Software Engineering made easy

How to write better code

1. Introduction to Software Engineering

This book is currently undergoing revisions. Feedback is highly valued. Either through a merge request (access rights need to be granted) or by commenting the PDF sent to marco.gaehler@gmx.ch. It is advisable to submit many small merge requests rather than a single large merge request. Feedback may include precise recommendations for enhancement or general reflections. Given the limited progress of the book thus far, I welcome broader feedback. Tipos are getting fixed using some AI tool, so don't worry about them.

Pearson Germany has expressed interest in publishing this book; however, it would be in the German language. I am hopeful that it will be published in English as well.

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

- If you have some smaller suggestions for improvement, wirte them inside a `//...//` comment. This is easier to find than a few characters changed in a long line in git. For bigger changes, create a new issue or write me an email.
- If anyone is an expert on Copilot and has ideas how to integrate it into this book, let me know. The responsible person from the publisher would like to have some tips on Copilot, but I'm not an expert on this topic.
- What is architecture? Or leave this chapter away all together? Read book "Fundamentals of Software Architecture"? (chapter Architecture)
- What is mocking and stubs? (chapter Testing)
- Domain driven design -> what is the idea behind entities, aggregates and value objects? (chapter Domain driven design)
- Which design patterns are important? (chapter Design patterns)
- Do I need the chapter logging?
- Restructure and sort the chapters somehow. Similar levels of abstractions should be close to each other.
- Some more examples on refactoring (what technique exactly? Converting a function into a class?)
- Design patterns -> write short explanations to all of them? Or select some of them? Or leave this chapter away completely.

- How to work with existing code? If the code is not as nicely written as explained here. Reread WELC again.
- How to organize software projects -> Felix
- And some more chapters towards the end of the book need to be improved.

2. Table of content

// figure out how to sort the different chapters and sections

3. One sentence summary

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

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

I, Marco Gähler, 35 years old at the time of writing, studied physics at ETH Zurich, Switzerland. I worked as a teacher for a few years before deciding 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 the Quantum Controller Software.

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 that explains everything I learned about good programming practices and everything else during the few years I spent in industry. Such that every person with a little bit of knowledge of a programming language can boost their programming skills 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.

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 mean, my English is fairly poor, and I was never really good at writing essays in school. And I only decided what chapters to write as I was reading other books. But the feedback I received was very good. People praised this book for being well-structured, well-written, and easy to understand. I hope you will agree with this assesment. Even Pearson Germany was interested in publishing it. This motivated me to keep writing and getting it published.

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
- Felix Gähler
- Claudia Gähler
- Linus Gasser
- ... you?

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.

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 what he was talking about. 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 these 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. It provides an overview of the most important topics so that you do not get overwhelmed by the infinite number of decisions a programmer 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 small, I frequently had to simplify them.

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 absolute truth. Probably, there are hardly any absolute truths in programming; there are only trade-offs. 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.

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. 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.

A word about Copilot

My publisher had the idea to include Copilot, one of the new AI code generation tools available at the time of writing, in this book. And indeed, there are quite a few cases where Copilot can be useful when writing code. I

tried to create examples that correspond to the chapters. Though I couldn't provide examples for every chapter. In many cases, the content of the chapters was too generic, as if code examples would be very helpful. In these chapters where I provided code examples, Copilot was generally able to generate useful code. Though sometimes I had to experiment a bit to achieve satisfactory results, I didn't always obtain the desired code. There is still a lot of research to be done on how to use Copilot in the best way.

In general, it can be said that AI code generation is already a very useful tool. It can significantly improve your productivity and the quality of your code if used correctly. It also helped me write this book here. Though you always have to be cautious. While Copilot is not perfect, it can generate code that is not always correct. It provides only some suggestions. Or as it is called: "Copilot". It's not a replacement for a software engineer; it's just a tool that assists you with your work. You still have to guide it in the right direction.

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

In this 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. 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. Most chapters consist of my personal interpretations of more specialized books Thus, I mention the book I was reading as a foundation for the corresponding chapter. And of course, all of this book is biased by my personal opinions and reasonings.

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 all so vague. We have to sit down and analyze the situation. I even found moral support from a fellow physicist.

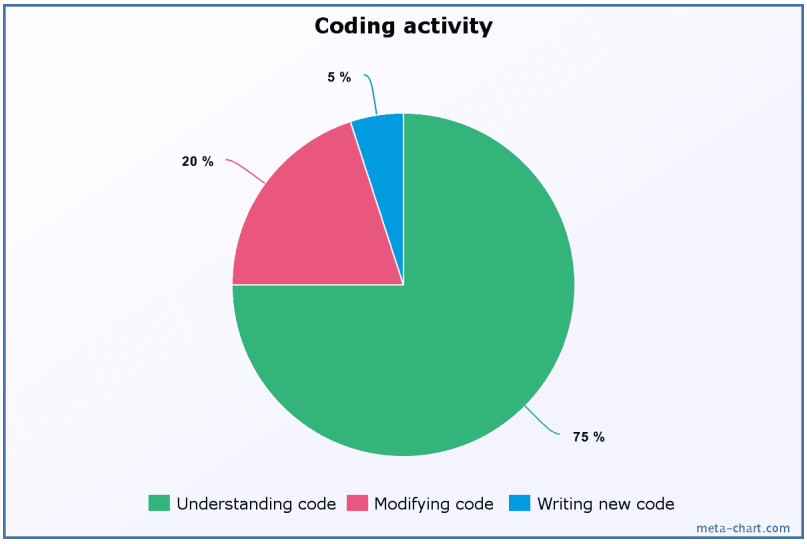
Let me start with a very blunt question: What do you think a software engineer does?

"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.

The first rule of software engineering:

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.

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.

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 third 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.

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 fourth 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.

These four rules will accompany us throughout our book.

//make a list of the 4 rules in a box

- We write code that is easy to understand.
- We write automated tests that cover all of our code.
- We constantly clean up our code.
- We write code to create value for our customers.

7. Good code: a list of rules

// this chapter needs some citations of other chapters.

"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.

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 acceptance tests, and possibly integration 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]. There is no 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 internal [classes]. They should have a strong coupling between the variables [Data types] and methods [Functions] and 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. Creating only a part of an object due to missing information is akin to a supply chain issue in coding. This can become very confusing. 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 (~< 100 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. Apparently, 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.

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.

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 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.

Although practicality beats purity: Yes, there are times when you are allowed to break the rules explained in this book.

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. 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 are not good at math. It's so simple and logical. It took me 12 years of school to learn how to calculate a differential. And I was comparably good! Humans are not made to think logically. The computer can execute the following line of code without any issues, but I doubt any reader 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 we can't break it down into smaller pieces.

We are limited by the amount of complexity we can imagine. 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 your own 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")
```

While it is still generally recommended to use a class instead of named arguments. It orders 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 bad idea. It is clear that they should have structured the arguments using dataclasses. Or, as we have learned, they should have used a toolbox.

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, at least in my opinion, much easier to understand.

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. It 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 `int t == time_elapsed()`. This is a very common mistake.

Scope of variables

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 extremely 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.

It is generally advisable to minimize the number of variables. 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.

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.

"The Single Responsibility Principle advises to separate concerns to isolate and simplify change." - Klaus Iglberger [https://youtu.be/motLOioLJfg?t=1003]

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. When applied correctly, any type of code will become an order of magnitude more readable.

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 them. This could potentially 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 switch-case or 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. 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 there is no longer any need for any if statements.

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

Another option is to use a dictionary.

```
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. 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??// For a three-fold repetition, there are certainly no more excuses. In the case of three-fold repetitions, you have to refactor the code immediately.

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

Every bit of logic is handled in exactly one place in your code. You have no code duplication. You are not allowed to copy and paste any code. Do not Repeat Yourself. DRY. This, however, does not only apply to copy-paste code. It can also happen that there are two pieces of code quite far apart in the code performing very similar tasks. If you see redundant code, you should start refactoring it.

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 updates. Done.

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.

Again, 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 incorrectly decide 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. Yet, 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.

Another example is a company. Every company has a job hierarchy. Even though some modern companies attempt to maintain a flat structure, some form of hierarchy still exists. At every level of this hierarchy, there is a different task. The lowest level comprises the factory workers. They do the actual work. However, the other levels are also necessary. The department head has to ensure that all his employees are content, or at least that they fulfill their job responsibilities. As you move up the hierarchy, the work becomes more focused on the company's strategy. It involves more politics and HR. This is the natural way companies are organized. Big companies won't work in any other way. The CEO cannot manage all 10,000 employees by himself, nor can he know every detail of every process within the company. He needs to establish a job hierarchy. He has to delegate his work and let others take care of the time-consuming details. He needs these levels of abstraction. Self-organizing companies without a hierarchy often do not work out very well.

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. 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.

Creating good levels of abstraction is probably the most important task 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 Myers [Effective 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:

```
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:

```
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. 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 done. Neither in a car engine nor in your infrastructure code.

The domain level

// Is this redundant with the DDD chapter?

//add something about domain levels. Write more exactly what the differences between the domain level and high level code are.

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 you 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 significantly 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. Because 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. Most devices can now handle both voltages.

An example of poor design is the USB Type-A port (used by USB 2 devices). 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 USB 2 device. 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, which is used by USB 3 devices, 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.

// to the reviewers: remove this example as there are too many of them?

Combustible car engines operate best at around 2000-3000 rpm. At lower rotations, the engine could not operate properly; running it faster makes it inefficient and noisy. This problem is mitigated by the gearbox, which allows your car to operate at a wide range of velocities. Now, there used to be a minor issue with the gearboxes. The user had to manually change gears using a clutch. Most car drivers quickly get the hang of it, but it is certainly not user-friendly. Most car drivers only want to get to work, the restaurant, etc. They only want to change the speed of the car. They do not want to worry about either the gearbox or the clutch in their car!

Now there is a well-known solution: automatic transmission. A car can be driven at any chosen speed, and the automatic transmission will select the most suitable gear. Problem solved. You pay a small fee for the automatic transmission, but you'll never have to think about it again. When you push the gas pedal, you control the speed of your car, while all the technical details are managed by the onboard electronics of your car. Now we only have to wait for self-driving cars to eliminate the steering wheel and the gas pedal altogether.

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] on 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.

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.

```
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 answers are 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:

```
# 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`.

```
# 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" [citation: 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.

// TODO search images without copy right

On the left-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 right-hand side. The solution on the left-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.

```
def voltage_to_light_density(voltage):
    # example function that converts voltage to light density
    return voltage * 10
```

Now, this function returns the orthogonal data for this specific example. Of course, the transformation required in your code will 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" [https://youtu.be/MBRoCdtZOYg]

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. Even in Europe. 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? We have to be like lawyers. The law defines every crime as precisely as possible and assigns it a unique name. This is what we need.

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]

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 but 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. 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?
3. 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.
4. `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. Where Magic is being used in a negative context here. 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.
5. 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]
6. 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.
7. 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.
8. 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.
9. 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.

10. Use common English words that are familiar to everyone, and avoid abbreviations unless they are commonly used in spoken language, such as "CEO", etc.
11. 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 *importanter* than the usage of proper English. Natural languages have some deficiencies when it comes to explaining things unambiguously. The following is perfectly viable code in Python: `for fish in fishes`.
12. 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.
13. 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.
14. 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.
15. 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 rules.
16. 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.
17. 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()`.
18. Attach units to a variable name if they exist. For example, `timeout_duration_ms`. Though, once again, consistency is more important.
19. Avoid using negated terms (and preferably avoid booleans altogether). `is_not_empty` is more difficult to read than `partially_full`.
20. 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()`.
21. 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.
22. 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 because "high" and "low level" objects are relative terms. If you are the developer of the `math` library, the `sin` function is considered a high-level object, while for everyone else, it is seen as a low-level object. I don't think this rule is very helpful. I would argue that objects utilized in general literature (at commonly agreed-upon interfaces, typically have short names, whereas items falling in between may require longer names to adequately describe them.

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.

```
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.

```
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:

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

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

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

```
# 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 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. The class variables and the slightly different context are not fundamentally different. Class variables are similar to output arguments. I hope it is generally clear from the context whether an explanation applies to functions, methods, or both.

Do one thing only

Due to the SRP [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. In fact, 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.

Let's create a brief example of a one-line function. We have a pandas object (a Python object for tables) `all_data` and we want to filter it by a `key` and a `value`. I believe this function enhances the clarity of the code that utilizes it as it operates on a higher level of abstraction.

// maybe find another example?

```
#example of a one line function
def filter_data_by_key(data, key, value):
    return data[data[key] == value]

# example code that uses this function
data = read_csv("pupils.csv")
john = filter_data_by_key(data, "name", "John")
# ...
```

I think this code is much easier to read than this:

```
data = pd.read_csv("pupils.csv")
john = data[data["name"] == "john"]
# ...
```

The reason is that the first version of the code reads much more like normal English. The function describes what it does in words. Meanwhile, the second version of the code uses cryptic Python syntax that reveals details you typically do not need to know. And in those rare cases where you need to know, you can still look it up.

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:

```
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:

```
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:

```
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. Thus, 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 swapping 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 example of temporal coupling is 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. 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.

Now, there are very few functions with zero arguments (though of course there are plenty of methods with zero arguments that use class variables as a replacement for function arguments, the argument applies only to functions and not to methods). 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].

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.

Avoid functions with more than three arguments whenever possible. 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.

In classes, the issue of the number of arguments becomes even worse. Methods can additionally access all the class variables. The equation is very simple:

```
Total variables = method arguments + class variables
```

Global variables should not be used; therefore, we disregard them. Still, when combining function arguments and class 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 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.

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. The code does not become more readable when this rule is violated. 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.

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. In my opinion, 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 workarounds 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. Use them if it's not a performance-critical task where reusing an existing data structure is required.

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. Function arguments are delicate as they can cause significant confusion.

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

Copilot

Copilot, like any other piece of relatively simple code, 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}
]

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

//rename to types of classes and break up the chapter?

"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. Therefore, it is time we had an in-depth discussion about them.

This book does not aim to provide an exact explanation of what classes are. Even though the explanations provided here do not require much prior knowledge. This chapter is about how classes should be used. 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 I think 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.

An example of a dataclass:

```
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 helpless.

Classes are very similar to structs. Besides some technical details, the only real difference is the encapsulation of variables and functions (methods), 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 easily 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. Even extremely important. 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, delegating class, and the pure method class, are not commonly used. I created them on my own 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.

Data Class

We have already briefly mentioned the data class 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, 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 too many variables (ideally no more than around eight), it is advisable to split up the data class into sub-classes. This enhances the general overview in the toolbox.

```
from dataclasses import dataclass

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

Pure Method Classes

A class may have no member variables at all. It consists only of public methods. In Java and C#, writing such classes is necessary because every function must be implemented within a class. 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 free-standing functions over classes with only public methods, but I guess that's a matter of taste.

In Python, many functions defined in libraries are not part of classes. This can be seen as follows. If we want to call the `sin` function, we have to write the following code:

```
import math
math.sin(0)
```

But we can also call it as

```
from math import sin
sin(0)
```

If we define the math library ourselves and put the `sin` function inside the `math` class, the code would look as follows:

```
# inside math.py
class Math:
    def sin(angle):
        return # implementation of the sin function

# inside the main code
```

```
from math import Math
Math.sin(0)
```

Having the math library contain free functions instead of a class with only public methods isn't much of a difference, but in my opinion, dealing with a set of functions is more intuitive than having them inside a class.

For everyone interested in C++: In C++, there is no way to determine if a function is inside a class or not. A function has exactly the same signature whether it is a static method of a class or a freestanding function in a namespace. The standard (`std::`) library, for example, is a namespace. This has the huge advantage that the standard library can be split up into many small parts. You can import only what you really need.

Delegating Class

The delegating class is a combination of a data class and a pure method class. All variables are private, and all functions are public. Calls to one of these functions are all delegated to one of the member variables. Thus, the functions are all fairly simple, usually containing only one line of code. 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. Any other part of the car does not need to know anything about the temperature or its control.

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

    def set_temperature(self, temperature):
        self._AC.set_temperature(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 a 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 to 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 three member variables and about 100 lines of code, depending on the general complexity of the class. Consider using a few functions instead of a worker class. 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.

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.

