3 Leveraging Patterns to Satisfy Quality Attributes and Document Architectural Decisions

This chapter consists of material from the following publications:


Abstract

All of software design involves making decisions and reifying those decisions in code. The decisions made during software architecting are particularly significant, as they have system-wide implications, especially on quality attributes. However, the impact of the architectural decisions may be first understood much later; when the system architecture is difficult to change. This can result in difficulties in implementing measures to achieve quality attributes. In addition, architects often fail to document their decisions well. This leads to architectural erosion: decisions made during later development might conflict with the original architectural decisions and thus cause significant system disruptions.

Architecture patterns address these challenges by capturing structural and behavioral information, and encouraging architects to reflect on decisions, in a way that does not interfere with the natural architectural design process. They can also 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. They are easy to use and provide a rich set of information about rationale, consequences, and related decisions.

3.1 Introduction

Throughout the software design process, developers must make decisions and reify them in code. The decisions made during software architecting are particularly significant in that they have system-wide implications, especially on the quality attributes. Because of its wide ranging impact and enduring nature, software architecture must fulfill a unique dual role. On the one hand, it must “look forward” and lay the foundation for software yet to be designed. On the other hand, it must provide a “window to the past” into which future designers may look to understand the architecture and its rationale. Unfortunately, both these tasks are very difficult, making software architecture one of the very daunting jobs in software development.
On the “looking forward” side, we see that designing an architecture so that it achieves its quality attribute requirements is one of the most demanding tasks an architect faces [15]. 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 [76]. 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 [12]. Requirements may not be sufficiently specific and are often a moving target. Finally, the consequences of decisions made are often overlooked [15]. As a result, architectural rework is common.

On the “window to the past” side, we find that architects often fail to adequately document their decisions because they do not appreciate the benefits of documenting such decisions, they do not know how to document them, or they do not perceive when decisions are made. This lack of thorough documentation can significantly disrupt the system when decisions made later, during subsequent development iterations, conflict with the original architectural decisions.

Fortunately, software architecture patterns can simultaneously address both these problems, resulting in an architecture that can better satisfy its quality attributes, and one where the rationale of design decisions is more transparent to future designers and architects. In order for them to be effective, though, information about their relation to documentation of decisions and their impact on quality attributes must be understood. This chapter explores these relationships and shows how architecture patterns can be leveraged to address these twin problems.

The remainder of this chapter is organized as follows: Chapter 3.2 explores these two problems in more depth. Chapter 3.3 gives information about the impact of common architecture patterns on common quality attributes. Chapter 3.4 describes patterns as an approach to architecture documentation, and compares them to architectural decisions in general. Chapter 3.5 discusses practical uses, and chapter 3.6 discusses limitations of the use of patterns. Related work is found in chapter 3.7.

3.2 Problem Overview

3.2.1 Challenges of Consequences of Architecture
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 [81]. 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 [55]. 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.

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 [12]. Tyree and Ackerman note that traditional architecture methods do not focus on the rationale for an architectural decision and the options considered [129]. Kruchten notes that the reasoning behind a decision is tacit knowledge, essential for the solution, but not documented [93]. 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. [17] 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 [34]. Consequences that are overlooked are often not found until this time, and are expensive to fix.

### 3.2.2 Challenges in Capturing Decisions

Most software architecture documentation describes the system’s structure from different views [91]. Ideally, this documentation also records decisions that architects made while designing the system. Recording only the decision does little good, however; for the documentation to be truly useful, architects must also capture the alternatives considered, their expected consequences, and the rationale—that is, the reasons for selecting a particular alternative. Our discussion of decision documentation here refers not just to the decision but rather to all of its aspects. Unfortunately, this wider definition is what architects most neglect when
documenting their decisions.

Current research trends in software architecture focus on architectural decisions [91] as first-class entities and capture their explicit representation in the architectural documentation. Such documentation extends the typical views of a software system’s architecture—such as the interacting components and connectors—with explicit representations of the architectural decisions that convey the rationale underlying a particular design [81].

The ultimate goal of documenting architectural decisions is to alleviate a major problem in the field: architectural knowledge vaporization [29]. This knowledge vaporizes because architects fail to record their decisions, so significant information about a software system’s architecture is unavailable during the development and evolution cycles. These decisions can’t be explicitly derived from the architectural models. And, because they exist merely as tacit knowledge in the heads of architects or other stakeholders, they inevitably dissipate. As the well-known saying in software architecture goes, “If something is not written down, it does not exist.”

