O'REILLY®

C++ Software Design

Design Principles and Patterns for High-Quality Software

Klaus Iglberger

O'REILLY®

C++ Software Design

Good software design is essential for the success of your project, but designing software is hard to do. You need to have a deep understanding of the consequences of design decisions and a good overview of available design alternatives. With this book, experienced C++ developers will get a thorough, practical, and unparalleled overview of software design with this modern language.

C++ trainer and consultant Klaus Iglberger explains how you can manage dependencies and abstractions, improve changeability and extensibility of software entities, and apply and implement modern design patterns to help you take advantage of today's possibilities. Software design is the most essential aspect of a software project because it impacts the software's most important properties: maintainability, changeability, and extensibility.

- Learn how to evaluate your code with respect to software design
- Understand what software design is, including design goals such as changeability and extensibility
- Explore the advantages and disadvantages of each design approach
- Learn how design patterns help solve problems and express intent
- Choose the right form of a design pattern to get the most out of its advantages

"This book will raise virtually every C++ programmer's game. It is packed with practical design patterns and fascinating ideas. I learned far more from this book than I ever expected."

—**Mark Summerfield**
Owner of Qtrac Ltd

Klaus Iglberger is a freelance C++ trainer and consultant. He shares his 15 years of C++ expertise in popular training courses around the world and is a frequent speaker at C++ conferences. Since earning his PhD in 2010, he has focused on large-scale software design and on improving the maintainability of software.

PROGRAMMING

US $79.99 CAN $99.99
ISBN: 978-1-098-11316-2

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

Praise for *C++ Software Design*

Even after knowing design patterns for a long time, this book showed me a surprisingly large number of new aspects on how to use them properly in the context of C++ and the SOLID principles.

—Matthias Dörfel, CTO at INCHRON AG

I really enjoyed reading the book! Studying the guidelines made me reconsider my code and improve it by applying them. Can you ask for more?

—Daniela Engert, senior software engineer at GMH Prüftechnik GmbH

One of the most entertaining and useful software design books I've read in a long while.

—Patrice Roy, professeur, Collège Lionel-Groulx

It has been over 25 years since the Gang of Four's Design Patterns changed the way programmers think about software. This book changed the way I think about Design Patterns.

—Stephan Weller, Siemens Digital Industries Software

C++ Software Design

Design Principles and Patterns for High-Quality Software

Klaus Iglberger

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

C++ Software Design
by Klaus Iglberger

Copyright © 2022 Klaus Iglberger. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Amanda Quinn
Development Editor: Shira Evans
Production Editor: Kate Galloway
Copyeditor: Sonia Saruba
Proofreader: Piper Editorial Consulting, LLC
Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

September 2022: First Edition

Revision History for the First Edition
2022-09-21: First Release
2023-12-08: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098113162* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *C++ Software Design*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-11316-2

[LSI]

Table of Contents

Preface

In your hands you're holding the C++ book that I wish I would have had many years ago. Not as one of my first books, no, but as an advanced book, after I had already digested the language mechanics and was able to think beyond the C++ syntax. Yes, this book would have definitely helped me better understand the fundamental aspects of maintainable software, and I'm confident that it will help you too.

Why I Wrote This Book

By the time I was really digging into the language (that was a few years after the first C++ standard had been released), I had read pretty much every C++ book there was. But despite the fact that many of these books were great and definitely paved the way for my current career as a C++ trainer and consultant, they were too focused on the little details and the implementation specifics, and too far away from the bigger picture of maintainable software.

At the time, very few books truly focused on the bigger picture, dealing with the development of large software systems. Among these were John Lakos's *Large Scale C++ Software Design*,[1] a great but literally heavy introduction to dependency management, and the so-called Gang of Four book, which is *the* classic book on software design patterns.[2] Unfortunately, over the years, this situation hasn't really changed: most books, talks, blogs, etc., primarily focus on language mechanics and features—the small details and specifics. Very few, and in my opinion way too few, new releases focus on maintainable software, changeability, extensibility, and testability. And if they try to, they unfortunately quickly fall back into the common habit of explaining language mechanics and demonstrating features.

1 John Lakos, *Large-Scale C++ Software Design* (Addison-Wesley, 1996).

2 Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994).

This is why I've written this book. A book that does not, in contrast to most others, spend time on the mechanics or the many features of the language, but primarily focuses on changeability, extensibility, and testability of software in general. A book that does not pretend that the use of new C++ standards or features will make the difference between good or bad software, but instead clearly shows that it is the management of dependencies that is decisive, that the dependencies in our code decide between it being good or bad. As such, it is a rare kind of book in the world of C++ indeed, as it focuses on the bigger picture: software design.

What This Book Is About

Software Design

From my point of view, good software design is the essence of every successful software project. Yet still, despite its fundamental role, there is so little literature on the topic, and very little advice on what to do and how to do things right. Why? Well, because it's difficult. Very difficult. Probably the most difficult facet of writing software that we have to face. And that's because there is no single "right" solution, no "golden" advice to pass on through the generations of software developers. It always depends.

Despite this limitation, I will give advice on how to design good, high-quality software. I will provide design principles, design guidelines, and design patterns that will help you to better understand how to manage dependencies and turn your software into something you can work with for decades. As stated before, there is no "golden" advice, and this book doesn't hold any ultimate or perfect solution. Instead, I try to show the most fundamental aspects of good software, the most important details, the diversity and the pros and the cons of different designs. I will also formulate intrinsic design goals and demonstrate how to achieve these goals with Modern C++.

Modern C++

For more than a decade, we've been celebrating the advent of Modern C++, applauding the many new features and extensions of the language, and by doing so, creating the impression that Modern C++ will help us solve all software-related problems. Not so in this book. This book does not pretend that throwing a few smart pointers at the code will make the code "Modern" or automatically yield good design. Also, this book won't show Modern C++ as an assortment of new features. Instead, it will show how the philosophy of the language has evolved and the way we implement C++ solutions today.

But of course, we will also see code. Lots of it. And of course this book will make use of the features of newer C++ standards (including C++20). However, it will also make an effort to emphasize that the design is independent of the implementation details and the used features. New features don't change the rules about what is good design or bad design; they merely change the way we implement good design. They make it easier to implement good design. So this book shows and discusses implementation details, but (hopefully) doesn't get lost in them and always remains focused on the big picture: software design and design patterns.

Design Patterns

As soon as you start mentioning design patterns, you inadvertently conjure up the expectation of object-oriented programming and inheritance hierarchies. Yes, this book will show the object-oriented origin of many design patterns. However, it will put a strong emphasis on the fact that there isn't just one way to make good use of a design pattern. I will demonstrate how the implementation of design patterns has evolved and diversified, making use of many different paradigms, including object-oriented programming, generic programming, and functional programming. This book acknowledges the reality that there is no one true paradigm and does not pretend that there is only one single approach, one ever-working solution for all problems. Instead it tries to show Modern C++ for what it truly is: the opportunity to combine all paradigms, weave them into a strong and durable net, and create software design that will last through the decades.

I hope this book proves to be the missing piece in C++ literature. I hope it helps you as much as it would have helped me. I hope that it holds some answers you have been looking for and provides you with a couple of key insights that you were missing. And I also hope that this book keeps you somewhat entertained and motivated to read everything. Most importantly, however, I hope that this book will show you the importance of software design and the role that design patterns play. Because, as you will see, design patterns are everywhere!

