Software Engineering Made Easy

1. Introduction to Software Engineering

This book is currently undergoing revisions. Feedback is highly valued. Please send me the commented PDF to marco.gaehler@gmx.ch. It is advisable to submit many small changes rather than a single large one. Feedback may include precise recommendations for enhancement or general reflections. I welcome broader feedback. Tipos are getting fixed using some AI tool, so don't worry about them.

Getting started

The current configuration of the project is quite neat when working with Visual Studio Code with the "Markdown all in one" and the "Markdown PDF" extensions. Open the readme.md file and select the outline located on the left-hand side. This provides an overview of all the chapters.

This is a book about software engineering, similar to "Clean Code" by Robert C. Martin and "The Pragmatic Programmer" by Thomas & Hunt. The current document is only a rough draft, though it's making progress. The initial chapters feel already quite good, the latter portion of the book requires significant revision.

Things to write

- Some more chapters towards the end of the book need to be improved.

2. Table of content

3. One sentence summary

// mention which part the chapters are in?? // fix the whole enumeration

Parts: Part 1: First things first 6-12 Part 2: Components of Code 13-17 Part 3: Testing 18-20 Part 4: Design Principles 21-22 Part 5: Programming 23-29 Part 6: High level design 30-32 Part 7: Existing code 33-36 Part 8: Miscelaneous 37-41 Part 9: Collaborating 42-50 Part 10: Final remarks 51-54

1. .
2. .
3. .
4. .
5. Preface: My personal story behind this book.
6. Software Engineering: We destill some basic rules what Softwar Engineering is about.
7. Good Code, an overview: A short list that summarizes the most important points of this book.
8. Understandable Code: An attempt to explain what we understand and what we don't understand.
9. The Single Responsibility Principle (SRP): We discuss why it is of utmost importance that every piece of code does exactly one thing.
10. Levels of Abstraction: Many very complex objects may be combined to form a new object that is fairly easy to comprehend.
11. Interfaces: What is an interface? And how do I design a good interface?
12. Naming: Naming is the most difficult part of this book because there are so many rough rules but no clear cut answers.
13. Functions: As discussed in the chapter on the SRP, functions should do exactly one thing. Therefore, they should be short.
14. Classes: We learn how to structure classes beyond the old getter and setter non-sense.
15. The utilization of inheritance is generally discouraged due to its tendency to result in poor code quality.
16. Data Types: What types of primitive data are there? And why should you be cautios with using booleans and strings?
17. Properties of Variables: There are additional properties of variables primarily determine the scope within which they can be utilized and whether they are mutable or immutable.
18. Testing: Testing is of utmost importance to keep your code correct over the test of time.
19. Types of Tests: We discuss unit tests, and functional tests.
20. Writing better code with tests: Having tests forces you to write better code as tests require good interfaces.
21. SOLID principles: Explaining the SOLID principles that Robert C. Martin (Uncle Bob) came up with.
22. Software Engineering principles: Some general software engineering principles from [https://youtu.be/XQzEo1qag4A]
23. Programming Paradigms: We briefly discuss the differences between object oriented (OO), procedural, and functional programming.
24. Programming Languages: A brief overview on the most commonly used programming languages.
25. Physical Laws of Code: Surprisingly, code obeys some physical laws as well.
26. Bugs, Errors, Exceptions: A discussion how you should deal with and prevent bugs.
27. Complexity: There is always a certain amount of complexity in a certain problem that will be reflected in the code.
28. Dependencies: How do you deal with dependencies between files and code snippets?
29. Decoupling: ? // remove chapter
30. Software Architecture: A brief overview what software architecture is about.

31. Design patterns: We look the factory pattern to see what design patterns are.
32. Domain Driven Design: An introduction to Domain Driven Design (DDD).
33. 3rd party software: How to deal with 3rd party software.
34. Refactoring Fundamentals: Refactoring is about keeping your code in shape.
35. Refactoring techniques: ...?
36. Refactoring Legacy Code: There are a whole lot of problems with projects that don't follow the rules explained in this book.
37. Performance Optimization: Don't optimize until the very end of a project.
38. Comments: Replace comments with code whenever possible.
39. Logging: For single threaded code where the user uses only an API, logging is not needed. On the other hand, logging might be needed for distributed systems where race conditions may occure.
40. Data files: If you store data, there may be better file formats than csv files.
41. Setting up a project: If you start with a new project, you should first get the whole infrastructure right.
42. Tools: An overview of the most important tools that you should use, starting with Git.
43. Working in teams: Working in teams has some advantages and drawbacks.
44. Code review: Let your code be reviewed by others.
45. Agile: Agile is a way to organize your work in a team and offers an alternative to waterfall.
46. Requirements Engineering: What should you actually program?
47. Planning: Before you start coding, you should roughly plan what you are going to do.
48. DevOps: DevOps is considered the way to go regarding the organization of your projects.
49. Mental Health: Working in IT can be very stressful. Here are some tips how to deal with it.
50. Hiring and getting hired: How to get a job as a Software Engineer.
51. Examples: // ? rewrite chapter?
52. About Copilot: A short overview of Copilot. // move to the introduction?
53. Outlook: Some personal advice and wishes for your future.
54. Abbreviations: Abbreviations used in this book.
55. Index: The index of this book.

Chapters to work on:

- Decoupling (Remove?)
- Software Architecture (Ask Volker?)
- Domain Driven Design (Ask who?)
- Examples (Rewrite?)
- About Copilot (who knows anything about Copilot?)
- Requirements Engineering (Felix)

4. The short story behind this book

"Software development is a journey. Every bug fixed is a lesson learned." - ?

I studied physics at ETH Zurich, Switzerland. And I worked as a teacher for a few years before I decided to switch to software engineering. I worked for a few years as a software engineer at Zurich Instruments, a company that develops electronic devices used in quantum computing. There, I was mostly busy developing software for quantum computers.

At the beginning of my time there, I was still a novice in software engineering, but I quickly picked up a lot of new skills. At the same time, I was in touch with many PhD students and realized how poorly written their code was. This is when I came up with the idea to write a book about software engineering. I wanted to write a book to myself at the end of my studies. When I knew the basics of programming, but nobody told me how to structure my code properly. I wanted to write a book that explains everything I learned about good programming practices during the few years I spent in industry. Such that every person with a little bit of programming knowledge can improve their skills to a reasonable level by reading this book. That being said, reading this book will, of course, not be enough to become a proficient software engineer. It also takes a lot of practice and continuous learning.

I wasn't really sure where this book would take me. In the beginning, I didn't even think this would become a real book. I was just writing down my thoughts and ideas. Things that I once thought of, but couldn't find any literature about or only scattered over many different places. My English is fairly poor, and I was never really good at writing essays in school. But as I was reading other books, I got some more inspiration on what to write. And as this text got longer and I got some encouraging feedback from my reviewers. Therefore, I decided to publish this book.

The two books which inspired me the most are the best sellers [Clean Code] by Robert C. Martin and [The Pragmatic Programmer] by Andrew Hunt and David Thomas. I like how they both give some general advice how to write better code. But, of course, I didn't want to make a copy of these books. Even though there are some similarities, especially with Clean Code. I tried hard to find out why I liked these books and what could be improved on. For instance the code examples given in these books. On one hand, The Pragmatic Programmer has no code at all, meanwhile the Clean Code frequently has code which is too long to understand.

Therefore, I tried to write a book that has useful topics and at the same time some concrete code examples. I really tried to keep the code examples as short as possible in order to make them easy to understand. Always imagine this little piece of code being inside a huge codebase.

If you like long functions and classes, I have to warn you that I don't them too much. I prefer to structure code with many short functions and very few, small classes. Classes may be useful tools, but they are frequently abused for writing bad code using the member variables as "mini-globals" [The Art of Readable Code]. Instead, classes should in my opinion usually do little more than organizing data, as done in C-style structs.

I hope you'll enjoy reading this book, Marco Gähler

Thanks to

There are some people who read through this book and were very helpful in giving me feedback. I want to thank them here:

- Volker Obermeit
- Rafael Gort
- Marios Karakasis
- Felix Gähler
- Hans Märki
- Bernhard Brodowsy
- Fabian Mäser
- ... you?

Furthermore I'd like to thank Uwe Schmitt and the Research Software Engeneering (RSE) community in Zurich.

I would also thank to Martin Fowler, Robert C. Martin, and Dave Thomas, amoung others, for their moral support and their great books. Though, little surprisingly, they didn't have time to read through this book.

Copilot and Wordvice helped me a lot writing this book. Copilot at times gave me some inspiration on how to finish a sentence and Wordvice helped me out with improving the language. When revising this text, I just realized once again how poor my English was.

I would also like to thank the team of Pearson Germany for their support. Most notably Birger Peil.

5. Preface

"I have been consistently disappointed by the quality of CS [computer science] graduates. It's not that the graduates aren't bright or talented, it's just that they haven't been taught what programming is really all about." - Robert C. Martin

In 2007, I had my first semester at university. It was the first time I learned programming. We learned C++ and I found it very confusing. Especially things like plain old arrays, pointers, const expressions, etc. I struggled to understand these concepts. They just felt wrong. There were numerous unresolved questions about writing the code correctly, and I didn't know where to get good advice. I passed the exam, but I was somewhat dissatisfied.

Three years later, I took a course on computational physics. There, I had to write slightly bigger programs. It worked, but I struggled a lot. The code was dreadful, and I knew it. But I didn't know how to make it any better. Changing things was hard, and I learned how to use a debugger. I still have all my university files, but I haven't dared to look at this code ever since. Already thinking about it makes me shudder.

After my studies, I wanted to improve my programming skills, so I read the book "Effective Modern C++" by Scott Meyers. A great book. But it wasn't made for me at that time. It deals with many details of C++ and I barely understood anything because I lacked the necessary background knowledge. The book was too advanced for me.

A few years later, I decided to give programming another shot. I found a company that was looking for people with programming and physics expertise. So, I thought I might have a chance, despite my poor programming skills. At the job interview, I was asked a few very technical (and in hindsight not particularly useful) C++ questions, and I could answer most of them thanks to the book I read. I got the job.

In the beginning, I struggled a little. I was overwhelmed by the amount of code. I didn't know which IDE to use, and the build process I used was flawed. You name it. Still, I received some good feedback from my boss. A few months later, I had my first significant feature implemented. It also had automated tests, and the code was much cleaner than similar features. Another month later, I implemented my second feature. Everyone in the company expected this task to be very challenging, but I found a neat way to implement it.

My boss wrote most of the code, and the success of the company was largely dependent on his efforts. He knew everything, but I hardly ever understood a thing when he explained his code. In many topics, I lacked the necessary background information.

Around that time, the company hired additional software developers. Especially one of them made a huge impression on me. I could ask him almost anything, and he was able to provide me with a simple answer. He understood the concepts on which our code was based, enabling him to grasp the fundamental structure of our code and organize the remaining elements. But it was also the way he worked. He wrote small functions covered with automated tests. He was also refactoring the code. One Monday morning, he arrived at the office. He opened a massive merge request to refactor some code across the entire codebase. He broke down this huge and widespread problem into small chunks and wrote tests for the new implementation. He opened my eyes. He made me realize that the way he worked was so much better. So much more structured. This was proper engineering. Real software engineering.

There is so much I learned in those few years. And the basic principles are so easy to learn. You just need someone to teach you. This book is what this book is about. No fancy code, just fundamental principles. An overview of the most important topics so that you do not get overwhelmed by the infinite number of decisions a software engineer has to make. This book contains numerous real-world examples that do not require any code. I want to explain principles that are very general and do not require any code to explain. In fact, software engineering is, in some respects, very similar to other fields of engineering. Therefore, a car is often a better example to explain my point than some fancy piece of code that you struggle comprehending.

Of course, I also included some code examples. I didn't want this book to be too abstract. Though in order to keep the examples easy to understand, they had to be small, and I frequently had to simplify them a lot.

I'm not a great software engineer, not at all. And my English is fairly lousy. But maybe this is a good thing when writing a book. It will be easy to understand as I tried to keep all chapters concise and easy to understand. It keeps the book short and motivates you to read it all because everything I wrote is important. At least, that's what I hope.

I'm not God, and this is not the Holy Bible. This book aims to assist you with your programming problems, but it does not contain the absolute truth. Probably, there are hardly any absolute truths in programming; it's not math. Software engineering is an empirical field. We have to find out what works best by trying different things and explain it a few sentences. There are many trade-offs to be made as some of these rules may be contradictory. I hope that the recommendations and trade-offs I provide in this book will help you write better code. And if you don't agree with some of my recommendations, that's fine. You will certainly be able to find examples where my general recommendations will not apply. Feel free to create a YouTube video to explain your point if you can come up with a better rule how to do something. I would be delighted if you taught me how to become a better software developer.

Reading this book is only one step in your career. Next, you have to get out into the real world. Get a job. Write code and learn how to apply the principles you have learned here. It is hard; this will take your whole life. Many others face similar problems. Talk to them, improve your solutions, and get smarter. Become a real software engineer.

Enjoy this book and good luck with your career.

What this book is about

In the first chapter, we want to examine how code should look like. What kind of rules there are to judge the quality of code and some of my personal recommendations what kind of features of your programming language you should, or rather shouldn't, use. In my opinion, there are numerous practices in object-oriented (OO) programming that are predominantly utilized for historical reasons. In reality, they usually lead to poor code and should be abandoned. In fact, pretty much everything other than plain classes and interfaces should be used with care in OO programming.

But OO programming is by far not the most important topic in this book. No matter how good or bad your use of OO features is, you can still write good or bad code regardless. There are more important concepts to learn from this book. Most notably, the Single Responsibility Principle (SRP), basics of interfaces, testing, and naming. Furthermore, there are several chapters on how to work with code that has not been written up to current standards and how to collaborate with other programmers. Topics that are highly important but are frequently neglected in books on software development.

This book contains relatively few code examples. It's more about general concepts of software engineering, rather than concrete code examples. Still, some concepts are easier to understand with a few lines of code. Therefore, I tried to create some code examples. Even though it's quite challenging to find concise examples that are still expressive enough to fit into a book. As for the programming languages I chose, mostly Python and some C++. Not because these languages would be better than, for example, JavaScript, but rather because these are the languages I know. I chose two programming languages because there are some concepts that I can only explain using one or the other. Though there are only a few things that depend on the programming language. Most of the explanations provided here consist of general recommendations that are applicable to almost any programming language.

"Software Engineering is the application of an epmirical, scientific approach to finding efficient, economic solutions to practical problems in software" - David Farely [](Modern Software Engineering, p. xxii)

This book aims to provide clear answers to simple problems in software engineering. I also attempt to provide answers to challenging problems like naming, but these are typically quite vague, as in other books. This is what makes the problems so challenging and software engineering exciting. The only thing that truly helps with challenging problems is a lot of experience. It would take too much explanation or code to explain all the details. I can only attempt to present all the various arguments for certain trade-offs, and then you will need to do all the reasoning by yourself. This is why software engineering is challenging. This is why it is fun. There are too many problems without any clear solutions. And you have to deal with them all by yourself.

This book is about engineering. It's about finding ways how to write better code. It's not a strictly scientific approach, it's more of an empiric approach. Thus, there is no absolute truth and there are no proofs in this book. I will rather give you some general advice on best practices. Due to this reason, there are only a few references available for specific topics but no scientific studies. Most chapters consist of my personal interpretations of more specialized books. Thus, I mention the books I was reading as a foundation for the corresponding chapter. And of course, all of these books are biased by my personal opinions and reasonings.

Who this book is for

This book was initially intended for PhD students. I know quite a few who spend a lot of time programming but have never really learned how to do it properly. After learning the basic syntax of a programming language, they began writing code. But it was dreadful. They never learned how to write good code. They never learned about the necessity of small class sizes or the significance of tests. And there are many more points that I am going to explain throughout this book.

Of course, this book is not only for PhD students. There are also many programmers who never learned proper software engineering. Though as you are reading this book here, chances are that you have already read some other books and that you are familiar with many of the things I am writing about. But I believe there are still some novel ideas in this book that you can utilize to enhance your code.

At the same time, I'd like to mention what this book isn't about. It doesn't teach you fancy modern topics in computer science. It doesn't teach you how to develop artificial intelligence, high-performance computing, web development, databases, distributed systems, etc. I simply lack the scope and knowledge to cover all these topics. Though I'm confident that the principles taught throughout this book will help you with these topics, as they all require good software engineering skills.

Writing this book

Writing a book about software development is hard. Harder than writing a book about physics or math. Physics and mathematics are precise sciences, and you can derive all your formulas. Software engineering, on the other hand, is not an exact science. I can only give some examples to support my claims, but no proofs. And you can certainly find counter examples if you try. There are some rules of thumb at most, but there may be two rules contradicting each other. The best example is the naming of variables, functions, and classes. There is the rule that "a name should be short, yet concise". Good luck finding such a name.

I can only provide you with some general rules of thumb, and you will have to determine how to apply them yourself. This will take practice, and it is preferable to work together with a more experienced coworker who can assist you in case you have any questions. This book aims to serve as a manual, but ultimately, you will need to learn how to apply the recommendations given here on your own.

-# Part 1: First things first

6. Software Engineering

"If I had an hour to solve a problem, I'd spend 55 minutes thinking about the problem and 5 minutes thinking about solutions." – Albert Einstein

The Life of a Software Engineer

I understand that you want me to begin and provide you with some sophisticated code examples. And I'm sorry to inform you that this is not happening. We don't even know yet what this book should be about. Of course, you want to become a great software engineer, get a job at Google, earn a lot of money, and live a happy life. But this is not how it works. We first have to sit down and analyze the situation.

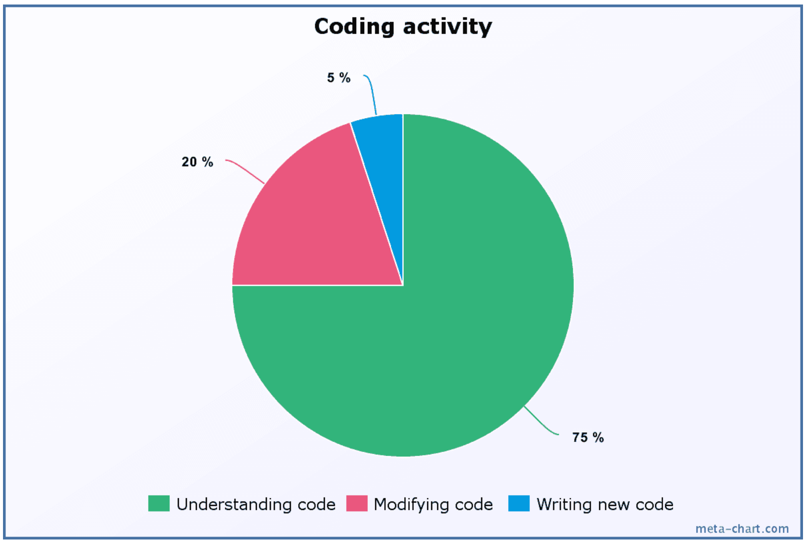
Let me start with a very blunt question: "What do you think a software engineer does?"

"He writes code" may be your first response.

"He engineers software" is a very smart one.

Indeed, these answers contain some truth. But writing code only represents a small portion of your future workday. One thing you will do is the same as what we are doing right now: analyzing a problem and trying to figure out what to do next.

You will, of course, spend a fair amount of time with your precious code. But I have to disappoint you once again. It will be like in a marriage. You spend most of your time cleaning up or discussing things. The part that is truly enjoyable only covers a small fraction of it. The following plot, with highly unscientific numbers that I found somewhere on the internet, sums it up nicely.



You definitely need to take a second look to fully understand the meaning of this plot. You will spend only 5% of the time implementing new features! 5%! Not to mention all the meetings you have to attend as well. Of course, these numbers are only a very rough estimate. They depend on many factors. If you are working on a new project where no refactoring (code clean up) is required yet, you will have less code to read. Ultimately, you will spend more time coding. In a very large project, it takes more time to implement changes. It can take a year to become fully productive in a large project! But the company has been generating revenue from this code for a long time, so prioritizing the addition of new features is no longer as crucial. Either way, I will continue the discussion with the value from the plot.

The most obvious and undeniable conclusion we can draw from the plot is that software engineering is not about writing code. It's about reading code! If you can reduce the time required to read code by half, you save more time than you spend writing code in total. By a lot.

I came up with five fundamental rules for software engineering. The first one is:

We write code that is easy to understand.

Good code is not fancy; it is not complex, and it is not necessarily short. Good code is simple. It is as simple as it gets. Reading good code is not like reading Shakespeare. It's ... it's rather like watching some politicians... Who's vocabulary consists of only 1000 different words. This is beneficial when speaking on television. Everyone understands you. Even when people are tired or uneducated, they enjoy listening to you. They don't have to focus in order to understand you. I sometimes feel embarrassed because of my poor English skills. But writing these lines is really cheering me up. Most people reading this book are not native English speakers, and therefore, my somewhat limited language skills may actually be helpful in that regard. It makes this book easier to understand. And with code, it's fairly similar. Simple code is good because it is easy to comprehend.

Good code utilizes only the essential syntax provided by a programming language. It is great if you don't know a programming language too well. You avoid falling into the trap of using fancy but useless features. Don't learn programming languages. Learn programming. Unless you work for Google or another company developing highly specialized code, you will never need all the gimmicks that modern programming languages have to offer.

Writing correct code

Now, it is not only important to ensure that the code works, but we also have to verify its correctness. The crash of two Boeing airplanes in 2018/2019 was not the first time that software bugs led to catastrophic damage. Nor will it be the last time, unfortunately.

We don't want to be responsible for people dying or companies going bankrupt. We want to write impeccable code. We want to ensure that there are no bugs to the best of our ability. We constantly check that our code is correct. We test our code. We let our computers test our code. We write code to test our code!

The second rule of software engineering:

We write automated tests that cover all of our code.

Along with this rule comes the fact that we all make mistakes. We're only human after all. So we have to write our code accordingly.

The third rule of software engineering:

We write code that can have as few possible bugs as possible.

Cleaning up code

Now let's return to our lovely plot. There is one more substantial chunk of work. Modifying the current code. Also known as refactoring. Yes, as astonishing as it sounds, you have to clean up your code just the same way as you have to clean up your kitchen. This process is called refactoring, and its importance cannot be understated. It helps you keep the logic of the code under control by sorting things out. All the time, over and over again. Without refactoring, your code quite quickly becomes a huge mess, making it difficult to implement any changes. And there will be a million places where bugs can hide. Though changing code always carries the inherent risk of potentially breaking it. This is one of the reasons why we need good tests. If we have good test coverage, we can change the code with confidence that we won't break it.

The fourth rule of software engineering:

We constantly clean up our code.

Now you have an idea of what the life of a programmer will look like. Now you know what to look out for. Now we can do what you wanted me to do half an hour ago. I can explain the fundamental principles of writing good code.

Writing code for a purpose

You might already be working in a company, or you will be soon. Your boss is not going to let you write code for a month just because you like it. You will be spending a considerable amount of time in meetings and engaging in discussions with others to determine precisely what you should do. What your customers want.

The fifth rule of software engineering:

We write code to create value for our customers.

If you don't like meetings nor customers, you can stay at home and write whatever code you like. But unless you are a genius, the chances of anyone paying you for that are very low. It is more rewarding to write mediocre code that is being used than to write brilliant code that no one cares about.

The five rules of software engineering

These five rules will accompany us throughout our book.

//make a list of the 5 rules in a box

- We write code that is easy to understand.
- We write automated tests that cover all of our code.
- We write code that can have as few possible bugs as possible.
- We constantly clean up our code.
- We write code to create value for our customers.

7. Good code: a list of rules

// remove this chapter? we already have the one sentence summary

"Truth can only be found in one place: the code." - Robert C. Martin

This is an attempt to distill a list of rules that enable you to assess the quality of code. Unfortunately, I didn't find a way to sort these points. They represent the vastness of software engineering.

By definition, good code is easy to understand [preface]. Also, for new software developers on the team. With good code, even marketing people may comprehend some of your technical discussions as you use the same language [domain driven design].

Good code is well-tested [testing]. It includes unit and functional tests [types of tests]. Especially a good coverage with unit tests is essential as it compels you to write high-quality code [writing better code with tests]. At the same time, unit tests significantly reduce the number of errors in your code.

Pretty much all your code follows the SRP [Single Responsability Principle]. Functions, classes, modules. Everything. The build process only requires one command. This makes the code much easier to understand, and naming also becomes simpler.

Names should be short yet concise [Naming].

Do not repeat yourself (DRY) [section Do not Repeat Yourself]. Do not copy-paste code around. But also avoid conceptual code duplication. Code duplication is terrible as you can never be sure if making a single change is sufficient, or if it needs to be applied in multiple other locations. This leads to bugs and high maintenance costs very quickly.

Classes should have high cohesion [classes]. They should have a strong coupling between the variables [Data types] and methods [Functions]. On the other hand, they should have weak coupling to other classes. Due to constantly adding functionality, classes tend to lose cohesion. Then, they have to be broken up into several smaller classes [Refactoring].

It feels easy to add features and change code. Thanks to the test coverage [Testing] you have a safety net, and well-structured code makes it apparent where new features belong [Physical Laws of Code].

A function name explains you what the function does [Naming]. There is no surprising behavior. The same holds true for classes and variables. Functions have no side effects [Functional Programming].

There are no magic numbers. Assigning the magic number to a variable with a suitable name makes the code much clearer to understand [Naming].

Define variables right where they are used. Always assign a value to them immediately.

Create objects all at once. There are cases where objects are created only partially due to missing information. This is should be avoided and akin to a supply chain issue. Gather all the pieces needed to create an object before doing son. An object should be created completely or not at all. Throw exceptions if objects cannot be created at once.

Write short functions (~< 10 lines) and classes (~< 30 lines) [Single responsability principle]. These are very rough estimates and depend on numerous factors. Usually, their length is limited by the SRP and the level of abstraction [Levels of abstraction]. Complicated functions and classes should be kept short to prevent their complexity [Complexity] from getting out of hand.

Keep the dependencies between different parts of the code minimal [Dependencies]. Especially when dealing with third-party libraries [Third-Party Libraries], it is advisable to create a wrapper [Interfaces] around them. This is helpful when you want to replace it.

Using a debugger [Bugs, Errors, Exceptions] frequently is a strong sign that you have lost control of the code. Normally, automated tests [Testing] should cover all bugs, rendering the debugger unused.

YAGNI: You Aren't Going to Need It. Plan ahead the structure of your code, but refrain from implementing anything you don't need yet [Planning]. Chances are, you will never need it. Only architects have to speculate on what will be used in the future [Software Architecture]. Developers implement only features that will definitely be utilized.

The solution representing the natural logic of the problem is usually the best [Domain driven desing]. It has the lowest complexity [Complexity]. The complexity of the code is equal to the complexity of the actual problem to be implemented. The sales team can explain the domain logic, and you need to bake it into the code. Don't come up with your own logic on a problem you don't understand well.

Use the most basic features of your programming language. [Programming language]. Only use more complex features if you truly benefit from them. Avoid utilizing features of your programming language that resemble black magic.

Avoid nesting if loops. This violates the SRP and is highly prone to bugs [Single Responsability Principle]. Avoid nested try-catch blocks as well. It is preferable to avoid nested loops entirely. In fact, you should having many levels of indentation. These make code hard to understand.

Avoid using Boolean values [section Booleans] and logic as much as possible. Due to human deficiencies, these lines of code harbor the most errors. Try to avoid them as far as reasonably possible. Ensure that every branch of an if statement is tested.

Avoid passing Booleans as function arguments [Functions]. They are a strong indication of a violated SRP [Single Responsability Principle]. Resolve the consequences immediately and use dependency injection (DI) instead.

Avoid string comparisons [section Strings]. Use enums instead [seciton Enums]. Convert the string into an enum as soon as you have the string object available.

Write self-explanatory code. Only use comments for aspects that the code cannot explain on its own. [Comments]

The Zen of Python

The Zen of Python [https://peps.Python.org/pep-0020/#the-zen-of-Python] is a list of 19 guiding principles by Tim Peters. I find them very useful, so I decided to include some of them in my list, along with a brief explanation.

Explicit is better than implicit: Explicit code is easier to understand. Do not try to hide complicated logic as it will haunt you eventually.

Flat is better than nested: Nested code is hard to understand and error-prone.

Readability counts: Of course, it does. Readability is the most important metric of good code.

Special cases aren't special enough to break the rules: Stick to the rules explained in this book.

Although practicality beats purity: You are allowed to break the rules explained in this book if you have a valid reason.

Errors should never pass silently: If an error occurs, something went wrong, and the user should be informed about it.

There should be one, and preferably only one, obvious way to do it: Of course, there are always some details that you don't know how to deal with. In general, it is true that there should be only one way to implement a feature.

Now is better than never: later = never.

If the implementation is hard to explain, it's a bad idea: This suggests that the logic behind your solution may be flawed.

Namespaces are one honking great idea -- let's do more of those! Yes, namespaces are great. They help you structure your code and make it more readable because you know where a function or other piece of code originates from.

8. Understandable code

"Any fool can write code that a computer can understand. A good programmer writes code a human can understand." – Martin Fowler

How Humans Think

As we have discussed, good code is easy to understand. But what makes code easy or hard to understand? A computer understands everything. He doesn't care as long as the syntax is correct. And if there is a bug, the computer simply runs it. But we don't care about the computer. This book is written for humans. Code is written for humans. We have to ask ourselves: When does a human understand something? Or what do humans struggle with?

Humans are fundamentally different from computers. We can achieve incredible feats, yet we also have significant weaknesses. The evolution adapted us to our environment. We were made to live in the forest, hunt animals, and socialize with our clan. We needed keen eyes to spot our prey, a vivid imagination to grasp the terrain and wind direction, and familiarity with our hunting companions. These tasks necessitate a great deal of intuition and approximate reasoning. These are challenges that computers or robots struggle with. Though they improve, thanks to the emergence of artificial intelligence.

One thing is clear: Humans struggle to think logically. We are easily overwhelemed if there is no structure that we can understand. The computer can execute the following line of code without any issues, but I guess only very few readers would be able to determine the result within five minutes.

```
(lambda f, n: f(f, n))(lambda f, n: [(not n % 3 and "fizz" or "") +
    (not n % 5 and "buzz" or "") or n] + f(f, n+1) if n <= 100 else [], 1)
```

[https://www.quora.com/What-are-some-prime-examples-of-bad-Python-code]

The result is the famous "Fizz Buzz" game [https://en.wikipedia.org/wiki/Fizz_buzz]. Humans struggle with code like this because we struggle to structure it. It is too big to understand all at once, and most of us can't break it down into smaller pieces.

We are limited by the amount of complexity we can imagine. We are not able to understand this seemingly unordered pile of text. So, there is only one strategy that works: divide and conquer. Break up complex problems into many smaller pieces that you can understand. Maybe you will have to repeat this step recursively until you have small enough pieces that you can deal with. This is our area of expertise, where we excel in solving complex problems. Use your imagination!

This is how we are able to create extremely complex objects. We have to break them down into small parts that we understand very well and then build them together like Lego bricks. Every time we assemble a few pieces, we create something new that we give a name to and can explain to other humans what this thing does. It has a higher level of abstraction.

Most people driving a car have a good understanding of how it works. A car consists of various components such as an engine, wheels, brakes, a steering wheel, etc. We can mentally deconstruct a car into smaller parts that we can still comprehend. Now, if the car has a technical problem, we can usually make a fairly accurate guess about which of all these parts broke, even if you are not a car mechanic. With code, it should be the same.

Spaghetti code

So far, every programmer who told me they were working on a really complex problem simply wrote poor code. They all failed to break the problem into small pieces and reassemble them again. Or rather, they didn't realize they should do so and wrote spaghetti code instead. The code became so complicated that they were barely able to add any new features. If something is complex, then you absolutely have to break it down. As long as you can explain how something works in words, you can also write it in understandable code.

You should never underestimate the complexity that can arise from poorly written code. If you write a thousand lines of unstructured spaghetti code, it might cost millions to rewrite it. And this is no exaggeration!

This entire book is about writing code with low complexity. The sections on the Single Responsibility Principle, naming, and levels of abstraction are probably the most fundamental ones. It is all about learning how to write human-readable code.

Examples

Structuring Function Arguments

In the following code, one can easily mix up the different arguments since they are all of the same type. This is a very common problem in programming. The solution is to use a class object instead of a tuple.

```
def send_email(to, subject, body):
# ...
```

```
from dataclasses import dataclass

@dataclass
class Email:
    to: str
    subject: str
    body: str
email = Email(to="google", subject="new search engine", body="it's awesome")
def send_email(email):
    # ...
```

In Python (and C++ 20), this problem is less prevalent as keyword arguments are supported.

```
def send_email(to: str, subject: str, body: str):
    pass
send_email(to="google", subject="new search engine", body="it's awesome")
```

However, it is still recommendable to use a class instead of named arguments. Classes order the arguments in a logical way. The `email` is of a higher order than the three strings. It orders these three objects into one logical unit, making it much easier to understand the code. Some people start using keyword arguments for 20 function arguments. I think this is a really bad idea, even with keyword arguments. It is clear that they should have structured the arguments using dataclasses.

Complicated code

Let's review the FizzBuzz code mentioned above.

```
(lambda f, n: f(f, n))(lambda f, n: [(not n % 3 and "fizz" or "") +
    (not n % 5 and "buzz" or "") or n] + f(f, n+1) if n <= 100 else [], 1)
```

This example is challenging to comprehend because there is an excessive amount of logic concentrated on a single line. It is very challenging to keep track of all the logic that is happening here. Instead, it is much easier to understand the code if you break it into smaller pieces, as demonstrated in the following code.

```
output = []
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        output.append("fizzbuzz")
    elif i % 3 == 0:
        output.append("fizz")
    elif i % 5 == 0:
        output.append("buzz")
    else:
        output.append(i)
```

This is much easier to understand and therefore the better solution.

Assigning variables inside conditions

Avoid creating assignments within if statements. It is difficult to read and easy to make mistakes. I had to create a C++ example because such code is not possible in Python. Python returns an error message if you make an assignment inside an if statement.

```
if (int t = time_elapsed()) ...
```

The problem is that you can easily confuse this code with `t == time_elapsed()`. This is a common mistake. And as it's our job to prevent mistakes, we should avoid this kind of code.

In C++ and many other programming languages, it is possible to omit curly braces if there is only one line of code. However, this is error prone as people don't pay attention. As has happened before. It's not a big thing, but I recommend to always use curly braces.

```
if (time_elapsed() > 10)
    std::cout << "time is up" << std::endl;
    std::cout << "this will always be printed!" << std::endl;
```

Apparently, there's no such discussion in python.

Scope of variables

It is hard to keep track of variables. One way to mitigate this problem is to use them only in a small scope.

Avoid using `do while` statements if it's supported by your programming language of choice. The issue is that you need to keep track of the conditional variable throughout the entire range of the loop. This is very error-prone because keeping track of a variable over such a long time is challenging. It is much better to use a `while` loop and initialize the variable before the loop.

As `do while` loops are not supported in Python, I provide a C++ example.

```
int i = 0;
do {
    cout << i << "\n";
    i++;
    // code that doesn't use i
}
while (i < 5); // what was i again?
```

The following code using only a `while` loop is easier to understand. `do while` loops are frequently used if you want a loop to be exectuted at least once. But this can usually also be achieved by setting the variable of the loop and the condition in the loop accordingly.

```
int i = 0;
while (i < 5) { // you can find out what i is without scrolling
    cout << i << "\n";
    i++;
    // code that doesn't use i
}
```

Though the best solution is in most cases to use range based for loops. This eliminates the need for an index altogether. Since C++11 they are also available in C++.

```
for(const auto& car: cars) {
    cout << car << "\n";
}
```

It is generally advisable to minimize the number of variables in use. And they should have only a very small scope. This makes it much easier to keep track of them. If you have to keep track of a variable over a long period, you are very likely to make a mistake. Thus, eliminate intermediate results. Make logic as simple as possible. Avoid using control flow variables whenever feasible.

Limit the scope of all variables: avoid globals, keep classes and functions concise, etc. If the scope is larger than necessary, consider making the variable constant if possible. The scope of a variable should never be more than about 10 lines of code, the maximal recommended length of a function.

Approximate programming

If you aim to develop a program akin to TripAdvisor or Google Maps. Let's assume you have the longitude and latitude coordinates of each restaurant, you want to determine the closest restaurant. You have to calculate the distance on a sphere.

But in reality, such precision may not be necessary. You can simply take the coordinates and calculate the distance as you would on a map. This is sufficient for an estimate, rather than a very precise calculation. This makes the whole calculation much easier.

Copilot

Copilot is generally quite proficient at generating readable code. At times, it is even better than me. You can tell that Copilot learned "programming" based on a set of fairly well-written code. It is often beneficial to seek a second opinion from copilot as an inexpensive alternative to a code review. I think this is one of the areas where Copilot truly excels. The human-readable version of the Fizz Buzz code above was written by Copilot.

9. Single Responsibility Principle

Every object does exactly one thing. Everything is done by exactly one object.

There are various interpretations of the Single Responsibility Principle (SRP) [Clean Architecture]. I don't think the differences between them really matter. The highlighted version above represents my personal interpretation of the SRP. It is much more important that you understand the idea behind it.

The SRP is arguably the most crucial topic in this book and in all of software development. Every piece of code should have exactly one task. It is the foundation of readable and reusable code.

There is one common missconception regarding the SRP: Please note that the SRP does *not* state that every software developer is responsible for their own piece of code. The SRP focuses on code fragments, not on code ownership.

Do not Repeat Yourself

A direct consequence of the SRP is the "Do not Repeat Yourself" (DRY) principle [The Pragmatic Programmer]. You should avoid any kind of duplication in your project. You should not copy and paste your own code (copying from Stack Overflow is fine, though). Instead, you should refactor the code that would be duplicated into a dedicated function. If you have duplicated code, either copy-paste code, or conceptual duplication, it indicates that a task is not being performed by a single object but rather by two or more objects. Instead, write a function and use it from now on. This explanation of DRY covers most cases that violate the SRP.

The DRY principle not only applies to code. It also applies to processes such as constructing your project. If you have to execute many steps by copy-pasting them from some manual to build your project, something is wrong. Instead, you should automate the whole process. Write scripts to build and test your project. [97-things-every-programmer-should-know chapter 63, chapter 42]. The build should run through in one step without any warnings or errors. Warnings are unnecessary mental burdens. Even if ignored. Clean them up immediately. // where did I write something similar before? -> chapter automation?

The other case involves code that has emerged as duplicated over time. Frequently, the same piece of logic is required in multiple locations, leading to its repeated implementation due to a lack of knowledge. This kind of duplication must be refactored relentlessly. It is very difficult to detect this type of duplication as it accumulates over time. Who knows about every piece of code in a large program? Is it worth the effort to search through the entire codebase for a social security number parser, or would it be more efficient to write a new one from scratch? Writing a new one may be faster. However, this comes at a cost. If the social security number ever changes, it will be nearly impossible to locate all the bits and pieces of code related to it. This could become a significant source of bugs.

As I mentioned, it is difficult to keep track of this type of redundancies. There is no easy way to prevent them. The only way I could think of is keeping the parts of the software small and cohesive so that it is always more or less clear where a certain feature has to be implemented.

One common source of repetition is if statements. They look something like this:

```
if job == "president":
    residency = "White house"
# ...

if job == "president":
    security_standards = "very high"
```

etc. It is fairly common to have many repeating if statements. Such kind of `if` statements can be spread around the entire code base, violating the DRY principle. Though it would be quite simple to avoid them, for example by using polymorphism. Create a `President` class with the appropriate properties.

```
class President:
    def __init__(self):
        self.residency = "White house"
        self.security_standards = "very high"
```

Now you only have to create a `president` object once and pass it around. There is no longer any need for any if statements.

```
president = President()
location = president.residency
```

Another option is to use a dictionary. Dictionaries are great for avoiding switch/match or nested if statements.

```
president = {
    "residency": "White house",
    "security_standards": "very high"
}
```

Exceptions of DRY

The DRY principle does not always have to be strictly followed. It's not always worth trying to find this abstraction with only one repetition of a few lines of code. Also, the overhead of creating a new function might be higher than the benefit gained from refactoring the code. Following the DRY principle increases abstraction of the code, but at the same time also coupling. This is a serious issue.

One option is to refactor code only in case of a three-fold repetition. This is also in agreement with Test-Driven Development (TDD) [Writing better code with tests] where you only have to refactor if there is a threefold duplication of the code. // quote? Clean Craftsmanship??// Though generally I recommend refactoring as soon as you see some repetition that can be refactored out.

Advantages of the SRP

The importance of the SRP cannot be overstated. It alone makes your code an order of magnitude better when applied properly or worse when ignored. And it is fairly simple to learn. There are dozens of reasons why this is the case. Here are the most important ones.

Understanding

A function or class that implements only one thing will always be comparatively easy to understand. It all follows the same logic, and there will be no unexpected behavior. Additionally, the code for a specific problem will be concise as it focuses solely on its core functionality. All other duties are handled elsewhere.

Naming

Assigning names to objects is one of the most challenging tasks for a programmer and can be extremely frustrating. Names are either always too long or not expressive enough. This is an indication that you might have violated the SRP. If an object obeys the SRP, it does one thing. Naming an object that serves only one function is typically not that difficult. If you choose a name containing an "and", chances are high that you violated the SRP.

No Duplication

Any piece of logic should be implemented only once. This has the advantage that refactoring becomes comparatively easy since you only have to change the code in one location. Is your payment system in need of an update? Go to the `PaymentSystem` class and make the necessary changes.

Easy Testing

Writing unit tests becomes fairly simple as well. A class adhering to the SRP is not overly complex. Initializing class instances is likely not a significant issue, nor is comprehending the logic behind it. Understanding the concept of the class makes it easier to identify the key components for testing. Just look at the few public functions. As the class is straightforward, you will immediately be able to determine the expected output of the function.

Less Bugs

As the purpose of each class becomes clearer, it will be easier to structure the logic of your problem. You will only write code that makes sense. You will create fewer bugs. And it's very hard for those bugs to hide. Frequently, you will quickly identify why a bug appeared because it is immediately apparent which part of the code is responsible for the bug's behavior.

Let me provide a real-world example: You are wearing an orange T-shirt, although you should be wearing a white shirt. If the only time you have access to a wardrobe is in the morning after having a shower, you know that it was at that time that you made the mistake. Meanwhile, if you have access to the wardrobe all the time, you never know when you might have incorrectly decided to wear an orange shirt. This example nicely illustrates why it is important to restrict access to objects as much as possible and perform actions in only one location.

Bug fixing

Tracking down bugs will be much easier. You can understand fairly well what each class should do and, therefore, find unexpected behavior much quicker. Fixing a bug may seem harder at first glance. You are no longer allowed to randomly add an `if` statement in your code. This would violate the SRP and lead to bad code. Instead, you have to find a proper solution. Usually, this turns out to be easier than applying an unsightly hotfix. And especially, it really fixes the bug once and for all. All in all, we can conclude that fixing a bug becomes more challenging, but fixing it correctly becomes much easier if you adhere to the SRP.

Drawbacks of the SRP

There are very few drawbacks of the SRP that I could think of. The SRP is sometimes a bit too strict. It is not always worth obeying strictly. If a function is very short, it is not necessarily bad to have it duplicated. Adding a function to introduce an additional level of abstraction increases mental workload and may not always be justified. Though these are exceptions, rather than the norm. When in doubt, it is better to adhere to the SRP and refactor the code.

10. Levels of abstraction

"You can solve every problem with another level of indirection." – Andrew Konig

"Except for the problem of too many levels of indirection." – my hero

Levels of abstraction are an extremely important concept in software engineering. Yet, it doesn't receive the amount of attention it deserves. It applies to so many things around us, but so few people know about it. It's about taking a few objects and creating a new object with completely different properties. Something completely new emerges.

Real world example

You take a CPU, a motherboard, RAM, an SSD, and a power supply. Some of the most complex objects humankind has ever created. From some of them, you might have a rough idea of what they do, and maybe even how they work. When you assemble these parts, it becomes mind-boggling. So many extremely complex objects. And now we combine them. How is this going to end up? Surprisingly simple. You sit in front of it every day. It's a computer. And all your questions are gone. It represents a higher level of abstraction and is quite simple to use. As I write this book, I only care about the text software that I use. I don't care about the operating system (OS). I don't care about the computer that is standing on the floor. I don't care about the CPU inside. I don't care about the billions of transistors inside a device, nor do I care about the quantum mechanical effects that these transistors are based on. My text software relies on all these components, but I don't need to have any knowledge about them. All these things were abstracted away by the next higher level. The text processing program emerged from combining all these immensely complex objects.

One can also look at the problem from the bottom up. Quantum mechanics does not know anything about transistors. Transistors don't know anything about CPUs. CPUs don't know anything about computers, computers don't know anything about the OS and the OS doesn't know anything about my text software.

Some things, like quantum mechanics, just exist. We can't change them, but we can use them to create other objects. Transistors, among other components, are designed to operate inside a CPU. We can design transistors that meet the extremely stringent requirements for operating inside a CPU. You could take a CPU, break out a transistor, and use it on its own. It's just a transistor. Although it is an extremely small one. You would need an electron microscope to see it. The OS supplies an interface on which the text processing software runs, but the OS does not concern itself with the text processing software.

By combining existing objects, you create a level of abstraction. The new object has a higher level of abstraction than the previous ones. It may have completely different properties than the lower levels, also known as emergence. In theory, the higher-level object combines the complexity of all the underlying objects. However, if the higher-level object is well-designed, you no longer need to concern yourself with the lower-level objects. Just as it is very challenging to calculate the quantum mechanical properties of a simple molecule, you can still derive accurate predictions about the behavior of a combustion engine or the aerodynamics of an airplane by taking a statistical average of billions of molecules. In the very same way, I can work with my text processing software without having to worry about the OS installed on my computer.

Creating good levels of abstraction is one of the most important tasks in software engineering. This is the very core that enables us, as humans, to comprehend and address such exceedingly intricate tasks. You have to break them up into smaller and more manageable blocks that you can understand.

Programming Example

C++ is a fairly low-level programming language. Its widespread usage is mostly due to historical reasons. There are many aspects in which newer programming languages outperform older ones. But it's the same as always: The code is working and it will not be replaced due to some minor inconveniences in the programming language. About a decade ago, some of the most fundamental inconveniences were removed with the release of the C++11 standard.

C++ uses old school arrays. These commands allocate memory to store objects. If the programmer doesn't know how many objects there will be, he has to use the infamous `new` and `delete` commands to allocate memory on the heap and deallocate it in the end. These commands are extremely error-prone. They were extremely difficult to use. If you forgot to use `delete` in a corner case, the software would leak memory. Usually, you had to restart your operating system every few days for this reason. As it was leaking memory, it became slow.

Here is an example of how to use `new` and `delete`.

```
int * arr = new int[10];
arr[0] = 42;
// etc.
delete[] arr;
```

If you use `delete arr` instead of `delete[] arr`, you create a memory leak. Apparently, it is very easy to make mistakes when using `new` and `delete`, such that one should avoid using them altogether.

One of the main reasons Java became so popular in the 1990s was the introduction of the garbage collector. It took care of all the deletions. Meanwhile, there are still ways to create memory leaks in Java; however, most issues with memory management were gone. Without a doubt, that was a tremendous improvement at the time.

Though it turns out there is also a solution to the memory allocation problem using only pure C++ code. There is a simple pattern that ensures you always call new and delete in pairs. You create a class that calls `new` inside the constructor and `delete` in the destructor. No matter what you do, every object in C++ is guaranteed to call its constructor when creating and the destructor when deleting the object. The constructor and destructor are each called exactly once. Always. So, if we instantiate `new` inside the constructor and `delete` inside the destructor, they are both guaranteed to be called exactly once. The allocated memory is guaranteed to be freed. So, the entire allocation and deallocation process is guaranteed to function correctly.

Note that C++ also requires the use of smart pointers introduced in C++11 to ensure writing fully memory-safe code. But we won't be able to cover this topic here. The interested reader is referred to [citation: Effective Modern C++].

Here is a very simplified version of what the fundamental idea of the vector class looks like. Our custom `VectorClass` contains an array and manages its size. This requires some logic to understand, but ultimately, the user no longer needs to have any knowledge about the array inside the vector class.

[https://www.geeksforgeeks.org/how-to-implement-our-own-vector-class-in-c/]

```
class VectorClass {
private:
    int* arr;
    int capacity;
    int current;
public:
    VectorClass()
    {
        // allocate memory inside the constructor
        arr = new int[1];
        capacity = 1;
        current = 0;
    }
    ~VectorClass()
    {
        delete [] arr;
    }
    void push(int data)
    {
        // if the array is full, allocate more memory
        if (current == capacity) {
            int* temp = new int[2 * capacity];
            capacity *= 2;
        }
    current++;
    // etc.
    }
}
```

This idea of simplifying the usage of arrays changed C++. One of the biggest problems has been resolved. The user-friendliness has improved significantly. This pattern is used everywhere by everyone and has been called "Resource Acquisition Is Initialization" (RAII) by Scott Meyers [Effective Modern C++].

If there is a code pattern that everyone uses, it becomes part of the programming language. The `Vector` class was created. It is a higher-level object based on the array. It hides all the complex work associated with `new` and `delete` and provides an easy-to-use interface with all the essential functionality one would anticipate. The only price to pay is a slight decrease in performance due to the internal implementation details. This loss of performance is so minimal that you won't be able to measure it using any standard software. This is a perfect example that you should let the computer take care of what it can. The loss in performance is minimal, but the gain in usability is very significant.

Vectors are a higher level of abstraction than arrays. They are easier to use and superior to arrays in every aspect. Don't ever bother using old-school arrays. Don't even waste time learning more about arrays. I have told you everything you need to know.

The Abstraction Layers

// I think I have to rework this text here. Maybe I should move it into the architecture chapter?

In your code, you will also have different levels of abstraction. The upper levels always depend on the layer itself and on lower layers. The code in a layer never depends on higher levels, but only on lower levels. The code can be divided into different layers. I personally like to break it up into five layers. Though it has to be remarked that this is by far not the only way to sort the code. There are many different ways to approach it, and the number of levels depends on the complexity of the problem to be solved.

//create a Figure with levels of abstraction. Levels (bottom to top): Infrastructure – Domain level – application layer – API – acceptance tests/GUI. See DDD p.68 what the layers are used for there.

No matter if you are looking at horizontal layers as shown here, or at onion layers, there is always one rule: dependencies only go downward or inward. High levels always depend on low levels, but never on higher levels. This is the essence of the magic: my text processing software relies on the OS, but the OS doesn't need to have any knowledge about the text processing software since it operates at a higher level.

Furthermore, the dependencies should always be only one level deep. Even if some dependencies do not appear to rely on an intermediate level, they should still be routed through this level. This is important in order to decouple the code. For example, database access should always be redirected through the infrastructure layer and never be handed directly to the domain layer. You should only bypass levels of abstraction if it is absolutely necessary, for example, due to performance reasons. But this should be the exception rather than the rule.

Example of layered code

In the following code snippet, not all lines of code are at the same level of abstraction:

```python
def process_email():
    open_email()
    with open('attachment.txt', 'r') as f:
```

```
        print(f.read())
    close_email()
```

`open_email` and `close_email` are clearly functions at a higher level of abstraction than `with open ...`. In order to ensure that all the code is at the same level of abstraction, we need to relocate the `with open ...` code into a separate function. The code should look like this:

```python
def print_attachment():
    with open('attachment.txt', 'r') as f:
        print(f.read())

def process_email():
    open_email()
    print_attachment()
    close_email()
```

Now the code looks much better. All lines of code consist of function calls to higher-level functions. Every line of code within the `process_email` function is written in a way that resembles an English sentence rather than typical Python syntax. Note that the code has now become a little longer. This is not an issue. Readability counts, not the length of the code.

3rd party libraries

The lowest level of abstraction consists of the programming language and third-party libraries. You can't change those unless you replace them as a whole. Modifying code in a third-party library may be feasible in certain situations, but I strongly advise against it. Unless you incorporate the library into your codebase and treat it the same way as all your other code. Generally, this is an extremely bad idea as it involves a significant amount of work. The only reasonable approach is to contact the authors of the library and offer help to get your suggestions implemented. Therefore third-party libraries are on the lowest level of abstraction. They do not depend on any of your code.

Infrastructure code

One layer above the third-party libraries, we have our own low-level infrastructure code. These are generally all your basic data types and all the input/output (IO) code. All the technical details that the user will never see. Like the engine parts of your car. Parts that the user will not even know about. He can only guess how this stuff could be implemented, but if done properly, he will not have any clue how it's actually implemented. Neither in a car nor in your code.

The domain level

Then there is the domain level; see also [chapter Domain-Driven Design]. This is the core of your application. It contains all the business logic of your software. This is where all the complexity of your software lies. It takes an understanding of the business to comprehend this code. The domain model converts the low-level computer language from the infrastructure into human-readable text, although it still adheres to the syntax of a programming language! Every businessperson should be able to comprehend the ultimate outcome of this text.

The domain level is the part that is difficult to develop and cannot be purchased elsewhere. You have to do it yourself. This is what your company will earn money with. It's the core of your business.

The application level

The next level is the application-level code. Here, the code follows a logic similar to the problem we are solving. Variables and functions have the same names as those used by the salesperson. It also follows the same logic. If a marketing professional reviews the application-level code, they should be able to comprehend the process and potentially identify any errors.

API

One level higher is the API. This defines the interface between our code and the user. It is a wrapper around the application-level code. The API provides all the functionality that users would expect in an easy-to-use format. However, the API is not at the highest level. It is still one level below the Graphical User Interface (GUI). It is of utmost importance to decouple the API from the GUI. The API should have no knowledge of the GUI, and the GUI should solely utilize API functions! And the same applies to acceptance tests.

GUI and acceptance tests

At the highest level are the GUI and the acceptance tests, both at the same level. If you ever develop a GUI, ensure that its code is entirely decoupled from the rest of the system's code. The only interaction should be through your API. The same principle applies to acceptance tests [section Acceptance Tests]. The GUI and the acceptance tests operate at a higher level of abstraction compared to all other code you work with. Already, the programming language for the GUI is completely different. You may write HTML! Due to the SRP you are not allowed to write any logic in the GUI. Writing tests for the GUI can be challenging. Therefore, the only solution is to write acceptance tests at the API level and ensure that you never break the GUI by maintaining its simplicity.

Summary

As a summary, I want to emphasize once again the tremendous importance of abstraction levels. Different abstraction levels are the key reason we can comprehend highly complex systems. And it's your job to define the abstraction levels for your code. Avoid mixing different levels of abstraction.

11. Interfaces

"Make interfaces easy to use correctly and hard to use incorrectly" - Scott Meyers

Interfaces are closely related to levels of abstraction. Each level of abstraction has two interfaces. One is on the low-level side, and the other is on the high-level side.

In this chapter, we learn that interfaces exist not only in software but also in the real world. And we can learn a great deal from them. An interface is always the connection between a developer and a user. It is defined by the developer, but it should be designed from a user perspective, as the developer only has to implement it once, while users might have to interact with the interface thousands of times. Therefore, it pays off to design an interface properly, as it was already explained in the chapter on [levels of abstraction].

Real-world Interfaces

Functions, classes, libraries, and complete software or smartphone apps all have interfaces. Even technical objects, such as plugs, have an interface. The technical details may vary significantly, but the basic principles are very similar.

"Plugs," you may laugh. Yes, even plugs. Electric plugs in America look different from European ones. It is impossible to plug an American plug into a European socket, and vice versa. This is due to historical reasons, but at the same time, it is also a safety measure. It prevents you from connecting an American 110V device to the European 230V grid, potentially causing damage. It's fail-safe. It is a good design that they are not interoperable. Though most devices can now handle both voltages.

An example of poor design is the USB-A port. The USB cable appears symmetric on the outside, but in reality, it is not. Someone once said that you always need three attempts to plug in a device with a USB-A cable. The first time would have been right, but you didn't manage it. The second time was the wrong way around, and the third time you managed to plug it in. The USB-C port features a much more user-friendly design. You can plug in the cable either way. The lanes can be connected either symmetrically or asymmetrically. The technicians implemented a solution that enabled both types of connections. The two devices involved must negotiate with each other on how to utilize the various lanes of the cable. This was some additional work for the engineers. But, once solved, it becomes a very convenient solution for the users.

Another example are water tabs for showers, as previously discussed in the section on orthogonality. There are two tubes for cold and hot water where the plumber attached one valve to each. This was a pain to use. It took quite a while to set the temperature correctly, and once you changed the amount of water, the whole procedure started again. This was the engineer-friendly solution, not the user-friendly one. This was a bad interface. The new handles allow you to choose the amount of water and the temperature separately. This might be a bit more complicated to implement, but it's much more convenient to use.

Notice how both solutions have 2 degrees of freedom. A mathematician would refer to this as a coordinate transformation. With the old valve, you and all other users had to perform this transformation yourselves. With the new valve, this issue is resolved permanently through mechanical means.

I hope these simple examples gave you an idea of what good interfaces are about. If you design an interface, you should always know your customers. What do they do? How do they think? How will they utilize your product? This is of utmost importance. A good interface is user-centric. It represents the way the user thinks and conceals all the technical details.

Code Interfaces

Once again, understanding interfaces in general will enable you to write much better code. It's just the same as in the real-world examples above. Try to follow the same principles. Figuring out what the user really wants makes writing a well-designed interface quite easy. Writing some user code examples will help you a lot, as you'll learn in the section [Test Driven Development] (TDD).

Always define an interface from the user's perspective. What does the user want? How does he want to use your code? These are the important questions to ask. Don't just blindly return the values you have.

An interface that is designed from the engineer's point of view is usually poorly designed. It is designed from the wrong perspective. An engineer's interface is easy to implement but not necessarily easy to use, as engineers tend to focus on what they have. They lack the vision of what they could have. Thus, they miss the point of a good interface. An engineer's interface is like an old Nokia phone. The shape and functionality were mostly determined by the engineers' preferences. The designers had little to say and were only allowed to smooth out the edges slightly. Meanwhile, a good interface is more like an iPhone. Here it was the other way around. Designers instructed the engineers on the necessary tasks, resulting in a phone with a user-friendly interface. This is how you should design your interfaces. You need someone with a vision for how your code should be utilized. Not just an engineer who excels at implementing the code but lacks understanding of how to use it.

Interfaces are everywhere. Every function [Functions] or class [Classes] has an external interface and utilizes multiple interfaces from other functions or classes. This is why understanding good interface design is paramount. Especially with classes, it is challenging to define a good interface that allows the user to perform desired actions without revealing too many internal details of the class. When working with functions, it is important to consider the order in which function arguments should be arranged.

Example

This is a code example for a car. The car has a current `speed` and a `top_speed`. However, the user of this code doesn't know anything about these attributes. He only sees the public interface containing the methods `accelerate`, `brake`, and `get_speed`. He doesn't know anything about the implementation of this class.

```python
class Car:
    def __init__(self):
        self._speed = 0
        self._TOP_SPEED = 200

    def accelerate(self, amount):
        assert amount >= 0
        self._speed = min(self._speed + amount, self._TOP_SPEED)

    def brake(self, amount):
        assert amount >= 0
        self._speed = max(self._speed - amount, 0)

    def get_speed(self):
        return self._speed
```

APIs

"With a sufficient number of users of an Application Programmable Interface (API), it does not matter what you promise in the contract; all observable behaviors of your system will be dependent on by somebody." - Hyrum's Law

If you are expecting a comprehensive chapter that explains all the details of APIs, I'll have to disappoint you. This is a vast topic, and I can only scratch the surface here. I will only explain some of the most important aspects of APIs that I could think of.