Knowledge vaporization has consequences across the software industry, including expensive system evolution, lack of stakeholder communication, limited reusability of architectural assets, and poor traceability between the requirements, the architecture, and the implementation.

If the recording of architectural decisions is to become standard practice, then documenting decisions must be easy and somewhat automatic. To this end, researchers are investigating conceptual models, methods, processes, and tools for documenting decisions [35, 82, and 93]. However, in practice, architects still fail to document their decisions for many reasons. The most significant include the following:

- The substantial effort required to document and maintain architectural decisions seems greater than the perceived benefit.
- Architects sometimes make decisions without realizing it or without reflecting explicitly upon them, so they don’t know what to document.
- Rather than disrupt the creative flow of design, architects defer decision documentation until the architecture is essentially complete; at that point, they’ve often forgotten many decisions and the rationale behind them.
- Architects don’t know how to document their decisions.

Clearly, such difficulties make the process of documenting architectural decisions problematic, leading to loss of valuable architectural knowledge.
3.3 Architecture Patterns and Quality Attributes

As described above, patterns are particularly strong at identifying the consequences of architectural decisions, since the consequences of using patterns are part of the pattern. 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 [39, 121]. 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. [33], 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.

3.3.1 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 [76], 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. [33]. 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.

Table 3.1: Patterns’ Impact on Usability, Security, Maintainability and Efficiency

<table>
<thead>
<tr>
<th></th>
<th>Usability</th>
<th>Security</th>
<th>Maintainability</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layers</td>
<td>Neutral</td>
<td>Key Strength:</td>
<td>Key Strength: Separate modification and testing of</td>
<td>Liability: Propagation of calls through</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supports layers of access.</td>
<td>layers, and supports reusability</td>
<td>layers can be inefficient</td>
</tr>
<tr>
<td>Pipes and Filters</td>
<td>Liability: Generally not interactive</td>
<td>Liability: Each filter needs its own security</td>
<td>Strength: Can modify or add filters separately</td>
<td>Strength: If one can exploit parallel processing</td>
</tr>
</tbody>
</table>
### 3.3.2 Implications of the 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.

Table 3.2: Patterns’ Impact on Reliability, Portability, and Implementability

<table>
<thead>
<tr>
<th>Blackboard</th>
<th>Neutral</th>
<th>Key Strength:</th>
<th>Key Liability:</th>
<th>Liability:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Independent agents may be vulnerable</td>
<td>extendable</td>
<td>Hard to support parallelism</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Key Liability:</td>
<td>Difficult to test</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Liability:</td>
<td>Extendable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model View Controller</th>
<th>Key Strength:</th>
<th>Neutral</th>
<th>Key Strength:</th>
<th>Key Liability:</th>
<th>Liability:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Synchronized views</td>
<td></td>
<td>Coupling of views and controllers to model</td>
<td>Inefficiency of data access in view</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Presentation Abstraction Control</th>
<th>Strength:</th>
<th>Neutral</th>
<th>Key Strength:</th>
<th>Key Liability:</th>
<th>Liability:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Semantic separation</td>
<td></td>
<td>Separation of concerns</td>
<td>High overhead among agents</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Microkernel</th>
<th>Neutral</th>
<th>Neutral</th>
<th>Key Strength:</th>
<th>Key Liability:</th>
<th>Liability:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Very flexible, extensible</td>
<td>High overhead</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reflection</th>
<th>Neutral</th>
<th>Neutral</th>
<th>Key Strength:</th>
<th>Liability:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>No explicit modification of source code</td>
<td>Meta-object protocols often inefficient</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Broker</th>
<th>Strength:</th>
<th>Strength:</th>
<th>Strength:</th>
<th>Neutral:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Location Transparency</td>
<td>Supports access control</td>
<td>Components easily changed</td>
<td>Some communication overhead</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 3.2: Patterns’ Impact on Reliability, Portability, and Implementability

<table>
<thead>
<tr>
<th>Reliability</th>
<th>Portability</th>
<th>Implementability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Layers</strong></td>
<td>Strength: Supports fault tolerance and graceful undo</td>
<td>Strength: Can confine platform specifics in layers</td>
</tr>
<tr>
<td><strong>Pipes and Filters</strong></td>
<td>Key Liability: Error</td>
<td>Key Strength:</td>
</tr>
</tbody>
</table>

47
### Filters
- **Filters**: handling is a problem
- **Filters can be combined in custom ways**: 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
- **Neutral**: Liability: Complex structure

### Presentation Abstraction Control
- **Neutral**: Strength: Easy distribution and porting
- **Neutral**: 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.