Who This Book Is For

This book is of value to every C++ developer. In particular, it is for every C++ developer interested in understanding the usual problems of maintainable software and learning about common solutions to these problems (and I assume that is indeed *every* C++ developer). However, this book is not a C++ beginner's book. In fact, most of the guidelines in this book require some experience with software development in general and C++ in particular. For instance, I assume that you have a firm grasp of the language mechanics of inheritance hierarchies and some experience with templates. Then I can reach for the corresponding features whenever necessary and appropriate. Once in a while, I will even reach for some C++20 features (in particular

C++20 concepts). However, as the focus is on software design, I will rarely dwell on explaining a particular feature, so if a feature is unknown to you, please consult your favorite C++ language reference. Only occasionally will I add some reminders, mostly about common C++ idioms (such as the Rule of 5).

How This Book Is Structured

This book is organized into chapters, each containing several guidelines. Each guideline focuses on one key aspect of maintainable software or one particular design pattern. Hence, the guidelines represent the major takeaways, the aspects that I hope bring the most value to you. They're written such that you can read all of them from front to back, but since they're only loosely coupled, they enable you to also start with the guideline that attracts your attention. Still, they're not independent. Therefore, each guideline contains the necessary cross-references to other guidelines to show you that everything is connected.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/igl42/cpp_software_design*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*C++ Software Design* by Klaus Iglberger (O'Reilly). Copyright 2022 Klaus Iglberger, 978-1-098-11316-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

> O'Reilly Media, Inc.
> 1005 Gravenstein Highway North
> Sebastopol, CA 95472
> 800-998-9938 (in the United States or Canada)
> 707-829-0515 (international or local)
> 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/c-plus-plus*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Follow us on Twitter: *http://twitter.com/oreillymedia*.

Watch us on YouTube: *http://youtube.com/oreillymedia*.

Acknowledgments

A book such as this is never the achievement of a single individual. On the contrary, I have to explicitly thank many people who helped me in different ways to make this book a reality. First and foremost, I want to express my deep gratitude to my wife, Steffi, who read through the entire book without even knowing C++. And who took care of our two kids to give me the necessary calm to bring all of this information to paper (I am still not sure which of these two was the bigger sacrifice).

A special thank-you goes to my reviewers, Daniela Engert, Patrice Roy, Stefan Weller, Mark Summerfield, and Jacob Bandes-Storch, for investing their valuable time to make this a better book by constantly challenging my explanations and examples.

A big thank-you also goes to Arthur O'Dwyer, Eduardo Madrid, and Julian Schmidt for their input and feedback about the Type Erasure design pattern, and to Johannes Gutekunst for the discussions on software architecture and documentation.

Furthermore, I want to say thank you to my two cold readers, Matthias Dörfel and Vittorio Romeo, who helped catch many last-second mistakes (and indeed they did).

Last, but definitely not least, a big thank-you goes to my editor, Shira Evans, who has spent many hours giving invaluable advice about making the book more consistent and more fun to read.

CHAPTER 1

The Art of Software Design

What is software design? And why should you care about it? In this chapter, I will set the stage for this book on software design. I will explain software design in general, help you understand why it is vitally important for the success of a project, and why it is the one thing you should get right. But you will also see that software design is complicated. Very complicated. In fact, it is the most complicated part of software development. Therefore, I will also explain several software design principles that will help you to stay on the right path.

In "Guideline 1: Understand the Importance of Software Design" on page 2, I will focus on the big picture and explain that software is expected to change. Consequently, software should be able to cope with change. However, that is much easier said than done, since in reality, coupling and dependencies make our life as a developer so much harder. That problem is addressed by software design. I will introduce software design as the art of managing dependencies and abstractions—an essential part of software engineering.

In "Guideline 2: Design for Change" on page 11, I will explicitly address coupling and dependencies and help you understand how to design for change and how to make software more adaptable. For that purpose, I will introduce both the *Single-Responsibility Principle (SRP)* and the *Don't Repeat Yourself (DRY)* principle, which help you to achieve this goal.

In "Guideline 3: Separate Interfaces to Avoid Artificial Coupling" on page 24, I will expand the discussion about coupling and specifically address coupling via interfaces. I will also introduce the *Interface Segregation Principle (ISP)* as a means to reduce artificial coupling induced by interfaces.

In "Guideline 4: Design for Testability" on page 28, I will focus on testability issues that arise as a result of artificial coupling. In particular, I will raise the question of how to test a private member function and demonstrate that the one true solution is a consequent application of separation of concerns.

In "Guideline 5: Design for Extension" on page 35, I will address an important kind of change: extensions. Just as code should be easy to change, it should also be easy to extend. I will give you an idea how to achieve that goal, and I will demonstrate the value of the *Open-Closed Principle (OCP)*.

Guideline 1: Understand the Importance of Software Design

If I were to ask you which code properties are most important to you, you would, after some thinking, probably say things like readability, testability, maintainability, extensibility, reusability, and scalability. And I would completely agree. But now, if I were to ask you how to achieve these goals, there is a good chance that you would start to list some C++ features: RAII, algorithms, lambdas, modules, and so on.

Features Are Not Software Design

Yes, C++ offers a lot of features. A lot! Approximately half of the almost 2,000 pages of the printed C++ standard are devoted to explaining language mechanics and features.[1] And since the release of C++11, there is the explicit promise that there will be more: every three years, the C++ standardization committee blesses us with a new C++ standard that ships with additional, brand-new features. Knowing that, it doesn't come as a big surprise that in the C++ community there's a very strong emphasis on features and language mechanics. Most books, talks, and blogs are focused on features, new libraries, and language details.[2]

It almost feels as if features are the most important thing about programming in C++, and crucial for the success of a C++ project. But honestly, they are not. Neither the knowledge about all the features nor the choice of the C++ standard is responsible for the success of a project. No, you should not expect features to save your project. On the contrary: a project can be very successful even if it uses an older C++

1 But of course you would never even try to print the current C++ standard. You would either use a PDF of the official C++ standard or use the current working draft. For most of your daily work, however, you might want to refer to the C++ reference site.

2 Unfortunately, I can't present any numbers, as I can hardly say that I have a complete overview of the vast realm of C++. On the contrary, I might not even have a complete overview of the sources I'm aware of! So please consider this as my personal impression and the way I perceive the C++ community. You may have a different impression.

standard, and even if only a subset of the available features are used. Leaving aside the human aspects of software development, much more important for the question about success or failure of a project is the overall *structure* of the software. It is the structure that is ultimately responsible for maintainability: how easy is it to change code, extend code, and test code? Without the ability to easily change code, add new functionality, and have confidence in its correctness due to tests, a project is at the end of its lifecycle. The structure is also responsible for the scalability of a project: how large can the project grow before it collapses under its own weight? How many people can work on realizing the vision of the project before they step on one another's toes?

The overall structure is the design of a project. The design plays a much more central role in the success of a project than any feature could ever do. Good software is not primarily about the proper use of any feature; rather, it is about solid architecture and design. Good software design can tolerate some bad implementation decisions, but bad software design cannot be saved by the heroic use of features (old or new) alone.