An API is an extremely important component of your software. It is the public interface of your software. It is what everyone sees and uses from the outside. Everything we discussed in the interface section applies here as well, but in an API, it is crucial to get everything right. Having a bad API will cost you a lot of money. People won't buy your product if the user experience is bad. They would rather go to the company next door and buy their software. "They even support emojis!" Yes, sadly enough, supporting emojis is important nowadays for business reasons.

That was no joke, by the way. Apple once had an important security fix in their latest update. The update includes new emojis. For many users, emojis serve as a stronger motivation to install an update compared to a security fix.

APIs are an extremely complex subject. Not so much for technical reasons, but rather because you interact with users external to the company. They use your code hidden underneath the API. Every change you make in your code could potentially lead to a bug in your client's code. Even fixing a small bug in your own code. When maintaining an API, you have exactly one task: Never, ever break your clients' code! Now you might think this is doable. But I can promise you will have nightmares.

You are always allowed to add new functionality as long as you do not alter the functionality implemented with the old syntax. The old code is guaranteed to run exactly the same way as it did before, but you can also utilize some new functionality. Vice versa, you are never allowed to change or delete existing functionality. This could result in compilation errors or, even worse, bugs in the user code. And that's when customers go on a rampage. "Up to now, the code worked, but all of a sudden, it fails. What the **** did you do?" If you don't understand this harsh reaction, you've never had a work colleague randomly break your code once in a while. You would feel exactly the same.

Adding more functionality

You want to add a new option to one of your API functions, but there is a lot of existing customer code. This code does not currently utilize this new option and won't use it in the future. How can you add this option without breaking this old user code?

The answer is default arguments. The current behavior is set to be the default. After the update, the user can select an alternative option within the function call. This works in all modern programming languages. You don't even need an if statement.

Let's make a brief example. Let's consider the following function:

```python
# version 1.0
def my_super_function(arg1):
    return arg1
```

We can easily modify this function with the following code. We added a flag (`arg2`) that alters the functionality. The function now only returns the `arg1`, if `arg2` is set to `True`.

```python
# version 1.1
def my_super_function(arg1, arg2=True)
    if arg2:
        return arg1
```

However, you can also omit the `arg2`, and the functionality remains the same as it was before the code was changed.

```
my_super_funtion("hello")
```

returns `"hello"`, regardless of the version number.

Removing functionality, on the other hand, is really hard. This inevitably changes the behavior of existing functionality. You are not allowed to do so except under very special circumstances, as explained in the next section.

Semantic Versioning

APIs have version numbers. These are 2 or 3 numbers separated by dots. For example, "3.11.2" was the latest Python version at the time of writing. "3" represents the major version, "11" represents the minor version, and "2" represents the trace. The trace is only used in larger projects.

Every time you make a new release, you increase the version number.

- For bug fixes or internal improvements, you increment the trace number. This is for all kinds of changes that the user shouldn't notice or probably doesn't care about. The user should be able to switch to software with a higher or lower trace version without any issues.
- The minor version number is increased for new features. The changes explained so far are still backward compatible as they don't alter any existing functionality.
- The really big disaster begins with major version changes. Sometimes this is required. And it is dreadful. You might think that it's not so much effort for the customers to change some code. "HA!" Think again. Migrating most of the Python 2 code to the major version 3 took 12 years, and support for Python 2 was only discontinued a few years ago. The transition was quite a nightmare because many available libraries had not been updated yet. Users simply don't have time to update their code to a new major version of your library. So, if you don't want to lose them, you should make sure you don't break the old interface. Only increase the major version of your software if it is absolutely necessary.

Usually, companies support multiple API versions simultaneously. They know that their users need time to adapt to the new version. Some users will never adapt at all. They are forced to support the old API versions for many more years, even though a better API is available.

Orthogonality

Orthogonality is a mathematical concept. It has been used in software engineering by Thomas and Hunt in their highly recommended book "The Pragmatic Programmer" [](The Pragmatic Programmer). Orthogonality states that two objects are at a right angle in the current coordinate system. The first part of this sentence may seem intuitive, but what about the coordinate system...? Let me explain code orthogonality by providing a brief example that is familiar to everyone.

[](image Shutterstock) On the right-hand side, we have old-school water taps. The user has 2 degrees of freedom (if you're not into math: 2 function arguments), one for the amount of cold water and one for the amount of warm water. However, this is not what the user typically desires. It turns out that the user wants to be able to control the two degrees of freedom differently. He wants to control both the total amount and temperature of the water. The orthogonal solution from the user's perspective is shown on the left-hand side. The solution on the right-hand side is outdated. In the engineer's coordinate system, it is orthogonal.

However, nowadays, users have higher requirements and are no longer satisfied with the engineers' solutions. We expect this coordinate transformation into the user's coordinate system to be performed within the water tab.

In software engineering, we encounter exactly the same phenomenon. We have a downstream person (user) and an upstream person (developer). Both want to work with orthogonal data, but they may be operating in different coordinate systems. Now, it is always the upstream person's job to transform the output to make the data orthogonal in the downstream person's coordinate system. In similar cases, it is always the upstream person's duty to make the downstream person's life as comfortable as possible by converting the data handed over. This also makes sense from an economic standpoint: there is only one developer (upstream person), but many users (downstream persons). So, if the developer handles the coordinate transformation, only one person (or team) needs to do it, as opposed to all users having to do it themselves if the developers don't take on this task.

It may not always be obvious how the downstream would like an interface to look. When in doubt, the upstream should return the data in the most general representation. Make sure that no implementation details leak into the interface, even though this is sometimes easier said than done. This general interface has the highest likelihood of being orthogonal from the user's perspective. And try to minimize the interface as much as possible. Less is more.

Frequently, you cannot choose how the data looks when you work with it. For example, if it originates from a third-party library. The data at hand does not align well with the algorithm you intend to use for your specific problem. In this case, you should first orthogonalize the input data before continuing. Separating the orthogonalization and algorithm steps is much simpler than running an algorithm on a dataset that is not optimally set up from the beginning. A common example is the coordinate transformation between spherical (r ϕ θ) and Cartesian (x y z) coordinates. Some problems are easier to solve in one coordinate system, while others are more easily solved in the other coordinate system. In most cases, it's best to first convert the data into the appropriate coordinate system, rather than adapting the algorithm. This keeps the algorithm and the coordinate transformation separate, following the SRP.

Advantages of Orthogonal Systems

Working in an orthogonal system has many advantages:

- Errors propagate directly through the system and are easy to find. They don't spread out.
- Fixing these bugs is easier because the system is less fragile.
- Writing tests for an orthogonal system is easier.
- It decouples the code because the transformation acts as an adapter.

Example of an adapter

Let's say you have an electric sensor. It measures the amount of light in the room by detecting a voltage. However, this voltage is not the final value you want to work with. Instead, you want to know the density of light, measured in watts per square meter (W/m^2). So, you need a function that converts the voltage into the desired units. You do this transformation once where you get the measured voltage and this problem is solved once and for all.

```python
def light_density_from_voltage(voltage):
    lumen_per_volt = 10
    return voltage * lumen_per_volt
```

Now, this function returns the orthogonal data for this specific example. Of course, the transformation required in your code may look completely different.

Copilot

Copilot is generally not very good at writing interfaces. Instead, you should do this yourself and let Copilot fill in the gaps. This is generally the better approach than writing comments and letting Copilot define code based on them.

12. Naming

"And you will know, my name is the Lord!" – Samuel L. Jackson, Pulp Fiction citing the bible" [https://youtu.be/MBRoCdtZOYg]

This chapter is a futile attempt to help you find better names. If you are not satisfied by my lousy explanations, I recommend the book [The Art of Readable Code] which has some more detailed explanations.

The importance of Names

How long does a football game last? This is a very innocent question, although people may not agree on an answer. In Europe, most people would say 90 minutes, while in the United States, 60 minutes is the common answer. The reason for these different answers is very simple: names. There are two different sports that share the same name. This can cause some confusion.

The example was cute. Mixing them up may cause amusement, but it does not cause any harm. When it comes to city names, things can get a little trickier. If you miss a job interview because you drove to the wrong city named "Springfield" (this name is used in The Simpsons because it is a very common name in the US), it can be quite painful. For the police and healthcare system, it becomes even worse. When there are individuals with identical names present, it can become risky. If your namesake is a highly dangerous criminal, the police may become really rough because they are confused and think you could be dangerous. In a hospital, there are issues with using names as an identifier, and so far, there is no unique solution on how to solve it. Using the name combined with the birth date works out quite well, but it is no definite solution.

All these things happen for only one reason. Name collisions. Various objects sharing the same name. Names are everything. No matter what you look at, you can name it. A computer, desk, printer, etc. This is the very foundation of our natural language. Of every language. Including programming languages. In a programming language, we define things by giving them a name. Every variable, function, or class has a name. Every programming construct has a name. You can use this name to search for it on Google or Stack Overflow. If you don't know the name, you're in trouble.

Choosing good names is paramount in programming. You certainly don't want to encounter name collisions as explained above. It would cause a lot of confusion and could be the source of many errors in the future. But there is much more to consider when defining the name of an object. We are humans, and we need to be able to read and understand the code. This would not be possible if we used randomly generated names. We need names that provide us with an understanding of an object's purpose and characteristics. This is the only way we can create a mental image of what the code roughly does. It is necessary for everyone involved in the project to understand the meanings of all these expressions. What kind of properties does this object have? Here we can learn from the law. In the law, every expression has a set of properties. We have to do the same when writing software. This is the only way we can prevent missunderstandings and ensure that everyone understands the code.

Though, consistency in naming is more important than the actual name. If someone came up with an imperfect name, you either have to change it everywhere or stick to it.

Coming up with your own names is anything but easy. Especially new programmers really struggle to find good names. There are just too many possibilities for naming an object. But there are some rules you can follow, and at least some of the names are quite easy to find. Meanwhile, for other variables, even experienced programmers have to think deeply. In fact, naming consumes a significant portion of our programming time. We do it very often, and there is often no obvious solution; there might be only some vague recommendations. Or as Michael Feathers stated in his book "Working Effectively with Legacy Code":

"When naming a class, think about the methods that will eventually reside in. The name should be good, but it doesn't have to be perfect." [WELC p.340]

How to name things

As I already mentioned, naming is one of the most challenging aspects of programming. I tried to collect and synthesize some rules on the properties of good names. The result is a pretty long list of unfortunately quite vague recommendations when naming things:

1. Names should be short yet clear. Thus, there is a constant trade-off regarding the length of a name. Short names may be unclear, while long names may indicate that the object is difficult to describe. On the other hand, long names are not as detrimental as unclear names. When in doubt, choose a longer name. For example: Should you choose `p`, `price` or `price_of_apple`? The answer is: it depends on the context. As a rule of thumb, a variable name is fine if a new work colleague can understand it.
2. Use the same words for creating a name that you would use in a comment. If you use different words, either your abstraction may be bad or your naming is inconsistent.
3. Think about how you would articulate a word in an everyday conversation. Would you refer to it `price of an apple` or is the context of your conversation clear enough that only `price` is sufficient?
4. Classes and functions that adhere to the SRP are relatively easy to name because they perform only one task. Vice versa, if it's difficult to find a suitable name, reconsider whether the object adheres to the SRP and rewrite it accordingly.
5. `set_color(7)`. What does `7` mean? Avoid using raw values in your code. Plain values are referred to as Magic Numbers because their meaning is not immediately apparent. Your code should be understood! Always create a variable instead of using magic numbers. It is better to use `set_color(RED)`, where RED is a constant or, even better, an enum [section enums]. Both are much clearer.
6. Well-defined levels of abstraction result in clearly defined and unique properties. This helps with finding names. Maybe you have created a level of abstraction that also exists in real life. At the same time, functions and classes are required to be at a single level of abstraction in order to fulfill the SRP. [chapter levels of abstraction]

7. Name collisions between different libraries are common and nothing to worry about. Use namespaces to distinguish them. Use the `from ... import *` syntax in Python cautiously as this removes this potentially crucial information about where a function is defined.
8. Name collisions within the same library may occur occasionally and need to be resolved. Rename or even refactor one or both variables involved. They might perform very similar functions and should be refactored into a single object. Otherwise, you should be able to find clearly distinguishable names.
9. Use names that are commonly used in real life. Ensure that the object in the code and the actual object have very similar properties. You should be able to communicate with a domain expert about your code, and he should understand at least some of your problems. If he doesn't understand you, you probably used names or a model that do not exist in reality. You did a great job if a marketer understands your high-level code and can provide you with useful feedback.
10. Objects have names that are easy to distinguish. Differences in the names should be as early in the word as possible. `apple_price` and `orange_price` are preferred over `price_of_apple` and `price_of_orange`, although this preference can change if you have different properties of apples as well.
11. Use common English words that are familiar to everyone. Avoid abbreviations unless they are commonly used in spoken language, such as "CEO", etc. Whether an abbreviation is "commonly used" depends on the context.
12. You are allowed to adjust the language slightly and sometimes disregard grammar rules. If you have many `fish`, you may call them `fishes` to highlight the plural. Being able to understand the meaning of the code is more important than the usage of proper English. Natural languages have some deficiencies when it comes to explaining things unambiguously. The following code is perfectly viable in Python: `for fish in fishes`.
13. Avoid using "if," "and," or "or" in the names of your variables, functions, and classes. These concise terms may be appealing to employ, but they clearly indicate a breach of the SRP.
14. When a variable is utilized extensively throughout the code, it is important to name it thoughtfully. Consider using a name provided by the marketing team or existing theories and literature. If a variable is used for only about 5 lines, even i, j, or k are fine.
15. The name of a function should clearly indicate its purpose. There shouldn't be any unexpected behavior hidden in the code. For example, it shouldn't interact with global states, which is generally considered a poor practice.
16. snake_case notation is easier to read than camelCase or PascalCase. This is why I use snake_case notation for variables and functions and PascalCase for class definitions. Though it is more important to stick to the rules established in an ongoing project than coming up with your own notation.
17. Classes and functions should reveal their purpose through their names. This relieves the developers from reading the internals, thus saving a lot of time. The name should be a part of the domain language.
18. Prefer explicit names over implicit names; choose `hammer` over `nail_smashing_rod`. Avoid using generic terms such as "data," "information," or "manager". They don't tell you anything. The name `server_can_start()` is vague compared to `can_listen_on_port()`.
19. Attach units to a variable name if they exist. For example, `timeout_duration_ms`. Though, once again, consistency is more important.
20. Avoid using negated terms (and preferably avoid booleans altogether). `is_not_empty` is more difficult to read than `partially_full`.
21. Normal reasoning should be able to help you understand how an algorithm generally scales. A function `size()` should not have a time complexity of O(n). If you want to create a function that calculates the

size in O(n) time complexity, you should name it `compute_size()`.
22. At times, it is suggested to use a trailing underscore character for class variables. This is to distinguish them from local variables. However, I think this is a sign of poor code. If you require such a distinction, your methods are likely too lengthy, and your class may be too large.
23. According to Robert C. Martin, high-level objects have short names because they describe very general things. Low-level objects have long names because they are very specific [clean code?]. This rule, in my opinion, is quite inaccurate. It rather depends whether the object is part of a well-known interface or not. Generally names on interfaces are short because they are commonly agreed upon. For example "sin" in the math library.

Naming Antipatterns

Useles Words

Sometimes words within a name can be omitted without losing any information. For instance, instead of using `convert_to_string()`, the name `to_string()` is shorter and does not lose any crucial information. Similarly, instead of using `do_serve_loop()`, the name `serve_loop()` is just as clear. Similar words line `manager` do not add anything to the name of a variable and can therefore be omitted.

Generic Names

Another problem is using overly generic names, as shown in the following example:

```
class Rectangle {
    def size():
        # ...
}
```

What does `size()` exactly mean? It is a very generic name. Is it the area or the length of one side? The name is not specific enough. The name `area()` would be much better. Or `length()` if it represents the length of one side.

Here are some examples of generic words and some more specific alternatives. These examples are from the book [The Art of Readable Code].

Word	Alternatives
send	deliver, dispatch, announce, distribute, route
find	search, extract, locate, recover
start	launch, create, begin, open
make	create, set up, build, generate, compose, add, new

It is quite common for the author of a code to struggle with naming variables and opt for a very generic name. This, however, is really bad practice. Names should be as specific as possible. It is okay to use a generic name temporarily and replace it later when you are more smarter. Avoid using generic names in your code. They are a sign of laziness. Even Copilot can help you find better names.

Copilot

Naming is one of the most challenging tasks in programming, and Copilot is a great aid. One thing you can do is write some code and then let Copilot find appropriate names for you.

```python
def print_states(states):
    for a in states:
        print(a)
```

Here `a` is clearly not an appropriate name. Writing a comment to Copilot to search for a better name works out quite well.

```python
for a in states:
    # find a better name for this variable
```

Though Copilot needs some help to get started and I had to write the beginning `for` in order to get the following suggestion:

```python
for state in states:
    print(state)
```

This is pretty much what was expected. The same works out for function names as well.

```python
def some_fancy_function_name(b, c):
    return b + c
```

```python
# suggest a better function name
def add(b, c):
    return b + c
```

-# Part 2: Components of Code

13. Functions

"Functions should do one thing. They should do it well. And they should do it only." - Robert C. Martin [Clean Code, chapter 3]

Functions (and methods) are, along with classes, the backbone of modern object-oriented (OO) software. People just don't seem to care about functions as much as they do about classes. They are fairly simple to use, and there are only a few things to take care of. Still, there is quite a lot to know about functions as well. We will learn why functions must adhere to the SRP and should not have any side effects.

Throughout this book, we will distinguish between functions and methods, as is common practice among most authors. Even though I would personally like to refer to both of them as functions since they are essentially the same, just in slightly different contexts. Most of the concepts I write about functions also apply to methods, as they are very similar in many respects. Methods have access to class variables, which are comparable to additional mutable function arguments. If you regard them as such, almost all arguments hold for both, functions and methods. Remaining differences should be clear from the context.

Do one thing only

Due to the [SRP], functions should only cover one level of abstraction. Therefore, they have to be short. As a rule of thumb, functions should be at most about twenty lines long (that's what fits on my laptop screen without scrolling), although less than 10 lines is certainly preferred because shorter functions are much easier to understand. This is due to the fact that longer functions also contain more variables. So the complexity of a function scales quadratically with length!

There is absolutely nothing wrong with functions that cover only one line of code. One-line functions are extremely useful for enhancing code readability as they elevate all code to a consistent level of abstraction. But this is something that many programmers don't consider.

Here is a very short code snippet:

```
#define enums for Color and Flavor
if fruit.color == Color.yellow and fruit.flavor == Flavor.sour:
    make_lemonade(fruit)
```

This code is far from optimal. It is implicit. Of course, a fruit that is yellow and sour is a lemonade. But it takes thinking and it is not immediately clear. Let's look at the following code instead:

```
def is_a_lemon(fruit):
    return fruit.color == Color.yellow and fruit.flavor == Flavor.sour

if is_a_lemon(fruit):
    make_lemonade(fruit)
```

Here we refactored the relevant code into a function. It is only one line, but it makes the code so much more readable.

Levels of indentation

"If you need more than 3 levels of indentation, you're screwed anyway, and should fix your program." - Linus Torvalds

The easiest way to assess the complexity of a function is by counting the number of levels of indentation. Having no or very little indentation in your functions is always a very good sign. This implies that there is hardly any complex logic concealed within a single function. Having nested `if/else`, `while`, or `for` loops would violate the SRP because the function has two tasks: resolving the logical operator and performing other work. Having only a few levels of indentation in a function automatically makes it easier to name and understand. At the same time, one needs to get used to the formatting of such code. Most code is typically written at the first level of indentation.

A frequent problem is deeply nested `if/else` clauses. [https://youtu.be/rHRbBXWT3Kc]

```
button = input("")
if button != "":
    if not is_sleeping():
        if not is_eating():
            attack()
        else:
            print("Cannot fight while eating")
    else:
        print("Cannot fight while sleeping")
```

This code is very hard to understand. It has too many negations. As I mentioned earlier, there are too many levels of indentation. This issue can be resolved by rearranging the `if/else` clauses. Avoid letting them span across the entire codebase. Check if the `button` is empty and return if it is.

```
button = input("")
if button == "":
    return
if not is_sleeping():
    if not is_eating():
        attack()
    else:
        print("Cannot fight while eating")
else:
    print("Cannot fight while sleeping")
```

We can apply this technique to the other `if` clauses as well. The resulting code will look like this:

```
button = input("")
if button == "":
    return
if is_sleeping():
    print("Cannot fight while sleeping")
    return
if is_eating():
    print("Cannot fight while eating")
    return
attack()
```

Assuming that this code is written inside a function, we have two levels of indentation, so we are compliant with Linus Thorwalds' rule.

With this technique, the code became much easier to read. Of course, one could also use `if/else` clauses instead of the `if... return` statements. Depending on the complexity of handling the conditions, one could consolidate all conditions within a dedicated function that performs all the necessary checks. Something like this:

```
def can_fight(button, fighter):
if button == "":
    return false
if is_sleeping(fighter):
    print("Cannot fight while sleeping")
    return false
if is_eating(fighter):
    print("Cannot fight while eating")
    return false
return true
```

Assuming that global variables are not used, I had to include the `fighter` object as a function argument. Instead, one could have also have written this code inside a class. But these are technical details.

Anyway, we have seen some approaches on how to deal with nested `if/else` clauses. There is usually no perfect solution, but at least we have improved it significantly compared to the initial code.

Naming

Naming becomes less challenging (I would love to write "easier", but it's never easy...) if you follow the rules below:

- The name is a summary of the function content.
- There are no hidden behaviors within a function.
- There is no unexpected behavior within a function.
- The entire function body is one level of abstraction lower than the function name.

The following function clearly has a side effect:

```
counter = 0
def log_in(email_address):
    counter +=1
    check(email_address)
```

The function name does not indicate the presence of a hidden counter, making this hidden behavior that should be avoided. Additionally, side effects may lead to temporal coupling [see next section] because the order of calling functions with side effects matters.

A more suitable name for this function could be `log_in_and_increase_counter`, but this would reveal that the function performs multiple tasks, contradicting the SRP. A function name should not contain an `and` as this indicates a violation of the SRP.

Side effects can also become a significant issue when testing code. When calling the function `log_in` twice, the value of `counter` will be different each time. This will make the tests very fragile due to temporal coupling. And as we will learn in the chapter on testing [Writing better Code with Tests], brittle tests are a strong indication of poor code quality.

Temporal Coupling

Temporal coupling occurs when tasks can be performed in an incorrect sequence. Sometimes the code enforces the correct order, and sometimes it does not. Most notably, temporal order is not enforced by classes. Class methods can usually be called in any order. There is nothing enforcing the correct order. The class variables are there all the time. Let me make a brief example:

```python
class Shopping():
    def get_money(self, amount):
        self.money = amount

    def create_shopping_list(self, shopping_list):
        self.shopping_list = shopping_list

    def go_shopping(self):
        # use the shopping_list and money
```

Apparently you need to get money and create a shopping list before you go shopping. The correct usage of this class is as follows:

```python
shopping = Shopping()
shopping.get_money(50)
shopping.create_shopping_list(["apple", "banana"])
shopping.go_shopping()
```

This sequence of function calls follows the natural order of the shopping process. First, you need money and a shopping list before you go shopping. However, this order is not enforced by the code. One could also swap two of the function calls as follows:

```python
shopping = Shopping()
shopping.get_money(50)
shopping.go_shopping()
shopping.create_shopping_list(["apple", "banana"])
```

Now you go shopping before creating a shopping list. In fact, the call to `create_shopping_list` is probably superfluous because the shopping list might not be used anymore. Instead, you go shopping with a shopping list, which is either empty or non-existent. Either way, this behavior is apparently undesired. It would probably be best to throw an exception in this case.

This is one of the drawbacks of OO code. Methods change the state of the object. Therefore, it is very difficult to enforce that the methods are called in the correct order.

One advantage of procedural code is that such issues are less likely to occur. Code doesn't always have to be OO. Sometimes, other paradigms produce better code. Let's examine the procedural version of this code.

```
money = get_money(50)
shopping_list = create_shopping_list(["apple", "banana"])
go_shopping(money, shopping_list)
```

In this case, it is physically impossible to go shopping without having a shopping list, as shown below:

```
money = get_money(50)
go_shopping(money, shopping_list)
shopping_list = create_shopping_list(["apple", "banana"])
```

After having swapped the last two lines, this code cannot be executed anymore because the variable `shopping_list` is not initialized at the `go_shopping` function call. When executing the code above, you will get an error: `NameError: name 'shopping_list' is not defined`. This prevents you from calling the functions in the wrong order.

Long story short: Ensure that your functions never have side effects. Functions and methods should only affect the class instance or, if necessary, mutable arguments. If possible, enforce temporal order, for instance, by utilizing functional programming.

Another source for temporal coupling are global or static variables.

```
counter = 0
def log_in(email_address):
    counter +=1
    check(email_address)
```

Here, the value of the `counter` depends on how often `log_in` has been called before. Now it might make sense to have such a counter in your code; however, this may lead to all kinds of problems. For example, when testing it [Testing?].

Number of Arguments

As for the length of the function, the number of arguments should be kept to a minimum as well. This simplifies the function significantly. Having too many variables is always a sign of too little cohesion while at the same time, the function becomes difficult to understand. Here, I try to provide a rough estimate of the number of variables a function or method may have. But this ultimately depends on the overall complexity of the code, etc.

Note that member variables to methods behave like arguments to functions. Therefore you have to add up method arguments and member variables to get the total number of variables. And we assume there are no global variables.

Now, there are very few functions with zero arguments. These functions are the simplest; they always behave consistently. There's not much to test, but at the same time, there isn't much that such a function can do. Especially if it's a pure function [section Functional Programming] which doesn't have any side effects..

As a function has more arguments, it can encompass more functionality. Yet, at the same time, it will become more complex. Functions with one or two arguments are usually fairly easy to handle and should cover most of the code. Functions with three arguments are already quite complex. They are difficult to understand and challenging to test.

Try to avoid functions with more than three arguments. This shouldn't be a significant burden. A plumber can manage to carry all his tools with only two hands, thanks to the invention of the toolbox. Why shouldn't we be able to juggle everything with just 3 arguments? We can use our equivalent to a toolbox: the data class (Python) or struct (C++) [Classes]. If you are struggling to fit all the variables you require into three struct objects, it may be necessary to rethink your function design.

When combining method arguments and member variables, it is very easy to exceed the recommended limit of 3 variables. This is the main reason why I don't recommend using classes excessively. Instead, I generally suggest opting for procedural or functional programming. When dealing with classes with complex methods, there should be as few class variables as possible. Otherwise, its complexity will blow up [worker classes].

A method may access certain class variables. However, one does not know this until one has read all of the methods and sub-methods involved. Furthermore, one must check whether a method modifies the class variables, unless it utilizes the C++ `const` expression. It is advisable to minimize the total number of variables in use. This is the only way to keep the code maintainable.

Following the SRP, functions can only be either a query or a command [https://en.wikipedia.org/wiki/Command%E2%80%93query_separation], but never both at the same time. In the best case scenario, you save one line of code by avoiding an extra check. But at the same time, you make the code more confusing because handling two responsibilities is much harder than dealing with just one. And potentially saving one line of code is not worth violating the SRP. A common antipattern in this regard is returning a boolean flag with a set command.

```
if set_node("money", 50):
    go_shopping()
```

Here, the `set_node` function performs two actions simultaneously. It sets a value and returns a boolean. This certainly doesn't help with understanding the code. I would find it better if `set_node` raised an exception if it failed.

Copilot

Copilot can help reduce the number of arguments by structuring them using dataclasses. But the suggested code is not always better.

Let's say we have the following code:

```
def do_something(a, b):
    return a + b
```

With the command `put a and b into a dataclass`, we receive the following suggestion. Now, as I already mentioned, the suggestions from Copilot are not always improvements. Whether such kind of refactoring makes the code more readable is a highly specific question and has to be decided by the reader.

```
@dataclass
class Numbers:
    a: int
    b: int

def do_something(numbers: Numbers):
    return numbers.a + numbers.b
```

Output arguments

One very irritating thing is functions that alter the value of their arguments. This is also a very common source of bugs as it is something quite unexpected. Now, once again, in C++, one can clarify this by specifying the type of the argument. One can pass the argument by reference to make it modifiable, or by const reference to make it non-modifiable. However, in other languages, this clarity has to be inferred from the context of the function.

Changes to function arguments can be challenging to keep track of. For this reason, a function should always modify at most the first argument. Modifying two arguments violates the SRP and is even more confusing. If you change the value of an argument, it has to be the most important argument. It's a dual-purpose input and output argument. So, it has to be special. It has to be first.

Output arguments can be compared to class instance objects. They are essentially both function arguments that may change their values. Just that the class instance is obviously a very special variable. The function acting on those variables may change the value of the output argument or the class instance, thus potentially causing side effects. This is sometimes necessary, but at the same time, it is undesired behavior as it is difficult to keep track of.

As always, output arguments give you a lot of power when used wisely. But at the same time, they can also be a source of very unreadable code when used carelessly.

Return Values

Return values are, in my opinion, very normal, yet many OO programmers tend to dislike them. These OO programmers work with class methods that manipulate the existing class instances. But return values have a distinct advantage as their intention is clearer. It states: this is a new value. Compared to: This method may alter a variable of the class instance. And once again, keep in mind the SRP. A function should only have either a return value, an output argument, or change a class instance. But never two of them at the same time.

Return values are central in functional programming. In functional programming, you are not allowed to alter the values of existing objects. So, you don't have the issue of function arguments changing their values. The workaround are return values. They have the advantage that it is obviously a new object with new properties. For each state the code is in, there is a different set of variables. You'll never have to track the state of a variable because each variable has a unique state. After every step of your computation, you create a new variable so that you will never store different information inside a single variable. You just create a new one.

All together I think there is really nothing wrong with return values. In performance-critical code, creating new objects all the time may be a problem. But in most cases, I recommend working with return values, rather than mutable objects. They are more explicit and easier to understand.

Here is a small Python example. There are two ways to sort elements in a list. You can either use the `sorted` function, which returns a new list, or you can use the `sort` method, which changes the list in place. I generally recommend using the `sorted` function because it makes it clearer that the return value is a new list with different properties than the function argument. When using the `sort` function, the programmer may forget that the elements in the list are now sorted.

```
L = [1, 6, 4, 3, 3]
sorted_L = sorted(L)
print(sorted_L)

L.sort()
print(L)
```

Summary

As a summary, I would like to emphasize the importance of considering both the length of a function, but also the number of arguments. This is especially true for methods and functions that modify the value of a function argument. Changing function arguments is delicate as it can cause significant confusion.

Return values are completely acceptable, even if some OO programmers tend to dislike them. Use return values every time the object created is not too large to cause performance issues.

Copilot

Copilot is quite proficient at generating new functions from scratch. The following code required only minimal guidance. The usage of dependency injection [section Dependency Injection] with the `condition` argument was also suggested by Copilot. I really like Copilot for this part because I often forget how to use lambdas, and sometimes it generates really nice code. The only question is whether the code is truly performing as intended.

```
books = [
    {'title': 'The Alchemist', 'author': 'Paulo Coelho', 'price': 10.2},
    {'title': 'The Prophet', 'author': 'Kahlil Gibran', 'price': 12.3},
    {'title': 'The Little Prince', 'author': 'Antoine de Saint-Exupery', 'price':
8.5}
]
```

```python
def print_books_where(condition):
    for book in books:
        if condition(book):
            print(book['title'], book['author'], book['price'])

def author_is(author):
    return lambda book: book['author'] == author

print_books_where(author_is('Paulo Coelho'))
```

14. Classes

"I think it's a new feature. Don't tell anyone it was an accident." — Larry Wall

Classes are undoubtedly one of the cornerstones of modern code and the essence of OO programming. Unless you are one of the few functional programmers, chances are high that you use them every day. Surpringly, there is very little written about classes in the common literature beyond the fact that they should have high cohesion, be small, etc. Therefore, it is time we had an in-depth discussion about them.

One can distinguish between different types of classes based on the methods and variables they utilize. We will try to understand when a function or variable should be public or private, and I will explain why plain getter and setter methods should generally be avoided.

Data Classes and Structs

The C programming language was specified well before object-oriented programming was developed. It doesn't support classes, but it has something similar: structs. It is similar to a dataclass in Python. A struct is a user-defined object that contains various types of variables. It is also possible to nest structs within each other. Structs are extremely useful because they allow us to store various data together inside a single object. It's like a toolbox. In theory, you may also store functions within a struct, though this is generally not done, at least not in C++. For storing variables and methods simultaneously, we use classes.

In Python, the equivalent of a struct is a data class. Here is an example of a dataclass:

```python
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int
    city: str

p = Person('John', 30, 'New York')
```

Nowadays, structs (or dataclasses) are not anymore as commonly used. Especially the Java community was avoiding such kinds of objects at all costs until they introduced `record` classes with Java 14 [https://docs.oracle.com/en/java/javase/17/language/records.html]. Though there is absolutely nothing wrong with structs. In fact, structs are really helpful. Code without structs is like a plumber without a toolbox. It's quite unstructured.

Classes are very similar to structs. Besides some technical details, the only real difference is the encapsulation of variables and functions (methods) by making them private, as well as the introduction of inheritance. You can decide for each variable or method whether it should be public (accessible to the outside) or private (usable only within a class). This makes classes strictly more powerful than structs.

But more powerful is not always better. A gun is more powerful than a knife, but simultaneously, it is also more dangerous. You can shoot yourself in the foot. So, actually, it might not be the best idea to own a gun because the dangers may outweigh the advantages. With great power comes great responsibility!

Private or Public

If you are not accustomed to working with classes, this may be very confusing. Why would you like to keep anything private at all? Isn't it easier to make everything public?

Indeed, this is a very important question. Once you are able to create a class and immediately decide which members should be private or public, you are already a fairly good programmer. To put it briefly, it has to do with power once again. Making a variable private reduces the control available to the user. Never give users more power than necessary.

Let's explore the reasons for having private variables and functions. We need something where you only interact with the surface and have very limited ways to engage with it. It's not hard to find an example. This description applies to almost everything around you. Once again, we can take a look at a car. A car is a highly complex object. It contains an engine, brakes, and many other parts. You don't even want to know. You only want to drive it. You need the gas pedal, the brake pedal, and the steering wheel.

You have this absolutely massive object, and essentially, you can only do three things with it: increase the speed, reduce the speed, and change the direction. And miraculously, that's all you need. As long as your car is running, you don't care about anything else. I correct myself: you don't want to know about anything else. Everything else works automatically as it should. It's like magic. You don't want to adjust the fuel pump, modify engine settings, or tamper with the servo control of the steering wheel. It works, and it's fine. You don't want to deal with the internal workings of the car. You don't even want to be able to take care of these parts. These are the private parts of the car and should not be touched by you. Only a mechanic should maintain them.

There is one very simple rule of thumb for determining which parts of a class should be public or private. If a class has no functions, it is a struct, and all variables should be public. Otherwise, as few functions as possible should be public, and all variables should be private. But we will look at this rule in more detail in the next sections.

Different Kinds of Classes

I like to categorize classes by the number of variables and the complexity of the functions they contain. Ensuring that your classes fit into one of these categories is helpful for fulfilling the SRP.

Some of the following class names, namely the worker class and the delegating class, are not commonly used. I defined them by myself because I believe it is crucial to differentiate between various types of classes based on the number of variables and functions they contain, as well as their complexity.

Note that all classes explained here, except for the data class, can also be written as functions passing additional variables around. Classes tend to grow too much and using only functions forces you to write more cohesive code. This is a typical example of where some feature (classes) makes your life easier, but at the same time allows you to write worse code. The simple question is now: when should you write a class, or replace it with a set of functions? This is a very difficult question to answer. My answer is: I don't know. I just have a preferance for functions. I guess it's a matter of taste.

The only case I know of where you need a class is when you need a constructor and destructor. For example for memory management with `new` and `delete` as done in C++ vectors.

Data Class

We have already briefly mentioned the data class (a.k.a. struct) before. It has no member functions, and therefore, all variables are public. It wouldn't make sense to have private variables if there are no functions because the variables wouldn't be accessible at all. The data class has no functionality by itself, other than a constructor. But it is great for storing data. As mentioned in the previous section, it's like a toolbox, and the variables are the tools inside. If a data class has more than about eight variables, it is advisable to split up the data class into sub-classes. This enhances the general overview in the toolbox.

```python
from dataclasses import dataclass

@dataclass
class InventoryItem:
    name: str
    unit_price: float
    quantity_on_hand: int = 0
```

Pure Method Classes

A pure method class may have no member variables at all. It consists only of static public methods. In programming languages as Java and C#, writing such classes is necessary once in a while because every function must be implemented within a class. I regard this as an OO obsession of these programming languages. In other programming languages, however, there is not much need for pure method classes. In C++ and Python, you can define the corresponding functions as free-standing functions instead. In fact, I prefer a set of free-standing functions over classes with only static public methods, but I guess that's to some degree a matter of taste.

```python
class Math:
    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def subtract(a, b):
        return a - b
```

Delegating Class

The delegating class is a combination of a data class and a pure method class. All variables are private, and all methods are public, though they are not static as they use the private variables. Calls to one of these methods are all delegated to one of the member variables. Thus, most methods are all fairly simple, usually containing only one or two lines of code. If the methods are longer, consider using some helper functions.

A delegating class is similar to a car. It consists of many complex parts, each with its own functionality. You may control all these parts using a simple interface. Setting the temperature will simply send the corresponding command to the air conditioner (AC), which then needs to handle all the complex logic. Most likely, the AC is the only part of the car that needs to know about the temperature.

Note that in this example here, it was not necessary to write a helper function. I just did so as I saw it fit.

```python
# This helper function could also be written as a method of the Car class. But
# as I've already mentioned before, I prefer free-standing functions.
def _set_temperature(AC, temperature):
    AC.turn_on()
    AC.set_temperature(temperature)
    # ...

class Car:
    def __init__(self, AC, engine):
        self._AC = AC
        self._engine = engine
        # ...

    def set_temperature(self, temperature):
        _set_temperature(self.AC, temperature)

    def set_speed(self, speed):
        self._engine.set_speed(speed)

    # ...
```

Worker Class

Worker classes implement complex algorithms in your code. Some people may argue that these are the only real classes. Most design rules for classes apply specifically to worker classes. Worker classes consist of very few private variables and no public variables. They often include some rather complicated private methods and few public methods. Worker classes are the only type of classes with private methods. Other classes do not have complicated methods to hide; they only hide variables.

This implies that worker classes are the only classes that perform complex tasks that should be hidden from other programmers. At the same time, worker classes are extremely dangerous. Excessive complexity can be easily concealed within a single worker class, making it incomprehensible to anyone. You have to ensure that your worker classes are small and well-tested. In fact, a worker class isn't that different from a function, where the function arguments correspond to the member variables. Therefore, a worker class should never have more than around three member variables and about 100 lines of code, depending on the general complexity of the class. Preferably less. An alternative to a worker class is a set of functions. Functions have to explicitly pass around the variables, which might make the code easier to understand and test, even if the code overall becomes slightly longer. This is a viable alternative if the cohesiveness is not that high. As far as cohesiveness can be measured.

As a general rule of thumb, one can say that a worker class has become too complex if you struggle to write tests for it. This is a clear indication that it's time to break up the class into smaller pieces. For more details, refer to the chapter on testing.

It is challenging to create a good example for a worker class that is not overly complicated for this book. So I tried to create a somewhat artificial one. Instead of this simple recursion, imagine a highly complex algorithm that is difficult to understand.

```python
class Worker:
    def __init__(self):
        self._data = [1, 2, 3]

    def add_entry(self, number):
        # some complicated logic
        self._data.append(number)
        self._data.sort()
        self.read_out_entry()

    def read_out_entry(self):
        entry = self._data.pop()
        print(self._data)
        self.add_entry(entry)
```

Just for completeness, this class could be rewritten using only functions as follows. We pass the `data` around as a function argument instead of using a class variable.

```python
def add_entry(number, data=[1,2,3]):
    # some complicated logic
    data.append(number)
    data.sort()
    read_out_entry(data)

def read_out_entry(data):
    entry = data.pop()
    print(data)
    add_entry(entry, data)
```

Another example for a worker class are classes implementing memory management in C++. For example the vector class or smart pointers. Though these classes are somewhat special as they rely on the constructor and destructor the take care of the memory management. Contrary to most other worker classes, I don't know how to replace such a class using only functions and data classes as the constructor and destructor are special functions of the class.

Abstract Base Class

The abstract base class (ABC in Python) has a different name in every programming language. In Java, it's called an Interface. This class type defines only the interface (shape) of a class. It does not contain an actual implementation or variables. It contains only public method declarations. Variables are a hidden detail that should not be defined in an interface. One must write classes that inherit from this abstract base class in order to implement it. In Python and other dynamically typed languages, you don't need interfaces, but they can make the code easier to understand. In C++ and Java, it is crucial to use interfaces to split up the code into smaller components, minimize compilation time, and enable runtime polymorphism. We'll go into more details in chapter [Inheritance]

In Python, a class has to inherit from `ABC` to be an abstract base class. The methods are all `abstractmethod` and they do not have any implementation. The ABC defines an interface that several programmers can implement with their own classes. This is especially useful if you collaborate with other programmers as everyone knows what they have to implement.

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def feed(self):
        pass
```

An alternative to abstract base classes in Python is protocols. If you want to understand the differences, I recommend watching the following video [https://youtu.be/EVa5Wdcgl94]. I don't have a definitive opinion on whether to use abstract base classes or protocols. And I don't think it's worth bothering about it unless you are a really experienced programmer familiar with such details.

Implementation Class

This class inherits from a pure abstract base class defined above and implements it. It contains public functions and may also include member variables. It may be anything: a worker class, a delegating class, or a pure function class. Though the pure function class is the most common. This class is implementing an interface, and due to the SRP, it shouldn't do anything else.

```python
class Sheep(Animal):
    def feed(self):
        print("Feeding a sheep with grass.")
```

The abstract base class `Animal` serves as the blueprint for the class `Sheep`. Sheep must adhere to the pattern specified in `Animal`. `Sheep` must implement all the functions defined in `Animal`. This makes it somewhat foolproof as you'll get a warning when misspelling a method name or forgetting to implement a method.

Inheritance Classes

Let's discuss the inheritance of non-abstract base classes. This becomes much trickier. As there is quite a lot that can go wrong, I advise against using it. For the sake of completeness, however, I still write down my thoughts.

Inheritance classes typically are delegating classes. They can also be worker classes, although this is not recommended due to the complexity of the resulting class. However, as I mentioned before, I generally do not recommend using inheritance, except for defining interfaces. Anything you can achieve with inheritance can also be accomplished through composition. Avoiding meddling with inheritance can potentially help you avoid a lot of trouble. I have read some books that discuss various refactoring techniques and code snippets that work perfectly well, except when using inheritance in non-abstract base classes (or global variables) [WELC]. The main problem usually arises from overriding base class functions, which can cause all kinds of issues. Furthermore, you don't really gain anything by using this type of inheritance.

Here is one of the issues with using inheritance.

```
class Animal():
    def feed(self):
        print("Feeding meat.")

class Sheep(Animal):
    def feed(self):
        print("Feeding grass.")
```

Writing such code may seem attractive at first glance. You don't have to redefine the `feed` method for animals that eat meat. It's already implemented. But saving a few lines of code does not merit code quality. Stability is. And this code is unstable. It is brittle. A single typo can create a hard-to-track bug. Let's assume you define a method `fed` inside `Sheep` instead of `feed`. Then the sheep will most likely be fed with meat, and there is no way the computer can warn you. Inheritance allows you to reuse functions, so you only need to write them once in the base class and not in the derived classes. This, however, has the drawback that you don't have to overwrite it, and a small typo changes the method that you call. This is in contrast to abstract base classes, where you are required to override the base class method, and you will receive an error if you make a typo in the method name.

In the case above, when using an abstract base class, this kind of bug is not possible. The code is a little longer, but much more stable. The following code will return an error message before executing it because `Lion` does not implement the `feed` method.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def feed(self):
        pass

class Lion(Animal):
    def fed(self):
```

```
        # oops...
        print("Feeding meat.")

if __name__ == "__main__":
    lion = Lion()
```

This error prevents you from creating a bug that might be very hard to track down, costing you a lot of time. Instead, you immediately receive a message indicating that there is something wrong with your code, `TypeError: Can't instantiate abstract class Lion with abstract method feed`.

In C++ and Java, there is the `override` keyword, whereas in Python, it is not an official language feature. But this only partially fixes the problem. You may forget to specify that a function is `override` or you may redefine a function entirely. Remember: We're humans, we make mistakes. These problems cannot occur when using abstract base classes.

General Recommendations

It may be convenient to add a method to a data class or a variable to a pure function class. However, this will quite certainly make the code worse as it violates the SRP.

The most common error is mixing up the worker class and the delegating class. You can easily end up with a fairly complex function within a delegation class that utilizes numerous member variables. This design is flawed because the delegation class operates at a high level of abstraction, while the worker class is intended to be a low-level object. Mixing different levels of abstraction is detrimental. Refactor the complex part into a separate class or function and call it from the delegating class. This should do the job.

One step further

As you have seen, I like to have my classes as small as possible. The best class is no class, with the exception being the data class. The delegating class and the worker class can also be written as a combination of data classes and functions. This may seem like a very strict approach, but it is done so in the C programming language, which is still in use for the linux kernel. If programming in this way was really an issue, they would have changed the programming language a long time ago.

The only things that requires classes because of their destructors are the C++ vectors and smart pointers. Furthermore I don't know how to implement some of the design patterns [Design Patterns book], [chapter Design Patterns] without classes.

Functions vs. Methods

There are two ways to modify the value of an exisitng object using a function. Either pass a mutable object as an output argument or use a class instance where the function is a method acting on it. The two cases look like this:

```
a.b() # method
b(a)  # function
```

There are only minor differences between these two lines of code. Let's compare them:

- In both lines, the variable `a` can be modified. Which is not necessarily a good thing, though.
- The function has only access to the public variables of `a`, while the method can also access its private variables.
- In C++, const can be applied in both cases.

In summary, there are no significant differences. The free-standing function `b(a)` offers slightly better decoupling and should therefore be preferred. But ultimately, it all comes down to readability. Which version is easier to understand? The function or the method? Let's look at the following code:

```
# Function:
if contains(names, "Donald"):
   print("Make America great again")
# Or method:
if names.contains("Barak"):
   print("Yes we can")
```

In this case, I certainly prefer the second option, using a method, due to the readability point of view. It's so much clearer. It reads like an English sentence. From a coding perspective, I prefer the first option utilizing the function. This is one of the cases where the principles explained in this book will not provide a definitive answer on how to address this problem. It can only provide arguments for one solution or the other. You will eventually have to make a judgment call on your own. If you can identify strong arguments for and against both solutions, you are a good programmer. Once you manage to make the right decision, you are a great programmer.

In software engineering, there are always numerous factors to consider. This was just another example. Probably, there are more arguments for one solution or the other that I may have missed. This might render the entire discussion obsolete.

Constructors and Destructors

Constructors and destructors are very special methods. The constructor is called once every time an object is created. This has severe consequences that many software developers are not aware of. Most notably, writing tests can become nearly impossible if the constructor contains too much logic or has side effects. The author of the code may have assumed that there would be only one class instance and planned accordingly. This might be true in his particular part of the code he wrote. However, his assumption might break down when writing unit tests [section unit tests]. When creating objects, it is important to ensure that they can exist independently without any interactions between them. Otherwise, your tests will become very fragile.

Therefore, always ensure that the usage of a constructor is foolproof. A constructor should be as simple as possible. Every action performed by a constructor, such as allocating memory or opening a file, must be reversed by the destructor. This constructor/destructor pair is the only way to guarantee that all effects of the constructor are undone. This also means that you should not define a counter in the constructor unless the effect is undone in the destructor. Preferably, your code follows the "rule of zero" [effective C++, Scott Meyers]: do not define any custom constructors or destructors. Leave this to the compiler. It will make your code much easier. [https://youtu.be/9BM5LAvNtus?t=2438] In data classes, for instance, a constructor is auto generated.

The only real functionality which has to be implemented inside a constructor and destructor that I know of, is memory management. In C++, you have to use the `new` and `delete` operators to allocate and deallocate memory and the best way to do this is inside the constructor and destructor.

The following code creates a counter for the class instances. It looks neat and it might be useful to have such a counter at times. However, it will be challenging to test this counter. Every time you add a new test case or change the order of the tests, the counter values will change. When you change the order of test execution, your tests will break.

```
from itertools import count

class Obj(object):
    _ids = count(0)

    def __init__(self):
        self.id = next(self._ids)

obj0 = Obj()
print(obj0.id) # prints 0
obj1 = Obj()
print(obj1.id) # prints 1
```

You will witness what I said here once you write a complex constructor and attempt to write unit tests for it. It is already apparent that in a random test case, you will need to define a variable `obj` with a somewhat arbitrary value `24`. This is a clear indication that something is wrong.

```
# Somewhere in your tests. This looks wrong!
assert obj.id == 24
```

Now, if you add one more class instance before this line, the counter will change, and the test will break. If you have a good intuition for code, you might have also realized that this `24` is a very strange number showing up out of the blue. There has to be something wrong with it.

Getter and Setter Methods

In Java, it is common practice to define a class and make all its variables private. Then the developer clicks a button in the IDE and public getter and setter methods are automatically generated. One for each variable. This is extremely widespread, and in my opinion, an absolutely terrible habit. There is also a point in the C++ core guidelines that supports my claim. You should "Avoid trivial getter and setter functions". [C.131 Cpp guidelines, C++ Core Guidelines explained (Rainer Grimm)]

In order to understand my claim, we have to distinguish between the different types of classes.

Data Classes

In data classes, all variables are public. Everyone can work directly with the variables. There is no need for setter or getter functions. Just access the variables directly.

The Java community may argue that this approach is unfavorable because you should decouple everything. They were decoupling the implementation of the class (the variable) from the interface (the getter and setter). Yes, they do decouple them. But they might have never really thought about the outcome. They decouple in a significantly inferior manner compared to simply providing direct access to the class variables.

Let me provide an example. We have the class `Bottle`. We look only at the private variable `_size` (the underscore indicates private variables in Python). For this discussion, we don't need more than one variable. Now first of all, it's worth mentioning that in the normal data class, `size` is a public variable, while in the version with getters and setters, it becomes a private variable `_size`. Accessing it should only be done through the getter and setter functions; that's the whole idea behind it.

```
class Bottle
   def get_size(self):
       return self._size
   def set_size(self, size):
       self._size = size
```

It is claimed that writing getters and setters has the following advantages [https://www.w3schools.com/java/java_encapsulation.asp]:

- Better control of class attributes and methods: I don't fully understand what this means. You can read and write the variables in both cases.
- Class attributes can be made read-only (if you only use the get method), or write-only (if you only use the set method): If you want to make a variable read-only, you can make it constant instead. This prevents changing the variable as well. It is uncommon to have write only variables.
- Flexibility, the programmer can change one part of the code without affecting other parts: This statement is incorrect, as demonstrated below.
- Increased security of data: I am not sure what this point exactly means. One advantage is that you can track a variable with the debugger [chapter Bugs, Errors, Exceptions]. In fact, this is the only advantage I see of writing getters and setters. But having to do this is a clear indication that your code is bad.

Let's look at decoupling. You want to rename `_size` to `_volume`. It's easy to do; there are only two places where `_size` is used. Replace them with `_volume` and you're done.

```
class Bottle
    def get_size(self):
        return self._volume
    def set_size(self, size):
        self._volume = size
```

Do you see the problem? You didn't make any improvements at all. Everyone still uses the `get_size` method, which now returns a `_volume`. This will cause a lot of confusion. You would have to rename the getter and setter functions as well. It decoupled the code only in theory. Writing getters and setters is only useful if the value returned by the getter is the result of a calculation, elevating the code to a higher level of abstraction. But then it's no longer a pure getter function... If you want to decouple your code you should write a proper adapter instead.

Having accessor methods for class variables only makes sense if they elevate the code to a higher level of abstraction. Then you really gain something. And it is really a form of decoupling. But this is not possible in data classes as there is no abstraction that could be decoupled.

Long story short: Avoid using plain getter and setter methods for data classes. They don't improve anything. Avoid using code generation tools that automatically create getters and setters for all variables.

Worker Classes

Member variables of a worker class should be private because they should not be accessed from the outside. They should not be accessed through raw setter or getter methods. Member variables in a worker class should be encapsulated within the class and accessed only by the methods of the class. These variables can be considered as intermediate results of internal calculations. They are not intended to be public. Neither directly nor via accessor methods. Hence, there is certainly no reason to write setter methods in worker classes. And there is hardly ever a reason to write a pure getter method. And no, writing a getter method for testing purposes is just bad programming practice.

Delegating Classes

The only class type that has something similar to plain getter and setter methods are the delegating classes. But also here, you should not write them. The names of these classes already contradict the concept of getters and setters. Getters and setters are accessor functions and not delegators. Accessors do not increase the level of abstraction; meanwhile, this is exactly the idea of a delegating class.

One example of a delegating class is a car, as we have already seen before. You can adjust the temperature by invoking a `set` function. But this is not a fundamental property that had been established in the car production line. It's simply an environmental parameter regulated by the air conditioning system and a temperature sensor.

There is also no reason to access the `air_conditioning` via a get method. We wanted to abstract it away. It would only make sense to define a get method to measure the current temperature, though one could argue wether get would be the right name for this method. I would rather name it `read_temperature`. Anyway, there is certainly no reason to write trivial getter or setter functions for delegating classes. A function `get_air_conditioning` would not make any sense.

```python
class Car:
    def __init__(self, air_conditioning):
        self.air_conditioning = air_conditioning

    def set_temperature(self, temperature):
        self.air_conditioning.set_temperature(temperature)

    def read_temperature(self):
        return self.air_conditioning.read_temperature()
```

I hope I managed to convince you not to write bare getter and setter methods in any kind of classes. Don't write getters and setters because someone told you so. Only write them if you really found a reason to do so. This should be the exception rather than the norm.

Coupling and Cohesion

[https://youtu.be/XQzEo1qag4A?t=159]

"Classes should have high cohesion within themselves and low coupling between each other." - Robert C. Martin

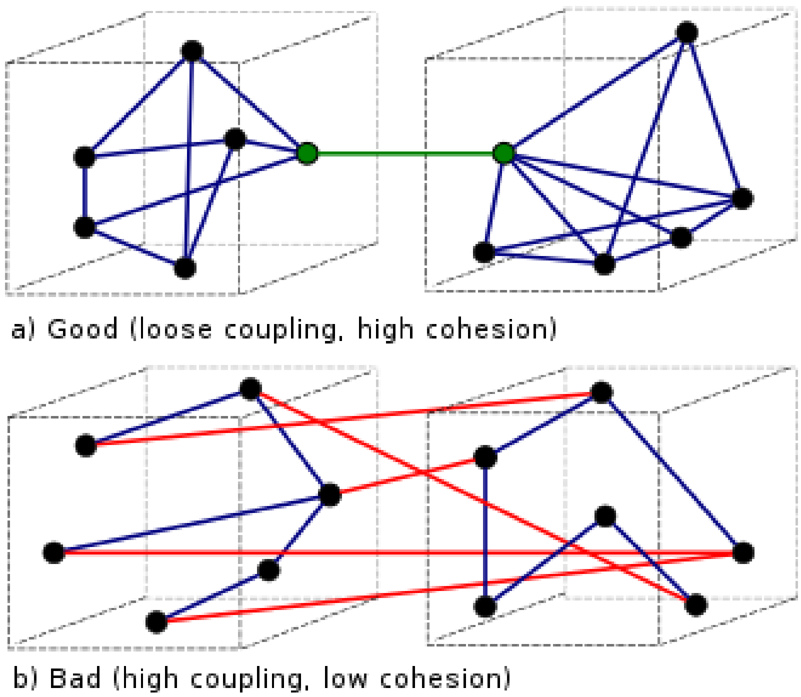
If you don't understand these expressions, we could rewrite it as follows: "There should be significant interaction among methods and variables within classes and minimal interaction between classes." This is indeed a very important rule. However, like most rules in software engineering, it has to be taken with a grain of salt.

As a very simple rule of thumb, you can search the whole class for a variable. If most methods use this variable, it has high cohesion and the variable should stay in the class. Variables that are only used by very few methods should be removed from the class and passed on as a function argument.

Worker Classes

The rule cited above by Robert C. Martin was intended for worker classes. Worker classes are a common origin of poor code because they often become overly complex. When breaking worker classes into smaller pieces, this rule is very useful. It gives you a hint on how to break them into pieces. Cluster your methods and variables into small groups. There should be a lot of interaction within the groups and little interaction between the groups. You may also need to rewrite a few methods before dividing the class into smaller parts. It will be worth the effort. If you manage to do this, it will certainly make your code easier to understand. And you have become a much better software engineer.

// get a better image without copy right



a) Good (loose coupling, high cohesion)

b) Bad (high coupling, low cohesion)

[fundamentals of software architecture p. 43, LCOM metric]

Two classes have low coupling if the number of interaction points between them is relatively low. Ideally, every class completes its work and then passes it on to the next one, similar to a relay race or functional programming. Each class would have an interface consisting of only one function. High coupling, on the other hand, is like two classes playing ping-pong. The classes all have a comprehensive interface containing numerous functions that call each other several times in a specific order. This quickly becomes terribly complex. The worst-case scenario is when two classes call each other recursively. I could hardly imagine any worse code than that! This is about the strongest coupling there is (besides inheritance). Neither of the two classes can be changed without also changing the other one. Such code is solid as a rock. You will never be able to change them again.

I hope from this description you already understand that strong coupling makes the code very difficult to understand. Additionally, it also makes it brittle, which isn't any better. It becomes increasingly difficult to make any changes without breaking it. Implementing new features will take a long time, and fixing bugs is challenging because it is not clear what each class is supposed to do exactly. Having strongly coupled code can become a nightmare.

There always has to be some amount of coupling [Decoupling]. Code cannot exist without it. It's the glue holding everything together. But the level of coupling between classes should be minimized because too much glue makes everything sticky. As long as your code works, there is never too little coupling. Furthermore, there are techniques to decouple your code. Creating an adapter between two classes, for instance, can provide more flexibility. This allows you to modify the classes independently, and you will only need to adjust the adapter when necessary.

Anyway, the rule of high cohesion and low coupling is a good rule of thumb when dealing with worker classes. If you stick to it, you can properly structure your classes. One alternative would be using some functions. Though also there, you have to pay attention that there isn't too much coupling between them.

Other class types

Maybe you have realized by now why this rule about high cohesion does not apply to all kinds of classes. A pure data class has very little cohesion. The variables are only placed into a data structure because they share some similar properties and are generally used together. Like a hammer and a screw driver both being tools stored in a tool box. Splitting a data class requires barely any effort at all. You may split it however you like. A delegating class also has very little cohesion. Nevertheless, these classes are extremely valuable as they allow you to structure your code. This rule about cohesion mostly applies to worker classes.

Inheritance

Coupling is one of the reasons why I recommend avoiding the use of inheritance. Inheritance is one of the strongest coupling available in software development. The derived class inherits all the implementations of the base class functions. Vice versa, the behavior of the base class functions may change if some function calls are overridden by the derived class. Inheritance can obfuscate the code, and removing inheritance at a later stage is extremely challenging.

Static Expression

I discourage the use of static methods. It's not terribly bad, but it's another example of these misguided object-oriented concepts. Let's first look at static methods. Isn't it strange: you write a class with all kinds of member variables, and then there is one static method that doesn't need any of these variables, yet it is still within the class? Didn't we say we wanted to keep classes small? It should have high cohesion? A static method has as little cohesion as a variable in a data class. Close to zero.