---

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.

### 3.4 Architecture Patterns and Documenting Decisions

Architecture patterns are solutions to general architectural problems that developers have verified through multiple uses and then documented. They thus offer an effective way to capture some of the most significant design decisions and provide appropriate alternative solutions for various design challenges. Pattern documentation includes the pattern’s usage context, a recurring design problem solved by a recurring solution that resolves the general challenges of the problem, and the solution’s consequences.

Patterns help mitigate the four primary documentation challenges as follows:
• Architecture patterns include general structural and behavioral information, making it easier and faster to document architectural decisions.

• In applying architecture patterns, architects make decisions that encourage them to both reflect on those decisions and consider related issues.

• Pattern selection is indispensible to the architecting process, so architects can record related decisions with little effort. Pattern usage thus fits within the natural flow of the architecture design process.

• Patterns follow an easily understood form, which is highly compatible with proposed description templates for architectural decisions.

As the following descriptions and comparisons show, architecture patterns and architectural decisions have much in common.

3.4.1 Patterns: Coupling Structure and Consequences

One of architectural patterns’ key benefits is that they capture the system’s general architectural structure—which is typically well known and easily recognized—and couple it with consequences that are often not as readily recognized. This is particularly useful when attempting to reconstruct architectural decisions: the system’s structure indicates the (explicit or implicit) architecture pattern. The pattern description, in turn, indicates consequences of the architectural decision (especially with respect to quality attributes). These consequences are, in effect, less apparent decisions derived from the primary decisions. Developers can use this valuable knowledge to evaluate an architecture, although they can more precisely measure a pattern’s actual impact on quality attributes through thorough analysis, such as quantitative performance analysis. The particular pattern variant used also indicates whether alternative variants or related patterns might be applied.

3.4.2 Decisions: Capturing Key Information

An architectural decision is a decision that affects the system architecture. Jan Bosch proposes that a decision consists of requirements and a solution, and that each design decision addresses some system requirements while leaving others unresolved [29].

According to Bosch, design decisions might

• add components to the architecture,

• impose functionality on existing components,

• add requirements on components’ expected behavior, or
add constraints or rules on part or all of the software architecture.

He goes on to state that an architectural decision can represent many solution structures, including an architectural style or pattern.

A crucial consideration of design decision documentation is which information to collect. That is, what critical information about a decision should you convey to other architects and developers? Key information includes the issue being designed, the decision made, the alternatives considered, and the reasoning behind the decision. Anton Jansen and Jan Bosch characterize this information as a problem, motivation, cause, context, potential solutions, and decision [81]. Jeff Tyree and his colleagues describe this and other important information about decisions and give a sample template for recording them [129].

A second important consideration is to determine what kinds of information comprise architectural decisions. Philippe Kruchten [91] describes several types of design decisions:

- Existence decisions relate to the behavior or structure in the system’s design or implementation.
- Non-existence decisions describe behavior that is excluded from the system.
- Property decisions state an enduring, overarching system trait or quality, which might include design guidelines or constraints.
- Executive decisions are those driven by external forces, such as financial imperatives.

Another consideration here is the important distinction between two knowledge types [95].

- *Application-generic knowledge* is an architect’s implicit knowledge, gained through previous experiences in one or more domains (such as architectural patterns, tactics, or reference architectures).
- *Application-specific knowledge* involves all the decisions made during a particular system’s architecting process, as well as the architectural solutions that implemented the decisions.

