Kai Niklas

# BECOME A BETTER SOFTWARE ARCHITECT

## Actions and insights from practical experience

Design – Theory
Design – Practice
Decide
Simplify
Code
Document
Communicate
Estimate
Balance
Coach
Consult
Market

# Become a Better Software Architect - Actions and insights from practical experience

**by Dr.-Ing. Kai Niklas**

Publish date: 26th May 2019

Get updates and news on
bettersoftwarearchitect.com

Do you like the book? Tweet it: “Just became a #betterSoftwareArchitect with these actions and insights by @kniklas bettersoftwarearchitect.com”

# Contents

# Part I.

# Foreword

# 1. About this book

Several years ago, I was asked: “How have you become a software architect?”. We talked about necessary skills, experience and the amount of time and dedication it took to build up knowledge. Moreover, I went through the steps which I took. Which technologies I have worked with or tried out, and what I have learned during my professional and non-professional career.

This conversation has triggered myself and I started to structure the topics for my personal growth. “What makes a good software architect?”, I wondered, and “How can I improve to become a better software architect?”. I read articles and books, and of course talked with peers.

In this book, I want to share an overview of my insights with you. Which skills are most important, and how to improve them to become a (better) software architect.

This book addresses software engineers who want to learn and understand more about the work of an architect. Further, it gives insights for software architects who want to extend their existing knowledge.

## 1.1. What things will you learn?

The book is divided into two major sections:

- ▷ Understand the different *roles* and required *skills* of software architects
- ▷ *Insights* on how to improve the skills based on my personal experience

This book will not get into each detail but tries to give you a broad overview. Links to definitions, interesting books, videos or webpages are given to dig deeper into details there. It is not the goal of this book to describe each existing concept again. Only major concepts which are generating benefit within the context of this book will be described and discussed in more details. Many concepts are linked to the corresponding wikipedia article for quick look ups.

## 1.2. Focus on the human

From my experience, the human as such is highly important. Building up a "T" shape skill-set with broad and deep knowledge is important, but it is not enough to succeed in the domain of software engineering or architecture. Humans are building software, not machines. Humans need more than facts to build great software.

I personally feel, that the human is often forgotten. Not only in literature, but also in day to day life. Hence, I decided to write this book around a software architect: A human who is more than a walking book. I tried to balance this book between hard and soft skills in the area of software architecture.

## 1.3. Are the things proven to work?

The collection of insights, that are the foundation of this book, reflect my personal point of view, based on my experience of the last 15 years in IT. I try to be as unbiased as possible and refer to scientific

sources whenever possible. Moreover, I give reading recommendations to further resources with similar or opposite thoughts, whenever applicable. This gives you the ability to decide by yourself if my saying is sound.

All actions and insights are written in small chunks, so that they are easy to consume and understand. Every chunk includes a description, the benefits, and at least one concrete example. With this you have a general applicable insight which can be adapted to your situation, as well as a demonstration on how to apply it. For many insights I added further readings to books or interesting web-pages.

# Part II.

# Introduction

# 2. The Software Architect

*"Design and programming are human activities; forget that and all is lost."*

*(Bjarne Stroustrup)*

To understand how a software architect can improve its skills, it is beneficial to recap what software architecture is and what a software architect does. For software architecture there are several good definitions. Further, it is important to also distinct between the traditional understanding of software architecture and *agile* software architecture. Based on definition, understanding of different specializations, flight levels and typical activities, the core skills every software architect should have is derived.

## Contents

## 2.1. Definitions – Software Architect

Before diving into details, let's have a look at two definitions first to align on the term "software architecture" and what a software architect is:

> *"**Software architecture** is the fundamental organization of a system, represented by its components, their relationships to each other and to the environment, and the principles that determine the design and evolution of the system." [8]*

> *"A **software architect** is a software expert who makes high-level design choices and dictates technical standards, including software coding standards, tools, and platforms. The leading expert is referred to as the chief architect." [12]*

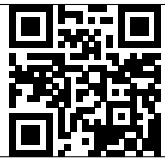
### Agile Software Architecture

The classical definition for software architecture is still valid for agile software architecture. But there is one main difference: It's the mindset and the derived mechanics how to tackle architecture questions and design.

In the classical world, the whole design is determined at the very beginning. This is called Big Design Up Front (BDUF or BUFD) [6]. Within agile software development the concrete design decision is kept open as long as possible, until enough information is available and key learning have been made to decide on a concrete design. The assumption is, that at the beginning the functional and non-functional requirements are too vague to determine the concrete design.