I fully understand that there are programming languages in which functions must remain within a class, and static functions are the only way to write standalone functions. In all other languages, however, I recommend avoiding the use of static methods as they do not add any additional functionality or improve the code. In C++, you can mimic a static function using a namespace. The resulting function call will be indistinguishable. At the same time, you can split a namespace over many files, as is done for the `std::` namespace, for example.

As we are discussing static functions, we can also discuss static variables as used for instance in languages like C++. Static variables are similar to singletons, and testing classes containing static variables can be challenging. Avoid using singletons and static variables. As soon as you start writing unit tests for static variables, you'll see why I discourage using them. They are like global variables and can be changed everywhere. This can easily end up in a nightmare.

Drawbacks of Classes

"You wanted a banana, but what you got was a gorilla holding the banana and the entire jungle." - Joe Armstrong

Classes are frequently misused for writing poor code without the programmers realizing it. They just think it would be normal. The most common problem is that classes become too large. It is just too convenient to write everything inside a single class. Having all the member variables readily available makes it easy to work this way. In some cases, I had the feeling that the authors of some code aimed to write all the code within a single class. This is extremely problematic. If a single class covers the entire code, then the member variables become ... global variables! [Additional Properties of Variables] Member variables are also called mini-globals in [The Art of Readable Code] for this purpose. With too many member variables the entire code turns into a Big Ball of Mud. [https://en.wikipedia.org/wiki/Big_Ball_of_Mud]

But also for slightly smaller classes, member variables can be problematic. They represent a hidden state. It is generally preferred to pass variables as function arguments to functions and methods, rather than having member variables. This makes the functions easier to test since you don't have to set up a class instance. Be careful with class variables. Or even worse, inherited variables. Keep your classes small to limit the scope of your class variables or replace classes completely with functions, if possible.

If you write a class where all method implementations consist of a single line (delegating class) or you have no methods at all (data class), the number of class variables is not too critical. These classes contain very little complexity. If you have more than about 6 to 8 member variables, you should consider organizing them into subclasses. However, as soon as you have to write complex methods, you have to be extremely careful, as things might otherwise get out of hand. The combination of complex methods and numerous member variables causes the complexity to skyrocket. When dealing with complex methods, it is recommended to keep the number of variables to one or two, as advised in the section on worker classes. Or even better, replace the class with a few functions if you can find a reasonable way to eliminate all member variables. Writing tests will become much easier.

It's a good rule of thumb to say that the class design is probably okay as long as writing unit tests works out fine and you don't feel the urge to test private functions because the class implementation is too complex. Make classes as small as possible while remaining convenient to work with.

Conclusions

I believe we can agree on the fact that OO programming is important, and it is essential for everyone to know about it. While classes have advantages, they are also a common source of bad code. Classes have a tendency to grow and become a Big Ball of Mud. Many people are simply unaware that this is an issue. As a rule of thumb, we can say that your class design is acceptable as long as writing tests is not problematic.

Follow the rule of "use composition, not inheritance". Avoid using friend classes and other complex OO constructs unless necessary in your programming language. Unless you absolutely need them for your code design. Inheritance introduces very strong coupling, which should be avoided at all costs. The same can be said for friend classes, albeit to a somewhat lesser extent.

Numerous guidelines are written only for worker classes. For example, the rule that classes should have high cohesion. It seems like most people overlooked the other types of classes. A data class lacks cohesion, but it is still a valid object.

Avoid writing plain getter and setter functions for your classes. They don't improve anything over direct access to variables.

Furthermore, you should generally prefer functions over methods. Though sometimes methods may make the code more intuitive to read and are thus preferred. As always, life as a software engineer is not easy. There are so many things to consider...

Copilot

Just as with functions, Copilot can also generate classes from scratch. It is generally recommended to write the name and perhaps some properties of the class and let Copilot take care of the rest. In this example, I only wrote `class Car`, and Copilot wrote the rest.

```
class Car():
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def get_descriptive_name(self):
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()
```

The only drawback is that Copilot suggests the function `get_descriptive_name` instead of defining the Pythonic `__str__` method. Furthermore, if you already know what members and methods a class should have, you are probably faster writing it yourself instead of asking Copilot to do it. Copilot is only beneficial if you need ideas on how to structure your class.

15. Inheritance

"Inheritance is the base class of evil" [https://youtu.be/bIhUE5uUFOA]

Why inheritance should not be used: [https://youtu.be/da_Rvn0au-g]

Inheritance is considered to be one of the integral parts of OO programming and certainly one of the most widely used. Inheritance is often described as an "is a" relationship. A sheep is an animal. Therefore, the sheep class has to inherit from the animal class. But as always, there is more to it.

Two Types of Inheritance

There are two types of inheritance: implementation inheritance and interface inheritance. Interface inheritance is used to define and implement interfaces. In C++, these base classes consist of only pure virtual functions that will be implemented in the derived classes. This type of inheritance is perfectly acceptable. Actually, it is needed for many different purposes, such as runtime polymorphism.

In Python the same behavior can be implementend using abstract base classes. Even though you don't need inheritance in Python for polymorphism. It is sufficient that two classes implement the same interface and then you can exchange them.

```python
import abc
# abc stands for Abstract Base Class, a Python thing

class Base(abc.ABC):
    @abc.abstractmethod
    def print_a(self):
        pass

class Derived(Base):
    def print_a(self):
        print("derived")
```

There is not much more to say about interface inheritance. It is a good thing, and you should use it whenever more than one class will implement this interface. Interface inheritance is a method to define interfaces and implement them, though in python this is not necessary. If you don't like it, you may simply ignore it.

Implementation inheritance inherits the implementation of the base class. Here, all kinds of different problems may occur that we'll look at in this section.

```python
class Base:
    def print_a(self):
        print("base a")

    def print_b(self):
        print("base b")

class Derived(Base):
    def print_a(self):
        print("derived a")
```

Drawbacks of Inheritance

Implementation inheritance comes with several issues and should be avoided. In the C++ Core Guidelines, there are at least a dozen points to consider when working with implementation inheritance [C++ Core Guidelines explained]. More modern languages like Go and Rust don't even support implementation inheritance. [https://golangbot.com/inheritance/]

Tight Coupling

The most obvious problem with implementation inheritance is that we may create very long inheritance chains. I once read an article about a piece of code that had 10 levels of inheritance. It turned out to be absolutely disastrous. There is hardly any stronger coupling between code than in inheritance. It was impossible to apply any changes nor to remove all the inheritance. The inheritance structure resembled a tree, with its roots entangling all the surrounding code. The code lost all its fluffiness and became solid as a rock.

I consider the widespread use of implementation inheritance as an outdated dogma. It is your responsibility to write code that is easy to understand. Don't let yourself get bothered by someone saying that a `sheep` is an `animal` and you should, therefore, use inheritance. It will almost certainly not improve the code, so you can conclude the discussion. You are probably developing a model of a `sheep` that doesn't need to know about `animals`. You have to be pragmatic. If a `sheep` does not need to be aware of the `Animal` class, there is no justification for it to inherit from it.

Inheritance is Error-Prone

There are several other issues with inheritance. This is already evident from Michael Feathers' book "Working Effectively with Legacy Code", where he provides numerous examples that he aimed to refactor. In about half of the cases, there were issues with inheritance or global variables because these things can come out of nowhere. It's just too easy to create bugs with inheritance. One misspelled function will not override the base class function as intended, potentially creating a bug. Even if you delete a function from a derived class, the code will still compile because of the presence of the base class function. Meanwhile, without inheritance, you would get a compiler error for pretty much any kind of typo.

Though it has to be said that with the `override` keyword or attribute, this problem has been resolved in some programming languages like C++ and Java. Still, I would recommend avoiding the use of inheritance and always using `override` when necessary to prevent nasty bugs.

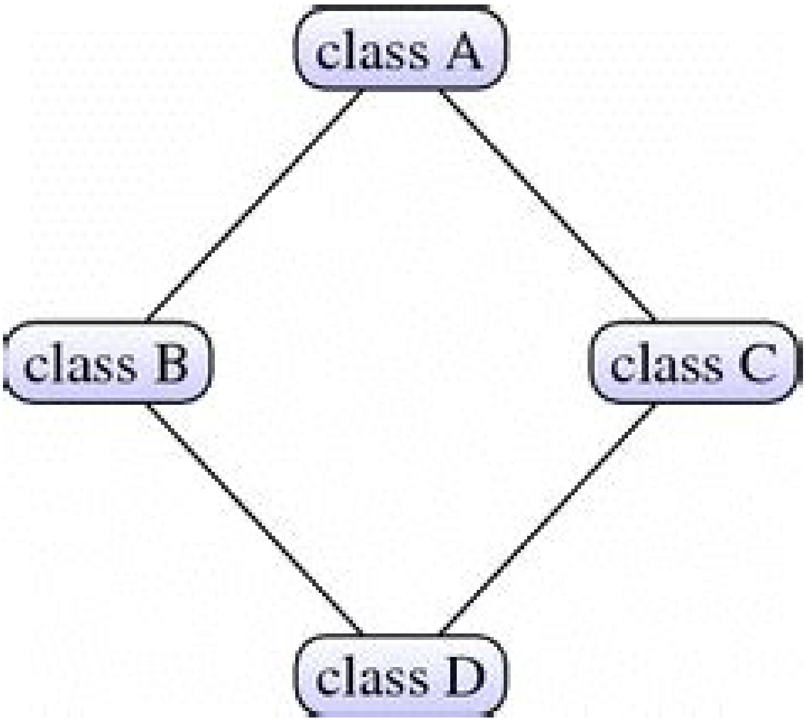
Obscure code

Additionally, there is a problem with variables inherited from the base class. These are nearly as detrimental as global variables. One doesn't know where they come from. Imagine a variable obtained from 10 levels of inheritance. And there are dozens of methods that can modify them. This is absolutely terrifying. With composition, on the other hand, you would have to dig your way through all the variables. This seems like a drawback at first sight, but it turns out to be a distinct advantage as you always know exactly where you are in the instance chain. For this reason, it is generally not recommended to nest inheritance, and I recommend using composition instead. And honestly, I don't see why inheritance should be used at all, except for defining interfaces. Code reuse can be better implemented using composition or simple functions.

Implementation

The implementation of inheritance can be a complex task, especially for some of the early OO programming languages like C++. In the early days, compilers struggled to handle many tasks. The danger was very high that a programmer created very subtle bugs. Even today, it is still challenging to use inheritance correctly in some programming languages. Implementing inheritance in C++ requires a considerable amount of knowledge and care to prevent bugs. It is fragile. Avoid fragile code.

// get a better image without copyright restrictions.



When using multiple inheritance, there is an additional issue known as the diamond problem. Let's say we have a base class called `A`. `B` and `C` inherit from `A`. So far so good. Now there is a class `D` inheriting from classes `B` and `C`. Classes `A`, `B`, and `C` all have a function `f` implemented. Which function `f` should `D` use? The one from `B` or the one from `C`?

This leads to all kinds of nasty ambiguities regarding which functions should be used. For this reason, some languages, like Java, do not support multiple inheritance. And while I consider single inheritance to be a bad practice, multiple inheritance should definitely be avoided.

Overriden Baseclass Functions

Implementing inheritance properly can be challenging. Especially when dealing with constructors and overridden functions, there is quite a bit you have to know about v-tables and other technical aspects. The chances of making errors are significant. This issue can be avoided by refraining from using inheritance. Inheritance is simply too error-prone. Though this is better in Python than in C++.

Sometimes inheritance can be confusing. Let's consider the following example:

```python
class Animal():
    def feed(self):
        print(f"eating {self.get_food()}")

    def get_food(self):
        return "grass"

class Lion(Animal):
```

```
    def get_food(self):
        return "meat"

if __name__ == "__main__":
    lion = Lion()
    lion.feed()
```

Is the lion now eating grass or meat? Of course, it's eating meat. Using overridden functions in the base class can quickly become confusing. This is the simplest version of the Yo-yo problem, [https://en.wikipedia.org/wiki/Yo-yo_problem] where the programmer has to switch between reading the code of the base class and the derived class in order to understand the code. The derived class not only depends on the base class, it's also the other way around. By breaking the encapsulation of the base class, we introduce a mutual dependency. This is so confusing; it is dreadful. Please refrain from writing such code.

Of course, this can be avoided by using the final keyword in some programming languages. But it is just another example of why, in my opinion, inheritance should be avoided. It's just another fix for something that is inherently wrong. As I mentioned, in my opinion, there is simply too much that can go wrong with inheritance. Inheritance from a class should only be possible if it is explicitly allowed.

In inheritance, the derived class inherits all the functions defined in the base class. This might be more than what is actually required. The interface of the derived class is larger than necessary. This violates the Interface Segregation Principle [chapter SOLID Principles]. Having to write tests for unused functions in the interface is only the most obvious problem.

Advantages of Inheritance

There are quite little advantages, but none of them justify using implementation inheritance. The only thing that I can think of is code reuse. But it is not worth the drawbacks that come along implementation inheritance, as mentioned above.

The only real use case of inheritance, in my opinion, is the definition of interfaces using interface inheritance.

Inheritance and Composition

[https://www.studysmarter.co.uk/explanations/computer-science/computer-programming/inheritance-in-oops/]

To conclude this chapter, let me provide a brief example to illustrate the distinctions between inheritance and composition. In the class `Lion`, the `lion` can directly access the `food` object from the base class.

```
class Animal():
    def __init__(self, food):
        self.food = food

class Lion(Animal):
    def __init__(self):
        super().__init__(food="meat")
```

```
lion = Lion()
print(lion.food)
```

In the class `Car`, the `taxi` has to access the `power` object through the `engine` object.

```
class Engine():
    def __init__(self, power):
        self.power = power

class Car():
    def __init__(self):
        self.engine = Engine(power=322)

taxi = Car()
print(taxi.engine.power)
```

Now, there may be many programmers who prefer the code used for the `lion`, for example, because it is shorter. But in my opinion, this is very bad. The `lion` code is implicit. And implicit code should generally be avoided because it is not as clear as explicit code. As the Zen of Python states: "Explicit is better than implicit." The code used with the `taxi` is clearly preferred as it is explicit. Write a little bit (one word!) more code inside the print statement, but the clarity really makes up for it. The code using the `taxi` is much clearer because it indicates the origin of the `power` variable. It is a variable within the `engine`. This is the primary reason why I recommend using composition instead of inheritance. It makes the code much clearer. It is explicit.

If `power` is used frequently, you can also write a method for it,

```
    def power(self):
        return self.engine.power
```

Conclusions

You don't gain much by using inheritance. Using composition is, in most cases, a perfectly viable alternative. If your code looks messy when you start using composition instead of inheritance, you probably wrote messy code all along. You just didn't see it because the inheritance was hiding it. This is another negative aspect. Composition generally makes the code more readable and easier to understand. It is also less error-prone. And it is much easier to test. Implementation inheritance is a common source of poor code quality. It should generally be avoided.

There are also some more esoteric concepts, such as friend classes. At first sight, friend classes seem like a good idea because they make writing code easier. However, in the long term, this has similar issues to making private variables public. In most cases, it results in poorly written code that lacks proper encapsulation. Just ignore friend classes and similar concepts and never look back. There are very few cases where friend classes are truly beneficial. [https://google.github.io/styleguide/cppguide.html#Friends]. Write your code using the most common language features, and only consider using fancy language features if they genuinely enhance your code.

16. Data Types

"Primitive obsession [https://refactoring.guru/smells/primitive-obsession] is a code smell in which primitive data is used excessively to represent data models." - David Sackstein

There are hundreds of built-in data types. But using too many primitive data types is also known as "Primitive Obsession". Avoid excessive use of built-in data types. Instead, you should use custom types (classes) as much as possible. This makes the code more readable and easier to write.

Using custom types (classes) is highly recommended. For example, you should always use a class `Money` when appropriate and avoid using floating-point numbers. Utilizing custom types enhances the readability and simplifies the writing process of the code. It prevents you from primitive obsessions.

Primitive obsession is a very common phenomenon. Integer values are often used to represent time, even though there is typically a dedicated time class in most programming languages. Strings are used to store all kinds of information, as we will see in an example below.

Here is a list of data types that I typically use. They are called differently in most languages. I write the Python name and in brackets the C++ name: floats, ints, lists (vectors), enums, Booleans, strings, dicts (maps), trees, classes, (pointers).

I will provide explanations for all these types except floats, integers, and classes. I don't have much to say about floats and ints, except that I typically avoid using unsigned ints, as advised by the Google Style Guide. Classes are discussed in their own section due to their significance.

Lists

Lists are the workhorse in programming. Whenever you deal with several values that should all be treated in the same way, they belong in a list. I would like to emphasize the importance of being treated equally. When working with a list, it is important to iterate through all elements and perform the same operation on each of them. If you only need one value from a list, it is likely that you should not use a list.

Here is an example of how not to do it:

```
fruits = [‘apple’, 1.5, 3.1, ‘banana’, 0.8, 2.1]
print(fruits[4]) # ...?
```

I intentionally made this code so terrible for you to understand. Strings and numbers cannot be equal objects, so they should not be placed side by side in the same list. In C++, this kind of list isn't even possible because C++ vectors cannot contain objects of different types. At least not without attending a highly advanced course in C++ black magic. In Python, on the other hand, this code is syntactically correct, and it is often tempting to write such a list. Please resist this temptation!

Your code becomes really bad if you store different things in the same list. The `print(fruits[4])` in the code example above is extremely error prone because you have to count the elements in the list. This list does several things at once and it violates the single responsibility principle (SRP) all by itself. Use data classes instead [clases].

Apparently, three values inside this list always belong together. A first improvement would be to use a list of lists,

```
fruits = [[‘apple’, 1.5, 3.1], [‘banana’, 0.8, 2.1]]
```

This provides some structure to the list, making it less likely that this data structure will be used incorrectly. This inner list is still far from optimal. We should use a data class instead.

The code should be rewritten as follows:

```
@dataclass
class ShoppingItem:
    name: str
    weight: float
    price: float

apples = ShoppingItem(name='apple', weight=1.5, price=3.1)
bananas = ShoppingItem(name='banana', weight=0.8, price=2.1)

shopping_list = [apples, bananas]
```

Now the code is much longer, but it is also much better. It is much easier to read and understand. As we learned, the only quality measures of code. All the elements inside the list are equal. All of them are `ShoppingItems`. If you can do something, you should iterate over all elements and treat them equally. The data structure is now also pretty save. Correlated data is all stored together. It is almost impossible to confuse the weight of the apple and the banana. And it's also pretty hard now to make an error when creating the list.

We can summarize: Lists are very common. They should always contain objects of equal meaning. If you want to create a list with groups of objects, you should create a data class for these groups and make a list of instances of these classes. If you only need to access a single object from a list, it is likely that your code is bad. Always iterate over the entire list and treat all elements equally.

Enums

Enums are something that even many experienced software developers don't know about. You don't really need it. But they should know that enums make your code much better. There are several alternative methods to write code without utilizing enums. They are all bad.

```
# 1. boolean:
is_blue = True

# 2. string:
favorite_color = "blue"

# 3. integer:
favorite_color = 7
```

```python
# 4. class instance:
class Blue:
    pass
favorite_color = Blue()

# 5. enum:
from enum import Enum
class Color(Enum):
    BLUE = 1
favorite_color = Color.BLUE
```

The first four options all have some severe drawbacks.

Booleans

The first one is extremely ugly. What does `is_blue = False` mean? Is it red? Invisible? Undefined? There are simply too many different options that can confuse the developer. Avoid using booleans in general.

Strings

The second one looks reasonable at first sight. Just write `"red"` and you have another color. But at the same time, it's easy to introduce bugs. If you write `"blu"` instead of `"blue"` you might introduce a bug that could result in strange behavior. Without you noticing either that you have a bug or where the error comes from. The compiler won't be able to help you with this bug. Avoid using string comparisons as they are prone to errors.

Ints

Third option: 7? A color? No. Please, don't do this to me. This is an example of a magic number and should be avoided. You would always have to look up the number corresponding to one color. This would be very tedious and error prone. Unless it is a well-known international color standard, for example RGB: `blue = RGB(0,0,255)`.

Classes

Fourth option: Using types is not the best choice in this case. It can be verified using `isinstance(blue, Blue)`, but this process is laborious and not feasible in C++, for instance. Using classes in this case does not offer any advantages, only drawbacks.

Enums

Fifth option: The best solution is certainly using an enum. Even if it takes getting used to it. Enums may seem slightly unusual at first glance due to the `Color::` prefix, and there is no way to alter this. However, this code is really solid and foolproof. If you write `Color::BLU`, you will get an error message because you most likely did not define a color `BLU` inside the enum. This is infinitely better than having a bug [Bugs, Errors, Exceptions]. Furthermore, most Integrated Development Environments (IDEs) and programming languages support auto-completion for enums. Gone are the times when you had to look up magic values in the manual. Enums are great. Use them wherever you define a selection from a limited number of options.

Enums can only be used if you know all possible options when writing the code. If the user can define custom options, string comparison must be used. Though cases where you really have to make string comparisons are rare. It is rare to encounter a situation where you receive a random string and then invoke a function based on its content. The only thing you usually have to do with random strings is pass them on without altering them.

Booleans

"Have a seat, my son. There is something very important that I have to tell you. If you hear it for the first time, it may be very shocking. But it has to be said: Booleans are evil."

"What? But... how...? This can't be. Booleans are only a theoretical construct. It's everywhere. The entire binary system consists of Boolean values. What do you mean?"

"Yes, of course, you are right. Let me explain. It's somewhat similar to alcohol. Alcohol does not do any harm if it is inside a bottle. You can drink it and have a great time, maybe the best time of your life. But at the same time, it can cause a car accident or start a pub brawl. Humans can't handle alcohol. This is why some people say that alcohol is evil. There is a very similar issue with Booleans. Booleans can be used for great things. But at the same time, using Booleans can lead to the creation of bugs. Humans struggle with Booleans. They mix it up too often. And even worse than Booleans lead to if statements. But okay, maybe we should not call them evil, but dangerous."

Of course, I may be exaggerating slightly. But there's no denying: Humans struggle with Booleans and if statements. They lead to bugs. You will create bugs with if statements if you keep using them. Here are some points to consider when working with if statements:

- Good code design results in fewer if statements.
- Polymorphism can be utilized to avoid using if statements.
- Resolve `if` statements as early as possible. Use Dependency Injection (DI) instead of booleans.
- Avoid nesting if statements. Excessive levels of indentation are a sign of poor code quality.
- Avoid passing Booleans as function arguments.
- Consider using enums instead of booleans.
- Ensure that your unit tests cover all branches of if-else statements.
- Avoid using traditional C++ or Java iterators. Looping over iterators requires comparisons. Range-based loops are much safer and easier to use.

Here is an example how to use DI instead of passing on booleans:

```
if __name__ == "__main__":
    if "debug" in sys.argv:
        reader = DebugReader()
    else:
        reader = Reader()
    main(reader)
```

Well yes, we didn't get rid of all `if` statements. This will never be possible. But further down in the code, there are no more booleans. Instead you only have the `reader` object that you can use polymorphically.

Match case statements

// This section really needs some consideration: When should a switch/match be used?

In case you have a `match case` statement (a Pythonic expression, in other languages called a `switch` statement), you should encapsulate it inside a dedicated function or use a dictionary.

This is not how the code should look:

```
def do_a_lot_of_work():
    # a lot of code here
    # ...
    city_name = "Zurich"
    match city_name:
        case "Zurich":
            return 8000
        case "Bern":
            return 3000
```

This code is flawed for a very simple reason: it almost certainly violates the SRP. The likelihood is high that this `match case` statement will be repeated multiple times in your codebase. Instead, the `match case` statement should be refactored into its own function.

```
def post_code(city_name):
    match city_name:
        case "Zurich":
            return 8000
        case "Bern":
            return 3000

print(post_code("Zurich"))
```

The best solution, in my opinion, is using a dictionary and abandoning `match case` statements altogether. This is shorter and easier to read. If desired, you can still wrap the dictionary in a function.

```
post_codes = {
    "Zurich": 8000,
    "Bern": 3000,
}
print(post_codes["Zurich"])
```

For larger dictionaries, this may still appear quite verbose. But this code will be hidden at a low level of abstraction.

Dictionaries can also be used polymorphically. Depending on the key, it creates an object of a different type. This will prevent some `if` statements in the future, as polymorphism generally does.

```
class Zurich:
    def postcode(self):
         return 8000

class Bern:
    def postcode(self):
        return 3000

cities = {
    "Zurich": Zurich(),
    "Bern": Bern(),
}

zurich = cities["Zurich"]
print(zurich.postcode())
```

A little side remark: `match case` statements were only introduced with Python 3.10. This is because they are not supposed to simply replace the switch-case statements, as seen in C++ for example or as shown in the examples here. [For the full story, please visit https://youtu.be/ASRqxDGutpA]

In summary, one can say that `match case` statements are not bad at all. Though they could easily be replaced by dictionaries, and they should be wrapped inside a function to make them reusable and adhere to the SRP. Additionally, they are a great match with polymorphism in the creation of objects to prevent further `if` statements.

For Loops

A long time ago, for loops required the use of booleans. Fortunately, these times are long gone. I add this topic here as an example how to get rid of boolean comparisons.

There are many ways how to implement for loops. And if you use them, there are better and worse ways how to use them. Let's start with the classical C++ version:

```
#include <vector>
#include <iostream>

std::vector<int> numbers = {1, 2, 3, 4, 5};
for (int i = 0; i < numbers.size(); i++) {
    std::cout << numbers[i] << std::endl;
}
```

Note that I had to use C++ for this example. I wouldn't know how to write such a terrible for loop in Python. There is so much that can go wrong here. You have the `<` sign which does a comparison. And comparisons are dangerous because you can always mix up `<` and `<=`.

One level better is this Python code:

```
numbers = [1, 2, 3, 4, 5]
for i in range(len(numbers)):
    print(numbers[i])
```

Here we got rid of the comparison. But we still have the index `i` which can be mixed up. Of course, there are times where you need this index. But generally you can use an enumeration instead.

```
for i, number in enumerate(numbers):
    print(i, number)
```

Assuming that you don't need the index `i`, by far the best code is the following:

```
for number in numbers:
    print(number)
```

Here you don't need any comparison nor an index.

Filtering Lists

Another question: how should you filter lists?. The classical way is to use a for loop and an if statement. But there are two alternatives: list comprehensions and lambdas. Both solutions are quite similar (even in performance), so feel free to use the one you prefer.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odds = [n for n in numbers if n % 2 == 1]
ten_times = [n*10 for n in odds]
plus_five = [n+5 for n in ten_times]
average = sum(plus_five) / len(plus_five)
print(average)
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
plus_five = map(lambda n: n+5,
            map(lambda n: n*10,
            filter(lambda n: n % 2 == 1, numbers)))
five_list = list(plus_five)
average = sum(five_list)/len(list(five_list))
print(average)
```

Note that the filtering part of both versions could be written in a single line if you insist. And I'm not showing the loop version as it's too much code.

Strings

"You should never use two different languages in a single file. English is also a language." - Adapted from Robert C. Martin [Clean Code]

After pointers and Booleans, strings are arguably the third most error-prone data type. Programmers often compare two strings for equality. One of them is written in plain text in the code. A string possibly twenty characters long. If a single character is wrong, you have a bug, and there is no way the computer is able to know and warn you. Of course, you can make this code work. But it is extremely brittle. You should eliminate such risks whenever possible. String comparison is a potential source of errors, and we should strive to avoid them whenever feasible. Remember, programming is all about avoiding potential sources of errors. As we have already seen, you should always consider using enums if you want to perform string comparisons.

Stringly typed objects

Some people even start to encode all kinds of logic into strings. This is dreadful. At times, this is also referred to as "stringly typed" to emphasize the importance of using appropriate types instead of strings. // See also "primitive obsession"

Here are some examples of strings storing all kind of information that shouldn't be stored as strings [https://www.hanselman.com/blog/stringly-typed-vs-strongly-typed]:

```
robot.move("1","2")
# Should be int like 1 and 2, and maybe better a point

getattr(dog, "bark")
# Dispatching a method passing in a string that is the method's name. dog.Bark()

message.push("transaction_completed")
# Could be an enum
```

I found the following example in the book "Clean Code" on page 128, where Robert C. Martin (a.k.a. Uncle Bob) did some refactoring on a unit test. I quite like the book. It served as a model for this book here. But in this example, he somehow went haywire. What he explained all made sense, but he missed that one should never write code the way he did.

He encoded five Boolean states `{heater_state, blower_state, cooler_state, hi_temp_alarm, low_temp_alarm}` into a single string `"hbCHl"`, where each character encodes whether it was too hot or not, too cold or not, etc. Capital letters represent `true`, while lowercase letters represent `false`. It's such a beautiful example of the kind of logic that can be implemented in strings. But he completely missed the point. Strings should never ever be used to encode logic. To make matters worse, the letter `"h"` is even used twice. Like this, the code becomes even more fragile because the state relies on the order of the characters.

The unit tests written by Uncle Bob look quite nice at first glance. But it takes some knowledge to understand what these five characters are supposed to mean. Without appropriate background knowledge, it is impossible to understand the meaning of this string. The order of the characters within this string may seem arbitrary, but they must be in the correct sequence. This is not something that should show up in a unit test nor in your code.

Now let's consider how we could improve things. We have five states that can each be either true or false. Writing a list with 5 Booleans is probably the first thought, something like `water_state = [False, False, True, True, False]`. This is an improvement over the string logic, but it still requires significant restructuring. All elements in a list should be treated equally and accessed simultaneously. But here, you will probably need only one element at a time: `needs_hot_water != water_state[0]`. Accessing the first element with `[0]` is a clear indication that we should not use a list [section lists].

A better solution is to use a dataclass that stores five different variables. One Boolean value replacing each character in the string above.

```
from dataclasses import dataclass

@dataclass
class WaterState{
    heater_state: bool
    blower_state: bool
    cooler_state: bool
    high_temp_alert: bool
    low_temp_alert: bool
}
```

Still, this is not optimal yet. What does `heater_state = true` or `= false` mean? Let's define an enum instead to make the code more readable.

```
from enum import Enum
from dataclasses import dataclass

class State(Enum):
    on = True
    off = False

@dataclass
class WaterState:
    heater_state: State
    blower_state: State
    cooler_state: State
    high_temp_alert: State
    low_temp_alert: State
```

Now the `heater_state` can be either `on` or `off`. This is much more intuitive to read.

Once you found this solution, it looks so natural. This code is much more readable than the encoded string. It is definitely worth the extra effort required to write this struct and enum. The code has now become significantly longer, but remember: we always code for readability, not for the fewest lines of code.

The code utilizing this dataclass, including the unit tests, is very straightforward. Opposite of the string solution, there is no need for logic, comparison, or anything similar. It is simply obvious how to use it.

```
if water_state.high_temp_alert == State.on:
    print("Attention: the water is too hot")
```

Natural Language

Many software products are available in many different countries. They have to be available in many languages. But you don't want the translator to write his translations into your code, nor does the translator want to deal with your code. He wants to work with only the text visible to the user. He wants the text to be placed in a dedicated text file so that he knows exactly what to translate. There is no arguing with that. Thus, it is your job to extract all the human-readable text from your code. Upon start-up, your software reads this text file and assigns the various strings to the corresponding variables. Selecting a different language is as easy as selecting a different file.

Ultimately, you are left with barely any strings at all. You replaced them with enums, proper logic, and a file containing human-readable text. Only when reading or writing a text file do you briefly have to deal with strings. Then you immediately convert it into data. In theory, at least. For small projects, it is not always worth the effort to convert all strings into objects or dedicated text files.

Dictionaries

When defining your variable, you have two different choices on how to proceed. You may either use normal variables or a dictionary (a map in C++).

```
a = 0
b = 1
```

```
vars = {"a" : 0, "b" : 1}
```

These lines do something very similar. They both assign the value `0` to `a` and the value `1` to `b` (okay, in the case of the dictionary, it is rather `"a"`, but you get my point). Yet, there is a fundamental difference. In the first line, the programmer knows that he needs variables `a` and `b` as he writes the code. In the second case, we have a dynamic data structure. Maybe the programmer knew that there would be `"a"` and `"b"` used as keys. Maybe he didn't, and these dictionary entries were generated by user input that the programmer had no control over.

If the developer knows all the variables that are needed, it is generally advisable to use normal variables. If the data originates from an external source, such as a text file where he doesn't the content, he must use a dynamic data structure like a dictionary. At first, this may sound a little confusing. But think about cooking recipes. You might have a few recipes that you define in your code, where the name of the recipe corresponds to the name of the variable. Or, you can write a parser that reads them from a cookbook into a dictionary. Here you have to use a dynamic data structure because you don't know in advance how the dishes are called and what kind of ingredients will be needed.

Dictionaries are closely related to JSON and XML files. They are essentially similar to a nested dictionary serialized into a string. If you ever need to read JSON files, the resulting data structure will be a nested dictionary that you might further convert into nested class instances.

Trees

It is not too often that I've had to create a tree myself, yet I have been working on tree structures for a significant part of my programming career. Trees are an extremely important data structure. When dealing with a recursive data structure, it is highly likely that you will be working with a tree. This allows you to utilize many standard algorithms that are very efficient, typically with a time complexity of `O(N log(N))`. If you implement your own algorithms, ensure that they are recursive and write automated tests that cover the corner cases.

Pointers

Python, as most other modern programming languages, doesn't use pointers. C++ on the other hand used pointers extensively. Pointers were used to point to a specific location in your memory and access the corresponding value. However, pointers are still used to implement polymorphism. Pointers are arguably the most powerful yet potentially risky objects in the programming world. With pointers, pretty much anything can go wrong. Fortunately, they are barely needed these days. Vectors and smart pointers have implemented in C++ for and completely replaced pointers. Vectors, for example, also use a pointer, but it is hidden deep inside their implementation.

The only remnant where pointers are still needed for technical reasons is interfaces. Use pointers only for interfaces and opt for modern smart pointers (unique pointer or shared pointer) and you will be fine.

17. Properties of Variables

Once again, things only got started with the introduction to the data types. The hard part is not choosing a data type, but figuring out how to deal with them. How to facilitate interaction between them. Here, one can easily create a huge mess if things are not considered properly. Even experienced programmers do not always know how to structure them properly. It is challenging. And I'm trying to explain to you at least some very fundamental ideas to look out for.

The most common way to structure data is by using nested classes and lists, where one class contains instances of other classes. There's certainly nothing wrong with that, but sometimes there are better solutions.

Variables do not only have a type, but they can also have additional properties that we will explore in this chapter. They can be compile-time constant, constant, mutable, member, static, dynamic, global or several of them at once. All these various types of variables have distinct scopes within which they can be accessed and modified. As is often the case in programming, it is very convenient to have access to a variable at all times, similar to a global variable. At the same time, this approach is very likely to result in poor code quality as this variable is tightly coupling everything together. Therefore, you should always choose a variable type that is just modifiable enough to work with but doesn't grant more accessibility than necessary.

Compile-time constant

Compile-time constants are the least powerful variable type. They are known at the time you write the code and will never change their value. In Python, there is no way to enforce const'ness. But it is generally agreed upon that variables written in all uppercase are constant and may not be changed, `PI=3.14`. In C++, there is the `const` keyword that enforces const'ness of a variable. `const double pi=3.14`. Now it is no longer possible to change the variable `pi`, or the compiler will return an error. Keep these constants stored separately and avoid cluttering your code. Otherwise, there is nothing you can do wrong with them.

Runtime Constant

Compared to compile-time constants, runtime constants do not know their values at the time of compilation. The values will be assigned at runtime upon the creation of the object.

Once created, you can pass and copy them around as much as you please. You are always guaranteed to deal with the same object. You can even declare a constant global variable and avoid the main issues associated with global variables. Though it is still recommended to pass them around as function arguments instead. If it's global, it will be acting as a hidden state, making it much harder to write tests.

Note that in functional programming, all variables are runtime constant. The only way to change a variable, is to create a new one. This is a severe restriction, but at the same time it makes reading the code easier as you don't have to consider output arguments.

Mutable Variables

"Immutable types are safer from bugs, easier to understand, and more ready for change" - [https://web.mit.edu/6.005/www/fa15/classes/09-immutability/]

In many ways, mutable variables can be compared to class instances. They are both very powerful, yet at the same time, they are tricky to deal with as they may change their values. This can easily lead to bugs. On the other hand, writing code without mutable variables (or class instances) is very challenging. If you want to understand the level of difficulty, I can recommend you to try functional programming. The problem with mutable variables is, perhaps not surprisingly, their mutability. Values may change, even if they are just a function argument. This makes it so hard to keep track of their value.

One option is to work more with immutable objects. For example, you can replace the following code:

```
prime_numbers = [11, 3, 7, 5, 2]
prime_numbers.sort()
```

with something that does not change the list. Instead it can return a new list.

```
prime_numbers = [11, 3, 7, 5, 2]
sorted_prime_numbers = sorted(prime_numbers)
```

At first sight, the two options look pretty much equal. The first one changes the list instance, while the second one returns a new list. However, there is a quite distinct difference. The first one passes a mutable variable, which is error-prone. Furthermore, it reuses the variable, which is a minor violation of the SRP. [https://youtu.be/I8UvQKvOSSw?t=2133]

Returning a new variable, as demonstrated in the second code snippet, is a much safer option and is preferred. Furthermore, the second version of the code is much clearer because the variable is not reused. The different names create a clear distinction between the unsorted and the sorted list.

On the other hand, the second solution may create a performance bottleneck as it requires more memory if the initial value does not go out of scope. This could pose a problem for large lists, particularly within loops. Though this is usually not a significant issue because the `prime_numbers` go out of scope, and the memory will be recycled. Furthermore the `sort` function anyway requires some memory.

Member Variables

Being a member variable is by far the most common property of a variable. Yet, there is a lot that can go wrong as well, as member variables are mutable variables simultaneously. Most of the information you need to know is explained in the section on classes [Classes]. As long as your class design is appropriate (classes should be small!), the methods are well-designed (with no unexpected side effects, as far as this is possible in a class...), you are mostly fine with using member variables. Though you have to be careful with them.

Member variables have essentially the same issue as global variables, but within a more restricted scope. They are a hidden state. This is one reason why classes have to be small in order to limit the extent of this hidden state. If the class becomes too big, the member variables are very similar to global variables as they can be accessed from almost anywhere in the code.

Passing output arguments to functions can make the code less clear. The best solution would be passing around only immutable variables. However, it would also be too difficult to code in this manner. Functional programming works this way, but it is not too wide spread, even though it exists longer than OO programming. OO seems to strike a balance between the accessibility and privacy of variables and functions. But you always have to be aware of this and make sure you maintain the balance so that it doesn't tip over to the accessibility side. Keep your classes small and make everything private that can be. When in doubt, use immutable objects.

Static Variables

Static variables are member variables that share the same value across all class instances. Let's briefly figure out when to use them.

If a static variable is const, one could also create a const variable outside the class instead. Except if this is not allowed, as in Java, for instance.

Having const static variables doesn't make much sense as they can also be stored outside of a class.

If a static variable is not constant, it is likely intended to modify the value of the variable in all class instances simultaneously. This is a side effect. This is dark magic! This is dreadful!! Never use dark magic. Avoid using static variables.

And if you don't believe me, try writing unit tests for a class that contains static variables. You won't be able to change the order of the tests because it might cause them to fail. This is the very definition of brittle.

Global Variables

You might have heard about global variables. That they are bad, and you should never use them. This is indeed true. Let me provide an everyday example to illustrate why this is the case.

Let's say you have to give a bag to a friend. But you are not able to meet. Now, your solution is to place it in the middle of a very busy square, and he can pick it up later on. Are you now thinking...? No! NO! Don't even think about it! There is NO WAY this is ever going to work. Everyone around can compromise the integrity of the bag. And they will. Believe me, they certainly will. This is the problem with global variables. Millions have tried this attempt before you, and millions have failed. No one has found a solution on how to safely work with global variables. Do NEVER use global variables. If you believe that using a global variable is the only solution to your problems, you should seek assistance to review your code and address some fundamental issues. Relying on global variables will only exacerbate the situation.

Of course, it's slightly different if the bag weighs 1000 tons and no one can move it. Not even Superman. This is not a variable anymore. This is a constant. You define it once, and it will never change. But even here, it is considered bad practice to make it global. Pass the variables around as function arguments to make the dependencies apparent.

Now, as you may have already realized, global variables are problematic because any line of code can alter their value. Everywhere. You cannot rely on them. You never know if someone compromises their integrity. This also makes the code incredibly hard to understand, as the workflow becomes extremely entangled. All of a sudden, there is temporal coupling between different function calls if they modify this variable. You have to follow every trace where the variable could be changed. This is the very definition of spaghetti code. And once again, if you don't believe me try writing tests for code containing global variables. They will break all the time.

The opposite of functions using global variables are pure functions. Pure functions are functions that depend solely on their input arguments and do not produce any side effects. You'll always know exactly what they do and can rely on. They will never change any hidden state.

Comparison of Variable Properties

The variables we examined vary in terms of how easily they can be changed. Starting with a local compile time constant that cannot be changed and accessed only locally, to a global variable that everyone can access and change. This level of accessibility must be selected carefully for every variable you work with. You can barely write code with only compile-time constants, but if you use only global variables, you'll soon end up with spaghetti code. Generally, it's best to choose variables with the least possible effects that still allow you to implement what you want. Prefer too little accessibility over too much. You can still increase it later on.

Here is a rough list of how variable types are sorted by the accessibility they have, starting with the most accessible:

Compile-Time Constant < Constant < Immutable Object < Mutable Object < Class Variable < Inherited Variable < Singleton = Global Variable

There is certainly nothing wrong with constants, especially with compile-time constants. It's just that they can't do much. They are just there and do nothing. They store a fixed value, and you are always free to read it. If you enjoy working with constant or immutable objects, I recommend functional programming. In functional programming, everything is constant.

Immutable (but non-constant) objects can only be used within the current scope. When passed as a function argument, their value cannot be changed. If you use immutable objects, you cannot have output arguments, which, in my opinion, is generally a good thing. Due to the SRP, a function should change the value of only one variable, and in my opinion, this should be the return value because it is more evident what the code does. In general, you should avoid output arguments anyway. The only difference to constant objects is that immutable objects can be reassigned to a different value.

With mutable objects, you have to be careful because it may be unexpected that a function call changes the value of an argument. Make sure your functions modify at most the value of the first argument, as altering other arguments can lead to confusion. This is not a strict law, but rather a convention. Making multiple changes through a single function call is also a violation of the SRP and should be avoided. If possible, create a new object instead of modifying an existing one. The only reason I could think of why one should use mutable objects is performance. Creating new objects all the time may be slow. Though with modern compilers this issue may be partially resolved.

Dealing with class variables can be quite tricky. There are too many ways they can disrupt the workflow and cause side effects. They may be used, of course, but I provide detailed explanations in the chapter on classes [Classes] about the considerations that need to be taken into account to avoid causing chaos. Class variables and mutable objects both allow for modifying an object. At the same time, this is also precisely why they are difficult to deal with. Furthermore, class variables are accessible in a significantly broader scope, throughout the entire class. This is fine for small classes, but one of the reasons why classes should not be too big. Otherwise, the class has too much hidden state that will confuse the reader.

Inherited variables are even worse than class variables. It is not easy to see where an inherited variable is defined. It's like receiving a couple of tools without knowing their origin or ownership. If you need to exchange a tool, you may be unsure of what to do. Compare this to a composition that provides you with an organized toolbox to work with. Inherited variables make the code more difficult to understand. And there's no apparent reason why one should use inheritance. And no, the few words saved are not a reason. The number of words used is not a measure of the quality of code. Readability is. And readability is certainly better with composition than with inheritance. This is one of the main reasons why it's better not to use inheritance at all [Inheritance].

A Singleton is a class that can have at most one instance. If you create objects of this class in several locations, they all share the same class instance. There are very few cases where singletons are truly useful. This is mostly the case for connections. It allows multiple sections of your code to utilize the same connection to your database, web server, mobile phone, etc. If you have limited communication and only a few relatively large datasets, this is not necessary. You wouldn't gain much from using the singleton pattern. Every class or library can connect to the database to retrieve data when needed and disconnect when finished. For many small database requests, using a singleton may significantly increase performance. However, singletons are commonly abused to act as a global variable. And this is really bad. For this reason, it is generally discouraged to use singletons unless you truly understand why you need one [https://github.com/97-things/97-things-every-programmer-should-know/tree/master/en/thing_73].

-# Part 3: Testing

18. Introduction to Testing

// if you don't use TDD, insert errors into the production code to test the tests. https://github.com/97-things/97-things-every-programmer-should-know/tree/master/en/thing_95

"Algorithms + Data Structures = Software" Adapted from Niklaus Wirth

"Abstractions + Testing = Engineering" - me

=> Software Engineering = Algorithms + Data Structures + Abstractions + Testing

It may sound surprising to you, but proper testing is an absolutely essential step toward writing better code. It *forces* you to write better code. In fact, this was the first chapter that I wrote for this book, precisely for this reason. In the following chapters, we will learn why tests are crucial, how to write them effectively, and what to consider when creating tests.

A short story about tests

In the early days of software engineering, people wrote code and packaged it into the software they were selling. Before the release, the entire company had to pause all its work for two weeks to manually verify that all the features were implemented correctly. The software developers had to work night shifts to fix the bugs as soon as possible because otherwise the release of the software would be delayed.

But that's not the end of it. Of course, the company wants to make more money. They added some minor features to this extensive software and resold it. But here comes the problem: before they could release the software and make a lot of money, they had to redo the quality assurance process all over again. All the code that has been changed since the last check needs to be tested. *All* the code has to be tested because developers also changed the code used by old features. Once again, the entire company will be on a two-week freeze.

Obviously, this is highly frustrating. Before every release, you have to test a feature that didn't change at all, yet the team could have introduced some bugs. Before every release, you waste two weeks of your time on the same boring and repetitive task. Before every release, the company spends millions to test things that have already been tested several times before. And even worse, as the software grows, the number of bugs increases. Some of them even slip through the expensive testing. As the bugs become more challenging to fix, the release gets delayed. It's a nightmare.

During another terrible release, the company is on the verge of collapsing. The CEO comes to meet the development team. His tie is hanging loose, and he looks really tired. Apparently, it has been days since he last slept. And he says, "Guys, it cannot go on like this. These tests are killing us. We need the following: Here is a screen. At any time during the development process, I want to have a list of all the features that are currently not working according to the specifications. If everything works, it should be green. If you make this work, I'll pay you one hundred million dollars."

Silence filled the room. One hundred million?? You may laugh. But there are a lot of companies that would actually pay this amount for such a feature. It's an enormous amount, but at the same time, the efforts required for such a feature are incredible. There are millions of lines of code and tens of thousands of features. It's hard to find anyone in the company who knows what the specifications are. It will take years to get these automated tests working, and there is a possibility that the company will go bankrupt before completing all the tests.

On the other hand, the benefits for the company would justify this expenditure. At first, you might think, "Ah, spend one hundred million for saving two weeks of testing??". But there is so much more to it.

1. You can release anytime the screen is green. If the team works well, you can release every day (known as a "nightly build").
2. If a customer needs a feature urgently, you can quickly implement it and send him the nightly build.
3. There are fewer bugs because automated tests are more reliable than manual testing.

And that's only the marketing side of it. Equally important is the developers' perspective on this screen. So far, you have always been afraid that you would break some feature when changing code. A feature was working fine until suddenly, it broke down. Nobody realized when it happened. You'll spend the rest of your work life in constant fear. This situation is worse than the zombie apocalypse because you know it will never end. There is nothing that can make you feel safe again. You may never want to touch a single line of code again unless absolutely necessary, as you fear breaking something.

But now, all of a sudden... magic! If you accidentally broke a feature, you will know imediately. The screen indicates that everything is alright! Your paranoia starts to fade. You regain confidence in your code. In your abilities. In yourself! You can start replacing all this old, ugly code that has been patched together like a Frankenstein monster. Things were welded together by force because the author was hesitant to rewrite the existing code to create a cleaner solution. Suddenly, things look fine again.

You go to your CEO, give him a hug, and a box of chocolates. You thank him for saving your career and you repay him the one hundred million dollars.

Did I exaggerate a little to make my point? Maybe. But the exaggeration is smaller than you may think. The importance of writing automated tests cannot be overestimated. Tests are no guarantee to make your software project a success. But I can tell you that projects without automated tests are doomed and will fail sooner rather than later.

I hope this serves as sufficient motivation for you to read through this chapter and genuinely attempt to write tests on your own. As always, it's not easy at the beginning. It takes used to writing tests. Ask the internet and others for advice, and you'll get a fairly good idea of how to write them.

Test Example

Here is a small real-world example of how a test works.

1. Ensure that the coffee machine is clean and equipped with coffee, water, and electricity. Press the coffee button. Wait until the coffee has finished brewing.
2. Taste the coffee. If you like it, the test passes. Otherwise, it fails.
3. Discard the cup and the leftover coffee.

This is it. Tests always consist of a few instructions that should be easy to understand. The result of the test can only take on two values. It passed (you liked the coffee) or it failed (you didn't like it). If it failed, you should call a technician to fix it. Or even better, you could write a script that automatically calls the technician.

Tests consist of three stages that are conducted on a test bench.

1. Setup: Prepare everything for the test
2. Execution: Check if the requirements are fulfilled
3. Tear-down: Clean up all the objects created for the test

Structure of a Software Test

In software, we follow the same process as with the coffee machine in the example above. In every programming language, there is a major testing library dedicated to this purpose. They all function similarly, regardless of the programming language you use. The Python testing library is called pytest.

Here is a small example of a class we want to test:

```python
# inside vector.py
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance_to(self, other):
        return ((self.x-other.x)**2)**0.5
```

The corresponding test looks as follows:

```python
# inside test_vector.py
from vector import Vector
import math

def test_distance(self):
    v1 = Vector(0,0)
    v2 = Vector(1,1)
    assert math.isclose(v1.distance_to(v2), 2**0.5)
```

We can run the test in the command line with

```
pytest
```

The relevant part of the output is this:

```
E    assert False
E     +  where False = <built-in function isclose>(1.0, (2 ** 0.5))
```

Apparently I made a mistake in the implementation. The values checked in `isclose` are different. In my calculation I forgot to take the y-component into account. The correct implementation of the `distance_to` function would be

```python
    def distance_to(self, other):
        return ((self.x-other.x)**2 + (self.y-other.y)**2)**0.5
```

Now the test passes.

In case you find the error message returned by the test not informative enough, you can can add an error message either separately or directly to the assert.

There are different ways to change the error message of a failing test. The easiest is adding a message to the `assert` as follows:

```python
def test_function():
    a = 1
    b = 2
    assert a == b, f"is = {a}, should = {b}"
```

This returns the following error message:

```
AssertionError: is = 1, should = 2
```

As you can see, it's pretty simple to write a test. Not only in Python. There are testing libraries available for all major programming languages. From the perspective of the testing library, you won't need to learn much more than what I have explained here for a considerable period.

Once again, the difficulty lies not in the usability of the testing framework. The much harder questions are what, when, and how you should test. Let's have a look at the code and try to understand.

When

Our class vector contains the member function `distance_to`. It is part of the class interface and, therefore, must be tested. This is the price you pay for public functions. Or rather, it's a small fraction of the price you pay for having public functions. Keep functions private whenever possible. Private functions offer greater flexibility since you can modify them freely without the need for testing. For public functions, you have to ensure that the interface remains the same. Ensuring a function is public means committing to keeping the function unchanged. Avoid altering existing interfaces; it requires a significant amount of work. You'd have to adapt a lot of code, including your tests. Write tests for public functions to assert that you don't accidentally change their behavior and break your code.

How

Inside the test_vector.py file, we write the test case. Before you miss it, I'd like to emphasize the very first line. We want to test the `Vector` class. We have to import the corresponding file (or library).

Next, we will define the test case. Every test case receives a unique name. This name will appear in the test report if this test fails. It is good practice to give the test case a name that explains what the test does. These names may be up to one line long if necessary. You don't use these names anywhere else, so it doesn't hurt much to have very long test names.

Inside the test, we start with the setup part. In order to test the `distance_to` function, we need two vector objects `v1` and `v2`. In the following line, we calculate the distance between `v1` and `v2`.

Now comes the execution of the test. We check if the test result is correct.

Good inputs should thoroughly test the code. But they should also be simple so that they are easy to read. For tests, it is even more important that they are easy to read than for normal code. Tests should not contain any complex logic. Just follow the pattern of the example above. Set up, check, tear down. Make it as easy as possible.

General Thoughts about Tests

"Tests can only prove the existence of bugs, not their absence." - Edsger Dijkstra

One of the main misunderstandings about tests is that they are supposed to prove that there are no errors present. This corresponds to Dijkstra's fundamental attempt to mathematically prove that a certain algorithm is correct. This failed miserably. Programming is too complex for such fundamental approaches. They won't work because the complexity in any decent-sized program is too high. It is simply impossible to prove that a program is correct. And therefore, it is also impossible to write tests that prove that a program is correct.

Many people believe that the sole purpose of writing tests is to discover or prevent bugs. They couldn't be further from the truth. Of course this is one of the reasons why we write tests, but another reason is probably even more important: Tests enable us to fixate the behavior of the code.

Double Entry Bookkeeping

Robert C. Martin compared programming with tests to double entry bookkeeping [Clean Craftsman]. I really like this comparison. In both cases, you have two independent truths (creditor and debtor, or code and tests, respectively) that must produce the same result. Once both propositions yield equal results, it is highly likely that this outcome is correct. Especially if one of them is as simple as the test code. It is unlikely that the same mistake was made in both the code and tests when implementing them independently.

Having two absolute truths allows you to refactor one of them. You still have the other truth to ensure that the final result is correct. This allows you to refactor the code while leaving the tests unchanged. Or you may change the tests while leaving the code as is. The other, untouched component always serves as a ground truth against which you can compare your changes. This allows you to refactor your code without the fear of breaking it. If your tests fail for an unknown reason, you can simply revert your changes.

Here is a very small example of a function with a test.

```python
def add(a, b):
    return a + b

# inside test_add.py
def test_add():
    assert add(1, 2) == 3
```

Now, if you want the function `add` to return a different result, you'll also have to change the test accordingly. Each change has to be applied in both the code and the test.

On the other hand you are also free to refactor the `add` function as you please. As long as all tests still work, you are most likely fine.

Understand what you do

When writing tests, there are numerous factors to consider. The example above was very simple. In actual code, you have to work with much more complex objects. With many more arguments. But all together, it comes down to one point: Do you really understand what you want to test? If not, there is no need to start writing a test. It would never work. It would be a waste of time. Rewrite your code to simplify it or seek assistance to better understand the problem you need to solve.

A Few Recommendations

Ensure that all the tests pass. Tests that do not pass are worthless. Even worse, they are a nuisance. When you run the tests, failing tests can be confusing. They will confuse your coworkers. Everyone will waste time trying to fix the failing test. There is only one solution to prevent this: all tests have to pass all the time. Therefore, ensure that your Continuous Integration (CI) enforces that all tests pass. Tests that do not pass should be deleted.

In the setup phase, it is very common to have helper functions that create all the necessary objects. These are standard Python functions that generate the required objects. You might even have a utility code file for all the tests. It contains some fairly static objects like helper functions or class instances that you might need in many different tests.

There are also some aspects to be mindful of during the execution phase of the test. The first mistake that almost everyone made was checking two floating-point numbers for equality. Due to rounding errors, this will probably fail. There are specific approximate checks you should use instead. As the `isclose` function is used in the example above.

Then it is common to miss some of the if-else branches. These tests are crucial because most bugs tend to hide in conditional statements. Ensure that you address all cases, if needed, by utilizing a code coverage tool.

Similarly, you have to make sure your tests cover all the corner cases. This is one of the reasons why tests should be written by the same person who wrote the actual code as well. The developer knows what the corner cases are, unlike some other testers.

When refactoring code, ensure that you only modify either the code or the tests. This is the only way you can be sure that the changes are correct. If code and tests have to be changed simultaneously, the modifications should be straightforward.

Quality of Test Code

Tests are somewhat special, but ultimately, they are still code. In a serious project, the test code is likely longer than the actual production code you work with. When considering this, it becomes apparent that when writing tests, certain coding guidelines must also be adhered to.

When it comes to tests, it is even more crucial for the code to be easy to understand compared to regular code. It can't be too easy. You are even allowed to repeat yourself, at least a little. Or, as Jay Fields [Working Effectively with Unit Tests] put it: "When writing tests, you should prefer DAMP (Descriptive And Maintainable Procedures) to DRY."

Still, you should value the SRP. The code in tests should be clear to the reader. Refactor it as you would with any other code. Keep the functions short, use appropriate names, and eliminate unnecessary duplication. These aspects are often overlooked when writing tests. Even though the requirements for test code are somewhat different from those for production code.

Number of test cases

Probably the most challenging decision is determining the values you want to use in your tests. Writing one test case for a function is much better than having none. But maybe you just got lucky, and your code works exactly for this one number?

For a single-argument function, I recommend testing all possible corner cases and approximately two random values. As you wrote the function, you should know the corner cases: division by zero, passing an empty array as a function argument, and ensuring that the file used exists, etc. This is one of the reasons why a person writing the code should also write the unit tests. Only this person knows the corner cases. The acceptance tests, on the other hand, should be written by an independent person. But we'll come to that later.

When dealing with functions that have many arguments, it can become quite challenging to write tests. If you have 3 arguments and you would like to test 3 values for each, you end up with `3^3 = 27` test cases. This is quite a lot. Now, you really have to ensure that you understand what you are doing.

Here is an example of a function with three arguments. I have not documented all the test cases, but you can imagine how tedious this process might become.