Software Design: The Art of Managing Dependencies and Abstractions

Why is software design so important for the quality of a project? Well, assuming everything works perfectly right now, as long as nothing changes in your software and as long as nothing needs to be added, you are fine. However, that state will likely not last for long. It's reasonable to expect that something will change. After all, the one constant in software development is change. Change is the driving force behind all our problems (and also most of our solutions). That's why software is called *soft*ware: because in comparison to hardware, it is soft and malleable. Yes, *soft*ware is expected to be easily adapted to the ever-changing requirements. But as you may know, in reality this expectation might not always be true.

To illustrate this point, let's imagine that you select an issue from your issue tracking system that the team has rated with an expected effort of 2. Whatever a 2 means in your own project(s), it most certainly does not sound like a big task, so you are confident that this will be done quickly. In good faith, you first take some time to understand what is expected, and then you start by making a change in some entity `A`. Because of immediate feedback from your tests (you are lucky to have tests!), you are quickly reminded that you also have to address the issue in entity `B`. That is surprising! You did not expect that `B` was involved at all. Still, you go ahead and adapt `B` anyway. However, again unexpectedly, the nightly build reveals that this causes `C` and `D` to stop working. Before continuing, you now investigate the issue a little deeper and find that the roots of the issue are spread through a large portion of the codebase. The small, initially innocent-looking task has evolved into a large, potentially risky code modification.[3] Your confidence in resolving the issue quickly is gone. And your plans for the rest of the week are as well.

Maybe this story sounds familiar to you. Maybe you can even contribute a few war stories of your own. Indeed, most developers have similar experiences. And most of these experiences have the same source of trouble. Usually the problem can be reduced to a single word: *dependencies*. As Kent Beck has expressed in his book on test-driven development:[4]

> Dependency is the key problem in software development at all scales.

Dependencies are the bane of every software developer's existence. "But of course there are dependencies," you argue. "There will always be dependencies. How else should different pieces of code work together?" And of course, you are correct. Different pieces of code need to work together, and this interaction will always create some form of coupling. However, while there are necessary, unavoidable dependencies, there are also artificial dependencies that we accidentally introduce because we lack an understanding of the underlying problem, don't have a clear idea of the bigger picture, or just don't pay enough attention. Needless to say, these artificial dependencies hurt. They make it harder to understand our software, change software, add new features, and write tests. Therefore, one of the primary tasks, if not *the* primary task, of a software developer is to keep artificial dependencies at a minimum.

This minimization of dependencies is the goal of software architecture and design. To state it in the words of Robert C. Martin:[5]

> The goal of software architecture is to minimize the human resources required to build and maintain the required system.

Architecture and design are the tools needed to minimize the work effort in any project. They deal with dependencies and reduce the complexity via abstractions. In my own words:[6]

> Software design is the art of managing interdependencies between software components. It aims at minimizing artificial (technical) dependencies and introduces the necessary abstractions and compromises.

3 Whether or not the code modification is risky may very much depend on your test coverage. A good test coverage may actually absorb some of the damage bad software design may cause.

4 Kent Beck, *Test-Driven Development: By Example* (Addison-Wesley, 2002).

5 Robert C. Martin, *Clean Architecture* (Addison-Wesley, 2017).

6 These are indeed my own words, as there is no single, common definition of software design. Consequently, you may have your own definition of what software design entails and that is perfectly fine. However, note that this book, including the discussion of design patterns, is based on my definition.

Yes, software design is an art. It's not a science, and it doesn't come with a set of easy and clear answers.[7] Too often the big picture of design eludes us, and we are overwhelmed by the complex interdependencies of software entities. But we are trying to deal with this complexity and reduce it by introducing the right kind of abstractions. This way, we keep the level of detail at a reasonable level. However, too often individual developers on the team may have a different idea of the architecture and the design. We might not be able to implement our own vision of a design and be forced to make compromises in order to move forward.

The term *abstraction* is used in different contexts. It's used for the organization of functionality and data items into data types and functions. But it's also used to describe the modeling of common behavior and the representation of a set of requirements and expectations. In this book on software design, I will primarily use the term for the latter (see in particular Chapter 2).

Note that the words *architecture* and *design* can be interchanged in the preceding quotes, since they're very similar and share the same goals. Yet they aren't the same. The similarities, but also differences, become clear if you take a look at the three levels of software development.

The Three Levels of Software Development

Software Architecture and *Software Design* are just two of the three levels of software development. They are complemented by the level of *Implementation Details*. Figure 1-1 gives an overview of these three levels.

To give you a feeling for these three levels, let's start with a real-world example of the relationship among architecture, design, and implementation details. Consider yourself to be in the role of an architect. And no, please don't picture yourself in a comfy chair in front of a computer with a hot coffee next to you, but picture yourself outside at a construction site. Yes, I'm talking about an architect for buildings.[8] As such an architect, you would be in charge of all the important properties of a house: its integration into the neighborhood, its structural integrity, the arrangement of rooms, plumbing, etc. You would also take care of a pleasing appearance and functional qualities—perhaps a large living room, easy access between the kitchen and the dining room, and so on. In other words, you would be taking care of the overall

7 Just to be clear: computer science is a science (it's in the name). Software *engineering* appears to be a hybrid form of science, craft, and art. And one aspect of the latter is software *design*.

8 With this metaphor, I'm not trying to imply that architects for buildings work at the construction site all day. Very likely, such an architect spends as much time in a comfy chair and in front of a computer as people like you and me. But I think you get the point.

architecture, the things that would be hard to change later, but you would also deal with the smaller design aspects concerning the building. However, it's hard to tell the difference between the two: the boundary between architecture and design appears to be fluid and is not clearly separated.

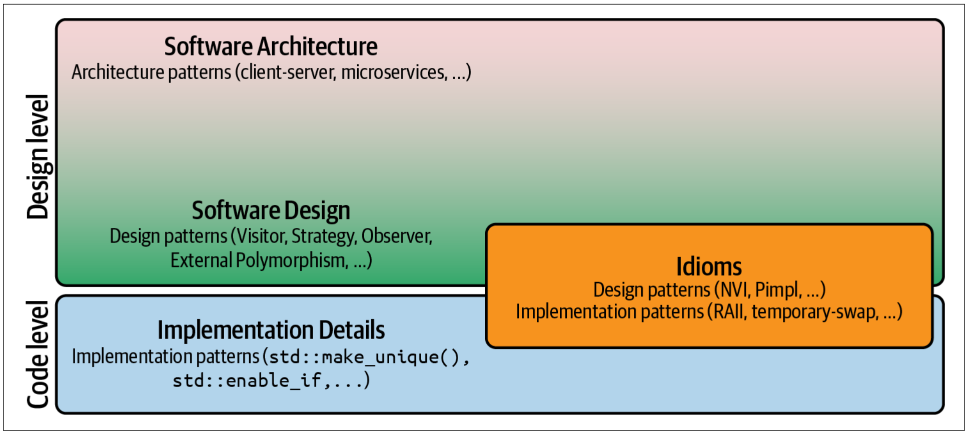
Design level
Code level
Software Architecture
Architecture patterns (client-server, microservices, ...)
Software Design
Design patterns (Visitor, Strategy, Observer, External Polymorphism, ...)
Idioms
Design patterns (NVI, Pimpl, ...)
Implementation patterns (RAII, temporary-swap, ...)
Implementation Details
Implementation patterns (std::make_unique(), std::enable_if, ...)

Figure 1-1. The three levels of software development: Software Architecture, Software Design, and Implementation Details. Idioms can be design or implementation patterns.