Agile architecture evolves over time (growth mindset), whereas classic architecture is fixed at the very beginning (fixed mindset). If you

want to learn more about the differences of those two mindsets, have a look at the following video:



Growth Mindset vs. Fixed Mindset (YouTube)
by John Spencer (http://bit.ly/2H0FBrg)

## 2.2. Architecture Flight Levels and Domains

Architecture is applied on several "flight levels" of abstractness. The level influences the importance of necessary skills. Different levels require different skills. Clustering or naming the levels can be done differently. I prefer the segmentation into these 3 levels:

- ▷ **Application Architecture**: The lowest level of architecture. Focus on one single application. Very detailed, low level design. Communication usually within one development team. Driven by concrete functional and non-functional requirements.
- ▷ **Solution Architecture**: The mid-level of architecture. Focus on one or more applications which fulfill a business need (business solution). Some high, but mainly low-level design. Communication between many development teams. Aligning functional and non-functional requirements with overarching business goals.
- ▷ **Enterprise Architecture**: The highest level of architecture. Focus on many solutions. High level, abstract design, which is detailed out by solution or application architects. Communication across the organization. Highly business goal driven.

Architects are often seen as the glue between different stakeholders. Three examples:

- **Horizontal**: Bridge communication between business and developers or different development teams.
- **Vertical**: Bridge communication between developers and managers or managers and senior management.
- **Technology**: Integrate different technologies or applications with each other.

Another dimension of architects is their specialization in a specific domain. Some common examples are:

- Software / Application
- Infrastructure / Network
- Database / Data
- Security
- Integration
- DevOps / Automation
- Business / Process
- Organization

Figure 2.1 puts all the dimensions together. Different architecture "specializations" can be applied on different levels. For example, a data architect on application level is concerned about data models within the application, whereas the data architect on enterprise level is driving the overall data-warehouse (DWH) initiative. Both are data architects, but with different goals and challenges.

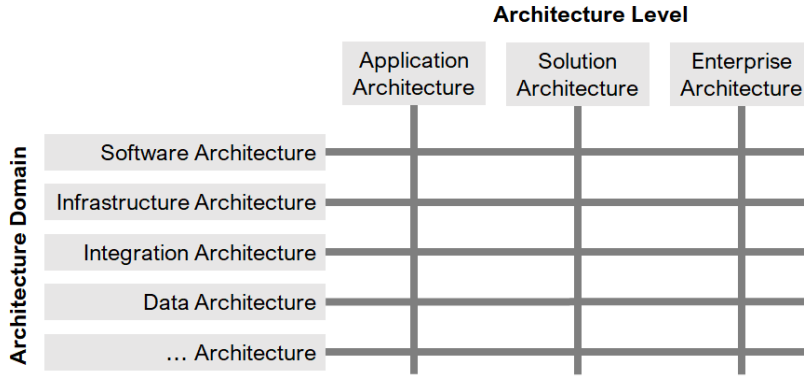



Figure 2.1.: Levels and domains of software architects

## 2.3. Typical Activities of Software Architects

To understand the necessary skills an architect needs, we need to understand typical activities. The following (non-final) list contains from my perspective the most important activities:

- ▷ Define and decide development technology and platform
- ▷ Define development standards, e.g., coding standards, tools, review processes, test approach, etc.
- ▷ Support identifying and understanding business requirements
- ▷ Design systems and take decisions based on requirements
- ▷ Document and communicate architectural definitions, design and decisions
- ▷ Check and review architecture and code, e.g., check if defined patterns and coding standards are implemented
- ▷ Collaborate with other architects and stakeholders
- ▷ Coach and consult developers

- ▷ Define, detail out and refine higher level design into lower level design

**Note**: Architecture is a continuous activity. Especially, when applied in agile software development. Thus, these activities need to be repeated.

## 2.4. Important Skills of Software Architects

To support the laid-out activities, specific skills are required. From my experience, read books and discussions, the skills every software architect should have can be boiled down to the following:

> **Design, Decide, Simplify, Code, Document, Communicate, Estimate, Balance, Coach, Consult, Market**

Depending on the level of architecture, and the concrete role, some skills are more important than others. For example, an enterprise architect does not need to have deep coding skills. But an understanding of what developers are doing all day is helpful. Whereas application architects should be involved in coding. Hence, improving in this area is beneficial.