```
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_entries()

    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:

```
def add_entry(number, data):
    # some complicated logic
    data.append(number)
    data.sort()
    read_out_entries(data)

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

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 divide 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.

```
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.

```
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.

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 many 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()
    lion.feed()
```

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.

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. These problems cannot occur when using abstract base classes instead of `override`.

Other Types of Classes

I remember once having written a class that contained some public variables and one public method. Unfortunately, I don't remember why I wrote such a hybrid class. It shows that the types of classes above are not a definitive list; however, I think it's still a good rule of thumb.

General Recommendations

It is recommended to stick to the class types mentioned above. 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 single responsibility principle.

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.

// new chapter here?

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
```

Both lines of code have in common that the method/function `b` can modify the value of `a`.

It turns out that for most criteria, these two function or method calls are pretty much equal:

- In `b(a)` it is assumed that `a` is not changed unless the name of `b` states otherwise. Though the same holds for `a.b()`.
- `b(a)` can access only the public variables and functions of `a`, while `a.b()` can access everything. This is probably the biggest difference here. Thus, `b(a)` decouples the code better than `a.b()`. Though this is not a crucial difference.
- In C++, constness can be enabled for `a` in both `b(a)` and `a.b()`. In the first case, `a` has to be passed as a `const` element; in the second case, `b` has to be made a `const` method. In Python, there is no built-in way to create a constant function. Though with some black magic, one can create a decorator that

> makes a function or method constant. [https://stackoverflow.com/questions/71394120/Python-equivalent-for-const-methods-with-type-hints]

In summary, there are no significant differences. `b(a)` offers somewhat better decoupling and should therefore be somewhat preferred. 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

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: 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]

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!
if obj.id == 24:
    # ...
```

Now, if you add one more class instance before this line, the counter will be different, 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. When the developer clicks a button in the IDE, public getter and setter methods are automatically generated. For each private variable, a setter and getter method are created to access and modify the 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)]

Here we 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
- 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 very odd to have only variables written.
- 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 [section ?]. 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...

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

The other case is mostly the "normal" worker classes. It is very possible to have getter and setter functions here. If they refer directly to a variable within the same class, it is a hint that your class design is bad. Access functions should elevate you to a higher level of abstraction, whereas the plain getter and setter methods do not achieve that. Instead, they have to access or change data located in a nested data structure. They just make it more convenient to get there.

But again, the 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 generally no reason to write getter or setter methods in worker classes.

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. 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.

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

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

The car class only delegates the `set_temperature` function call to the `air_conditioning`. And also, the `air_conditioning` does not have a temperature variable. It has only a temperature sensor that measures the temperature and has the capability to adjust the temperature.

This might be just one example, but there are no other variables in the car where you can directly use a get or set call. It doesn't enhance the code by creating a `get_air_conditioning` function.

I hope I managed to convince you not to write bare getter and setter methods for member variables. Especially not to all at the time.

Coupling and Cohesion

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

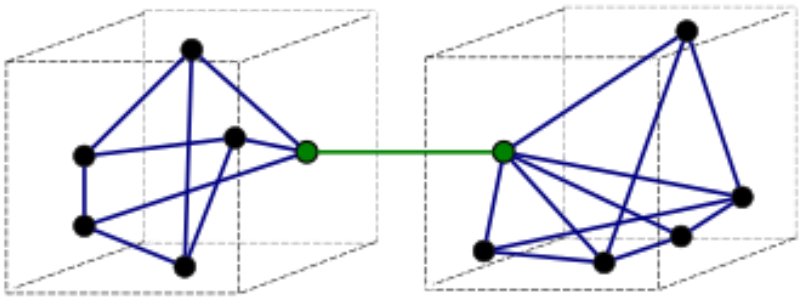
"Classes should have high cohesion within themselves and low coupling between each other." - Robert C. Martin

If you don't understand these expressions, we could rewrite it as follows: "There should be significant interaction among methods and variables within a class 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.

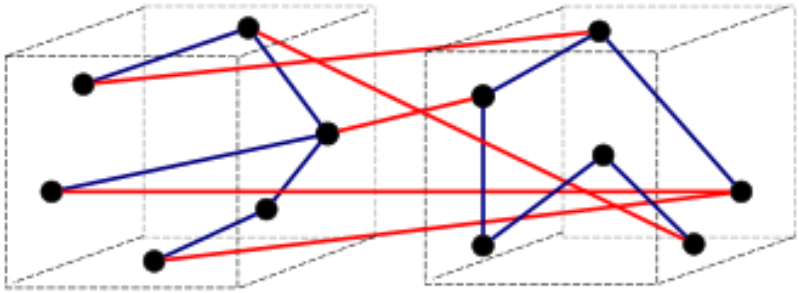
Worker Classes

The rule defined 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]

Unfortunately, it is very difficult to provide an example of such a complex class here. Thus, I can only explain here in rough terms how you could break down such a complicated class into smaller components.

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. 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. Additionally, 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.

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 similarities. Splitting a data class requires no 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 can be nearly impossible.

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 "free" 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 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 certain 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] 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. This makes the methods 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.

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 may not 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 getter and setter functions for your classes. They don't improve anything.

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

"Favor composition over inheritance" - Basic rule of class design

Or my version of this rule: "Use composition, not inheritance"

"Inheritance is the base class of evil" [https://youtu.be/2bLkxj6EVoM?list=PLM5v5JsFsgP21eB4z2mIL8upkvT00Tw9B]

[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.

```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("a derived")
```

There is not much more to say about interface inheritance. It is a good thing, and you should use it. It is a method to define interfaces and implement them. In C++, there is no way around it; in Python, you can omit it if you want.

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. Thus, when I write about inheritance in this section, I always mean implementation inheritance.

```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 therefore be avoided whenever possible. In the C++ Core Guidelines, there are at least a dozen points to consider when working with implementation inheritance [C++ Core Guidelines, C++ Core Guidelines explained (Rainer Grimm)]. 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 or 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.

One of the main issues here is that all levels of inheritance have access to the variables at the base level. The variables and methods are not sorted. They are just there. Meanwhile, with composition, you only need to

consider the location of variables within the class. This clarification helps identify the origin of a variable and likely deters you from delving into objects nested deeply in the hierarchy, unless you intentionally choose to do so. Therefore, one can say that composition properly implements encapsulation, while inheritance does not.

Furthermore, 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.

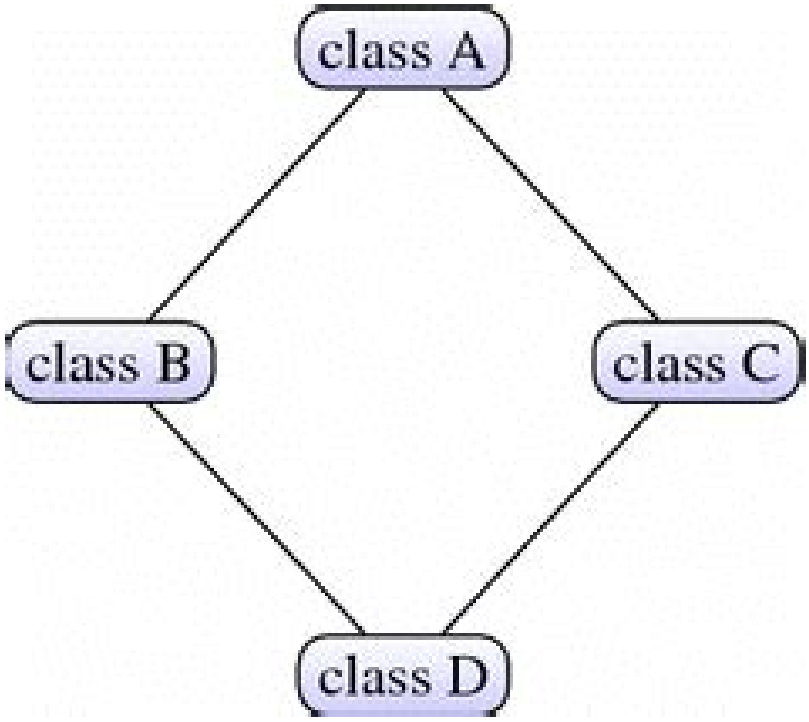
Obscure code

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 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. If you follow my advice and avoid using inheritance, you won't have to deal with such technicalities.

// 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:

```
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. As I mentioned, in my opinion, there is simply too much that can go wrong with inheritance. Inheritance 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 concept I can think of is code reuse. But it is not worth the drawbacks that come along, as mentioned above.

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

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.

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. Inheritance is a common source of poor code quality. It should be avoided if possible. Use only interface inheritance.

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]
```

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!

The second problem is that we don't know the meaning of these numbers. There is no appropriate name for this list that fully explains its contents. This list violates the single responsibility principle all by itself.

And the third code based on this data structure will inevitably become brittle. It's screaming for bugs. You can pretty much do everything wrong, and I promise you will.

Apparently, three values inside this list always belong together. In C++, we would create a struct for it; in Python, we use a data class.

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.

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 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

# 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 error. Avoid using string comparisons as they are prone to errors.

Sometimes, objects of this kind are also referred to as "stringly typed". Strings are being misused for storing various types of data for which they are not intended. Here are some examples [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
```

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. Unless this is a well-known international color standard. For example, in the RGB standard, `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 because you most likely did not define a color `BLU` inside the enum. You will receive an error message. 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."

I may be exaggerating slightly. But it's true. Humans struggle dealing with Booleans and if statements. Accept your fate and learn to deal with it.

- Good code design results in fewer if statements.
- Polymorphism can be utilized to avoid using if statements.
- Resolve `if` statements at the lowest level of abstraction possible.
- 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.

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 function or use a dictionary. The only place where `match case` statements (or nested `if else` statements) are allowed is encapsulated inside a dedicated function.

This is not how the code should look.

```
# a lot of code here
# city_name = ...
# use a match case statement to get the post code
match city_name:
    case "Zurich":
        return 8000
    case "Bern":
        return 3000
    # case ...
```

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

post_code_Zurich = 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,
}
```

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():
         return 8000

class Bern:
    def postcode():
        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.

Strings

"You should never use two different languages in a single file. English is also a language." - Unknown

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"

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, Uncle Bob somehow went haywire. What he explained all made sense, but he somehow 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. At least it would be if it weren't so outrageous what he did here. Avoid using strings to encode values of a different type. To make matters worse, the letter `"h"` is even used twice. This code becomes more fragile because the state relies on the order of the characters.

The unit tests written by Uncle Bob are 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.

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 one 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 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

Serious 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, and the translator also doesn't want to deal with your code. He wants 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. Instead, the code should read all the human-readable text from this specific file. 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.

Dicts

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, 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 what kind of ingredients will be needed.

Dictionaries are closely related to JSON and XML files. They are essentially similar to a nested dictionary converted 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 for the corner cases.

Pointers

C++ 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 essentially all the functionality that pointers were used for. 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. Be happy if you use Python because you don't have to bother with pointers at all.

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, or global. And possibly many more. 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 due to 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 permissions 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.

In C++, there is also the `constexpr` keyword to indicate that an expression can be evaluated at compile time. This allows the compiler to perform various optimizations, as many expressions can be evaluated at compile time.

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 constant. If you want to change a variable, you have to create a new one.

Constant Class Instances

In C++, you can enforce an object to remain the same for as long as it exists by using the `const` keyword at the time of creation. In Python, you cannot enforce it, but you can use the all uppercase notation to hint that a variable may not be changed. The usage of `const` is straightforward, even though it might be a little confusing at first. Everything that should never be changed should be kept constant and defined at the time of creation. Once you create an object, it will remain the same until you decide to discard it (once it goes out of scope). This makes the life of a programmer much easier and prevents the abuse of variables. This is once again related to power. Life becomes easier when you lack the power to alter an object that should remain unchanged. It prevents you from making a mistake by altering the object.

// But things are not always that easy. In some cases, you might have all properties as constants except for one. Or is it? Let's look at an example.

// remove this example completely? Is it useful and "correct"?

We have a bottle. It has a color, size, and material. All of them are fixed when the bottle is created. In our code, that would be the constructor. These properties can never be changed. You could only replace the bottle with a different one. An instance of the Bottle class should be declared as constant. In Python, constant objects are defined using all uppercase letters.

```
from dataclasses import dataclass

@dataclass
class Bottle:
```

```
    color: str
    size: float
    material: str

if __name__ == "__main__":
    BOTTLE = Bottle()
```

Note that color and material should probably not be a string, but rather an enumeration of `Color` or `Material`, respectively. We only use strings here for the sake of simplicity.

The member variables of the class are not constant. Instead, I decided to make the class instance constant.

// classes should not have const members; instead, they should be private. [https://youtu.be/O65IEiYkkbc?t=3072]

So far, so good. We created a bottle that is constant. However, having an empty bottle is quite useless. You want to use it. You fill the bottle with water and drink it later. This is a very simple example, but in our code, it raises some fundamental questions. We have a constant bottle object with a not constant amount of water in it. How are we going to solve this problem?

The first attempt involves removing the constness and adding the amount of water to the class variables. This is a dreadful idea. Now anyone could change the color of the bottle. Never remove a const'nes from an object that is constant.

We make all variables private and write getters for them. The amount of water can be adjusted using the functions `fill` and `release`. This is an improvement, but it still feels wrong. I have a grudge against getter and setter functions as explained in the chapter [?].

A completely different idea is to keep the amount of water separate from the bottle. We create a new class `WaterBottle` that contains a constant `Bottle` and a variable amount of `Water`. This also prevents the `bottle` class from growing too large and helps maintain structure in the code.

```
from dataclasses import dataclass

@dataclass
class WaterBottle:
    BOTTLE: Bottle
    amount_of_water: float

@dataclass
class Bottle:
    color: str
    size: float
    material: str
```

This is just one example of how to combine const and non-const objects. One always has to consider whether an object or only parts of it should remain constant. Such considerations are important as const'nes is a crucial property of variables. You shouldn't consider const'nes as restricting you, but rather as something that fixes certain behaviors.

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, experiment with 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 violates 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. This creates 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.

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. 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. Not your friend. 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 global variables are pure functions (yes, I compare variables with 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 change it later on.

Here is a rough list of how variable types are sorted by the accessibility they have, starting with the least. It is not so easy to compare them all, so this comparison here should be taken with a grain of salt.

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. Or at least not much. 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 (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 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. So, you shouldn't have output arguments anyway.

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.

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

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" Marco Gähler

=> 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.

After 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 more than enough companies that would actually pay this amount for such a feature. It's an enormous amount, but at the same time, the efforts required 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 would know. 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 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. Ask the internet and others for advice, and you'll get a fairly good idea of how to write them.

Test Examples

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:

```
# 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:

```
# 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

```
    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:

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

This returns the following error message:

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

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. Write tests for public functions to assert that you don't accidentally change their behavior.

