Design Decisions in Software Architecture

Mikko Herranen
Department of Computer Science
University of Helsinki
Helsinki, Finland
Email: mikko.herranen@iki.fi

***Abstract*—Software architectures have high costs for change. Therefore choosing the right architecture for a software project from the start is highly preferable. The suitability of a particular architecture depends on the software project at hand. Choosing an architecture is usually a balancing effort between different requirements and potentially subjective quality attributes. Software architectures also erode during evolution. The erosion is caused by knowledge vaporization, which is in turn caused by poor documentation. Unfortunately, documenting a software architecture, and especially the design decisions leading to that architecture, is not a trivial task. Viewing the software architecture as a composition of a set of explicit design decisions is a way to tackle knowledge vaporization by improving documentation. Design decision tools such as Archium are an effort to apply the design decisions concept in practice. Architecture patterns are an alternative approach that can be thought of as an application of the same principle. In addition to improved documentation, architecture patterns offer an easy to use, readily available and familiar solution to architectural design problems although they do not completely remove the need to make application-specific decisions.**

I. Introduction

Software architectures have high costs for change. Therefore reducing change by choosing the right architecture for a software project from the start is highly preferable. Unfortunately, making architectural decisions is not easy, and therefore poor decisions are made. Refraining from making decisions is not a solution either; decisions will be made regardless of whether they are conscious decisions or not. To quote Falessi et al, "every software system has a software architecture; it can be implicit or explicit." [1] It is the job of the architect to recognize when decisions are being made and react properly but this is part of what makes architecture decisions hard.

Software rarely lives on in the exact form it was originally written. Instead, software evolves during its lifetime due to changing requirements. Architecture evolution means changing the architecture by making new design decisions or removing obsolete ones in order to satisfy the new requirements in harmony with the existing design decisions [2]. Unfortunately, software architectures tend to erode during evolution. *Knowledge vaporization*, i.e. the loss of knowledge about the reasoning behind the architectural design decisions, is a key reason for software architecture erosion. While personnel changes are a factor in knowledge vaporization, it is mainly caused by lacking or outright non-existent documentation. Knowledge vaporization can be mitigated by proper and rigorous documentation practice.

However, there are some less obvious difficulties in documentation [3]. For one, architects do not in fact always realize that they are making a significant architectural decision or do not know how to document the decisions they make. There is also a tendency to delay documentation lest the creative flow is disrupted by the documentation work. The effort required might also be perceived too great compared to the benefits. We will return to this issue in later sections.

The design decisions concept is a way to tackle knowledge vaporization. The concept requires viewing the software architecture as a composition of a set of explicit design decisions. Applying the design decisions concept reduces knowledge vaporization of design decisions by making the decisions an explicit part of the architecture.

Poor architectural decisions are a related issue. Typical problems caused by poor architecture include duplication of business logic across multiple systems, tight coupling of business and program logic, and higher than desired maintenance costs [4]. Architecture patterns help the architect in decision making by reusing earlier tried and tested combinations of design decisions [2]. Architecture patterns capture sets of design decisions and can thus be thought of as a practical application of the design decisions concept.

In this paper we look at the design decisions concept both from an abstract and a more practical point of view. The design decisions concept itself is rather abstract but there are efforts, namely the Archium software and meta-model, that aim to systematically apply it in practice. Design patterns can be seen as a more pragmatic manifestation of the design decisions concept.

This paper is organized as follows. First, research methods are briefly discussed in Section II. Then, the results are presented in Section III. Section III is further organized into subsections dealing with quality attributes, common architectural problems, the design decisions concept, and architecture patterns. Finally, the results are discussed in Section IV and possible future work items in Section V. The paper is concluded in Section VI.

II. Methods

We began the work by studying design decision papers found mainly through Google Scholar. ISI Web of Science was used to a lesser extent. We soon discovered the link between design decisions and architecture patterns by forward and reverse snowballing (following citations and finding papers that cite one of the papers we use). This paper is largely based on the papers listed in the references although some parts reflect the author's experiences working as a software developer.