These decisions would be the end of your responsibility, however. As an architect, you wouldn't worry about where to place the refrigerator, the TV, or other furniture. You wouldn't deal with all the nifty details about where to place pictures and other pieces of decoration. In other words, you wouldn't handle the details; you would just make sure that the homeowner has the necessary structure to live well.

The furniture and other "nifty details" in this metaphor correspond to the lowest and most concrete level of software development, the implementation details. This level handles how a solution is implemented. You choose the necessary (and available) C++ standard or any subset of it, as well as the appropriate features, keywords, and language specifics to use, and deal with aspects such as memory acquisition, exception safety, performance, etc. This is also the level of *implementation patterns*, such as `std::make_unique()` as a *factory function*, `std::enable_if` as a recurring solution to explicitly benefit from SFINAE, etc.[9]

9 Substitution Failure Is Not An Error (SFINAE) is a basic template mechanism commonly used as a substitute for C++20 concepts to constrain templates. For an explanation of SFINAE and `std::enable_if` in particular, refer to your favorite textbook about C++ templates. If you don't have any, a great choice is the C++ template bible: David Vandevoorde, Nicolai Josuttis, and Douglas Gregor's *C++ Templates: The Complete Guide* (Addison-Wesley).

In software design, you start to focus on the big picture. Questions about maintainability, changeability, extensibility, testability, and scalability are more pronounced on this level. Software design primarily deals with the interaction of software entities, which in the previous metaphor are represented by the arrangement of rooms, doors, pipes, and cables. At this level, you handle the physical and logical dependencies of components (classes, function, etc.).[10] It's the level of design patterns such as Visitor, Strategy, and Decorator that define a dependency structure among software entities, as explained in Chapter 3. These patterns, which usually are transferable from language to language, help you break down complex things into digestible pieces.

Software Architecture is the fuzziest of the three levels, the hardest to put into words. This is because there is no common, universally accepted definition of software architecture. While there may be many different views on what exactly an architecture is, there is one aspect that everyone seems to agree on: architecture usually entails the big decisions, the aspects of your software that are among the hardest things to change in the future:

> Architecture is the decisions that you wish you could get right early in a project, but that you are not necessarily more likely to get them right than any other.[11]
>
> —Ralph Johnson

In Software Architecture, you use architectural patterns such as *client-server architecture*, *microservices*, and so on.[12] These patterns also deal with the question of how to design systems, where you can change one part without affecting any other parts of your software. Similar to *Software design* patterns, they define and address the structure and interdependencies among software entities. In contrast to design patterns, though, they usually deal with the key players, the big entities of your software (e.g., modules and components instead of classes and functions).

From this perspective, Software Architecture represents the overall strategy of your software approach, whereas Software Design is the tactics to make the strategy work. The problem with this picture is that there is no definition of "big." Especially with the advent of microservices, it becomes more and more difficult to draw a clear line between small and big entities.[13]

10 For a lot more information on physical and logical dependency management, see John Lakos's "dam" book, *Large-Scale C++ Software Development: Process and Architecture* (Addison-Wesley).

11 Martin Fowler, "Who Needs an Architect?" *IEEE Software*, 20, no. 5 (2003), 11–13, *https://doi.org/10.1109/MS.2003.1231144*.

12 A very good introduction to microservices can be found in Sam Newman's book *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. (O'Reilly).

13 Mark Richards and Neal Ford, *Fundamentals of Software Architecture: An Engineering Approach* (O'Reilly, 2020).

Thus, architecture is often described as what expert developers in a project perceive as the key decisions.

What makes the separation between architecture, design, and details a little more difficult is the concept of an *idiom*. An *idiom* is a commonly used but language-specific solution for a recurring problem. As such, an idiom also represents a pattern, but it could be either an *implementation pattern* or a *design pattern*.[14] More loosely speaking, C++ idioms are the best practices of the C++ community for either design or implementation. In C++, most idioms fall into the category of implementation details. For instance, there is the *copy-and-swap idiom* that you may know from the implementation of a copy assignment operator, and the *RAII idiom* (Resource Acquisition Is Initialization—you should definitely be familiar with this; if not, please see your second-favorite C++ book[15]). None of these idioms introduce an abstraction, and none of them help to decouple. Still, they are indispensable to implement good C++ code.

I hear you ask, "Could you be a little more specific, please? Isn't RAII also providing some form of decoupling? Doesn't it decouple resource management from business logic?" You're correct: RAII separates resource management and business logic. However, it doesn't achieve this by means of decoupling, i.e., abstraction, but by means of encapsulation. Both abstraction and encapsulation help you make complex systems easier to understand and change, but while abstraction solves the problems and issues that arise at the Software Design level, encapsulation solves the problems and issues that arise at the Implementation Details level. To quote Wikipedia:

> The advantages of RAII as a resource management technique are that it provides encapsulation, exception safety […], and locality […]. Encapsulation is provided because resource management logic is defined once in the class, not at each call site.

While most idioms fall into the category of Implementation Details, there are also idioms that fall into the category of Software Design. Two examples are the *Non-Virtual Interface (NVI) idiom* and the *Pimpl idiom*. These two idioms are based on two classic design patterns: the *Template Method* design pattern and the *Bridge*

14 The term *implementation pattern* was first used in Kent Beck's book *Implementation Patterns* (Addison-Wesley). In this book, I'm using that term to provide a clear distinction from the term *design pattern*, since the term *idiom* may refer to a pattern on either the Software Design level or the Implementation Details level. I will use the term consistently to refer to commonly used solutions on the Implementation Details level.

15 Second-favorite after this one, of course. If this is your only book, then you might refer to the classic *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd ed., by Scott Meyers (Addison-Wesley).

design pattern, respectively.[16] They introduce an abstraction and help decouple and design for change and extensions.

The Focus on Features

If software architecture and software design are of such importance, then why are we in the C++ community focusing so strongly on features? Why do we create the illusion that C++ standards, language mechanics, and features are decisive for a project? I think there are three strong reasons for that. First, because there are so many features, with sometimes complex details, we need to spend a lot of time talking about how to use all of them properly. We need to create a common understanding on which use is good and which use is bad. We as a community need to develop a sense of idiomatic C++.

The second reason is that we might put the wrong expectations on features. As an example, let's consider C++20 modules. Without going into details, this feature may indeed be considered the biggest technical revolution since the beginning of C++. Modules may at last put the questionable and cumbersome practice of including header files into source files to an end.

Due to this potential, the expectations for that feature are enormous. Some people even expect modules to save their project by fixing their structural issues. Unfortunately, modules will have a hard time satisfying these expectations: modules don't improve the structure or design of your code but can merely represent the current structure and design. Modules don't repair your design issues, but they may be able to make the flaws visible. Thus, modules simply cannot save your project. So indeed, we may be putting too many or the wrong expectations on features.

And last, but not least, the third reason is that despite the huge amount of features and their complexity, in comparison to the complexity of software design, the complexity of C++ features is small. It's much easier to explain a given set of rules for features, regardless of how many special cases they contain, than it is to explain the best way to decouple software entities.