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 fairly explains what it tests. These names may be up to one line long if necessary. You don't use these names anywhere else, so it doesn't hurt 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

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

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 manipulate one of them. You still have something to check 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.

```
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 as well. Each change has to be applied in both, the code and the test.

Understand what you do

There are a lot of things to consider when writing tests. The example above was very simple. In real code you have to deal 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 no, there is no need to start writing a test. It would never work. It would be a waste of time. Rewrite your code to make it simpler or get someone to help you understand the problem you should solve. Don't write anything, unless you understand the problem and you know what you want to do.

A few tips

Make sure all the tests pass. Tests that don't pass are worthless. Even worse, they are a nuisance. When you run the tests, failing tests will confuse you. They will confuse you're coworkers. Everyone will waste time trying to fix the failing test. There is only one single solution to prevent this: all tests have to pass all the time. Thus make sure your Continuous Integration (CI) enforces that all tests pass. Tests that don't pass should be deleted.

In the setup part it is very common to have helper functions that create all the objects needed. These are normal Python functions that create the desired objects. You might even have a util code file for all the tests. It

contains some fairly static objects like functions or class instances that you might need in a lot of different tests.

There are also some things to watch out for in the execution part of the test. The first mistake almost everyone made was checking two floating point numbers for equality. Due to rounding errors this will probably fail. There are dedicated approximate checks you should use instead. As the `isclose` function used in the example above.

Then it is common to miss some of the if-else branches. These are very important to test as most bugs usually hide in conditional statements. Make sure that you cover all cases, if necessary using a code coverage tool.

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

You should make sure that you always change only either the code or the tests if you refactor code. This is the only way you can be sure that the changes are correct. If code and tests have to be changed at once, the changes should be straight forward.

Quality of test code

Tests are somewhat special, but ultimately, they are still code. Probably the test code is longer than the actual production code that you work with in a serious project. Thinking about it, it becomes apparent that when writing tests, some coding guidelines have to be followed as well.

With tests it's even more important that the code is easy to understand than with normal 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. Do so by refactoring it as you would with any other code. Keep the functions short, find appropriate names and remove excessive duplication. These things frequently get overlooked when writing tests. Even tough the requirements on test code are somewhat different than for production code.

Number of test cases

Probably the hardest decision is what values you want to use in your tests. Writing one test case for a function is much better than nothing. 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 about two random values. As you wrote the function, you will know the corner cases: Division by zero, passing an empty array as a function argument, the file used does not exist, 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.

For functions with many arguments it becomes very tricky to write tests. If you have 3 arguments and for each one you would like test 3 values you end up with `3^3 = 27` test cases. This is quite a lot. Now you really have to make sure you understand what you are doing.

Here is an example of a function with three arguments. I have not written down all the test cases, but you can guess how tedious this 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
    # ...
```

There might be cases where 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, thus `3*3 = 9` test cases. This sounds much more reasonable, though it's still quite a lot.

Usually the function arguments are not independent, or at least it's not so clear how they interact. Otherwise they wouldn't be in the same function. And it's genrally not feasible to write 27 test cases, that's just too many. Just do your best instead. Try to test all corner cases and add a few random ones. If the function consists of well written code that doesn't look like hiding bugs deliberately, you should be pretty much fine. And even more important: try to keep the number and complexity of the function arguments low.

I'd like to state again that it is important to really know what a function does. As I already wrote several times you have to test the corner cases. And you only know them if you know the code. Many corner cases you won't find by chance. Nor can you figure out if 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, he's missing this very crucial information and either has to read and understand all the code or just guess what it does. Both cases are suboptimal.

You may also have structured objects as an input or output of a function. This can become worse than having three variables by orders of magnitude. The structured objects may have a thousand fields, for example elements in a list. Everything we discussed so far becomes peanuts. But we can still achieve reasonable test coverage if we try. First of all, all elements in a list have to be treated equally in your code. Never use 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 usually fairly independent and can be tested accordingly. Most of the entries from a large structured object are probably not even needed inside a function as the object is fairly generic. If you have a nested struct as a function argument, only pass the substructs that are actually used inside the function. Only change the values that really have an influence on the object under test. This way you can reduce the number of test cases significantly.

Again, it all comes down to the programmer knowing the relationship between different objects. Which parts of the object are really used inside the function? Write one generic test with default values for every part of the object. Then write also tests for some specific values of the object. Though again, here you'll have to figure out which the important values are.

As you can see, the most complexity in tests originates from sub optimal code. If you write good code, the tests will be easy to write. In good code, there are few arguments with little 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 compareably easy.

Stages of a test

As we have seen, a test generally consists of 3 stages.

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

The second stage is the execution of the test. Here you run the function you want to test and assert that the results are as expected.

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

Setup and Teardown

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

Setup and teardown are the functions automatically called at the beginning and the 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. And at the end of the test, the interpreter or compiler cleans up all the variables as they go out of scope.

In most cases, and especially in unit tests, there is no need for a teardown function. Everything a unit test should do will be cleaned up at the end of its execution. However, if you wirte acceptance tests that use text files, databases or something else that is presistant, things become tricky. Your tests might need temporary files, change values in databases, have network connections, etc. It becomes messy. You need a fool proof way that your file handling always works the same way, irrespectively of the outcome of a test. Even if it throws an uncaught exception. This is where setup and especially teardown really come into play.

For the file creation there is not that much that can go wrong. You create it from code or copy it from another location. This is to be implemented in the setup part of the test or using some function. If you copy files or databases, make sure the original file is write protected. Otherwise you might change it by accident.

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. Because you'll probably change the order of the tests at some point and 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 with your test design is seriously flawed.

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 normal function call for deleting the file will never be executed. There would be a mess of undeleted files. This might impact future runs of the tests and they might become flaky (sometimes they pass, somethimes they don't). And flaky tests is one of the worst cases as it confuses everyone.

This problem can be solved by implementing the teardown function that is guaranteed to be always executed, no matter the result of the test. This is where the dedicated teardown function comes into play. It is guaranteed to be executed, even if there is some error occurring inside the test. Only in very serious cases like a segmentation fault, the teardown may not be executed. Though this is only a problem with low level languages as C++.

Anyway, try to write tests that don't need files or IO. 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 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 teardown_class(self):
        print("teardown_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 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.

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 normal code. As written above, in tests DAMP is more important than DRY.

Each test has exactly one purpose. It tests exactly one function or method. Testing many functions inside a single test is bad practice. Write helper functions for the setup of a test and then writing an additional test cases will be simple. 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 make an example 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 this 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 removing some duplication.

One last remark to this test: If you are not used to writing unit tests, you might think that the test name is quite long. 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 even good practice to have a long test name as it makes the test report more readable. A test name of about 50 characters is completely normal.

Test body

Most things about the test body have already been said in the example above. The test body is generally consisting of one function or method call, followed by a few assertions. There are some purists, saying that a test should contain only one assertion. I'm not sharing this opinion, even if they have a point. I think there are cases where it doesn't hurt to have more than one assertion in a single test case. Otherwise the tests become too verbose in my opinion.

Problematic tests

Just as with normal code, there are some signs that a test can be problematic.

Dependent tests

It will happen frequently that you have a test that can only pass if another test passes. They are coupled. For example, you have a function that creates a file and wirtes a number to it. You write a test that calls this function and checks the existence of the file.

Next you write a function that reads the contents of this file. In the 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 won't be possible to read it. If the first test fails, the second test inevetably fails as well. This kind of dependency is bad design and violates the SRP. For one failing feature, only one test should fail. This makes it much clearer where the error comes from. Having 50 failing tests at once is really annoying as it's not obvious why the tests fail. Is it for a single reason or do they all fail for a different reason?

Unfortunately having all the tests completely separated is a very hard, 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 degree. In Python you can skip tests if a requirement for the test is not met. Tests can be made depending on each other with the `@pytest.mark.dependency` attribute. This allows us to skip tests that would fail as another test already failed.

```
pip install pytest-dependency
```

```
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, `test_b` will be skipped as it depends on `test_a`.

The output will be `1 failed, 1 skipped` as the `test_b` was skipped. And in the short test summary info, only `test_a` is listed as failed. Only once `test_a` is fixed, `test_b` will be executed.

For unit tests, dependent tests are generally not a problem. Each test covers only one unit which shoudln'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 each other. 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'll be 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 rather that it destroys the confidence of the team into 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 save tests. Write for example a test that checks the network connection. All tests relying on the network connection will have a dependency on this test. Structuring the tests like this can reduce the flakiness by orders of magnitude. And it is very important to design your tests such that flakiness cannot occur.

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 things that can fail, like on the file system nor on network connections. This is one of the reasons why you should avoid testing any input/output (IO) for unit tests and reduce it as much as possible for all other tests.

The following test is flaky and always fails in the morning:

```
from datetime import datetime

def test_time():
    assert datetime.now().hour < 12
```

Of course this is a pretty dumb example but it's less exoctic than you may think. Tests (and code) have probably already failed for similar reasons.

Brittle tests

Tests that are over 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. Instead testing the formatting is just a waste. Even worse, it is a needless liability because it tests something that should not be tested. Something the result does not depend on. Instead use a json library to get only the actual values stored in the file and compare those. This is what we are really interested in. Never use any string operations when reading a json file. This is the very definition of brittle code!

Another example of brittle tests are tests for methods that should be private, but are made public in order to test them. This prevents you from refactoring this function as it is now part of the public interface. Changing it will break the tests, even if the real public interface is not changed. This is why private methods should not be made public in order to test them. If you really feel the urge to test a private method, you should refactor it into a separate class.

Here is an example of a brittle test. Again, it is a pretty dumb example. But I'm sure people have already read out json strings character by character. Please always use libraries instead for such purposes. Otherwise the code becomes brittle.

```
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}
```

Random numbers

If you ever use random numbers in your code, you might get stuck with your tests. You think. Because how can you test something that’s random? Well, you can. Your random numbers are usually not really random. Your computer fakes them. It uses an algorithm to create numbers that look random, but it's still creating numbers in a deterministic order. Always use exactly the same random number algorithm and seed (starting value) to get reproducible results for every test case. Only use real random numbers once you ship your software.

The Beyonce rule

A common question is "what to test?". A very simple answer is: everything. This is certainly a correct answer, though you cannot always test everything equally extensively. You'll simply lack the capacity for that. Instead, at google they came up with the Beyonce 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 holds for most of your code. You do like your code, don't you?

Not Automatable Tests

As software engineers, we want to automate everything, tests included. However, this is not always possible. There are still things that we can barely automate. One example are image processing algorithms. How much can you compress an image such that it still looks good? This is very hard to tell with an automated test and is better judged by humans. Also if you run some complex simulation, like the aerodynamics of an airplane, you cannot write a test to check that your simulation yields the correct result. Simply because you don't know the correct result. You can only judge from your experience that the result makes sense. 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 mixture of two different types. But in general, the following 3 categories are sufficient.

1. Unit tests test the behavior of individual functions, classes and modules.
2. Integration tests test interaction between the modules.
3. Functional tests test the behavior of the complete software.

As we will see, each of these categories has its own right of existance 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 and some of them we will go into more details later on, while others we just 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 called end-to-end (E2E) or acceptance tests.

The small unit tests are the foundation of the testing infrastructure. They can be executed quickly. Meanwhile going towards bigger tests, they may take longer to execute and are testing the interaction of components, rather than individual components themselves. Thus, bigger tests are more likely to find bugs, but at the same time they are not appropriate for locating 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 of some C++ software in Python as it was easier to process the resulting text files.

Unit tests

First we have to figure out, why unit tests are acutally needed. A lot of programmers work as follows: They write a function and then they have to figure out if it works correctly. In order to do so, they use print statements or use the debugger. They run the code and check if the results are correct. Let's look at the following example.

```
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 thrown away and no one knows anymore what the code is actually supposed to do. Whether it still works. When changing the function, you have to test it all over again. Everything. Every time. By hand! This is a typical example of a procedural DRY violation that should be optimized away. And the solution are unit tests.

Unit tests cover comparably small pieces of code. Usually they test a public method or a standalone function. In the example above, the unit tests would check everything that is checked using print statements. The unit tests for the `square` function would look something like this:

```
def test_square():
    assert square(1) == 1
    assert square(2) == 4
    assert square(5) == 25
```

This does pretty much the same as the print statements above. But with the very important difference that this test code here is going to stay. It goes into the test suite and it will stay there forever. 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) and that you'll have to change 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 change the according unit tests.

It may sound surprising to you, but unit tests are the foundation of the testing infrastructure. They are even more important than functional tests. This is because unit tests are fast and they can give you precise information about what piece of your code contains a bug, meanwhile functional tests can only tell you that something is wrong within the whole code base and they take a lot of time to execute. Therefore, having your whole code covered with unit tests will have a similar effect as having it covered with functional tests. However, unit tests have the advantage that you'll know quite exactly where an error occurred and they are much faster to execute.

The drawback being that unit tests do not test if these building blocks are connected correctly. Unit tests cannot test the interaction of different code blocks.

Testing files in unit tests

Usually, unit tests only need a setup and an execution phase. There is no tear down function required as unit tests don't interact with any files or databases that you would have to delete in the end.

"Why...? How? No files? No database?"

Yes, good point. According to the SRP, a function or class should do only one thing. Therefore it should not read a text file and do some complicated calculation. Reading a text file should be done in a dedicated function. This function will not have a unit test. But it is not necessary as reading a file and returning it as a string is no difficult task that needs to be tested automatically. And it will be covered by functional test.

Let's say we have the following code.

```
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 reading the file is very simple. It is not necessary to test it. Instead, it can be easily exctracted into a separate function. This is called 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 wraped the code reading out the file into a separate function. The rest of the code is written into a dedicated function. For this function one can easily write a unit test because it doesn't 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 and the remaining test becomes much smoother. In this case here this small layer is the function `read_share_values` which reads the file into a string. Uncle Bob calls this a "Humble Object" [Clean Craftsman p.157]. It is a small layer that is unlikely to fail and therefore does not need to be tested. It is just a thin wrapper around the function reading the file.

The same holds also for database access or the current time value. You write a small wrapper function that does nothing but calling the database or returning the current time. Then you pack everything else into a separate function that you can test.

When writing integration or functional tests, an even better solution is writing 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 probably the most important part of this chapter. This is not only because of the prevalence of classes, but also because classes tend to become messy without any unit tests.

First of all, classes tend to become too big. 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 under test. 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 deal with private methods in big classes. Apparently, the testing framework doesn't have access to private methods. No one has, except for the class itself and maybe some friends classes. A first attempt is making the private methods public. This, however, is not recommended. You should not make methods public, only in order to test them. This will lead to crippled code with too many public methods, which is the exact 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 also 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 also some private methods, you should resist. This is a clear sign that your private methods are too complex. Make these private methods a class on their own with a public interface that you can test.

Classes that are hard to instantiate are another problem. For example, if an object is hard to construct, or 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 real code, it may be guaranteed that all the required conditions are met such that you never run into trouble. For instance that 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.

As a summary one can say the following things about classes and tests:

- Classes should be small and contain few member variables
- If you feel like testing private methods, you should refactor them into separate classes
- The constructors should be simple and not rely on any fancy logic

All these rules are implied by the topics we covered so far. But now we have a reason why we absolutely have to obey them. The unit tests force us to do so.

Here's an example how to refactor a complicated 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 += 1
        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.

```
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 += 1
        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 anyway belongs to the `Engine` class, not into the `Car` class.