III. Results

In this Section we focus on the problems found in software architecture and offer some solutions to them. We focus on two specific solution domains: the design decisions concept and architecture patterns that can be seen as an application of the design decisions concept. The Section is organized as follows. First, quality attributes are discussed in Section III-A. Then, typical architecture problems are discussed in Section III-B. The design decisions concept and the Archium software and meta-model are introduced in Section III-C. Architecture patterns are discussed in Section III-D. Section III-E briefly discusses a few actual, well-known design patterns.

A. Quality Attributes

A quality attribute is a desired non-functional property such as reliability, usability, or security [5]. Quality attributes are standardized by the ISO 9126 standard which lists six primary and 21 secondary quality attributes [5]. Quality attributes are, unless explicitly defined, subjective. For example, a term such as *performance* might have different meaning to different stakeholders: it might mean worst-case latency to someone but efficient use of system resources to someone else.

Qualifying quality attributes is complicated. Things to consider include the way quality attributes are described and the uncertainty and importance estimate assigned to each of the attributes. Falessi et al enumerate four different ways to describe a quality attribute [1].

- **Just a term.** The quality attribute is described with just a term, e.g. *performance*. This approach requires little effort from those involved but may be prone to misunderstandings.
- **Term and use case.** In this approach a use case is included in addition to the term.
- **Term and measure.** In this approach a measure is added to the term, e.g. *latency* and milliseconds.
- **Term, use case, and measure.** Finally, in its most complete form a quality attribute is defined by all three sub-attributes.

The uncertainty of a quality attribute can be defined as the risk of an alternative failing to provide a specific level of fulfillment [1]. For example, an innovative new technology may promise enhanced performance with a higher risk than an older, well-known technology. Falessi et al provide several different uncertainty levels of a quality attribute [1], the most important of which are "not expressed at all" and several levels of "expressed". The main drawback with not expressed uncertainty is that two alternatives may exhibit the same performance but different uncertainties. If the uncertainty is not expressed, it cannot be considered as part of the decision making process.

Quality attributes are not equally important to the stakeholders. The most common way to describe the importance of a quality attribute is the direct weight method which represents the stakeholders' desire for a quality attribute by means of a scaled value [1]. Other, less common, methods include weight elicitation and utility curve methods [1].

B. Problems in Software Architecture

Poor software architecture practices cause various problems [2]. For one, the decisions made are cross-cutting and intertwined which leads to the fragmentation of design decision information, making it hard to find and change those decisions. Second, design rules and constraints are violated. Because the rules and constraints set during the design process are lost, the future architects of the project are not aware of them and can violate them easily, which in turn leads to architectural drift and increased maintenance costs. Finally, obsolete design decisions are not removed which accelerates software erosion. As a result of these issues, the developed software has a high cost for change and they erode quickly [2].

Many software architectures are "mystical": there is no apparent logic behind the choices made in the design of the architecture [4]. This is at least in part an illusion created by poor or outright missing documentation. Architecture decisions are usually not documented explicitly but are implicit; knowledge of the architecture is "embedded" in the source code of the software. This has several downsides: the design decisions are cross-cutting and intertwined, design rules and constraints are violated, and obsolete design decisions are not removed [2]. Properly documenting architecture decisions is important because architects make them in complex environments and they involve trade-offs [4].

Because of the lack of proper documentation, the stakeholders, e.g. the developers and other architects, do not have the energy to go through the architecture which results in degradation of the architecture [4]. Why is architectural documentation neglected then? Architects neglect documentation for various reasons.

- The effort required to create and maintain the documentation might be perceived too great compared to the value it delivers [3].
- Architects do not always realize that they are in fact making a significant architectural decision that requires documentation. This second point is important; if the architect does not realize a design decision is being made, she does not know what to document. A related issue is that the architect might not know how to document her decisions [3].
- There is a tendency to delay documentation to avoid disrupting the creative flow (perceived or real) the architect is in. When the architect finally has the time to document the architecture, many decisions and the rationale for them have been forgotten [3].
- Architects may have to work in a hurry. In a perfect world the architect has enough time to perform all her tasks but unfortunately we do not live in a perfect world and sometimes projects are rushed due to business reasons.

