1 Introduction

Software systems are certainly among the most complex constructions that human beings have ever conceived and built, so it's not surprising that some software projects fail, and legacy systems often remain unmodified for fear they will simply stop working. In spite of this complexity, I still encounter project teams that are in control of their software systems, regardless of their industry, technology stack, size, or age. Adding functionality and fixing bugs in such legacy systems involves much less effort than I would have imagined, and new employees can be trained with reasonable effort. What do these project teams do differently? "How do they manage their software so effectively in the long run?"

Sustainability of software architectures

The main reasons for long-term success or failure in software development and maintenance can be found on many different levels. These include the industry, the technology used, the quality of the software system, and the qualifications of the users and developers. This book focuses on the **sustainability of software architecture**. I will show you which factors are most important for maintaining and expanding a software architecture over many years without making significant changes to your staffing, budget, or delivery schedule.

1.1 Software Architecture

50 definitions

Computer science has not been able to commit itself to a single definition of software architecture. In fact, there are more than 50 different definitions, each highlighting specific aspects of architecture. In this book we will stick to two of the most prominent definitions:

Definition #1:

"Software architecture is the structure of a software product. This includes elements, the externally visible properties of the elements, and the relationships between the elements." [Bass et al. 2012]

Architecture Views

This definition deliberately talks about elements and relationships in very general terms. These two basic materials can be used to describe a wide variety of architecture views. The static (module) view contains the following elements: classes, packages, namespaces, directories, and projects—in other words, all the containers you can use for programming code in that particular programming language. In the distribution view, the following elements can be found: archives (JARs, WARs, assemblies), computers, processes, communication protocols and channels, and so on. In the dynamic (runtime) view we are interested in the runtime objects and their interactions. In this book we will deal with the structures in the module (static)[1] view and show why some are more durable than others.

The second definition is one that is very close to my heart. It doesn't define architecture by way of its structure, but rather the decisions made.

Definition #2:

"Software architecture = the sum of all important decisions

Important decisions are all decisions that are difficult to change in the course of further development." [Kruchten 2004]

Structure vs. decisions

These two definitions are very different. The first defines what the structure of a software system consists of on an abstract level, whereas the second refers to decisions that the developers or architects make regarding the system as a whole. The second definition defines the space for all overarching aspects of architecture, such as technology selection, architectural style selection, integration, security, performance, and much, much more. These aspects are just as

1 The structures of the module view usually also influence the distribution view. Section 7.2 contains a proposal for displaying the distribution view in the module view.

important to an architecture as the chosen structure, but are not the subject of this book.

Decisions create guardrails

This book deals with the decisions that influence the structure of a software system. Once a development team and its architects decide on the structure of a system, they have defined **guardrails for the architecture.**

Guardrails for your architecture

Create an architecture that restricts the design space during the development of the software system and gives you direction in your work.

Guardrails for development

Guardrails allow developers and architects to orient themselves. The decisions are all channeled in a uniform direction and can be understood and traced, giving the software system a homogeneous structure. When solving maintenance tasks, guardrails guide all participants in a uniform direction, and lead to faster and more consistent results during adaptation or extension of the system.

This book will answer questions regarding which guardrails lead to durable architectures and extend the life of a software system.

1.2 Sustainability

Short-lived software

Software that is only used for a short period of time shouldn't have an architecture that is designed for sustainability. An example of such a piece of software is a program that migrates data from a legacy system into the database of a new application. This software is used once and then hopefully discarded. We say "hopefully" because experience has shown that program parts that are no longer used can be still found in many software systems. They are not discarded because the developers assume that they might need them again later. Also, to delete lines of working code that were created with a lot of effort isn't done lightly. There is hardly a developer or architect who likes to do this.[2]

2 In order to perceive discarding software as something positive, the clean code movement has introduced "Code Kata" workshops in which the same problem is solved several times and the code is discarded after each step.