While there is usually a good answer to all feature-related questions, the common answer in software design is "It depends." That answer might not even be evidence of inexperience, but of the realization that the best way to make code more maintainable, changeable, extensible, testable, and scalable heavily depends on many project-

16 The Template Method and Bridge design patterns are 2 of the 23 classic design patterns introduced in the so-called Gang of Four (GoF) book by Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. I won't go into detail about the Template Method in this book, but you'll find good explanations in various textbooks, including the GoF book itself. I will, however, explain the Bridge design pattern in "Guideline 28: Build Bridges to Remove Physical Dependencies" on page 250.

specific factors. The decoupling of the complex interplay between many entities may indeed be one of the most challenging endeavors that mankind has ever faced:

> Design and programming are human activities; forget that and all is lost.[17]

To me, a combination of these three reasons is why we focus on features so much. But please, don't get me wrong. That's not to say that features are not important. On the contrary, features *are* important. And yes, it's necessary to talk about features and learn how to use them correctly, but once again, they alone do not save your project.

The Focus on Software Design and Design Principles

While features are important, and while it is of course good to talk about them, software design is more important. Software design is essential. I would even argue that it's the foundation of the success of our projects. Therefore, in this book I will make the attempt to truly focus on software design and design principles instead of features. Of course I will still show good and up-to-date C++ code, but I won't force the use of the latest and greatest language additions.[18] I *will* make use of some new features when it is reasonable and beneficial, such as C++20 concepts, but I will *not* pay attention to `noexcept`, or use `constexpr` everywhere.[19] Instead I will try to tackle the difficult aspects of software. I will, for the most part, focus on software design, the rationale behind design decisions, design principles, managing dependencies, and dealing with abstractions.

In summary, software design is the critical part of writing software. Software developers should have a good understanding of software design to write good, maintainable software. Because after all, good software is low-cost, and bad software is expensive.

Guideline 1: Understand the Importance of Software Design

- Treat software design as an essential part of writing software.
- Focus less on C++ language details and more on software design.
- Avoid unnecessary coupling and dependencies to make software more adaptable to frequent changes.
- Understand software design as the art of managing dependencies and abstractions.

17 Bjarne Stroustrup, *The C++ Programming Language*, 3rd ed. (Addison-Wesley, 2000).

18 Kudos to John Lakos, who argues similarly and uses C++98 in his book, *Large-Scale C++ Software Development: Process and Architecture* (Addison-Wesley).

19 Yes, Ben and Jason, you have read correctly, I will not `constexpr` ALL the things. See Ben Deane and Jason Turner, "constexpr ALL the things", CppCon 2017.

- Consider the boundary between software design and software architecture as fluid.

Guideline 2: Design for Change

One of the essential expectations for good software is its ability to change easily. This expectation is even part of the word *soft*ware. *Soft*ware, in contrast to *hard*ware, is expected to be able to adapt easily to changing requirements (see also "Guideline 1: Understand the Importance of Software Design" on page 2). However, from your own experience you may be able to tell that often it is not easy to change code. On the contrary, sometimes a seemingly simple change turns out to be a week-long endeavor.

Separation of Concerns

One of the best and proven solutions to reduce artificial dependencies and simplify change is to separate concerns. The core of the idea is to split, segregate, or extract pieces of functionality:[20]

> Systems that are broken up into small, well-named, understandable pieces enable faster work.

The intent behind separation of concerns is to better understand and manage complexity and thus design more modular software. This idea is probably as old as software itself and hence has been given many different names. For instance, the same idea is called *orthogonality* by the Pragmatic Programmers.[21] They advise separating orthogonal aspects of software. Tom DeMarco calls it *cohesion*:[22]

> Cohesion is a measure of the strength of association of the elements inside a module. A highly cohesive module is a collection of statements and data items that should be treated as a whole because they are so closely related. Any attempt to divide them up would only result in increased coupling and decreased readability.

In the *SOLID* principles,[23] one of the most established sets of design principles, the idea is known as the *Single-Responsibility Principle (SRP)*:

20 Michael Feathers, *Working Effectively with Legacy Code* (Addison-Wesley, 2013).

21 David Thomas and Andrew Hunt, *The Pragmatic Programmer: Your Journey to Mastery*, 20th Anniversary Edition (Addison-Wesley, 2019).

22 Tom DeMarco, *Structured Analysis and System Specification* (Prentice Hall, 1979).

23 SOLID is an acronym of acronyms, an abbreviation of the five principles described in the next few guidelines: SRP, OCP, LSP, ISP, and DIP.

A class should have only one reason to change.[24]

Although the concept is old and is commonly known under many names, many attempts to explain separation of concerns raise more questions than answers. This is particularly true for the SRP. The name of this design principle alone raises questions: what is a responsibility? And what is a *single* responsibility? A common attempt to clarify the vagueness about SRP is the following:

Everything should do just one thing.

Unfortunately this explanation is hard to outdo in terms of vagueness. Just as the word *responsibility* doesn't carry a lot of meaning, *just one thing* doesn't help to shed any more light on it.

Irrespective of the name, the idea is always the same: group only those things that truly belong together, and separate everything that does not strictly belong. Or in other words: separate those things that change for different reasons. By doing this, you reduce artificial coupling between different aspects of your code and it helps you make your software more adaptable to change. In the best case, you can change a particular aspect of your software in exactly one place.

An Example of Artificial Coupling

Let's shed some light on separation of concerns by means of a code example. And I do have a great example indeed: I present to you the abstract `Document` class:

```
//#include <some_json_library.h>  // Potential physical dependency

class Document
{
 public:
   // ...
   virtual ~Document() = default;

   virtual void exportToJSON( /*...*/ ) const = 0;  ❶
   virtual void serialize( ByteStream&, /*...*/ ) const = 0;  ❷
   // ...
};
```

This sounds like a very useful base class for all kinds of documents, doesn't it? First, there is the `exportToJSON()` function (❶). All deriving classes will have to implement the `exportToJSON()` function in order to produce a JSON file from the document. That will prove to be pretty useful: without having to know about a particular kind of

24 The first book on the SOLID principles was Robert C. Martin's *Agile Software Development: Principles, Patterns, and Practices* (Pearson). A newer and much cheaper alternative is *Clean Architecture*, also from Robert C. Martin (Addison-Wesley).

document (and we can imagine that we will eventually have PDF documents, Word documents, and many more), we can always export in JSON format. Nice! Second, there is a `serialize()` function (❷). This function lets you transform a `Document` into bytes via a `ByteStream`. You can store these bytes in some persistent system, like a file or a database. And of course we can expect that there are many other, useful functions available that will allow us to pretty much use this document for everything.

However, I can see the frown on your face. No, you don't look particularly convinced that this is good software design. It may be because you're just very suspicious about this example (it simply looks too good to be true). Or it may be that you've learned the hard way that this kind of design eventually leads to trouble. You may have experienced that using the common object-oriented design principle to bundle the data and the functions that operate on them may easily lead to unfortunate coupling. And I agree: despite the fact that this base class looks like a great all-in-one package, and even looks like it has everything that we might ever need, this design will soon lead to trouble.

This is bad design because it contains many dependencies. Of course there are the obvious, direct dependencies, as for instance the dependency on the `ByteStream` class. However, this design also favors the introduction of artificial dependencies, which will make subsequent changes harder. In this case, there are three kinds of artificial dependencies. Two of these are introduced by the `exportToJSON()` function, and one by the `serialize()` function.