You can use this list as a compass to determine the areas in which you are strong and where you can potentially improve yourself. Let's go through one by one. For every skill I have laid out some actions or insights to follow up and to improve in that area.

# Part III.

# Skills of an Architect

# 3. Design – Theory

> *"If you think good architecture is expensive, try bad architecture."*
>
> *(Brian Foote and Joseph Yoder)*

What makes a good software design? This is one of the most often stated question I receive. I will make a distinction between design in theory and in practice. To my experience, having a mix of both is most valuable. Let's start with theory.

In this section you will learn more about the classic design patterns, patterns and anti-patterns in general and how they will help to create applications. Further, you will learn more about quality measures and metrics to increase understanding of why architects put an emphasize on design and non-functional requirements.

## Contents

## 3.1. Know the basic design patterns

Design patterns are one of the most important tool architects need to have to develop maintainable code and systems. With patterns you can reuse design to solve common problems with proven solutions. Patterns are not libraries or functions which can be copy and pasted. They are rather solution blueprints and need to be implemented and adapted to the realities in the code.

### Benefits

Design patterns are a toolkit of *tried and tested solutions* to common problems in software design. Knowing patterns is useful as it teaches how to solve various problems, e.g., using principles of object-oriented design.

Design patterns define a *common language*, which is helpful to communicate more efficiently. Further, you can build on top of existing knowledge and adapt to your problems.

### Example

Let's assume you want to lay out the code structure for a user interface, have little experience and using no frameworks. You start programming, and after every iteration and every new feature, it gets harder and harder to maintain the code. At this point you may ask yourself: "I am not the first person on earth who did something like that. Isn't there a common solution to structure code for front-ends?". And the answer is yes. There are many patterns to structure your front-end code, e.g., Model-View-Controller (MVC) [14]. With MVC you get a very clear structure of the code. And the best is, that everyone who already knows MVC understands your code as well.

You may say, that MVC is old and outdated. But the base concept of MVC is still valid, and it is still around in many applications and frameworks. For example, the framework "Spring MVC" is using it as its base architecture. Based on MVC, new patterns emerged which are used in recent frameworks, e.g. Model View ViewModel (MVVM)[15] is used in Angular, Ember.js, Vue.js, ReactJS, etc.

### Further Reading

▷ **Design Patterns: Elements of Reusable Object-Oriented Software** [1] written by the "Gang of Four" (GoF) is one of the most important books about design patterns, and a must read to everyone who is in software development. Although, the patterns were published more than 20 years ago, they are still the basis of modern software architecture.

Find an overview of popular patterns on wikipedia (http://bit.ly/2PPuW5B)

## 3.2. Dig deeper into patterns and anti-pattern

If you already know the basic GoF patterns, then extend your knowledge with more software design patterns. Or dig deeper into your area of interest.

Additionally, learn about anti-patterns, which are the opposite of patterns. Whenever they occur in your design, the risk is high, that the solution will be ineffective or leads to massive problems.

**Important**: Patterns and anti-patterns are not only applicable to design, e.g., object-oriented programming. You can find them every-

where, e.g., enterprise organization, project management, software design, software development, software maintenance, programming, configuration management and many more.

## Benefits

Having the knowledge of patterns, especially in your domain, helps you to get your work done more efficiently. The experience you made by yourself is probably the most important. But your time is limited, and you cannot try out everything by yourself. You need to rely on others. Patterns are a great source of knowledge. Especially, those written by trustworthy people, including their rationale.

With anti-patterns you can more easily stop implementations or behaviors which are leading into the wrong direction. Your gut feeling will most likely tell you, that something is wrong with the design, but you cannot find the right arguments. With anti-patterns this is easier, as they often come with concrete examples.

**Important**: Patterns and anti-patterns are important, but it is easy to over-engineer and apply too many patterns. This will lead to ineffective systems. I observed this once in an university project where they tried to solve every problem with a pattern. I would call that an anti-pattern, too. Keep the right balance.

## Example – Pattern Deep Dive

If your job or interest is to integrate applications with each other, then you could do a research for common patterns and anti-patterns. As said above, you are not the first person on earth and you should build on top of existing knowledge to avoid mistakes others did already. For application integration the book “Enterprise Integration Patterns” [2] written by Gregor Hohpe is a great source for patterns.