Copilot

Copilot can be a significant help when writing tests. I wrote a function to convert literal numbers into roman numbers and created a unit test file. Copilot started implementing the unit tests right away and without further instructions.

```
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,

```
    # 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 wanted this change, when looking at the code it is not quite clear if this is an improvement over the original code. We have removed some redundancy and use only one assert. On the other hand the redundancy was not that bad and the old code was very easy to understand, which is maybe even more important than removing the repeating code. This is a decision that takes human judgment and I'm still not sure which one is the better solution.

Integration tests

Integration tests lie in between unit and functional tests with respect to all: size, granularity and execution time. Integration tests only test the interaction of several pieces of code. The interaction of several libraries/modules for instance. The public interface of the libraries under test is not connected to other libraries, but rather to fake or mock objects. These allow a library to be tested alone, without creating a functional test.

Integration tests, at its own right, are just at least as important as functional tests. Integration tests are different from functional tests as they use fakes and stubs to mimic the behavior of collaborating objects, while functional tests use the real objects.

// is there anything else to write here? About fakes, mocks and stubs?

Functional tests

Functional tests do what most people would intuitively expect from a test. Some marketing person, e.g. the Product Manager (PM), orders a new feature. He tells you, more or less exactly, what this feature should do and gives you some examples. The feature is complete once these examples can be executed with 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 was changing the underlying code. So there is one thing you'll always do: for every new ticket you write a functional test.

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 failing examples in your documentation.

Functional tests are user centered. The user doesn’t know anything about the internals of the code. He doesn't want to know anything about the internals of the code. He has only the interfaces you give him: GUI, API, keyboard, webcam, etc. And this is all he cares about. He wants to watch a YouTube video. He wants a high image quality and a 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 the input of a webcam sounds pretty hard.

True. But when making a few simplifications, the effort becomes fairly 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 only of some html and CSS code, images, buttons and graphs. These things are hard to test automatically, but they contain no logic that is likely to contain bugs. As mentioned before, this is called a Humble Object. This layer is hard 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 quite certainly fine. It is a thin layer that doesn't contain any logic and it's not able to hide bugs.

Of course, if you leave away the GUI layer, the tests become similar to integration tests. It is up to you to decide whether you want to call them integration tests or functional tests. I prefer to keep calling them functional test as the API is still a public interface to your software.

Writing tests on the GUI level is quite hard. Though there are tools, for example Selenium [https://www.selenium.dev/], that automate the 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 just too many programs that are not structured as recommended in this book. They cannot be tested otherwise as they don't have an API that is well separated from the GUI. Meanwhile there is considerable demand for testing these programms never the less. Needless to say that using these testing tools adds significant overhead to the testing efforts required.

Testing on the API level is in comparably 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 result and you’re done. However, there is one problem with functional tests. In practice you have to deal with potentially huge files, databases and slow network connections. This may slow down your tests considerably. Additionally, the files or databases first have to be created. This can be done either with some script or by copying them from another location.

The output of the tests may be potentially huge files as well. Comparing the results of these big files may also be not too helpful. One tiny difference in these huge files won't tell you much about what is broken. One option for improving the performance is comparing hash values instead of comparing complete files. It won't tell you more than that the files are different, but at least it is much faster to compute.

One solution is to use small files. This makes the tests run faster. However, having only tests with comparably small data sets and files is not representative to the every day usage of your software. You absolutely have to run also performance tests with realistic data sets. Otherwise you might run into all kind of performance problems at the release.

Functional tests are important, but they cannot tell you where an error comes from. Furthermore, functional tests are quite frequently highly correlated. A single bug in your infrastructure code can cause many tests to fail. Therefore it is important to make functional tests `@pytest.mark.dependency()` as explained in the section on [Dependent tests]. You should always combine functional tests with unit tests in order to locate the source of the bug.

Other kinds of tests

Performance tests

One test class that frequently gets forgotten are performance tests. Functional tests frequently are created for small databases in order to reduce the 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 are running with realistic parameters to prevent bad user experience due to slow response times.

There are many different kinds of performance tests. The most common one 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 find the limits of the system.

Explorative tests

Explorative tests are written for finding bugs that the developer might not have thought about. They are usually executed by the testing or quality assurance team. They are not automated and are not part of the test suite. They are just executed and if a bug is found it is reported to the developer. Otherwise it is just ignored. Explorative tests are not a replacement for unit or functional tests. They are just an additional tool to find bugs.

Executing explorative tests takes some experience about what could go wrong. What corner cases might have been missed by the programmers?

When to run tests

It is very important that all tests, explorative tests excluded, 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 fasts. Each one of them takes only a few milliseconds to run. All together they shouldn't take more than a few seconds. Split them up in subgroups if your program becomes too big and it takes more than a few seconds to run them all. It is important that unit tests are fast as they are run all the time. You should run them every few lines of code that you wrote.

Code is only allowed to be merged into master if the all unit tests pass. This means that every programmer has to run the unit tests before creating a merge request (MR) [devops] the same way as he has to make sure the whole projects compiles. It is mandatory to fix code that broke unit tests, otherwise it won't be merged.

Now let me repeat: It is mandatory that all unit tests pass before an MR can be merged into master. 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 formating and the compilation of the code. This is just another mandatory requirement inside the MR, along with the fact that the code has to compile. This is the only way to ensure that the unit tests are guaranteed to pass all the time.

With functional tests it becomes a little bit trickier. Functional tests are slow and can't be run before every MR. It would slow down the whole development too much. Therefore you can't guarantee that all functional tests to be run all the time. Instead you have to set up the CI to run the functional tests overnight ("nightly build"). And if a test fails, it should send an email to all the developers who changed something the last day that the tests fail. The team then has to sit together and figure out why this is the case. 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 gets resolved as soon as possible.

Integration tests are somewhere between unit and functional tests. If they take only a few seconds to run, they can be run before every MR. If they take longer, they should be run overnight. The same holds for performance tests.

Who should write tests

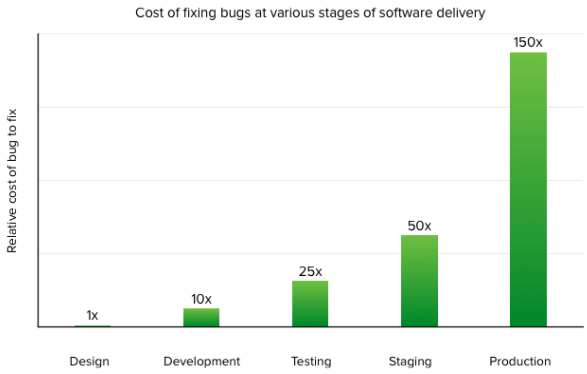
With unit and integration tests, it's quite clear that the corresponding developer has to write the tests. He knows the code best and he knows what the code is supposed to do. Unless you work in the automotive, medical or aerospace industry, where the tests are written by a dedicated tester as they are highly regulated.

With functional and performance tests, the situation is not that clear. Should the tests be written by someone from the development team, from the marketing side or by an independent tester? As always in software engineering, such questions have no easy answer. There are just some trade offs to be made between the different solutions.

Having a developer write the tests has the advantage that he knows the code. He knows where the difficulties lie. He can target these difficulties by writing dedicated tests. A developer might also know what the custormers want and where generally the issues are. This helps as well to target the most important areas of the code.

On the other hand, having an independent tester has some advantages as well. He doesn't know about the weaknesses of the code. Instead he writes more explorative tests. These tests might find bugs that were not expected by the developers as they are in areas of the code they were not expecting to contain bugs. Additionally the developers are usually over confident about the quality of their code. They think the code is better than it actually is. This is why it is good to have an independent tester who is not biased by the code. Furthermore, independent testers are usually closer to the customer and write tests that are closer to the actual use case.

Tests should be written as early as possible. This is not only true for unit tests, but also for functional tests. Writing tests at the end of a project has the drawback that possible issues will be very hard to resolve as the whole software is nearly finished and making changes has become very difficult and possibly expensive. As a general rule of thumb, the price of fixing a bug increases exponentially with time.

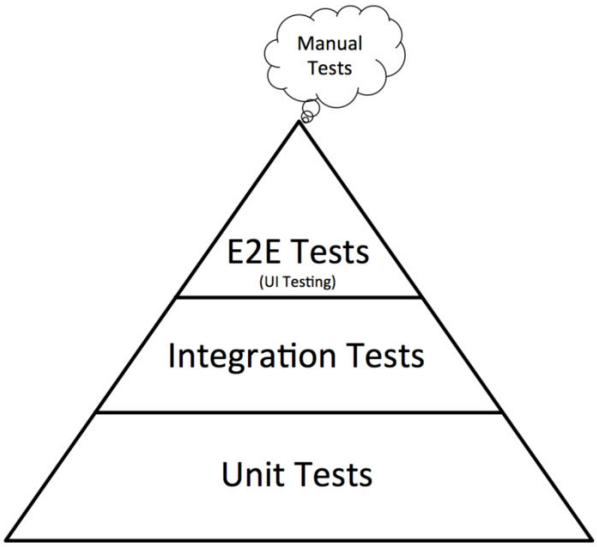


// get an image without copy right

The testing pyramid

We've defined here 3 categories of functional tests, additional to the performance tests and the exlorative tests. From the fine grained unit tests up to the very coarse functional tests. As a rule of thumb one can say that the testing suite of any program should consist of a lot of small grained and few 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 return a detailed feedback if something is broken. They are like testing the individual parts of a car radio before assembling it. Unit tests prevent the usage of faulty parts. Roughly 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 pyramid. They are like testing the assembled radio in a test stand. Integration tests are coarser than unit tests and can't locate errors as precisely. But they are still useful to check the functionality of the radio. About 15% of all tests are integration tests.

The functional tests should only check that the installation of the radio in the car worked out 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 to check that a program really works. There are always some things that can go wrong, even if all unit tests pass. However, the feedback you get from an functional test is very limited. It will mostly tell you that something is off, but you'll 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, chances are low that you'll have a lot of failing functional tests.

Once you know that the engine, the gear box and the brakes of a car work and are playing together correctly, there is not much left to test on the completely 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 to write correct code. They are equally important to improve the code you write. At least if you embrace them and you are not just write tests for the sake of it.

Unit tests

Unit tests make sure your code is correct on the small-scale level. Thanks to unit tests, you don’t have to check manually anymore 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 force you to write better code. When writing unit tests, you realize right away if your code is good or bad. If it's hard to write a unit test, your code has some design issues and you should redesign it.

During the setup phase of the test, you have to create all the objects required. If this becomes more tedious than you would expect it to be, your data may be spread in places where it doesn't belong. This is a very strong indication that the design of your code is bad and should be reworked. When writing a test you are a user of your own code. And your code should be easy to use as we have learned in the chapter on interfaces. Thus, if your code is hard to use, it is bad.

In good code, all the relevant data is easily accessible and constructing it manually for a test case is fairly simple. Preferably you have one big object with fairly static information that you can reuse in all tests and few small, dynamic objects that are different in every test.

If you write a test you have to know the expected outcome of the function call. If you struggle for the simplest cases, chances are high your functions are too complex. They should be simplified. Rewrite the code until you can explain to your colleagues what the code actually does. Until you can write a test case. Otherwise you'll run into huge problems down a bumpy road.

You will be running the unit tests all the time. After every function you defined, after every successful compilation, after every coffee you drink and certainly every time you pull code from the repository. It gives you a constant feedback whether everything is fine or if you just broke something. 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 may not take more than a few milliseconds. You'll 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 proper 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'll really make progress.

Integration and Functional Tests

At first, it sounds great to write integration, or even functional tests. With comparably few tests you cover a fair amount of the code base. But this comes at a price.

There is much more code covered by a single integration test than by many unit tests combined. This has the advantages that a single test is much more likely to find a bug. But also has its drawbacks. The fact that integration or functional tests cover so much code makes them much slower and less precise when it comes down to locating an error. Fixing bugs found by integration or functional tests is much harder than fixing bugs found by a unit test. At the same time, integration tests are also more brittle than unit tests. The interfaces are much bigger and there is much more code underneath that can change. 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 of existance. They help improving your code as well as they force you to write proper interfaces. It is important to write these tests starting 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 prove that the user stories are really working. To prove that the software really works. Or to write a test case if a bug was found.

Testing existing code

When working with code, you'll 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 is very hard to figure out the weaknesses and the logic behind existing code. Usually, there are corner cases that are really hard to find if you don't know about them. Additionally it might be hard to set up all the tests as there are no interfaces and objects are hard to create. Writing tests for existing code can be really difficult.

Many people misunderstand the idea behind 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. Rather it’s about writing an automated documentation of what the code does at the moment. 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 a bluntly: If you have enough users, someone will certainly rely on buggy behavior of your API. Users are smart and they probably found a way to deal with your buggy code. Fixing the bugs might destroy the user code.

As you write tests, always make sure they fail when you expect them to. It already happened to me several times that some tests were unexpectedly passing without writing any code. Such checks prevented me from hours of frustrating bug fixing. The source was some build problems where I tested the wrong binary, but sometimes 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 not to test private methods as it breaks the encapsulation. Instead you were supposed to refactor the class and extract the private function into a new class on its own. With existing code you have to be a little bit more pragmatic. You can’t just take any code and refactor it as you like. This will certainly introduce bugs. Making these private methods public is indeed the only way how you can test the class. Once the class is refactored, you should clean it up and make all methods private again as it should be.

Apparently this way of dealing with private methods is a hack that shouldn’t be applied unless necessary. And it also makes apparent how important it is to write tests right away and to keep constantly refactoring before the problem goes out of control.

//more about testing existing code? See Michael Feathers book, WELC.

Asserts

There were times when people thought that using assert commands in to the production code was a good replacement for writing tests. There are also books favoring this approach [citation?]. 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 guess today everyone agrees that tests and production code should be in different files, possibly even in completely different folders.

Secondly your production code is not made to run automated test cases. Asserts are only executed if you run the software you create. It will highlight you any violations of the assertions along the way, but this is not something that can be automated. It can be used at most like something of an emergency sign. You should rather focus on writing better tests than using asserts in production code.

Don't get me wrong. There is nothing wrong with asserts in general. You just shouldn't use asserts in production code as a replacement for tests. The following two code snippets are perfectly normal code and pretty much identical:

```
def root(x):
    assert x >= 0, "smaller 0"
    return x**0.5
```

```
def root2(x):
    if x < 0:
        raise AssertionError("smaller 0")
    return x**0.5
```

This is because `assert` throws an `AssertionError` if the required condition is violated. You can even add a message to the `assert` command with the `, "smaller 0"` syntax. The only advantage of the second code snippet is that you can use custom exceptions as we'll learn in chapter [?].

Test Driven Development

So far, we wrote tests to check if our code works correctly. 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 like beforehand. 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. You have to define the interface of a class before writing its implementation. With TDD you decouple the code as your tests force you to.

In software development it may happen frequently that you have some model in mind that is supposed 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’ll never need this complex structure. Instead you can write test cases for what you really need and make sure these test cases all pass. Everything else you can take care of later on, once you know it’s really needed. On a code level, YAGNI can be prevented by writing the tests first. If you don't need a piece of code to make the tests pass, just don't write it. Even if you really think that it would be important, beautiful and possibly even fun to write this piece of code. It's not needed now and chances are it will never be needed at all.

Maybe you do not fully understand yet how TDD is really going to work. Don’t worry. You should maybe first get some experience with normal tests. At least if you don’t immediately see how a test should look like. Or if you don’t know how the final interface of the code will look like. Yes, there are several things about TDD that seem a little odd and it takes time getting used to it. But it is worth the effort. Keep trying it once in a while and start using TDD more and more often.