First, `exportToJSON()` needs to be implemented in the derived classes. And yes, there is no choice, because it is a pure virtual function (denoted by the sequence `= 0`, the so-called *pure specifier*). Since derived classes will very likely not want to carry the burden of implementing JSON exports manually, they will rely on an external, third-party JSON library: *json*, *rapidjson*, or *simdjson*. Whatever library you choose for that purpose, because of the `exportToJSON()` member function, deriving documents would suddenly depend on this library. And, very likely, all deriving classes would depend on the same library, for consistency reasons alone. Thus, the deriving classes are not really independent; they are artificially coupled to a particular design decision.[25] Also, the dependency on a specific JSON library would definitely limit the reusability of the hierarchy, because it would no longer be lightweight. And switching to another library would cause a major change because all deriving classes would have to be adapted.[26]

25 Don't forget that the design decisions taken by that external library may impact your own design, which would obviously increase the coupling.

26 That includes the classes that other people may have written, i.e., classes that you do not control. And no, the other people won't be happy about the change. Thus, the change may be *really* difficult.

Of course, the same kind of artificial dependency is introduced by the `serialize()` function. It's likely that `serialize()` will also be implemented in terms of a third-party library, such as protobuf or Boost.serialization. This considerably worsens the dependency situation because it introduces a coupling between two orthogonal, unrelated design aspects (i.e., JSON export and serialization). A change to one aspect might result in changes to the other aspect.

In the worst case, the `exportToJSON()` function might introduce a second dependency. The arguments expected in the `exportToJSON()` call might accidentally reflect some of the implementation details of the chosen JSON library. In that case, eventually switching to another library might result in a change of the signature of the `exportToJSON()` function, which would subsequently cause changes in all callers. Thus, the dependency on the chosen JSON library might accidentally be far more widespread than intended.

The third kind of dependency is introduced by the `serialize()` function. Due to this function, the classes deriving from `Document` depend on global decisions on how documents are serialized. What format do we use? Do we use little endian or big endian? Do we have to add the information that the bytes represent a PDF file or a Word file? If yes (and I assume that is very likely), how do we represent such a document? By means of an integral value? For instance, we could use an enumeration for this purpose:[27]

```
enum class DocumentType
{
   pdf,
   word,
   // ... Potentially many more document types
};
```

This approach is very common for serialization. However, if this low-level document representation is used within the implementations of the `Document` classes, we would accidentally couple all the different kinds of documents. Every deriving class would implicitly know about all the other `Document` types. As a result, adding a new kind of document would directly affect all existing document types. That would be a serious design flaw, since, again, it will make change harder.

Unfortunately, the `Document` class promotes many different kinds of coupling. So no, the `Document` class is not a great example of good class design, since it isn't easy to change. On the contrary, it is hard to change and thus a great example of a violation of the SRP: the classes deriving from `Document` and users of the `Document` class change for many reasons because we have created a strong coupling between several

27 An enumeration seems to be an obvious choice, but of course there are other options as well. In the end, we need an agreed-upon set of values that represent the different document formats in the byte representation.

orthogonal, unrelated aspects. To summarize, deriving classes and users of documents may change for any of the following reasons:

- The implementation details of the `exportToJSON()` function change because of a direct dependency on the used JSON library
- The signature of the `exportToJSON()` function changes because the underlying implementation changes
- The `Document` class and the `serialize()` function change because of a direct dependency on the `ByteStream` class
- The implementation details of the `serialize()` function change because of a direct dependency on the implementation details
- All types of documents change because of the direct dependency on the `DocumentType` enumeration

Obviously, this design promotes more changes, and every single change would be harder. And of course, in the general case, there is the danger that additional orthogonal aspects are artificially coupled inside documents, which would further increase the complexity of making a change. In addition, some of these changes are definitely not restricted to a single place in the codebase. In particular, changes to the implementation details of `exportToJSON()` and `serialize()` would not be restricted to only one class, but likely all kinds of documents (PDF, Word, and so on). Therefore, a change would affect a significant number of places all over the codebase, which poses a maintenance risk.

Logical Versus Physical Coupling

The coupling isn't limited to logical coupling but also extends to physical coupling. Figure 1-2 illustrates that coupling. Let's assume that there is a `User` class on the low level of our architecture that needs to use documents that reside on a higher level of the architecture. Of course the `User` class depends directly on the `Document` class, which is a necessary dependency—an intrinsic dependency of the given problem. Thus, it should not be a concern for us. However, the (potential) physical dependency of `Document` on the selected JSON library and the direct dependency on the `ByteStream` class cause an indirect, transitive dependency of `User` to the JSON library and `ByteStream`, which reside on the highest level of our architecture. In the worst case, this means that changes to the JSON library or the `ByteStream` class have an effect on `User`. Hopefully it's easy to see that this is an artificial, not an intentional, dependency: a `User` shouldn't have to depend on JSON or serialization.

I should explicitly state that there is a *potential* physical dependency of `Document` on the select JSON library. If the `<Document.h>` header file includes any header from the JSON library of choice (as indicated in the code snippet at the beginning of "An Example of Artificial Coupling" on page 12), for instance because the `exportToJSON()` function expects some arguments based on that library, then there is a clear dependency on that library. However, if the interface can properly abstract from these details and the `<Document.h>` header doesn't include anything from the JSON library, the physical dependency might be avoided. Thus, it depends on how well the dependencies can be (and are) abstracted.

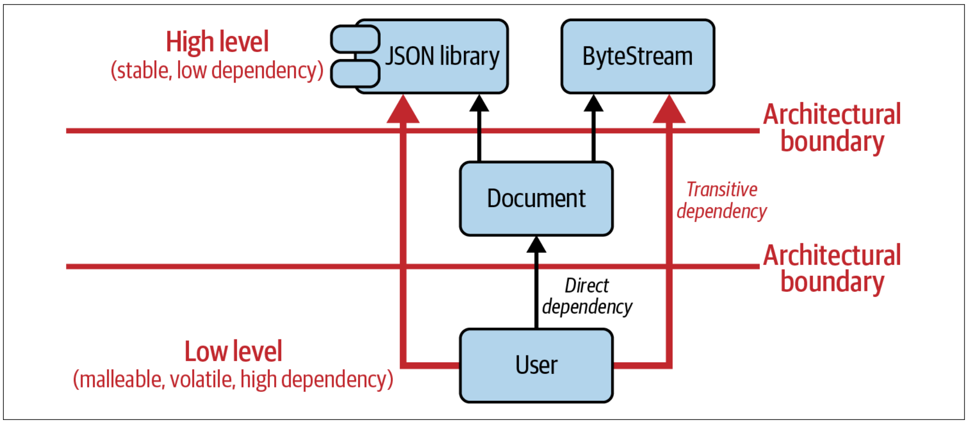

Figure 1-2. The strong transitive, physical coupling between `User` and orthogonal aspects like JSON and serialization.

"High level, low level—now I'm confused," you complain. Yes, I know that these two terms usually cause some confusion. So before we move on, let's agree on the terminology for high level and low level. The origin of these two terms relates to the way we draw diagrams in the *Unified Modeling Language (UML)*: functionality that we consider to be stable appears on the top, on a high level. Functionality that changes more often and is therefore considered to be volatile or malleable appears on the bottom, the low level. Unfortunately, when we draw architectures, we often try to show how things build on one another, so the most stable parts appear at the bottom of an architecture. That, of course, causes some confusion. Independent of how things are drawn, just remember these terms: *high level* refers to stable parts of your architecture, and *low level* refers to the aspects that change more often or are more likely to change.