When it comes to documentation, traditional approaches, such as RM-ODP (Reference Model for Open Distributed Processing) and RUP (Rational Unified Process), are inadequate in several ways [4]:

- They do not convey changes to previous versions very well. For example, developers are not interested in reading through a lengthy component model just to find a few key items that have changed. Developers might also not be skilled in reading component models or object diagrams.

- Traditional approaches do not convey the architecture's implications, such as training needs or organizational impacts, well.
- Traditional approaches do not convey rationale for the decisions or considered options very well. Because of the lack of documented rationale behind the design decisions, the same already answered questions about the architecture get asked time and again.
- Finally, the traditional methods do not provide traceability of decisions back to requirements.

Typical problems and challenges encountered while making architectural decisions are [1]:

- The requirements are often informally captured due to the subjective nature of quality attributes.
- Quality attribute requirements are difficult to specify in an architectural model.
- Certain types of requirements cannot be easily defined in the early development phases of an iterative software project, which forces architects to do guesswork.
- Stakeholders have different (and differing) points of view regarding the system, caused by conflicting goals and expectations.

One final thing to note is that the architecture guides the evolution of the software. This last point is important: poor architectural design decision made early in the life of a software project may have long-lasting ill effects that carry far into the future of the software.

C. Design Decisions Principle

By now we have established the importance of making good architectural design decisions and documenting them properly. How does the architect accomplish that then? Unfortunately, there is no silver bullet to architecture design because there is no single best architecture [1]. Instead, the architecture of a software project is a compromise resulting from balancing different requirements and quality attributes.

Architecture decisions concept is a key to demystifying "mystical" architectures [4]. An architectural decision consists of requirements and a solution. Each design decision addresses some system requirements while leaving others unresolved [3]. Software architecture can be viewed as a composition of a set of architectural design decisions [2]. Adopting this view reduces knowledge vaporization of design decisions by making the decisions an explicit part of the architecture. Exact definition: "A description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture." [2]

Treating software architecture as a set of architectural design decisions has several benefits. It helps the the architect with guarding the conceptual integrity of the software architecture [2]. Explicit design space exploration inherent in this approach prevents the architect from making obvious mistakes. Analysis of both the software architecture and the design process. Improved traceability of the design decisions and their relationship to features, design aspects, concerns, and among themselves. To measure a decision's architectural significance, an architect should ponder whether the decision at hand affects one or more system quality attributes. If so, the architect should make and document the decision completely [4].

Archium is an attempt to bridge the gap between the design decisions concept and actual real-world architecture design. It is both a software tool and an architectural meta-model that represents design decisions [2]. In its authors' words, Archium promotes design *using* change rather than *for* or *with* change [2].

A software architecture is described in Archium as a set of changes or design decisions which together form the architecture [2]. In other words, the architecture of a software is the sum of a set of design decisions.

The Archium meta-model consists of three sub-models: an architectural model, a design decision model, and a composition model [2]. The architectural model describes the architecture of the software while the design decision model describes the design decisions made during the architecture design of the software. Finally, the composition model relates the changes of the design decision model to the architectural model.

Our treatment of Archium is necessarily brief. The reader is encouraged to read the paper by Jansen and Bosch [2] and visit the Archium website [1].

D. Architecture Patterns

Whereas Archium might leave something to be desired, architecture patterns offer an easy to use, readily available and familiar solution to architectural design problems. Architectural patterns are similar to software patterns but instead of a single design issue they concern the entire software, describing high-level system architecture and its implications [3].

Architecture patterns can be defined as "solutions to general architectural problems that developers have verified through multiple uses and then documented" [3]. Architecture patterns and design decisions are complementary concepts; when the architect chooses a design pattern, she implicitly commits to a set of decisions, namely those associated with the pattern [3].

Architecture patterns are a possible solution to documentation issues. They help the architect in the documentation effort in several ways, mitigating the problems mentioned in Section III-B. Let us now look each of the problems in turn and see how patterns help the architect to solve the issue [3].