```
def f(a,b,c):
    return a+b+c

# my_test.py
def test_f():
    assert f(0,0,0) == 0
    assert f(1,0,0) == 1
    assert f(0,1,0) == 1
    # ...
```

In some cases, the variables don't interact with each other. They are independent. You may test them independently. The number of tests reduces to about 3 for each variable, resulting in `3*3 = 9` test cases. This sounds much more reasonable, although it's still quite a lot.

Usually, the function arguments are not independent, or at least it is not clear how they interact. Otherwise, they wouldn't be in the same function. And it's generally not feasible to write 27 test cases; that's just too many. Just do your best instead. Test all corner cases and include a few random ones. If the function consists of well-written code that doesn't appear to be intentionally hiding bugs, you should be in good shape. And even more importantly, try to keep the number and complexity of the function arguments low.

I'd like to reiterate the importance of truly understanding the functionality of a function. As I have already mentioned several times, you need to test the corner cases. And you can only identify them if you know the code. Many corner cases are not easily discovered by chance. Nor can you figure out whether some variables are independent of each other or not. This is just one of the reasons why you have to write tests right along with the actual code. If someone else has to write the tests for your code, they are missing this crucial information and either have to read and understand all the code or just guess what it does. Both cases are suboptimal.

You may also have structured objects as input or output of a function. This can become significantly worse than dealing with three variables by orders of magnitude. Structured objects may contain a multitude of fields, such as elements in a list. Everything we have discussed so far becomes insignificant. But we can still achieve a reasonable test coverage if we try. First of all, all elements in a list have to be treated equally in your code. Avoid using a single list to store different elements! This is a fundamental rule when dealing with lists, see chapter [?]. It allows you to write tests for a fairly short list and deal with only one element of it. Or at least by selecting one element from a long list. All the other elements will behave the same. The only corner case you'll have to take care of is the empty list.

But also in large structured objects, the complexity is usually manageable. Most of the entries are typically quite independent and can be tested accordingly. Most of the entries from a large structured object are probably not even necessary within a function, as the object is quite generic. If you have a nested struct as a function argument, only pass the sub-structs that are actually used inside the function. Only change the values that truly influence the object under test. By following this approach, you can significantly reduce the number of test cases.

Again, it all comes down to the programmer understanding the relationship between different objects. Which parts of the object are actually utilized within the function? Write a generic test with default values for each component of the object. Then also write tests for specific values of the object. Though again, here you'll have to figure out which values are important.

As you can see, most of the complexity in tests originates from suboptimal code. If you write good code, the tests will be easy to write. In good code, there are few arguments with minimal nested structure, and all elements in a list are treated equally. Therefore, the number of test cases is low, and setting up the required data structures is comparatively easy.

Stages of a Test

As we have seen, a test generally consists of three stages.

The first stage is the setup. It creates all the necessary objects for the test. Usually, this consists of initializing all variables. For functional tests, however, this may also involve copying or creating files, or even databases.

The second stage involves executing the test. When testing, execute the function you wish to test and verify that the outcomes align with your expectations.

The third stage is the teardown. It cleans up all the files you created during the setup and execution stages of the test.

Setup and Teardown

Writing the setup and execution stages of a test is usually fairly easy. It's just normal code inside a test function, so we won't go into too much detail about those. The most difficult part is the teardown.

Setup and teardown are functions that are automatically called at the beginning and end of a test, respectively. This is ensured by the testing framework. Though most of the time they are not needed. The setup can also be replaced by a few helper functions. There is absolutely nothing wrong with that. At the end of the test, the interpreter or compiler cleans up all the variables as they go out of scope.

In most cases, especially in unit tests, there is no need for a dedicated teardown function. At the end of its execution, a unit test should clean up everything it does. However, if you write acceptance tests that use text files, databases, or something else that is persistent, things become tricky. Your tests may require temporary files, modifications to database values, network connections, etc. It becomes messy. You need a foolproof way to ensure that your file handling always works the same way, regardless of the outcome of a test. Even if it throws an uncaught exception. This is where setup and teardown really come into play.

For file creation, there is not much that can go wrong. You can create it from code or copy it from another location. This is to be implemented in the setup part of the test or using a specific function. When copying files or databases, ensure that the original file is write-protected. Otherwise, you might change it accidentally.

The tricky part is deleting the files at the end of the test. And yes, it has to be at the end of the current test rather than the beginning of the next test. Since you will likely rearrange the order of the tests at some point, cleaning up at the beginning of the test would not work anymore. Cleaning up at the beginning of a test is a fairly desperate measure and an obvious sign that something is seriously flawed with your test design. Every test should be able to assume that the everything is cleaned up before it starts.

It may sound very simple to delete a file at the end of the test, but if the test fails, for example, due to an uncaught exception, it aborts. All the code that follows in the normal control flow will be skipped. A typical function call to delete the file will never be executed. There would be a mess of undeleted files. This might impact future runs of the tests, causing them to become flaky (sometimes they pass, sometimes they don't). Flaky tests are one of the worst scenarios because they confuse everyone.

This problem can be solved by implementing the teardown function, which is guaranteed to be always executed, regardless of the test result. This is where the dedicated teardown function comes into play. It is guaranteed to be executed, even if there is an error occurring inside the test. Only in very serious cases, such as a segmentation fault, the teardown may not be executed. Though this is only a problem with low-level languages such as C++.

Anyway, try to write tests that do not require files or input/output operations. It makes things much easier. Especially with unit tests, you won't have to deal with setup and teardown functions.

Here is an example of a test with the special setup and teardown functions. [https://code-maven.com/slides/Python/pytest-class]

```
class TestClass():
    def setup_class(self):
        print("setup_class called once for the class")

    def setup_method(self):
        print("  setup_method called for every method")
```

```
    def teardown_method(self):
        print("  teardown_method called for every method")

    def teardown_class(self):
        print("teardown_class called once for the class")

    def test_one(self):
        print("    before")
        assert False
        print("    after")
```

The captured output is this:

```
------------------------------------- Captured stdout setup ----------------------
--------------
setup_class called once for the class
  setup_method called for every method
------------------------------------- Captured stdout call -----------------------
--------------
    before
------------------------------------- Captured stdout teardown -------------------
--------------
  teardown_method called for every method
teardown_class called once for the class
```

It is showing that the teardown functions are called even if the test fails, while a normal function like this `print(" after")` statement is not executed.

Whether you should use the method or the class function depends on your tests.

Helper functions

A test is also a programming object. Accordingly, it has to follow the basic rules, for example, the SRP. Though you don't have to follow the SRP as strictly as in regular code. As written above, in tests DAMP is more important than DRY.

Each test serves a singular purpose. It tests exactly one function or method. Testing multiple functions within a single test is considered bad practice. Write helper functions to set up a test, making it easier to add more test cases. You may even use a little bit of copy-paste code in tests if it makes the code more readable! Having many smaller tests forces you to structure them better and improves the overall overview.

Let's provide an example of how helper functions can make a test case easier to read.

```
def test_car_accelerates_if_gas_pedal_is_pushed():
    engine = Engine()
    wheels = [Wheel() for _ in range(4)]
    board_electronics = Samsung_TV()
    initial_speed = 0
```

```
    car = Car(engine, wheels, board_electronics, initial_speed)

    car.push_gas_pedal()

    assert car.speed == 1
```

This test has the problem that the setup takes much more than just one line. Thus we should create a helper function that takes care of the creation of the car.

```
def create_standing_car():
    engine = Engine()
    wheels = [Wheel() for _ in range(4)]
    board_electronics = Samsung_TV()
    initial_speed = 0
    return Car(engine, wheels, board_electronics, initial_speed)

def test_car_accelerates_if_gas_pedal_is_pushed():
    car = create_standing_car()
    car.push_gas_pedal()
    assert car.speed == 1
```

Now the test case looks much better. There is only one line for the setup, one line for the action we want to test, and one line for the assertion. Of course, we could also create the `car` object in a single line, but that is not the point here. The point is that the test case is much easier to read and understand.

The helper function can probably also be used in other test cases, reducing the total amount of code needed and possibly eliminating some duplication. One open question is where the line `car.push_gas_pedal()` belongs to. Here I prefer to have a 4th stage: setup, *execution*, checks, teardown. I like this explicitnes and don't see how this execution stage could be part of the setup or the checks. In my opinion it does something quite different and therefore deserves its own stage.

One final note regarding this test: If you are not accustomed to writing unit tests, you may find the test name to be quite lengthy. But this is not a problem. The test name is only used in the test report. It is not used anywhere else. Thus, it doesn't hurt to have a long test name. It is actually good practice to have a long test name as it enhances the readability of the test report. If you read a test report you should be able to pinpoint the error by reading the name of the test. A test name of 50 characters is completely normal.

Number of Assertions

There are some purists who argue that a test should contain only one assertion. I'm not sharing this opinion, even if they have a point. I believe there are cases where it is beneficial to include more than one assertion in a single test case. Otherwise, the tests become too verbose, in my opinion.

Problematic Tests

Just as with regular code, there are certain indicators that a test may be problematic.

Dependent Tests

It is common to encounter situations where a test can only pass if another test passes as well. They are coupled. For example, you have a function that creates a file and writes a number to it. You should write an acceptance test that calls this function and checks for the existence of the file (this is not a unit test, it may fail if there is not enough disk space).

Next, you write a test that reads the contents of this file. In this test, you will first call the function to create the file and then call the function to read it. Now there is a problem: These two tests are related. If the code fails to create a file, it will not be possible to read it. If the first test fails, the second test inevitably fails as well. This type of dependency represents poor design and violates the SRP. For one failing feature, only one test should fail. This makes it much clearer where the error originates. Having 50 failing tests at once can be extremely frustrating because it is not immediately clear why the tests are failing. Is it for a single reason or do they all fail for different reasons?

Unfortunately, having all the tests completely separated is a very difficult, if not an impossible task. There is always some correlation between the results of tests. But there is a technical solution that helps to some extent. In Python, you can skip tests if a requirement for the test is not met. Tests can depend on each other using the `@pytest.mark.dependency` attribute. This allows us to skip tests that would fail because another test has already failed.

```
pip install pytest-dependency
```

```python
import pytest

@pytest.mark.dependency()
def test_a():
    assert False

# skip test_b if test_a fails
@pytest.mark.dependency(depends=["test_a"])
def test_b():
    print("This will never be printed.")
    assert False
```

As in this example, `test_a` is always going to fail, so `test_b` will be skipped as it depends on `test_a`.

The output will be `1 failed, 1 skipped`. Only `test_a` was executed (and failed) while `test_b` was skipped. Once `test_a` is fixed, will `test_b` be executed as well.

For unit tests, dependent tests are generally not an issue. Each test covers only one unit, which shouldn't depend on any other units. Thus, unit tests are independent. For integration or functional tests [chapter Types of Tests], this is a different story. They can easily become dependent on one another. This is why it is important to keep track of the dependencies of the different tests.

Flaky Tests

Tests that do not always return the same result are called flaky. This is extremely bad. It's just like a false alarm once in a while. You may become annoyed and start ignoring it. Or maybe even worse, the alarm doesn't go off even though it should. Try to avoid flaky tests at all costs. It won't take much effort to rerun the tests. But the main problem is that it undermines the team's confidence in the test suite. You will never know if a test is failing due to your changes in the code or because, for example, the network is down. At times, rerunning a test might help, but this is only a superficial fix.

The only real solution is writing fail-safe tests. Write, for example, a test that checks the network connection. All tests that rely on the network connection will be dependent on this test. Structuring the tests in this way can significantly reduce flakiness. It is crucial to design your tests in a way that prevents flakiness.

Especially unit tests should never be flaky. A test only becomes flaky if some part of the code under test is flaky but this should never be the case for unit tests. Unit tests should not depend on things that can fail, such as the file system or network connections. This is one of the reasons why you should avoid testing input/output (IO) for unit tests and minimize it as much as possible for all other tests.

The following test is unreliable and consistently fails late at night:

```python
from datetime import datetime

def test_time():
    assert datetime.now().hour < 23
```

Of course, this is a pretty dumb example, but it's less exotic than you may think. Tests (and code) have probably already failed for similar reasons. People find plenty of reasons to write such kind of code.

Brittle Tests

Tests that are overly specified are called brittle. They break when changing the code in seemingly unrelated places. One example is testing a JSON file for formatting, even though the contents of the JSON file [chapter data files] does not depend on the formatting. The formatting does not matter. It does not change any of the values in the file. Testing the formatting is just a waste. Even worse, it is an unnecessary liability because it tests something that should not be tested. Something that the result does not depend on. Instead, utilize a JSON library to extract only the real values stored in the file and then compare them. This is what we are really interested in. Avoid using string operations when reading a JSON file. JSON should never be read as a string and parsed by your custom library. This is the very definition of brittle code!

```python
import json

def test_json_stable():
    x = '{"a": 1, "b": 2}'
    y = json.loads(x)
    assert y == {'a': 1, 'b': 2}

def test_json_brittle():
    x = '{"a": 1, "b": 2}'
    y = {x[2]: int(x[6]), x[10]: int(x[14])}
    assert y == {'a': 1, 'b': 2}
```

Another example of brittle tests is testing methods that should be private but are made public in order to test them. This prevents you from refactoring this function because it is now part of the public interface. Changing it will break the tests, even if the actual public interface remains unchanged. This is why private methods should not be made public for testing purposes. If you truly feel the need to test a private method, you should refactor it into a separate class. You should always just test interfaces, they are more stable than implementations.

Random Numbers

If you ever use random numbers in your code, you might get stuck with your tests. You think. How can you test something that is random? Well, you can. Random numbers generated are typically not truly random. Your computer generates them. It uses an algorithm to generate numbers that appear random, but it still produces numbers in a deterministic sequence. Always use the same random number algorithm and seed (initial value) consistently to ensure reproducible results for each test case. Only use truly random numbers once you have deployed your software.

The Beyoncé Rule

A common question is "What to test?". A very simple answer is: everything. This is certainly a correct answer, although you cannot always test everything equally extensively. You will simply lack the capacity for that. Instead, at Google, they came up with the Beyoncé Rule. [Software Engineering at Google] She sings in her song, "If you like it, then you should have put a ~~ring~~ *test* on it." Apparently, this applies to most of your code. You do like your code, don't you?

Exceptions and Tests

One thing people people frequently forget to test are exceptions. And you should not only test the type of the exception, but also the exception message. Both, the exception type and the message, are important. That's why they exist.

Test-Driven Development (TDD) may help to avoid this issue. With TDD you first write the test and you'll realize right away that the your tests already pass. Which is an indication that there is something wrong with your tests.

Not Automatable Tests

As software engineers, we aim to automate everything, including tests. However, this is not always possible. There are still things that we can hardly automate. One example is image processing algorithms. How much can an image be compressed while still maintaining good quality? This is very difficult to determine with an automated test and is better assessed by humans. When running complex simulations, such as analyzing the aerodynamics of an airplane, it is impossible to create a test to verify the accuracy of the simulation results. Simply because you don't know the correct result. You can only judge if the result makes sense based on your experience. There are still things that are better tested by humans than computers.

19. Types of Tests

There are different types of tests, depending on their scope. There are several different categories of tests. Though, for the sake of simplicity, I'd like to reduce it to only 3 different types. Please note that the distinction between the different types of tests is not always clear. There are some tests that are a combination of two different types. But, in general, the following three categories are sufficient.

1. Unit tests assess the behavior of individual functions, classes, and modules.
2. Functional tests assess the behavior of the entire software system. // rename acceptance tests to functional tests globally?

As we will see, each of these categories has its own right to exist as they each cover different parts of the code. They are all important and should be used in combination. There are also other types of tests that we will delve into later, while others we will simply ignore. Also, the naming of the different types of tests is not standardized. There are different names for the same type of test. For example, functional tests are also referred to as end-to-end (E2E) or acceptance tests.

The small unit tests are the foundation of the testing infrastructure. They can be executed quickly. Meanwhile, as tests become more complex, they may take longer to execute as they are designed to assess the interaction of components rather than individual components themselves. Thus, larger tests are more likely to find bugs, but at the same time, they are not suitable for pinpointing them.

Functional tests can also be written in a different programming language and by a different person than the underlying code. They depend, for example, on the API that might be written in a different language. I wrote functional tests for some C++ software in Python because it was easier to process the resulting text files.

Unit Tests

First, we have to figure out why unit tests are actually needed.

Many programmers follow this workflow: they write a function and then need to determine if it works correctly. To achieve this, they utilize print statements or the debugger. They run the code and check if the results are correct. Let's look at the following example.

```python
def square(x):
    return x**2

print(square(1))
print(square(2))
print(square(5))
```

This works. People worked like this for decades. But it's absolutely terrible. The print statements will be deleted once the code works. The checks will be discarded, and no one knows anymore what the code is actually supposed to do. Whether it still works. When modifying the function, you have to test it again. Everything. Every time. By hand! This is a typical example of a procedural DRY violation that should be optimized. The solution is unit tests.

Unit tests cover relatively small sections of code. Usually, they test a public method or a standalone function. In the example above, the unit tests would verify everything that is typically verified using print statements. The unit tests for the `square` function would look something like this:

```python
def test_square():
    assert square(1) == 1
    assert square(2) == 4
    assert square(5) == 25
```

This code snippet essentially performs the same function as the print statements mentioned earlier. But with a very important difference, this test code will remain. It goes into the test suite and will remain there indefinitely. Or at least as long as you still have the `square` function defined. This test will be executed every time you run all the unit tests. You'll know if the code still works, even after changing the underlying implementation. The only drawback is that it takes a millisecond for each test to execute, and these numbers may add up as you keep writing unit tests. Additionally, you will have to modify the test code if you change the implementation. But the last point is actually a good thing. It prevents you from inadvertently changing the behavior of the code. You have to assert that you want the behavior to change. If you change the actual code, you also have to update the corresponding unit tests.

It may sound surprising, but unit tests are the cornerstone of the testing infrastructure. They are even more important than functional tests. This is because unit tests are fast and can provide precise information about which part of your code contains a bug. In contrast, functional tests can only indicate that something is wrong within the entire code base and they require a significant amount of time to execute. Therefore, having your entire code covered with unit tests will have a similar impact as having it covered with functional tests. However, unit tests have the advantage that you will know precisely where an error occurred, and they are much faster to execute.

The drawback is that unit tests do not verify whether these building blocks are connected correctly. Unit tests cannot assess the interaction between different code blocks.

Testing Files in Unit Tests

Usually, unit tests only require a setup and an execution phase. There is no tear-down function required for unit tests since they do not interact with any files or databases that need to be deleted afterward.

"Why...? How? No files? No database?"

Yes, that's a good point. According to the SRP, a function or class should only perform one task. Therefore, it should not read a text file and perform complicated calculations. Reading a text file should be done in a dedicated function. This function will not have a unit test. But it is not necessary to test automatically because reading a file and returning it as a string is not a difficult task. The code will be covered by functional testing.

Let's say we have the following code.

```python
def share_values(filename):
    with open(filename,'r') as f:
         file_content = f.read()
    share_values = parse_share_values(file_content)
    # ... and much more code
    return share_values
```

The section of this code that reads the file is very simple. It is not necessary to test it. Instead, it can be easily extracted into a separate function. This method is referred to as the "Wrap Method" by Michael Feathers [WELC, p.70]

```
def get_share_values(file_content):
    share_values = parse_share_values(file_content)
    # ... and much more code
    return share_values

def read_file(filename):
    with open(filename,'r') as f:
        return f.read()

def share_values(filename):
    file_content = read_file(filename)
    return get_share_values(file_content)
```

Here, we wrapped the code for reading the file into a separate function. The rest of the code is written within a dedicated function. For this function, one can easily write a unit test because it does not depend on the file system. A test might look as follows:

```
def test_get_share_values():
    file_content = "Apple, 150.3"
    assert get_share_values(file_content) == {"Apple": 150.3}
```

This is similar to the GUI layer for functional tests. You pack everything you don't want to test into a thin layer that is unlikely to fail, making the remaining test much smoother. In this case, this small layer is the function `read_share_values`, which reads the file into a string. Uncle Bob refers to this as a "Humble Object" [Clean Craftsman p.157]. It is a thin layer that is unlikely to fail and therefore does not need testing. It is simply a thin wrapper around the function that reads the file.

The same holds true for database access or retrieving the current time value. You write a small wrapper function that does nothing but calls the database or returns the current time. Separate the remaining code into a distinct function that can be tested independently.

An even better solution is to implement Dependency Injection (DI) [https://martinfowler.com/articles/injection.html] as explained in the next chapter. But for the moment, we'll leave it with the small wrapper function.

Testing classes

Writing unit tests for classes is arguably the most crucial aspect of this chapter. This is not only due to the prevalence of classes but also because classes tend to become messy without any unit tests.

First of all, classes tend to become too large. They have too many member variables and complicated methods. Both will make it very hard to write unit tests. Member variables share the same issues as function arguments do [Classes]. Member variables increase the dimensionality of the problem being tested. This leads to many more possible test cases than should be required for good class design, as discussed in chapter [Testing].

Furthermore, there is the issue of how to handle private methods in large classes. Apparently, the testing framework does not have access to private methods. No one has, except for the class itself and perhaps some friends' classes. One initial approach is to change the private methods to public. This, however, is not recommended. It is not advisable to make methods public solely for testing purposes. This will result in convoluted code with an excessive number of public methods, which is the complete opposite of encapsulation. For the same reason, you should resist the temptation of making the test a friend class of the class under test. Therefore, unit tests (and certainly all other tests) should only test the public interface of a class. It should test the class as a whole. If you are tempted to test private methods, you should resist. This is a clear sign that your private methods are too complex. Consider creating a separate class for these private methods with a public interface that can be tested.

Classes that are difficult to instantiate pose another problem. For example, if an object is difficult to construct, or if the constructor has side effects that are not guaranteed to be undone by the destructor. Such as opening a file, incrementing a counter, etc. In the actual code, it can be ensured that all necessary conditions are met so that you never encounter any issues. For instance, if you are instantiating a class only once. When running unit tests, however, these guarantees may be broken in some cases, leading to undesired behavior. For these reasons, the constructors should be small and not execute any fancy operations.

In summary, the following points can be made about classes and tests:

- Classes should be small and contain few member variables
- If you want to test private methods, consider refactoring them into separate classes.
- The constructors should be simple and not have any side effects

All these rules are implied by the topics we have covered so far. But now we have a reason why we absolutely have to obey them. The unit tests compel us to do so.

Here is an example of how to refactor a complex private method into a dedicated class.

```
class Car:
    def __init__(self, engine):
        self.engine = engine
        self.speed = 0

    def push_gas_pedal(self):
        self.speed += 10
        self._increase_rpm()

    def _increase_rpm(self):
        self.engine.rpm += 1000
```

Apparently this code is bad because `increase_rpm` should be part of the `engine`. I made this code deliberately this bad in order to fix it now. Let's assume we want to test the `_increase_rpm` method. We can refactor it into a separate class.

```python
class Engine:
    def __init__(self):
        self.rpm = 0

    def increase_rpm(self):
        self.rpm += 1000

class Car:
    def __init__(self, engine):
        self.engine = engine
        self.speed = 0

    def push_gas_pedal(self):
        self.speed += 10
        self.engine.increase_rpm()
```

Now the code is much better. By moving the method `increase_rpm` into the `Engine` class and making it public, we can now test it. Furthermore, this method belongs to the `Engine` class, not the `Car` class.

Copilot

Copilot can be significantly helpful when writing tests. I have written a function to convert numerical values into Roman numerals and have created a unit test file. Copilot started implementing the unit tests without any additional instructions.

```python
from refactoring import roman_number

def test_roman_number():
    assert roman_number(1) == 'I'
    assert roman_number(2) == 'II'
    assert roman_number(3) == 'III'
    # ... and tests up to number 42
```

However there are two minor things that I'd like to have improved. First of all there should be preferably only very few asserts per test. Here we have 42 of them.

Second, the test is testing things that were not even implemented in the code. The roman number function was only implemented for values up to a value of 3. So it seems as if Copilot somehow guessed what kind of tests were needed but did not check what is actually implemented.

The code above can be refactored, for example using a dict,

```python
    # refactor this code to use a dictionary
    dictionary = {1: 'I', 2: 'II', 3: 'III', 4: 'IV', 5: 'V'}
    for key in dictionary.keys():
        assert roman_number(key) == dictionary[key]
```

Even if I desired this change, upon reviewing the code, it is not entirely clear whether this is an enhancement over the original code. We have removed some redundancy and now only use one assertion. On the other hand, the redundancy was not that significant, and the old code was very easy to understand, which may be even more important than removing the repetitive code. This decision requires human judgment, and I am still unsure which solution is better.

Functional Tests

Functional tests perform tasks that align with most people's intuitive expectations of a test. Some marketing personnel, for example, the Product Manager (PM), orders a new feature. He tells you, more or less exactly, what this feature should do and provides you with some examples. The feature is considered complete once these examples can be executed using your software. As you don't want to end up in the same situation as in the story in the previous chapter [Testing] with the desperate manager, you write automated tests that cover the examples. This is a fairly good guarantee that the feature is still working, even if someone were changing the underlying code. So, there is one thing you will always do: write a functional test for every new ticket.

If you publish code examples as part of your API documentation, you should write a functional test for every single one of them. There's nothing more embarrassing than including failed examples in your documentation.

Functional tests are user-centered. The user lacks knowledge about the internal workings of the code. He doesn't want to know anything about the internals of the code. He only has the interfaces you provide: GUI, API, keyboard, webcam, etc. This is all he cares about. He wants to watch a YouTube video. He wants high image quality and fast response time. He doesn't care what kind of fancy algorithms the thousands of Google employees developed to control all the server farms.

Sounds good. But at the same time, it seems extremely difficult to write these tests. Testing a GUI or webcam input seems quite challenging.

True. But when making a few simplifications, the effort becomes quite reasonable. Most importantly, you need to have well-structured code. As shown in Figure [levels of abstraction...?] the GUI is an abstraction level higher than the API. Don't mix the two! The GUI code consists of HTML and CSS code, images, buttons, and graphs. These things are difficult to test automatically, but they do not contain any logic that is likely to have bugs. As mentioned before, this is called a Humble Object. This layer is difficult to test but unlikely to fail. Every mouse click corresponds to a function call to the underlying API. If the GUI looks fine, it is most likely functioning correctly. It is a thin layer that does not contain any logic and is unable to hide bugs.

Of course, if you neglect the GUI layer, the tests are not real functional tests anymore. Maybe one should rename them. However, I continue referring to them as functional tests since the API remains a public interface to your software.

Writing tests at the GUI level is quite challenging. Though there are tools, such as Selenium [https://www.selenium.dev/], that automate clicks on the GUI and translate them into API calls. It is generally recommended to keep the number of GUI test cases as low as possible. However, there are simply too many programs that are not structured as recommended in this book. They cannot be tested otherwise because they don't have an API that is well separated from the GUI. Meanwhile, there is considerable demand for testing these programs nonetheless. Needless to say, using these testing tools adds significant overhead to the testing efforts required.

Testing on the API level is comparatively easy. You can translate each button click from the GUI examples directly into API function calls. Write a test that makes the API calls, checks the results, and you're done. However, there is one problem with functional tests. In practice, you have to deal with potentially large files, databases, and slow network connections. This may significantly slow down your tests. Additionally, the files or databases must first be created. This task can be accomplished either by using a script or by copying them from another location.

The output of the tests may potentially result in large files as well. Comparing the results of these large files may not be very helpful. One tiny difference in these large files won't provide much insight into what is malfunctioning. One option for improving performance is to compare hash values instead of complete files. It won't provide more information than indicating that the files are different, but at least it is much faster to compute. And remember: functional tests are not there to pinpoint the source of a bug. They are just an indication that something might be wrong.

One solution is to use small files. This makes the tests run faster. However, having only tests with relatively small data sets and files is not representative of the everyday usage of your software. You absolutely have to run performance tests with realistic data sets as well. Otherwise, you might run into all kinds of performance problems at the release.

Functional tests are often highly correlated. A single bug in your infrastructure code can cause many tests to fail. Therefore, it is important to mark functional tests in python with `@pytest.mark.dependency()` as explained in the section on [Dependent tests]. You should always combine functional tests with unit tests to pinpoint the source of the bug.

Other Kinds of Tests

Unit and functional tests are not the only kinds of tests. As we have seen, there are, among others, also integration and performance tests. Here we'd like to briefly have a look at what other kinds of tests there are.

Integration Tests

As already mentioned above, integration tests are a mixture of unit and functional tests. They test entire components of the software, such as a module. As you want to isolate this module, you have to mock all the other modules that it depends on. This may be a considerable amount of work, as you have to write mocks for every other module. You can pass these mocks using dependency injection (DI) or run for example a mock dataserver which is very much simplified.

This effort may be worth it as integration tests are much faster to run than functional tests. Therefore you can run them more often and they are a better help to pinpoint bugs.

Performance Tests

Performance tests are one type of test class that is frequently overlooked. Functional tests are often developed for small databases to minimize execution time. But this leads to the problem that executing the code with normal-sized databases is not tested, and chances are that this would be unacceptably slow. For this reason, it is important to write performance tests that run with realistic parameters to prevent a poor user experience caused by slow response times.

There are many different types of performance tests. The most common type of testing is load testing, where, for example, the number of users is increased until the system breaks down, or one measures the response time of the system. It is the goal of these tests to determine the system's limits.

Explorative Tests

Explorative tests are designed to uncover bugs that the developer may not have considered. They are typically carried out by the testing or quality assurance team. They are not automated and are not part of the test suite. They are executed, and if a bug is found, it is reported to the developer. Otherwise, it is just ignored. Explorative tests are not a substitute for unit or functional tests. They are just an additional tool to find bugs.

Conducting exploratory tests requires some experience in anticipating potential issues. What corner cases might have been overlooked by the programmers?

When to run Tests

It is very important that all tests, excluding exploratory tests, are run automatically. This is the only way to ensure that they are always run. When they are run exactly, however, depends on the kind of test.

Unit tests are fast. Each one of them takes only a few milliseconds to run. All together, they shouldn't take more than a few seconds. Split the unit tests up in subgroups if your program becomes too large and it takes more than a few seconds to run all of it. It is important that unit tests are fast because they are run frequently. You should run them every few lines of code that you write.

Code is only allowed to be merged into the master branch if all unit tests pass. This means that every programmer has to run the unit tests before creating a merge request (MR) [devops] in the same way as ensuring that the whole project compiles. It is mandatory to fix the code that broke the unit tests; otherwise, it will not be merged.

Now let me reiterate: It is *mandatory* that all unit tests pass before an MR can be merged into the master branch. This is a rule that should be automated. Set up the Continuous Integration (CI) [devops] accordingly. It should check the unit tests just the same as it checks the formatting and the compilation of the code. This is just another mandatory requirement within the MR, in addition to ensuring that the code compiles. This is the only way to ensure that the unit tests always pass.

When it comes to functional or performance tests, things get a little trickier. They are slow and cannot be run before every MR. It would slow down the entire development process too much. Therefore, you can't guarantee that all functional and performance tests will be run all the time. Instead, you have to set up the CI to run them overnight ("nightly build"). If a test fails, it should send an email to all the developers who made changes on the previous day. The team must then gather and determine the reasons behind this situation. Usually, it is fairly obvious why the tests failed, and it won't take much time to figure out who broke the test and how. But it is important that the problem is resolved as soon as possible.

Integration tests are once again a mixture between the two. They are usually too slow to be run after every code change. But they don't have to be. They are only checking complete modules and not every code change alters its behavior. Integration tests should take only a few minutes, compared to potentially hours of the functional tests. This allows them to be run in every MR.

Who should write Tests?

With unit tests, it is evident that the responsible developer must write the tests. He knows the code best and understands what it is supposed to do. Unless you work in the automotive, medical, or aerospace industry, where the tests are written by a dedicated tester due to high regulations.

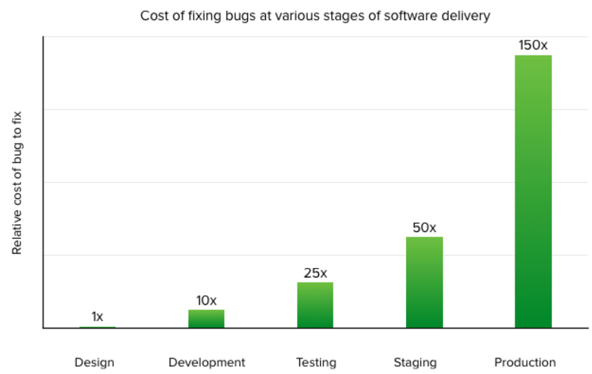
When it comes to functional and performance tests, the situation is not that clear. Should the tests be written by someone from the development team, the marketing side, or an independent tester? As always in software engineering, such questions have no easy answer. There are trade-offs to consider when choosing between different solutions.

Having developers write the tests has the advantage that they know the code. They know the difficulties. They can address these challenges by creating specific tests. A developer might also understand what the customers want and where the common issues lie. This also helps to target the most critical areas of the code.

On the other hand, having an independent tester also has some advantages. He doesn't know about the weaknesses of the code. Instead, he writes more explorative tests. These tests might uncover bugs that developers did not anticipate, as they are in areas of the code where bugs were not expected. Additionally, developers are usually overconfident about the quality of their code. They think the code is better than it actually is. This is why it is beneficial to have an independent tester who is not influenced by the code. Furthermore, independent testers are typically closer to the customer and create tests that closely resemble the actual use case.

Tests should be written as early as possible. This principle applies not only to unit tests but also to functional tests. Writing tests at the end of a project has the drawback that possible issues will be very hard to resolve because the entire software is nearly finished, making changes very difficult and potentially expensive. As a general rule of thumb, the cost of fixing a bug increases exponentially over time.

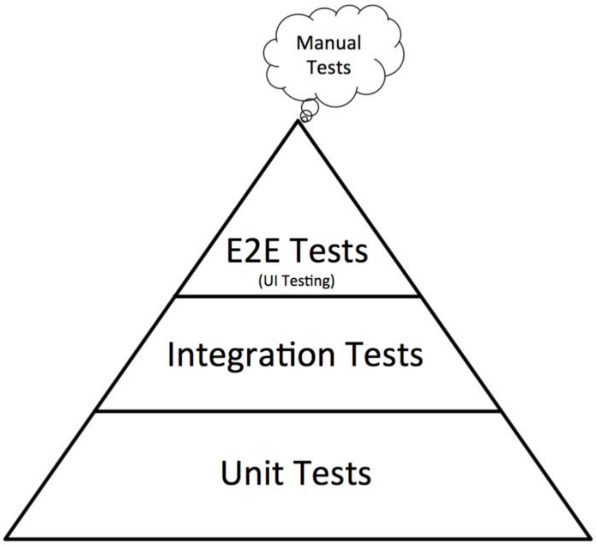
// get an image without copy right



The Testing Pyramid

We have said that we should write mostly unit and functional tests, maybe also integration tests. Additionally we have performance tests and exlorative tests. As a rule of thumb, one can say that the testing suite of any program should consist of many fast, small-grained tests and comparably few slow, coarse tests.

Unit tests are the foundation of the testing pyramid. They are generally the most useful as they check each part individually and can provide detailed feedback if something is broken. They are like testing the individual parts of a car radio before assembling it. Unit tests prevent the use of faulty components. Roughly, an estimated 80% of all tests should be unit tests. [software engineering at google].



// get an image without copy right

Integration tests are the second level of the testing pyramid. They are like testing the assembled radio on a test stand. Integration tests are coarser than unit tests and cannot pinpoint errors as precisely. But they are still useful for checking the functionality of the radio. About 15% of all tests are integration tests.

Functional tests should only verify that the installation of the radio in the car was successful as expected. Turning it on once should be completely sufficient as there is not much more that can still go wrong.

Functional tests are the least common. They are very valuable for verifying that a program actually works. There are always some things that can go wrong, even if all unit tests pass. However, the feedback you receive from a functional test is very limited. It will mostly indicate that something is wrong, but you will spend a lot of time debugging the cause of this issue. On the other hand, you don't need too many functional tests. If you have good test coverage with your unit and integration tests, the likelihood of experiencing numerous failing functional tests is low.

Once you confirm that the engine, gearbox, and brakes of a car are functioning properly and working in harmony, there is not much more to test on the fully assembled car. If it runs, it's probably fine. Only about 5% of all tests are functional tests.

20. Writing Better Code with Tests

"Quality is a product of a conflict between programmers and testers." — Yegor Bugayenk

[https://www.testim.io/blog/test-automation-benefits/]

Tests are not only important for writing correct code. They are equally important for enhancing the code you write. At least if you embrace them and you are not just writing tests for the sake of it.

Unit Tests

Unit tests ensure the correctness of your code at a small-scale level. Thanks to unit tests, you no longer have to manually verify if the results of a function or class are correct. The unit tests check it automatically. But this is only half the reason why they are so important. The other half might be a little bit unexpected for you: unit tests compel you to write better code. When writing unit tests, you realize immediately whether your code is good or bad. If writing a unit test is difficult, writing normal code will be difficult as well. It indicates that there are some design issues in your code, and you should consider redesigning it.

During the setup phase of the test, you have to create all the required objects. If this task becomes more tedious than expected, your data may be scattered in inappropriate locations. This is a strong indication that the design of your code is poor and needs to be reworked. When writing a test, you are a user of your own code. And your code should be user-friendly, as we have learned in the chapter on interfaces. Thus, if your code is difficult to use, it is considered poor quality.

In good code, all the relevant data is easily accessible, and constructing it manually for a test case is fairly simple. Preferably, you should have one large object with relatively static information that you can reuse in all tests, along with a few small, dynamic objects that vary in each test.

If you write a test, you have to know the expected outcome of the function call. If you struggle with the simplest cases, chances are high that your functions are too complex. They should be simplified. Rewrite the code until you can explain to your colleagues the functionality of the code. Until you can write a test case. Otherwise, you'll run into significant problems along a bumpy road.

You will be running the unit tests continuously. After every function you define, after every successful compilation, after every coffee you drink, and certainly every time you pull code from the repository. It provides constant feedback on whether everything is functioning properly or if something has been broken. This is invaluable. The only price you pay is the execution time of the unit tests. Keep them small and fast. A single unit test should not take more than a few milliseconds. You will be running hundreds, if not thousands, of tests all the time, so execution time is crucial.

Finally, I would like to emphasize once again the importance of this chapter. Learn how to write effective unit tests. Read this chapter again or, even better, search for more elaborated examples. There are thousands out there. And most importantly, once again, write tests yourself and discuss the design questions with your colleagues. This is how you will really make progress.

Integration and Functional Tests

At first, it sounds great to write integration or even functional tests. With relatively few tests, you can cover a significant portion of the codebase. But this comes at a price.

Integration tests cover much more code than the sum of many unit tests. This has the advantage that a single test is much more likely to find a bug. But it also has its drawbacks. Integration or functional tests cover a large amount of code, which makes them slower and less precise when it comes to pinpointing an error. Fixing bugs identified through integration or functional tests is much more challenging than addressing bugs discovered through unit tests. At the same time, integration tests are also more brittle than unit tests. The interfaces are much larger, and there is a substantial amount of code underneath that can be altered. And ultimately, these tests are expensive to run. They are huge and they are slow.

Despite these drawbacks, integration and functional tests have their own right to exist. They help improve your code and also encourage you to write proper interfaces. It is important to write these tests from the beginning of a project to ensure that your components and the API are easy to use and to locate potential bugs early. Functional tests, for example, are important to demonstrate that the user stories are truly functional. To prove that the software really works. Or to write a test case if a bug is found.

Testing Existing Code

When working with code, you will end up writing tests for existing code. I know, in theory, this shouldn't happen, yet reality and theory do not always agree. Writing tests for existing code is much harder than writing tests along with new code. It can be challenging to identify the weaknesses and the underlying logic in existing code. Usually, there are corner cases that are really hard to find if you did not write the code yourself. Additionally, it might be challenging to set up all the tests because there are no interfaces, and creating objects is difficult. Writing tests for existing code can be really difficult.

Many people misunderstand the concept of testing existing code. It is not so much about finding bugs in the existing code. At least if you are not part of the Quality Assurance (QA) team. It's about creating automated documentation of the current functionality of the code. And yes, you read correctly: What it does at the moment. Even if you find some bugs, you should not fix them right away, as users might rely on this buggy behavior [API]. As Hyrums law [software engineering at google] states: "With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behavior of your system will be depended on by somebody." Or, to put it bluntly: If you have enough users, someone will certainly rely on buggy behavior of your API. Users are smart, and they have likely found a way to handle your buggy code. Fixing the bugs might destroy the user's code.

As you write tests, always ensure that they fail when you expect them to. It has happened to me several times that some tests unexpectedly passed without writing any code. Such checks prevented me from hours of frustrating bug fixing. The source of the issue was some build problems where I tested the wrong binary. Sometimes, it was also because I simply didn't understand the logic behind the existing code, and I didn't realize the feature was already implemented.

In the previous chapters, I recommended against testing private methods as it violates encapsulation. Instead, you were supposed to refactor the class and extract the private function into a new class of its own. With the existing code, you have to be a little more pragmatic. You can't simply take any code and refactor it as you please. This will certainly introduce bugs. Making these private methods public is indeed the only way you can test the class. Once the class is refactored, you should clean it up and make all methods private again, as they should be.

Apparently, this approach to handling private methods is considered a hack and should only be used when necessary. It also highlights the importance of writing tests immediately and consistently refactoring to prevent the problem from escalating out of control.

//more about testing existing code? See Michael Feathers' book, WELC.

Assertions

There were times when people thought that using assert commands in production code was a good replacement for writing tests. This is so terribly wrong!

The most obvious reason is that using asserts inside production code is a violation of the SRP. You are writing tests inside production code. I believe that nowadays it is widely accepted that tests and production code should be stored in separate files, and ideally in distinct folders.

Secondly, your production code is not designed to run automated test cases. Assertions are only executed if you run the software you create. It will highlight any violations of the assertions along the way, but this process cannot be automated. It can be used as something akin to an emergency sign. It is advisable to prioritize improving the quality of your tests over relying on asserts in production code.

Don't get me wrong. There is nothing wrong with asserts in general. Using asserts in production code as a substitute for tests is not recommended. The following two code snippets are perfectly normal and almost identical:

```python
def root(x):
    assert x >= 0, "smaller 0"
    return x**0.5
```

```python
def root2(x):
    if x < 0:
        raise AssertionError("smaller 0")
    return x**0.5
```

This is because the `assert` statement raises an `AssertionError` if the required condition is violated. You can even add a message to the `assert` command using the `, "smaller 0"` syntax. The only advantage of the second code snippet is that you can use custom exceptions instaed of the `AssertionError`, as we'll learn in chapter [Bugs, Errors, Exceptions].

Now as I said, this code is perfectly fine. But it is no replacement for unit tests as it doesn't test anything. It only checks the precondition of the function.

Test Driven Development

So far, we have written tests to verify the correctness of our code. We wrote the tests once we were done with the code. But there is nothing wrong with writing the tests upfront. It is called Test-Driven Development (TDD) [Test Driven Development: By Example, K. Beck, 2002]. In fact, I recommend using TDD in general. It forces you to think more about what you want to do. You have to figure out how the test should look before writing it. Once the test is written, you need to think about how to implement the feature. The importance of the test cannot be understated. It helps you understand what you really have to do. The test forces you to structure your code accordingly, which is a really good thing. Before writing the implementation of a class, you must define its interface. With TDD you decouple the code because your tests compel you to do so.

In software development, it may happen frequently that you have a specific model in mind intended to solve your problem. But it turns out to be too complex, and somehow, you don't manage to get it working. This might be a case of YAGNI (You Aren't Gonna Need It) [https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it]. Chances are you will never need this complex structure. Instead, you can write test cases for your specific requirements and ensure that all these test cases pass. Everything else can be taken care of later, once you determine that it is truly necessary. On a coding level, YAGNI can be prevented by writing tests first. If you do not need a piece of code to make the tests pass, simply do not write it. Even if you truly believe that it would be significant, aesthetically pleasing, and perhaps even enjoyable to write this piece of code. It is not needed now, and chances are it will never be needed at all.

Perhaps you do not fully understand yet how TDD really works. Don't worry. You should maybe first gain some experience with standard tests. If you don't immediately see how a test should look, at least. If you are unsure about the final appearance of the code's interface. Yes, there are several aspects of TDD that may seem a little unusual, and it takes time to become accustomed to it. But it is worth the effort. Keep trying TDD once in a while and start using it more and more often.

How TDD Works

You should write one test for the feature you want to implement or the bug you want to fix. I repeat: one and only one test. If you have functional and unit tests (which I hope you do), you might have one pending test case for each of them. There should be a test case that currently fails. If both, unit and functional tests, pass, you can take a day off.

Just kidding. If a test passes for an unknown reason, this is a serious issue that you have to investigate. Perhaps a feature has already been implemented, or your test may not be evaluating its intended functionality and needs to be refined.

Otherwise, you should start implementing. Figure out why the test fails. For new features, it's usually obvious. The test is currently evaluating a feature that has not been implemented yet. Now it's your task to write just enough code so that the test passes. No less and no more. You don't have to write great code at this step. Just make sure you find a satisfactory solution to ensure the test passes.

Once the test passes, you might have to refactor the code a little to get it back into shape. You have already written all the necessary test cases as a safety precaution. And then you are allowed to write the next test case until you are done with the feature and the acceptance test passes as well.

There is a simple pattern for writing code in TDD.

1. Write a failing test.
2. Write code until the test passes.
3. Refactor if necessary.

These three steps must be repeated over and over again until you have completed your ticket.

Also, with TDD you have to do some significant refactoring occasionally. This is inevitable and has to be taken into consideration. These refactorings involve entire components, requiring you to work with multiple classes simultaneously.

The Importance of TDD

As we learned in the chapter on interfaces, they should always be defined from the user's perspective. With TDD, you are adopting the user's perspective of your code. When writing a test, you are a user of the corresponding piece of code. Therefore, writing your tests before the code forces the programmer to adapt the code to the test. This is a good thing. It forces the programmer to write code that adapts to the user, making the interface of the code more user-friendly. Even if you don't apply TDD all the time, you still get used to write code that is easy to test and therefore also easy to use.

Example of TDD

TDD is best understood by examining a brief example. Let's write a program that converts Arabic numbers (the ones we use) into Roman numerals. As we learned just now, we start by writing the first test case.

```python
# inside test_roman_numbers.py
from roman_numbers import *

def test_one():
    assert roman_numbers(1) == "I"
```

If we run the test, it fails as was expected. But we can make it pass easily.

```python
# inside roman_numbers.py
def roman_numbers(_):
    return "I"
```

This may look odd at first sight, but it is perfectly viable code in TDD. There is no duplication and it does everything required to make the test pass, even if the function argument just gets ignored. As there is nothing to refactor, we can continue with the second test.

```python
def test_two():
    assert roman_numbers(2) == "II"
```

```python
def roman_numbers(n):
    if n == 1:
        return "I"
    else:
        return "II"
```

The code from the initial test is no longer adequate. We have to use at least some `if/else` clauses. You might feel the urge to refactor this code. But, at least for the time being, we leave it as it is. The need to refactor this code is not strong enough yet.

There is a rule of thumb stating that a one-time repetition of the code is acceptable and does not need to be refactored immediately. Only if the same code is repeated two times or more, should it be refactored, as this may lead to a better understanding of the problem. However, this rule contradicts the DRY principle to some extent. As you can see, we have few strict rules in software engineering. It is always a trade-off between different principles.

```python
def test_three():
    assert roman_numbers(3) == "III"
```

```python
def roman_numbers(n):
    if n == 1:
        return "I"
    elif n == 2:
        return "II"
    else:
        return "III"
```

Now the `if/else` statements start to take over. We have 3 possible cases and with a little bit of thinking we find an easy way to refactor them away. The new version of the code might look like this:

```python
def roman_numbers(n):
    return n*"I"
```

Let's add a fourth test:

```python
def test_four():
    assert roman_numbers(4) == "IV"
```

We don't know yet how to deal with `numbers > 4`, so we may return any value we want.

```python
def roman_numbers(n):
    if n == 4:
        return "IV"
    return n*"I"
```

For 5 we can just continue with the same pattern.

```python
def test_five():
    assert roman_numbers(5) == "V"
```

```python
def roman_numbers(n):
    if n == 5:
        return "V"
    elif n == 4:
        return "IV"
    return n*"I"
```

Two tests later we are again at the point where we have to refactor. This time we have to think a little harder how the logic of the function really works. One possible outcome of this refactoring is the following code:

```python
def roman_numbers(n):
    num = ""
    while n >= 5:
        num += "V"
        n -= 5
    while n >= 4:
        num += "IV"
        n -= 4
    while n >= 1:
        num += "I"
        n -= 1
    return num
```

In a second refactoring step we wrap the whole while loops into a single for loop.

```python
def roman_numbers(n):
    roman = ""
    arabic_to_roman = {5:"V", 4:"IV", 1:"I"}
    for arabic in arabic_to_roman:
        while n >= arabic:
            n -= arabic
            roman += arabic_to_roman[arabic]
    return roman
```

Supporting larger numbers can be achieved by adding them to the beginning of the `arabic_to_roman` dictionary. Note that I used a dictionary instead of a list of lists. This is because, as I mentioned in chapter [Data Types], all list elements should be treated equally. Thus, having a list `[[5, "V"], [4, "IV"], [1, "I"]]` would violate this principle. On the other hand, this approach using a dictionary is a little bit fragile. It is only guaranteed to work for Python versions >= 3.7 because dicts are guaranteed to maintain their order only since then. The following solution would probably be the best option as it is more robust, even though it is slightly longer. Though here we go into the realm of premature optimization.

```python
from dataclasses import dataclass

@dataclass
class NumberPair:
    arabic: int
    roman: str

def roman_numbers(n):
    roman = ""
    arabic_roman = [NumberPair(5, "V"), NumberPair(4, "IV"), NumberPair(1, "I")]
    for number_pair in arabic_roman:
        while n >= number_pair.arabic:
            n -= number_pair.arabic
            roman += number_pair.roman
    return roman
```

The remaining tests and implementations are straightforward. I'll leave them as an exercise for the reader.

Stubs, Fakes, and Mocks

// chapter 19 Types of tests, Mock and Stubs has some redundancy with this chapter. Maybe merge them?

In many instances, you may need to write a test, but the code you intend to test includes elements that you prefer not to test. For example a database or an internet connection. You want to create a fake database that returns the expected value and never fails. The solution is to create your own database. Not a complete one, of course. One that does only what you really need for this test case. It implements every function you call and returns the values you desire. You may need to incorporate a significant amount of logic into the fake database to achieve the desired behavior, depending on the complexity of your test cases. Perhaps you need separate mock databases for various tests. You might need a dedicated database that throws an exception in some special cases. Programming fake objects altogether is a lot of work, and it makes the code rigid because not all the functionalities of the fake object are implemented.

There are many ways to set up a fake object. We will only look at two of them: faking and mocking.

Mocking

[clean craftsmanship, p.118]

The first approach is to utilize an existing database and modify some of its functionalities using a mocking framework. In the following example, we simulate the result of reading a CSV file. In Python, this can be easily achieved using the Mock library. Most other programming languages have similar mocking libraries too.

```python
from important_stuff import read_csv

from unittest.mock import Mock

def read_csv(file_name):
    # ...
    return [1,2,3] # ...

def test_mock_important_stuff():
    # Override the `read_csv` function defined in important_stuff.py and return 
some values.
    read_csv = Mock(return_value=([4, 5, 6]))
    assert read_csv("unexisting_file.csv") == ([4, 5, 6])
```

This test passes even if the file passed as an argument does not exist. An alternative to using the mocking framework is to use dependency injection. This is explained below.

Mocks have some predefined behavior. In this case, it simply returns the values defined. Mocks are different from fakes, as they mimic real behavior to some extent. Setting up mocks is much easier compared to fakes.

However, there is a caveat with mocks. There is a better solution: Dependency Injection (DI). I like DI much better than mocking. In my opinion, mocking is a hack to get away with sub-optimal code and should not be used, unless you have to work with existing code where DI is not an option. Here is what the code looks like with DI instead of mocking:

```
def read_csv(filename):
    # ...
    return [1, 2, 3]

def mock_reader(_):
    return [4, 5, 6]

def read_data_from(reader, filename):
    return reader(filename)

def test_mock_important_stuff():
    data = read_data_from(mock_reader, "")
    assert data == [4, 5, 6]
```

With this code here you can define your own reader function without the use of a mocking library. And I think the code has become much clearer.

Faking

[clean craftsmanship, p.118]

A fake is a functional replica of the object you intend to substitute, although it is a simplified version. For example, the fake CSV reader in the following example does not read a file from the disk; instead, it simply returns a string stored in the code. When running tests, this approach is typically sufficient without the inconvenience of managing the file system, where your original data could be easily deleted or tampered with by others.

In the following example, the `FakeCSVReader` does not write the data to a file but stores it in a local variable.

```
class CSVReader:
    def __init__(self, filename):
        self.filename = filename

    def write(self, data):
        with open(self.filename, 'w') as f:
            f.write(data)

    def read(self):
        with open(self.filename, 'r') as f:
            return f.read()

class FakeCSVReader:
    def __init__(self, _):
        pass

    def write(self, data):
        self._data = data
```

```
    def read(self):
        return self._data
```

This `FakeCSVReader` clearly does not have a complete implementation of the `CSVReader`. It has just enough capacity to store some data and retrieve it later. But this might be enough to make your tests pass. Fakes should be used whenever a mock is not sufficient for your test case. The fake has much more functionality.

Dependency Injection

Faking and mocking are closely related to dependency injection (DI).

When using DI you can create a new object from scratch, for example, an object that returns an API key. Now let's first look at the code without DI.

```
import os

class ApiClient:
    def get_api_key(self):
        return os.getenv("API_KEY")

def main():
    print(ApiClient().get_api_key())

if __name__ == "__main__":
    main()
```

The API key is generated within the `ApiClient` class. This is bad for several reasons. Testing it is challenging. It is not easy to replace the API key with a fake one. Secondly, it is difficult to reuse. If you want to use the same API key in another class, you will need to copy the code.

If you want to change the `api_key` for testing purposes. One thing you can do is make the following selection:

```
import os
import sys

class ApiClient:
    def get_api_key(self, is_testing):
        if is_testing:
            return "1234"
        return os.getenv("API_KEY")

def main(is_testing):
    print(ApiClient(is_testing).get_api_key())

if __name__ == "__main__":
    is_testing = "testing" in sys.argv
    main(is_testing)
```

This, however, is considered bad practice. As we learned in the section on booleans [chapter Data types], such selections should not be postponed. Passing around booleans or strings is considered bad practice. Even if you replace the boolean with an `enum`, you should still avoid postponing this decision.

A better solution is using DI. Already in the line `is_testing = "testing" in sys.argv` we know whether we want to use a fake API key or the real one. Thus we can already at this point select the corresponding `ApiClient` and then pass it on, rather than some flag. Like this, the desicion is resolved on the highest level of abstraction, inside the main function.

```
import os

class ApiClient:
    def get_api_key(self, is_testing):
        if is_testing:
            return "1234"
        return os.getenv("API_KEY")

def main(is_testing):
    print(ApiClient(is_testing).get_api_key())

if __name__ == "__main__":
    is_testing = "testing" in sys.argv
    main(is_testing)

import os
import sys

class ApiClient:
    def get_api_key(self):
        return api_key = "API_KEY"

class FakeApiClient:
    def get_api_key(self):
        self.api_key = "1234"

def main(client):
    print(client.get_api_key())

if __name__ == "__main__":
    if "testing" in sys.argv:
        main(FakeApiClient())
    else:
        main(ApiClient())
```

In DI you pass a higher object like a class instance or a function as a function parameter as done here in the call of the `main` function. Here we used the `api_client` as a function argument which can be either a `FakeApiClient` or a normal `ApiClient`.

DI is a little odd at the beginning. I recommend you read this section again, search some additional examples online and play around with it. The important part is that you select the relevant behavior as early as possible in the program, rather than passing around boolean variables.

Using DI is generally a highly recommended practice and should always be employed when dealing with IO, time, random numbers, API keys, selecting an algorithm etc. This is because you can easily replace the injected code with something else.

The only drawback of DI is that the object has to be passed through the entire stack until it reaches the point where the API key is actually used. This leads to functions containing many arguments. But what would the alternatives be?

1. Do not pass any additional argument through the stack. This would prevent you from altering/testing the code.
2. Pass a string or an integer through the entire stack and make a selection based on its value, similar to how it is done with the `"testing"` value in the second example. This wouldn't be any better than passing the `ApiClient` object. Rather the opposite. It is better to pass a high-level object than passing a string, as this allows for making a selection based on it.

When you need to make a selection, such as when you want to change a value for testing purposes, using DI is the optimal choice. Delaying the decision would require you to pass around a boolean or string instead, which is considered bad practice. Selections should always be resolved promptly. And DI allows you to do exactly that.

DI is very similar to the strategy design pattern. The main difference lies in what you want to achieve. DI is primarily used for testing, while the strategy pattern is generally employed to enable the user to make a selection at runtime.

One of the few downsides of DI is that it may make the code harder to understand. To understand what's going on, you have to look through many functions to figure out which code got injected. For this reason, it is recommended to use DI only for things that are expected to change. It should be mostly used for the reasons mentioned above: IO, time, random numbers, API keyt and other things you want to change when testing or running the code, respectively.

Summary

Don't worry if you haven't understood everything. I just briefly explained dependency injection, faking, mocking, etc., which are all fairly advanced topics. I hope you have grasped some of the fundamental ideas I attempted to explain here. They can be useful, and the ideas behind them are very important. Especially TDD and DI were crucial topics in this chapter. If you want to have an in-depth look at some of the things we discussed here, I recommend exploring these two topics. There is plenty of literature available that delves into much more detail than I have provided here.

As always, many books only focus on OO programming. They only explain dependency injection for classes. However, having classes is not a strict requirement for dependency injection or the strategy pattern. In programming languages that support function pointers or duck typing, you can also pass different function objects as function arguments. This has the advantage that you don't have to deal with base classes and so on. Function pointers are not commonly used because they can only be utilized for simple objects, whereas complex objects are typically injected. While I generally recommend using dependency injection with class objects instead of injecting function objects. Simply because it can be used in all major programming languages in the same way, and you don't have to learn anything new.

So far, we have covered the technical implementation and introduced mocking and faking. But the real problem is yet to come. The question is how and what to test. Apparently, it is not a viable solution to create a full database simulation every time it is required. This is not only a hell of a lot of work. It also makes the code inflexible.

Copilot

Copilot is knowledgeable about dependency injection, as demonstrated in the Copilot example in the chapter on functions [chapter functions]. The difficult part is figuring out how to make Copilot utilize DI. When implementing a regular reader and a function, and defining the `reader_type`, Copilot recognizes that Dependency Injection should be used in the final line of the code here. I once again had to write very little code to make Copilot understand my intentions. I had to define the `class` lines as well as the `reader_type = "mock"` line, and Copilot did the rest.

```python
class CSVReader:
    def __init__(self, filename):
        self.filename = filename

    def read(self):
        with open(self.filename, 'r') as f:
            return f.read()

class CSVReaderMock:
    def __init__(self, filename):
        self.filename = filename

    def read(self):
        return "Mocked CSV data"

reader_type = "mock"
if reader_type == "mock":
    reader = CSVReaderMock("data.csv")
else:
    reader = CSVReader("data.csv")

def process_data(reader):
    return reader.read()

print(process_data(reader))
```

-# Part 4: Design Principles

21. SOLID principles

"It is not enough for code to work." Robert C. Martin

Source: [https://youtu.be/pTB30aXS77U], [https://youtu.be/9ch7tZN4jeI] and [Clean Architecture]

The solid principles were named by Robert C. Martin. SOLID is named after 5 general rules how to write object oriented (OO) code. These are:

1. Single Responsibility Principle (SRP)
2. Open closed principle (OCP)
3. Liskov substitution principle (LSP)
4. Interface segregation principle (ISP)
5. Dependency Inversion principle (DIP)

These 5 quite general rules describe mostly how classes, and also code in general, should be structured and interacting with each other. Obeying them helps with the design of the code.

Interestingly enough, many people agree on the fact that these principles are (or at least were) important, but there is no exact common agreement what some of these principles mean exactly. In my opinion, these principles hold for compiled languages as Java and C++. For interpreted languages like Python, only the SRP is really important and the OCP and the LSP are somewhat useful. The ISP and the DIP are nice to know but they are only important in compiled languages. We'll see why in a minute.

Single Responsibility Principle

The SRP has already been explained in its own chapter due to its enormous importance. At the time of writing I counted 60 occurances of the abbreviation "SRP" in this book here!

Open Closed Principle

"Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification." - Bertrand Meyer

The Open Closed Principle (OCP) was first mentioned by Bertrand Meyer in 1988. It is stated that an object should be open for extension and closed for modification. The original version suggests using inheritance to achieve this goal. [Object-Oriented Software Construction, B. Mayer] This is an unfortunate choice. Robert C. Martin suggested using interfaces instead [?]. Interfaces allow you to add multiple implementations at a relatively low cost, while modifying the interface itself can be quite expensive. Each class implementing that interface would also need to be modified.

Our code should be stable with respect to extensions in the future, but not to changes. If the requirements change, we must also modify our code. This is inevitable. But we shouldn't have to change our code if someone else wants to extend their code. Therefore, the solution is to utilize abstractions at possible abstraction points. This allows the user of our code to extend it without requiring any modifications to our code.

Let's consider a brief example. We have a class containing some postal codes of Swiss cities. If we want to add another city, we would need to include an extra function in this class. The class `City` is not open for modification. We have to modify it every time we add another city. This class does not adhere to the OCP.