The "Year 2000" problem

Most of the software we program today lives much longer than expected. It is often edited and adapted. In many cases software is used for many more years than anyone could have imagined at the coding stage. Think, for example, of the Cobol developers who wrote the first major Cobol systems for banks and insurance companies in the 1960s and 1970s. Storage space was expensive at the time, so programmers thought hard about preserving storage space for every field saved on the database. For the Cobol developers at the time, it seemed a reasonable decision to implement years as two-digit fields only. Nobody imagined back then that these Cobol programs would still exist in the year 2000. During the years prior to the turn of the millennium, a lot of effort had to be made to convert all the old programs to four-digit year fields. If the Cobol developers in the 1960s and 1970s had known that their software would be in service for such a long time, they would have used four-digit fields to represent years.

Our software will get old

Such a long lifetime is still realistic for a large number of the software systems that we build today. Many companies shy away from investing in new development, which generates significant costs that are often higher than planned. The outcome of new developments is also unknown, and users too have to be taken into consideration. In addition, the organization is slowed down during the development process and an investment backlog arises for urgently needed extensions. At the end of the day, it is better to stick with the software you have and expand it if necessary. Perhaps a new front end on top of an old server will suffice.

Old and cheap?

This book is rooted in the expectation that an investment in software should pay for itself for as long as possible. New software should incur the lowest possible maintenance and expansion costs in the course of its lifetime—in other words, the technical debt must be kept as low as possible.

1.3 Technical Debt

The term "technical debt" is a metaphor coined by Ward Cunningham in 1992 [Cunningham 1992]. Technical debt arises when false or suboptimal technical decisions are made, whether consciously or unconsciously. Such decisions lead to additional effort at a later point in time, which delays maintenance and expansion.

Good intentions

If there are capable developers and architects on the team at the beginning of a software development project, they will contribute their best experience and accumulated design know-how to creating a long-lasting architecture with no technical debt. However, this goal cannot be ticked off at the beginning of the project according to the principle "First we will design a long-lasting architecture and everything else will be fine from then on."

In truth, you can only achieve a long-lasting architecture if you constantly keep an eye on technical debt. In figure 1-1 we see what happens when technical debt grows over time in comparison to what happens when it is reduced regularly.

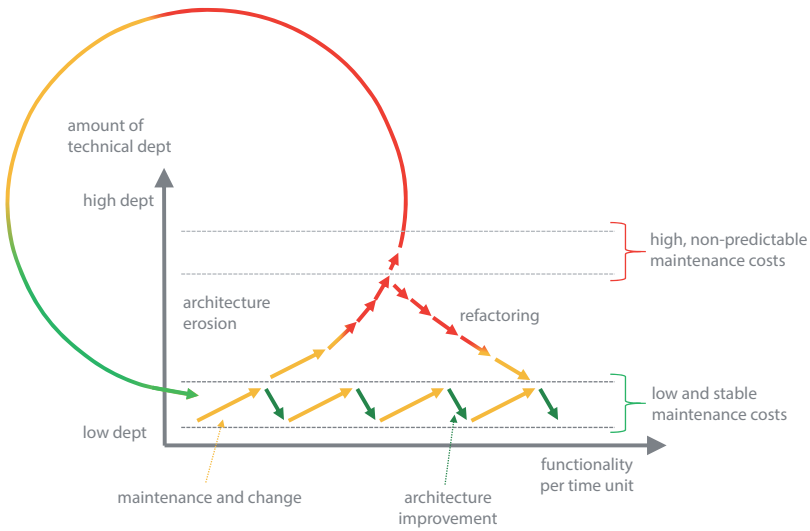

Figure 1-1
Technical debt and architectural erosion

A quality-oriented team

Imagine a team that is continuously developing a system using releases or iterations: if the team focuses on quality it will knowingly pile on new technical debt with each add-on (the yellow arrows in fig. 1-1). During the development of an expansion, the team will already be

thinking about what needs to be done to improve the architecture. Meanwhile (or after the expansion), technical debt will be reduced again (indicated by the green arrows in fig. 1-1). A constant sequence of expansion and improvement occurs. If the team works this way, the system remains in a corridor of low technical debt with a predictable maintenance effort (see green bracket in fig. 1-1).