These two knowledge types are related in that application-generic knowledge is used to make decisions for a single application, and thus constructs application-specific knowledge.
As we noted earlier, a key difficulty with architectural decisions is in getting people to record the critical information surrounding a decision, rather than just recording the decision itself. To this end, researchers are developing tools to make the recording process as easy and unobtrusive as possible [35, 82]. In addition to tools that explicitly document architectural decisions, model-driven software development researchers have developed tools for defining architectural metamodels with constraints and model-checking features. We can easily extend MDSD tools to metamodels for architectural decisions (for example, following the templates described in the next section) and use them to define and automatically check formalizable constraints that result from an architectural decision. This hypothesis, however, remains to be tested in practice; we’re not yet aware of any MDSD tools that can effectively record architectural decisions.

3.4.3 The Pattern-Decision Relationship

Architecture patterns and architectural decisions are complementary concepts. Using a pattern in system design is, in fact, selecting one of the alternative solutions and thus making the decisions associated with the pattern in the target system’s specific context. For example, an architect designing a user interface structure might consider two alternative patterns: Model-View-Controller and Presentation-Abstraction-Control. The MVC pattern divides the application into components that contain the core functionality and data (the model); the views presented to the user; and the user-input controller. The PAC pattern creates a hierarchy of cooperating agents, each of which manages its own data display. The PAC pattern is very extensible but is less efficient than MVC. So, in deciding which pattern to use, the architect must consider the target system’s performance and extensibility needs.

The major difference between architecture patterns and architectural decisions is in the scope of information each contains. Each architectural decision document describes an individual decision about the target system. In contrast, patterns describe solutions that have proven successful in multiple applications. Thus, architectural decisions are specific, but tentative; patterns are proven, but general. When designing systems, architects consider patterns as alternative solutions. In relation to the two knowledge types described earlier, architectural decisions comprise application-specific knowledge, whereas architecture patterns comprise application-generic knowledge.

Although patterns and decisions have different origins, we can investigate their relation by comparing how they’re documented. Architectural decisions include the issue to be decided, the alternative solutions, the decision made, and the reasons for the decision. Similarly, a pattern describes the issue (in a problem section) and the decision (in a solution section). Alternative solutions are motivated by forces
(different variants of the solution) and justified in a rationale section. Table 1 shows the typical sections in patterns documentation (Frank Buschmann and colleagues offer examples in their book [33]), architectural decision documentation [129], and their correspondence.

Table 3.3: Pattern and architectural decision documentation

<table>
<thead>
<tr>
<th>Pattern section</th>
<th>Decision section</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td>Patterns represent generic knowledge, so pattern names give the pattern a recognizable, reusable name to facilitate communication; as decisions are knowledge specific to the current situation, they’re not intended to serve as a “language” among the architects/developers.</td>
</tr>
<tr>
<td>Problem</td>
<td>Issue</td>
<td>The pattern’s problem statement roughly corresponds to the issue raising a decision. In both cases, it expresses a stakeholder’s concern that must be addressed.</td>
</tr>
<tr>
<td>Category</td>
<td>Group</td>
<td>Some pattern authors categorize their patterns in some scheme; correspondingly, decisions are grouped. The decision groups are usually rather clear because they’re rooted in a concrete decision process, whereas pattern categories are often rather abstract.</td>
</tr>
<tr>
<td>Status</td>
<td></td>
<td>Status information, such as pending, decided, or approved, refers to concrete realization of a decision. As generic knowledge, a pattern doesn’t need such a section.</td>
</tr>
<tr>
<td>Context</td>
<td>Assumptions, constraints</td>
<td>A pattern’s context and a decision’s assumptions and constraints both set the scene and characterize the situation in which the pattern can be and the decision is applied.</td>
</tr>
<tr>
<td>Solution varies</td>
<td>Positions</td>
<td>A decision’s positions are the alternatives that have been considered to tackle the issue. This roughly corresponds to two parts of the pattern text: the forces describe various concerns that can lead to different solutions; the variants of the solution represent alternatives in solving the problem by balancing the concerns in a different way.</td>
</tr>
<tr>
<td>Solution</td>
<td>Decision</td>
<td>A pattern’s <em>solution</em> describes the generic solution to the recurring design problem covered by the pattern. This corresponds to the concrete <em>decision</em> that resolves the issue of a decision.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Rationale</td>
<td>Argument</td>
<td>A pattern describes the generic <em>rationale</em> of applying the pattern’s solution in relation to the forces. Similarly, a decision’s argument section explains why the decision was made.</td>
</tr>
<tr>
<td>Resulting context/consequences</td>
<td>Implications</td>
<td>A pattern’s <em>resulting context</em> section describes the context that is created by applying the pattern. A pattern’s consequences section describes the consequences of its application. These sections correspond to a decision’s <em>implications</em>.</td>
</tr>
<tr>
<td>Example, known uses</td>
<td>Known uses</td>
<td>Known uses are the sources from which the pattern has been mined; examples show how to apply the pattern’s generic solution in a specific way. Because decisions are concrete knowledge, neither known uses (there’s only one) nor examples are needed.</td>
</tr>
<tr>
<td>Related patterns</td>
<td>Related decisions, requirements, artifacts, or principles</td>
<td>A pattern’s solution often leads to a context in which other related patterns can be applied. This corresponds directly to the related decisions, requirements, artifacts, or principles of a decision.</td>
</tr>
<tr>
<td>Notes</td>
<td>Notes</td>
<td>In decision templates, <em>notes</em> can be taken during the decision process as part of the communication between stakeholders. Even though a lot of communication usually occurs when patterns are written, notes aren’t explicitly recorded but informally captured in other sections of the pattern template or in the verbose text in the other pattern sections.</td>
</tr>
</tbody>
</table>