```
class Cities:
    def zurich_postal_code(self):
```

```
        return 8000
    def bern_postal_code(self):
        return 3000

def print_all_postal_codes():
    cities = Cities()
    print(cities.zurich_postal_code())
    print(cities.bern_postal_code())
```

If the user of this code wishes to add another city, they will need to do so within our own code inside the `Cities` class. We have to *modify* the class `Cities`. This is the opposite of what the OCP wants to achieve. The OCP wants to separate the user code from the interface.

Instead, we can create an interface called `City` and implement it for every city we are interested in. We are free to add an additional city if we choose to. We don't have to change any existing class or interface. Instead, we can create a new object to *extend* the implementation of the city interface. The code below adheres to the OCP.

```
from abc import ABC, abstractmethod

class City(ABC):
    @abstractmethod
    def postal_code(self):
        pass

class Zurich(City):
    def postal_code(self):
        return 8000

class Bern(City):
    def postal_code(self):
        return 3000

cities = [Zurich(), Bern()]
for city in cities:
    print(city.postal_code())
```

Now, this code, on the other hand, fulfills the OCP. If the user wants to add another city, they can create as many additional cities as they want, and we don't have to worry about it. The base class `City` defines the interface, which is sufficient for us to work with any class the user adds. Of course, there are many other possible implementations of the OCP. Especially in a dynamically typed language like python. The code above is just an example.

If you like design patterns, the code here is a classic illustration of the OCP. It is the strategy design pattern [Design Patterns]. However, you might also consider using the decorator pattern, which also satisfies the OCP.

Liskov Substitution Principle

// see https://youtu.be/pTB30aXS77U

"If it looks like a duck, quacks like a duck, and needs batteries, you probably have the wrong abstraction." - Wisdom of the Internet

Implementation of interfaces should not blindly adhere to the "is a" principle. This is only a rule of thumb and not sufficient. Instead, the implementations really have to share the same interface.

For example, credit cards and PayPal should not implement the same payment system interface, even though they are both payment methods. The credit card requires a card number, while PayPal requires an email address [https://youtu.be/pTB30aXS77U?t=455]. This leads to a situation where you are unsure about what the payment interface should accept as an input argument.

```
class Payment:
    def make_payment(amount, card_number_or_email_address)
```

This logical contradiction regarding whether the second argument should be (email address or card number) violates the Liskov substitution principle. Credit card payments and PayPal payments should not use the same interface.

Instead, the selection of the credit card number or the email address should be done later, within the specific classes. It is there that these credentials have to be requested from the user.

```
from abc import ABC, abstractmethod

class PaymentSystem(ABC):
    @abstractmethod
    def make_payment(self):
        pass

class PayPal(PaymentSystem):
    def make_payment(amount):
        # ask the user for the email address
        # make a payment using the email address

class CreditCard(PaymentSystem):
    def make_payment(amount):
        # ask the user for the credit card number
        # make a payment using the credit card number
```

Interface Segregation Principle

"Clients should not be forced to depend upon interfaces that they do not use." — Robert C. Martin

The Interface Segregation Principle (ISP) is similar to the SRP for interfaces. Interfaces should be split up into many small parts. This is important in order to maintain low coupling. You don't want to import and compile a huge library solely for one small feature. If there are separate logical blocks within a library, ensure that they are also made available separately.

Here, file A does not comply with the ISP. It does two independent things. Most other code only needs one of these functions. They are independent. Thus, they should be in different files.

Now in Python, this is not such a big deal because you can import each function individually. Even if you import the whole file A, it's not a big deal. It's not becoming slow. In C++, on the other hand, adding unrelated functions in the same file is really a no-no. In C++, you always include an entire header file at once, and you will have to compile everything that is included with it. There might be a hefty price to pay if file A is too large.

```
// C++
// file A.h
int function_1(){
    return 1;
}

int function_2(){
    return 2;
}

// and many more functions
```

// add graphs on the file dependencies below

The solution is to split up the file A into two subfiles, A1 and A2. The goal is to find a way to achieve this so that the majority of the other files utilize only one of the newly created files, A1 and A2. A1 and A2 should exhibit high cohesion within themselves, but there should be low coupling between them. Ideally, the amount of code that you'll have to import is reduced by roughly half.

This process of breaking up files can be repeated until it is no longer possible to reduce the amount of imported code, or the number of imports would be growing unreasonably fast. At this point, you have finished segregating file A.

```
// C++
// file A1
int function_1(){
    return 1;
}

// file A2
int function_2(){
    return 2;
}
```

A well-known example of interface segregation is the standard library in C++. All the functionality is defined inside the `std::` namespace (`std::` a namespace, not a class!), but the whole library is split up into many different files. Importing the entire standard library solely for the sake of using a small portion of it would significantly increase compilation times tremendously.

Another common example is defining an enum inside a class, while other parts of the code might also need access to this enum. This section of the code needs to import the entire class that contains the enum, even though it only requires this simple enum and nothing else. This code imports more modules than necessary.

```python
# inside ImportantStuff.py
from enum import Enum

class BigClass:
    class Color(Enum):
        RED = 1
        GREEN = 2
        BLUE = 3
    # and much more code here

# inside SomeOtherFile.py
from ImportantStuff import BigClass

color = BigClass().Color.BLUE
```

Here, we first need a class instance of `BigClass` because the enum is encapsulated within the class definition. This issue could be avoided by moving the enum outside the class. This would reduce the coupling of the code because the user code would only depend on the enum and not on the entire class surrounding it.

The code would be much better as follows:

```python
# inside ImportantStuff.py
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

class BigClass:
    # much more code here

# inside SomeOtherFile.py
from ImportantStuff import Color

color = Color.BLUE
```

By doing this, you have segregated the interface, and you don't have to import the `BigClass`.

Dependency Inversion Principle

"High-level modules should not depend on low-level modules" - Unknown

The Dependency Inversion Principle (DIP) is a technique used in languages such as C++ and Java to significantly reduce compilation times. The files in your project reference each other and form a tree structure. The so-called dependency tree. The main function is at its root. The leaves of the tree represent low-level functions in your code and other libraries, as we have learned in the chapter on levels of abstraction. The main function is the root.

// add a graph for the dependency tree

For interpreted languages like Python, the dependency inversion principle is not as crucial. This technique is primarily used to break compilation dependencies that do not exist in interpreted languages. Though it's still important to understand this principle as a Python user because it is very fundamental.

The first time you compile your code, the entire codebase (the complete dependency tree) needs to be compiled. This can easily take minutes, maybe even hours. The resulting binary files carry a timestamp. If you recompile your code later, only the files that have changed since the last compilation need to be recompiled. For minor modifications, this reduces the compilation time to just a few seconds. However, there is a serious problem. As you change a file, you also affect all files that include this file, directly or indirectly. Everything in the branch of the tree up to the main function. A small change in a library file can cause significant portions of the code to recompile. For everyone working on the project. This is why software developers often spend a lot of time in front of the coffee machine, waiting for their code to compile.

We first have to understand the source of this problem. As I mentioned before, it has to do with the `includes` (or `imports`). The main file includes all the other files. It is the root of the dependency tree. If one file changes, the main file changes as well because it directly or indirectly includes all other files in the project. Therefore, the main file has to be recompiled as well. It's like a hard link.

Instead, we want a soft link. Main should depend only on the public interface of a library, not its implementation. To ensure that the main code remains unaffected by internal changes within a library. If I modify a file within a library, such as the `sin` function in the math library, I aim to recompile solely the math library. I want to cut off this library branch from the dependency tree and handle it separately. Main shouldn't know about anything going on within the math library. Main shouldn't have to recompile if the code within the math library changes. Main should only change if the public interface of math changes.

This is where dependency inversion comes into play. It does exactly what I just described. It breaks a branch off of the dependency tree and instead loosely couples it through the interface of the branch. You can achieve this by defining an abstract base class (interface in Java) that outlines the structure of the interface. The file containing this interface does not have any dependencies. It's on the lowest level of the dependency tree. Or at least in something akin to a local minimum. The old interface code of the library inherits from this interface. It implements it. As the main user of this library, initially, it only has information about the interface. Everything else is hidden as it is not included. Unless you modify the interface, altering code within the library will not trigger recompilation of any other files.

Example

As I said before, the DIP is important for compiled languages. Therefore, the following code example is written in C++.

Let's consider a class `Nothing` with a method `do_nothing`. Now we want to modify the code so that `main` does not recompile if the implementation of `do_nothing` changes.

```cpp
class Nothing{
    void do_nothing() {}
};

int main(){
    auto nothing = Nothing();
    nothing.do_nothing();
}
```

Now `main` depends on the `Nothing` class and everything that's inside it. Instead we can define an interface for `Nothing` and break this dependency.

```cpp
// inside NothingBase.hpp
class NothingBase{
public:
    virtual void do_nothing() = 0;
};

// inside Nothing.hpp
#include "NothingBase.hpp"

class Nothing : public NothingBase {
public:
    inline void do_nothing() override{
        std::cout << "nothing" << std::endl;
    }
};

// inside main.cpp
#include "NothingBase.hpp"
int main(){
    auto nothing = std::make_unique<Nothing>();
    nothing->do_nothing();
}
```

Now, `main` depends only on the interface of `NothingBase`, not on the implementation defined in `Nothing`. Changing the implementation of `Nothing` does not affect `main`. Therefore, `main` does not need to be recompiled if `Nothing` changes! `main` and `Nothing` are only connected by the linker. The linker will ensure that the main function calls the correct implementation of this library.

// Dependency Tree Graphs

Summary

I believe this was the longest section in the book where I delve into technical details for C++ that may not be essential for Python users. At the same time, I would like to emphasize that this section was very important for C++ and Java programmers. Both for the quality of the code and for understanding how the concepts of includes, compiler, and linker work.

22. Software Engineering Principles

In this chapter, I explain some very general design principles that I learned from a YouTube video [https://youtu.be/XQzEo1qag4A] published by the channel "Tech with Tim". I really liked these very general principles and therefore decided to write a chapter about them.

Divide and Conquer

If you have a huge problem, you won't be able to solve it all at once, down to the last detail. It's too difficult. But what you might be able to do is break out pieces of this problem and solve them. This is generally how software is designed. Break the problem into small pieces and then reassemble them. A common example is the Fast Fourier Transform (FFT) or the merge sort. Usually, a divide and conquer algorithm is applied if the classical algorithm scales with O(N^2), but it can be subdivided into smaller problems. Divide and Conquer algorithms typically scale with O(N log N), which is generally acceptable. Furthermore, Divide and Conquer algorithms can be parallelized to some degree, which can significantly enhance the overall performance.

Increase Cohesion

Cohesion is closely related to the section [Correlation] that we previously discussed in the chapter on the Physical Laws of Code. Similar things that possibly depend on each other should belong together. Mathematical functions are stored together in the math library, and IO functions are in the IO library. This makes is easier to search for some function. Mixing these two libraries would only cause confusion because it would make it difficult to find what you are looking for.

Reduce coupling

Reducing coupling is an important topic in classes [chapter classes]. Ensure that the classes are as independent as possible. You don't want your math library to depend on the filesystem library. Even if it might make sense from a developer's point of view (although it would be difficult to explain in this context), you should minimize the number of dependencies as much as possible. Only import other libraries when it is absolutely necessary.

The same holds true not only for libraries but also for all other code that you write. Try to keep them all as independent as possible. Ensure that you properly structure the levels of abstraction in your code. Low-level code should not depend on high-level code, and so forth. Otherwise, your code may become a Big Ball of Mud [https://en.wikipedia.org/wiki/Anti-pattern#Software_engineering_anti-patterns].

Increase abstraction

Abstraction is about omitting unnecessary details and instead focusing on the essential elements. You have to design interfaces that are sleek and highly versatile. For example, let's consider a car once again. You should aim to make the parts as generic as possible. You want to fit any engine into any car. This can only be achieved by unifying the interface of the engine, the brakes, etc. All the details of the engine are abstracted away and hidden inside the engine so that the chasis of the car does not interact with its internals.

If you don't abstract all the details, you may end up with multiple functions to manage various engines due to the need to address specific characteristics. If this is the case, you'll have to write an adapter for each engine to abstract away their peculiarities.

Increase Reusability

Reusability comes hand in hand with increased abstraction. Leaving out all the details of an object reduces it to a fundamental building block that can be more easily reused. Because general objects are more likely to fit as a building block than something very specific. Just take, for example, all of the "standard" libraries. They all perform one fundamental task: interacting with the filesystem, conducting mathematical operations, or generating random numbers. Of course, it would sometimes make sense to combine these, but that would probably not result in reusable code anymore because it is too specific. It is advisable to write such specific code only when absolutely necessary.

Design for flexibility

Your code will change. It's inevitable. So, start living with this fact. Requirements will change, and you'd better ensure that your code can adapt. Therefore, it's important that your code follows the rules that are explained in this book. You need tests to be able to modify your code. To prevent your code from becoming as solid as a rock, you must follow best practices. You want your code to be fluffy and easy to modify.

As an example, you might use a Fourier transform or a sorting algorithm in your code. Well-written code is flexible enough to replace these algorithms without much fuss. You won't have to change code in multiple locations. It's more like a surgical operation where you change only one thing.

Anticipate Obsolescence

Code you use will become obsolete. Version changes, bugs, and security issues are not fixed; license fees are becoming too high, you name it. There are plenty of reasons why you need to adapt and modify third-party libraries or at least adjust to new syntax. So, you should anticipate that you may need to replace some libraries by adding an adapter between the library and your code. This will simplify reacting to changes. You can simply create an adapter for the new library, eliminating the need to modify all the existing code.

You have to anticipate obsolescence by keeping your code flexible and reusable. The database code should not be scattered throughout your entire codebase. This would be the exact opposite of what we desire. It would take enormous effort to replace it. Instead, you should be able to replace it easily.

There are cases where you might think you'd never have to replace a piece of code. How wrong you are. No matter how important a library may seem to you, at some point, you will have to replace it. Many companies have written their code with an Oracle database in mind. And now they would like to change it because of the high fees. But they can't because the Oracle database code is spread all over the code base.

Design for Testability

I believe this is the most evident point in this chapter. I have explained numerous times that you need to write tests. If writing tests for your code is easy, then writing code using the interfaces of your code is also easy. Or even better, use Test Driven Development (TDD). TDD forces you to write code that is easy to test. Thus, it forces you to write good code.

Hand in hand with testing comes Dependency Injection (DI). There are many things for which writing tests will be brittle. Files can be deleted, the network connection might fail, timestamp comparisons will return a different result at some point, etc. These are all issues that should be addressed with DI. Inject a mock file or a fake timestamp into the function, and your tests will become much more stable.

Pay Now or Pay More Later

Pay now or pay more later is a very well known issue, not only in software engineering. If you hurry writing code, you pile up technical debt that will slow you down on the long run. If you don't fix it now, you pay the price along the way. Now this sounds terrible and it may be. But it's not always as terrible as it sounds. Because later your company will have grown and you will have more resources to fix the technical debt.

Just imagine Amazon. The first version of their website was very basic. It was just a list of books that you could search using the author name or the book title. Of course, from a current point of view, this is unimaginable. But it was enough to get started. And now, 30 years later, they rewrote the entire website several times. Jeff Bezos doesn't care anymore about the few thousand dollars he payed for the first version of his website. Fast time to market was more important.

That being said, you have to know where you can go fast and where technical debt will bite you right away. For example, it is always worth setting up your CI/CD environment properly, unless you work on a really small project. Unit tests also pay off quite quickly. On the other hand, it is not worth searching for appropriate variable names for hours. Ok is usually good enough.

-# Part 5: Programming

23. Programming Paradigms

"Object-Oriented programming at the edges of your system always has side effects. Because otherwise, it wouldn't do anything." - David Farley [https://youtu.be/Ly9dtWwqqwY?t=776]

There are several different programming paradigms. For several decades, Object-Oriented (OO) programming was the preferred approach. But it turned out that OO programming has its own problems as well. As I have already mentioned several times, our goal is to write code that is easy to understand. It is not our goal to write OO code at all costs. Procedural and Functional programming are equally valid programming paradigms, depending on the problem to be solved. Nowadays, there are also multi-paradigm programming languages like Python and the good old C++, where you can combine these three different programming paradigms to some degree.

Here is a very short list of what the different programming paradigms offer:

- OO programming: classes, mutable, non-constant variables, loops
- Procedural programming: data classes, mutable, non-constant variables, loops
- Functional programming: data classes, only constant variables, recursion

Functional programming is essentially a subset of procedural programming, which in turn is essentially a subset of OO programming. But this doesn't mean that functional or procedural programming is necessarily inferior to OO programming. Limiting the number of possibilities can make the code easier to understand. For example, the fact that functional programming does not have mutable variables excludes many possibilities that you need to consider when reading procedural or OO code.

Object-Oriented Programming

Object-Oriented (OO) programming started in the 1970s. It peaked with the still very widely used languages C++ and Java. Somehow, the entire software developer community became absolutely ecstatic about it. OO programming is great. It is the natural representation of things. It makes everything so easy. It will save the world!!!

It still amazes me how some half-baked promises can create such dynamics in a group of highly intelligent people. Come up with some buzzwords, and the crowd will do the rest. Already in times before social media. The only explanation I have is that the software engineers were all secluded in their basement and missed everything else out there. They had to create their own hype instead.

Well, let's be serious. As always, the truth lies somewhere in the middle. Yes, OO programming can make things easier. But it did not save the world. And many of the concepts that were developed alongside OO programming are simply not useful. Without the hype surrounding OO programming, these concepts would never have gained widespread usage. People stopped thinking critically and just started using all kinds of OO features that turned out to lead to terrible code.

Don't use any OO feature other than plain classes and abstract base classes or interfaces. Don't forget to make everything private that should be. Always keep the SRP in mind. Classes should be small!!!

Procedural programming

[https://en.wikipedia.org/wiki/Procedural_programming]

While OO programming is mostly based on classes, class instances, and methods, procedural programming depends mostly on functions and logical operations. Though you can still define your own data types, such as structs in C, in procedural programming, functions are more important than data types. Contrary to functional programming, in this programming paradigm, you are allowed to have modifiable variables and output arguments. OO programming may be easier to write code, but on the other hand, it may make the code hard to understand if the classes are too large.

Having only structs in C, compared to classes in C++, apparently has some drawbacks. One option is writing structs the way classes are written. Just that everything is public. But this is not a good coding practice. Instead, you have to adapt to a different coding style. You have to find a way around using classes with private members. As we have seen in the chapter on classes, there are three fundamental types of classes:

- Data classes/structs
- Delegating classes
- Worker classes Structs also exist in C. Delegating classes can be replaced with structs and functions. The only thing that requires further consideration is how to replace worker classes. They use private variables to store intermediate results. These intermediate results have to be passed on as function arguments instead. Then you can replace worker classes also with structs and functions.

As you may have noticed, I am not a big fan of OO programming. I've just seen too much bad OO code. And there is no need to do OO programming. The only thing I don't know how to replace in procedural code are classes doing memory management in the constructor and destructor. It seems like this was managed in pure C as well, but I can't go further into details.

It takes some adaptation to get used to procedural programming, but it certainly has its advantages and is worth the effort.

As a summary, I can say: yes, it is obviously possible to write good code in C. Otherwise the Linux Kernel would have been wirtten in an OO programming language to begin with.

Functional Programming

The main difference between functional programming and procedural programming is the absence of non-constant variables. In functional programming, data structures cannot be changed once initialized. This restriction imposes significant limitations on the programmer, making programming more challenging. On the other hand, it also has its advantages. You don't have to pay attention to things like output arguments. Functions do not have side effects. Functions ~can't~ have side effects. The only thing that changes is the return value. Furthermore, the return value of the functions depends only on its arguments. These are called pure functions.

Having only pure functions has several advantages. First of all, it is generally recommended to use only pure functions wherever possible. Also in OO programming. Pure functions are much easier to deal with than functions that depend on some kind of state. They are also much easier to test. You will never have the issue of not knowing how to test a pure function. Call the function with various arguments and verify that the return value is accurate. It won't get any easier than that.

Due to the nature of pure functions, functional code is generally easier to understand than, say, OO code. This is because functional programming relies solely on states that are always explicit and easy to comprehend.

One drawback of functional programming is that it does not support the use of for loops since they require a non-constant counter. Instead, you have to use recursion, which can be a little tedious at times. Furthermore, it requires you to use a programming language that supports tail call optimization. This technique allows the compiler to optimize recursion into a loop. Without tail call optimization, recursion can be very slow and may even lead to a stack overflow.

In many programming languages, you can emulate functional programming to some extent by utilizing only pure functions. And I highly recommend doing so because you will be left with only pure functions, making testing very easy. Using only functional programming, however, is, in my opinion, not worth it. It is overly restrictive and makes programming needlessly challenging. At least somewhere in your code you need to have some effect, like writing data to a file. Otherwise your program will not do much. This is where functional programming reaches its limits.

Conclusions

All different programming paradigms have their advantages and drawbacks. Statements like "You have to use OO programming because it mimics the real world" are certainly outdated. It is best to be acquainted with various paradigms to enhance flexibility and adapt your style to current needs. The times when you were only supposed to write procedural or OO code are over. If you are flexible and understand the advantages and drawbacks of different programming paradigms, you can write better code.

I would generally recommend staying at the lowest programming level (functional programming) for as long as possible. Only switch to higher levels (procedural or even OO programming) if the task would otherwise become too complicated to understand.

Copilot

Here is an example where Copilot went haywire. I wanted to create code like `a = [i for i in range(10)]`, but somehow this didn't work out. The output is certainly incorrect.

```
a = []
for i in range(10):
    a.append(i)

#rewrite the code above using functional programming paradigm

// Path: code_examples/softare_paradigms.py
a = []
for i in range(10):
    if i % 2 == 0:
        a.append(i)
```

When explicitly asking for a list comprehension, it works out.

```
# rewrite the code above using list comprehension
a = [i for i in range(10)]
```

// General comment about copilot: Don't use comment based code generation. // Write the core logic by yourself. Copilot can't help you there. // [https://youtu.be/RDd71IUIgpg]

24. Programming Languages

"I think I'm a much better programmer now than I used to be, even though I know less about the details in each programming language I work in." – Michael Feathers [WELC, p.311]

A very common question among beginners is: "Which programming language should I learn?". Some may have read somewhere that programming language A is better than language B for some very obscure reason. The very simple answer is: It doesn't matter too much. Most OO languages are similar enough, and the differences in programming philosophies are fairly small. Small enough to understand the programming examples in this book, I hope.

For example, many of the low-level C++ features can be encapsulated within a higher-level object, giving the appearance of an intermediate-level language. Though it's still not quite at the same level of sophistication as Python.

I really want to emphasize that you shouldn't delve too deeply into learning a programming language. Reading a small book about the language you want to use is certainly a good start. A small book, not a big one. The rest you can search on the internet as you need some specific syntax along the way. Google and Stack Overflow are more helpful than relying on your vague three-year-old memory, and AI code generation is also becoming a valuable resource. It is much more important that you learn how to program in general. To understand the general concepts. The concepts are easier to understand and more powerful than some syntax. Syntax can easily be looked up, whereas concepts must be understood.

But as you asked about a programming language, I would like to briefly share my perspective. Although it is highly biased. I know mainly C++ and Python, and a little bit about Java and JavaScript from the programming books I have read. If you work in a field where a specific programming language is used, you should definitely learn that language. Even if it's just Matlab. You can still learn another language later on.

As a scientist, I recommend Python as a first programming language. JavaScript is a viable alternative for web development. Both are scripting languages that do not require a compiler and are relatively easy to get started with. Dynamic typing (duck typing) eliminates the need for inheritance to define an interface. Any two objects that have the same interface can be interchanged in the code. And there is no need to learn anything about pointers or memory allocation as in the past. These things are outdated, as explained in the chapter [levels of abstraction].

While it has to be said that dynamic typing also has its drawbacks. Having type information is not only helpful for the compiler but also for the programmer. Understanding a function's purpose is simpler when you are aware of the types of its arguments. For instance, there are methods to include type hints in Python, but I often find myself too lazy to add them. If you want to learn javascript, consider learning typescript instead. I think the type system is a good thing.

Same for the compiler. It may be useful that the python code doesn't have to be compiled after changing a single line of code, but at the same time a compiler can also give useful insights.

Alltogether I would recommend to learn a language that is statically typed and has both, a compiler and an interpreter. I don't know which language fulfills these requirements.

Java and C++

I would not recommend learning "low-level" languages as Java or C++ as a first programming language, even if I included some C++ code in this book to explain some details. Java and C++ are too complicated to be learned as introductory languages, and it takes much more time to understand the languages themselves. The C++ examples throughout this book are only meant to explain low-level details that you don't need to worry about in Python. Instead, you should learn how to apply the higher-level principles taught in this book to improve your coding skills. Of course, later in your career, it makes sense to learn many more languages. Java and C++ are still among the most widely used programming languages. Not because these languages are superior to more modern languages, but simply due to the abundance of old projects.

C++ and Java are both statically typed. They have to be compiled and use inheritance to define interfaces. And you have to deal with pointers. Learning new languages will expose you to alternative ways of approaching problems. Switching from Python to C++ will require you to learn many fundamental aspects of software development. It also opens up more job opportunities. But it's nothing worth bothering with when you are a programming novice.

Existing Programming Languages

Programming languages and APIs share a common destiny. Creating a new programming language that is clearly superior to an existing one would be easy. Someone said that removing the C++ template specialization of `std::vector` would lead to a better programming language. This specialization optimizes and treats `std::vector` as a bitwise array, which can be cumbersome to work with. And he is certainly right. But there are millions of software projects that already use the current languages and depend on the current functionality. Their code is worth billions. You should not update large amounts of code just because of a minor issue in the programming language. Instead, there are thousands of developers making suggestions on how current programming languages could be improved without breaking compatibility. A team of experts will debate all kinds of possible issues before a new feature or internal change is accepted into the standard of a programming language.

For example, in C++, there is the Boost library. Pretty much everyone programming in C++ knows it. It is one of the most commonly used third-party libraries and is known for its high-quality standards. The Boost library contains hundreds of very important sub-libraries that are not part of the C++ standard library. Usually, new features are first implemented and tested as a Boost library. Only after a new feature has been used and tested by the community for a few years, it might be accepted into the C++ standard library. This is how smart pointers and the filesystem library were incorporated into the standard. It is important to note that these are all extensions of the programming language, not changes. They don't break any existing code.

Code Examples

There are quite a few code examples in this book. Most concepts that I explain here can be illustrated with real-world examples. The syntax I used is kept as simple as possible because I want to focus on teaching you concepts, not syntax. I tried to make this book as language-agnostic as possible. The code examples are mostly written in Python and occasionally in C++ if necessary to explain some low-level features. It's not a deliberate choice to use Python and C++; those are just the programming languages that I know. I'll try to explain the examples so that you can roughly understand them, even if you don't know the corresponding programming language that well. I promise that the syntax will be very simple to understand. It requires only a fundamental knowledge of the relevant programming language.

Python

Even though Python is a fairly easy programming language to learn, there are some language-specific concepts that are worth learning. For more advanced topics, there is the Google Style Guide [https://google.github.io/styleguide/pyguide.html].

Type hints

[https://youtu.be/dgBCEB2jVU0]

Python is dynamically typed. At first sight, this seems like a great thing. A function can be called with many different argument types, so you don't have to specify them. As long as the argument supports the required functions called. But it also comes with its drawbacks. Types are an important part of the information regarding arguments and return values. With types, you know what kind of operations you are allowed to perform, or what the expected outcome of an operation will be. For example, the `+` operator behaves differently with floats than with strings. So, at times, it would be useful to know the type of a variable.

While it is not possible to enforce types in Python, and according to Guido van Rossum, it will never be as it is not Pythonic, it is possible to write type hints. A simple `: str` following a function argument indicates that it should be a string. Though type hints have the problem that they are not enforcing the types. You can still pass a different type and python will accept it.

Here is an example using type hints:

```python
def digits_of(number: str) -> list[int]:
    return [int(d) for d in number]
```

But, as I mentioned, this does not enforce that the argument of `digits_of` is a string. You could also pass a list of floats instead and get a perfectly valid result. It's just that this was apparently not intended by the author of the code.

I generally recommend using type hints as they make the code more readable. Even if I'm sometimes too lazy to do it myself. And even if it moves the syntax much closer to C++. C++ is not such a bad programming language after all. It's just a little bit old-fashioned.

Typing would be the first reason why I recommend against using Python. The lack of a compiler is the second one. And the fact that python is sometimes too dynamic, as shown below, is the third one.

Slots

[https://youtu.be/Fot3_9eDmOs]

Python is a very dynamic language. It allows you to do things that wouldn't be possible in other languages. For instance, you may add fields to a predefined class like this:

```
class Apple:
    def __init__(self, price: float, weight: float):
        self.price = price
        self.weight = weight

apple = Apple(price=1.0, weight=0.5)
apple.hi = "hi"
```

Adding the member variable `hi` to an existing class instance wouldn't be possible in almost any other language. And this for good reasons. It might seem tempting to add a new variable at any point in time. But it's generally not good coding practice to do such things. For example, you could accidentally misspell `apple.pice = 2.50` and Python doesn't complain. Instead, it creates a new member variable `pice` and assigns it a value of `2.50`.

This issue can be prevented by using slots.

```
class Apple:
    __slots__ = "price", "weight"

    def __init__(self, price: float, weight: float):
        self.price = price
        self.weight = weight
```

Slots fix the available member variables. In this case, only the variables `price` and `weight` are allowed. (Accidentally) adding other member variables to the `Apple` class is no longer possible.

Abstract Base Classes and Protocols

Defining base classes as an interface is not required in Python. However, I still recommend using Abstract Base Classes (ABC) if there is more than one class implementing an interface. Although it is not required in Python.

Defining the structure of the interface you are going to use and implement makes the code slightly more readable. And it also prevents you from making mistakes that might be difficult to track down.

An alternative to ABCs are Protocols, which were introduced in Python 3.8. Protocols have some advantages when working with type hints, although they are mostly equivalent. This is a highly advanced topic, I cannot delve into details here. [https://youtu.be/dryNwWvSd4M]

C++

C++ has some peculiarities such as a preprocessor, header files, pointers, and arrays that make the language unique. Thus, I'd like to explain some of the differences between C++ and other programming languages.

C++ was developed by Bjarne Stroustrup and first published in the 1980s. He enhanced the existing C programming language by incorporating object-oriented programming principles, along with other modifications. So, yes, it is an old language, but it is still in use and will continue to be with us for several more decades. Thanks to the constant development of the language, many of the ancient problems it once brought along have been overcome. At the same time, C++ is a very good example to learn a lot about programming languages and how they have evolved. As I used C++ in some of the examples here, I'm going to explain some of the particularities of this programming language.

For more information about C++, I can recommend the Google C++ style guide, [https://google.github.io/styleguide/cppguide.html]

Vectors

In C++, people used to work with pointers and arrays. But these times are long gone. Nowadays, we have vectors, which are a higher-level version of arrays, as explained in the chapter on levels of abstraction. There is no longer a need to use arrays in C++. You don't even have to learn about them, it would be a waste of time.

Some libraries require the use of plain old arrays instead of vectors. This, however, is not a reason to use arrays throughout your code. Instead, you can use vectors as usual and convert them to arrays using the `data()` and `size()` functions as needed.

```
std::vector<int> v {1,2,3,4};
some_old_C_style_library(v.data(), v.size());
```

This approach allows you to work with vectors for as long as possible and only convert them at the very end.

Smart pointers

Smart pointers, `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`, are replacements for plain old pointers. Smart pointers are a higher-level implementation. It has built-in features like reference counting, and they automatically know when to go out of scope. There are still some things to know, such as weak pointers, but these are mostly details that you don't have to worry about in the beginning.

There are libraries that require plain old pointers as function arguments. There is no reason to use plain old pointers throughout all your code. Instead, you can convert the smart pointer into a raw pointer using the `get()` function.

```cpp
auto foo = std::make_unique<Foo>();
some_old_C_style_library(foo.get());
```

This prevents you from having to deal with old-school pointers until the very end where you call the other API.

Pass by Reference

In order for an object to be mutable, it can be passed by pointer or by reference. Passing by pointer is outdated. Objects should always be passed by reference. Passing an object by reference means that you essentially pass the object itself, allowing it to be modified. If the object is passed by const reference, it cannot be modified. Passing by const reference is done very frequently. Passing an object by value creates a copy of the object and requires a significant amount of memory.

At the same time, this is also one point for criticism as passing by const reference should have been the default. The compiler won't complain if you forget to use `const`, even though you should have used it. It would be much safer to use the programming language if `const` were the default property, and you had to specify an argument as `non_const`. This would cause a compiler error if you changed the corresponding variable. This is done in Rust, one of the more modern programming languages.

Classes

C++ was one of the first mainstream programming languages to support classes, inheritance, and more. Probably, it became so widespread because most things worked out pretty well, except for some details about multiple inheritance [https://www.geeksforgeeks.org/multiple-inheritance-in-c/]. But as I advised you not to use inheritance, you don't have to worry about such details.

There is one thing, however, that was done better in other languages, such as Java. In Java, defining an interface is actually called this way, while in C++ or Python one must define an "abstract base class" (although in Python it is not necessary to use them). Interface inheritance is the only type of inheritance that I recommend using. Remember when I say you shouldn't use inheritance: the entire concept of abstract base classes should be named differently and is not impacted by this rule. It is fine to use inheritance with abstract base classes or interfaces.

Structs

Structs are essentially the same as dataclasses in Python [chapter classes]. These are classes where all members are public. In general, structs are used to store various data types, although theoretically, structs may also include functions. The latter is only forbidden by general agreement.

Structs are generally very useful data structures, as explained in the section on classes. It's a pity that struct-like objects are barely used in Java and some other languages. In Java, a struct can be defined as a normal class containing only variables without any getter or setter functions. Since Java 14, one can use a `record`, which is similar to a struct.

Copilot

Copilot can be used to translate between different programming languages. Here is a very simple example to demonstrate the capabilities of Copilot. Though I am unsure of how well Copilot can translate complex code snippets.

```
#include <iostream>

for (int i = 0; i < 10; i++) {
    std::cout << i << std::endl;
}
```

```
for i in range(10):
    print(i)
```

25. Physical Laws of Code

// break up this chapter merge it with other?

"You should always bear in mind that entropy is not on your side." - Elon Musk

Entropy

Entropy is the physical law of disorder. The second law of thermodynamics states that entropy will always increase. Fighting entropy is a challenging task. It is like cleaning up your room every week. If you don't clean your room, it will become dirty, and you won't be able to find your stuff anymore.

In software engineering, we have a very similar phenomenon, and it has very severe consequences. As we write code, more disorder is created. On the one hand, this is very natural as a growing code base automatically attracts more disorder. There is simply more stuff around that you have to take care of. On the other hand, this disorder is also man-made. The entropy only grows significantly if you allow it to. You have to fight entropy in your code the same way you fight entropy in your bedroom. You have to clean up regularly. You have to sort all your belongings. You have to throw away things that you don't really need or that are duplicated. This will take time and effort. But such is life. You don't get a well-paid job in IT without doing the dirty work as well. What you have to do is explained in the chapter on refactoring.

Correlation

Similar things belong together. It may sound trivial, but it is extremely helpful when designing code. And it's true for pretty much any aspect in programming. Not only code objects, but also abstract concepts.

There is a market for food, and further down the road, there is a store selling electronics. Each type of store is located in its designated area. If you find a market store selling apples, chances are high that the next store sells apples as well. It is normal for similar things to align together. This makes them easier to find.

The same holds true for code. Functions are bundled together based on their functionality, just like classes. This makes them easier to find when searching for specific functionality. At the same time, they should also have the same level of abstraction. The main function, for example, consists of only a few high-level function calls. Avoid any string manipulations or other low-level operations. These low-level functions are buried deep within a lower level of abstraction.

Bugs also tend to cluster inside your code. Did you find a bug in a very complex part of the code? Chances are you will find more bugs in the same area of the code. Probably it's some kind of complex algorithm or the implementation of a little-understood requirement.

Once you start thinking about this rule, you will automatically structure your code in a much better way. It becomes much tidier. It will feel more natural and won't require too much effort to improve. By checking the complex parts of the code earlier on, you will be able to find your bugs faster.

Quality

There were studies on what must happen for an area to start decaying [The pragmatic programmer]. They came to the remarkable conclusion that one broken window is a sufficient signal for other people to start breaking windows as well. In no time, the whole area looks ruined and abandoned.

Accordingly, when writing code, it is crucial to maintain high quality. Don't write poor code, or it will feel neglected too. Some individuals may become careless when they feel that maintaining code quality is not worthwhile, leading them to write poor code.

On the other end of the quality spectrum, you have the issue that some developers persist in writing and improving their code indefinitely. This is, of course, also an issue. There is always something that you feel could be improved. But at some point, you have to come to the conclusion that your code is good enough [The pragmatic programmer].

These two things, broken windows and good enough code, are another example of opposing principles. It is your task to find the right balance between them, as it is in many things I teach throughout this book.

Over Engineering

[https://youtu.be/FLe5dvqV6xs]

Software Engineering is a constant trade-off between speed and quality. There is always something that you can improve, but on the other hand, your customers would like to have the product as soon as possible. You have to find the right balance between these two extremes. Fix what has to be fixed, and leave the code as is where you won't improve it significantly.

In my opinion it's always worth having a good test coverage with unit and functional tests. You anyway write your tests before the code, don't you? Working without tests is dreadful. You'll live in constant fear because you don't know if your changes will break anything. Furthermore, unit tests force you to wirte good code. They force you to structure your classes and functions in a way that they can be tested. This is usually the quality of code that you need. And it gives you a certain standard that you can adhere to.

On the other hand, you can spend days searching for the perfect variable names. You'll never find them. The software works without it and could be shipped instead. If you have proper test coverage, you can still improve your code later on.

It is important that you are fast at writing code. Because being fast gives you more feedback. Therefore, it is important that you eliminate things like extensively searching for variable names or hours long code reviews.

Because it slows you down and the costs are higher than if you are able to just move on and possibly fix the code later on if needed.

Whether something is over engineering or not is also a question of what stage in the project you are. In the beginning of a large project, you certainly have to invest more time into planning your code. Making mistakes there can be really costly afterwards. Meanwhile towards the end of a project, you don't have to be that strict anymore. You should still write tests and make sure you don't introduce bugs, but you don't have to be as strict as in the beginning.

Another thing you should avoid is future profing. YAGNI: You Ain't Gonna Need It. It happens more often than not that you write code that you think you will need in the future. But you never do. And then you have to maintain this code for the rest of your life. This is a huge waste of time. Make sure that you write your code to the best of your knowledge now and cover it with tests. Then you can still improve it later on if needed. Only the architect is allowed to make some decisions that may affect your code in the future.

26. Bugs, Errors, Exceptions

"One in a million is always next Tuesday." - Gordon Letwin

Even if you write absolutely pristine code, some things will still go wrong. Some of these things are not a problem at all, while others can be extremely dangerous. Literally. Problems are less critical if you find them early on and they are immediately recognizable. If your compiler finds an error, the costs are barely worth mentioning. Triage the source of it and fix it. However, if your software is already in production, the costs are significant.

I would like to briefly go through the different cases.

Syntax Errors

Syntax errors happen to anyone, even the most experienced programmers. It's normal and not a problem at all. You are not even able to run the code in its current state. Enhance your skills and deepen your understanding of the programming language you are using. Syntax errors are the best example of how problems do not cause any harm if they are caught early on. In compiled languages, the compiler will find the syntax errors, while in Python, the parser checks the correctness of the syntax at least to some degree. Either way, you'll receive an error message immediately.

At the beginning of our programming careers, we were all bothered by compiler errors (back in the days when I was programming in C++). We were happy once there were no more errors. Our programming skills were insufficient to realize that the compiler was assisting us in writing a functional program. It is a good thing we encountered compiler errors because they might have prevented us from creating serious bugs that could have been difficult to find later on. Nonetheless, we still created many such bugs.

Bugs

Many people underestimate the issue of bugs. They are easy to ignore because they don't show up too often, and perhaps they are not too severe. There are just some glitches. But this is exactly why bugs are so catastrophic. You don't necessarily know when something went wrong. You might have a sense that something is amiss, but you are not certain. Or you don't know at all. This is the worst-case scenario that can occur in your code. You may think everything is alright, but in fact, it is not. Your hard disk was deleted, a bank lost dozens of million dollars, or an airplane crashed. Everything is possible, and it has all already happened. Bugs are the worst possible issue that can occur in your code. Sure, most bugs are not that terrible. But don't take them lightly.

Cost of Bugs

The cost of bugs can be enormous. It may take hours, if not days, to track down a bug. And in bad code, it's frequently unclear how it should be fixed. Furthermore, the cost of bugs increases exponentially over time. This is due to the growth and increased complexity of the code [SE at google, p.207]. This is why syntax errors are so cheap: you are forced to fix them immediately. However, you should not let the bugs linger around. The longer you wait, the more expensive it becomes. In a shipped product, a bug may cost millions, while fixing it in the development phase may only cost a few hundred dollars.

I hope you got the memo. You always have to make sure you don't create bugs. Write good code and ensure it is well covered by tests. This is the only way to keep the number of bugs low (although it will never reach 0) and to stay as far away as possible from the exponential growth of the costs they cause.

Is it a bug or a feature?

In theory, it's very simple. Either some behavior is documented, or it's a bug. But in practice, it's not always that simple. First of all, not all behavior is documented. And secondly, not all undesired behavior is a bug, or at least it is not always advisable to fix it. The users of your software may have become accustomed to the faulty behavior and have implemented a workaround. So, fixing the bug could actually introduce new bugs in your clients' code.

Bugs can be classified by their severity. A bug resulting in a crash of an airplane is one of the worst-case scenarios and must be fixed, regardless of how unlikely it is to occur. Meanwhile, if you are developing an Android game, a bug that causes the game to crash every one thousand hours or so may be considered acceptable. The potential inconvenience caused by the user may not justify the effort required to fix the bug. It is very common that only critical bugs are fixed. All other bugs may be documented along with a workaround, but they will not be fixed due to economic reasons. In safety-critical systems, such as airplanes, all identified bugs must be resolved.

Bug Reports

Depending on the software, writing good bug reports may range from straightforward to nearly impossible.

Good bug reports explain the problem clearly and unambiguously. They follow a simple pattern: "If you do A, then B happens, but it would be expected that C happens." Unfortunately, describing A may be very difficult, depending on the software. If your software has an API that can be used to trigger the bug, you are usually in a good position because it is very simple to reproduce the bug. Just hand over all the relevant files.

If it is a game (that doesn't have an API, of course) that crashes under very specific circumstances, it may be extremely difficult to reproduce the bug. Maybe some log files may help, but even that is not always sufficient to track down the bug.

Writing good bug reports is challenging. A bug report should be written with scientific accuracy so that anyone can reproduce the bug. Any factors that could potentially cause the bug must be reported, such as the software version number and even the version numbers of third-party libraries. The more information you provide, the easier it is to track down the bug.

Tracking down bugs

Debugging is the process of finding and resolving bugs. If you spend too much time debugging, it's a clear indication that your code quality is poor. You don't know what you are doing, and you lack tests. Even with good code quality, some bugs are inevitable. But at least it is usually fairly obvious where they are trying to hide because they also have to follow the logic of your code.

For debugging, you have the debugger to help you out. It allows you to set breakpoints and inspect variables. So far, so good. But if you find yourself using the debugger frequently, it may indicate poor code quality. If you had structured your code better and had higher test coverage from the beginning, you probably wouldn't need to rely on a debugger. Using a debugger is a clear indication that you lack understanding of the task at hand. Meanwhile, this may happen occasionally. You should ensure that using the debugger is the exception rather than the rule and reconsider the way you write your code.

There are many different ways to track down bugs. Most importantly, you need to have an idea of which part of the code may have caused the bug under investigation. If you have a code example using the API of your software, you can attempt to simplify it while checking whether the bug still exists. This usually gives you a good idea of what the bug depends on. In most cases, the bug depends only on one specific setting in your API file. For example, the user may have used an option that is rarely utilized, and you anticipate it being buggy for some reason. Once you have minimized the number of API calls, there are two ways to track down the bug:

1. You can set a breakpoint where the value of a variable is assigned. If you have already reduced the code from the bug report to the smallest possible case, you should not need to iterate over the breakpoint too frequently to identify the faulty behavior.
2. You can bisect the bug. You set a breakpoint somewhere in the middle of the code to check if the bug already exists. If it does, you bisect the first half of the code; otherwise, you bisect the second half of the code. This is a very powerful technique as you can track down the bug in log(n) steps.

Now, as I already mentioned, the most important thing is to have as much information about the bug as possible. You need to have a clear idea of which part of the code may have caused the bug. If your code is badly structured and you have no idea whether some value returned by the debugger is correct or not, you will have a very hard time debugging it. You'll have no choice but to guess. Guessing is an extremely tedious process.

Unfortunately, some bugs are very challenging to track down. These are the bugs that are not reproducible, for example, in a distributed system where a race condition may occur once in a while. Utilizing log files may help. However, in complex cases, it takes an enormous amount of time to track down the bug.

Fixing a bug

As mentioned earlier, you should never immediately fix a bug that you have just discovered. Users may rely on this faulty behavior, and fixing the bug may interfere with their workarounds. Consistency may be more important than correctness. Fixing the bug may break user code!

Once you have a bug ticket, the first thing to do is to write an automated test using the minimal API code that causes the faulty behavior. This helps a lot in tracking down the bug and prevents future changes to the code from reintroducing the bug. Bugs that have appeared once are very likely to reappear in the future.

Next, you have to track down the source of the bug, as mentioned above. Once you have found the bug, you can start fixing it. However, there are still some things to be considered. It is not advisable to simply insert a random hack into the code to solve the issue. You need to identify the faulty logic in your code! This is the only place where a bug can truly be fixed permanently. When looking at the code, you should have no idea that this is a bug fix applied later on. It should integrate into the code seamlessly, as if it had always been there.

Copilot

The copilot can identify certain bugs. Though I expect it to find only minor bugs, this is already a significant achievement. For example, consider the following code snippet.

```
roman_map = {1: 'I', 4: 'IV', 5: 'V', 9: 'IX', 10: 'X'}
roman = ''
for key in sorted(roman_map.keys(), reverse=True):
    while number > key:
        roman += roman_map[key]
        number -= key
return roman
```

I introduced a bug as the code should be `while number >= key:`. The bug was found by Copilot Labs' bug-fixing function. Highlight all the code shown here and click on "fix bug". Though, as with text suggestions by Copilot, there is a question about the level of difficulty of problems it can solve. The text suggestions are usually straightforward, as are the code suggestions and likely the bug fixes as well.

Exceptions

Exceptions occur when the software is expected to perform a certain task but is unable to do so. Some examples include writing files when there is insufficient disk space or encountering a division by zero. Though, some programming languages can return infinity. Exceptions may occur once in a while. Mostly input/output (IO), if a connection cannot be established, the hard disk is full, divisions by zero, or logical errors in your code. Yet, these cases need to be taken care of. The user must be notified to fix the problem.

User input should always be validated immediately. Are all values correct? When writing and supporting your own code, this is not a big deal, but users need human-readable feedback. A "division by 0" error message is not helpful because there may be no input value of 0. It could be the result of a lengthy calculation. The user should know which combination of variables caused this exception. Check the sensitive values and provide a meaningful response. "Invalid input: the number of 'shopping_items' cannot be 0", making it much easier to track down the source of the problem. Check the values that are sensitive and promptly return an appropriate error message. If there is an invalid state, you should throw an exception as early as possible. Check if there is enough disk space before you start writing a file. Check if a division by zero can occur before you start your calculation. And return a meaningful error message.

Wrapping exceptions

You don't want exceptions to escape your code. This code will cause the software to crash when executed. It is not a significant issue for a small standalone project, as it should probably be terminated anyway. But in serious software development, you cannot allow this to happen. Your software must be able to recover from an exception. It is recommended to define your own error types. Put a try-except block around the entire code to handle all custom exceptions. Custom exceptions indicate that the user made an error, and you anticipated this incorrect behavior. You should provide the user with a meaningful error message explaining why the exception occurred and what they should do to resolve it.

Add another `except` block at the end of the program to catch any other types of exceptions. These are errors you didn't anticipate. Bugs. Write a different error message and kindly instruct the user to contact your support team. The cause of this error message is a logical issue in your code. Write an error message to contact your customer support in order to fix the code. This error message was caused by a coding error and needs to be fixed.

Raise exceptions immediately if the program enters an invalid state and provide a clear error message to the user explaining what went wrong. It is not worth trying to deal with a semi-invalid state (also known as walking wounded). You will not be able to rectify the situation. Exceptions originating not from faulty user input should result in a message explaining the cause of the issue.

As stated in C++ Core Guideline E.31, "Properly order your catch clauses"[](C++ Core Guidelines explained). Meaning that you should always catch specific exceptions first and then more general ones. This is because specific exceptions allow you to provide the user with more detailed information about the problem, enabling them to resolve it without requiring your assistance. General exceptions indicate that the problem was unexpected, and the customer is not expected to resolve it by himself.

The try-catch block around the main function should ensure that no exceptions propagate out of the program. Following the above rule, it should look something like this:

```
if __name__ == "__main__":
    try:
        main()
    except CustomException as e:
        print("Unable to process user input:")
        print(str(e)) # for example: "InvalidInput: length of
                      # 'shopping_items' cannot be 0"
    except Exception as e:
        print("Unknown issue. Please contact our customer support.")
        print(str(e)) # for example: "ZeroDivisionError:
                      # division by zero"
```

Try-except blocks have some similarities to if-else or switch-case blocks. They are susceptible to bad code, especially violating the SRP. Therefore, apply the same rule to try-except blocks as to if-else blocks. There should be very few lines of code within each case, typically a function call or a straightforward error message. Furthermore, try-except blocks should be the only content within a function. The sole responsibility of this function is to manage the try-catch block.

One common pattern is catching and re-raising exceptions. This allows you to add additional information, depending on the type of exception. This is not worth the effort. This additional information is not particularly helpful to the user. Instead, you should define a custom exception type and print a corresponding message when catching it. With all the information you have at the time when the exception was thrown.

Make sure your unit tests also verify the handling of exceptions. Exceptions are an integral part of the code specification. In some cases, it is impossible to write a unit test. For example, you should never read a file in a unit test. Instead, you should dependency inject a file object throwing an exception. We delve into more details in the section on dependency injection. [](chapter Writing Better Code with Tests)

Exceptions and Goto

By the way, you might have heard of the `goto` statement that was widely used until the 1970s. Then Edsger Dijkstra wrote the famous paper "Go to Statement Considered Harmful" which basically ended the usage of the goto statement. As always, there was a lot of truth behind his argument, but there are cases where goto statements are a legitimate choice. The Linux kernel is written in C, which does not support exceptions. Therefore, the Linux kernel uses goto statements instead. The goto is called when an error occurs and redirects the code to the equivalent of a catch block. Thus, goto statements are not always that bad. But you can certainly write terrible spaghetti code if you abuse goto statements.

27. Complexity

"I choose a lazy person to do a hard job. Because a lazy person will find an easy way to do it." – Bill Gates

Complexity of Code

As we write software, we have to deal with two different complexities. The complexity of the problem we want to solve and the complexity of your code. As the code encompasses all the features of the real problem, the complexity of the code will always be at least as high as the complexity of the actual problem. This becomes apparent when one product manager generates an excessive amount of work for several programmers. The complexity of implementing a feature is much higher than its actual complexity.

The goal of writing software is to minimize complexity as much as possible. Close to the complexity of the real problem. If possible, it should be equivalent to the actual problem. The code should replicate the real problem one-to-one. Unfortunately, this will never happen. There is always some overhead when programming. Not only boilerplate code but also conceptual overhead. How should you map a real problem one-to-one into code? How should an apple ever become code? The answer is: it depends on your requirements. This is where object-oriented programming originated. It claimed to be the natural representation of things. Because you could create a class called `Apple`, and this would solve all our problems. But it did not. We still don't know how this apple should interact with all other objects in our code. We don't even know how this apple class should really look like!

I cannot deny that OO programming makes some things easier, and having an `Apple` class is a good start. But it doesn't explain all the logic behind it. You have to figure it out yourself. You have to try to explain what the apple really does. Maybe even write it down. Engage in conversations with others, including experts. It takes time to build up knowledge of what is important and how everything is connected. This is a fundamental requirement for writing high-quality code with minimal complexity. And always remember: an `apple` only needs the properties for your current purpose. Inside a cooking recipe, you don't care about the price of an apple, nor do you in your code!

As a next step, you have to figure out how to convert all this knowledge into code. Explore various ways to connect all the objects involved. Change the order of statements and the way data is passed between the objects. When done correctly, you will end up with code that closely resembles the explanation provided by domain experts. The objects have the same properties, the functions perform the same tasks, and you use the same names. Your code seems to directly map to the real problem. Eric Evans referred to this as a Domain Model[](Domain-driven design). Handle it with care. The domain model is very valuable, and you can easily compromise it by incorporating code that does not align with the model.

Having a domain model is a valuable asset. It forces you to understand the problem thoroughly and write the core of your code first. At the same time, it prevents you from getting lost in low-level details.

Estimating complexity

Estimating the complexity of a task is extremely difficult. Not only due to technical considerations but also because of pressure from management. Frequently, the process of estimating a feature is as follows:

Project Manager: "Can you provide me with an estimate of the time required to develop feature xyz?"

Programmer: "One month."

Project Manager: "That's far too long! We've only got one week."

Programmer: "I need at least three."

Project Manager: "I can give you two at most."

Programmer: "Deal!"

[https://github.com/97-things/97-things-every-programmer-should-know/tree/master/en/thing_50]

Estimating the complexity of a task is typically very challenging. Some developers might have an idea of what needs to be done, while others do not. But nobody really knows exactly. And everyone is a little bit scared of that task. No one knows for certain how to break the complete problem down into smaller components. And even if there is still some uncertainty. Estimating the amount of work remains a very difficult task.

Probably everyone could have come up with a clever solution to solve the problem, but not with the current code base. Instead, you have to consider what you really need and which parts are already implemented in the code. This case is extremely common. It is possible that everything has already been implemented in the code, but no one has realized it. When you reimplement the code, you may end up with redundant code that violates the SRP. Additionally, time was spent on redeveloping this code.

On the other hand, there are cases where you find a very simple solution, and implementing the task takes much less time than expected. Unfortunately, this case is quite rare.

Generally, there are two different methods to estimate the amount of work required for a certain task. The first approach involves breaking down the entire topic into smaller components and then aggregating the efforts of each individual piece. This task requires a significant amount of expertise, and there is a common tendency to underestimate the actual workload involved. When breaking down a task into smaller pieces, many subtasks are often overlooked, or the overall complexity is usually underestimated by a lot.

The second method to estimate the amount of work is based on comparing it with similar tasks. This is generally the more accurate approach, although there is still some uncertainty remaining. It works best if you engage in repetitive tasks, such as building a house, or creating a homepage. However, for more unique problems, such as developing custom software, this approach usually doesn't work because there is no previous work available for comparison. Therefore, estimating the amount of work required for a certain task is still quite tricky.

Ultimately, it doesn't really matter how hard you try to get a good estimation. You will always be underestimating by a factor of two to four. You could also refer to it as guessing rather than estimating. The only conclusion is that you should avoid guessing and simply try to implement the task as well as possible.

Precision and Accuracy

"Saying that pi = 17.630231 is more precise but less accurate than saying that pi = 3." - David Farley

There are many companies prioritizing precise answers over accurate ones. This is nonsense. The precision should always be as good as the accuracy. If someone asks you how long it will take, you should provide an estimate in terms of an order of magnitude: hours, days, weeks, or months. Only if you have several different options can you determine which one will take the least time.

Single line complexity

A common subject of discussion is the level of logic present in a single line of code. There are very different opinions. On one side, we have Linus Thorwalds. In the Linux kernel, the maximum line length used to be 80 characters when using the C programming language, and the length of indentations is 8 spaces. It is absolutely impossible to write more than one or maybe two operations on a single line of code. Try it yourself. It is really worth writing such code once in a while. You will learn quite a bit about what code can look like.

On the other end of the spectrum are some Python programmers. It seems like turning the addition of as much logic as possible into a single line into a sport. Honestly, I believe this is a rather unhealthy habit. You don't gain anything by saving lines of code. At the same time, every single line becomes increasingly convoluted. You won't understand it anymore. For this reason, the maximum line length set by the Google Style Guide is 80 characters. For both Python and C++. [](https://google.github.io/styleguide/pyguide.html Section 3.2), Additionally there are restrictions on list initialization. For example, it may not loop over two different variables, as shown in the following example.

```
[[[0] * (i + j) for i in range(2)] for j in range(3)]
```

One alternative is to refactor out one of the loops:

```
l = []
for j in range(3):
    l.append([[0] * (i+j) for i in range(2)])
```

Or we can do it the old way:

```python
def create_matrix():
    matrix = []
    for j in range(3):
        row = []
        for i in range(2):
            row.append([0] * (i + j))
        matrix.append(row)
    return matrix
```

When in doubt, resist the temptation to split up the code and avoid using single-line initialization. Here I would prefer the second solution using a list initialization using one variable.

Black magic code

Your code will contain some complexity. There's no doubt about it. The only question is how you deal with it. One point is that you have to be honest. Some programmers try to hide complex code using all kinds of black magic. This approach may work occasionally, but the code will be cursed. You can keep working on the code, but occasionally you see this black magic and you'll become petrified. Your only thought will be: "I hope I'll never have to touch this."

It is much better to be honest. The problem has a certain complexity, and we break it down into smaller pieces that we can solve. Do not hide the complexity; make it apparent.

28. Dependencies

"If you automate a mess, you get an automated mess." — Rod Michael

In this chapter, we are discussing files that depend on each other. An inevitable evil.

The Early Days

In the early days, people wrote code in a single file. This has several drawbacks. It's very easy to lose track of the code, and it can be challenging when you need to replace a part of it. For example, if you found a faster library. Even worse, the library is only available as a binary. Then you cannot use it at all.

These are some of the considerations that led programmers to split their code into multiple files. How do you instruct the computer to compile the complete code from these files? Apparently, there are some solutions, but this is an ongoing discussion.

In C++, the problem becomes even more challenging because the compiler requires the header files. It is possible to compile a C++ program using the command line for a single file, but it becomes impractical for larger projects. If you use C++, it is inevitable that you learn a build tool, such as CMake or Meson.

All programming languages have import or include statements at the beginning of the files. Even with the build tools of C++, you still need them. They might bother you, but at times they are quite handy. They are an indicator of some very bad patterns in your code.

The dependency graph

If you draw a plot with all the files represented as circles and their interconnections as arrows, you should obtain a directed acyclic graph. The trunk of this graph is the file containing the main function, representing the highest level of abstraction. As you move up on the graph, the level of abstraction decreases.

// create a figure of this graph

Now, the first thing to look out for in this abstraction graph is two arrows pointing in opposite directions. This means that two files are importing each other. Depending on the language, this may result in anything from normal behavior to undefined behavior or errors. But even if it works, it is a very poor design. If you have mutual dependencies, there is no clear distinction between the levels of abstraction. It's just a mess.

The simplest solution is to merge these files. However, this is only a superficial fix. You really have to determine the relationships between the functions and classes in the files. Maybe you need to reorder them, or perhaps you have to rewrite the corresponding code from scratch.

Breaking up Dependencies

Circular imports are not a common problem because they are easy to spot, and experienced programmers typically do not encounter such issues. With good coding habits, circular dependencies won't occur. The much more common problem is having too many dependencies. This makes the code very sticky. It is challenging to provide specific numbers to quantify the problem because it depends on numerous factors. Breaking up a file is usually beneficial, but it also leads to an increase in the number of dependencies. This is inevitable. As a rule of thumb, breaking a file into two is a good thing if the number of dependencies increases only a little, and it should be reconsidered if the number of dependencies almost doubles. The latter means that most code requires all the code from this file, so it makes sense to have it all bundled together.

How you break up a file is an even harder question. Sometimes you can easily group the code into clusters, while other times it is challenging to discern what belongs together. If you have divided a class into two classes, you can definitely separate some of the code into a new file.

The most important step toward reducing dependencies is to focus on your code. Ensure that similar code fragments are placed in the same location. Having database access spread throughout the code is typically a significant red flag. The logic of your code should be concentrated in a few key areas. Make all the database requests at once, whenever possible, and store the results in an instance of a data class. Afterward, you can pass around this class instance, and there is no need to think about the database anymore. And just like that, you have eliminated many dependencies while simultaneously enhancing your code.

The most challenging aspect is reducing dependencies by enhancing the overall structure of the code. Good code has simple logic, which in turn has few dependencies. This, however, is quite tricky to achieve, and even if I could, explaining it here would be barely possible.

Circular Dependencies

Circular dependencies typically occur when two classes exchange data between each other. Class A requires data from class B, which in turn requires some data from class A. You should be able to determine whether class A or class B corresponds to the higher level of abstraction. Let's assume that class A is the higher-level class calling class B at some point. Now, Class B should never have to call class A. B is at a lower level and knows nothing about the high-level class A. This leads to only one solution: class A has to call class B exactly once and hand over all the data that class B needs to return the final result.

Long story short: The high-level object calls the low-level object and passes all the necessary data at once. The low-level object returns the final result at the end of the calculation. This resolves the problem of circular dependencies and organizes the levels of abstraction. The only tricky part is that the high-level object does not know exactly what the low-level object needs.

Example

This example is intentionally simple. I hope no one would write code like this. It's just to make a point. Here we have a circular dependency between the functions `a` and `b`. Apparently, this makes the code much more convoluted and harder to understand than necessary.