A chaotic team

If the team doesn't aim to constantly preserve the architecture, the architecture of the system is slowly lost, and maintainability deteriorates. Sooner or later, the software system leaves the corridor of minor technical debt (indicated by the ascending red arrows in fig. 1-1).

Architectural erosion

The architecture erodes further and further. Maintenance and the extension of the software become increasingly expensive until you reach the point where every change becomes a painful effort. In figure 1-1 this case is made clear by the red arrows becoming shorter and shorter. As the erosion of the architecture increases, less and less functionality can be implemented, fewer bugs can be fixed, and fewer adaptations to other quality requirements can be achieved per unit of time. The development team becomes frustrated and demotivated and sends desperate signals to the project's management. Such warnings are usually registered far too late.

Too expensive!

If you are on the path depicted by the ascending red arrows, the sustainability of your software system will continuously decrease. The software system becomes error-prone, the development team gets a reputation for being sluggish, and changes that used to be possible within two person-days now take up to twice or three times as long. All in all, everything happens much too slowly. In the IT industry, "slow" is a synonym for "too expensive". That's right, technical debt has accumulated and with every change you have to pay interest on the technical debt principal plus the cost of the expansion.

Returning to good quality

The way out of this technical debt dilemma is to retroactively improve architectural quality. As a result, the system can be pulled back into the corridor of low technical debt step by step (see the red descending arrows in fig. 1-1). This path requires significant resources (both time and money) but still represents a reasonable investment in the future. After all, future maintenance will involve less effort and will be cheaper. For software systems that once had good architecture, this procedure usually leads to rapid success.

The CRAP cycle

The situation is completely different if the corridor of high technical debt is reached and the maintenance effort becomes disproportionately high and unpredictable (see the red bracket in fig. 1-1). I am frequently asked what "disproportionately high maintenance" means. A general answer that is valid for all projects is of course difficult to provide. However, in various systems with good architecture I have noticed that for every 500,000 lines of code (LOC), one or two full-time developers are required for maintenance. In other words, 40-80 hours per week per 500,000 LOC is a good starting point for determining the time needed to fix bugs and make small adjustments. If new functionality is to be integrated into the system, you will of course require even more capacity.

When I visit a company to evaluate an architecture, the first question I ask is about the size of the system(s). Secondly, I ask about the size and efficiency of the development department. If the answer is, "We employ 30 developers for our Java system of 3 million LOC, but they are all busy with maintenance and we can hardly get any new features implemented ..." I immediately assume that it is an indebted system. Naturally, setting such an expectation is harsh, but it has usually proved helpful as an initial hunch.

If the system has too much debt to be maintainable and extensible, companies often decide to replace the system with a new one (see the colored circle in fig. 1-1). In 2015, to my great delight, Peter Vogel described the typical lifecycle of a system with technical debt as a "CRAP cycle". The acronym CRAP stands for C(reate) R(epair) A(bandon) (re)P(lace)[3]. If repairing a system seems fruitless or too expensive, the system is left to die and eventually replaced.

However, this last step should be approached cautiously. As early as the beginning of the 2000s, many legacy systems written in COBOL and PL/1 were declared unmaintainable and replaced by Java systems. Everything was supposed to get better with this new programming language, and this promise was made to the managers who were tasked with funding the new implementation. Today, a number of these eagerly built Java systems are full of technical debt and generate immense maintenance costs.

3 *https://visualstudiomagazine.com/articles/2015/07/01/domain-driven-design.aspx*

Causes of technical debt

In the course of my professional life to date, I have repeatedly encountered four major causes of technical debt:

1. No knowledge of software architecture
2. Complexity and size of software systems
3. Architectural erosion arises unnoticed
4. A lack of understanding of custom software development processes on the part of managers and customers

These four factors usually occur in combination and often influence each other.

1.3.1 No Knowledge of Software Architecture

Programming ≠ Software Architecture

When a development team starts a new project, I always try to include an experienced developer-architect in the team. Every developer can program, but knowledge of sustainable software architecture only comes with experience.