As table 1 shows, patterns and architectural decisions also differ in their documentation format. Although they have many of the same sections, pattern descriptions focus on timeless, generic knowledge (and hence have a name, examples, and known uses), whereas decision templates focus on concrete knowledge relating to a specific situation (and hence contain elements such as status
Another interesting aspect is how the two facilitate solution selection. In the patterns realm, architects can derive alternative solutions in two ways. First, as table 1 shows, an individual pattern can provide alternative solutions by resolving the forces in different ways using different variants. The Pipes and Filters pattern, for example, might appear in different variants such as purely sequential, forks/joins, feedback loops, and so on. Second, two or more patterns can be complementary in a specific decision topic. For example, when deciding on interacting components’ distributed communication, you might choose the Client-Server, Peer-to-Peer, or Publish-Subscribe pattern or combine two or all three.

As table 1 shows, patterns can support traditional architectural documentation. The patterns provide application-general knowledge in the areas of assumptions, constraints, positions, implications, and related decisions. The architect might wish to augment this information with application-specific knowledge; in this case, the pattern serves as a reminder of issues to consider. In some cases, a pattern contains nearly all the desired decision documentation (albeit at a general level). In such cases, the architect must document little beyond the decision itself. So, using patterns in decision documentation can minimize the efforts necessary to document extra information, such as design considerations, consequences, and so on.

When using pattern-oriented knowledge, it’s important to understand the consequences of applying the pattern on functional and (especially) nonfunctional system aspects. When you decide to use a pattern, you decide to accept its consequences. The Layers pattern, for example, partitions software in a way that often results in many function calls, which might decrease performance. In deciding to use this pattern, you must consider its performance impact. However, because the Layers pattern supports security levels in the application, you might use it if you want to adopt a particular security model and implementation. This brings up an important advantage of using patterns with respect to decision-making: A decision’s consequences are rarely fully understood or even anticipated. Because patterns are based on extensive prior experience, the consequences are generally well understood and described in the pattern documentation. Thus, pattern usage can help you understand the consequences beforehand and document them for future reference.

### 3.5 Using Patterns: Practical Considerations

Architectural design is an especially challenging decision-making process because it involves frequent trade-offs: A given structure often satisfies a few requirements at others’ expense. Furthermore, a decision’s consequences might introduce new requirements, so you might have to trade off a solution’s benefits with the additional
system burdens it entails. Trade-offs are particularly rich and complex among a system’s nonfunctional attributes. For example, deciding to implement a certain security approach might impact the system’s performance and usability. Because of the interaction complexities among performance, usability, and security, architects might be particularly unaware of their decisions’ consequences on such nonfunctional system aspects.

The architecting process is highly intuitive. To develop an architecture, architects use their own past experiences, others’ experiences, and whatever application-generic architectural knowledge is available. Using a proven and systematic approach to architecting is highly desirable—you get no style points for originality in software architecture!

During architecting, architects periodically consider one or more of the key architectural drivers—that is, the most important system-affecting requirements. They consider alternative structural approaches, decide on one or more, and repeat the process. Ideally, they should record these decisions as they happen. However, as we noted earlier, they generally document the decisions later, if at all.