```python
def a(counter):
    if counter > 0:
        b(counter -1)

def b(counter):
    print(counter)
    a(counter)

a(5)
```

Now, the first thing to note is that there is no clear level of abstraction. `a` calls `b` and `b` calls `a`. They are somehow both on the same level of abstraction. This is bad. We could simplify it by inserting the definition of `b` into the function call inside `a`.

```python
def a(counter):
    if counter > 0:
        print(counter-1)
        a(counter-1)

a(5)
```

This is already much simpler. Of course it could be simplified even further by removing the recursion all together. But this is only a side remark.

```python
def a(counter):
    for i in range(counter-1, 0):
        print(i)
```

As a summary, one can say that circular dependencies should be avoided altogether. This task is usually not too difficult if you use proper levels of abstraction, and it significantly enhances the readability of the code. Even a single recursive call can often be refactored to improve code readability.

29. Decoupling

"Before software should be reusable, it should be usable." — Ralph Johnson

[](Refactoring, Martin Fowler), [](The Pragmatic Programmer)

Coupling is a crucial aspect of software engineering. Without coupling, it would not be possible to write code. Coupling is the glue that holds everything together. But too much glue is bad because everything becomes sticky. In bad code, everything depends on each other. Every module or file imports dozens of other files. This is a significant issue because if you want to change one file, you might have to change a dozen others. Ensure that the coupling is minimized. This keeps the code soft and flexible. It is the ultimate goal to have completely decoupled code. This makes it easy to work with. It makes it reusable.

This is one of the reasons why global variables and inheritance are not recommended. Global variables are detrimental as they tightly couple the entire codebase. It's worse than importing something everywhere. All your code dependencies start intertwining with each other. This is absolutely deadly. Never use global variables.

Inheritance is not quite as bad, but almost. Everything that depends on a derived class also automatically depends on its base class. You are not only coupling the derived class to the base class but also vice versa. You can barely change one without changing the other. This is not how flexible code is supposed to be. Avoid using inheritance.

Microservices, on the other hand, are highly decoupled. They are chunks of code that can be called and executed independently. Microservices are somewhat similar to functional programming, where you have independent functions that run autonomously. Microservices and functional programming both involve calling a function or a piece of code that returns a value.

Only ask for things you directly need. This is another advantage of functional programming or microservices. If you need to validate an email, you can utilize the email validator, which can be either a microservice or a pure function. The email validator returns a result and resets. You only received what you asked for, nothing more. There are no semi-useful objects wobbling around that you don't know how to deal with. You need exactly what is around. This is the strength of functional programming.

-# Part 6: High level design

30. Software Architecture

"Architecture: The decisions you wish you could get right early." - Ralph Johnson

In this chapter, I am only providing a high-level introduction to what software architecture is. I'm not an expert on software architecture, and I'll leave the details to more specialized books. [](Clean Architecture), [](Fundamentals of Software Architecture)

There are many people who misunderstand the work of "real" architects. Architects do not simply create a plan and pass it on to the construction company. Instead, they closely monitor the construction because there are always questions and problems arising that need to be addressed.

In software engineering, it is similar. As far as I understand the expression "software architecture", architects are not only responsible for designing the cornerstones of the software. They also have to monitor the entire process of software development as there will always be fundamental questions along the way.

Here is an example of how I understand the term "software architecture" based on my work experience: "I worked on a quantum compiler. We used an Abstract Syntax Tree (AST) to represent the gate operations. The gates were then translated into electrical pulses that were played by our devices. The compiler was comprised of numerous visitors who traversed the AST and sequentially conducted all calculations and optimizations." Anyone familiar with an AST and the visitor design pattern will have a good understanding of the code I was describing. In four sentences, I described the basic data structure (the AST) as well as the fundamental algorithm (the visitor pattern) used in the code.

The end of Architecture

One question is: where does architecture end? How detailed does it get? And, in my opinion, the answer lies in how far the architect plans it. There is no fixed boundary. What is clear, however, is that the architect cannot work out all the details by himself. If he did, there would no longer be a need for software engineers. So, unless a project is very small, he only has time to focus on the high-level design. All the technical details have to be worked out by the engineers. The architecture is never perfect, and there will always be implementation questions from the software engineers. One way architects can stay connected with the development team is by writing code themselves. They can write code examples or tests that use the code and do something interesting with it. [DDD p. 61 ?]

Designing Interfaces

One of the primary responsibilities of a software architect is to define the building blocks (libraries) and interfaces of the entire software system. Some of the interfaces may be only "partial interfaces", indicating that they are interfaces within a library. It fulfills all the requirements of a real interface, and the library could easily be divided into two parts at this point. It is an internal interface that is not exposed to the outside.

A partial interface has the advantage of requiring only a limited amount of maintenance for versioning, etc. However, there is a risk that the interface may become obsolete over time as programmers begin to work around it.

It is the architect's job to determine at the outset where and what kind of interfaces are required. He has to foresee the future. The YAGNI principle does not always apply to an architect. What if it turns out that we really needed that interface after all? Implementing an interface in existing code will be very costly.

Separate Libraries

Increase cohesion within a library and reduce coupling between them. It's exactly the same principle as for classes, just on a higher level.

In every large codebase, you will have to work with multiple libraries. Some of the software components are developed internally, while others are third-party libraries. There are many factors to consider when making such decisions. The very first question is: Do you need another library? Can you implement the required functionality within an existing library? There are mechanisms that favor either smaller or larger libraries.

These advantages for either side lead to trade-offs in library sizes. Generally, it is advantageous to establish a dedicated library if there is a suitable opportunity.

Coupling

Interestingly, all the explanations about coupling and cohesion made for classes are also applicable to libraries. It is important to note that libraries should not become too large and rigid. You don't win a prize for writing the largest library in the company. One library that covers every object that exists. It just won't work!

An apple can have a color, a flavor, and a price. There can be three different areas: graphical rendering, food, and shopping. Each one uses exactly one property, and it makes no sense to mix them up. Keep the libraries separate and write glue code between them if needed. That's the only way to go. Just trust me. Don't write monolithic software that tries to replicate the entire world. It won't work.

In every large codebase, you will have to work with multiple libraries. Some of the software components are developed internally, while others are third-party libraries. There are many factors to consider when making such decisions. The very first question is: Do you need another library? Can you implement the required functionality within an existing library? There are mechanisms that favor either smaller or larger libraries.

These advantages for either side lead to trade-offs in library sizes. Generally, it is advantageous to establish a dedicated library if there is a suitable opportunity.

Coupling

Interestingly, all the explanations about coupling and cohesion made for classes are also applicable to libraries. It is important to note that libraries should not become too large and rigid. You don't win a prize for writing the largest library in the company. One library that covers every object that exists. It just won't work!

An apple can have a color, a flavor, and a price. There can be three different areas: graphical rendering, food, and shopping. Each one uses exactly one property, and it makes no sense to mix them up. Keep the libraries separate and write glue code between them if needed. That's the only way to go. Just trust me. Don't write monolithic software that tries to replicate the entire world. It won't work.

31. Design Patterns

Design patterns [](Design Patterns, Elements of Reusable Object-Oriented Software) refer to a specific arrangement of classes, methods, and inheritance that give rise to unique properties in the resulting object. There are about two dozen commonly recognized design patterns and numerous books explaining them. I have neither the space nor the knowledge to write about all of them. I just added this chapter because I think it's important that you learn about design patterns.

Factory

I'll show the so-called factory pattern as an example to give you an idea of what design patterns are all about. It's a very simple pattern. Chances are that you have already implemented a factory before, even if you were not aware of this pattern.

```python
class Car:
    def move(self, speed):
        if speed > 200:
            raise Exception("Cannot move that fast")
        print(f"Car is moving at {speed} km/h")
```

```python
class SpaceShip:
    def move(self, speed):
        print(f"SpaceShip is moving at {speed} km/h")

def factory(type):
    if type == "car":
        return Car()
    if type == "spaceship":
        return SpaceShip()

vehicles = []
vehicles.append(factory("car"))
vehicles.append(factory("spaceship"))

for vehicle in vehicles:
    vehicle.move(100)
```

When looking at this code, you might ask yourself: "what is the point"? This is because the factory in python doesn't seem to be anything special. In Python, implementing a factory is particularly easy due to duck typing. In strongly typed languages like C++, you would have to use a base class and pointers to implement the `vehicles`.

The crucial point of the factory is that you can create objects of different types depending on a string or whatever else you provide. Factories are generally very useful for the creation of objects. They take some arguments and return an object. In some cases, factories are classes. But there is nothing wrong with defining them as functions.

One last remark: Instead of using the if statements in the factory, you could also use a dictionary. In my opinion, this makes the code better and shorter.

```python
factory = {
    "car": Car(),
    "spaceship": SpaceShip()
}

vehicle = factory["car"]
```

32. Domain Driven Design

"The complexity of your code should be at most as complex as the problem space it inhabits and no greater." - David Whitney

This chapter is highly influenced by Eric Evans' book [Domain Driven Design](Domain Driven Design, Eric Evans). The book covers mostly conceptual topics such as the domain model and bounded context. This, along with the concept of "Ubiquitous language" (Evans), forms the heart of the book and will be explained in this chapter. Though there are some more books on this topic by now. [For example](Learning Domain-Driven-Design, Vlad Khononov).

Ubiquitous Language

In software engineering, there are very few topics that are described purely mathematically. Most notably finance, physics, and engineering. Most other topics are described using natural language. This is a significant challenge because it is difficult to incorporate such a topic into code. How do you implement an apple? The answer is: it depends on who you are talking to.

It takes a lot of effort to understand a topic well enough to be able to implement it. Engaging in extensive discussions with domain experts about the topic is essential. Only through these discussions can you learn how their domain model is built up and what the underlying mechanisms are. This common language between developers and domain experts was named "Ubiquitous Language" by Eric Evans. It is of utmost importance that the development teams learn this language used by the domain experts. The development team should communicate effectively using this language, and implement it into the code. A domain expert must be able to understand the general discussions among developers. He has to be able to tell when something is off because there is something that doesn't make sense to him. For instance, if the developers mix up the usage of atoms and molecules in a chemistry simulation. Usually, domain experts can detect issues much earlier than developers. If there are expressions used in the code that do not exist in the domain, it is most likely incorrect.

Developing this Ubiquitous language is of utmost importance for the whole project. Only a well-developed shared language between developers and domain experts enables high-level discussions about the domain required for the success of the entire project. Developing such a language requires a significant amount of effort. Developers and domain experts need to stay in constant communication and continuously refine their language usage to enhance the model built upon this language. Play around with this language. Attempt to alter the vocabulary. Try to create new phrases. This is an important aspect of the Ubiquitous Language. You have to develop this language like children learning to speak a natural language. Find easier and more effective ways to express your thoughts, regardless of how silly they may seem initially. Utilize the insight gained in this way to enhance the domain model. Ensure that the business experts understand what you are discussing.

Thinking about the code in English language also helps, even if you don't do much DDD. The following explanation from the book [The Art of Readable Code](The Art of Readable Code, Boswell and Foucher) can help you improve your coding skills significantly:

1. Describe what code needs to do, in plain English, as you would to a colleague.
2. Pay attention to the keywords and phrases used in this description.
3. Name your variables and functions to match this description.

Especially if you're struggling to translate your thoughts into code, these steps may help you organize your ideas, making it much easier to write the code. If you cannot articulate the problem or your design in words, there is likely something flawed. This technique also allows you to avoid comments as you name your variables the way you would describe them in the English language.

The Domain Model

A model is a simplification of something real. A computer game, for instance, is always a model of some kind of reality. Interestingly, a computer game does not necessarily become better if the model or the graphics are more realistic. But rather if the model is more focused on making a point. If it emphasizes the core domain of what the game is all about while leaving out unnecessary details.

When writing code, we implement a model of reality. A model that closely resembles the problem we are trying to solve. Not one that is closest to reality. The model must cover the domain of interest. The field in which you are working. The model needs to simplify the domain you are working on to the bare minimum required to fulfill your programming task.

The domain model is a high-level concept that needs to be described. This can be done in several different ways. The most common descriptions are UML diagrams. These are commonly used to illustrate the relationship between different classes. However, UML diagrams are not always the ideal choice for describing code. UML has several deficiencies (and I don't like them too much, that's why you don't see any of them in this book).

Documentation and Planning

First, UML diagrams support only a somewhat limited amount of interactions between classes or class instances. There are often more effective ways to describe code than using a class diagram. Maybe a piece of text or a diagram illustrating the temporal dependency of a process would be suitable. It does not really matter how you represent the domain model, as long as you understand it.

Secondly, one always has to consider that UML diagrams should remain small. Some development teams have printed out their entire code base as a UML diagram. But this practice is largely ineffective. There are too many objects and interactions between each other in such a graph, as if it could be useful. It's just like a map with too many details. Buried in all the information you won't find what you are looking for. A map should be simple and easy to understand. It should only show what you are interessted in. And the same applies to UML diagrams. Keep them small and concise. Focus on what you are interessted in.

There have been attempts to create a programming language similar to UML, but they have all failed for various reasons. Graphical programming simply isn't any better than textual programming. On the contrary. Graphical programming generally lacks important tooling like version control or testing frameworks. Furthermore, a significant amount of information may be lost during the creation of the diagram. UML is not a complete programming language and it will never be. UML diagrams cannot explain the entire functionality of your code. Thus, keep UML diagrams small.

Instead, you can use any type of document you prefer. At times, it is better to create a temporal order of a process than a class diagram. Or you can create a diagram with class instances instead of class definitions. After all, it's called Object-Oriented Programming, not Class-Oriented Programming.

As with all documents, the documentation of the domain core should be kept up to date or archived. There is a risk that the documentation and the code may diverge over time. Documentation has similar drawbacks as those described in the chapter on comments. It takes a lot of effort to keep documentation up to date.

Though documentation has its merits. Code is often too detailed to effectively explain its functionality. And there are plenty of things that code alone cannot explain. It has to be complemented possibly by comments and some additional documentation.

Ensure that design documents extensively utilize the Ubiquitous language. If the documentation does not use the same terms as defined in the Ubiquitous language, it is not useful. It doesn't help to explain what you are trying to implement. It only creates confusion.

Implementing a Model

There are cases where you cannot implement a model you have developed. It would be simply too complex. It just doesn't work as planned. This is a clear sign that your model is not optimal. A domain expert can explain it, so you should be able to implement it. In theory, the complexity of the domain model should not exceed the complexity of the problem it tries to implement. This is the optimal scenario where a developer can explain the code to the domain expert, and the domain expert can understand it. They would simply talk about the same thing, the same logic. In this case, the development of the code would feel very easy as everything just falls into place.

In reality, finding the optimal model is a challenging process. Most likely, you'll end up in an iterative loop switching between coding, modeling, and refactoring until you have a breakthrough when you suddenly realize what the optimal model should look like.

// graph of the breakthrough?

Decouple the domain-model code from your other code, as explained in the section on The Abstraction Layers. This is important for maintaining a clean and concise domain code. Violating this rule would also violate the SRP as the domain model is located on a different abstraction level than for example the database code. The domain model contains the actual conceptual complexity of the final software. Therefore, it should not be cluttered with non-model related things such as infrastructure or GUI code. Keep the domain model slim.

Domain Levels

Not every part of the software can be treated with equal priority. You'll have to prioritize what is important. There will be various domains in your project. For example, the core domain. The core domain is the most crucial domain of your project and must be treated as such. The core domain is what your company makes money with; it is the unique aspect that sets your company apart from other companies. Try to keep core domain concise; only the most essential elements should be included in it. Your most experienced developers should be working on this topic.

Around the core domain, you will have several other domains. Each domain typically implements one class of features to support the core domain. For example, an infrastructure domain may involve managing the database or the math library. Keeping the domains separate is important as it prevents you from creating a Big Ball of Mud.

Each domain corresponds to a piece of code, such as a library. The different domains are fairly independent of each other. They are only linked through their interfaces. Otherwise, there doesn't have to be much resemblance between the different domains. For example, the ubiquitous language does not have to be the same across different models. On the contrary. The ubiquitous language is expected to vary among different models, and at the interface, there is an adapter that functions as a translator between the different languages.

As one example, a flight may refer to the time between takeoff and landing. But there may also be direct flights or flights with stopovers. This is an example where one expression may have different meanings, depending on the type of model you are working with. Therefore, it is always important to keep in mind the type of domain model you are currently working in and the specific type of flight you are discussing.

Domain Specific Language

A Domain Specific Language (DSL) is a language is a language specifically tailored to the needs of a certain problem. This DSL has to resemble very closely the ubiquitous language as it should be the code version of how marketing people would talk about the problem. However, it is not exactly defined how this language should be implemented. Obviously it has to be a programming language of some sort. It can be one relying on heavy usage of preprocessor macros, or a language like Gherkin that resembles spoken language. Yet it can also be a normal API using function calls.

Honestly, I don't like the preprocessor or Gherkin approach. If that was the best way to program, our programming languages would all work this way. Which is not the case, probably for a reason. I prefer to design a dedicated API to every problem that a marketing person can still understand with a little bit of programming knowledge. That's also way less work than developing a preprocessor or Gherkin language.

Domain Boundaries

As your code base grows, it becomes more and more difficult to keep working with a single domain model. There are processes that tend to tear the domain model apart. An object may have very different properties, depending on what part of the code you are working on. For example a `user` has different properties in the payment domain than in the GUI domain. Of course it would be preferable to have a single domain model for the whole code base, but this is not a requirement for good code. You may have several different models, depending on which part of the code you are working on. There will be interfaces between the different models. And the big question is: how do you deal with the different models?

Bounded Context

A bounded context is everything within a boundary. Typically, a domain model corresponds to a bounded context, where the boundary represents its interface. The interface regulates what goes in and out of the bounded context. Bounded contexts are important as they separate specific problems from the enterprise-wide codebase. One example of a bounded context is the math library. The names used in this library may also be applied in other contexts, but `sin`, `cos`, etc. have a very specific and well-defined meaning within this bounded context. These expressions should not be reused within the math library. On the other hand, the terms `sin` and `cos` may be utilized in other bounded contexts and carry a completely different significance.

Typically, domain models consist of one bounded context each. All the problems mentioned so far for the domain models are true for the bounded contexts as well. Mostly that if they get too large, they tend to be torn appart and that objects carry too much information.

Unified Model

The attempt to keep the model unified is the most obvious one. Though it is challenging to sustain the necessary level of communication to uphold this condition. A good way to enforce this communication is through Continuous Integration (CI). CI compells the team to merge frequently and at an early stage, making any disparities between the model and the actual code evident at an early stage. The automated tests enforce the behavior of the model and warn the developers if they are inadvertently changing it.

On the other hand, working on a unified model is not always possible because for larger projects, the forces that tear the single model apart become too large. In an enterprise-scale software, it is simply not possible to work in a single model. The various requirements for the code may become too extensive. In a single model, the `user` object for instance, becomes too complex as it continues to grow over time. At some point, it is easier to work with several different `user` objects, each of which is smaller and with different attributes, making them easier to manage.

Context Map

A context map is important when a model is divided into two or more parts. All parts are now separate bounded contexts. They are individual domain models with clearly defined boundaries. You'll need a translation map to convert one model to the other. The translation map is similar to an adapter pattern. It converts one interface to the other one.

Shared Kernel

Two bounded contexts may share a common sub-context. This is typically the core domain utilized by various domain models. Having a shared domain core means that all involved models must ensure that the domain core is always in sync. This can be done by the CI. Additionally, it takes considerable communication between teams. Otherwise, the core domain may become fragmented.

Anticorruption layer

The anticurrupiton layer is similar to the adapter design pattern. You add a small layer around your context. If the other code changes, you only have to adapt your anticorruption layer and not the entire code base. This can save you a lot of work.

The anticurruption layer can be located on the outgoing part of an interface, though typically the users of an interface write it themselves. Just to make sure that their code won't break if the developers change something.

Separate ways

Sometimes, the burden of maintaining models collectively becomes too big and it becomes apparent that collaborating further is no longer worth the effort. There is very little overlap between the two models, so cutting them apart is not a significant issue. If you need a feature from another model, simply reimplement it. Having a little bit of redundancy between two models may be preferable to coupling them together. If the redundancy become too significant, it may be worth implementing a shared kernel.

Developer-Client Relationship

The model is split into two parts, and one development team is relying on the other team's model. If the upstream team (the developers) is willing to cooperate (for financial or political reasons) with the downstream team (the client), the two teams can establish a developer-client relationship where the downstream team can request features for implementation by the upstream team. The success of this relationship hinges on politics and the cooperation of the upstream team within the company.

Conformist

The conformist is a model where the downstream team just follows the upstream team. The downstream team doesn't have anything to say. Needless to say that this is not a very preferable solution for the downstream team, though it is sometimes inevitable.

Building Blocks of DDD

//[https://stackoverflow.com/questions/77425208/when-do-you-use-entities-value-objects-and-aggregates-ddd]

In [Domain-Driven Design](Domain Driven Design, Eric Evans), Eric Evans introduced, among others, the terms entities, services, value objects, and aggregates. These are various models used to differentiate between objects with diverse properties. Generally, the building blocks of domain-driven design are implemented in OO design. In most cases, this is the easiest choice to model the functionality of the building blocks. However, other programming paradigms may also be chosen.

I'd like to point out that you don't have to implement everything using entities, value objects, etc., as explained here. It should be regarded as just a different way to think about how to structure your code.

Entities

[https://youtu.be/4rhzdZIDX_k]

Entities are unique objects. Their lifetime typically spans most of the code's lifespan, and they possess unique properties such as an ID. Humans are a very simple example. Every human is unique, and there are efforts to assign some form of identification to each individual. Though this is harder than it sounds. Obviously, names are not suitable as a unique identifier. The social security number is used in some places, but not everyone has one, and there is nothing comparable in many countries outside the US. For many websites, the email address is used, and sometimes the phone number is also required.

Another example of entities is seats in a stadium. Each customer buys a ticket for a specific seat. Thus, the seats and the customers are both entities. They are both unique objects. For each customer, exactly one seat is reserved. Every seat has a unique ID. Two seats are only considered equal if their IDs are the same. Even if all other properties are the same, if the IDs are not identical, the seats are not considered equal.

Now, it is different if the tickets are not assigned to a specific seat (general admission). If the customers can sit in any available seat. Then the seats and customers are no longer considered entities. They are just one object among many. They become exchangeable. They become value objects.

Opposite to value objects, entities are not constant. They may have internal state which changes over time. Only their ID has to remain constant. Whether this ID is really required depends on the situation. Gerneally it is sufficient that you have access to a variable. The name of this variable already serves as an ID inside your code. However, if you want to store this object in a database, you certainly need some ID to retrieve this object later on.

Value Object

Value objects are essentially the opposite of entities. Value objects are defined solely by their properties. They do not have a unique ID. One example is apples in the supermarket. We can regard them as indistinguishable. The only interesting aspects of an apple are its flavor and price. Other than that, it can be replaced at any time. Value objects are immutable. You can only set the properties of a value object during its creation. Thus, if you don't like your apple, replacing it with another one is the only option you have. It's not possible to change its properties.

Having value objects is extremely useful, even if you don't care much about DDD. Value objects are generally small custom types, such as a price. The price is set in its constructor and cannot be changed thereafter. Furthermore, the constructor can ensure that the price is valid; for example, it cannot be negative. Therefore, the constructor takes care of the checks and you won't have to bother with it any longer. Another example is the usage of an email address object. This object can ensure that the email address is valid. Therefore, using value objects is much better than using a string for the email address.

Value objects also help against the primitive obsession. Here is a small example of an email address as a value object:

```
def _check_email_address(address):
    assert address.count("@") == 1
    assert address.count(".") >= 1

class EmailAddress:
    def __init__(self, address):
        _check_email_address(address)
        self.ADDRESS = address
```

It fulfills the requirement of a value object. You can only set the value once in the constructor and the validity of the adress is checked there as well. It is not possible to change the adress afterwards. Class instances of `EmailAdress` are immutable. Value objects are therefore perfectly suited for use in functional programming.

As you may have noticed, value objects don't fit into any of the class types we discussed so far. A value object is a data class with additional checks in the constructor.

Now, the question remains: when should an object be considered an entity or a value object? As I mentioned before, value objects are immutable. So, if you have an object, like the apple mentioned above, that will never change its properties, it is likely to be a value object. On the other hand, if something is important enough to change its properties, it should be considered an entity. In general, you should have much more value objects than entities in your code.

Services

Services are used for operations on value objects or entities. A good service has 3 properties [](Domain Driven Design, Eric Evans, p. 105):

- The operation does not naturally align with an entity or a value object.
- The interface of the service is defined in terms of the domain model.
- The service does not have any internal state that can change over time.

A service is an operation on the domain model. Its name is part of the Ubiquitous language. Services are typically represented by functions.

While entities and value objects are generally too fine-grained to be reused, services are of medium granularity and thus appropriate for reuse.

Aggregates

Aggregates are a special type of entity. They typically consist of other entities and value objects. The goal of an aggregate is to form an object whose operations never violate the systems invariants. A very common example of an aggregate are smart pointers or the vector class in C++. They are both implemented following the "Resource Aquisition Is Initialization" (RAII) pattern [](Effective modern C++). Both store some data and encapsulate the memory management at the same time. When using the interface of a vector, the mechanisms underneath it always make sure there is enough memory allocated and that it will be deleted once the vector goes out of scope. And as a user, you don't even have to think about these invariants. They are hidden within the implementation.

Another somewhat more general example of an aggregate is a car. The car has a global ID. It consists of an engine, a chassis, and four tires. Let's say that the tires wear out, and occasionally, you have to replace them. This makes them an entity. Meanwhile, the engine and the chassis never change their state. These are value objects. The entire car can only be accessed from the outside. The engine, chassis, and tires can only be accessed from within the car object.

Once the tires are worn out, they are disposed of at the recycling plant. The recycling plant is likely represented by a different domain model compared to the car. At the recycling plant, no one cares anymore about how worn out a single tire is. The recycling plant simply consists of one huge pile of old tires. The tires become valuable object. However, we are not going to model the recycling plant in our code here.

Here is an example of how the car entity could be modeled in code. `Engine`, and `Chassis` are value objects, `tires` are entities.

```
MAX_DISTANCE = 1000

class Tire():
    def __init__(self):
        # invariant: self._distance_remaining >= 0
        self._distance_remaining = MAX_DISTANCE

    def drive(self, distance):
        self._distance_remaining -= distance
        assert self._distance_remaining >= 0

    def get_distance_remaining(self):
        return self._distance_remaining

def replace_tire_if_needed(tire, distance):
    if tire.get_distance_remaining() < distance:
        tire = Tire()

def drive(tires, distance)
    assert distance <= MAX_DISTANCE
    for tire in tires:
        replace_tire_if_needed(tire, distance)
        tire.drive(distance)

class Engine():
    # ...
    pass

class Chassis():
    # ...
    pass
```

```
class Car:
    def __init__(self):
        self._tires = [Tire() for _ in range(4)]
        self._engine = Engine()
        self._chassis = Chassis()

    def drive(self, distance):
        drive(self._tires, distance)
```

You might have realized that `Car` is a delegating class. Delegating classes typically meet the requirements of an aggregate. A delegating class hides all the functionality within the class so that it is only accessible through the class instance itself. The `car` instance is accessible as it is a variable. A unique ID would only be needed for saving the `car` object in a database and retrieving it. If you want to save the car information to a database, you would need an identifier, such as the license number. The entire car would be saved intact under this ID.

It is not possible to modify the instance of the `Car` and its internals in any other way. It is not possible to violate the car's invariants. All the variables are encapsulated within the `Car`, and the methods are guaranteed to maintain the invariants.

The class `Car` may have the following invariants. It needs:

- a unique identifier (as it's an aggregate)
- 4 tires (value objects) with `_distance_remaining >= 0`
- 1 engine (entity) with an ID
- 1 chassis (value object) Upon construction, these invariants are guaranteed by the constructor of the class, assuming that the `ID` is indeed unique. All the other functions (services) that act on the car must ensure that these invariants are not violated. The method `car.drive()` calls the function to replace tires if they are worn down, but there are still four tires with a positive `_distance_remaining` after the replacement and driving some distance.

Aggregates should always be treated as whole objects, for example, saved to or loaded from a database. Aggregates are always entirely within the domain level of your code. It can't be any other way because all parts of an aggregate are on the same level of abstraction. Having one element of the infrastructure level inside an aggregate would violate the SRP.

As the root entity is the only thing accessible from the outside, it is comparatively simple to enforce the invariants of the aggregate. For example, every car has four wheels that are not yet worn down. All accessor functions must pass through the root entity. Thus, this is the place where you can enforce the invariants. There, you can define functions as `drive` that take care of ensuring the wheels are in good condition and replace them if necessary.

Aggregate instances are often created by a factory or another creational design pattern. These patterns allow us to outsource the creation of a fairly complex object. This is in accordance with the SRP. If the instantiation of an object is fairly complex, it is a noteworthy task and should be handled in a dedicated object. Furthermore, the factory can also take care of the invariants of the class instance at its creation.

Just to clarify the concept of an aggregate, let's consider an example where it is violated.

```python
class Engine:
    _is_started = False

    def start(self):
        self._is_started = True

    def is_started(self):
        return self._is_started

class Car:
    def __init__(self, engine):
        self._engine = engine

    def start(self):
        self._engine.start()

    def get_engine_status(self):
        return self._engine.is_started()

engine = Engine()
car = Car(engine)
car.start()
print(engine.is_started()) # True
```

Now that the engine is started, we can use it. This, however, is bad code. The engine is now part of the `car` aggregate and should only be accessed through the `car` interface. Here it is accessed directly. You can also think of a real car where you install the engine, start the car, and then remove the engine for inspection. This feels wrong. And as it's wrong in the real world, it is most likely wrong in the code domain as well.

The solution is the function `get_engine_status()` that I defined in the `Car` class as well. Now, the aggregate is properly encapsulated, and you can access the relevant properties of the engine through the car's interface. Note that this way, we don't need the intermediate `engine` object. Instead, we can directly instantiate it inside the `Car` constructor.

```python
car = Car(Engine())
```

Of course, it is sometimes more convenient to have an `engine` as a temporary object. For example, if the readability is affected because the engine is created by a function with a very long name. To achieve this, you should create the mentioned object along with the `car` instance within a specialized factory to encapsulate this temporary object. There, you construct the `engine` and then pass it on to the `car` constructor. The engine should never be visible from the outside. It should be accessible only through the `car` interface.

33. 3rd party software

"Prefer visa over power shell" – some YouTube video

There are thousands of companies selling software parts. For many problems, there are open-source solutions available. This is great, but as always, there is a price to pay.

No airplane engineer would start developing his own jet engine, and no programmer would write their own database software. Even if they don't like the products available, they can still purchase something from the market. Everything else is simply crazy; it's too expensive. Other companies are developing databases, and you are not going to compete with them. You want to do other things instead. You have found your niche elsewhere and you plan to remain there unless there is a compelling reason to completely change your business. You outsource everything that you don't really have to do yourself.

In software engineering, there are not many products available to address all possible problems. But still, there are quite a few suppliers available who can assist you in solving some of your issues. Especially when it comes to infrastructure.

It is possible that you have a bad feeling about this approach. You want to do everything by yourself. You don't want to pay other companies for their libraries. I can assure you that your feelings are natural. But you have to get over it. It's just not worth doing everything by yourself. You haven't developed your own operating system, the database you're using nor the cloud service. Instead, you earn a good amount of money every year by working on your core domain. You do what you can do best. And if you can save time by outsourcing other parts of the code, that is great. This also allows you to save on maintenance costs, which are typically even more expensive than the actual development of the software.

Using third-party libraries or software is great. Most of the time. But sometimes it also has its issues. Some companies did not adhere to coding standards, and now they are in trouble. And there are many more sources of trouble. Most famously, all the customers using Oracle databases who didn't decouple the database from the rest of the code. The code extensively utilizes Oracle database queries, making it challenging to switch to a different database vendor. These companies now pay substantial licensing fees and cannot evade them.

Another problem is libraries with comparatively few contributors. At some point, there might be no one left to maintain the code. In most areas, you can still use such a library for a while, but you should look out for a different solution. A lot of problems can arise when using software that is no longer supported. If you truly require this software, you may want to consider becoming a contributor yourself.

Everything explained here is also true for IT services and infrastructure (Amazon Web Services (AWS), github, google maps, ...). These services are great, but you should always have the ability to change your supplier. For example, there are medium-sized companies that require a significant amount of computing power. They started the company using AWS, just like everyone else does. It's just too convenient. As the company grew, the AWS bill reached millions, prompting the company to migrate to its own server infrastructure. [https://youtu.be/XAbX62m4fhI]

In short, third-party software and services are generally excellent. They may save you a lot of work and money. But you have to make sure you don't get stuck with it. You have to stay flexible. Decouple the third-party library from your code. Write a lightweight adapter between your code and the library. And if you don't, make sure that you really want to stick to this specific third-party code for a long time. As always, if you can write tests using dependency injection and similar techniques, you are probably fine. In order to mock the database, you need an interface that can be used to support other databases as well. So, you are flexible.

The very big question is always when you really need such an adapter. Most of the time, I am too lazy to write one. But you certainly need one when dealing with databases. Call all the database-specific queries only within this thin layer. The entire remaining code is a database-syntax-free zone. This makes it very simple to exchange the database. You only have to replace the wrapper. You might have to modify some of the implementation since the functionality varies slightly between databases. But this is a small price to pay compared to the millions you paid to Oracle so far.

You should reconsider using a third-party library if it has only a few developers. If there is a reasonable alternative, you might be better off avoiding it. On the other hand, if this code is crucial for your software, it would be beneficial to participate in the project and contribute as a developer. In fact, pretty much all major software companies support the software projects they rely on. Some projects received so much additional manpower that they ran out of work to do. And even the unthinkable happened: Microsoft became one of the biggest contributors to the Linux kernel!

-# Part 7: Existing Code

34. Refactoring Fundamentals

"If you wait until you can make a complete justification for a change, you've waited too long." – Eric Evans

There are books about refactoring techniques [Refactoring] which are highly recommendable. Still, the most important aspect of refactoring is that you know how good code looks like (and that you have plenty of tests to back you up). If you have a vision how the code should look, you are always going to find a way, how to make it better.

There will be change

If code exists for a long period (which is usually sooner rather than later), it will need to adapt to changes. When you add new features, the build system might change, the database could be altered, and you will need to adjust your code to fit the new environment. This is inevitable. Only if you write extremely low-level code with minimal dependencies, you might be safe. Or if you develop mobile apps that are expected to last only 1 or 2 years. In all other cases, you have no choice but to adapt to the changing environment. Your code has to remain flexible. You have to keep it in shape. Ensure that you can adapt to change.

Don't Let Your Code Rot

The most fundamental rule about refactoring is that you shouldn't let your code rot to begin with. Always make sure your code is well-tested and well-structured. This will save you a lot of pain in the future. Once you reach the point of having classes that are a thousand lines long, you will struggle to regain control of your code. By writing properly tested code from the beginning, you'll save a lot of time in the long run. Not only will it be easier to refactor your code, but your code quality will also improve, requiring less refactoring.

Even without external changes, it is important to refactor your code once in a while. We have to face the sad fact that our perfect code deteriorates over time. Every line of code you add is a potential source of deteriorating code quality. You may introduce duplication, enlarge the class size, or disrupt the logical order in your code. Over time, the code becomes messy and needs to be cleaned up. Sometimes it is also compared to entropy, the physical law of disorder [](The Pragmatic Programmer). Fighting entropy is hard. It takes a lot of effort, as explained in the section on entropy [](chapter Physical Laws of Code).

I assume that everyone reading this book is familiar with some of the reasons why code rots. The very first example is copy paste code. Copy-paste code should be banned altogether. Instead of rewriting a function to meet its new requirements, one simply duplicates it and modifies a line or two in its new position. Another issue arises when adding more and more features to an existing class. Also, the features that you added are in the wrong place in your code and need to be moved to the correct location. These are some of the reasons why code rots and requires regular refactoring.

Refactoring and Automated Tests

Refactoring means to change the code without altering its functionality. This is what people didn't do in very old code. They were afraid that they would break the existing functionality. That they would introduce bugs. It's as if they didn't clean up the kitchen because they were afraid they might break something. And they didn't see the reason why they should have cleaned up the kitchen. They only had a nagging doubt that something was wrong, but they couldn't pinpoint the exact issue. Long story short, the next person had to cook in a dirty kitchen. At some point, there were so many dirty dishes in the kitchen that they didn't even notice the bugs that could hide underneath each and every dirty plate. People using the kitchen were afraid of introducing bugs when refactoring, but in the end, they still ended up with bugs. They didn't clean up the kitchen or refactor the code. They started encountering numerous bugs further down the road because the whole code became a mess.

I really hope you understand that not refactoring is not an option. A cook has to clean up the kitchen continuously, just as you have to refactor your code. All the time. Refactoring is an integral part of your job, not just an optional feature. You are responsible for refactoring your code. Therefore, we have to help you overcome your fear of refactoring and your fear of introducing bugs. You need a safety net. Something that automatically notifies you when you introduce a bug... you need... automated tests! Unit tests, functional tests, performance tests, etc. Just make sure your tests cover pretty much all the functionality of the code you want to refactor. There are tools available to highlight the lines of your code that are covered by tests. Or, you can also modify one line of code and check if any of the tests fail, although this approach is not very productive.

If you are confident about the test coverage, you can do pretty much anything you want. Whatever code you dislike, simply discard the code and rewrite it from scratch. Alternatively, utilize a third-party library if one is available. As long as the tests pass, you are most likely fine.

Keep Refactorings Small

Most refactoring is usually minor in scale. Renaming a variable. Breaking up a class into two new classes. Removing duplicate code. Extracting functions. Meanwhile, rewrites of entire features are relatively uncommon.

The biggest mistake one can make with refactoring is waiting too long. If you have a gut feeling that your fundamental data structure could be an obstacle, you should act right away. Discuss with your work colleagues whether this is truly the correct choice and explore alternative options. Peripheral code can still be refactored later on. But if the core of your code is rotten, you will have a big issue fixing it. And it will only get worse if you don't act quickly. As always, the core of your code needs the highest priority.

Probably you do some smaller refactorings quite often. But not really in a structured manner. You refactor code as soon as you encounter something you don't like. This is honorable. But there is a very simple workflow that I can recommend to everyone. It's: write code – test – refactor. For every feature you implement, you should follow this pattern. Or even better, you can also write tests - code - refactor, as explained in the section on Test-Driven Development [](chapter Writing Better Code with Tests). This pattern is great because you can focus on one thing at a time. You can start by writing mediocre code. Maybe you are unsure about how to name a variable, or you might be inclined to create a class that is too large. There may be duplicated code. Certainly, it would be better to write flawless code from the outset. But you cannot multitask. You cannot develop code and make it perfect at the same time. You're not perfect. Learn to deal with your imperfections and refactor your code.

Then you write the tests. Some tests may fail because your imperfect code might contain bugs. When you fix the bugs, the code becomes even more messy. Even if you had written sublime code to begin with, due to the inevitable bug fixes, you would still have to refactor at some point. This is something that was overlooked by the waterfall development process. You never write perfect code to start with. You always miss some details that you have to fix later on. It always takes some refactoring to end up with good code.

Finally, you refactor. You review all the code you have written since the last time you refactored. Possibly, there is also existing code that has been around for a long time and could be merged with your new code because it is very similar. The code will probably look more complicated than you would expect. Then you try to rethink the logic of the problem you just solved. Can you modify the algorithm so that you can eliminate all the `if` statements for the corner cases? Do you need to sort your data structures differently to improve the code?

There are hundreds of things you could do to improve the quality of the code. When examining the code, identify the key elements that require modification. Try to write good code and trust your instincts. But make sure you also get some real work done between the refactoring sessions. The code will never be perfect. But at some point it will be good enough. Move on once this is the case. Don't get stuck in endless discussions about variable names. Go ahead and write some new code again.

Levels of Refactoring

Maybe you have a simple question: At what level should you refactor? Should you only refactor small elements, or should you delve deep into the core of your software?

Let me provide another brief example. Lets assume you are planning to build a house and enjoy cooking. Ensure that there is ample space in the kitchen for all your equipment. You are very pleased. This is the equivalent of a first draft of your code. Everything looks perfect.

Yet dishes get dirty, so you have to clean up the kitchen every day. Otherwise, you'd be left with a huge mess in no time. This corresponds to the everyday refactoring done by a software engineer. Ensure you eliminate code duplications, name all variables appropriately, and clean up anything you find unsatisfactory along the way.

Over time, as you occasionally purchase additional kitchen gadgets, you may find yourself running out of space. You need to sort out all the old devices you no longer need and utilize your Tetris skills to neatly fit everything back onto the shelves in an organized manner. Make sure you can still find your belongings. This is an intermediate refactoring.

At some point, you buy another device and realize there is not enough space for your equipment anymore. There is only one solution. You need a bigger kitchen. You need to plan how much additional space you will require for the next few years and decide whether to tear down some walls or expand your house. Now, this will be a very demanding and expensive refactoring.

I hope you received the memo. Small refactoring should be done all the time. Every few lines of code. The costs are low, and it keeps your workspace clean. Intermediate refactoring costs more and affects a significant portion of your codebase. It should be discussed with your work colleagues during the coffee break and may be done together. Big refactoring is really labor-intensive. It is done only every few months and requires careful planning and dedicated meetings because there is a lot at stake.

Refactoring is Dynamic

Waterfall refactoring is bound to fail for the same reasons that most waterfall projects do. Refactoring is concrete. Just like regular coding. It consists of a learning process. It's a feedback loop. Refactoring usually needs to be done incrementally, and endless planning sessions are a waste of time. Every couple of lines you write, you learn so many new things that require you to adapt the refactoring plans. Possibly, you may even have to abandon these grand plans altogether because you realize they just won't work. You can have as many beautiful plans as you want. If they don't work out, they are worthless.

You have to face the facts. Waterfall refactoring is not working out. Instead, you have to follow the actual dynamics of making changes, learning more about your code, and adapting your future changes. These three steps are the only way refactoring can be done.

// Make circle graphic: changes to be made, make changes, more changes to be made

Refactoring certainly has the highest impact when you have gained a new understanding of the problem you are trying to solve. This feature allows you to rewrite an entire piece of code at once, enabling you to make significant progress in improving your code quality. Eric Evans refers to this as "Refactoring towards deeper insight", [Domain-Driven Design].

The Circle of Doom

There is something very mean about refactoring. Refactoring good code is easier than refactoring bad code. For example, dealing with code that includes global variables, numerous dependencies, and large classes is always challenging, whether you are writing new code, tests, or performing refactoring. In all cases, you have to understand what the code really does. For writing new code and tests, this is bad enough. But with refactoring, it can become a nightmare due to the presence of a "circle of doom." You may find yourself delaying refactoring tasks because it can be challenging to comprehend poorly written code. But over time, this will only exacerbate the situation. Until you reach the point where refactoring becomes essentially impossible, and you are paralyzed. You would need to refactor your code because it is poorly written, but you are unable to do so because it has deteriorated to a point where improvement is no longer possible.

Don't slack off on refactoring. You'll pay the price sooner rather than later. Make sure you always keep your code well-organized; this will greatly simplify your life. And always remember to keep writing tests. Missing tests are the most obvious sign that your code is getting out of hand.

However, there is one thing you should always consider while refactoring: even if you don't like the behavior of your code as you are refactoring, you should not change it. The behavior of the software may not be changed. Even if it's a bug, you should reconsider fixing it because users may rely on that bug.

When to Refactor

It is generally a good idea to do refactoring. Most developers do too little refactoring rather than too much. Still, there are some general recommendations on when to refactor or not.

Every few lines of code you write, you should consider refactoring them. It is not always necessary, though it is by far the best moment. You still know what you just programmed, and you might have an idea of what is left to improve. Maybe you just introduced some code duplication? Additionally, you are always working in a tidy workplace, which increases your productivity. Code that is well taken care of is much easier to modify.

As mentioned earlier, you should always refactor the code you have just written. This is the number one rule. Furthermore, you should adhere to the Boy Scout rule: "Leave the campground tidier than you found it." Always refactor a little more than you should. This helps fight code entropy.

Refactor when you find a bug. Don't just add a patch that might resolve the issue superficially. Search for the actual source of the problem. Consider if there is any redundant code that needs to be fixed or refactored for better efficiency. Find a viable solution for the bug, which may involve some refactoring. This is time well spent because usually bugs tend to cluster. So if you found one bug, there might be more.

If you add a new feature, it may not seamlessly integrate into the existing codebase. Most likely, the code has not been cleaned up, or the other authors simply didn't know how the code should look in the future. Hence, the code has a different structure than what is required for this new feature. But now, as you're adding this new feature, you're more intelligent. You might have an idea of how the code should really look for the feature to fit in. Now, don't force the feature into the existing codebase. Refactor instead and ensure the new feature integrates seamlessly. Maybe transform a data structure as explained in the section on Orthogonality [](chapter Single Responsability Principle). Altogether, this is less work. And especially, the code will ultimately be in a much better condition. Write an adapter to seamlessly integrate the feature into the existing code base.

During code review, you can also perform refactoring. Team up with the code author and engage in pair programming. This type of review is much more motivating than a standard one because it facilitates better knowledge exchange and significantly enhances the review's output.

What to Refactor

Generally, you should refactor the code that you work with. In some cases, you may refactor code that you have just walked by, but this should not be the rule. If there is no reason for you to touch that code at the moment, you shouldn't refactor it. It is important in software engineering to know when to postpone certain tasks. And this is one of the cases. If no one is currently working with a particular piece of code, there is no immediate need to refactor it.

Once in a while, you have to do a significant refactoring. One that you don't just do between writing a few lines of code, but it will take considerable effort to get it done. You should probably discuss this topic with your work colleagues, as opposed to the smaller refactorings that you do on your own.

Last but not least, it is your code. You are responsible. You are the one to decide when it's time for refactoring. Don't ask your boss for permission to refactor. Just do it when you have to.

Refactoring Process

Writing code follows a similar process to the one I use when writing this book. I first started by jotting down the basic ideas. A rough draft of the content I wanted to include in this book. Some ideas I had for a long time, while others I acquired while reading other books. I read the text repeatedly, reworking it several times. I clarified points, removed redundancies, rearranged chapters, and added explanations where necessary. Every time I started to understand my text better, I could further improve it. Until I reached the point where the text said what I wanted it to. Until I had gathered all my knowledge from my head and organized it into a human readable text. Or as Ward Cunningham put it: "By refactoring, I move the understanding from my head into the code."

35. Refactoring Techniques

"The fewer methods a class has, the better. The fewer variables a function knows about, the better. The fewer instance variables a class has, the better" - Robert C. Martin [Clean Code]

The techniques explained here mostly require an existing set of automated tests because changes to the code may introduce bugs otherwise. Refactoring can also be done without tests. In most cases, this game is very dangerous to play. Even if some techniques seem safe to be applied without tests, there is always some latent danger of breaking the code in some way. Especially if you have global variables or overridden functions, it becomes tricky. Refactoring code in compiled languages is easier than in interpreted languages because the compiler performs valuable checks on names, functions, types, and so on.

There is a wide range of concrete refactoring techniques to be applied in specific cases. I will only briefly explain some of them. Most of the concepts originate from the book "Refactoring" by Martin Fowler [] (Refactoring, Improving the Design of Existing Code). In the following, I will group these techniques into two categories: one category mostly explained in [Refactoring](Refactoring, Improving the Design of Existing Code) for good code, and the other category from [Working Effectively with Legacy Code](Working Effectively with Legacy Code, Feathers) for bad legacy code with global variables, inheritance, no tests, etc.

The techniques explained in this chapter are unlikely to create bugs. The good refactoring usually consists of a sequence of small changes and the code should almost be in a working state all the time. However you never know. And in very convoluted code, I wouldn't dare to break a class into pieces without having it covered with unit tests.

When following the rules taught in this book, you should be writing good code. It is well-tested, contains clear interfaces, no global variables, and no side effects. Still, you have to refactor once in a while. But it's comparatively easy because you can focus on the refactoring part. The tests are already in place to ensure that you don't break anything. In this section, you will learn some techniques that you can apply when refactoring.

Where to start

Usually it takes me very little time to see whether some code needs to be refactored and if yes, what the problems are. The most frequent problem is classes and functions which are way too long. Those are also very easy to spot, but unfortunatley they are not always that easy to fix. Along with too long functions comes the problem of too many levels of indentation. This is another sign that the logic of the code should be simplified.

Missing unit tests is also a very common problem. Though this one is even harder to fix in a code review. You simply lack information on what a piece of code does. Furthermore, it is certainly the task of the original programmer to write tests, not yours.

Another common problem that is easy to spot are comments. If the expression used to explain something in a comment is different than the name of the variable, something is off. Usually the variable should be renamed. Though renaming has not the highest priority. It is more important to fix the logic of the code.

Breaking classes

Breaking classes into smaller pieces is one of the most commonly used refactoring processes. As it is tempting to write too big classes to begin with, and due to the fact that classes tend to grow over time, they have to be split once in a while. Classes should be small. And in my opinion, the best classes are the ones that don't exist. Use dataclasses and a few functions instead to organize your code. Functional programming does have its merits.

Too many methods

One issue of classes is excessive usage of methods. As I've already explained before, I prefer having freestanding functions over methods. They are decoupled and you can deal with them more freely. For instance, you can move these functions into a new class if you like, with only minor effort. Furthermore, it is easier to write unit tests for freestanding functions compared to methods. The technique behind it is fairly simple. Just search for functions that use only "few" class variables. Remove this method from the class and create a function out of it. Instead of passing `self` in Python, you have to pass all the function arguments explicitly.

```python
class House:
    def __init__(self, address):
        self._address = address

    def _print_address(self):
        # do something more with _address
        print(self._address)

    def wrapper(self):
        # do something with _address
        self._print_address()
```

Here we can refactor the `_print_address` method out of the class. The only variable needed by this method is `_address`. So we have to pass this variable as a function argument.

```python
def _print_address(address):
    # do something more with address
    print(address)

class House:
    def __init__(self, address):
        self._address = address

    def wrapper(self):
        # do something with _address
        _print_address(self._address)
```

This is of course a very artificial example and the question is, wherther it's worth doing this refactoring. If the class is very small, it doesn't really matter. But as the class grows, it becomes more and more important to increase cohesion and refactor out everything that doesn't need to be within the class. Here, `_print_address` is such an example. It requires only one member variable from the class, so it has comparably little cohesion. It is therefore a candidate to be refactored out into a separate function. If you don't refactor such methods out of your classes, your code becomes rigid and making changes to the class will require a lot of effort.

In our example, we were able to reduce the number of methods that need access to `_address` from 2 to 1. This is a very significant improvement. If you now have to change the shape of the class by changing the `_address` member variable, it has become much easier. You'll only have to change one method class.

Of course, this example is so small that you could completely dismantle the class. It has only one variale that is only used in one method. This is not what you should write a class for. The only class like thing here should be the `address`, which should probably be a dataclass.

Structuring variables

If variables are always used together in the same methods, they have high cohesion. This means that they probably belong together. They should be stored in a dataclass. Let's look at the following example:

```
class Fish:
    def __init__(self, name, age, weight):
        self._name = name
        self._age = age
        self._weight = weight
        self._price = 10.0

    def print_fish(self):
        print(f"{self._name} is {self._age} years old and weighs {self._weight}
kg.")
```

Here, the variables `_name`, `_age` and `_weight` are all used together. So it makes sense to store them in a dedicated dataclass. Note that `PersonalDetails` is not really a property you would expect from a fish. On the other hand, fish usually don't have a name. It's all just an example.

```
from dataclasses import dataclass

@dataclass
class PersonalDetails:
    name: str
    age: int
    weight: float

    def __str__(self):
        return f"{self.name} is {self.age} years old and weighs {self.weight} kg."

class Fish:
    def __init__(self, personal_details):
        self._personal_details = personal_details
        self._price = 10.0
```

```
    def print_fish(self):
        print(self._personal_details)
```

And probably we could rename the `print_fish` method into `print_personal_details` since it only prints the personal details of the fish. As the `print_fish` method now depends only on the `_personal_details` variable, we could also make it a freestanding function that takes only this variable as an argument.

Possibly this code above is still coupled too strongly. The `personal_details` and the `price` are not part of the same domain. It might make sense to store the price in a dictionary with the name separate from the other information. So if you want to have complete information about the fish, you would have to combine the information from the `personal_details` and the `price` dictionary. That might be the best solution. But this is a different story.

Too many variales

A very common problem is classes having too many variables. This is a so called "God Class". It does too much and should be broken into smaller pieces. There are several techniques how to deal with this issue. First of all, you should look at the variables. The search function of the IDE will tell you, how often a variable is used within a class. If a variable is used only two or three times and the class is 100 lines long, you should try to remove the variable from the class. It has too little cohesion. Though, this is easier said than done. You will have to figure out how the few methods using this variable get access to it. But maybe this is also a sign that the class design is bad and you should refactor these few methods out of the class as well.

Let's take a look at the following class `Boat`. It is not that big because I already refactored out the `Appearance` into a dedicated dataclass.

```
from dataclasses import dataclass

@dataclass
class Appearance:
    COLOR: str # should be an enum, but for brevity I use a string
    WEIGHT: float
    LENGTH: float

    def __str__(self):
        return f"The boat is {self.COLOR}, {self.LENGTH} meters long and weighs
{self.WEIGHT}."

class Boat:
    # The __init__ function should be refactored and not take that many arguments.
    # But that's a different story.
    def __init__(self, engine, price, color, weight, length):
        self._engine = engine
        self.PRICE = price
        self.APPEARANCE = Appearance(color, weight, length)

    def describe_appearance(self):
        print(self.APPEARANCE)

    # ...
```

We have all these physical properties of the boat where the `price` doesn't really fit into. So I'd like to split the class into two pieces. One class contains all the physical properties that I call `BoatFramework`, the other thing is the price, where we don't even need a dedicated class for.

```python
from dataclasses import dataclass

@dataclass
class Appearance:
    COLOR: str # should be an enum, but for brevity I use a string
    WEIGHT: float
    LENGTH: float

    def __str__(self):
        return f"The boat is {self.COLOR}, {self.LENGTH} meters long and weighs 
{self.WEIGHT}."

class BoatFramework:
    def __init__(self, engine, color, weight, length):
        self._engine = engine
        self.APPEARANCE = Appearance(color, weight, length)

    # ...

class Boat:
    def __init__(self, engine, color, weight, length, price):
        self._boat = BoatFramework(engine, color, weight, length)
        self.PRICE = price

    def describe_appearance(self):
        print(self._boat.APPREARANCE)

    def describe_price(self):
        print(f"The boat costs {self.PRICE}.")

    # ...
```

Usually these two techniques shown here are difficult to implement because the variables are used in many methods and it's not clear how they could be sorted in a better way. Furthermore when breaking a class into smaller pieces it requires new names. This may be a difficult task, but at the same time it is also an indication whether the way you broke the classes appart makes sense. If you can't find a reasonable way you probably designed your class badly.

Splitting classes

One of the hardest tasks is splitting big classes into pieces. Here we have a very small, but suboptimal class that I made deliberately that bad. Here everything is implemented inside the `Shopping` class. This is bad, because for every comestible, you have to extend the class. This violates the Open Closed Principle (OCP) [] (chapter SOLID).

```python
class Shopping:
    def __init__(self):
        self._apple_tracker = []
        self._eggs_tracker = []

    def get_shopping_items(self):
        return self._apple_tracker + self._eggs_tracker

    def add_apple(self):
        self._apple_tracker.append('apple')

    def add_egg(self):
        self._eggs_tracker.append('egg')

shopping = Shopping()
shopping.add_apple()
shopping.add_egg()
print(shopping.get_shopping_items())
```

Instead you should have a class for each comestible and inside the `Shopping` class, you have only one list for all different comestibles at the same time. The different behavior is implemented inside ever individual class, obeying the OCP. This allows you to simply add a new comestible without changing the `Shopping` class.

Also testing this class may be hard. Possibly you'll have to use a mocking library to alter the functionality of some functions that are implemented inside this class. This is a sign that the class is badly designed. Instead we can use dependency injection to inject the comestibles into the class. The class itself just contains one list storing all the items together.

Note that the code is now slightly different and much longer. We don't have separate `apples` and `eggs` lists anymore. We have a list of `items` that can be of any type. Therefore the order in which the elements are stored may be different. But this is probably not a big deal since all elements inside a list should be treated equally. On the other hand, the `apples` and `eggs` were stored in separate lists, but they were pirvate and never used alone. So there is no problem with storing them in one list together.