Unusable software

If nobody in the team cares about sustainable architecture, the resulting system is likely to be maintenance-intensive. The architecture of these systems evolves over a period without any planning. Each developer fulfills her own personal ideas on architecture and/or design for her part of the software. "It's a legacy system!" is often heard in the latter case.

Starting with technical debt

In this case, technical debt is accumulated right from the beginning of and increases continuously. The usual attitude to such software systems is that they somehow grew up under a bad influence. Systems like this can often no longer be maintained after a relatively short period of time. I have even seen systems that have become unmaintainable after just three years.

Significant refactoring

The architectural and design ideas of architects and developers must initially be questioned and their quality standardized in order to move these systems closer to the corridor of low technical debt using whatever possible means. Overall, this is much more complex than getting a system with previously good architecture back on track. However, large-scale quality refactoring can be broken down into manageable sub-steps. After some initial small improvements (quick wins) the quality gain becomes noticeable through faster

maintenance. Such quality-improving work often costs less than a new implementation, even if many development teams understandably enjoy new development projects much more. This positive attitude to a new development project is often accompanied by underestimation of the complexity of the task.

1.3.2 Complexity and Size

The complexity of a software system is fed by two different sources: the use case for which the software system was built and the solution (the program code, the database, and so on).

An appropriate solution for the problem must be found within its specialized domain—a solution that allows the user to carry out the planned business processes using the software system. These factors are known as "problem-inherent" and "solution-dependent" complexity. The greater the complexity of the problem, the greater the solution-dependent complexity will be[4].

Problem-inherent complexity

This correlation is the reason why cost predictions and software development duration are often estimated too low. The actual complexity of the problem cannot be determined at the beginning of the project, so the complexity of the solution is underestimated many times over[5].

This is where agile methods apply. Agile methods only estimate the functionality that is to be implemented up to the end of each iteration. The complexity of the problem and the resulting complexity of the solution are rechecked time and again.

Solution-dependent complexity

Not only the complexity inherent to the problem is difficult to determine. The solution, too, contributes to the complexity. Depending on the experience and methodical strength of the developers, the design and implementation of a problem will vary in complexity. Ideally, a solution will only be as complex as the problem. In this case, we can say that it is a good solution.

Essential or accidental?

If the solution is more complex than the actual problem, the solution is not a good one and a corresponding redesign is necessary. The difference between better and worse is called the *essential*

4 see [Ebert 1995], [Glass 2002] and [Woodfield 1979]
5 see [Booch 2004] and [McBride 2007]

and *accidental* complexity. Table 1-1 summarizes the relationship between these four complexity terms.

Table 1-1 *Complexity*

	Essential	Accidental
Problem-inherent	■ Complexity of the domain	■ Misunderstandings about the domain
Solution-dependent	■ Complexity of technology and architecture	■ Misunderstandings about technology ■ Superfluous solution elements

Essential = inevitable

Essential complexity is the kind of complexity that is inherent in the nature of a project. When analyzing the domain, developers try to identify the essential complexity of the problem. The essential complexity inherent to a domain leads to a correspondingly complex solution and can never be resolved or avoided just by using a particularly good design. The essential complexity of the problem has thus become the essential complexity of the solution.

Accidental = superfluous

In contrast, the term "accidental complexity" is used to refer to the elements of complexity that are not necessary and can therefore be eliminated or reduced. Accidental complexity can arise from misunderstandings during analysis of the domain as well as during implementation by the development team.

If no simple solution is found during development due to incomprehension or lack of an overview, the software system is already unnecessarily complex. Examples of unnecessary complexity are multiple implementations, integration of unneeded functionality, and disregard of software design principles. However, developers sometimes risk additional accidental complexity if, for example, they want to try out new but unnecessary technology during development.

Software is complex

Even if a team manages to incorporate only essential complexity into its software, the immense number of elements involved makes software difficult to master. In my experience, an intelligent developer can retain an overview of about 30,000 lines of code and anticipate the effects of code changes in the other places. Software systems in productive use today tend to be considerably larger than this. We are more likely talking about a range of 200,000 to 100 million lines of code.