Patterns play an important role in this decision-making process. For certain decision topics, architects might select one or more patterns or a single pattern’s variants as alternative approaches. When they select the pattern, its usage documents an architectural decision. This has several key benefits. First, the solution has been proven to work. Second, because the literature describes patterns in detail, documentation of pattern-associated decisions already exists. Third, many architecture patterns include documentation of their consequences and system impact, including on nonfunctional requirements. Thus, architects can easily learn which further trade-offs they must consider.

As we now describe, there are several advantages and limitations to using patterns as a primary method of architectural documentation.

3.5.1 Benefits of Patterns Use

Perhaps the biggest challenge of architectural documentation is capturing the critical information surrounding the decision itself. Doing this takes time, effort, and attention; consequently, developers tend to avoid interrupting the design flow to document their work. However, postponing documentation increases the risk that they’ll forget critical issues or forgo documentation altogether. This is precisely where patterns shine: Their use is easily noted (without interrupting design), and, at the very least, the additional information reminds architects about what issues to document later. The application-generic knowledge of rationale, forces, and consequences is an important first step. Patterns address the principal difficulties of
recording decisions as follows:

- **The substantial effort required to document and maintain architectural decisions seems greater than the perceived benefit.** Because patterns include a description that matches architectural decisions’ required description, using the pattern is a starting point for documenting that decision. Even if developers expend no additional documentation effort, the pattern name itself refers to the generic pattern description and thus offers at least minimum documentation.

- **Architects sometimes make decisions without realizing it or without reflecting explicitly upon them, so they don’t know what to document.** Applying patterns per se signifies that some of the most significant architectural decisions have been made. Furthermore, patterns explicitly state the system quality attributes’ consequences, and this helps architects recognize their decisions and implications. Patterns also contain references to related patterns, which help architects think about alternative solutions and eventually select one based on a rationale.

- **Rather than disrupt the creative flow of design, architects defer decision documentation until the architecture is essentially complete; at that point, they’ve often forgotten many decisions and the rationale behind them.** Patterns fit well within several well-established architecture design methods. They also emerge naturally through the design process without disrupting the creative flow. Nonetheless, developers can easily document decisions related to the pattern’s usage afterwards by simply reusing the pattern description information. Finally, architecture patterns fit well into the tools that support architecting methods, and we expect such tools to become more mature and more widely used.

- **Architects don’t know how to document their decisions.** Patterns contain much of their own documentation. They’re also compatible with emerging decision documentation formats and tools. Patterns also remind architects of issues to consider.

### 3.6 Limitations of Patterns Use

Architecture patterns don’t relieve the architect of all responsibility for documenting decisions. First, the architect must still document application-specific decisions. Second, not all decisions have appropriate patterns. While additional architecture patterns have been and will continue to be written [11], some architecture areas will never have patterns. So, architecture patterns will always have a limited solution space.
Similarly, you can’t capture some architectural decisions in terms of patterns because they depend on the project’s concrete scope and domain. Technology-related decisions (such as deciding on a specific technology vendor) or organizational decisions (such as company guidelines or project team setup) are just two examples of project-dependent decisions that have severe consequences for the resulting architecture. A fourth limitation relates to the fact that architects often use multiple patterns together. If they don’t understand the various pattern interactions, they might select conflicting patterns. This problem is exacerbated by the fact that architects tend to use architecture patterns unsystematically. In chapter 7, we propose an approach to support pattern selection based on desired quality attributes, and systematic design decisions based on patterns. This problem can also be ameliorated by effective architectural reviews using patterns, as described in chapter 8. In addition, chapter 4 describes a systematic analysis of relationships of patterns and quality attributes. It also includes annotations of architectural diagrams, which helps address issues with effective documentation.

Finally, an important challenge with patterns is what to do if developers use the wrong pattern but don’t discover this until well into the implementation phase. As with any architectural decision, backing out is difficult. However, we might draw on the rich information that patterns contain to reduce such difficulty. To our knowledge, this area has yet to be researched at all.

3.7 Related Work
Several quality attribute centered software architecture methods take an intuitive approach, including the QASAR method [28] and the attribute driven design (ADD) method [15]. 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 [12]. 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 [72], 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 [17].