This book is applicable in various areas, whenever two applications need to exchange data. Whether it is an old-school file exchange from some legacy systems or a modern microservice architecture.

### Example – Anti-Patterns

You can find many resources of design anti-patterns. Especially, the "new" way of building applications in a microservice architecture style has many pitfalls. To succeed with microservices, patterns and anti-patterns are very helpful to better understand the mechanics and what to avoid in your design. Often you do not see the impact of your decisions in the early stage of development, but late, when it gets costly to change.

One good resource to get you started is for example the book Microservices antipatterns and pitfalls [9] by Mark Richards. Mark lays out 10 common anti-patterns and pitfalls and provides solutions to avoid them.

There are many more resources for anti-patterns. Not only in the design area. But also, for software development itself. For example Stefan Wolpers described roughly 160 Scrum Anti-Patterns in his guide [17].

### Further Reading

- ▷ **Enterprise Integration Patterns** [2] by Gregor Hohpe
- ▷ Microservices antipatterns and pitfalls [9] by Mark Richards
- ▷ Scrum Anti-Patterns Guide [17] by Stefan Wolpers

## 3.3. Know quality measures and metrics

Defining and applying architecture is not an end in itself. There are good reasons why patterns, guidelines and coding standards are defined, applied and controlled. It is to build high quality software products.

Software quality has various definitions. For the context of this book I would like to highlight the following two closely related notions (compare [16]):

- **Software functionality:** Indicates how good the software fulfills functional requirements or business needs. This is tightly related with the term *fit to purpose*, which stands for how ideal the software is for the given use and context.
- **Software structure:** Indicates how good the software meets non-functional requirements, which directly influence the delivery of the functional requirements, e.g., maintainability or extensibility.

There are many quality measures for different purposes. Probably the most known and common measures are the following, according to ISO 9126 [13], but there are many more:

- **Functionality**: Suitability, Accuracy, Interoperability, Security
- **Reliability**: Maturity, Fault tolerance, Recoverability
- **Usability**: Understandability, Learnability, Operability, Attractiveness
- **Efficiency**: Time behavior, Resource utilization
- **Maintainability**: Analyzability, Changeability, Stability, Testability
- **Portability**: Adaptability, Installability, Co-existence, Replaceability

To meet non-functional as well as functional requirements, the following actions can be taken:

1. **Patterns & Guidelines**: The usage of proven patterns and avoidance of anti-patterns can help to increase non-functional requirements. For every quality aspect, specific guidelines can be defined, e.g., write and document code in a specific way so that it stays maintainable.
2. **Static code analysis (SCA):** At design time it is possible to check the code automatically against common coding standards or calculate metrics. The most prominent example where we can see this live in action is Eclipse. During coding the IDE is constantly checking not only the syntax, but applies additional checks, which for example detect missing exception handling or potential null-pointer issues. One popular tool for managing and checking code regularly is SonarQube [11].
3. **Dynamic (runtime) analysis:** Unfortunately, software cannot be judged solely by analyzing the code. Thus, it is necessary to run and "play" with the software (system). Performance analysis, security testing, or memory leak detection are three examples which run the software and simultaneously capture data to identify the potential cause within the code.
4. **Chaos Engineering:** Playing and experimenting with software systems in production gained some popularity recently, to build confidence of the system's capability to handle unexpected conditions. Chaos Monkey [5] by Netflix is one popular tool which can for example shut down servers automatically and test the fault tolerance of the system.

### Benefits

Often developers or architects do not understand or even know the reasons why they are applying and following specific architecture

styles, patterns or guideline. This leads to frustration and ignorance. If you are only told to do something, you are unlikely or even unwilling to follow. This does not help to craft good software. But if you can give reasons, it helps to establish the right mindset. Of course, the initial effort is higher, but in the long run, you will gain the benefits, as everyone has understood and applies it. Physiologic experiments demonstrate, that providing reasons to kids why they should not take a specific toy is twice as effective in the long run as simply forcing them to not take the toy[1].

With static code analysis tools, e.g., SonarQube, you can measure code and detect potential problems. You can do this continuously and integrate it into your automation tools, e.g., Jenkins. You could even define a threshold and stop deployments or releases if the quality is too low based on the defined metrics.

When applying static or dynamic code analysis tools, it is important to set the right mindset with developers and managers. It is not about controlling people or teams. It is not about judging that one team is better than another. It is not about monetary penalties for external vendors. The reason is, to support and help developers to create better software.