Architecture reduces complexity

All these arguments make it clear that developers require software architecture that gives them the greatest possible overview. Only then can they navigate their way around the existing complexity. If developers have an overview of the architecture, the probability of appropriate software changes being made increases. When they make changes, they can take all of the affected areas into account and leave the functionality of the unaltered lines of code untouched. Of course, additional techniques are very helpful, such as automated testing, high test coverage, architectural education/training, and a supportive project and enterprise organization.

1.3.3 Architectural Erosion Takes Place Unnoticed

A drawn-out process

Even with a capable development team, architectural erosion occurs unnoticed. How does this happen? Well, it's often a long, drawn-out process. During implementation, developers increasingly deviate from the architecture. In some cases, they do this consciously because the planned architecture does not meet the increasingly evident requirements. The complexity of the problem and the solution were underestimated and demands changed within the architecture, but there is no time to consistently follow these changes through for the entire system. In other cases, time and cost issues arise and must be solved so quickly that there is no time to develop a suitable design and rethink the architecture. Some developers are not even aware of the planned architecture, so they unintentionally violate it. For example, relationships are built between components that disregard prescribed public interfaces or run contrary to the modularization and layering of the software system. By the time you notice this creeping decay, it is high time to intervene!

Symptoms of severe architectural erosion

Once you have reached the nadir of architectural erosion, every change becomes unbearable. No-one wants to continue working on such a system. In his article *Design Principles and Design Pattern*, Robert C. Martin summed up these symptoms of a rotten system [Martin 2000]:

Rigidity

- **Rigidity:** The system is inflexible to modification. Each modification leads to a cascade of further adjustments in dependent modules. Developers are often unaware of what is happening in

the system and are uncomfortable with changes. What starts as a small adjustment or a small refactoring leads to an ever-increasing marathon of repairs in ever more modules. The developers chase the effects of their modifications in the source code and hope to have reached the end of the chain with every new realization.

Fragility

- **Fragility:** Changes to the system result in errors that have no obvious relationship to the modifications made. Each adjustment increases the probability of new subsequent errors in surprising locations. The fear of modification grows, and the impression is that the developers are no longer in control of the software.

Immobility

- **Immobility:** There are design and construction units that already solve a similar task as the one that is currently being implemented. However, these solutions cannot be reused because there is too much "baggage" surrounding the unit in question. A generic implementation or separation is also not possible because reconstructing the old units would be too complex and error-prone. Usually the required code is copied, as this requires less effort.

Viscosity

- **Viscosity:** If developers need to make an adjustment, there are usually several options. Some of these options preserve the design, while others break it. If such "hacks" are easier to implement than the design-preserving solution, the system is described as viscous.

Development teams must constantly fight these symptoms to keep their systems durable and make customizing and maintenance fun in the long run. If only the costs didn't exist ...

1.3.4 We Don't Pay Extra For Quality!

Architecture costs extra money

Many customers are surprised when their service providers—either external or in-house—tell them that they need more money to improve the architecture and thus the quality of the software system. Customers often say things like, "It already works! What do I gain if I spend money on quality?" or, "You got the contract because you promised you would deliver good quality. You can't demand more money for quality now". These are very unpleasant situations. As software developers and architects, our goal is to write software

with sustainable architecture and high quality. At this point, it is not easy to explain that an evolving architecture is an investment in the future and saves money in the long run.

Custom software development = unplannable

These situations often arise because the customer/management doesn't realize (or doesn't want to know) that custom software development is an unplannable process. If new, unprecedented software is developed, the essential complexity is difficult to master. The software itself, its use, and its integration into the context of work organization and changing business processes are unpredictable. Possible extensions or new forms of use cannot be foreseen. These are essential characteristics of custom software development!

Software ≠ industrially produced goods

Today, every software system is custom-developed software and integration into the customer's IT landscape is different every time. The technological and economic developments are so rapid that a software architecture and the resulting system that represents the ideal solution for today will reach its limits by tomorrow. These constraints lead to the conclusion that software is not an industrially manufactured product. Instead, it is a custom solution that makes sense at a given point in time, with an architecture that will hopefully endure for a long time but that must continue to evolve. This includes both functional and non-functional aspects, such as internal and external quality.