How TDD works

You write one test for the feature you want to implement or the bug you want to fix. I repeat: one and only one test. In case you have functional and unit tests (I hope so), you may have one open test case for each of them. There should be test case which fails at the moment. If both, unit and functional tests pass, you can take a day off.

Just kidding. If a test passes for unknown reason this is a serious issue that you have to investigate. Maybe a feature is already implemented, maybe your test is not testing what it should and has to be refined.

Otherwise you start implementing. Figure out why the test fails. For new features it's usually obvious. What the test is testing is simply not yet implemented. Now it's your task to write just enough code such that the test passes. No less and no more. You don't have to write great code in this step. Just make sure you find a good enough solution such that the test passes.

Once the test passes you might have to refactor the code a little to get back into shape. You already wrote all the required test cases as a safety net. 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 how you write code in TDD.

1. Write a failing test.
2. Write code until the test passes.
3. Refactor if needed.

These three steps you have to repeat over and over again until you are done with your ticket.

Also, with TDD you have to do some bigger refactoring once in a while. This is inevitable and has to be taken into consideration. These refactorings involve complete components and you'll have to deal with several classes at once.

Importance of TDD

As we learned in the chapter on interfaces, they should always be defined from the user perspective. With TDD you are taking the user 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 is adapting to the user and therefore the interface of the code becomes more user friendly.

Example of TDD

TDD is best understood by looking at a short example. Let's write a program that converts Arabic numbers (the one we use) into roman numbers. As we learned just now, we start by writing a first test case.

```
# 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 expected. But we can make it pass easily.

```
# inside roman_numbers.py
def roman_numbers(_):
    return "I"
```

This code 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.

```
def test_two():
    assert roman_numbers(2) == "II"
```

```
def roman_numbers(n):
    if n == 1:
        return "I"
    else:
        return "II"
```

The code from the first test is not sufficient anymore. We have to use at least some `if/else` clause. You might feel the urge to refactor this code. But at least for the time being we leave it as is. The need to refactor this code is not yet strong enough.

There is a rule of thumb saying that a one time repetition of the code is fine and does not immediately have to be refactored out. Only if the same code is repeated three times or more, it should be refactored as you might understand the problem somewhat better. However this rule is contradicting the DRY principle to some degree. As you can see, we have hardly any strict rules in software engineering. It is always a trade off between different principles.

```
def test_three():
    assert roman_numbers(3) == "III"
```

```
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:

```
def roman_numbers(n):
    return n*"I"
```

Let's add a fourth test:

```
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.

```
def roman_numbers(n):
    if n == 4:
        return "IV"
    return n*"I"
```

For 5 we can just continue with the same pattern.

```
def test_five():
    assert roman_numbers(5) == "V"
```

```
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:

```
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.

```
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 bigger numbers can be done by prepending them to the `arabic_to_roman` dict. Note that I used a dict instead of a list of lists. This is because as I mentioned in chapter [chapter Data types] that list elements should all 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 dict is a little bit fragile. It is only guaranteed to work for Python versions >= 3.7 as only since then, dicts are guaranteed to keep their order. The following solution would probably be the best as it is more solid, even though it is a bit longer:

```
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 reamining tests and implementations are straight forward. I'll leave them as an exercise for the reader.

Stubs, Fakes and Mocks

// chapter 12 Types of tests, Mock and Stubs has some redundancy with this chapter. Maybe merge them?

There are many cases where you have to write a test but the code you want to test contains something you don’t want to test. Like a database or an internet connection. You want to have a fake database that returns the value you expect and will never fail. The solution is writing a database on your own. Not a complete one. One that does only what you really need for this test case. It implements every function you call and returns some values that you want. You may have to implement quite some logic into the fake database to implement the desired behavior, depending on how complex your test cases should be. Maybe you need different fake

databases for different tests. You might need a dedicated database that throws an exception in some special case. All together programming such fake objects is a lot of work and it makes the code rigid as not all the functionality 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 one is to use an existing database and change some of its behavior using a mocking framework. In the following example we mock the result when reading a csv file. In Python this is easily done using the Mock library. Most other programming languages have similar mocking libraries as well.

```
from important_stuff import read_csv

from unittest.mock import Mock

def read_csv(file_name):
    # ...
    return # ...

def test_mock_important_stuff():
    # Override the `read_csv` function defined in important_stuff.py and return
some values.
    read_csv = Mock(return_value=([7], [8], [9]))
    assert read_csv("unexisting_file.csv") == ([7], [8], [9])
```

This test passes, even if the file passed as an arugment is not existing. 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. This is contrary to fakes, which have an implemented behavior that mimics the real behavior to some degree. Mocks are much easier to set up than fakes.

Faking

[clean craftsmanship, p.118]

A fake is a working version of the object you want to replace, albeit it's a simplified one. For example your fake csv reader in the example below does not read a file from the disk, but just returns a string stored in the code. For running tests this is usually good enough without having the drawback of dealing with the file system, where your original data could easily be deleted or messed with by anyone else.

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` has clearly not a complete implementation of the `CSVReader`. It has just enough to save some data and read it again. 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 clearly 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 returning an API key. Now let's first look at the code without DI [https://Python-dependency-injector.ets-labs.org/introduction/di_in_Python.html].

```
import os

class ApiClient:
    def __init__(self):
        self.api_key = os.getenv("API_KEY")  # <-- dependency

def main():
    client = ApiClient()

if __name__ == "__main__":
    main()
```

Here the API key is generated inside the `ApiClient` class. This is bad for several reasons. First of all, it is hard to test. You cannot easily replace the API key with a fake one. Secondly, it is hard to reuse. If you want to use the same API key in another class, you have to copy the code.

Let's say you want to change the `api_key` for testing purpose. One thing you can do is the following selection:

```
import os

class ApiClient:
    def __init__(self, selection):
        if selection == "production":
            self.api_key = os.getenv("API_KEY")
        else:
            self.api_key = "1234"

def main(selection):
    client = ApiClient(selection)

if __name__ == "__main__":
    main("production")
```

But this is bad practice. As we'll learn in the chapter on strings [chapter Data types], such selections should not be delayed. Passing around strings is bad practice. And even if you replace the string with an `enum`, you should still avoid delaying this choice.

A better solution is using DI. Here the API key is generated outside the `ApiClient` class. It is passed to it in the constructor as a function argument. This makes it easy to replace the API key with a fake one. It also makes it easy to reuse the API key in other classes.

```
import os

class ApiClient:
    def __init__(self, api_key):
        self.api_key = api_key # <-- dependency is injected

def main(client):
    # ...

if __name__ == "__main__":
    main(
        api_client=ApiClient(
            api_key=os.getenv("API_KEY"), # <-- here you can change the api_key
        )
    )
```

Apparently, you can select the `api_key` inside the `main` function and pass it on as a function argument. Like this you can easily replace it with a fake one. This is very useful for testing.

Using DI is generally a very recommended practice and should always be used when dealing with IO, time, random numbers, etc. For the very simple reason that you can very easily replace the code that you inject with something else.

The only drawback of DI is that this object has to be passed through the whole stack down to 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 testing the code.
2. Pass a string or an integer through the whole stack and make a selection based on its value as done with the `"production"` 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 depending on which you'll make a selection.

It turns out that if you have to make a selection, for example because you want to change some value for testing, using DI is the best option. Delaying the decision would require you to pass around the string instead, which is bad practice. Switch case selections should always be resolved as soon as possible. And DI allows you to do exactly that.

DI is very similar to the strategy design pattern. The main difference being what you want to achieve. DI is mostly used for testing, meanwhile the strategy pattern is generally used to allow the user to make a selection at run time.

As a downside for DI one has to mention that it makes the code harder to understand. You have to look through many functions to understand what's going on. For this reason it is recommended to use DI sparingly. It should be mostly used for the reasons mentioned above: IO, time, random numbers, etc.

Summary

Now don’t worry if you haven’t understood everything. I just explained very briefly dependency injection, faking, mocking etc., which are all fairly advanced topics. I just hope you got some of the basic ideas I tried to explain here. They can be useful and the ideas behind them are very important. Especially TDD and DI were really important topics in this chapter. If you want to have an in depth look at some of the things we discussed here, I recommend looking at these two topics. There is plenty of literature out there going into much more detail than I did 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. You can also pass different function objects as function arguments in those programming languages supporting function pointers or duck typing. This has the advantage that you don’t have to deal with base classes and so on. It’s just not used that often because usually you want to inject complicated objects and function pointers can only be used for simple objects. Though generally I would recommend using dependency injection with class object instead of injecting function objects. Simply because it can be used in all major programming languages the same way and you don’t have to learn anything new.

So far for the technical implementation and the introduction to mocking and faking. But the real problem is only to come once again. The question is how and what to test. Apparently, it’s no solution to write a complete database simulation every time it is needed. This is not only a hell lot of work. It also makes the code rigid.

Copilot

Copilot knows about dependency injection as well as seen in the Copilot example in the chapter on functions [chapter functions]. The difficult part is rather how to make Copilot use DI. Though when implementing a normal reader as well as a function and defining the `reader_type`, Copilot understands that it should use

Dependency Injection in the last line of the code here. I once again had to write only very little code to make Copilot understand what I want to do. I had to define the `class` lines as well as the `reader_type = "mock"` line and copilot did the rest.

```
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

// Quote from uncle bob?

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
3. Liskov substitution principle
4. Interface segregation principle
5. Dependency Inversion principle

These 5 very 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 how these principles should be applied nor what they mean exactly. In my opinion, these principles hold for compiled languages as Java and C++. For Python users only the SRP is really important, 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

// OCP is always a tradeoff what it is closed and open for. Adding more types or more functionality? [https://youtu.be/fwXaRH5ffJM]

The Open Closed Principle (OCP) was first mentioned by Bertrand Meyer in 1988. It says that an object should be open for extension and closed for modification. The original version states that one should use 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 as many implementations at comparably low cost, while it is fairly expensive to change the interface itself. Each class implementing that interface would have to be changed as well.

Our code should be stable with respect to extensions later on, but not to changes. If the requirements change, we have to change our code as well. This is inevitable. But we shouldn't have to change our code if someone else wants to change his code. Thus the solution is to use abstractions at potential abstraction points. This allows the user of our code to extend it without us having to change anything in our code.

Let's make a small example. We have a class containing some postal codes of Swiss cities. If we want to add an additional city, we'd have to add an additional function to this class. The class `City` is not closed for modification. We have to modify it every time we add another city. This class is not obeying 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 wants to add another city, we have to do this within our own code inside the class `Cities`. We have to *modify* the class `Cities`. This is the opposite of what the OCP want to achieve. The OCP want to separate the user code from the interface.

Instead we can create an interface `City` and implement it for every city we are interested in. We are free to add an additional city if we want 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 obeys 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, he can create as many additional cities as he wants and we don't have to care about it. The base class `City` defines the interface and that's enough for us to work with any class the user adds.

The example mentioned here is the classical example for the OCP. It is the strategy design pattern [Design Patterns]. However, you might also use the decorator pattern which fulfills the OCP as well.

Liskov Substitution Principle

// see https://youtu.be/pTB30aXS77U

"If it looks like a duck, it quacks like a duck, and it needs batteries, you probably have the wrong abstraction." - wisdom of the internet

Implementation of interfaces shouldn't blindly follow 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 the situation where you don't know what the payment interface should take as an input argument.

```
class Payment:
    def make_payment(amount, card_number_or_email_address)
```

This logical contradiction about what the second argument should be (email address or card number) is a violation of the Liskov substitution principle. Credit card payments and Paypal payments should not implement the same interface.

Instead the selection of the credit card numer or the email address should be done later on, inside the specific classes. It is there that these credentials have to be asked 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

class CreditCard(PaymentSystem):
    def make_payment(amount):
        # ask the user for 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 like the SRP for interfaces. Interfaces should be split up into many small parts. This is important in order to keep the coupling low. You don't want to import and compile a huge library only because you need one small feature of it. If there are some logical blocks within a library that are separate, make sure that they are made available separately.

Here the file A does not follow the ISP. It does 2 independent things. Most other code needs only one of these functions. They are independent. Thus, they should be in different files.

Now in Python this is not such a big deal as you can import each function individually and even if you import the whole file A it's not a big deal. It's not becomming slow. In C++ on the other hand, adding unrelated functions into the same file is really a no no. In C++ you always include a whole header file at once and you'll have to compile everything that comes with it. There might be a hefty price to pay if the file A would be too big.

```
// 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 do this, such that most of the other files use only one of the newly created files A1 and A2. A1 and A2 should have high cohesion within themselves but there should be low coupling between them. Idealy, the amount of code that you'll have to import is reduced roughly by half.

This process of breaking up files can be repeated until it is no longer possible to reduce the amount of code imported, or the number of imports would be growing unreasonably fast. At this point you finished the segregation of the file A.

```
// C++
// file A1
int function_1(){
    return 1;
}

// file A2
int function_2(){
    return 2;
}
```

A well known example where the interface is segregated 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 whole standard library only because you need some part of it would increase the compilation times too much.

Another common example is defining an enum inside a class, while other parts of the code might need access to this enum as well. This other part of the code has to import the complete class containing the enum, even though it doesn't care about anything else than this simple enum. This other code imports much more code than it would have to. And the solution is pretty simple. One can just extract the enum from the class and have it stand alone. Then it fulfills the ISP.

```
# 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 hidden inside of the class definition. This could be avoided by moving the enum outside the class. This would reduce the coupling of the code as the user code depended only on the enum and not on the whole class around it.

The code would be much better as follows:

```
# 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
```

Like this you 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" - ?

Dependency Inversion Principle (DIP) is a technique used in languages as C++ and Java to reduce the compilation times considerably. The files in your project include each other and form a tree. The so-called dependency tree. The main function is at its root. The leaves of the tree are low level functions of 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 so important. This is mainly a technique to break compilation dependencies which don't exist in interpreted languages. Though it's still good to know this principle as a Python user as it is very fundamental.

The first time you compile your code, the whole code base (the whole dependency tree) has to be compiled. This can easily take minutes, maybe even hours. The resulting binary files carry a time stamp. If you recompile

your code later on, only the files that changed since the last compilation have to be recompiled. For small changes, this reduces the time required for compilation to 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 huge parts of the code to recompile. For everyone working on the project. This is why software developers have so much time to spend 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, main changes as well as main directly or indirectly includes all other files of 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. Such that main won't be affected by internal changes of the code within a library. If I change a file in a library, say the `sin` function inside the math library, I want to recompile only the math library itself. I want to cut off this library branch from the dependency tree and deal with it independently. Main shouldn't know about anything going on within the math librarry. 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 couples it loosely by the interface of the branch. You can do that by defining an abstract base class (interface in Java) that defines the shape of the interface. The file containing this interface doesn't have any dependencies. It's on the lowest level of the dependency tree. Or at least in something like a local minimum. The old interface code of the library inherits from this interface. It implements it. As main uses this library, at first is has only the information of the interface. Everything else is hidden as it's not included. Unless you change the interface, changing code inside the library will not cause anything else to recompile.

Example

As I said before, the DIP is important for compiled langagues. Therefore the following code example is written in C++.