**Note**: If the culture and mindset in your company cannot handle transparency and handling errors well, I would discourage from introducing such tools. A missing mindset will start quality discussions and reduce the overall productivity and moral. Thus, first establish the mindset.

### Example – Testability

The more often you want to release your software, the more needs to be automated, so that you can assure the speed of delivery and

[1] Unfortunately, I do not find the source for that experiment. It was stated by Vera F. Birkenbihl, a German management trainer and author in one of her great talks.

quality. For example, amazon released already in 2011 their software every 11.6 seconds[2].

In particular, the automation includes test automation. You are not only testing code fully automatically, but also the environments. Hence, software, and even environments, need to be build in a way that they are testable. One popular solution is to use docker, so that the environment looks the same everywhere.

To achieve a high testability of software, guidelines and patterns are also important. For example, the usage of interfaces in combination with dependency injection allows to exchange classes during test runs with mocks. Mocks could for example return dummy values, instead of establishing a connection to a database, or simulate network outages by returning an error after a longer period.

One option to measure testability at design time is to measure the "Cyclomatic Complexity", which calculates the complexity of software [4]. The metric calculates how many linear independent paths the code can take. The more paths the code can take, the more test cases are necessary to have a proper test coverage. It is possible to reduce the complexity by breaking down complex code into smaller pieces and hence, simplify testing.

## Further Reading

- ▷ **Introduction to Software Quality [7]** by Gerard O'Regan
- ▷ **Metrics and Models in Software Quality Engineering** [3] by Stephen Kan
- ▷ **Chaos Engineering** [10] by Rosenthal et al.

[2] https://news.ycombinator.com/item?id=2971521

# Part IV.

# Appendix

# 4. About the author

**Dr.-Ing. Kai Niklas** has been working for more than 15 years in the IT sector in various roles and different industries. Currently he helps clients in the finance sector as principal consultant to innovate in the role of a software architect. Kai is specialized in modern software architecture and agile software development practices, application integration, DevOps, business process automation and SAP for insurance. He has a broad background in software engineering and architecture in different roles: Software engineer, software architect, application architect, solution architect, integration architect, system engineer and enterprise architect. He studied mathematics, computer science and did his doctoral degree in the field of service-oriented architectures.

Connect with me on LinkedIn
(http://bit.ly/3029Tl9)

# Bibliography


[1] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[2] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[3] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2003.

[4] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[5] Netflix. Chaos Monkey, 2019. https://github.com/Netflix/chaosmonkey [Online; accessed 16-April-2019].

[6] Kai Niklas. Big Design Up Front, 2019. https://en.wikipedia.org/wiki/Big_Design_Up_Front [Online; accessed 16-April-2019].

[7] Gerard O'Regan. *Introduction to Software Quality*. Undergraduate Topics in Computer Science. Springer International Publishing, 2014.

[8] Ralf H. Reussner and Wilhelm Hasselbring. *Software architect*. dpunkt.verlag, 2008. http://www.handbuch-softwarearchitektur.de/ [Online; accessed 16-July-2018].

[9] M. Richards. *Microservices Antipatterns and Pitfalls*. O'Reilly Media, 2016.

[10] Casey Rosenthal, Lorin Hochstein, Aaron Blohowiak, Nora Jones, and Ali Basiri. *Chaos Engineering*. O'Reilly Media,, 2017.

[11] Switzerland SonarSource S.A. Continuous Inspection | SonarQube, 2019. https://www.sonarqube.org [Online; accessed 16-April-2019].

[12] Wikipedia. Software architect, 2018. https://en.wikipedia.org/wiki/Software_architect [Online; accessed 16-July-2018].

[13] Wikipedia. ISO/IEC 9126, 2019. https://en.wikipedia.org/wiki/ISO/IEC_9126 [Online; accessed 10-January-2019].

[14] Wikipedia. Model view controller, 2019. https://en.wikipedia.org/wiki/Model-view-controller [Online; accessed 03-May-2019].

[15] Wikipedia. Model view viewmodel, 2019. https://en.wikipedia.org/wiki/Model-view-viewmodel [Online; accessed 03-May-2019].

[16] Wikipedia. Software quality, 2019. https://en.wikipedia.org/wiki/Software_quality [Online; accessed 10-January-2019].

[17] Stefan Wolpers. Scrum Anti-Patterns Guide, 2019. https://age-of-product.com/scrum-anti-patterns [Online; accessed 05-April-2019].