abbreviated information; more detailed information is needed. This information might be incorporated into a tool that functions as an architectural decision support system such as knowledge frameworks for quality attribute requirements as proposed by Bachmann et al [2].

References

1. Avgeriou, P., Zdun, U.: Architectural Patterns Revisited - a Pattern Language. In: 10th European Conference on Pattern Languages of Programs, Irsee, Germany (July 2005)
2. Bachmann, F., Bass, L., Klein, M., Shelton, C.: Designing software architectures to achieve quality attribute requirements. In: IEE Proceedings, vol. 152 (2005)
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Reading, MA (2003)
4. Bass, L., Nord, R., Wood, W., Zubrow, D.: Risk Themes Discovered Through Architecture Evaluations. SEI Report CMU/SEI-2006-TR-012 (2006)
5. Bosch, J.: The Design and use of Software Architectures. Addison-Wesley, London (2000)
6. Bosch, J.: Software Architecture: The Next Step. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 194–199. Springer, Heidelberg (2004)
7. Burnstein, I.: Practical Software Testing. Springer, Heidelberg (2003)
8. Buschmann, F., Meunier, R., Rhonert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley, West Sussex, England (1996)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1995)
10. Garlan, D.: Software Architecture: a Roadmap. In: Proceedings of Future of Software Engineering, Limerick Ireland (2000)
11. Harrison, N., Avgeriou, P., Zdun, U.: Architecture Patterns as Mechanisms for Capturing Architectural Decisions. IEEE Software (September/October 2007)
12. Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A., America, P.: Generalizing a Model of Software Architecture Design from Five Industrial Approaches. Journal of Systems and Software 30(1), 106–126 (2007)
13. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture, pp. 7–8. Addison-Wesley, Reading, MA (2000)
14. International Standards Organization: Information Technology - Software Product Quality - Part 1: Quality Model, ISO/IEC FDIS 9126-1
15. Jansen, A.G., Bosch, J.: Software Architecture as a set of Architectural Design Decisions. In: Proceedings of WICSA 5, pp. 109–119 (November 2005)
16. Klein, M., Kazman, R.: Attribute-Based Architectural Styles. Technical Report CMU/SEI-99-T2-022 (October 1999)
17. Kruchten, P.: The 4+1 View Model of Architecture. IEEE Software 12(6) (1995)
18. Kruchten: The Rational Unified Process: an Introduction, 3rd edn. Addison-Wesley, Reading (2004)
19. Kruchten, P., Lago, P., van Vliet, H.: Building up and reasoning about architectural knowledge. In: Hofmeister, C., Crnkovic, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, Springer, Heidelberg (2006)
20. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Addison-Wesley, Reading, MA (1996)
21. Tyree, J., Ackerman, A.: Architecture Decisions: demystifying Architecture. IEEE Software, 19–27 (March/April 2005)