Fortunately, increasing numbers of customers are beginning to understand the terms "technical debt" and "sustainability.

1.3.5 Types of Technical Debt

Many types of technical debt and their variants are mentioned in discussions about technical debt. Four of these are relevant to this book:

Code Smell

- **Implementation debt:** The source code contains "code smells", such as long methods, code duplicates, and similar.

Structural Smell

- **Design and architecture debt:** The design of classes, packages, subsystems, layers, and modules is inconsistent or complex and does not fit the planned architecture.

Unit Tests

- **Test debt:** Tests are missing or only the positive case is tested. The test coverage with automated unit tests is low.

Documentation

- **Documentation debt:** There is no documentation, or the documentation that exists is incomplete or outdated. The overview of the architecture is not supported by the documents. Design decisions are not documented.

Basic requirement: low test debt

Most of the hints, suggestions, and good and bad examples you will find in this book relate to the first two types of technical debt. You will see how such debt arises and how it can be reduced. However, this debt can only be reduced safely if the level of test debt is low or is reduced while you work. In this respect, low test debt is a basic requirement. On the one hand, documentation of the architecture is a good basis for the architectural analysis and improvements dealt with in this book (i.e., low documentation debt helps with the analysis). On the other hand, architectural analysis also produces documentation for the analyzed system, thus also reducing documentation debt.

1.4 The Systems I Have Seen

After completing my computer science studies in 1995 I worked as a software developer, before moving on to software architecture, project management, and consultancy. Since 2002 I have often been invited to examine the quality of software systems. In the beginning I was only able to look at the source code, but this aspect has been tool-supported since 2004. Architectural analysis and improvement developed with thorough, tool-assisted checks of the source code according to certain criteria. In Chapters 2 and 4 you will see how analysis and improvement take place technically and organizationally.

Sizes and languages

In the course of time, I have reviewed systems written in Java (130), C++ (30), C# (70), ABAP (5), PHP (20), and PLSQL (10). TypeScript and JavaScript will soon be added to this list. Each of these programming languages has its own peculiarities, as we will see in Chapter 3. The size of a system (see fig. 1-2) also influences how the software architecture is (or should be) designed.

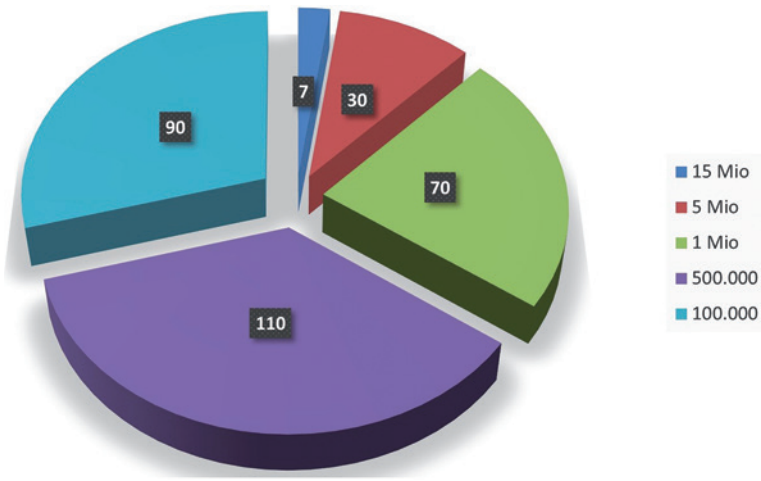

Figure 1-2
Sizes of analyzed systems in lines of code (LOC)

Lines of code

The "lines of code" (LOC) specification in figure 1-2 includes the lines of executable code, blank lines, and comments. If you exclude comments and blank lines, you need to subtract an average of 50 per cent of the LOC. Typically, the ratio between executable and non-executable code is between 40 and 60 per cent, depending on whether the development team placed begin and end markers ({}) on separate lines. The sizes of systems analyzed in this book are quoted for code written in Java/C#, C++, and PHP. These languages all have a similar "sentence length". ABAP programming is much more wordy and generates two or three times as much source code.