```python
from abc import ABC, abstractmethod

class Shopping:
    def __init__(self):
        self._shopping_basket = []

    def add_item(self, item):
        self._shopping_basket.append(item)

    def get_shopping_items(self):
        return [str(item) for item in self._shopping_basket]

class ShoppingItem(ABC):
    @abstractmethod
    def __str__(self):
```

```
        pass

class Apple(ShoppingItem):
    def __str__(self):
        return 'apple'

class Egg(ShoppingItem):
    def __str__(self):
        return 'egg'

shopping = Shopping()
shopping.add_item(Apple())
shopping.add_item(Egg())
print(shopping.get_shopping_items())
```

This is of course a very simple example and I made some changes to the functionality which is usually inacceptable. In reality the functions may be much more complex than this. But still, I think it was worth splitting up this class in many smaller classes because it can now be extended much more easily.

For testing purposes, you can also create a `FakeShoppingItem` if you want. This entire technique here we have already learned in the section on Dependency Injection (DI). This code here is similar to a factory.

Renaming

Even though renaming barely alters the structure of the code, it should be done frequently. Not only for good code, but also for legacy code. Finding good names is one of the most challenging tasks in programming because assessing the quality of names is very difficult. There are some general rules on how naming should be done, yet it is still not easy at all. This leads to the fact that there are many objects with suboptimal names. And as you write some code, it may happen that you spot something for which you happen to know a better name. Then rename this object. This is the only way names improve over time. Don't assume the author of the code knew it better. You now have much more information at hand that simplifies finding a good name.

Though you have to pay attention. People get used to names. If a name for an object has become familiar to the entire development team, you shouldn't change it, even if you have a better name. Renaming it would cause too much confusion. For this reason, it is better to name central elements of your code at the beginning of the development and not change them later on.

One possibility is to start with mediocre names initially and then search for better names towards the end of programming a few lines. Then, Copilot can also help you find better names.

Note that renaming is generally a fairly save refactoring technique and there are plenty of tools helping you. In many cases, the search function of your IDE may do the job as well. However, things may get tricky if you use global variables or if you have overridden functions. In this case, you should be very careful because it's easy to miss some variables or functions.

Scratch refactoring

In chess, a rule of thumb suggests that you should silently communicate with your pieces during your opponent's turn. You should ask them where they would like to be and thus get a sense of their preferred position. In programming, there is something quite similar. Scratch refactoring [](Working Effectively with

Legacy Code) is not about improving code; it is only about getting an idea of how the code could look. Just refactor as you like without worrying about bugs or similar issues. Figure out how the code should look in an ideal scenario. But also try to implement some of the edge cases to challenge your dream implementation and understand its limitations. I like the concept of scratch refactoring very much because it gives you an idea of how the code could look instead.

Once you're done refactoring, discard everything and do a regular refactoring, attempting to implement the ideas you just acquired. Pay attention to not simply reimplementing the code you envisioned previously. You may have overlooked certain technical details, and the solution from scratch refactoring may not turn out as expected. After all, the scratch refactoring was just a dream...

Extract function

If I have a function or method that is too long or not cohesive enough, I can replace some of the code with a newly created function. This is one of the most important refactoring techniques because excessively long functions are a common issue, and extracting functions is the primary method to manage them. If there are not too many variables involved, the technique is fairly simple. The biggest challenge is finding suitable names for the newly created functions.

Let's consider this very simple code snippet. We have already observed that this violates the SRP since printing a string and calling a function represent two different levels of abstraction.

```
def print_content():
    # print some stuff

print("author: Marco Gähler")
print("********************")
print_content()
```

The solution is taking the explicit print statements into a dedicated function and call this function instead.

```
def print_header():
    print("author: Marco Gähler")
    print("********************")

def print_content():
    # print some stuff

print_header()
print_content()
```

For once, you are allowed to use copy-paste to create the new function since the old code will be deleted anyway. Even better is to cut (Ctrl+X) and paste the code snippet, but that's a detail.

You can also utilize Copilot to extract this function. Just write the command "move the print statements into a dedicated function," and Copilot will take care of the rest for you. As always, you should pay attention to ensure that the solution is correct. In this case, it happened to me that Copilot suggested an incorrect solution.

There is not much more to know about extracting functions than what I just showed here. It is a simple yet crucial refactoring technique. This is probably the most commonly used refactoring technique besides renaming. The only thing you have to watch out for is the variables used by the newly created function. If the code is inside the class, you might consider making the function a member function of the class as well. Otherwise, you may end up having to pass too many arguments to the function. Though this would be a sign of poor class design because the class has too many member variables [](chapter classes). You may extract methods from this class later on if needed.

Inlining functions

Inlining functions is the opposite process of what we have just seen and is rarely used. Replace a function call with the function body. Apparently, this makes the surrounding function longer when the copied function body has more than one line. This is generally not desirable because most functions are already long enough. Inlining functions only makes sense for one- or maybe two-line long functions, or if you are planning to refactor the surrounding function and split up the old function. One advantage of inlining functions is that you don't have to come up with a function name. Though usually this isn't a good sign if you don't know how to call a function.

Dependency Injector

As we have already seen, Dependency Injection (DI) is a very helpful tool. Among other benefits, it allows us to inject mock objects to test functionality that would otherwise be untestable. So, the question is: Can we retrofit DI onto an existing class?

The answer is yes. And it is fairly simple. Especially in interpreted languages, it's super simple, though it is also more dangerous than in compiled languages. In compiled languages, the compiler will notify you if you have forgotten to update one of the function calls to match the new signature. In interpreted languages, you have to rely on your tests to find out if you forgot to adapt one of the function calls. This is more dangerous because you may have forgotten a function, and the code will fail in production.

Let's assume we once again have our `Car` class containing an engine.

```python
class Engine:
    def __init__(self, power):
        self.power = power

class Car:
    def __init__(self):
        self.engine = Engine(power=100)

    def drive(self):
        print(f"Driving with power {self.engine.power}")

car = Car()
car.drive()
```

The `Engine` object should not be instantiated inside the `Car` constructor for several reasons. Most importantly, the constructor should be as simple as possible. It shouldn't create anything that isn't necessary. It is better to pass the `Engine` as an argument. Instead, the `Car` constructor should look like this:

```
# ...

class Car:
    def __init__(self, engine):
        self.engine = engine

    # ...

car = Car(Engine(power=100))
car.drive()
```

For more refactoring examples, I would like to refer the interested reader to the book Refactoring Second Edition. It contains numerous examples and is an excellent resource for learning how to refactor code. And, of course, I recommend doing a lot of refactoring yourself in order to learn it.

Copilot

It seems as if Copilot sometimes struggles with code refactoring. At times, it doesn't find a reasonable refactoring suggestion. Though with a little experimentation and providing Copilot with some hints, it works out quite well. Here is an example:

```
def roman_number(number):
    if number == 1:
        return 'I'
    elif number == 2:
        return 'II'
    elif number == 3:
        return 'III'
    # Code used for later:
    # elif number == 4:
    #     return 'IV'
    # elif number == 5:
    #     return 'V'
```

I commented out the last two cases that were suggested by Copilot. Having only the first 3 cases and asking Copilot to refactor this code, it makes the following suggestion:

```
    # refactor this code to use a dictionary instead of if/elif/else
    return {1: 'I', 2: 'II', 3: 'III'}.get(n, 'Invalid number')
```

This works out but it's not quite what I wanted. After commenting out the case for 4 and 5 and explicitly asking for a while loop, the code looks as follows:

```python
# refactor this code using a while loop
roman = ''
while number > 0:
    if number >= 5:
        roman += 'V'
        number -= 5
    elif number >= 4:
        roman += 'IV'
        number -= 4
    elif number >= 1:
        roman += 'I'
        number -= 1
```

This is almost what I wanted, except that one could use a dict with number-roman number pairs. Asking explicitly for using a dict, the code looks pretty much how I wanted it to be.

```python
# refactor this code using a dictionary
roman = ''
roman_map = {1: 'I', 4: 'IV', 5: 'V'}
while number > 0:
    for key in sorted(roman_map.keys(), reverse=True):
        if number >= key:
            roman += roman_map[key]
            number -= key
            break
return roman
```

I particularly like the part with `for key in sorted(roman_map.keys(), reverse=True):` where Copilot makes sure that the entries in the dict are sorted, even if the dict itself is not sorted as it was the case before Python 3.7. In [chapter Testing] I used a list with class objects to achieve the same result. But I have to admit that the solution here is quite neat.

This code can be further refactored with the following command:

```python
# refactor this code using a for loop
roman = ''
roman_map = {1: 'I', 4: 'IV', 5: 'V'}
for key in sorted(roman_map.keys(), reverse=True):
    while number >= key:
        roman += roman_map[key]
        number -= key
return roman
```

As always, Copilot works best if you give it some step by step instructions. It is not always able to find the best solution by itself. Though it is still a great help for refactoring code.

36. Refactoring Legacy Code

"To me, legacy code is code without tests." - Michael Feathers

Up to this point, everything was great. We had no restrictions whatsoever. We assumed we were working on a greenfield project. I could tell you whatever I wanted. There were no restrictions because of the existing code base. I told you to write unit tests, and you started writing unit tests. Now, however, we will start working with existing code bases. We will learn how to handle legacy code: code without tests [](Working Effectively with Legacy Code). Or even worse, code without interfaces.

So far, we have mostly discussed refactoring techniques for code that is covered by tests. Refactoring code without tests would be dangerous. But, unfortunately, this is precisely the issue with numerous projects. There are so many projects out there without tests. Due to global variables, functions with side effects, complex constructors, and missing interfaces, it is very challenging to write tests for them. In these cases, you may start to feel afraid to make changes to the code as you are supposed to do during refactoring. There's just too much that can break without testing. This is apparently a really bad thing. No one likes to live in fear. In your own code, you can prevent this situation by meticulously testing all the code you write. However, if you work on an existing project, you will have to face the demons.

When you start working on a project with bad code, you might be motivated to suggest a complete rewrite. You may do that, although I do not recommend it. A complete rewrite is rarely an option. It takes years, costs millions, and very often the final code is not significantly better. Generally, it is better to improve the existing code. Once you have identified something you want to enhance, you write tests and start refactoring. This may seem tedious to you, but you always have to consider that the code was written by many programmers over many years. It will not be fixed in a few months.

Refactoring untested code is usually a very hard task. There are entire books about it [](Working Effectively with Legacy Code), [](Refactoring Second Edition). And if the code is already pretty bad, refactoring becomes even harder. The most common issues on the macro level are:

1. No interfaces
2. No tests
3. Obscure code
4. No time (or budget) to fix it

And on the micro level we have a few more indications that things will get tough:

1. Functions with side effects and global variables
2. Huge classes and functions
3. Objects that are hard to construct
4. Inheritance chains

If you want to divide a large class into smaller parts, consider the following approach. It has no tests, and you are uncertain about the side effects it might have. This is bad because the functional changes introduced are bugs. The only way to prevent these changes is by having plenty of regression tests.

How can you refactor legacy code?

First of all, you have to change as little code as possible to implement tests.

1. Identify change points ("Seams")
2. Break dependencies
3. Write the tests
4. Make your changes
5. Refactor

The difficult points are numbers 1 and 2. The rest involves textbook refactoring. Though it will take time, especially when writing tests.

No Useful Interfaces

Any code always has at least two interfaces: the input and the output. When one or both of them are a GUI, it becomes nearly impossible to write functional tests for the software. Furthermore, you can't write unit tests if there are no internal interfaces. Without tests, it is nearly impossible to refactor and add interfaces. It is really bad, but it's not a lost cause. You can still try to refactor slowly. Though it will be painful, and you will constantly have to watch out for bugs. A salesperson will have to check your work constantly to ensure that you haven't introduced any bugs. The entire refactoring process will likely take years. Perhaps a complete rewrite is indeed the better choice. I hope you never find yourself in this kind of situation.

No Tests

Code without tests is one thing. One can still write them later on, even though it takes much more effort than doing it right away. The real issue is usually the low quality of the code. Code without tests tends to be of low quality. While it may have some interfaces, the classes are excessively large, making the instantiation of objects difficult. This is one of the few cases where you are officially allowed to cheat. You may change private methods to public in order to test them. Once the refactoring is done, you should make it private again. Though that may take a year or two. Once you have some test coverage, you can break the classes into smaller ones.

If you are working on an existing project, the test coverage may be insufficient. This is a serious issue. Not only from a technical point of view, but also from a political perspective. Due to the low test coverage, one might introduce bugs when refactoring. And the last person to touch the code becomes responsible for it because who else is supposed to know how it works? So it becomes yours to support. However, this is not what you intended. You only wanted to improve it, not own it. Ultimately, people are afraid of refactoring the code because they will become responsible for it, not so much because it would be difficult. Therefore, the developers stop refactoring, and the code decays even faster.

One trick to avoid this political issue is the so-called "onion layer code." Instead of fixing a piece of code in place, you can write a wrapper around it and address all the issues inside the wrapper. By doing this, you avoid taking ownership of the code, yet you can still fix bugs. However, this comes at a cost of having all these fairly useless wrapper layers around your code, where you could have fixed the code properly instead. Don't let politics get in the way of good code.

Extremely Long Functions

Let's be honest. A function, or even worse, a method of about a thousand lines is an absolute nightmare. It lacks interfaces. It is not tested. No one will ever understand it with all its corner cases. There are so many variables present that it is difficult for anyone to comprehend the state your code is in. It is absolutely impossible. No one is ever going to touch such a function. You might be able to make some minor adjustments, but you are not addressing the core issue. The only way to truly change it is through a complete rewrite. The hardest part is obtaining the specification of what the function has actually done so far. If bugs are absolutely not allowed, you'd better leave the function as it is. Just work your way around it and accept the fact that at some point you'll have to rewrite it.

Seams

Writing tests is a noble endeavor, but it is not always easy. As I explained previously, the ease with which you can write tests largely depends on the quality of your code. In order to write tests, you need something tangible to work with. Michael Feathers refers to this as a "seam." "A seam is a place where you can alter behavior in your program without editing in that place." [](Working Effectively with Legacy Code) Vice versa, you can edit it elsewhere, at the so-called enabling point.

There are several different ways to implement seams. The best seams are interfaces using dependency injection. They are very easy to deal with and resemble typical code. Just create a new implementation of the interface or inject it, and you are done.

Some of the seams described in Working Effectively with Legacy Code alter the behavior at the compiler level, either through the linker or the preprocessor. Needless to say, implementing such fancy seams is a rather desperate measure. Such techniques strongly resemble black magic and should be avoided. And they are only possible in old languages like C++.

The most common scenario involves passing function arguments. It is not mentioned in Working Effectively with Legacy Code and the following code is simply a less effective version of using dependency injection, but it still serves as a seam.

```
def f(debug):
    if(debug):
        # ...
    else:
        # ...
```

However, passing a boolean as done in the code above is generally considered bad design. It is much better making the choice earlier on and passing on an object by dependency injection. The code above should be used at the highest level and create objects that will be used with dependency injection. For example:

```
import sys

class Reader:
    def read(self):
        print("reading")

class DebugReader:
    def read(self):
        print("debug reading")

def main(reader):
    reader.read()
```

```python
if __name__ == "__main__":
    if "debug" in sys.argv:
        reader = DebugReader()
    else:
        reader = Reader()
    main(reader)
```

By doing this, you will never end up with a boolean flag that you need to resolve later on. This is a much better design. Just create the `DebugReader` immediately and pass it as a function argument. The only potential issue is that you might have to pass several debug objects around instead of just one boolean variable. In the unlikely event that you have multiple debug objects (typically, there are very few), you can organize them within a data class.

Now, in this case, the seam is where the `read` function is being called, and the enabling point can be anywhere from the command line call of this program to the function inside which the `read` function is called. In our case, we can write a test and provide a different reader to the `main` function. Thus, we have a great deal of flexibility when it comes to controlling our code. This is a good seam.

Problems with Missing Enabling Points

Usually, just passing a number or a string is not sufficient to significantly alter the behavior of the function. It only yields a slightly different result. Variables generally do not alter the control flow of your code. The only two things that should significantly alter the behavior of your code are booleans and DI objects. And since you are not supposed to use booleans, you are back to using DI, as explained above.

The piece of code you are holding in your hands, between the enabling point and the seam, may be too large, and you may not have a clear idea of what you should test exactly. In the extreme case, the only tests you can write are functional tests. This is the issue of missing interfaces. Writing tests along with, or even before, the code forces you to define enabling points and seams that are close together. It forces you to write interfaces, thereby promoting good code quality.

If your code doesn't have any interfaces or an API that you can use to write your tests, you are completely screwed. I'm sorry, there's no other way to say it. And no, I'm not exaggerating. Spaghetti code without tests can be an enormous issue and it appears that there is no clear solution. A friend of mine was developing gas turbines. One individual developed software that could generate a full CAD model of a turbine. Now, the problem was that this person retired, and the code was a 15,000-line-long mess. The company paid millions in a desperate attempt to refactor the code but failed. In the end, they just wrote a wrapper around this piece of code and left it as is.

Sketches

Creating sketches and diagrams can help you find ways to refactor your code. This doesn't have to be UML diagrams. It can be anything that helps you understand your code. It can be a form of temporary behavior or what Feathers referred to as a "scratch refactoring". Essentially, a draft code that provides a rough idea of how the final code might look, without addressing all the intricate details that make real refactoring challenging. These tools help you better understand your code and make it easier to write the actual refactoring code.

How do I get the Code under Test?

This is a difficult topic. First of all, you have to be aware of the magnitude of the problem you are facing. In well-written software, the test code is at least as long as the production code. In highly regulated environments such as the aerospace industry, it may take several times longer than that. If you want to write tests for a longer piece of software that has not been tested at all, it will most likely take years. So, achieving high test coverage is generally not feasible. You'll have to be more pragmatic than that.

What Tests should I write?

The tests you should write depend mostly on the code you have. In most cases, it is difficult to write tests because of the absence of interfaces. You'll have to work with whatever you have. Of course, it would usually be best to write unit tests. However, that might not be possible if classes cannot be instantiated, methods are excessively long, etc. It might be the only option to write functional tests using the API of your software. As we have learned in the chapters on testing, this is not ideal. Functional tests take too long to execute. But beggars can't be choosers. At least you have something to work with. Lack of an API to work with is even more challenging. In this case, you are completely screwed. Writing onion code, which consists of wrappers around existing software, may be the only option. This limitation significantly hinders your ability to implement new features.

Sprout Method

Let's say you have a method or function that you cannot test, but you need to add some functionality. How do you do that without deteriorating the code quality any further? The solution is to add a new function or method that you can test and call from the old function. This is called a sprout method (or function) [WELC p. 58].

Assume we have the following code that we cannot test for some reason. In reality, it would, of course, be much more complicated. I have simplified it enough to create a readable example.

```python
def post_entries(transactions, entries):
    # ...
    for entry in entries:
        entry.post()
    transactions.get_current().add(entries)
```

Now we only want to add the valid entries to the transactions and execute the `post` function. It seems as if we'd have to create a temporary list and add an if statement.

```python
def post_entries(transactions, entries):
    # ...
    valid_entries = []
    for entry in entries:
        if entry.is_valid():
            entry.post()
            valid_entries.append(entry)
    transactions.get_current().add(valid_entries)
```

This, however, makes the untestable code even more complex. Instead we can create a new function that extracts the new functionality. This new function can be tested, so you can apply TDD.

```
def get_valid_entries(entries):
    valid_entries = []
    for entry in entries:
        if entry.is_valid():
            valid_entries.append(entry)
    return valid_entries

def post_entries(transactions, entries):
    # ...
    valid_entries = get_valid_entries(entries) # new line
    for entry in valid_entries:
        entry.post()
    transactions.get_current().add(valid_entries)
```

```
def test_get_valid_entries():
    entries = [Entry(is_valid=True), Entry(is_valid=False), Entry(is_valid=True)]
    valid_entries = get_valid_entries(entries)
    assert len(valid_entries) == 2
```

So, we managed to add only one additional line of code to the original function. All the other code was placed inside the `get_valid_entries` function. This new function is now also unit tested.

37. Performance Optimization

"Premature optimization is the root of all evil." - Donald Knuth

No Optimization Needed

Performance is one of the most overestimated topics in programming. This has historic reasons. Computers used to be extremely slow and expensive. Therefore, it was worthwhile to spend a significant amount of time enhancing every aspect of your algorithm. Back in the day, low-level languages like Fortran or even Assembler allowed you to do so. But the performance of computers has been growing exponentially for the last 50 years, while the price of computers has dropped considerably. Modern programming languages, such as Python, are no longer prioritizing performance. But rather on usability. Simply because it is more important to write readable code than to write fast code.

As we have learned, the primary goals of a software engineer are to create value for the customer by writing code that is easy to understand, correct, and well covered with tests. Performance is not a primary objective. It is hardly ever an issue if the code is not optimized for performance. Hardly anyone cares about optimization anymore. Nowadays, computers are fast enough to run most standard programs at a reasonable speed without the need for optimization.

I'd like to point out that the coding style I recommend does not necessarily lead to optimized performance. In my explanations, I didn't care about performance until now. Instead, I was recommending coding style for readability and reusability. The problem is that all this polymorphism that I recommended requires lookups in the so-called v-table, and this is slow. There are YouTube videos that explain these things in great detail. So, yes, the code I recommend you to write is comparatively slow. But it does not matter. When do you need millions of function calls for this slow polymorphic code? Probably never. It is unlikely that the code I recommend you to write will ever be the bottleneck of your software.

One issue with performance optimization is the fact that modern processors have many cores. In order to use them, you have to parallelize your code. This task can be tedious, and even if you complete it, you may not always gain anything. The overhead of orchestrating results between the different cores may be significantly slower than the single-threaded version of your code.

Optimization Maybe Needed

Let's say you start developing an application that you believe requires high performance. You may be unsure about when to start optimizing the code. Right from the beginning? Should you plan your algorithms to be faster? How should you proceed?

First of all, it is not recommended to optimize the code at all. In fact, it is best to ignore the topic of performance for the time being. Write your code using the typical test-code-refactor work cycles [section TDD]. When done well, the result will be code that is modular, stable, easy to understand, and well-tested. Code that meets all your requirements, except for performance.

You may have felt the need to write highly optimized code to meet the performance requirements. But you didn't know for sure. And now is the time to test your assumption. If you need to execute your code just once and it requires 2 days to complete, consider running it during the weekend. Spending hours on optimization would be a waste of time.

If your code takes an hour to run and you use it every day, it is worth getting a profiler to check the bottlenecks of your code. Most code you encounter typically has very few bottlenecks. Usually, it involves complex calculations on a large data structure that scales worse than O(N*log(N)) . This is going to be the one and only point where you'll have to optimize. As you have written great code, it is very easy to identify this bottleneck using a profiler. For example, it turns out to be a self-written Fourier transformation operating on a list with 10,000 elements. As you start reading through the code, you realize that the algorithm you have implemented scales with O(N^2). Such poor scaling is typically unacceptable. When seeking advice, you turn to the internet. You can find Fourier transform libraries that scale with O(N*log(N)). As your code is well-structured, you can simply remove your custom Fourier transform function call, adjust your data structure slightly, and utilize the library you discovered. Now your code runs within seconds. Done. You won't have to worry about anything else.

Optimizing Certainly Needed

Finally, there are the few cases where you have to develop the software from scratch and focus on optimization. But these cases are very rare. These are mostly simulation software, games, websites containing a large amount of data, or infrastructure code for huge server farms where not only performance but also energy consumption is a major concern. If the code can be parallelized, it will become much more complicated due to the additional complexity involved in designing data structures and algorithms. As a very rough rule of thumb, it takes approximately twice the amount of time to write parallel (or distributed) code compared to linear code, but it can easily be much more than that. There is a lot to learn if you want to write high-performance code. But you won't be alone. You'll likely be working in a team where every team member knows much more about parallel programming than I do.

There are many small things you can do to optimize your code, such as manual loop unrolling. Keep your hands away! The performance gains are negligible. When working with a compiled language, the compiler can optimize such things much better than you can. Major algorithms should be the focus of improvement since they typically account for 90% of the runtime. Optimizing the remaining 10% is usually not worth it.

Always keep in mind that code written with a focus on performance rather than readability is always very challenging to maintain. Due to the complexity of the code, it becomes very hard to understand!

-# Part 8: Miscellaneous

38. Comments

"Code is like humor. When you have to explain it, it's bad." – Cory House

As a very short rule of thumb, comments should not explain *what* a piece of code does, but *why*. *What* can be understood by examining the code. With the *why*, this is not possible. Was the code written in response to a specific ticket? Add a comment and with the ticket number.

Comments are a double-edged sword. While they may be useful at times, they can also be a liability. You always have to make sure you keep them up to date as you would with any piece of documentation. Additionally, comments tend to be a remedy for fixing bad code. And this is certainly not the intended purpose of comments.

Bad comments

"Comments? Don’t."

"Why?"

```python
def add(a,b):
    # This function returns the sum of the two arguments
    return a + b
```

I've seen similar comments before. Apparently, the programmer thought it was a good idea to write this comment.

I do not share this opinion. In my opinion, this is a useless boilerplate comment. Read the function name. It precisely explains the function's purpose. If you are unsure, refer to the implementation. This is precisely what distinguishes good code. When you read a function name, you know what it does. Good code is self-documenting. There is barely any need for additional comments. This comment is a violation of the SRP. It's a redundant explanation of the code's functionality.

"Yes, but it's only one line of comment. It can't hurt us," you might say.

"NO!"

Sorry, I just lost my temper. I shouldn't be so harsh with you. Many experienced programmers don't know, so why should you? I have to tell you that you are wrong. You can't believe how wrong you are. Maybe I haven't made myself clear enough so far. This comment is an absolutely useless liability. It makes a claim that will not always be true. The code will change as code always does. But the comment may be forgotten. Unlike function definitions or variable names, you cannot enforce that a comment remains in its correct location. You will eventually end up with a comment that is simply incorrect. It will confuse everyone who works on this code. It will cost time.

Not convinced? Do you believe you won't encounter these issues because you work meticulously?

"Ha ha. NO!"

Now you're certainly wrong this time. By now, you should know better. This is precisely what I'm trying to teach you throughout this entire book. You are human. Every human makes mistakes. I make mistakes, you make mistakes. It's inevitable. Accept your fate and learn how to deal with it. Code is good if you can make as few mistakes as possible. Removing unnecessary comments is essential. They violate the third rule of software engineering, "We write code that can have as few possible bugs as possible." Such comments are an unnecessary source of bugs.

You want to become a software engineer. So, stop using the English language and start reading code instead. Get used to it. The code contains the absolute truth. Not the comment.

Here is an example from the book [The Art of Readable Code](The Art of Readable Code, Boswell & Foucher). The book has some good ideas, but there are some examples that could be further improved. The original code was written in C++ and I translated it into Python as pseudo code.

```
class FrontendServer:
    view_profile(request)
    open_database(location, user)
    save_profile(request)
    extract_query_param(request, param)
    reply_OK(request, html)
    find_friends(request)
    reply_not_found(request, error)
    close_database(location)
```

Undoubtedly, this code is bad. It is very hard to read. The code lacks any visible structure.

The authors of this book organized the code, added comments, and eventually produced the following result:

```
class FrontendServer:
    # Handlers
    view_profile(request)
    save_profile(request)
    find_friends(request)

    # Request/Reply Utilities
```

```
    extract_query_param(request, param)
    reply_OK(request, html)
    reply_not_found(request, error)

    # Database Helpers
    open_database(location, user)
    close_database(location)
```

The code has certainly become much more readable. But this refactoring can be taken one step further. Adding these comments does not solve the fundamental issue. The class should be divided into three subclasses, with one parent dataclass containing the class instances. This logically separates the different parts of the class. The comments are just a workaround for suboptimal code.

Here is my rough suggestion on how to rewrite the code above.

```
from dataclasses import dataclass

@dataclass
class FrontendServer:
    profile: Profile = Profile()
    request_handler: RequestHandler = RequestHandler()
    database_handler: DatabaseHandler = DatabaseHandler()

class Profile:
    view(request)
    save(request)
    find_friends(request)

class RequestHandler:
    extract_query_param(request, param)
    reply_OK(request, html)
    reply_not_found(request, error)

class DatabaseHandler:
    open_(location, user)
    close(location)

# example usage of this code:
server = FrontendServer()
server.profile.view(request)
```

The resulting code is once again longer than the initial version. But it is much better structured, and there is no need for any comments. Note how we were also able to simplify certain parts of the code. For instance, we now define the function `view` instead of `view_profile`. The profile part of the function name is now clear due to the context within the `Profile` class or the function call as `profile.view`.

Here is another example from the same book. It suffers from a similar problem: the authors attempted to enhance the code by adding comments rather than improving the code itself.

This is the original code.

```python
# Import the user's email contacts, and match them to users in our system.
# Then display a list of those users that he/she isn't already friends with.
def suggest_new_friends(user, email_password):
    friends = user.friends()
    friend_emails = set(f.email for f in friends)
    contacts = import_contacts(user.email, email_password)
    contact_emails = set(c.email for c in contacts)
    non_friend_emails = contact_emails - friend_emails
    suggested_friends = User.objects.select(email__in=non_friend_emails)
    display['user'] = user
    display['friends'] = friends
    display['suggested_friends'] = suggested_friends
    return render("suggested_friends.html", display)
```

After the refactoring suggested in the book, the code is already much more readable. But once again, the code should not be commented. It should be refactored.

```python
def suggest_new_friends(user, email_password):
    # Get the user's friends' email addresses.
    friends = user.friends()
    friend_emails = set(f.email for f in friends)

    # Import all email addresses from this user's email account.
    contacts = import_contacts(user.email, email_password)
    contact_emails = set(c.email for c in contacts)

    # Find matching users that they aren't already friends with.
    non_friend_emails = contact_emails - friend_emails
    suggested_friends = User.objects.select(email__in=non_friend_emails)

    # Display these lists on the page.
    display['user'] = user
    display['friends'] = friends
    display['suggested_friends'] = suggested_friends

    return render("suggested_friends.html", display)
```

Here is my suggestion.

```python
def suggest_new_friends(user, email_password):
    friend_emails = get_friends_emails_of(user)
    contact_emails = import_email_addresses_from(user, email_password)
    non_friend_emails = contact_emails - friend_emails

    suggested_friends = find_suggested_friends(non_friend_emails)

    dict_items = create_dict(user, friends, suggested_friends)
    return render("suggested_friends.html", dict_items)
```

```
def get_friends_emails_of(user):
    return set(friend.email for friend in user.friends())

def import_email_addresses_from(user, email_password):
    contacts = import_contacts(user.email, email_password)
    return set(contact.email for contact in contacts)

def find_suggested_friends(non_friend_emails):
    return User.objects.select(non_friend_emails)

def create_dict(user, friends, suggested_friends):
    items = {}
    items['user'] = user
    items['friends'] = friends
    items['suggested_friends'] = suggested_friends
    return items
```

This time, the code only became slightly longer compared to other refactoring examples. But at the same time, it is much more readable. You can understand its functionality simply by looking at the top-level function `suggest_new_friends`. You don't have to read the details of the function. You can read the function names to understand their purpose. This is what makes code readable. Not the comments.

At times, it is very difficult to explain code using code alone. So, there is, of course, the temptation to use a comment to make it clearer. As in the following example, also from the book The Art of Readable Code:

```
// Rearrange 'v' so that elements < pivot come before those >= pivot;
// Then return the largest 'i' for which v[i] < pivot (or -1 if none are < pivot)
int Partition(vector<int>* v, int pivot);
```

I must say, I don't like this comment. It is very difficult to understand. And as always, providing a comment to explain what code does is suboptimal. Now, the first issue I see with this function is that it performs two tasks simultaneously. It orders the elements of the vector and returns the index of the last element that is smaller than the pivot. It has a mutable argument and a return value simultaneously. This is a violation of the SRP. The function should probably be split into two parts.

Additionally, there is something else that can explain code: unit tests. The test cases act as examples of how the code is supposed to be used and serve as an example at the same time. This is often more helpful than a difficult-to-read comment.

Commented Out Code

Another thing you might have seem somewhere is commented out code. Someone was developing a feature. Maybe he was replacing some code and wasn't sure how to implement the new version. So, he commented out the old code and started implementing. He somehow didn't understand all the details, but at some point, everything seemed to work. He knew that he was guessing more than writing structured code. He knew his work was really bad. Therefore, he decided to leave the old code in the repository and just commented it out, right beside the new code.

Commenting out code is absolutely dreadful. This is one of the candidates for the worst programming practices. What are you supposed to do with commented out code? Everyone reads it. Nobody knows how to deal with it. It's just causing confusion and wasting everybody's time. If we only had a tool to browse the history of the code... Something like Git...

Never use comments (or dead code) for that purpose. You have my permission to delete any commented-out code that you ever see. You may use this book as proof if needed.

TODO Comments

Another bad habit is TODO comments. When you implement a feature, you are responsible for ensuring that the implementation is ready to be merged into the master branch. It's ready to be merged when there is nothing important left to be done that would justify a TODO comment. Make sure you never merge any TODOs into the master branch. These tasks only lead to confusion, and there is never enough time to complete them. You should never implement a feature without a corresponding ticket. Additionally, for code refactoring, there is no need for a TODO comment. Therefore, once again, make sure you never merge any TODO comments into the master branch.

During the development of a feature, it is acceptable to use TODO comments. It might help you to organize your work. Just make sure to remove all the TODO comments before merging your changes into the master branch.

Comments Replacing Code

Introducing numerous small functions can somewhat hinder readability. It involves keeping track of and navigating through the function calls. Though this cost is very low if the functions are named properly. If all the functions perform as described, you can simply read the function names to understand what the code does. This is what makes code readable. Not the comments.

As a summary, I can say that all the small functions come with a price to pay. But adding comments to explain the code is not the best solution.

Useful comments

So much about why not to use comments. Now let's discuss situations where using comments is entirely appropriate.

I have explained that you should not use comments for anything that could (or should) be explained by the code itself. Vice versa, this means that comments are allowed to explain things that you cannot express in code. For example, you can add links to the source of a code fragment, library, or an explanation of an algorithm. It may also be useful to use comments in the interface of a library or API when using documentation software. Comments are typically used at the beginning of the file to include the boilerplate copyright statement.

Requirements

A very legitimate use of comments is to document requirements. The code has to be written this way due to certain requirements. Requirements are something that cannot be explained in code. They are usually written in a natural language. Despite this, they are still highly important for the software. At times, the requirements are the only thing that can explain why a certain piece of code looks the way it does. And the only way to explain this is by using comments. Please add the ticket number to the comment, or even better, copy the requirement text into the comment as the ticket may be edited later on.

Usually, the requirements are also expressed in an acceptance test. And I hope you do write acceptance tests. But acceptance tests are not sufficient. They are not visible in the code. You have to search for them. It is unclear which acceptance test corresponds to each line of code. Therefore, comments are the only way I can think of to connect the code to the requirements.

How to write comments

Just like code, comments should be concise and meaningful. In the following example, we have the opposite. What does "it" in the following sentence mean? Please avoid writing ambiguous sentences. [](The Art of Readable Code)

```
# Insert the data into the cache, but check if it's too big first
```

better:

```
# If the data is small enough, insert it into the cache.
```

Docstring

You may use docstring tools, such as Sphinx in Python, for automatically generating documentation. However, docstrings should only be used for external documentation. Never use docstrings for internal purposes. Why should you read a docstring documentation if you can read the source code and all its comments?

For using docstrings as documentation for external users, comments are also very useful. Furthermore, when commenting on external APIs using docstrings, completely different rules apply than for internal comments. When documenting an API, it is crucial to comment on the *what* rather than the *why*. The user doesn't have access to the code, or at least he's not supposed to read it. So he solely relies on the docstring comments. Therefore, you have to comment *what* your functions and classes do, and explain how to use them. Possibly by adding examples. The *why*, on the other hand, is not important at all.

As a short summary: Docstrings are very useful for documenting your external APIs, but not for internal code. Contrary to normal comments, docstrings should comment on the *what* and not the *why*.

Commenting magic numbers

Here we have an example of poorly written code, this time in C++. I found it in [The Art of Readable Code]. The authors correctly state that this code is hard to understand and I suggest some changes on their solution.

```
connect(10, false)
```

This code is bad as it is very hard to understand what `10` and `false` exactly mean. You'd have to look up the function definition to understand it.

Copilot suggests to improve the code by adding a comment at the end of the line. Honestly, this is a pretty bad solution.

```
connect(10, false)  # timeout_ms = 10, use_encryption = false
```

The suggestion in the book was adding the comments inside the function call. This is possible in C++ but it's not a good solution. It's an attempt to make bad code better by commenting it.

```
connect(/* timeout_ms = */ 10, /* use_encryption = */ false);
```

In my opinion, this solution is still far from optimal - it uses comments.

There are two better solutions to this problem. In Python, C++20, and most other modern programming language, keyword arguments are supported.

```
connect(timeout_ms=10, use_encryption=false)
```

The other solution is creating intermediate variables. The function arguments used here are magic numbers that have to be avoided.

```
timeout_ms = 10
use_encryption = false
connect(timeout_ms, use_encryption)
```

Summary

Use comments only for things that cannot be made apparent by the code itself, yet you think it’s still very important. Comment the *why* and not the *what*. If you write docstrings, it's the other way around. Comment the *what*, rather than the *why*.

Copilot

Copilot is not yet able to write more than boilerplate comments. The following comment was created by the document function of Copilot Labs. Copilot makes the mistake to comment the *what*. Apparently there is no way for Copilot to find out *why* you write some code. Therefore, I would recommend Copiliot only to write docstrings.

I find it very remarkable that Copilot is able to write such a comment, however it is still fairly useless.

```python
def roman_number(number):
    # The roman_map dictionary is a lookup table that maps numbers to
    # roman numerals. It is used by the to_roman function to convert
    # numbers into roman numerals.
    roman_map = {1: 'I', 4: 'IV', 5: 'V', 9: 'IX', 10: 'X'}
    roman = ''
    for key in sorted(roman_map.keys(), reverse=True):
        while number >= key:
            roman += roman_map[key]
            number -= key
    return roman
```

39. Logging

The basic idea of logging is to provide feedback on the steps that your software executed. It might help you find bugs. Now this sounds great, but in reality, there are several things to consider.

- The most obvious drawback is that logging requires time to be implemented. It's not a huge amount, but it may add up.
- Logging pollutes your code, similar to comments.
- Logging is typically unnecessary. Most code is deterministic. If you run the same code twice, it will produce the exact same results, with minor differences due to finite precision rounding errors. You don't need the logs. Run the code with the same settings as the user did and inspect your code using a debugger.
- If you struggle to find bugs in your code, you should focus on improving its quality instead. Simplify the structure and write unit tests. You will have fewer bugs, and they are easier to find.

At the same time, there are some cases where you can consider using a logger.

- It may be useful for non-deterministic software. For example, if you have several programs that communicate asynchronously with each other, such as in microservices. Race conditions may occur that you hadn't considered. Depending on the temporal order of the messages being sent. A logger can help you trace back to the source of a bug. Though finding such bugs is challenging, even with the best logger. You may be feeling overwhelmed by the volume of log files.
- In a GUI the logger could store all the actions performed by the user. This may also be helpful if the user encounters a bug.
- Finally, a logger may be helpful for the user send in auto-generated error reports if something goes wrong. Users can simply click a button to submit an error report containing all relevant data, eliminating the need to manually write the report themselves. This may be very useful as errors are almost inevitable, and the users are a very helpful group to test your software. As long as the bugs are not too subtle or too serious.

40. Data files

There are several file formats available for saving data or using them as an interface. A lot of people apparently don't even know the most important ones, so I would like to give you a very short introduction.

The file formats that I have used so far include CSV, JSON, XML, HDF5, and databases. Along with some custom file formats. There are, of course, many more such as YAML, TOML, etc. But for the sake of brevity, I won't explain those. The file formats mentioned here are sufficient to complete your work, and it won't require much effort to learn other file formats if necessary.

CSV

Comma Separated Values (CSV) is probably the simplest and one of the most common file formats. You can save numbers by separating them with commas or any other character you prefer. It won't get any easier. But this is also one of the weaknesses of CSV. In some natural languages, such as German, the comma character is used as the decimal separator. Thus, you cannot use a comma to separate different values as well. This overload of the comma character could lead to significant issues when reading a CSV file.

There is no standard for CSV files, so you can format them however you like. And that's at the same time the downside of it. People do whatever they want, and for every file, you have to write a new piece of code to read the data. Saving auxiliary data in CSV files is quite challenging. CSV saves only plain, unstructured lists.

Long story short: CSV is the file format commonly used by those who are not familiar with more advanced options, such as JSON, for example. Or someone who works with Excel.

Here is an example code snippet demonstrating how to read a CSV file with two columns separated by commas. Lines starting with `#` are comments:

```
def is_comment(line):
    return line.startswith("#")

def read_csv(filename):
    with open(filename) as file:
        x = []
        y = []
        for line in file:
            if is_comment(line)
                continue
            variables = line.split(",")
            x.append(float(variables[0]))
            y.append(float(variables[1]))
    return (x, y)
```

Copilot

Copilot seems to be getting along with csv files quite well. Though I would have expected the data to be saved in columns instead of rows. But as I already said, there is no real standard for CSV files, thus such things have to be specified to Copilot.

```
x = [1,2,3,4]
y = [5,6,7,8]
# write x and y to a csv file called 'data.csv'
```

This prompts the following code:

```
with open('data.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['x', 'y'])
    for i in range(len(x)):
        writer.writerow([x[i], y[i]])
```

Besides the forgotten `import csv`, the code does pretty much what it should.

JSON

The JavaScript Object Notation (JSON) file format is probably one of the best file formats for everyday purposes. The libraries are very user-friendly. It can save any data structure you want and is extremely widespread and thus supported. There are libraries available to automate the parsing of JSON files for all major programming languages. The output data structure consists of a combination of nested maps and arrays. It won't get any easier to read a file into data.

Once you use JSON in a more serious project, you might want to consider using a schema to validate your files for accuracy. You may use different schemas for different versions of your interface. Before you manually write a schema, there are tools available to assist you. You only have to ensure that your JSON file contains all possible fields to obtain a complete schema.

Thanks to schemas, JSON is also a meta language. It is possible to define a general pattern for how the JSON file should look. This defines a standard that enables easy file exchange between different projects. You can use schemas to define the version of a JSON based interface.

The following code creates a JSON file:

```
import json

def write_json(filename, data):
    with open(filename, 'w') as f:
        f.write(json.dumps(data, indent=4))

if __name__ == "__main__":
    data = {'x': [1,2,3], 'y':[4,5,6]}
    write_json("temp.json", data)
```

Meanwhile this code here reads out the data.

```python
import json

def read_json(filename):
    with open(filename) as f:
        return json.load(f)

if __name__ == "__main__":
    data = read_json("temp.json")
    print(data['x']) # prints [1,2,3]
    print(data['y']) # prints [4,5,6]
```

As you can see, working with json files is much easier and less error prone than working with CSV files. The underlying data structure is a dict, which is a pretty bullet proof way to work with data. There is hardly a way to introduce unnoticed bugs.

Copilot

Reading and writing json files using Copilot works out pretty well. Upon writing the following code

```python
a = [1,2,3]
b = [4,5,6]
# write a and b to a json file called 'data.json'
```

Copilot makes the following suggestion:

```python
import json
with open('data.json', 'w') as f:
    json.dump({'a':a, 'b':b}, f)
```

Also when reading a json file, the suggestion of Copilot is quite sound. Following the comment

```python
# read the json file into a and b
```

Copilot makes the following suggestion:

```python
with open('data.json', 'r') as f:
    data = json.load(f)
    a = data['a']
    b = data['b']
```

Thus, one can say that Copilot enables us to save time when writing code and conserving mental energy when working with JSON files.

XML

The eXtensible Markup Language (XML) is very similar to JSON. It's slightly older than JSON, and it doesn't support arrays as elegantly as JSON. Otherwise, there are only minor differences between the two formats. One thing people might miss in JSON is the lack of the ability to add comments. On the other hand, JSON is generally considered to be more easily human-readable. Other than that, there are only minor differences, such as XML supporting comments and namespaces.

Generally, I prefer JSON over XML. If I need to read an XML file, I utilize tools that convert the data structure into a JSON object and work with it in a familiar manner. I don't have to learn something new like this.

The following code converts an XML file into a JSON object:

```
import json
import xmltodict

with open("data.xml") as xml_file:
    data_dict = xmltodict.parse(xml_file.read())
    json_data = json.dumps(data_dict)
```

Writing XML files is rather tedious as well. Here I recommend as well to work with json data and convert it to XML just before writing.

```
import json
import xmltodict
json_string="""{"employee": {"name": "John Doe", "age": "35", "job": {"title":
"Software Engineer"}}}"""

Python_dict=json.loads(json_string)
with open("person.xml", 'w') as file:
    xmltodict.unparse(Python_dict, output=file)
```

Copilot

Copilot somehow struggles a lot when dealing with XML files. It seems as if it didn't know the xml libraries that would simplify it's life. Instead it assembles strings into an XML file by itself which results in very brittle code that is very hard to read.

```
x = [1,2,3]
y = [4,5,6]
# write x and y to an xml file called 'data.xml'
```

The first version of Copilot I used wasn't able to write proper XML files. Instead of using a library it was creating the string from scratch:

```python
with open('data.xml', 'w') as f:
    f.write('<data>\n')
    for i in range(len(x)):
        f.write(f'\t<x>{x[i]}</x>\n')
        f.write(f'\t<y>{y[i]}</y>\n')
    f.write('</data>')
```

Now this does the job, but this is certainly not how an XML file is supposed to be created. Reading XML files is even worse.

```python
x = []
y = []
# read the xml file into x and y
```

The code Copilot suggests is extremely brittle and doesn't even work in my case. Somehow Copilot doesn't seem to know about XML libraries that would simplify its life a lot.

```python
with open('data.xml', 'r') as f:
    for line in f:
        if '<x>' in line:
            x.append(int(line[3:-5]))
        elif '<y>' in line:
            y.append(int(line[3:-5]))
```

But during the last few months, Copilot improved and now writing XML files is done with an ElementTree. This is certainly a much better approach.

```python
import xml.etree.ElementTree as ET
x = [1,2,3]
y = [4,5,6]

root = ET.Element("data")

x_elem = ET.SubElement(root, "x")
x_elem.text = ",".join(str(i) for i in x)

y_elem = ET.SubElement(root, "y")
y_elem.text = ",".join(str(i) for i in y)

tree = ET.ElementTree(root)
tree.write("data.xml")
```

HDF5

HDF5 is the most common binary file format. It is designed to handle terabytes of data and optimized for high throughput. Most research facilities and companies that handle large amounts of data use this file format. It supports structured and auxiliary data. To access the data, you can either utilize the HDF5 library in your preferred programming language or download the free GUI software. Use HDF5 if you need to store multiple gigabytes of numerical data.

Working with HDF5 is, in my opinion, slightly less intuitive than working with JSON files. This is because HDF5 uses datasets that need to be created instead of simply accepting a dictionary. Though with the current version of the HDF5 library, the code becomes quite simple.

The following code saves a list of values in an HDF5 file.

```
import h5py

with h5py.File("temp.hdf5", "w") as f:
    dset = f.create_dataset("x", data=[1, 2, 3])
```

Reading a file returns an HDF5 file object. It may be a little intimidating at first, but it is fairly easy to work with. With many respects, it behaves similar to a dictionary.

```
import h5py

with h5py.File('temp.hdf5', 'r') as f:
    print(list(f.keys()))
    print(list(f['x']))
```

As HDF5 is a binary format you cannot look at the data using a text editor. Instead you have to use the HDFview software, https://www.hdfgroup.org/downloads/hdfview/

Copilot

Copilot seems to be getting along quite well with HDF5. On the following code snippet:

```
x = [1,2,3,4]
# write x to an hdf5 file called 'data.hdf5'
```

Copilot correctly complements it to

```
with h5py.File('data.hdf5', 'w') as f:
    f.create_dataset('x', data=x)
```

Also reading out data from an HDF5 file is no problem.

```
# read the hdf5 file into x
with h5py.File('data.hdf5', 'r') as f:
    x = list(f['x'])
```

Databases

Databases (DB) are used for a large amount of structured data that you want to analyze but doesn't fit into memory. Databases have a wide range of functionalities that enhance searching and manipulating data within the database. There are several vendors offering different technologies.

I never really cared much about DBs and I'd like to teach you other things instead. So, you better get your information elsewhere. I only know that proprietary DBs can be extremely expensive, and it's important to write your code in a way that allows for easy replacement of the database with another one. Otherwise, you may find yourself locked into paying substantial annual fees.

Also, make sure that a database is not the core of your software. It's just a place to save and access data. It can be replaced with a text file if necessary!

SQLite is probably the easiest database to use. That's why I am providing a brief example here. Compared to other databases, creating or migrating an SQLite database is straightforward.

Here is the code to create a database and add some movie objects.

```
import sqlite3
con = sqlite3.connect("tutorial.db")
cur = con.cursor()
cur.execute("CREATE TABLE movie(title, year, score)")
cur.execute("""
    INSERT INTO movie VALUES
        ('Monty Python and the Holy Grail', 1975, 8.2)
""")
con.commit()
```

Copilot

The database can also be created by Copilot. Let's define a movie object.

```
import sqlite3

class Movie():
    def __init__(self, title, year, score) -> None:
        self.title = title
        self.year = year
        self.score = score

indiana_jones = Movie('Indy', 1981, 8.5)
```

Copilot

```
# create a SQLite database containing the indiana_jones movie
conn = sqlite3.connect('movies.db')
c = conn.cursor()

c.execute('''CREATE TABLE movies
             (title text, year integer, score real)''')

c.execute("INSERT INTO movies VALUES (?, ?, ?)", (indiana_jones.title,
indiana_jones.year, indiana_jones.score))

conn.commit()
conn.close()
```

Custom File Formats

Similar to the CSV file, you can also define your own file format for things other than just numbers. You can define your own file with structured data. You can even define your own programming language, such as structured text, within your custom file format. You can do pretty much anything you like. You are a free person. Just don't expect to be paid for such a waste of time. If you aspire to become a successful software engineer, you must prioritize delivering value to the customer. You need to utilize JSON, HDF5 or a database. There is no need to define custom file formats.

41. Setting up a project

"If it's your job to eat a frog, it's best to do it in the morning. And if it's your job to eat two frogs, it's best to eat the biggest one first." - Mark Twain

[https://youtu.be/LfIPVIsH4ZU]

Many software developers start by writing code immediately when they have a task to complete. And they postpone all infrastructure work for as long as possible. They continue to compile code using the command line for as long as possible. They don't use Git. And they certainly don't use a Continuous Integration (CI) tool. This is dreadful. Set up these things at the beginning of the project.

Yes, it will take some time to get started. And yes, it's a painful process if you are not accustomed to it. But it is worth it. The very first reason why it is worth it is Do not Repeat Yourself (DRY). If you find yourself repeatedly typing the compilation command into the terminal, you are essentially repeating the same action over and over again. This is going to slow down the development process. This is much worse than spending the same amount of time at the beginning of the process because it interrupts your thoughts.

For small projects, setting up Git and a proper build tool hardly takes any time. After encountering a challenging bug for the first time, you will appreciate having version control in place. This will allow you to easily revert back to your previous changes. The same holds true for the build process. Typing in many long commands not only takes time but is also brittle. It is too easy to make a typo and screw up the build process in some unforeseen way that introduces hard-to-understand behavior. And especially when you have to cooperate with other developers, there is no way around using proper version control software and a build tool.

Similarly, for CI. It will take some time to set it up. But you will save a lot of time later on because you can be sure that the tests you wrote (and I really hope you have tests; otherwise, I recommend you read the chapter on [testing](chapter Testing)) always run. The code committed to the master branch has been compiled, and the tests pass without any errors.

Yes, setting up the infrastructure of a project may take some time. I already heard that setting up a project may take up to 20% of the total development time! But it is certainly time well spent. There are so many advantages to having a properly set up infrastructure.

- You have to learn how to use Git, CMake, and all the other tools anyway. So, it's good practice to get started with them as soon as possible.
- You will save a lot of time in the future. This will outweigh the time needed to set everything up now.
- Having properly set up tools makes it easier for new team members to get started. Users only need to clone the repository and run the build tool to get started.

////## Project Folder

// Add a plot showing the folder structure.

Code is primarily a collection of text files. One question arises: how do you manage them?

The very first thing to consider is the length of each file. It is advisable to keep them short, ideally around 100 lines per file, with a few hundred lines being somewhat acceptable. Having numerous relatively small files enhances the overall organization. Typically, one file should either contain a class or a set of similar functions. Classes with over 1000 lines should have been divided into smaller parts long ago. Therefore, files should not exceed 1000 lines. In fact, files should generally be much smaller than this limit.

The organization of files into folders varies depending on the programming language. Typically, the code is stored within the src folder, further categorized into subfolders as needed. Each subfolder usually represents a library of the project. It is important to ensure that only your own code and, where applicable, your tests are placed in these folders. Avoid including any auto-generated files in the src folder at all costs. Auto-generated files should never be included in version control as they only clutter it.

Generated files should be placed in the build folder. This makes cleaning up the build process simple. To remove all build files, just delete the build folder. This approach also simplifies version control. Add the build folder to the .gitignore file to ensure that generated files are never included in version control.

Acceptance tests should also remain outside of the src folder as these tests are quite independent of the code. They only use the public API. I would keep them in a separate folder next to src, usually within the same git project. You may also have them outside of the repository or even hand over the responsibility to the sales team if everyone agrees.

3rd party libraries belong in the lib folder. They are not part of the git project; therefore, the lib folder should be in the .gitignore file. You need another way to manage them. If you use a few libraries, manage them manually. In Python, you can use the package management software pip. Together with the requirements.txt file, this makes managing libraries quite simple. In other programming languages like C++, this is a much harder task as you have to do this yourself somehow. Dealing with libraries is certainly one of the drawbacks of older programming languages like C++, while Python or Rust have a very good package management system.

There are some additional files in a project.

1. Custom scripts are created for the installation and build of the project. The process of obtaining the project, downloading third-party libraries, building the project, running tests, and executing the project should all be streamlined to require just one single command.
2. The README.md file is displayed on the front page of the Git project. It typically includes installation instructions and a brief project description. In fact, this book was also authored as a README.md file in a Git project.
3. .gitignore is related to Git. It lists all files and folders to be ignored by Git. For example, auto-generated files or files that are too big to be managed by Git.
4. Some formatting, code quality checking, or other miscellaneous files.

There are a few pitfalls in arranging the files and folders of your project. However, as long as you follow the general best practices, you should be fine. Consult the resources available on the internet for guidance on your programming language of choice.

42. Tools

"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'." - Linus Torvalds

// I think this chapter needs some reworking. Or should we remove it completely?

There is a fair amount of software that is supposed to help you write more or better software. Here is a list of the most important classes of tools I have worked with so far:

Version control software (VCS), Command line, Continuous Integration (CI), Integrated Development Editor (IDE), debugger, profiler, formatter, code quality checker, ticketing system, Wiki, package manager, build tools, docstring, container applications, container orchestration, databases, and many more.

For all these classes of software, there are several different vendors and open-source solutions.

Version control software

Git is certainly the very first program to mention in this chapter. Git is everywhere. It’s the Version Control Software (VCS) that Linus Torvalds programmed because all the alternatives were too slow for managing the Linux kernel or had other drawbacks like licensing issues [https://en.wikipedia.org/wiki/Git]. Git is clearly superior to most other version control software, and there is no reason to learn anything else. Git is a de facto industry standard. Only Mercurial is a viable alternative, but it is not as widely used as Git.

The original Git software is a console application, but there are also proprietary software products with a GUI.

I recommend learning the classic (command line) Git as soon as possible. Every programmer should be able to work with it. The only difference between companies is how they use Git. For example, there are different methods for dealing with branches when merging them into the master branch.

Git, everywhere git

Git should not only be used for bare code. Git can also be used on any text file that you have. The build files should certainly be version controlled. Additionally, other pure text files are worth controlling with Git. For example, if you conduct research and have files with measurement results, these could also be version controlled. The cost you incur is negligible compared to the benefits of controlling all your files with Git.

If you write a book like this one, it is written in Markdown and version controlled with Git. This makes it easy to collaborate with reviewers, and at the same time, I always have a safety net when I mess up some of my text.

I won't delve further into details about Git. There are plenty of tutorials on the web that can teach you how to use Git. They will guide you on making commits, using branches, and performing merges. Remember, as I mentioned in the chapter on unit tests, unit tests are invaluable when dealing with merge conflicts. They will promptly notify you if you have resolved them correctly.

Copilot

Copilot chat can convert human-readable commands into git commands. For example, it converts `git create a branch named "hello branch"` into `git branch hello_branch`. Just start the command with `git` and you will get a git command.

Command line

The most common command-line software is Bash (The Bourne-again shell, a.k.a. shell) on Unix systems. However, the Windows-based PowerShell is a viable alternative. For many purposes, Python or other scripting languages can be used as well.

The command line is the Swiss army knife of software development. It is the glue that connects all the different tools together, enabling us to automate all the build processes. For this reason, command line tools are generally preferred over GUI-based tools. GUI-based tools, such as the file browser, are excellent for starting smaller projects. However, they quickly reach limitations as they do not scale up well for larger projects.

The shell is an extremely powerful and versatile tool for executing other programs, running scripts, and handling various commands related to configuration settings, the filesystem, networking, and more. It is definitely worth learning some of the basic functionalities when you have the chance to automate a shell process.

Copilot

// figure out something else to ask Copilot. git questions have already been answered before.

// Copilot for CLI might change how we use the command line (and all its programs with it). Now you no longer have to use google to find the syntax, but you can use Copilot CLI instead. https://youtu.be/8_0DJ9FOlOM?t=787 https://youtu.be/pw0SH7AHIFI -> how does this work exactly?