The first problem mentioned in Section III-B is that the architect might perceive the documentation effort too great compared to the benefits of documentation. Fortunately, the patterns have self-documenting properties: since patterns have a name and a description associated to them, choosing a pattern

[1]http://www.archium.net/

serves as the starting point for the documentation effort. This self-documenting nature is of help also when the architect does not know how to document the decisions or is working in a hurry. Architecture patterns also include structural and behavioral information which reduces the documentation effort [3].

Second problem is that, when making a decision, architects do not necessarily realize the significance of the decision being made and thus neglect documentation. In this case patterns help by forcing the architect to think about the decision at hand. Applying patterns indicates significant design decisions are being made [3]. Pattern descriptions also state the consequences to the system quality attributes. Because the pattern documentation describes the side effects of applying the pattern, the architect does not have to be an expert in all the quality attributes [5]. Pattern documentation also contains references to related patterns which helps architects think about alternative solutions.

Third problem mentioned in Section III-B is the tendency to delay documentation lest the creative flow is lost. Architecture pattern selection is a natural part of the architectural design effort, making it easy to document related decisions without disrupting the creative flow [3].

Unfortunately, patterns are not a silver bullet either. There are some problems with patterns [3]. The major issue with patterns is that they are general: there are not and never will be a pattern for every specific need. Instead, there will also always be a number of application-specific decisions that the architect has to make and document properly. There are also decisions which are made solely on business (instead of technical) grounds. For example, company policy might be to prefer Vendor A's software components to Vendor B's components because of acquisitions or other inter-company business deals. No design pattern can capture these kind of decisions. Architects also use multiple patterns together. If the architect does not understand the interactions between the patterns, she might apply conflicting patterns.

E. Example Patterns

We will now briefly discuss a few well-known architecture patterns. Let us begin with the Blackboard pattern The idea of the Blackboard architecture is to divide a complex task to smaller sub-tasks that clients process independently [6]. The blackboard is a shared repository from which the clients fetch the inputs for their computation and to which they write the results of the computation. A controller component monitors the state of the blackboard and controls the clients. The Blackboard pattern is suitable for tasks that are parallelizable, such as image or speech recognition application [6].

In the Interceptor pattern a framework offers a number of reusable services to the applications which register interceptor objects with the framework [6]. When an event comes in, the framework dispatches the event—possibly after some internal processing—to the appropriate interceptor object. In a sense, the interceptor objects extend the services already provided by the framework. For example, in a web framework such as Django [2], the web framework intercepts HTTP requests and dispatches events to the appropriate interceptors the application has registered. In the case of web frameworks the Interceptor pattern reduces the amount and complexity of application code compared to traditional library-based solutions.

[2]https://www.djangoproject.com/

The Model View Controller (MVC) pattern divides the application into three interconnected core parts [6]. The model part contains the state and persistent data of the application, including database access. The model has no knowledge of the view or the controller parts [7]. In its simplest form, it is simply a mindless data storage. The view part contains the views presented to the user. The view does not contain any application logic or state; it is simply responsible for visually presenting the state of the application—held by the model—to the user [7]. The controller part handles user input.

As an example of MVC, let us consider a toggle button widget in a mobile application. The button can be in two possible states: on and off. The user can change the state of the button by pressing it. The initial state of the button is stored in the model. When the widget is created, the view draws the button on the screen in the state described by the model. When the user presses the button, the controller receives the press, updates the model to reflect the new state, and notifies the view to redraw the button based on the (updated) state found in the model.

Benefits of the MVC pattern include loose coupling of application logic and view logic which makes replacing views or having multiple views for the same data easy.

Presentation Abstraction Control (PAC) is a related pattern [6]. A system may offer multiple diverse functionalities and present them to the user through a unified user interface. In PAC the system is decomposed into a tree-like hierarchy of agents. The agents at the leaves are responsible for specific functionalities while the agents in the middle combine the functionalities of the related lower level agents. The root agent at the top of the tree coordinates the entire whole. The individual agents are designed according to MVC [6].