Let's say we have a class `Nothing` with a method `do_nothing`. Now we want to change the code such that `main` won't recompile if the implementation of `do_nothing` changes.

```
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:
    void do_nothing() override;
};

// inside Nothing.cpp
#include <iostream>
#include "Nothing.hpp"

Nothing::do_nothing(){
    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 change `main`. Therefore, `main` does not need to be recompiled if `Nothing` changes! `main` and `Nothing` are only connected together by the linker. The linker will make sure the main function calls the correct implementation of this library.

// dependency tree graphs

Summary

I think this was the longest section in this book where I explain technical details for C++ that Python users don't necessarily need. At the same time, I'd like to emphasize that this section was very important for the C++ and Java programmers. Both, for the quality of the code, and also for understanding how the whole concepts of includes, compiler and linker work.

22. Software Engineering principles

In this chapter I explain some very general design principles that I saw on youtube, [https://youtu.be/XQzEo1qag4A] published by the channel "Tech with Tim". I really liked these very general priciples 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 at once. It's too difficult. But what you might be able to do is breaking out small pieces of this problem and solve those. This is generally how software is designed. Break the problem into small pieces and assemble them back together. A common example is the Fast Fourier Transfor (FFT) or the merge sort. Usually a divide and conquer algorithm is applied if a problem scales with O(N^2) or worse, but it can be subdivided into smaller problems. Divide and Conquer algorithms usually scale with O(N log N), which is generally acceptable. Furthermore Divide and Conquer algorithms can easily be parallelized, which may significantly boost the overall performance.

Increase Cohesion

Cohesion very much realtes to the section [Correlation] that we already discussed in the chapter on Physical laws of code. Similar things that possibly depend on each othr should belong together. Mathematical functions are stored together in the math library and IO functions are in the IO library. This makes sense as it simplyfies searching for some other functions that you might be looking for. Mixing these two libraries would only cause confusion as it would make it hard to find what you are looking for.

Reduce coupling

Reducing coupling is an important topic in classes [chapter classes]. Make sure that the classes are as independent as possible. You don't want your math library to be depending on the filesystem library. Even if it might make sense from a developer point of view (even though here it would be hard to explain), you should minimize the number of dependencies as far as possible. Only import other libraries for cases where it is absolutely inevitable.

The same holds not only for libraries, but also for all other code that you write. Try to keep them all as indipendent as possible. Make sure that you properly structure the levels of abstractions in your code. Low level code should not depend on high level code, etc. 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 leaving away unnecessary details and instead focusing on the absolute necessities. You have to design interfaces that are slim and very generic. For example take (once again) a car. You want 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 such that the outside does not interact with it.

If you didn't abstract all the details away, you may be left with several different functions how to take care of all the different engines because you have to take care of all the paritcularities. On the other hand, if you abstracted them all away, you can replace the engine with another one if you please.

Increase reusability

Reuseability comes hand in hand with inreased abstraction. Leaving away all the details of an object reduces it to a fundamental building block that can be reused more easily. Because general objects are more likely to fit as a building block than something very specific. Just take for example all ot the "standard" libraries. They all do exactly one fundamental thing: they deal with the filesystem, do mathematical operations or create

random numbers. Of course it would sometimes make sense to combine these, but that would probably not the reusable code anymore because it is too specific. You should only write such specific code if you really have to.

Design for flexibility

Your code will change. It's inevitable. So start living with this fact. Requirements will change and you'd better make sure 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 change your code. You have to follow the best practice in order to prevent your code becoming solid as a rock. You want your code to be fluffy and easy to change.

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 fuzz. You won't have to change code all over the place. It's more like a chirurchical operation where you change only one thing.

Anticipate Obsolesence

Code you use will become obsolete. Version change, bugs and security issues are not fixed, licence fees are getting too high, you name it. There are plenty of reasons why you have to adapt and change third party libraries or at least adapt to new syntax. So you better anticipate that you'll have to replace some library by adding an adapter between that library and your code. This will simplify reacting to changes. You can then just write an adapter for the new library and you don't have to change all the existing code.

You have to anticipate obsolence by keeping your code flexible and reusable. The database code should not be spread throughout the whole code base. This would be the very opposite of what we want. It would take enormous efforts to replace it. Instead you should be able to replace it easily.

There are cases where you might think, you'd never have to place a piece of code. How wrong you are. No matter how important some library might seem to you, at some time you'll have to replace it. There are so many companies out there who wrote 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 Oracle database code is spread all over the code base.

Design for Testability

I think this is the most obvious point in this chapter. I explained plenty of times that you have to write tests. Because if it's easy to write tests for your code it's also easy to write code using the interfaces of your code. 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 where writing tests for 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 things that should be solved with DI. Inject a mock file or a fake timestamp to the function and your tests will be much more stable.

Part 5: Programming

23. Programming Paradigms

// move this closer to programming languages

There are several different programming paradigms. For several decades Object Oriented (OO) programming was the way to go. But it turned out that OO programming has its own problems as well. As I already mentioned several times, it is our goal to write code that is easy to understand. It is not our goal to write OO code at all costs. Procedural or Functional programming are equally valid programming paradigms. Nowadays there are also multi paradigm programming languages like Python, or even the good old C++, where you can mix these 3 different programming paradigms.

Here is a very short list, of what the different programming paradigms offer:

- OO programming: classes, mutable variabels
- Procedural programming: mutable variables
- Functional programming: nothing. Yes, nothing.

Functional programming is basically a subset of procedural programming, which in turn is basically a subset of OO programming. But this doesn't mean that functional or procedural programming are necessarily worse than OO programming. Limiting the number of possibilities can make the code easier to understand. For example the fact that functional programming has no mutable variable excludes a lot of possibilities that you have to consider while reading code procedural or OO code.

Object Oriented programming

Object Oriented (OO) programming started in the 70ies. It peaked with the still very wide spread languages C++ and Java. Somehow the whole 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 buzz words and the crowd does the rest. Already in times before social media. The only explanation I have is that the software engineers were all sitting 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 makes things easier. But it did not save the world. And a lot of things that were developed along with OO programming are outright garbage. Without the hype around OO programming, these things would never have been able to get widespread usage. People stopped thinking critically and just started using all kind of OO features that turned out to lead to terrible code.

Don’t use any other OO feature than plain classes and abstract base classes or interfaces. Don't forget to make everything private that should be. And 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, for example structs in C. In procedural programming, functions are more important than data types. Though, contrary to functional programming, you are allowed to have mutable variables and output arguments. This

simplifies writing code at times but the code created this way is harder to understand due to the additional complexity.

Having only structs in C, compared to classes in C++ apparently has some drawbacks. Apparently taking all classes and converting them into structs would be perfectly viable, but making everything public that was hidden before is not a good coding practice. Instead you have to adapt to a different coding style. You have to find a way around classes with private members. As we have seen in the chapter on classes, there are three fundamental types of classes:

- structs
- delegating classes
- worker classes Structs also exist in C. Delegating classes can be replaced with structs. The only thing that requires some more thinking 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.

It takes some adaptation to get used to procedural programming, but it certainly has its advantages and is worth the effort.

Functional programming

The main difference of functional programming to procedural programming is the fact that there are no mutable variables. Datastructures can't be changed once initialized. This is a very strong restriction to the programmer and makes programming more difficult. On the other hand, it has also its advantages. You don't have to pay attention on things like mutable variables. Functions don't have side effects. Functions ~can't~ have side effects. The only thing that they change is the return value. Furthermore the return value of the functions only depend 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 where ever possible. Also in OO programming. Pure functions are much easier to deal with than functions depending on some kind of state. They are also much easier to test. You will never have the issue that you don't know how to test a pure function. Call the function with some different arguments and check that the return value is correct. It won't get any easier than that.

Due to the pure functions, functional code is generally easier to understand than, say, procedural code. This is because functional programming depends only on states that are always apparent and easy to understand.

One of the drawbacks is that functional programming doesn't allow the usage of for loops as they generally require a mutable counter. Instead you have to use recursion which can be a little tedious at times. Furthermore it requires you to use a programming language supporting tail call optimization. This is a technique that allows the compiler to optimize the 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 mimic functional programming to some degree by using only pure functions. And I highly recommend doing so as you'll be left with only pure functions that makes testing them very easy. Using only functional programming, however, is in my opinion not worth it. It is too restrictive and makes programming unnecessarily difficult.

Conclusions

All different programming paradigms have its 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 all different paradigms in order to be flexible and adapt your style to the current needs. The times where you were supposed to write only procedural or OO code are over. If you are flexible and you know the advantages and drawbacks of different programming paradigms, you can write better code.

I would generally recommend staying on the lowest programming level (functional programming) as long as possible. Only switch to higher levels (procedural or even OO programming) if it would otherwise become too complicated to understand otherwise.

Copilot

Here is an example where Copilot went haywire. I wanted it to create code like `a = [i for i in range(10)]`, but somehow this didn't work out. The output is certainly wrong.

```
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/RDd71IUlgpg]

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 frequent question from 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 of the Object Oriented languages are similar enough and the differences in the programming philosophies are fairly small. Small enough to understand my programming examples in this book, I hope.

For example, a lot of the low level C++ features for instance can be wrapped into a higher level object, making it look like an intermediate level language. Though it's still not quite such a high level language as Python.

I really want to emphasize that you shouldn't learn a programming language in too much detail. 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 in the internet as you need some specific syntax along the way. Google and Stackoverflow are a better help than your vague three-year-old memory and Copilot is also becomming a good help. 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, meanwhile concepts you have to be understood.

But as you asked for a programming language, I would briefly like to give my point of view. Though it is highly biased. I know mainly C++ and Python, and a little bit about Java and JavaScript due to the programming books that I read. If you work in a field where one specific programming language is used, you should certainly learn that one. Even if it's just Matlab. You can still learn some other langague later on.

As I am a scientist, I would recommend Python as a first programming language. Javascript is a viable alternative if you do web development. They are both scripting languages that don't need a compiler and are fairly easy to get started. As they use dynamic typing, you don't need inheritance to define an interface. Any two objects that have the same interface can be exchanged in the code. And there is no need to learn anything about pointers or memory allocation like in the old days. These things are outdated, as explained in the chapter [levels of abstraction].

Though it has to be said that dynamic typing has also its drawbacks. Having type information is not only a help for the compiler but also for the programmer. It is easier to understand what a function does if you know the types of the arguments. For instance, there are ways to add type hints in Python, but I'm frequently too lazy to add them.

I would not recommend learning Java or C++ as a first programming language, even if I use some C++ code in this book. C++ and Java are too complicated to learn them as an introductory language and it takes much more time understanding the language itself. The C++ examples throughout this book are only to explain low level details that you don't have to care about in Python. Instead you should learn how to apply the higher level principles taught in this book and elsewhere to improve your coding skills. Of course, later on in your career it makes sense to learn many more languages. Java and C++ are still among the most widely used. Not because these languages are better, but simply because there are so many old projects around.

C++ and Java are both staticly typed. They have to be compiled and use inheritance to define interfaces. And you have to deal with pointers. Learning new languages will show you other ways of thinking about some problems. Changing from Python to C++ you'll have to learn quite some basics of software development. It opens up more job opportunities as well. But it's nothing worth bothering with when you just start programming.

Existing programming languages

Programming languages and APIs share the same fate. It would be easy to create a new programming language that is clearly better than an existing one. Someone said that you could remove the C++ template specialization of `std::vector<bool>` and you had a better programming language (it is optimized and treated as a bitwise array which makes it annoying to work with). And he is certainly right. But there are

millions of software projects that already use the current languages and they depend on the current functionality. Their code is worth billions. You cannot update such quantities of code only because of such a small nuisance in the programming language. Instead, there are thousands of developers making suggestions how the current programming languages could be improved without breaking compatibility. A team of experts will debate about all kind of possible issues before a new feature or internal change will be accepted into the standard of a programming language.

For example: In C++ there is the boost library. Pretty much everyone programming C++ knows it. It is one of the most commonly used third party libraries and known for its high quality standard. 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 once 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 the smart pointers and the filesystem library made their way 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 some code examples in this book. Most concepts that I explain here can be explained with real world examples. The syntax I used is held as easy as possible as I want to teach you concepts, not syntax. I tried to make this book as agnostic to programming languages as possible. The code examples are mostly written in Python and sometimes in C++, if needed to explain some low level feature. 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 such that you can roughly understand them, even if you don’t know the according programming language too well. I promise that the syntax will be very simple to understand. It requires only the very basics of the corresponding programming language.

Python

Even though Python is a fairly easy programming language to learn, there are some things that are some language specific things 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. You don't have to write the types and a function can be called by many different argument types. But it also comes along with its drawbacks. Types are an important part of the information of 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 does something quite different 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's not Pythonic, it is possible to write type hints. A simple `: str` following a function argument to indicate that it should be a string.

Here is an example using type hints:

```
def digits_of(number: str) -> list[int]:
    return [int(d) for d in number]
```

But as I said, this is not enforcing that the argument of `digits_of` is a string. You could also pass a list of floats instead and have 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 it makes the code more readable. Even if I'm sometimes too lazy to do it myself. And even if it moves the syntax a fair amount closer to C++. C++ is not such a bad programming language after all. It's just a little bit old fashioned.

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 as such:

```
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 this member variable `hi` to an existing class instance wouldn't be possible in hardly any other language. And this for good reasons. It might seem tempting to add such 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. Rather, 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 fixes the available member variables. In this case, there are only the variables `price` and `weight` allowed. (Accidentally) adding other member variables to the `Apple` class is not possible anymore.

Abstract Base Classes and Protocols

I still recommend using abstract base classes (ABC), alhough it is not required in Python. It makes the code slightly more readable as it defines the structure of the interface you are going to use and implement. And it also prevents you from making mistakes that might be nasty to track down.

An alternative to ABCs are Protocols, which were introduced in Python 3.8. They are mostly equivalent, though Protocols have some advantage when working with type hints. Though this is a highly advanced topic and I cannot go into details here.

C++

C++ has some particularities like a preprocessor, header files, pointers and arrays that make the language somewhat special. Thus I'd like to explain some of the things where C++ tics while other programming languages toc.

C++ has been developed by Bjarne Stroustrup and published in the 80ies. He took the existing C programming language and added object orientation to it, along with some other things. So yes, it is an old language, but it is still around and will accompany us for several more decades. Thanks to the constant development of the language, it has overcome many of the ancient problems that it brought along. At the same time, C++ is a very good example to learn a lot about programming languages and how they evolved. As I used C++ in some of the examples here, I'm going to explain here 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 the array, as explained in the chapter on levels of abstraction. There is no more reason to use arrays in C++.

There are libraries that require plain old arrays instead of vectors. This, however, is no reason to use arrays throughout your code. Instead you can use vectors as usual and convert them to arrays using the `data()` and `size()` function as needed.

```
std::vector<int> vec {1,2,3,4};
some_old_C_style_library(vec.data(), vec.size());
```

Again, this allows you to deal with vectors as long as possible and to convert them only at the very end.

Smart pointers

Smart pointers, `std::unique_ptr`, `std::shared_ptr` and `std::weak_ptr`, are the replacement for the plain old pointers. Smart pointers are a higher-level implementation. It has things built in like reference counting and they know when to go out of scope. There are still some things to know like weak pointers, but these are mostly details that you don't have to care about in the beginning.

There are libraries that require plain old pointers as function arguments. This is no reason to use plain old pointers throughout all your code. Instead you can convert the smart pointer into a pointer using the `get()` function.

```
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.

Pass by reference

In order for an object to be mutable, it can be either 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 basically pass the object itself and thus it can 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 lot of memory.

Passing an object by reference or by const reference is an important difference. Passing an object by const reference means that it is not going to be changed by the function call. In fact, the compiler will lock this object and it won't be possible to change it.