Back to the problem: the SRP advises that we should separate concerns and the things that do not truly belong, i.e., the noncohesive (adhesive) things. In other words, it advises us to separate the things that change for different reasons into *variation* *points*. Figure 1-3 shows the coupling situation if we isolate the JSON and serialization aspects into separate concerns.

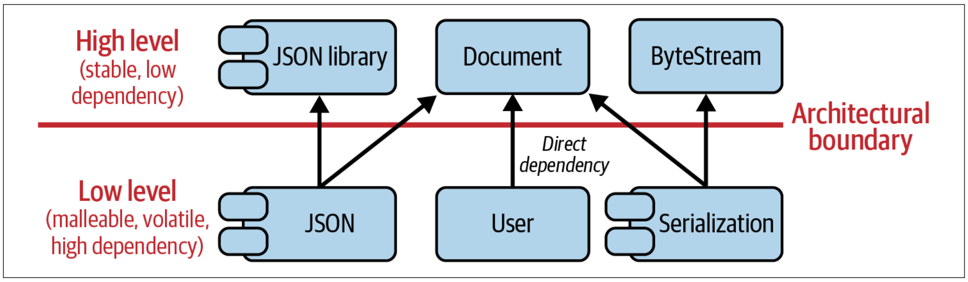

Figure 1-3. Adherence to the SRP resolves the artificial coupling between `User` and JSON and serialization.

Based on this advice, the `Document` class is refactored in the following way:

```
class Document
{
 public:
   // ...
   virtual ~Document() = default;

   // No more 'exportToJSON()' and 'serialize()' functions.
   // Only the very basic document operations, that do not
   // cause strong coupling, remain.
   // ...
};
```

The JSON and serialization aspects are just not part of the fundamental pieces of functionality of a `Document` class. The `Document` class should merely represent the very basic operations of different kinds of documents. All orthogonal aspects should be separated. This will make changes considerably easier. For instance, by isolating the JSON aspect into a separate variation point and into the new `JSON` component, switching from one JSON library to another will affect only this one component. The change could be done in exactly one place and would happen in isolation from all the other, orthogonal aspects. It would also be easier to support the JSON format by means of several JSON libraries. Additionally, any change to how documents are serialized would affect only one component in the code: the new `Serialization` component. Also, `Serialization` would act as a variation point that enables isolated, easy change. That would be the optimal situation.

After your initial disappointment with the `Document` example, I can see you're looking happier again. Perhaps there's even an "I knew it!" smile on your face. However, you're not entirely satisfied yet: "Yes, I agree with the general idea of separating concerns. But how do I have to structure my software to separate concerns? What do I have to do to make it work?" That is an excellent question, but one with many answers that I'll address in the upcoming chapters. The first and most important point, however, is the identification of a variation point, i.e., some aspect in your code where changes are expected. These variation points should be extracted, isolated, and wrapped, such that there are no longer any dependencies on these variations. That will ultimately help make changes easier.

"But that is still only superficial advice!" I hear you say. And you're correct. Unfortunately, there is no single answer and there is no simple answer. It depends. But I promise to give many concrete answers for how to separate concerns in the upcoming chapters. After all, this is a book on software design, i.e., a book on managing dependencies. As a little teaser, in Chapter 3 I will introduce a general and practical approach to this problem: design patterns. With this general idea in mind, I will show you how to separate concerns using different design patterns. For instance, the *Visitor*, *Strategy*, and *External Polymorphism* design patterns come to mind. All of these patterns have different strengths and weaknesses, but they share the property of introducing some kind of abstraction to help you to reduce dependencies. Additionally, I promise to take a close look at how to implement these design patterns in modern C++.

I will introduce the Visitor design pattern in "Guideline 16: Use Visitor to Extend Operations" on page 112, and the Strategy design pattern in "Guideline 19: Use Strategy to Isolate How Things Are Done" on page 140. The External Polymorphism design pattern will be the topic of "Guideline 31: Use External Polymorphism for Nonintrusive Runtime Polymorphism" on page 279.

Don't Repeat Yourself

There is a second, important aspect to changeability. To explain this aspect, I will introduce another example: a hierarchy of items. Figure 1-4 gives an impression of this hierarchy.

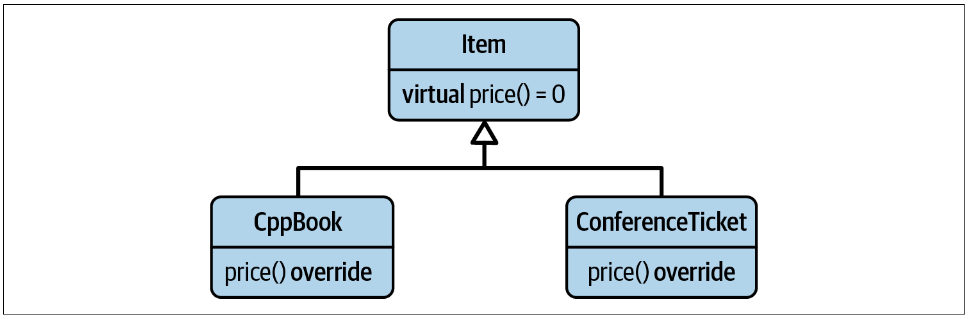

Figure 1-4. The `Item` class hierarchy.

At the top of that hierarchy is the `Item` base class:

```
//---- <Money.h> ----------------

class Money { /*...*/ };

Money operator*( Money money, double factor );
Money operator+( Money lhs, Money rhs );


//---- <Item.h> ----------------

#include <Money.h>

class Item
{
 public:
   virtual ~Item() = default;
   virtual Money price() const = 0;
};
```

The `Item` base class represents an abstraction for any kind of item that has a price tag (represented by the `Money` class). Via the `price()` function, you can query for that price. Of course there are many possible items, but for illustration purposes, we restrict ourselves to `CppBook` and `ConferenceTicket`:

```
//---- <CppBook.h> ----------------

#include <Item.h>
#include <Money.h>
#include <string>

class CppBook : public Item
{
 public:
   explicit CppBook( std::string title, std::string author, Money price )  ❸
      : title_( std::move(title) )
      , author_( std::move(author) )
      , priceWithTax_( price * 1.15 )  // 15% tax rate
   {}

   std::string const& title() const { return title_; }      ❹
   std::string const& author() const { return author_; }    ❺

   Money price() const override { return priceWithTax_; }   ❻

 private:
   std::string title_;
   std::string author_;
   Money priceWithTax_;
};
```

The constructor of the `CppBook` class expects a title and author in the form of strings and a price in the form of `Money` (❸).[28] Apart from that, it only allows you to access the title, the author, and the price with the `title()`, `author()`, and `price()` functions (❹, ❺, and ❻). However, the `price()` function is a little special: obviously, books are subject to taxes. Therefore, the original price of the book needs to be adapted according to a given tax rate. In this example, I assume an imaginary tax rate of 15%.