IV. Discussion

From a software developer's point of view, architecture patterns are a lighter and much more practical approach to software architecture design than elaborate design decision models such as Archium. Architecture patterns come into existence almost by themselves: when an architect designs a solution to a problem and if that solution has properties that make it reusable, a pattern is found. Then it is simply a matter of naming and describing the pattern, defining its consequences and constraints, and documenting it. The Archium model is very heavy in comparison.

However, as seen in Section III-D, architecture patterns are not without their problems. Even if patterns are rigorously applied, there will be application-specific decisions which still have to be documented properly. Another possible issue with architecture patterns is the potential tendency to see problems as fitting a certain pattern even if they do not. As a popular phrase puts it, "if all you have is a hammer, everything looks like a nail". In comparison, well-known object oriented design patterns address smaller, localized design issues, and most likely are less susceptible to this issue.

While the more formal applications of the design decisions principle show promise and are a more complete solution than architecture patterns, they have not gained much foothold in the information technology industry as far as we are aware.

V. Future Work

Falessi et al discuss an extensive characterization schema and ranking method for architecture decision making techniques [1]. A comparison of the techniques discussed in this paper, carried out with the Falessi et al method, should prove interesting.

Avgeriou and Zdun propose a pattern language that "acts as a superset of the existing architectural pattern collections and categorizations" in order to help finding and applying the appropriate architecture patterns [6]. It might be worthwhile to study architecture patterns from this point of view.

VI. Conclusion

It is important to make the correct architectural decisions early on in the life of a software project. This brings about two related problems: First of all, making the right decisions is difficult due to various human factors. A software architecture is a compromise resulting from balancing different requirements and potentially highly subjective quality attributes. Second, when an architectural decision is made, all aspects of that decision must be properly documented. Documentation is the best tool against software erosion caused by knowledge vaporization, but documenting architectural decisions is hard because the effort required might be perceived as too great compared to the value, there is a tendency to delay documentation work, and architects do not always realize they are making (significant) architectural decisions.

We looked at two different ways to make architectural decisions: the design decisions principle, embodied by Archium, and architecture patterns. According to the design decisions principle, the architecture of a software project can be viewed as a composition of a set of architectural design decisions, each of which consists of requirements and a solution and addresses some system requirements while leaving others unresolved. The design decisions principle reduces knowledge vaporization by including the decisions in the architecture. Archium is both a software tool and a meta-model for architectural design. It is an attempt to bridge the gap between the design decisions concept and actual real-world architecture design. Despite its benefits, the design decision principle has not gained a strong foothold in the information technology industry.

Architecture patterns are solutions to general architectural problems that developers have verified through multiple uses and then documented. While architecture patterns seem quite different at a first glance, they can be thought of as a different kind of application of the design decisions principle. When the architect chooses an architecture pattern, she implicitly commits to the set of decisions associated with that pattern. Architecture patterns help the architect in the documentation effort by possessing self-documenting properties and forcing the architect to think of the decisions at hand. Model View Controller (MVC) and Blackboard are examples of architecture patterns.

References

[1] D. Falessi, G. Cantone, R. Kazman, and P. Kruchten, "Decision-making techniques for software architecture design: A comparative survey," *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, p. 33, 2011.

[2] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*. IEEE, 2005, pp. 109–120.

[3] N. B. Harrison, P. Avgeriou, and U. Zdlin, "Using patterns to capture architectural decisions," *Software, IEEE*, vol. 24, no. 4, pp. 38–45, 2007.

[4] J. Tyree and A. Akerman, "Architecture decisions: Demystifying architecture," *Software, IEEE*, vol. 22, no. 2, pp. 19–27, 2005.

[5] N. B. Harrison and P. Avgeriou, "Leveraging architecture patterns to satisfy quality attributes," in *Software Architecture*. Springer, 2007, pp. 263–270.

[6] P. Avgeriou and U. Zdun, "Architectural patterns revisited–a pattern language," 2005.

[7] Y. Ping, K. Kontogiannis, and T. C. Lau, "Transforming legacy web applications to the mvc architecture," in *Software Technology and Engineering Practice, 2003. Eleventh Annual International Workshop on*. IEEE, 2003, pp. 133–142.