At the same time, this is also one point for critisism as passing by const reference should have been the default. The compiler won't complain if you forgot a `const` even though you should have used it. It would be much safer to use the programming language if `const` was the default value and you had to specify an argument `mutable`. This would cause a compiler error if you changed this argument. 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, etc. Probably it became so wide spread because most things worked out pretty well, except some details about multiple inheritance [https://www.geeksforgeeks.org/multiple-inheritance-in-c/]. But as I told 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, in Java for instance. In Java, defining an interface is actually called this way, while in C++ or Python one has to define an "abstract base class" (though in Python it is not necessary). This is the only kind of inheritance that I recommend using. Remember when I say you shouldn’t use inheritance: the whole thing with abstract base classes should be named differently and is not affected 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]. They are classes where all members are public. In general, structs are used to store different data types, though in theory structs may also contain functions. The later is only forbidden by general agreement.

Structs are generally very useful objects, as explained in the section on classes. It’s a pity 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 nor setter functions. Or since Java 14, one can use a record which is roughly the same as a struct.

Copilot

Copilot can be used to translate between different programming languages. Here is a very simple example to show the capabilities of Copilot. Though I don't know how difficult code snippets Copilot can translate.

```cpp
#include <iostream>

for (int i = 0; i < 10; i++) {
    std::cout << i << std::endl;
}
```

```python
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 says that disorder is always going to increase. Fighting entropy is a lot of work. It is like you cleaning up your room every week. If you don’t do it, your room will become dirty and you don’t find your stuff anymore.

In software engineering we have a very similar phenomenon and it has very severe consequences. As we write code, there is more and more disorder 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 stuff that you don’t really need or is duplicated. This will take time and effort. But such is life. You don’t get a well payed job in IT without doing the dirty part as well. What you have to do is explained in the chapter on refactoring.

Correlation

Similar things belong together. It sounds fairly trivial and 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 kind of store is in its own area. If you find a market store selling apples, chances are high that the next store sells apples as

well. It is just normal that similar things align together. This makes them easier to find.

The same holds true for code. Functions are bundled together by their functionality, as are classes. This makes them easier to find if you search for some specific functionality. At the same time, they should also have the same level of abstraction. The main function, for example, consists only of a few high-level function calls. No string manipulations or other low-level stuff. These low-level functions are buried somewhere in a deeper level of abstraction.

Also bugs tend to cluster inside your code. Did you find a bug in some very complicated 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 so much tidier. It will feel more natural and it doesn’t need too much work to make it better. And you will find your bugs faster as you check the complex parts of the code earlier on.

Quality

There were studies what must happen that an area starts to decay [The pragmatic programmer]. They came to the remarkable conclusion that one broken window is sufficient sign for other people to start breaking windows as well and within no time, the whole area looks ruined and abandoned.

Accordingly, when writing code, it is important to keep the quality high. Don't write bad code or it will feel abandoned as well. Others may start to become careless as they don't feel like keeping it in shape is not worth it and start writing bad code.

On the other end of the quality spectrum you have the issue that some developers just keep on writing and improving their code for all eternity. This is of course also an issue. There is always something that you feel like 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 for opposing rules. It is your task to find the right balance between them, as it is in many things I teach throughout this book.

Requirements

// Delete this once the Requirements Engineering chapter is done (?)

When buying a new car, you probably make a list of requirements. It needs to have five seats, ample of space for lugage, an AC, etc. Then you go to a car dealer and he'll show you a car that meets these requirements. Would you buy that car right away? Or do you first test drive it and look at some other cars as well? Even though chances are that you'll get back to that first car you've had a look at.

In software engineering we have a similar phenomenon. You have a problem and you're looking for a solution. There are several requirements and once you found a solution that meets all of them, you are happy and implement it. This would be the equivalent of buying the first car that meets all the requirements. Why would you change your behavior depending whether you buy a car or if you write a piece of code? The costs of your decision are comparable. Actually it might be even bigger in software engineering because you'll have to stick to this decision for a long time and live with its consequences.

Perhaps we should take more care when making software decisions just as we do when buying a new car. It may pay off on the long term.

26. Bugs, Errors, Exceptions

"If you don’t handle exceptions, we shut your application down. That dramatically increases the reliability of the system." — Anders Hejlsberg

"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 no problem at all, while others can be absolutely deadly. Literally. Problems are less critical if you find them early on and they are immediately recognizable. If your compiler finds an error the cost are barely worth mentioning. Triage the source of it an fix it. However if your software is already in production, the costs are significant.

I would briefly like to go through the different cases.

Syntax Errors

Syntax errors happens 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 this state. Fix it and try to improve your knowledge on the programming language you use. Syntax errors are the best example how problems don't cause any harm if they are caught early on. In compiled languages, the compiler will find the syntax error, in Python the parser checks the correctness of the syntax. Either way, you'll get an error message right away.

At the beginning of our programming careers, we were all bothered by the compiler errors (back in the days I was programming C++). We were happy once there were no more errors. Our programming skills were just too low to realize that the compiler was helping us on our way to write a functioning program. It is a good thing we got compiler errors because this might have saved us from creating serious bugs that could have taken ages to find. Never the less we still created many such bugs.

Bugs

A lot of people underestimate the problem of bugs. They are easy to ignore because they don’t show up too often and maybe they are not too bad. They are just some glitches. But this is exactly why bugs are so catastrophic. You don’t necessarily know something went wrong. You might have an idea something is off, but you are not sure. Or you don’t know at all. This is the absolute worst case that can happen in your code. You think everything is alright but in fact, it is not. Your hard disk got deleted, a bank lost several millions, an airplane crashed. Everything is possible and it all already hapened. Bugs are the absolutely worst thing that can happen to your code. Sure, most bugs are not that terrible. But don’t take them lightly.

Cost of Bugs

The cost of bugs is gigantic. It may take hours, if not days to track down a bug. And in bad code it's frequently not clear how it should be fixed. Furthermore the cost of bugs increases exponentially over time. This is due to the growth and the additional complexity of the code [SE at google, p.207]. This is why syntax errors are so cheap: you are forced to fix them imediately. However, you should not let the bugs linger around. The more

you wait, the more expensive it gets. In a shipped product, a bug may cost millions, while fixing it in the development phase it may cost only a few hundreds.

I hope you got the memo. You always have to make sure you don’t create bugs. Write good code and make sure it’s well covered by tests. This is the only way to keep the number of bugs low (though it will never reach 0) and stay as far away as possible from the exponential growth of the cost 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 gotten used to the faulty behavior of your software and implemented a work around. So fixing the bug would in fact introduce new bugs in the code of your clients.

Bugs can be clasified by their severity. A bug resulting in a crash of an air plane is pretty much the worst case and has to be fixed, no matter how unlikely it is to happen. Meanwhile if you write an Android game, a bug that causes the game to crash every onethousand hours or so may be acceptable. The possible nuisance of the user may not be worth the effort to fix the bug. It is very common that only critical bugs get fixed. All other bugs may get documented along with a work around, but they will not be fixed due to economic reasons. Only in safety critical systems, like airplanes, all known bugs have to be fixed.

Bug Reports

Depining on the software, writing good bug reports may be anything from straight forward to nearly impossible.

Good bug reports explain the problem in a very unambiguous way. They follow the 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 as it is very simple to reproduce the bug. Just hand over all the files involved.

If it is a game (that doesn't have an API, of course) that crashes under very specific circumstances, it may be extremely hard 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 hard. A bug report should be written with scientific accuracy, such that anyone can reproduce the bug. Anything that may cause the bug has to be reported, including things like the version number of the software and possibly 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 bad. 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 break points and inspect variables. So far so good. But if you use the debugger too often, it is a clear indication that your code quality is bad. If

you had better structured your code and a better test coverage to start with, you probably wouldn't have to use a debugger. Having to use a debugger is a clear sign that you don't know what you are doing. Meanwhile this may happen in a while, you should make sure that using the debugger is the exception rather than the rule and otherwise rethink the way you write your code.

There are many different ways to track down bugs. Most important of all, you need to have an idea, what part of the code may have caused the bug under investigation. If you have some code example using the API of your code, you can try to simplify it while checking, wheather the bug still exists. This usually gives you a good idea 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 some option that is rarely used and you expect it to be buggy for some reason. Once you have minimized the number of API calls, there are roughly two ways to track down the bug:

1. You can set a breakpoint where the value of a variable is set. If you already broke down the code from the bug report into the smallest possible case, you shouldn't have to iterate over the break point too often in order to find the faulty behavior.
2. You can bisect the bug. You set a breakpoint somewhere in the middle of the code and 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 said, the most important thing is that you have as much information about the bug as possible. You need to have a fair idea, about what part of the code may have caused the bug. If your code is badly structured and you have no idea wheather some value returned by the debugger is correct or not, you will have a very hard time bedugging it. You'll have no choice but to guess. And guessing is an extremenly tedious process.

Unfortunately, there are some bugs that are very hard to track down. These are the bugs that are not reproducable, for example in a distributed system where a race condition [https://en.wikipedia.org/wiki/Race_condition] may occur once in a while. Using log files may help. However, in complex cases it takes enormous amounts of time to track down the bug.

Fixing a bug

As already mentioned above, you should never fix a bug that you just found. Users may rely on this faulty behavior and fixing the bug may interfere with their workarounds. Consistency may be more important than corectnes. Fixing the bug may break user code!

Once you have a bug ticket, the first thing to do is writing an automated test using the minimal API code causing the faulty behavior. This helps a lot to track down the bug and it prevents future changes of the code to reintroduce the bug. Bugs that appeared once are very likely to reappear again 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. You should not just add a random hack that solves the problem somewhere in the code. You have to find the faulty logic in your code! This is the only place where a bug can really be fixed once and for all. When looking at the code, you should have no idea that this a bug fix applied later on. It should mend into the code as if it had always been there.

Copilot

Copilot is able to find some bugs. Though I expect it to find only minor bugs, this is already a real feat. For example take 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 fix bug function. Highlight all code shown here and click "fix bug". Though as with text suggestions by Copilot, there is the question how difficult problems it can solve. The text suggestions are usually fairly simple and so are the code suggestions and probably also the bug fixes.

Exceptions

Exceptions happen in cases where the software is supposed to do something but it unexpectedly can't. Some examples are writing files if there is not enough disk space left or a division by zero occurs. Though some programming languages can return infinity. These things may happend and there are not too many things in every day programming where an exception might occur. Mostly input/output (IO) if some connection cannot be established or the hard disk is full, divisions by zero, but also logical errors in your code. Yet these cases have to be taken care of. The user has to be noticed to fix the problem.

User input should always be validated right away. 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 of no use as there might be no input value 0. It could be the result of a long calculation. The user should know which combination of variables caused this exception. Check the sensitive values and return a useful message. "Invalid input: number of 'shopping_items' cannot be 0", makes it much easier to track down the source of the problem. Check the values that are sensitive and return an appropriate error message right away. If there is some 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 occure before you start your calculation. And return a meaningful error message.

Wrapping exceptions

You don't want exceptions to leave your code. This will crash the software executing it. It is not a big deal 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 has to be able to recover from an exception. It is recommended to define your own error types. Put a try except block around the whole code to catch all your custom exceptions. Custom exceptions mean that the user did something wrong and you were expecting this faulty behavior to happen. You should give the user a meaningful error message explaining why the exception occured and what he should do to fix it.

Add another except block at the end of the program in order to catch all other kind of exceptions. These are errors you didn't foresee. Bugs. Write a different error message and kindly ask the user to contact your

support. 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 the source has to be fixed.

Raise exceptions right away if the program goes into an invalid state and return a message to the user what went wrong. It is not worth trying to deal with a semi invalid state (also known as walking wounded). This is not worth the effort as you won't be able to fix the state. Exceptions originating not from faulty user input should result in a message about the cause what is wrong.

As the C++ Core Guideline E.31 states: "Properly order your catch clauses" [C++ Core Guidelines explained]. Meaning that you should always first catch specific exceptions and then more general ones. This is because specific exceptions allow you to give the user more specific information about the problem such that the user can fix it without your help. General exceptions are an indication that you were not expecting this problem to occure and the customer is not expected to fix it on his own.

The try catch block around the main function should make sure that no exceptions leave 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 similarity to if else or switch case blocks. They are susceptible to bad code, especially to 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, usually a function call or a simple error message. Furthermore, try except blocks should be the only thing within a function. The single responsibility of this function is managing the try catch block.

One common pattern is catching and reraising 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 really helpful to the user. Instead you should define a custom exception type and print an according message when catching it. With all the information you have at the time when the exception was thrown.

Make sure your unit tests check the exceptions as well, exceptions are part of the code specification. However, in some cases it is impossible to write a unit test. For example, you should never read in a file in a unit test. Instead you should dependenct inject a file object throwing an exception. We go 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 70ies. Then Edsger Dijkstra wrote the famous paper "Go to statement considered harmful"

[https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf] 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 doesn't have exceptions and thus 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 are writing 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 covers 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 also becomes apparent as one product manager creates more than enough work for several programmers. The complexity to implement a feature is much higher than the actual complexity.

The goal of the software is to keep the complexity as low as possible. Close to the complexity of the real problem. If possible equal to the real problem. The code should mimic the real problem 1 to 1. Unfortunately, this will never happen. There is always some overhead when programming. Not only boiler plate code, but also conceptual overhead. How should you map a real problem 1 to 1 into code? How should, say, an apple ever become code? The answer is: it depends on your requirements. This is where object-oriented programming came up. It claimed to be the natural representation of things. Because you could write a class `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 to you. You have to figure it out yourself. You have to try and explain what the apple really does. Maybe even write it down. Talk to other people, experts. It takes time to build up that knowledge what is important and how everything is connected. This is a fundamental requirement for writing good code with little 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 get an idea how you can convert all this knowledge into code. Take all the objects involved and connect them in different ways. Change the order of statements and how data is passed between the objects. When done correctly, you'll end up with code that resembles very much the explanation given by the experts of the domain. The objects have the same properties, the functions do the same things and you use the same names. Your code feels like a 1 to 1 mapping of the real problem. Eric Evans called this a domain model [Domain-driven design]. Handle it with care. The domain model is very precious and you can easily destroy it by adding code that doesn't fit into the model.

Having a domain model is a great asset. It forces you to understand the problem really well and write the core of your code first. At the same time, it prevents you from getting lost in low level details.

Estimating complexity

[Does this go into the agile section? Or into Requirements Engineering?]

Estimating complexity of a task is extremely difficult. Not only from a technical point of view, but also due to pressure from management. Frequently the estimation of a feature goes as follows:

Project Manager: "Can you give me an estimate of the time necessary 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 generally extremely hard. Some developers might have an idea what has to be done, others don't. But nobody really knows exactly. And everyone is a little bit scared of that task. Nobody knows for sure how to break the complete problem down into smaller pieces. And even if, there is still some uncertainty around, which makes estimating the amount of work a very difficult task.

Probably everyone could have come up with a neat solution for solving the problem, but not with the existing code base. Instead you have to consider what you really need and what parts are already implemented in the code. This case is extremely common. Pretty much everything might have been already implemented in the code, but nobody realized it. So you reimplement the code and you are left with redundant code violating the SRP. Additionally to the time used for re-developing this code.

On the other hand, there are cases where you find a very simple solution and implementing the task takes way less time than expected. But 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 one is based on breaking down the whole topic into small pieces and adding up the effort of all the individual pieces. This requires a lot of knowledge on the specific task and there is a strong tendency to underestimate the actual amount of work involved. When breaking down a task into smaller pieces, usually a lot of subtasks are being forgotten or the total complexity is usually underestimated.

The second method to estimate the amount of work is based on a comparison with similar tasks. This is generally the more accurate approach, though there is still quite some uncertainty left. Furthermore it is not that often the case that you already completed a similar task. Therefore, estimating the amount of work required for a certain task is still quite tricky to estimate.

Single line complexity

A frequent topic is the amount of logic 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, 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 something about how code can look like.

On the other end of the spectrum are some Python programmers. It seems like adding as much logic as possible on a single line would be a sport. Very honestly, I think this is a pretty bad 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 set to 80 characters. For both, Python and C++. [https://google.github.io/styleguide/pyguide.html section 3.2], [https://google.github.io/styleguide/cppguide.html#Line_Length] 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)]
```