```
git? How do I update the message of my last commit?
```

This returns the command `git commit --amend` along with a detailed explanation. The command can also be executed right away. Furthermore the `Revision` prompt enables you to request specific changes to the suggested content.

IDE

The Integrated Development Environment (IDE) is a type of software used for writing code. It's akin to Microsoft Word but tailored for programmers. There are numerous IDEs to choose from, some proprietary and others freely available. Personally, I never delved deep into IDEs. I simply use what my colleagues introduced me to: VS Code. I believe it's not necessary to invest a lot of time independently exploring the intricacies of a particular IDE. Therefore, I suggest you follow my approach and seek recommendations from your peers.

In all up-to-date IDEs, there are plugins available for most of the tools mentioned above. Consult with your work colleague to determine which ones you need. Dedicate a few hours to observing your colleague working in their preferred IDE to understand the purpose of the plugins. This time is not wasted; you will also gain insights into the code they write during this period.

It is worthwhile to learn some of the shortcuts in your IDE that enable you to modify code in different files more efficiently. This is beneficial as it enhances your workflow. However, do not overdo it initially, as you might end up wasting too much time. You can continue learning more shortcuts once you are familiar with your personal workflow. If you are keen on maximizing the use of shortcuts, you will need to learn the VIM text editor, which is operated solely by keyboard commands.

Continuous Integration

integration (CI) is the practice of merging all developers' working copies to a shared mainline several times a day.

This typically means checking in the latest changes of the code, compiling it if required, running all the tests, and building the final artifact.

There are several different suppliers of Continuous Integration (CI) software. I don’t know the exact differences, and you probably won’t need to either. If you work alone, you don’t need this, and in any reputable software company, this choice is made by others.

At the time of writing, Jenkins and GitLab are the most commonly used CI software.

Debugger

Probably everybody knows what debugging is. Because it is about the first thing you learn in programming: The code doesn't work, and I don't understand what it does. Let's walk through it and see what my variables do, for example, using print statements. However, there is a better way than using print statements: the Debugger.

Every programming language has its own debuggers, and IDEs usually support a debugger plugin for most major programming languages. It is useful to know some of the basic functionality of a debugger, such as setting breakpoints, navigating through the code, and examining the stack trace. However, in general, it is a sign of poor code quality if you have to rely on a debugger too frequently. Write small classes and functions where the purpose is clear, along with ample unit tests. By doing so, you should be able to identify the source of errors without relying heavily on a debugger. If necessary, feel free to use a debugger, especially when working with legacy code. Nonetheless, always prioritize maintaining high code quality and aim to minimize the reliance on a debugger. While a debugger is a valuable tool, its frequent necessity may indicate underlying issues in the code.

Profiler

A profiler allows you to measure the time needed to execute each part of the code. Depending on the type of programming you engage in, you may not need one frequently. It is advisable to use the profiler only when the program is running slower than expected, which may not occur frequently. Therefore, becoming familiar with the profiler is not essential at the outset of your programming journey.

Formatter

Pretty much all companies have a fixed ruleset on how code should be formatted. Some teams can debate for days about tiny details. If you start at a company, let someone set up the formatter for you. In most cases, this consists of copying some config file. And don't start endless discussions about the formatting details. Having the formatter set up properly will save you some pain afterward. If the formatter follows the wrong ruleset, you will have formatting changes in your merge requests. This is absolutely terrible because it hides the real changes. The formatter may change thousands of lines in a single MR, and you don't care about them. Real code changes are short, but you have to check them meticulously. Put both kinds of changes into a single MR, and you are done for. Judging such an MR becomes impossible.

If you work with old code that was formatted with an outdated ruleset, you have to run the formatter and create an MR before you start writing code. Or at least the formatting needs to be in a separate commit, though a separate, dedicated MR is preferred. If you change the formatting ruleset, run the formatter on all code and create a dedicated MR.

There are also some companies where every employee can use any formatting style they prefer. Once an employee creates an MR, the official formatter reviews the code as the initial step of the merge request process. This approach combines the benefits of both worlds: users have the freedom to choose their preferred formatter, while ensuring there is a default formatting that does not impact the merge requests. The only downside is the additional work for the DevOps developer who needs to implement this feature.

Personally, I don't care too much about the formatting style. If I have a choice, I use the default formatting with a tab width of 4 and a line length of 100. This is a reasonable compromise. Linus Torvalds (the guy from Git) has a very strict opinion on that topic. If you write code for the Linux kernel, you have to use a tab width of 8 and a line length of 80. Try writing code like that. You have to write extremely well to make all the code fit reasonably into this pattern. That's exactly why he's come up with this rule. The Google style guide also recommends a line length of 80 characters, though it is less strict.

Code Quality Checker

There are different programs available that can check your code for the most common quality issues. I don't know too much about them, but it's certainly worth a try. One example is the test coverage tool. Though this metric shouldn't be used as a business metric, use it to ensure that you have (almost) all code covered by your unit tests.

If you use C++ or any other compiled language, your most important quality checker tool is the compiler. Enable the "treat warnings as errors" setting for all warnings. You may find it annoying at first, but you will get used to it, and it will improve your code and prevent bugs. There is a reason why the compiler warns you about something; it does a better job of finding unassigned variables and other issues than you ever could. Why should you search for potential bugs yourself when the compiler can automatically do the job?

Pip, CMake

Some people may argue that Pip and CMake don't belong on this list. Of course, they are right. However, I still feel compelled to mention a sentence or two about them.

Pip is the Python package management software [https://pypi.org/project/pip/]. As a Python developer, it is essential to be familiar with pip. It is very user-friendly. By using the command line pip install numpy, you can easily install the numpy library. Done. A bit more challenging is managing the virtual environment (venv), which enables you to install all packages in an isolated environment . This is beneficial when working on various projects that need different versions of the same library.

CMake is the most commonly used build tool for C++. Meson is a more modern alternative that you can use for new projects. Make is outdated, and it is not recommended to use the Visual Studio build.

Ticketing system

Jira [should I add links for every software or for none??] is the most commonly used ticketing system and has little to do with code. It is very easy to use, and most of the work in Jira will be done by the manager writing the tickets.

This ticketing software is also helpful when managing a one-person project as it aids in organizing the work. It doesn't necessarily have to be a software project; you can also use it for all other kinds of projects. If you don't work for a company, there are also free alternatives available. However, I have never used any of them.

Wiki

Most companies use Confluence as their knowledge base, similar to Jira, which is an industry standard. Write general thoughts and high-level documentation here. However, keep in mind that information may become outdated quickly, so exercise caution.

You may also write high-level documentation of your code. However, avoid delving too deeply into details. Low-level details frequently change and become outdated, so it is preferable to refer to the code for accurate information.

Again, there are free alternatives available, although it is unlikely to encounter any alternatives in professional environments.

Docstring

The docstring software automatically creates documentation based on the comments in the code. It sounds like a good idea, although it should be used sparingly. There is minimal benefit in using docstrings for internal documentation since you can also refer to the code itself. Instead, docstrings should be utilized as documentation for external APIs that are accessed by your customers.

Every programming language has its own documentation tool. For Python, it's Sphinx, and for C++, it's Doxygen.

-# Part 9 Collaborating

43. Working in teams

[](97 things every programmer should know, thing 85), [](Software Engineering at Google, chapter 2)

"Humans are mostly a collection of intermittent bugs." - Brian Fitzpatrick

A good manager considers how things are done, but a great manager focuses on what needs to be done and entrusts the rest to his employees. He has faith in his team, which is far more motivating than someone who just yells orders.

The era of lone wolf programmers is over. Instead, you will spend most of your career working in teams. The narrative of the solitary prodigy programmer in the basement is a myth. Modern programming is a collaborative effort. Programmers not only collaborate with their peers but also engage with individuals from marketing, sales, and customers.

Collaborating with other programmers has its advantages and drawbacks simultaneously. Comparing programming in teams with working alone is akin to comparing a parliament with a dictator. A parliament may take more time to reach a conclusion, but the solution is typically superior to a decision made by a dictator.

At the same time, scaling up software projects only works with teams where all programmers are cooperating together. It is not possible for a dictator to work on his own project. He has to adapt and become part of the team.

Team Structure

In most projects, a team typically consists of around 4-10 software engineers, one project owner, and one project manager. The software engineers are responsible for the core work, putting in effort to write exceptional code, while the rest of the team also has their own tasks to complete, even if it may not be as visible to the developers.

The project owner (PO) is responsible for managing the tickets. They act as the interface between the project manager and the developers. The project manager (PM) oversees the entire project. They are in charge of communicating with customers to understand their wants and needs. These two aspects are not always identical. For instance, before its release, nobody desired a mobile phone without a keyboard, yet everyone eventually needed an iPhone. It is the PM's responsibility to discern such distinctions.

It is important that everybody in the team communicates with each other. Software engineers often discuss their code extensively. However, they frequently have questions about the ticket that the project manager (PM) needs to address. Conversely, the PM needs to be informed about the status of each feature to estimate the progress of the software.

The Bus Factor

The [bus factor] (https://en.wikipedia.org/wiki/Bus_factor) says how many team members would need to be at least hit by a bus before the project is doomed. The definition of this expression may sound a little absurd. And it is. But it has a point. Fortunately people don't get hit too often by a bus. But there are other risks to consider, such as team members falling ill or leaving their positions for various reasons. A low bus factor increases the vulnerability of the entire project.

Make sure the bus factor in your project is as high as possible. Ensure that there is a good amount of knowledge exchange between team members so that everyone has some understanding of all aspects of the project. This way, the entire project won't come to a halt just because one person is sick.

Developers work

The developers are the ones who do the real work. They are the ones who write all the code. For such hard work, it is important that they stick together. Only a tight pack of dedicated software engineers can accomplish the task. That's at least their perspective. Now, let's get a little bit more serious.

Software engineers have several tasks. The most obvious one is, of course, communicating with each other. This is why any modern software company provides a coffee machine with free coffee. Additionally, engineers need to discuss tickets, determine the fundamental structure of the code, and communicate during pair programming or code review sessions. While talking during code reviews is not mandatory, smaller MRs can be completed with written comments. However, when unsure, it is always preferable to engage in verbal communication. Particularly during times like the Corona pandemic, the personal connection has diminished, and situations can become challenging.

During pair programming, it is essential to communicate. Both participants engage in discussions to determine how the code should be structured. This facilitates crucial knowledge transfer, particularly when both participants are experts in different aspects of the code. Through this collaboration, both programmers gain insights from each other, leading to an enhancement in code quality. Consequently, the need for code review in the MR is eliminated. Although pair programming may require more time than working independently, the benefits often outweigh the additional time investment. It is important to remember that the most valuable resource in a company is not the code itself, but the knowledge. This knowledge is acquired through effective communication and collaboration.

Another fairly big job is going through Merge Requests (MRs), also referred to as Pull Requests. Everyone has to do it, and no one really likes it, but it has to be done. Nobody likes waiting a day for their code to be approved and merged. Therefore, reviewing MRs has to be done quickly. So, get up, open the browser, and select the first MR. Now, doing a code review is a tricky business. You can somehow sense that the code is not good, but it’s elusive. It's your job to pinpoint the issues without being too critical. Additionally, you may notice some duplicated code that needs to be refactored, among a dozen other things. It's time to reach out to the author of this MR. Some issues can't be resolved just by writing comments.

Communication

As mentioned above, teamwork is a key element in modern software engineering. Without good communication skills, teamwork is not possible. Therefore, it is important to learn how to effectively communicate with others, understand human flaws, and learn how to address them.

In a team, the most important language is not Java but English (or German in some of the projects I worked on). Use this language to communicate with other team members. If you find communicating with a computer challenging, reconsider. With a computer, you can simply search online for what to do, and most of the time it works. However, effective communication with other humans may require significantly more effort.

- Make sure you know what you are talking about know how communicate effectively with to the specific audience.
- Keep communicating. Ask questions. Let Let the other person talk. Once you stop receiving replies, there may be an issue.
- Make sure the other truly understood what you discussing.

There are probably hundreds of other rules, but the few mentioned here are the ones I know, even though I’m not that proficient at applying them. As a developer, you don’t have to be familiar with all these things. However, if you aspire to manage people, you need to develop a sense of how to communicate effectively with others. Consider exploring books or attending seminars on this topic, as this is not the appropriate place to delve into specifics.

Humans are all inherently flawed. They are insecure and tend to conceal their vulnerabilities. They dislike criticism and fear not being perceived as geniuses. However, being a genius is not a requirement. The majority of work is accomplished by competent, though not exceptional, programmers. This fear exacerbates the situation. Effective communication among developers is crucial. A team is significantly more productive when its members engage in constructive criticism and open dialogue.

It is okay to fail. Fail early, fail fast, fail often. Get feedback as early as possible and improve. This is the only way to make progress. Conversely, you always have something to teach. Discuss with your colleagues, and you can learn from each other. Don't be afraid; nobody is perfect.

Don't make statements like "this is bad". Critisism is has to be constructive; otherwise, it is pointless. Even worse, it can be seen as a direct attack. You need to be able to clarify why something is considered bad so that the original author has an opportunity to make improvements.

In order to excel, humans need psychological safety. This requires three things: humility, respect, and trust. Effective teamwork is not possible without these elements. Discussions can only be successful if all parties involved are treated equally.

Working with customers

[](97 Things Every Programmer Should Know, Thing 97)

Customers are only humans. Quite frequently, they don't say what they mean because they may not know it any better. Keep adapting your vocabulary to understand what specific words truly mean from the customer's perspective. Sometimes, this reveals a misinterpretation. For example, if customer and client have some completely different understanding. Do not assume that the discussion on requirements is complete after just one meeting with the customer. It is essential to stay in touch to receive continuous feedback to ensure that you implement what the customer desires, not just what they say.

Frequently, customers do not prioritize what is important. Or at least, things that are important to customers may not be important to the programmer. For instance, a software is only used if the GUI looks exactly the same as in the previous software. As long as the user does not have to learn anything new. Even if the old GUI was poorly designed, the customer refuses to adapt. You really have to come up with some significant improvements for your version to be accepted.

44. Code Review

"The computer was born to solve problems that did not exist before." — Bill Gates

[](software engineering at Google)

Code reviews are important for spreading knowledge and improving the quality of the code. This does not work without some criticism, so it requires a bit of intuition to know how to critique the code without insulting the author. Most importantly, you have to criticize the code rather than the author. But let's first take a look at how the whole code review process began.

A long time ago, in a kingdom far away, software developers began collaborating. They shared their code and worked on the same code simultaneously, which led to problems arising. They required software to manage the various versions of the code.

After some mediocre attempts to fix this issue, our savior emerged. Linus Torvalds, the hero of every fervent Linux developer, rescued us by creating Git. This resolved the issue of version control software definitively.

Unfortunately, Git was not yet the final solution. It was still possible to write poor quality code and merge it into the master branch. The only solution left was to dismiss this malicious developer.

But now comes the real solution: Merge Requests (MR) and code reviews. No user is able to make changes on the master branch all by themselves anymore. Before merging changes into the master branch, a user needs to create a public request and wait for someone else to accept it. Only then can the other developer allow the changes to be merged into the master branch.

Now, there are a few things to consider regarding code reviews. First of all, code reviews are great not only for keeping the code quality up to date but also for improving the programming skills of all developers. They are a great opportunity for knowledge exchange. Developers are obligated to look at each other's code and thereby learn a lot.

Drawbacks

However, there are some downsides as well. They can be severe enough that teams have even stopped using MRs altogether. Most importantly, everyone has to adhere to the rules. There is no way to prevent foul play by developers sabotaging the system in a way that renders the MRs useless or even counterproductive.

The first problem is people accepting merge requests without commenting, possibly without even reviewing the MR. This could be due to a lack of understanding, laziness, time constraints, or simply to do the MR author a favor. It might be more beneficial to avoid using MRs altogether.

The second issue is speed, which is crucial. It is of utmost importance to review MRs as quickly as possible. Prolonged idle times for MRs result in a significant decrease in developers' productivity. Moreover, it is highly frustrating to wait for an MR to be reviewed and not be able to continue working.

Another very serious problem is overly long MRs. It maybe impossible to judge the quality of a change with a hundred lines of code. It is advisable to keep the tickets, commits, and MRs small. Enormous MRs are a waste of time as they make it difficult for others to understand the changes being made. If a ticket becomes too lengthy, it is recommended to divide the code into several MRs and ensure that tickets are kept smaller in the future.

There is a widespread and fundamental misunderstanding regarding MRs. Don't expect the referee to find bugs; this is absolutely impossible. The referee doesn't have time to think through all these details. The author is responsible for writing error-free code along with good test coverage to prove that it is most certainly free of bugs. MRs are more about the general structure of the code and knowledge exchange. The referee can only check that there is a reasonable amount of test coverage.

Always be polite. Giving feedback in a merge request (MR) is akin to critiquing someone's code via email. It is a delicate task, so maintain professionalism by focusing your comments on the code itself, not the author. If discussions escalate to yelling in MRs, it may be time to consider leaving the job. The situation worsened during the COVID-19 pandemic when many developers had to work from home. It requires strong team spirit to handle written feedback in MRs effectively.

One thing I can highly recommend is looking at the code together, kind of a pair reviewing. In theory, the referee is supposed to understand the code all by himself, or at least that's my understanding of an MR. However, discussing the code with the author turns out to be a really good alternative. Especially for long or important MRs. Additionally, it keeps up the human touch. It is much harder to insult someone orally than written. This is a highly important feat.

Conclusions

In case you engage in pair programming, you may skip the code review phase altogether since two developers have already agreed that the code is acceptable. Pair programming also enables your team to share knowledge that would otherwise need to be addressed during the code review. This is one of the reasons why pair programming does not necessitate double the time. The code review process would consume a significant amount of time, which can be saved through pair programming.

For teams with little experience, I believe it is still important to create merge requests. In my opinion, the benefits outweigh the drawbacks. However, this is only effective if everyone adheres to the rules and provides prompt feedback.

With highly experienced programmers, on the other hand, one can skip the code reviews and engage in high-level discussions about the code instead. This approach is faster and typically just as effective. Highly experienced programmers only need to coordinate the high-level abstractions and do not have to review the low-level details. I hope your team reaches this stage quickly as it can significantly enhance productivity.

I generally recommend conducting code reviews. However, if code reviews become burdensome, which can easily happen, you have to reconsider your approach to work and perhaps explore alternative methods to disseminate information about your code. Just remember that sharing knowledge is crucial, albeit quite costly.

45. Agile

//// "All architectures become iterative because of unknown unknowns. Agile just recognizes this and does it sooner." - Mark Richards

Agile is the de facto industry standard when it comes to planning of software projects. But it has not always been this way. So how did we get there and what does Agile actually mean?

Problems of Waterfall

Until the the early 2000s, most software development teams were working according to the so-called waterfall scheme. For every project, there was an analysis, a design and an implementation phase. This sounds like a good thing to do, as other engineers work the same way. However, planning software top-down never really worked out as it was not possible to plan the complexity down to the last detail and changing requirements made things even worse. Brief, in many cases waterfall projects turned out to be a disaster.

The first problem of waterfall was missing feedback. The whole project was just one big pile of work and it was impossible to get a reasonable estimate on the time it takes to get all the work is done. Many projects failed spectacularly as at the deadline there was still a significant fraction of this pile left but no one informed the management beforehand.

The main issue however was, that people had the wrong mindset. They assumed one can plan software like building a house. One makes a plan in the beginning and gets a team of developers to execute it. This does not work out. It is simply not possible to plan a house down to the very last detail. The architect has to visit the construction site weekly, if not daily to fix problems that will show up. But that's not the only issue. Maybe even worse, since the team was working in waterfall mode, they were not in the right mind set to adapt to changing requirements or problems encountered during the implementation.

Agile was born

When planning a project, there are three simple truths [](Zühlke, www.zuehlke.com):

1. It is rarely possible to gather all the requirements at the beginning of a project
2. Users will change their minds
3. There will always be more to do than time and money will allow

These three truths are the reason why waterfall was never going to work. Instead a somewhat more adaptive approach was needed. A more ... agile one.

In 2001, a group of software engineers met for two days in the Rocky Mountains in order to improve the planning of software projects. The result was the Agile Manifesto [](Agile manifesto), [](Clean Agile), a brief guide line how software development should be done. Some of the points were:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

While the Agile Manifesto was about how a project should be run, there is also a Bill of Rights for the developers. The Bill of Rights states what kind of rights each individual in an agile process has.

- You have the right to know what is needed with clear declarations of priority.
- You have the right to produce high-quality work at all times.
- You have the right to ask for and receive help from peers, managers, and customers.
- You have the right to make and update your own estimates.

- You have the right to accept your responsibilities instead of having them assigned to you.

Work planning

// explain story based velocity rather than story point based planning?

The product owner has a set of requirements that the code should fulfill. This pile of work is broken down into small tickets. Where I'd like to emphasize the word small. Each ticket should be doable by one person during one sprint. Preferably it's smaller than that.

Every ticket is estimated for how much work it will take. The ticket size is quantified by the number of story points it gets. This is an artificial number to give the tickets a measurable size. Yet at the same time, the story points are vague enough to indicate that this value is only a vague estimation. In most projects, a story point is between one half and one day of work.

The ticket size is estimated at the so called sprint planning, a meeting where the next sprint is planned. For each ticket, the number of story points is estimated by the team. Usually every developer makes a hidden estimation and the average is the number of story points being assigned to the ticket. If there is a large discrepancy in the estimations, the team needs to discuss why this is the case. Probably some difficulty was missed, but it could also be that most developers underestimated the task. Unfortunately this estimation of tickets does not always work too well. It takes really good planning such that all developers know what has to be done in the ticket. Otherwise the estimations are way off. This is especially the case when the ticket is not well defined. In this case, the ticket probably has to be split up into smaller tickets.

For these reasons there is the attempt to work without any estimations of ticket size. Just cut the tickets as small as possible and count the number of tickets instead. This so called story based velocity has the advantage that you don't have to do the notoriously imprecise estimation of the ticket size. And breaking down the tickets into as small junks as possible is anyway recommended.

Tickets all have some business value. They have a direct effect on the user. This means, that every ticket is a vertical slice through the software stack. From the database through the back end code and to the GUI. Everything has to be worked on in a single ticket. So, either you know already how to work on each layer of the software stack, or you team up with someone else and do pair programming in order to fill the knowledge gap.

At the same time, one can write acceptance tests for every ticket. "... if the user clicks x, then the window closes." This test is also the acceptance criterion of the ticket. The ticket is accepted if the acceptance test passes.

// Write SMART Acceptance ciriteria [Zühlke, www.zuehlke.com] // move this to Requirements Engineering??

Specific: Use examples with values. Measurable: You have to be able to test it. Achievable: It should not depend on 3rd parties. Relevant: It should be important to the user. Time-bound: It should be done in a reasonable time frame.

Quality Assurance

In waterfall projects, the Quality Assurance (QA) was manually trying to find bugs in the existing software. This certainly does not fit anymore with agile. Instead, the QA should write the acceptance tests of every ticket. These tests should preferably be written before the developers finished working on the same ticket. This is quite similar to TDD and is called Behavior Driven Development (BDD) or at times Acceptance Test Driven Development (ATDD).

Finishing the acceptance tests before the developers finish the actual ticket is a hard task. One way to mitigate this issue is working ahead. The QA team can always try to be half a sprint ahead. This is not so easy as the sprint planning was not yet done. On the other hand, the PM should know quite well one sprint ahead what is going to follow next.

The Iron Cross

// make a graph of the iron cross // or remove this alltogether?

As in most other domains, there is frequent problem of projects not being done in time. There is of course always the solution of reducing quality far enough to make it in time.

In Software engineering, we have the rule of the Iron Cross: Good, fast, cheap, done. Choose three. Here are some options how the management can deal with these issues.

Good

Reducing the quality of the code is the first option. Albeit, it is probably the worst one. This will lead to bugs and the overall productivity of the team will plummet quite quickly. Quick and Dirty just doesn't work. Especially not on the long term.

Fast

Reducing scope of a project is usually the best option. In Agile, the important tickets were done at the beginning of the project. Furthermore, the work was cut vertically. Meaning that all the important stuff is already working. This allows the management to remove some of the less important tickets from the scope of the project.

Cheap

If a software project goes wrong it's usually not that cheap anymore, no matter what's being done.

One thing the management tends to do is throwing more developers at the problem. This, however, is not working out as planned. It takes time and effort to introduce the new developers into the project, leading to a short-term dip in the overall productivity, before ramping up.

Done

Changing the schedule helps a lot and is frequently the only option. As it was already the case in waterfall times. On the other hand, there are also plenty of projects where the scope of work is tuned in order to get the work done. It is quite amazing how often the core requirements of a project were overestimated. There are projects where some of the first requirements were in the end not implemented because it turned out that these requirements were not really needed.

Sprints

In Agile, the whole project is split up into pieces of one to four weeks, called sprints. During each sprint, there is a sprint planning, some time for implementing the features and a sprint presentation meeting in the end where the outcome of the sprint is being discussed. This structure results in regular feedback how the project is progressing. It allows the project manager to extrapolate the current progress and make rough a estimate on how long it will take until the next mile stone. This progress can also be used as a monitoring tool how well the development team is doing.

The first meeting of a spring is the sprint planning. It takes the whole team to discuss the tickets and which ones to scope into the sprint. The sprint planning for a two-week sprint may take a half a day.

Part of the meeting is the planning game (see Work Planning), where the story points for each ticket are estimated. This is required to plan the scope of the next sprint.

Next is the daily meeting. This meeting is not mandatory and it's very short. It is kind of replacing the coffee machine gossip. Everyone very briefly says what he's doing at the moment and if there are any blockers. There are no discussions in this meeting. Discussions are held afterwards.

Toward the end of the sprint, the software developers present their work done in the sprint presentation meeting. The idea of this meeting is for all the stake holders to get an idea what the status of the software is. And hopefully, the developers are proud to present their work done.

The last meeting is the retro perspective. Here the team meets to discuss anything that could improve the productivity of the development. Issues why the ticket size was estimated wrongly, blockers that were not resolved for too long, unresolved MRs, etc.

Becoming agile

What I tried to explain in this chapter so far was supposed to be something like a manual how to become Agile. The real effort, however, lies before you. There is no Agile a manual. It is more like a schema. And you can stretch this schema in many possible directions, whether it makes sense or not.

The most important point from Agile is that you should figure out by yourself what works best. And be honest with yourself. It may be more convenient to work alone for several weeks and hand in a pile of work in the end, than spending some time in meetings every two weeks. You don't know how your colleagues are doing. You lack knowledge how you are progressing. And not only you, also your project manager would like to know how things are going. This is a pretty important aspect of Agile: you gain a lot of information about the progress of the project that will help you to further plan the rest of the work.

Furthermore, there are some things that are absolutely mandatory, when working agile. You are not planning the whole software anymore at once in the beginning. Instead, you have to be able to adapt. Your code has to be flexible. What most people don't understand is that they would have to remain flexible also in waterfall mode as plaining everything from scratch isn't working out.

In order to be flexible, you have to be able to adapt your code. You have to change its structure. You have to refactor. This is a hard task and you're probably afraid that you may break something. But it's inevitable. You have to be able to change your code. That's your job. Instead, you have to mitigate your fear of breaking the code. And the only way to do so are automated tests. Loads of it. Pretty much every single line of your code should be covered by a test. This is the only way how Agile can ever work out.

46. Requirements Engineering

"Do the Right Thing and Do Things Right the First Time" - Marcia (Marci) Malzahn

Written by Felix Gähler

Requirements Engineering (RE) is the process of determining what you should implement. Because as we have learned, we may only implement code if it's really useful for our customers [chapter Software Engineering].

It may be surprising, why this chapter is even needed. Isn't it obvious what you have to implement? Unfortunately no. Many times it is even highly unclear what you have to implement. And you quickly wasted a few hundred thousand euros if the development team wastes a few months developing a feature which in the end isn't used. It is therefore very important to always be aware of what and why you are developing.

Stakeholders

As so often, you spend a large part of your work talking to other people. You have to find out who is important and who just feels important. You also have to consider the different characters in the team and act accordingly. Company politics. If you don't feel comfortable with this, you better stay in software development and don't switch to Requirements Engineering.

Stakeholders are all persons with an interest in the system. If I forget to ask a stakeholder, he will not be satisfied at the end. Therefore, I create a stakeholder list, in which I estimate how big the interest is, and how big the power. A stakeholder with great power must of course be kept happy, even if his interest is rather small. Stakeholders with great interest but little power I actively involve, because they are often the daily users. Stakeholders with great interest and great power I prioritize. And those with little interest and little power I can safely ignore, even if they may make the most noise. [Create graphic]

Goals, Context and Scope

First, you have to ask yourself what goals you want to achieve with the new system. It is important to define the goals as completely and consistently as possible. They should also be weighted according to their importance. It is worth investing some time in a precise goal description, which is coordinated with the stakeholders.

You always have to be aware of what exactly belongs to the new system and what does not (system boundary or scope). This also includes the question of what actually belongs to the system and what is predetermined from the outside. What is in the scope can also be changed in the project.

The scope and context can be represented in a use case diagram. This diagram shows which actors influence components of the system. // Insert example

In the scope and context there is initially a gray area. The smaller this area is, the smaller the risk of the project. It is therefore worthwhile to determine the context and especially the scope as precisely as possible and to update it regularly in the case of iterative approaches.

Goals, context and scope form the framework in which the requirements can now be determined.

Requirements Elicitation

Just as it is difficult to write good code, it is also difficult to describe a requirement in a ticket. It should not be too long, but still as clear as possible. It should also be well received by the developers. Not everything that seems logical to the author of a ticket is logical for the developer who will implement the ticket.

It's best to write a few examples in the ticket, which can also serve as a test case. These should include both the "happy case" and boundary conditions and special cases. The latter is often forgotten. For example, in a bank transfer it must be prevented that a negative amount is transferred.

The requirements should be formulated as neutrally as possible. They should describe the WHAT and not the HOW. The author of the feature should actually know nothing about the technical details of the code. In addition, developers often have better ideas on how to implement something than external persons.

It is crucial to involve the stakeholders in the requirements elicitation. This is hard work. It is time-consuming and labor-intensive. I try to find as much as possible and to "tease out" the stakeholders.

If you think it would be easy to find the requirements for a new system, you are wrong. The users, both outside and inside the company, are not waiting to discuss the advantages and disadvantages of the software with you for hours. They have work to do. Events such as company outings and aperitifs can help. Sometimes surveys and interviews are more helpful. The most important thing is that you are a good listener, so that people like to talk to you about their concerns and trust you.

Furthermore, you must also inform yourself independently of the users about the topic. It is not only important what the software can already do, but much more what the customers want. Often one thinks too narrowly. Just as Nokia did not want to develop a smartphone at the beginning of the 00s, although it was suggested by an engineer. Nokia simply saw no market for a phone without a keyboard.

The most important sources are:

- the stakeholders: clients, customers, users, managers, operators, developers, architects, testers
- documents: laws, norms, standards, concepts, specialist articles, error reports
- existing systems: old systems, predecessor systems, surrounding systems, competitor systems

When determining the requirements, there are 3 different factors:

- The basic factors are those that are often not even mentioned, as they are considered so self-evident. But they must not be forgotten.
- The performance factors are the focus of the users and are usually discussed.
- The enthusiasm factors are those that a user wishes for, although he does not know it. If you find these features, you have done a really good job.

The best way to find all the basic factors, performance factors and enthusiasm factors is to combine several survey techniques. Interviews for the performance factors and additional creativity techniques such as workshops or brainstorming for the enthusiasm factors. Field observation is well suited for the basic factors, which are considered self-evident by the interviewees and are therefore not mentioned at all.

Example hotel: That toilet paper must be available is not mentioned, as it is considered self-evident. But woe if it is missing when you are sitting on the toilet!

Documentation of the requirements

It is best to document the requirements continuously during the interviews, workshops, etc. The same or very similar requirements are of course mentioned again and again. It therefore makes sense to record the requirements in a uniform pattern so that duplicates can be easily recognized.

I document the requirements in natural language. This way, non-technical readers, suppliers, customers, managers can also be addressed. The stakeholders do not have to learn a tool first, but can work without training.

With the help of a glossary, I can define terms that are used in the requirements. This way, misunderstandings can be avoided. // example

I also document the requirements using formal models. These help to find gaps and contradictions and facilitate the later development of solution models. A model is like a map. A simplified image of reality that only shows the relevant things. The model shows this image from a certain perspective. It therefore makes sense to create several models that show reality from different perspectives.

Use-Case Model

With use case models, the business cases and their interactions can be modeled well. I therefore create a diagram that shows the system boundary and the actors that interact with the system.

In order to identify the use cases, I start with the actors. I think about what the system should do for the actor. Does it store data? Who triggers the action in the system?

Small systems typically have about 10 use cases, while large systems can have 50 or more. // Example

Class Diagrams

The business objects and required data can be well represented in class diagrams. //example

Prioritization of Requirements

You start with the most important tickets as you will never implement all of them. The requirements are therefore prioritized (in consultation with the stakeholders). It is not surprising that every stakeholder wants to see "his" requirement at the top! In order to objectify this conflict, there are prioritization models such as WSJF (Weighted Shortest Job First). This optimizes the economic benefit by comparing the costs of delay and implementation effort. The difficulty, of course, is to estimate these costs.

Review of Requirements

It is much easier and cheaper to fix a bug in the requirements phase than later in the testing phase or even in production. Therefore, it makes sense to review the requirements before they are implemented. Typical errors are incorrect, incomplete, contradictory, or unrealistic requirements.

It is very important to involve the right stakeholders in the review. Error search and error correction should be separated, the yield is then higher. In a rapidly changing environment, it makes sense to repeat the review periodically. Are we still up to date?

A simple and effectife review method is the review. A moderator invites stakeholders as reviewers, goes through the requirements with them and creates a list of findings with weighting. Based on the findings, the requirements are then to be corrected accordingly.

The reviewed and accepted requirements form the so-called baseline for development.

Administration of Requirements

In a small project, the requirements can easily be managed in a text document. Nevertheless, it is worthwhile to give each requirement a unique ID as a reference. The ID can also be used to document the code.

Requirements have other attributes in addition to ID and description. The most important are:

- Source of the requirement. This is important for queries.
- Priority of the requirements
- Status of the requirement. Is it implemented, tested, accepted?
- Verification tests. These are used to check whether the requirement is implemented correctly.

It is also important to manage changes systematically. If a requirement is changed during the project, I document exactly what was changed when. The changed requirement must again go through the review and be accepted. Then the question arises, where in the already implemented software the change has an impact.

In security relevant systems, traceability is a must. So the traceability from the acceptance test back to the source of the requirement.

Tools for Requirements Management

A common tool is Atlassian Jira. With this tool, requirements and test cases can be managed and assigned to developers for implementation. Also, detected errors (bugs, defects) can be recorded and prioritized in Jira.

47. Planning

// TODO: read through again. Is there duplication with the agile section?

"We take the most experienced engineer. He spends 2 days making various attempts to estimate the amount of work required. In the end we take the highest estimate and multiply it by two." – unknown

Planning major projects is extremely hard, not only in software engineering. Architects and civil engineers plan houses and streets all the time so they've become fairly good at it. But as soon as there is something much bigger they never did before, they start struggling. Frequently they are quite good but there are always cases where things go haywire. Not only at the Berlin airport.

With software development it is even worse. There are no small houses and streets that we can get some practice with. Unless you do very basic web or app development. Most software is simply way too complex and fairly unique. It's impossible to understand all the details. Even the fundamental logic of the problem is not always apparent. Somehow plans on writing software are always too optimistic and failed deadlines are standard.

This sounds very logical. We are all motivated and want to get things done. But our working speed is limited. It's slower than we want it to be. We need more time to understand problems and code, we have to change more code than intended and we also spend a lot of time with MRs and meetings. If your boss asks you when the software is going to be ready, try hard not to be too optimistic. It is very hard but making too optimistic guesses won't help anyone. You put yourself under pressure and ultimately you still miss the deadline.

Planning code in detail is a similar topic. But at least there is now a solution that seems to work in most cases.

For a very long time, software projects were developed using the waterfall approach. There is a team of developers who try to understand the topic and develop a model on the white board how the structure of the code should look like. Another team, or maybe even the same one, takes these ideas and implements them. Do I have to tell you how this ended? Let me give you some hints. People tend to underestimate complexity; people miss features and furthermore there are changing requirements. The result is a team of software engineers trying hard to implement what they were supposed to. At the same time, it doesn’t work as the planning team missed important details and over the time new requirements showed up. I heard of cases where this approach worked. A few. As well as a lot of disaster. Software projects are simply too complex as if the waterfall approach would work.

Now let’s go back to the civil engineer and his houses. It makes sense that the civil engineer plans and the construction workers build the house. That’s what they do. That’s their job. Over the time a construction worker gets an idea how the structure of a house has to look like. But still, he is never going to plan one. He wouldn’t know how. Vice versa, the civil engineer could maybe build a house, but it’s financially not interesting. It makes sense to have two different groups of workers taking care of planning and construction. And even here the civil engineer has to check the progress of the construction frequently and improvise in case of unexpected events.

In software engineering the planning and the development team both have the same education. The planning team might have a little more experience than the development team, but that’s negligible. Then why do you separate the two tasks? This creates only overhead and frustration. If the planning team is smart enough to plan the whole software on their head, they should also have enough experience to write the whole thing down in code. There's barely and overhead between planning the software and writing it down. When writing the code down, they will be able to see if everything really works out as plannned. Ultimately the planning team can make the whole job on their own.

Planning code

// move elsewhere? Rename section? Most of it is about UML diagramms.

A widely used tool to display interactions between classes are UML diagrams. To put it up front, I don’t like UML. It's generally a waste of time. You can also just briefly write the empty classes and connect them in code. It’s the same, just in a different representation. UML is the worse one. It would be easy to create the UML programming language. But no one has done it. Because graphical programming is terrible. It is harder to understand than code. Small UML diagrams are OK, but one can quickly get lost if they get bigger. Ask scientists about their experience with Labview. I prefer writing the code framework right away and save the effort for creating UML diagrams. However, feel free to try UML diagrams. If they are a great help for you or your team it doesn’t matter what I think.

One also has to consider the limitations of UML diagrams. The only represent classes and their relationships. This covers only a tiny fraction of a program. Quite frequently you have to understand the logic behind a problem where UMLs won’t help you. You need something different. Try out whatever you feel like. Some different sketch, a plot, a coffee break, a walk in the forest, ... As long as it helps you understanding the problem it does the job.

I also had such a moment during my master thesis when I was calculating the expected value of the experiment but I was stuck for a long time. One late afternoon a PhD student came by and talked for a little. He just casually mentioned every step of my calculation and within a minute I found my mistake. Talking to other people is usually the best way to solve a problem.

48. DevOps

// move this chapter further to the back? It contains some testing, but it's more a high level overview of the software development process.

Development and Operations, short DevOps, is the combination of Continuous Integration (CI) and Continuous Delivery (CD). In short, it is automating everything from the build, tests to the release. But in order to understand more precisely behind DevOps, we have to take a look at how software development teams used to work in the early 2000s. What kind of problems they had that DevOps promisses to solve.

The early 2000s

Working with code in the early 2000s was tedious. Not only were Integrated Development Environments (IDEs) lacking a lot of functionality that we take for gruanted nowadays, also building a project was usually a tedious task. Many projects were lacking a one-click-build and instead the developers had to go through a series of steps in order to build the executable. Then they used SVN as a version control tool because git didn't exist back then. They could just merge their code on to trunk (something similar to the main branch in git), possibly even without a merge request. And no one knew whether the code was really working. Code could go into production without anyone ever checking that it was working out! You could have even merged a commit that broke the build!

Most teams were not writing tests for their code. It just wasn't fashion back then. Only with the advent of Extreme Programming (XP), [Extreme Programming Explained: Embrace Change, K. Beck, 1999] and with the Agile Manifesto in 2001 [http://agilemanifesto.org/], testing started taking up. This was certainly a mile stone for the software development, but it added another problem to it. So far you had to do a build, which was already quite tedious. Now you also had to build and run the tests. It sounds easy, but once you consider that you don't just have one kind of test, but several different ones, it becomes apparent that doing all the builds by hand wouldn't scale anymore. You have unit tests, integration tests, functional tests, performance tests, etc. The only way to keep up with all this new work is automating it. Continuous Integration was born. Not only the build and the formatting of the code was automated. Everything was automated. Code could only be merged if all the invariants were met:

- The code is formated according to specification
- The static code analysis passes
- The build passes
- Building all the tests works
- All the tests pass

Even though there are companies that don't care about the formatting anymore. They just let the formatter run for every merge request on the server. Locally the developers can use whatever format they like. This takes some effort to set up the CI/CD pipeline but it saves work for the developers.

Then there is also the task of creating an executable. It used to take many manual steps as well. Which was again slow and error prone. Now this part of the Continuous Delivery.

We have the development (Dev) and the IT operations (Ops) bundled all together, these steps form DevOps. DevOps is automating everything that has to do with building, testing and releasing of the new software.

Getting a project

Furthermore getting started to work with an existing project was frequently a pain. There were so many things that could have gone wrong. "Where do I get the source code from?", "What libraries do I have to install?", "Why does the build not work?", "Ah, I have to use that specific version of this library?"

It was a pain. And in many companies it still is. There is a simple rule about getting started: It has to work with one command. Getting the repository has to be one command, setting up all the libraries has to be one command, building it has to be one command and running the executable has to be one command as well. If it's any more than one command per step, you have to write a script that does the work for you.

Benefits of DevOps

[https://www.atlassian.com/devops]

- Speed: Teams that use DevOps have a significantly faster development cycle. Building, testing and realeasing software becomes much faster.
- Collaboration: DevOps improves collaboration between team members. For example due to merge requests. This makes teams more efficient.
- Rapid deployment: DevOps allows for rapid deployment. This has become more and more important in the last years.
- Reliability: DevOps improves the reliability of the software. It is easier to find bugs and fix them. Also the software is more stable as it is tested more thoroughly.

49. Mental health

I didn't really think about this topic until I watched just another random youtube video about this topic[https://youtu.be/aK_Jq00Hd8E]. It's not exactly the topic I wanted to write about in this book to begin with, but as I look at the other chapters around here, it probably makes sense to write about it as well. Because mental health is a huge problem in software engineering. Trust me, I've been there as well. Of course there are also physical problems because we sit too much, but probably more prevalent are mental health issues.

First of all, we have to agree to the fact that we are not machines. We are humans. Our brain is just one of our organs and it can be damaged. And the most common cause of brain damage is excessive amounts of adrenalin and cortisone, two stress related hormons. These hormons are great as they allowed us to suppress pain and gain powers to fight off wild animals. But when exposed to them for a long time, they seriously damage our bodies and brains. And this frequently happens in software engineering. We are constantly under pressure to deliver and do not have sufficient time to calm down again. On the long term, this is a serious issue as it causes burn-out and depression.

One thing you certainly have to be aware of are your working hours. Working more does not make you more productive. You might work overtime before an important deadline and your adrenalin boost may help you with it. But this is no sustainable working model. You'll need some time to calm down again. Working less might in fact make you more productive. I usually work only 80% (= 33 hours a week in Switzerland) because of this reason. Of course I'm in this lucky position that I can afford to work less. Though some companies like Microsoft already experimented with a 4-day workweek as well, [https://4dayweek.io/case-study/microsoft]

Furthermore it is important that you don't respond to emails and phone calls in your free time [https://youtu.be/C4GOekfDrOQ].

There are several signs that you are at the brink of a burn-out and you should take them seriously. The easiest issue to spot are sleeping problems. This can be caused by too much adrenalin and makes you feel awake all the time. However your body and brain need some rest again to recover. In case you have serious sleeping problems you should definitely visit a doctor and take a step down at work.

Another reason is bad mood and mobbing at the work place. This should be adressed by your boss right away. And if he doesn't fix it it's time you look for another job. You probably can't fix this on your own and you're only risking your mental health by staying any longer. Even if you like your job, it's not worth it. And chances are that you'll find another job that you like as well. I recommend talking to some of your friends about your problems. Probably they can pinpoint some of your problems and help you solve them. And yeah, I've also been there. I also quit a job before because the mood in the team was so bad and my boss wasn't going to do anything about it.

On the other hand, there are also plenty of things that can make you feel better. Most notably if you have a good mood in your team. This is something that cannot be overstated. People who like working with their coworkers are less likely to suffer from mental problems and won't quit their job easily. It is said that already the old romans figured out that a good moral and motivation is important for their legions and a sense of humor was one of the criteria to be accepted into their legions [?].

Furthermore a rewarding work is also very important to keep your spirits high. For example if you frequently finish your tickets in time and you are praised for it by your boss. On the other side it is very depressing if your tickets are too big to be finished and you are constantly behind your schedule. This is a common issue in Agile development [section ?]. Your work is only rewarding if your team is realistic about how fast they can work.

50. Hiring and getting hired

[The Software Craftsman (by Sandro Mancuso)], [Cracking the Coding interview]

That’s the moment you’ve all been looking for your whole life. Your first real job. The first position as a software engineer. But how do you get there? What is the process behind getting hired? Or rather, what should the process behind getting hired look like?

Hiring

Let's say it frankly. Unfortunately, quite some job application processes suck. There’s no other way to put it. And the problem behind it is very simple. The application process is being led by a manager who likes numbers. He thinks that 5 years of professional Java development is a reasonable qualification. Even though there are plenty of developers with more than 10 years of experience who don’t manage to write reasonable code. They just never made the effort to learn anything by themselves. They keep writing the same old crappy code they did 10 years ago. Meanwhile someone working for 3 different companies for 1 year each probably has improved his programming skill significantly in the meantime.

Instead of the bulleted point lists of requirements, a company should rather describe in whole sentences what they are doing and who they are looking for.

Similarly for the interviews. It's about getting to know each other personally. This is a very hard task, but there's no way around it. This is why many companies hire psychologists to support the HR processes. So, ask personal questions. What did you do at your previous job? What were the challenges? How did you get along with the previous work colleagues? There are hundreds such questions and to none of them you will find an answer on the CV. Make sure you don't waste your time asking the standard Java questions. How can I create a memory leak? Etc. And if you do, make sure the Java version used for the questions is at least up to date.

Instead do some pair programming during the interview. Let the applicant bring his own laptop and give him internet access. He should be working on his laptop the way he's used to. It's not about testing his knowledge on the latest IDE or testing framework. It's about finding out whether he's smart and sharing the same coding values as you do. About having fruitful discussions on the code you are just writing. It's about simulating some real pair programming, as you will do together if the applicant gets the job.

Search for applicants with that something extra. Developers who are working on some open source project in their free time. There's hardly any better sign that someone is a very motivated and possibly also a skilled programmer. Join one of these software development groups, possibly sponsor an event. This is a great opportunity to get to know other software developers and hire them without the tedious application process.

Keep recruiting all the time. This is a difficult task as the number of proficient programmers is too small to cover all the open positions. Thus, you can't be too picky about when you are hiring your new team mate. If you have to hire someone under pressure, you'll end up hiring someone who is not quite up to the task.

Getting hired

Getting hired does not take quite as much know-how as hiring someone. For the simple reason that you are getting invited and mostly follow the hiring process. Yet at the same time you should always stay aware that you are an equal partner during the application procedure. If you don't agree with something you may very well just leave the recruiting process.

As already written above, it's about getting to know each other. Thus, you may also ask questions. In fact, you are expected to ask questions. If you don't know what else to ask, ask the developer what he's exactly working at and what kind of problems they are facing. This is something to get started with.

You shouldn't take the application process too serious. Just stay yourself. They ask for 3 years of experience? Well, that's what they wish for. But in reality, 2 years are usually enough if your application is otherwise convincing. Or if you're living in an area with few programmers around, which is basically all around the globe.

Make yourself seen with your application. Mention all kind of open source projects, blog posts and conferences you attended. This also makes a good start for the interview.

51. Examples

// remove? Write a separate book with examples?

So far, there was fairly little code in this book. Now I'd like to make one example, just to show you an application of some of the things we learned. Once again, I want to have a simple real world project. Assume we have a robot and we are going to give it some instructions. It's a smart robot that understands a lot of things, but the general planning we have to do ourselves.

Apple pie

User story

Your father comes for dinner next Sunday and you want to make him happy. Creamy apple pie makes him happy, for example.

Acceptance criteria: your father is happy

Acceptance test

Now let’s first write the acceptance test. If we invite our dad, he has to say that he's happy. We assume that he prints out his feelings on the console. Thus, we can just check the console output.

```
# inside acceptance_tests/test_dinner.py
import subprocess

def test_dinner_makes_dad_happy():
    p = serve_dinner_as_subprocess()
    assert(contains_happy(p.stdout))

def serve_dinner_as_subprocess():
    return subprocess.Popen(['Python', 'dinner.py'],
                            stdout=subprocess.PIPE,
                            stderr=subprocess.STDOUT,
                            )

def contains_happy(lines):
    for line in lines:
        if "happy" in str(line):
            return True
    return False
```

That’s it. We won’t have to touch the acceptance test anymore until the ticket is done. Note that I used the function contains to make the test a little more flexible. Your dad might say other things as well that we don’t care about.

The very first thing we do is running the test.

```
pytest acceptance_tests
```

And we see that it fails. That was to be expected, we didn’t implement anything so far. But it was still worth the few milliseconds we spent here. There are better places where you can save time.

Implementation

Let’s start with the implementation. It’s a fairly artificial example, so I can just make some assumptions. In the main function we create the apple pie and have our dad eat it. The big part of the work will be implementing the dad and the function to create the apple pie.

Also note that the easiest solution would probably be buying an apple pie from the next bakery. This would be a perfectly viable solution to this task here. But let's assume that we have to bake the pie ourselves.

Or course we have to make several assumptions on how the apple pie is to be implemented. Let's start with the high level code.

```
apple_pie = create("apple_pie")
dad = Dad()
dad.eat(apple_pie)
```

Next we implement dad and then the create function.

```
from enum import Enum

class Flavor(Enum):
    VERY_CREAMY = 1
    SALTY = 2

class Dad():
    def eat(food):
        if food.name == "apple_pie" and food.flavor == Falvor.VERY_CREAMY
            print("I’m so happy")
```

Note that the `Flavor` is neither inside the `Dad`, nor inside the `ApplePie` class, as we have learned in the chapter on the solid principles (ISP).

```
class ApplePie():
    def __init__(self):
        self.flavor = Flavor.VERY_CREAMY
        self.name = "apple_pie"

def create(food_name):
    food_dict = {"apple_pie" : ApplePie()}
    return food_dict[food_name]
```

// etc. Either finish the example or remove it.

Paint

// DDD p.259

Idea: We want to define paint of certain color that we can mix with each other and change its color accordingly. I would like to make some comments to the implementation in the book mentioned above. The code starts with a simple class paint and its variables.

```
class Paint:
    V: float
    R: int
    Y: int
    B: int
```

These member variables don't have expressive names at all. They are renamed to

```
class Paint:
    Volume: float
    Red: int
    Yellow: int
    Blue: int
```

This can be further improved. The red, yellow and blue values all represent a color. They are all the same, while the volume has a clearly different meaning. Thus we can refactor the RYB colors into a dedicated object to fulfill the single responsibility principle.

```
class Paint:
    volume
    color
class Color:
    Red
    Yellow
    Blue
```

So far so good. We made some smaller refactoring and the basic data structure looks good to go. Now comes the very tricky question: how should the syntax of mixing two colors look like?

```
# paint a, b, c
c = add(a,b)
c.add(a)
```

The first is the procedural #? Way, the second is the object-oriented approach. Besides this fundamental question, we also have to figure out what kind of values a and b should have after this operation. Additionally, we also might want to find another name than add.

First, I would like to answer the conceptual question. What happens with a and b? This is a somewhat philosophical question and without knowing the actual problem we'd like to solve there is no clear answer. We can only reason about it.

```
def add(paint1, paint2):
    Paint paint3
```

```
    volume = paint1.volume + paint2.volume
    Paint3.volume = volume
    paint3.color.red = (Paint1.color.red* paint1.volume + Paint2.color.red*
paint2.volume) / volume
    paint3.color.yellow = (Paint1.color. yellow * paint1.volume + Paint2.color.
yellow * paint2.volume) /volume
    paint3.color.blue = (Paint1.color. blue * paint1.volume + Paint2.color. blue *
paint2.volume) / volume
    return paint3
```

Now I see 3 different possibilities:

1. We leave paint1 and paint2 as is. We used a copy of the actual paints and didn't change the original paints.
2. We set the volume of both paints to 0. This is the equivalent of mixing the two paints and being left with two empty canisters containing no paint at all.
3. We set both paints to None. This would be somehow equivalent to throwing the canisters away.

As I said, all of them are perfectly reasonable choices. It is up to us to choose one of them, depending on what seems to be the most appropriate choice for each case. This also has consequences how we should call the add function and what the best way of implementing is.

For the first option, we don't change neither paint1 nor paint2. Here it makes sense to call the function add or even define the + operator if possible, in your programming language of choice. This is a legitimate choice as we don't expect add to change any of its function arguments.

Let's assume that we choose option 2 and we're left with 2 empty canisters of paint. Calling this function add is no longer an option. Instead we could call it mix or mix_in. Additionally, we have to deal with the question if we want to be more or less object oriented. We do have the following options:

```
paint3 = mix(paint1, paint2)
paint1.mix_in(paint2)
```

Now this is a matter of choice. A whole generation of programmers grew up hearing that the later option is the better one. It would be more natural. But honestly, I don't see why this is supposed to be so super natural. As already seen, for case number 1 I clearly prefer the non-OO solution, simply because we are used to the add function not changing any function arguments while with the mix function, we have to take a look at the definition in order to be sure. Even here, I still opt for the first option. It just feels more natural to me as the function is symmetric, while the OO solution is asymmetric for no apparent reason.

The solution

```
paint3 = mix(paint1, paint2)
```

Has one drawback. It creates a new object and it changes both function arguments. Now this is a very unfortunate solution. Changing one function argument is already bad enough and changing two is even worse. Now one solution would be passing a list of paints, `paint3 = mix([paint1, paint2])`

Reasonable programming dictates that all list elements are treated equally and thus they are either all altered together or none at all. Furthermore, we can implement a mix function for any number of paints.

Still, in the end I'm preferring option 1 (not changing paint1 and paint2) and implementing a simple add function. This follows the general conventions and minimizes confusion. It follows the Single Responsibility Principle as it only adds the two paints and doesn't alter anything else. Changing the volume of the function arguments can be done in a separate step, if needed.

And sorry folks, my preferred solution is not object-oriented, other than defining the pure data classes.

-# Part 10: Final remarks

52. About Copilot

The examples on Copilot shown throughout the code were all very short. This was done deliberately. Not only for the sake of keeping the problems easy to understand, but also in order to keep the suggestions from Copilot under control. Just as for a human developer, Copilot works best for incremental changes. It is not able to read your mind (even though sometimes it feels like it) and for complex changes it won't be able to make a correct suggestion. If there is a more difficult problem, Copilot frequently makes some undesired suggestions. The solution is to break down the problem into some smaller parts and maybe guide Copilot by writing the beginning of the code, i.e. the definition of a function.

Here is an example with a list of books. In the function `parse_line`, Copilot suggested `author, title = book.spilt(',')` which is wrong. It correctly anticipated that this list contains authors and titles of book. But interestingly enough, it didn't understand that the first and last names of the authors were split by a comma. Only after typing `last_name`, Copilot understood how the line should be parsed.

```python
books =["Rowling,J.K.,Harry Potter and the Philosopher's Stone",
        "Tolkien,JRR.,The Lord of the Rings",
        "Tolkien,JRR.,The Hobbit",
        "Martin,George R.R.,A Game of Thrones",
        "Martin,Robert C.,Clean Code",]

def parse_line(book):
    last_name, first_name, title = book.split(',')
    return f"{first_name} {last_name}: {title}"

print(parse_line(books[0]))
```

Copilot was also a help when writing this book, though for writing text I like it way less than for coding. A lot of suggestions were "simply wrong. But it was still a help to get some inspiration." (The quoted text was suggested by Copilot, the rest of the suggestion was not useful at all.) When writing text it becomes even more obvious that Copilot does not understand some things. For example it claimed to have suggested a quote I would use at the beginning of the book, but that was plain wrong. I didn't use the quote at the beginning of the book nor was it Copilot who came up with the quote.

Furthermore Copilot makes a lot of boilerplate suggestions. For exmaple it frequently suggests text like "This makes it easier to understand." which might be right, but it does not help the reader of this book.

Copilot and this book

I used Copilot to write parts of this book. At times it was a help to get some inspiration. For example I would have used the word help instead of inspiration here (Copilot seems to like the word "inspiration" when writing about itself). On the other hand, Copilot also suggested a lot of nonsense. So it's not like Copilot would make the job of an author obsolete any time soon. But it's a great tool to improve your writing.

Issues

Copilot is just some artificial neural network. It is not perfect, it is no compiler and it doesn't even know correct syntax. It's "only" really good with guessing code. Sometimes Copilot produces quite some nonsense. For example, Copilot seems to have analyzed my text and found a lot of "```py" text snippets. Therefore it suggested to write this "```py" quite frequently at the end of a sentence. And just now Copilot is telling me that it learned not to do this anymore. But this turned out to be wrong.

Some of the errors of Copilot & co. are very subtle. Chat GPT came up with some fake suggestion containin wrong citations when it was used in a legal case [https://www.legaldive.com/news/chatgpt-fake-legal-cases-generative-ai-hallucinations/651557/].

Also when I was playing around, the code produced by Copilot was sometimes wrong. Here is an example where I asked Copilot to refactor some fairly trivial code.

```python
def print_content():
    print("********************")
    print("hello")

# command to Copilot: move the print statements into a dedicated function
if __name__ == "__main__":
    print("author: Marco Gähler")
    print("********************")
    print_content()

# new code:
def print_header():
    print("********************")
    print("author: Marco Gähler")
    print("********************")

def print_content():
    print("hello")

if __name__ == "__main__":
    print_header()
    print_content()
```

Here Copilot changed the order of the first two `print` statements. Of course the order suggested by Copilot makes perfectly sense, but it is still wrong as it changed the output.

And as I'm writing these lines, I also get the feeling that Copilot has some narcistic traits. It wants me to write here that it is a great tool.

Copilot and the future

Copilot is indeed a great tool. It was a great help writing this book and the code examples. But it is far from perfect. I doubt that it will completely replace software engineers and authors any time soon. Though it may change the way we work. It is a great source of "inspiration" (again: suggestion by Copilot) when you don't know what to write. And at times its suggestions are just hillarious. What it can't do is reading your mind. You first have to give it some input. And even then it is sometimes hard to tell it what you want to do. This is why you are still better off reading this book and understanding the patterns explained here. Copilot is not a replacement for your brain.

53. Further reading

I learned quite some things reading books and watching youtube videos, even though not as much as I did when thinking about and discussing code at work. The selection of books may be somewhat biased by the algorithms used by Amazon and YouTube. There are probably plenty of other good books and videos out there, I just didn't know about them. Here are the books that I read so far:

The Pragmatic Programmer 2nd edition (Thomas, Hunt) This book is one of the inspirations to write my book here. It contains a lot of general advice on software development, tough ultimately only quite little of their recommendations made it into this book.

Clean code (Robert C Martin) The best seller. Uncle Bob explains how good code should look like. I followed many of his rules, quite some of them are in a similar way in this book.

Clean architecture (Robert C Martin) ?

Clean Agile (Robert C Martin) It’s a fairly brief explanation how agile software development is supposed to work.

The Art or Readable Code (Boswell, Foucher)

Software Engineering at Google (Winters et al.) They write extensively about testing at google. What else?

97 things every programmer should know (Kevlin Henney et al.)

Cracking the Coding Interview (Laakmann McDowell)

Design patterns (Gamma et. al) Probably one of the most influential software engineering books ever. It explains how classes can be combined to create some whole new functionality. Alternatively, you can also watch some youtube videos about the topic.

Domain-Dirven Design (Eric Evans) Certainly worth a read. But a tough one. Trying to understand what Evans wanted to explain made me understand a big deal and I learned a lot about the fundamentals of programming. Even if some parts of the book are clearly outdated.

Effective C++ (Scott Meyers) If you want to work with C++ this book is certainly worth a read. You learn quite something about the background and how to use the language. However, some parts are outdated and it's not a book for beginners.

Effective modern C++ (Scott Meyers) Scott explains the ideas behind C++11 and 14. This is at the time of writing probably the more useful book. But only for advanced C++ programmers.

Working with legacy code (Michael Feathers) This book is about working with code that doesn't have any tests and probably needs some refactoring.

Refactoring 2nd edition (Martin Fowler) Simply a great book on refactoring. The introductory example is simply amazing. Martin takes an innocent looking function and applies some of his refactoring steps. In the end there is some code that is super smooth. It has barely any indentations!

Software Engineering at google (Winters et al.)

BBV Cheat Sheet by Urs Enzler, https://en.bbv.ch/publikationen-category/cheat-sheet-en/

Google Style Guide, https://google.github.io/styleguide/

And several youtube channels: @alexhyettdev, @ArjanCodes, @ThePrimeTimeagen, @CodeOpinion, @derekbanas, @TechWithTim, @ContinuousDelivery,

54. Outlook

"Programming is learned by writing programs." — Brian Kernighan

Maybe you were surprised sometimes that there were so few code examples. But I hope you understood that they are not required. I wanted to explain fundamental concepts of software engineering. I wanted to give you an overview of the most important things to look out for. This should be a book that tells you the very basic rules that make your code better. There are not so many. But they are really important.

You might have realized that in software engineering for every problem there are a million of possible solutions. Even when writing these very simple examples in this book I have to reconsider how to do it best. For you it must be even worse. I remember how I was lost when I started programming. This book wants to help you. It explains a lot of things you shouldn't do or use. It's restricting you. I don't want you to get lost because of all the posibilities you have.

Soon comes the next big step. The real world. Writing code. Finally, you are there. And I have to let you go. I could write another book with code examples and explain why some code is better than the other. But there are plenty such books and I doubt I know better examples than the other authors do. Still, I hope that you'll get back to this book once in a while if you seek fundamental advice.

Your next step will be to apply all the things you learned on your journey so far. Write code. As much as you can. And always try to improve it. How can you make it easier to understand? How should you break that class into pieces? How is the test coverage doing? There are so many things to look out for. There are so many obstacles along the way. Find a good programmer to help you overcome them. Or even better, do an internship. (But make sure the company writes tests before you accept the job.) Talking with other programmers is important to understand how you can change your code to make it better.

I hope you learned a lot of things that will help you in your life as a software engineer. Good luck!

Marco

55. Frequently used Abbreviations

API Application Programmable Interface BDD Behavior Driven Development CD Continuous Delivery CI Continuous Integration DAMP Descriptive And Meaningful Phrases DB Database DI Dependency injection GUI Graphical User Interface MR Merge Request OO Object-Oriented QA Quality Assurance TDD Test-Driven Development YAGNI You Aren’t Going Need It