The `ConferenceTicket` class is the second example of an `Item`:

```
//---- <ConferenceTicket.h> ----------------

#include <Item.h>
#include <Money.h>
#include <string>

class ConferenceTicket : public Item
{
 public:
   explicit ConferenceTicket( std::string name, Money price )  ❼
      : name_( std::move(name) )
      , priceWithTax_( price * 1.15 )  // 15% tax rate
   {}

   std::string const& name() const { return name_; }

   Money price() const override { return priceWithTax_; }

 private:
   std::string name_;
```

28 You might be wondering about the explicit use of the `explicit` keyword for this constructor. Then you might also be aware that Core Guideline C.46 advises using `explicit` by default for single-argument constructors. This is really good and highly recommended advice, as it prevents unintentional, potentially undesirable conversions. While not as valuable, the same advice is also reasonable for all the other constructors, except for the copy and move constructors, which don't perform a conversion. At least it doesn't hurt.

```
      Money priceWithTax_;
   };
```

ConferenceTicket is very similar to the CppBook class, but expects only the name of the conference and the price in the constructor (❼). Of course, you can access the name and the price with the name() and price() functions, respectively. Most importantly, however, the price for a C++ conference is also subject to taxes. Therefore, we again adapt the original price according to the imaginary tax rate of 15%.

With this functionality available, we can go ahead and create a couple of Items in the main() function:

```
#include <CppBook.h>
#include <ConferenceTicket.h>
#include <algorithm>
#include <cstdlib>
#include <memory>
#include <vector>

int main()
{
   std::vector<std::unique_ptr<Item>> items{};

   items.emplace_back(
      std::make_unique<CppBook>("Effective C++", "Meyers", 19.99) );
   items.emplace_back(
      std::make_unique<CppBook>("C++ Templates", "Josuttis", 49.99) );

   items.emplace_back( std::make_unique<ConferenceTicket>("CppCon", 999.0) );
   items.emplace_back( std::make_unique<ConferenceTicket>("Meeting C++", 699.0) );
   items.emplace_back( std::make_unique<ConferenceTicket>("C++ on Sea", 499.0) );

   Money const total_price =
      std::accumulate( begin(items), end(items), Money{},
         []( Money accu, auto const& item ){
            return accu + item->price();
         } );

   // ...

   return EXIT_SUCCESS;
}
```

In `main()`, we create a couple of items (two books and three conferences) and compute the total price of all items.[29] The total price will, of course, include the imaginary tax rate of 15%.

That sounds like a good design. We have separated the specific kinds of items and are able to change how the price of each item is computed in isolation. It seems that we have fulfilled the SRP and extracted and isolated the variation points. And of course, there are more items. Many more. And all of them will make sure that the applicable tax rate is properly taken into account. Great! Now, while this `Item` hierarchy will make us happy for some time, the design unfortunately has a significant flaw. We might not realize it today, but there's always a looming shadow in the distance, the nemesis of problems in software: change.

What happens if for some reason the tax rate changes? What if the 15% tax rate is lowered to 12%? Or raised to 16%? I can still hear the arguments from the day the initial design was committed into the codebase: "No, that will never happen!" Well, even the most unexpected thing may happen. For instance, in Germany, the tax rate was lowered from 19% to 16% for half a year in 2021. This, of course, would mean that we have to change the tax rate in our codebase. Where do we apply the change? In the current situation, the change would pretty much affect every class deriving from the `Item` class. The change would be all over the codebase!

Just as much as the SRP advises separating variation points, we should take care not to duplicate information throughout the codebase. As much as everything should have a single responsibility (a single reason to change), every responsibility should exist only once in the system. This idea is commonly called the *Don't Repeat Yourself* (DRY) principle. This principle advises us to not duplicate some key information in many places—but to design the system such that we can make the change in only one place. In the optimal case, the tax rate(s) should be represented in exactly one place to enable you to make an easy change.

Usually the SRP and the DRY principles work together very nicely. Adhering the SRP will often lead to adhering to DRY as well, and vice versa. However, sometimes adhering to both requires some extra steps. I know you're eager to learn what these extra steps are and how to solve the problem, but at this point, it's sufficient to point out the general idea of SRP and DRY. I promise to revisit this problem and to show you how to solve it (see "Guideline 35: Use Decorators to Add Customization Hierarchically" on page 348).

29 You might realize I've picked the names of the three conferences I regularly attend: CppCon, Meeting C++, and C++ on Sea. There are many more C++ conferences, though. To give a few examples: ACCU, Core C++, pacific++, CppNorth, emBO++, and CPPP. Conferences are a great and fun way to stay up to date with C++. Make sure to check out the Standard C++ Foundation home page for any upcoming conferences.

Avoid Premature Separation of Concerns

At this point, I've hopefully convinced you that adhering to SRP and DRY is a very reasonable idea. You might even be so committed that you plan to separate everything—all classes and functions—into the most tiny units of functionality. After all, that's the goal, right? If this is what you're thinking right now, please stop! Take a deep breath. And one more. And then please listen carefully to the wisdom of Katerina Trajchevska:[30]

> Don't try to achieve SOLID, use SOLID to achieve maintainability.

Both SRP and DRY are your tools for achieving better maintainability and simplifying change. They are not your goals. While both are of utmost importance in the long run, it can be very counterproductive to separate entities without a clear idea about what kind of change will affect you. Designing for change usually favors one specific kind of change but might unfortunately make other kinds of change harder. This philosophy is part of the commonly known *YAGNI* principle (You Aren't Gonna Need It), which warns you about overengineering (see also "Guideline 5: Design for Extension" on page 35). If you have a clear plan, if you know what kind of change to expect, then apply SRP and DRY to make that kind of change simple. However, if you don't know what kind of change to expect, then don't guess—just wait. Wait until you have a clear idea about what kind of change to expect and then refactor to make the change as easy as possible.

> Just don't forget that one aspect of easily changing things is having unit tests in place that give you confirmation that the change did not break the expected behavior.

In summary, change is expected in *soft*ware and therefore it's vital to design for change. Separate concerns and minimize duplication to enable you to easily change things without being afraid to break other, orthogonal aspects.

30 Katerina Trajchevska, "Becoming a Better Developer by Using the SOLID Design Principles", Laracon EU, August 30–31, 2018.

Guideline 2: Design for Change

- Expect change in *soft*ware.
- Design for easy change and make software more adaptable.
- Avoid combining unrelated, orthogonal aspects to prevent coupling.
- Understand that coupling increases the likelihood for change and makes changes harder.
- Adhere to the Single-Responsibility Principle (SRP) to separate concerns.
- Follow the Don't Repeat Yourself (DRY) principle to minimize duplication.
- Avoid premature abstraction if you are not sure about the next change.

Guideline 3: Separate Interfaces to Avoid Artificial Coupling

Let's revisit the `Document` example from "Guideline 2: Design for Change" on page 11. I know, by now you probably feel like you've seen enough documents, but believe me, we're not done yet. There's still an important coupling aspect to address. This time we don't focus on the individual functions in the `Document` class but on the interface as a whole:

```
class Document
{
 public:
   // ...
   virtual ~Document() = default;

   virtual void exportToJSON( /*...*/ ) const = 0;
   virtual void serialize( ByteStream& bs, /*...*/ ) const = 0;
   // ...
};
```

Segregate Interfaces to Separate Concerns

The `Document` requires deriving classes to handle both JSON exports and serialization. While, from the point of view of a document, this may seem reasonable (after all, *all* documents should be exportable into JSON and serializable), it unfortunately causes another kind of coupling. Imagine the following user code:

```
void exportDocument( Document const& doc )
{
   // ...
   doc.exportToJSON( /* pass necessary arguments */ );
```