All these analysis have sharpened and deepened my understanding of software architecture and my expectations of how software systems should be built.

1.5 Who Is This Book For?

Programming

This book is written for architects and developers who work with source code on a daily basis. They will benefit most from this book because it points out potential problems in large and small systems as well as offering solutions.

Improving existing systems

Consultants with development experience, practicing architects, and development teams who want to methodically improve exist-

ing software solutions will find many references to large and small improvements in this book.

Learning for the future

Inexperienced developers will probably have trouble understanding the content in some places due to the complexity of the issues involved. However, they will still be able to learn the basics of how to build sustainable software architectures.

1.6 How To Use This Book?

The book consists of twelve chapters, some of which follow on from one another, but that can also be read separately.

Main contents

Figure 1-3 shows a schematic of the book's chapters. The two white chapters provide the basic framework of introduction and summary. The turquoise chapters are the theoretical parts of the book, and the dark blue chapter deals with organizational aspects. The light blue chapters contain many small and large practical examples.

Different Paths

Ideally, you will read the entire book from start to finish. However, if your time is limited, I recommend you read Chapters 1 and 2 first to give you a good foundation. You can then skip to Chapters 5 and/or 6 followed by Chapters 7, 8, 9, or 10. You can also jump

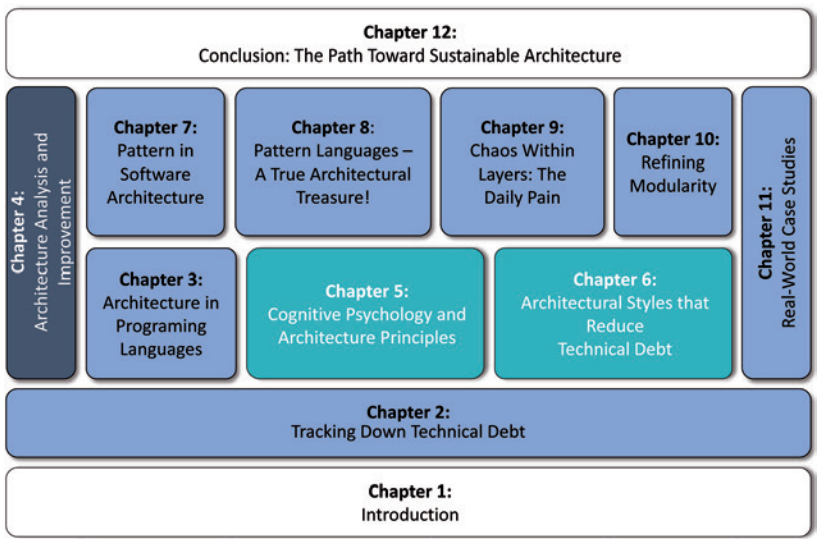

Figure 1-3
Structure of the book

directly from Chapter 2 to Chapters 4 or 8, and dive right into the procedure of architectural analysis or the case studies.

Chapter 1 lays the foundation for understanding sustainable architectures and technical debt. *Chapter 2* shows how to find and reduce technical debt in existing architectures. *Chapter 3* explains the specialties of programming languages in architectural analysis. *Chapter 4* explains which roles in architectural analysis and improvement have to work together to achieve a valuable result, and how architectures can be compared using the Modularity Maturity Index (MMI). *Chapter 5* deals with the question of how large structures must be designed so that people can quickly navigate their way around them. Cognitive psychology gives us clues as to which specifications lead to architectures that can be quickly grasped and understood. *Chapter 6* presents the architectural styles commonly used today. With their rules, they provide guardrails for software architectures. *Chapters 7, 8, 9,* and *10* describe the findings from various practical, real-world analysis and consultations. *Chapter 11* contains seven exemplary (anonymized) case studies that I find particularly interesting. To conclude, *Chapter 12* contains a brief summary of how architects, development teams, and management should proceed to improve the quality of their architecture. The appendix presents a range of analysis tools that I have enjoyed using in the course of my everyday work.