You'd have to do it the old way:

```
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, split up the code and don't use a single line initialization.

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 kind of black magic. This may work at times, but the code will be cursed. You can keep working on the code, but once in a while 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 is complex and we break down the complexity until we have some 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 depending 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 the overview of the code and it is hard if you have 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 can't use it at all.

These are some of the considerations that made programmers split up their code into many files. But how do you tell the computer how to build up the complete code from these files? Apparently, there are some solutions, but this is an ongoing discussion.

In C++ the whole problem becomes even worse due to the compiler requiring the header files. It is possible to compile a C++ program with the command line for a single file but it becomes unbearable for bigger projects. If you use C++, it is inevitable for you to learn a build tool, for example cmake or meson.

What all programming languages have in common are the import or include statements at the beginning of the files. Even with the build tools of C++ you still need those. They might bother you but at times they are quite handy. They are an indicator for some very bad patterns in your code.

The dependency graph

If you draw a plot with all the files as circles and how they import each other as arrows you should get a directed acyclic graph. The trunk of this graph is the file containing the main function, the highest level of abstraction. As you go up in the graph, the level of abstraction becomes lower.

// create a figure of this graph

Now the first thing to look out for in this abstraction graph are two arrows pointing in opposite directions. This means that two files import each other. Depending on the language this may cause anything from normal behavior, undefined behavior, or errors. But even if it works, it is very bad 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 fusing these files. However, this is only a superficial fix. You really have to find out the relationships between the function and classes in the files. Maybe you can reorder them, maybe you have to rewrite the corresponding code from scratch.

Breaking up dependencies

Circular imports are not a common problem as they are easy to spot and experienced programmers don't have such issues anyway. With good coding habits, circular dependencies won't show up. The much more common problem are too many dependencies. This makes the code become very sticky. It is very hard to give some numbers to quantify the problem, as it depends on a lot of factors. Breaking up a file is usually a good thing to do, but at the same time it increases 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 later means that most code needs all of the code from this file, so it makes sense to have it all bundled together.

How you break up a file is the even harder question. Sometimes you can easily bunch the code into groups, other times it is very hard to tell what belongs together. You can certainly split some of the code into a new file if you broke up a class into two classes.

The most important step towards lower dependencies is focusing your code. Make sure that similar code fragments are located at the same location. Having database access spread all over the code is usually a very bad sign. The logic of your code should be concentrated at a few spots. Make all the database requests at once, as far as possible, and store the results in a data class instance. Afterwards you can pass around this class instance and there is no need to think about the database anymore. And just like that you got rid of many dependencies and at the same time you improved your code.

The hardest part is reducing the dependencies by improving the general 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

Usually, circular dependencies occur as two classes exchange data between each other. Class A needs data from class B which needs some data from class A in return. You should be able to tell whether class A or class B corresponds to the higher level of abstraction. Let's assume class A is the higher-level class calling class B at some point. Now class B should never ever have to call class A. B is low level and knows nothing of 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 class B needs to return the final result.

Long story short: The high-level object calls the low-level object and hands over all the data required at once. The low-level object returns the final result at the end of the calculation. This resolves the problem of circular dependencies and sorts out 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 here is deliberately 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 it had to be.

```
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`.

```
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.

```
def a(counter):
    for i in range(counter-1, 0):
        print(i)
```

As a summary one can say that circular dependencies should be avoided all together. This is usually not too hard if you have proper levels of abstractions and it improves the readability of the code significantly. Even a single recursive call can often be refactored away and make the code more readable.

29. Decoupling

// I really have to rework this chapter.

"Before software should be reusable, it should be usable." — Ralph Johnson

[Refactoring, Martin Fowler], [The Pragmatic Programmer]

Coupling is a very essential part of software engineering. Without coupling, it wouldn't be possible to write code. Coupling is the glue that sticks everything together. But too much glue is bad as everything becomes sticky. In bad code, everything depends on each other. Every module or file imports dozens of other files. This is really bad because if you want to change one file, you might have to change a whole dozen. Instead you have to make sure that the coupling is as low as possible. This keeps the code soft and flexible. It is the ultimate goal to have comepletely decoupled code. This makes is 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 the worst as they instantly glue the whole code together. It’s worse than importing something everywhere. All your code starts depending on 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 automatically also depends on its base class. You are not only coupling the derived class to the base class, but also the other way around. You can barely change one without changing the other. This is not how flexible code is supposed to be. Don’t use inheritance.

Micro services on the other hand are very much decoupled. They are chunks of code that can be called and executed independently. Micro services are somehow similar to functional programming, where you have independent functions that all run by themselves. Micro services and functional programming both call a function or a piece of code that returns a value. [https://youtu.be/4GnjjocWGOE]

// A service locator is an intermediate object that knows about more or less everything. If you want something, ask the service locator. This is an anti pattern.

// instead of asking what you want, you go to the service locator and reach through the service locator. [https://youtu.be/RlfLCWKxHJ0] video on service locators

Only ask for things you directly need. This is another advantage of functional programming or micro services. If you have to validate an email, then call the email validator which does the job for you, where the email validator can be either a micro service or a pure function. The email validator returns a result and resets. You only got what you asked for, nothing else. There are no semi useful objects wobbling around that you don't know how to deal with them. You need exactly what is around. This is the strength of functional programming.

Law of demeter

// I don't exactly understand this law of demeter yet.

One common rule on coupling is the law of Demeter. Though it's not a very strict law. Martin Fowler called it "The occasionally useful suggestion of Demeter" [Refactoring p.192]. More formally, the Law of Demeter for functions requires that a method `m` of an object `o` may only invoke the methods of the following kinds of objects: [https://en.wikipedia.org/wiki/Law_of_Demeter], [https://www2.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf]

- `o` itself;
- `m`'s parameters;
- any objects instantiated within `m`;
- `o`'s attributes;

The idea is to avoid so called train wrecks where you start chaining methods to achieve something. For instance you shouldn't write code as

```
car.get_engine().turn_on()
```

Instead the `car` object should take over such function calls. Write a function inside the `Car` class `turn_on_engine()`,

```
car = Car()
car.turn_on_engine()

class Car:
    # define _engine somewhere
    def turn_on_engine(self):
        self._engine.turn_on()
```

This is a perfect example for a delegating class, as we have discussed in the chapter on classes.

Though as I already said before, the law of Demeter is only a vague recommendation and not a strict law. Don't become over enthusiastic about it.

Part 6: High level design

30. Software Architecture

// This chapter still needs a lot of work!

Architecture: "the decisions you wish you could get right early" - Ralph Johnson

About software architecture

Software Architecture is the high-level design of a software system. It's what you tell someone if you have to explain the structure of your code in 5 minutes. For example: "I worked on a quantum compiler. We used an Abstract Syntax Tree (AST) to represent the gate operations. These gates were then translated into electrical pulses that were played by our devices. The compiler consisted of many visitors that traversed the AST and performed all the calculations and optimizations, one after another." Anyone who knows what an AST and the visitor design pattern are will have a pretty good idea of the code I was describing. In 4 sentences I described the basic datastructure (the AST) as well as the basic algorithm (the visitor pattern) that was used in the code.

Layering code

// what goes here and what into the section The Abstraction Layers?

There are different ways to structure your code. You can have a layered architecture [Software Architecture Patterns?] or an onion architecture [clean architecture?]. Though in my opinion it does not really matter how you strucutre your layers. The important thing is only that they are structured and that the dependencies all go in one direction [https://youtu.be/KqWNtCpjUi8]. A low level object should never depend on a high level object. Furthermore a request should always go through all the layers and you shouldn't try to skip one. Even if this means writing a function that just passes on a command to a lower level without doing anything else.

Now this is a topic that is not that easy to understand. Why should you exactly write this function that doesn't do anything else than passing on a function?

The reason is coupling. Let's assume there are 3 levels of code. You want the low level to be coupled only to the intermediate level code. This has a very simple reason: If you violate this rule, the high-level code becomes also directly dependent on the low-level code. So in order to change the low-level code, you'll have to look out throughout your whole code base what other things depend on this low-level code. Meanwhile if you introduce these pass-through functions in the intermediate-level code, the only place where you have to change code is the intermediate level. The amount of work to change the code might be the same, but it's much easier to find all the pieces of code that need changes if you layer it properly. And if you have more than 3 levels, it becomes even worse if you don't separate the different levels properly.

Stability of code

//this section still needs some work

Let's take the following metric: we have a class A. How many (f_I) other classes have to be comipled in order to compile A? And how many (f_O) other classes depend on A? The stability of A is defined as I = f_O / (f_I + f_O). If the stability is 0, the class is very stable. It doesn't depend on any other class. If the stability close to 1, the class is very instable. It depends on many other classes [97 things every programmer should know, chapter 74]. The goal of refactoring is to increase the stability of the code, thus to decrease the number of classes that class A depends on.

Different Architecuter types

// should I eplain the different architecture types? horizontal, hexagonal, onion layers (clean architecture)?

// Client Server frequently has the question: should the logic go into the client device or into the server? The answer of this question has changed several times over the last few decades.

The end of Architecture

One question is: where does architecture end? Or how detailed does it get? And the answer is in my opinion: as far as 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. Because if he did, there would not be any need for software engineers anymore. So unless a project is very small, he has only time to take care of the very high level design. All the technical details have to be worked out by the engineers. Like real architects have to visit the construction site regularly, software architects also have to supervise the development process closely. Because the architecutre is never perfect and there will always be implementation questions from the software engineers. One thing architects can do to stay in touch with the development team is to write code themselves. They can write code examples or tests that use the code and do something interessting with it. [DDD p. 61 ?]

Designing Interfaces

One of the main jobs of a software architect is defining the building blocks (libraries) and interfaces of the whole software. Some of the interfaces may be only "partial interfaces", meaning that it’s an interface within a library. It fulfills all the requirements of a real interface and the library could easily be broken into two pieces at this point. It's an internal interface which is not exposed to the outside.

A partial interface has the advantage of needing only a limited amount of maintenance for versioning, etc. On the other hand, there is the danger that the interface gets lost over time as programmers start working around it.

It’s the architect’s job to figure out in the beginning where are what kind of interfaces required. He has to foresee the future. The YAGNI principle therefore doesn’t always hold for an architect. Because what if it turns out that we really needed that interface after all?

// write something about structuring classes into a library.

// Pretty much anything that holds for classes also holds for modules/libraries. Increase cohesion within a library and reduce coupling between libraries.

Separate Libraries

In every bigger code base, you’ll have to work with several libraries. Some of them are developed internally, others are 3rd party libraries. There are many things to consider when making such choices. The very first question is: do you need another library? Or can you implement the required functionality within an existing library? There are some mechanisms that favor smaller or bigger libraries.

// These rules were for domain models and not for libraries. Move it to the according location. Instead add the rules from Clean Architecture.

Reasons favoring bigger libraries are:

- Flow between user tasks is smoother when more is handled with a unified model
- If is easier to understand one coherent model than two distinct ones plus mapping between them
- Translation between two models can be difficult
- Shared language fosters clear team communication

Reasons favoring smaller libraries are:

- Communication overhead between developers is reduced.

- Continuous Integration is easier with smaller teams and code bases
- Larger contexts may call for more versatile abstract models, requiring skills that are in short supply.

These advantages for either sides lead to trade offs in library sizes. Generally, it is favorable to create a dedicated library if there is a corresponding opportunity.

Coupling

Interestingly, all the explanations about coupling and cohesion made for classes are also valid for libraries. You should pay attention that libraries are not becoming too large and rigid. You don’t gain a price for writing the biggest library in the company. One library that covers every object there is around. It just won’t work!

An apple can have a color, a flavor and a price. There can be three different libraries graphical rendering, food and shopping. Each one uses exactly one property and it makes no sense to mix them up. Keep them separate and write glue code between the libraries if needed. That’s the only way to go. Just trust me. Don’t write a monolith software that should mimic the whole world. It won’t work.

31. Domain Driven Design ???

// reread and rewrite this chapter? Or remove it completely???

"The complexity of your code should be at most as complex as the problem space it inhabits and no greater." - David Whitney

[https://github.com/ddd-referenz/ddd-referenz/blob/master/manuscript/] [https://youtu.be/kbGYy49fCz4]

This chapter is highly influenced by Eric Evans book "Domain-Driven Design" (DDD). The book covers mostly conceptual topics like the domain model and bounded context. This, along with the "Ubiquitous language" (Evans) it forms the heart of that book and will be explained in this chapter here. I did not understand everything that was explained in this book, so I just explain it the way I think it makes sense.

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 by the natural language. This is a huge issue as it is hard to bake such a topic into code. How do you implement an apple? The answer is: it depends who you are talking to.

It takes a lot of effort to understand a topic well enough to be able to implement it. It takes a lot of talking to domain experts about the topic. Only through these discussions you can learn how their domain model is built up and what the underlying mechanisms are. It is of utmost importance that the development team learns the language used by the domain experts, use among each other and implement it into the code. A domain expert has to be able to follow the general discussions between developers. He has to be able to tell when something is off as 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 the domain experts are able to tell much earlier that something is off than the developers. If there are expressions used in the code that do not exist in the domain, it is probably wrong. This common language between developers and domain experts was named "Ubiquitous language" by Eric Evans.

Developing this Ubiquitous language is of utmost importance for the whole project. Only a well-developed shared language between the developers and the domain experts allows high level discussions about the domain. It takes a lot of effort to develop such a language. Developers and domain experts have to remain continuously in touch and keep refining the use of their language and improve the model that is based on this language. Play around with this language. Try to change the words. Try to construct new phrases. This is an important part of the ubiquitous language. You have to develop the language like children learning to speak a natural language. Find easier and better ways to express what you want to say, no matter how stupid it sounds at first. Use the insight gained this way to improve the domain-model. Make sure the business experts understand what you are talking about. If you start using terms that they don't know, there is probably something wrong with your model. Do never use terms that are unknown to the experts, they are a sign for misguided logic!

Thinking about the code in the English language also helps even if you don't do much DDD. The following explanation from the book "The Art of Readable Code" can help you improve your coding skills by a lot:

1. Describe what code needs to do, in plain English, as you would to a colleague.
2. Pay attention to the key words and phrases used in this description.
3. Write your code to match this description.

Especially if you're stuck getting your thoughts into code, these steps may help you to order your thoughts and then writing the code becomes much easier. If you can’t describe the problem or your design in words, something is probably missing or undefined.

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. Interesstingly enough, a computer game does not necessarily become better if the model is more realistic. But rather if the model is more focused to make a point. If it emphasizes the core domain of what the game is all about, while leaving away unnecessary details.

When writing code, we implement a model of the reality. A model that resembles most accurately the problem we try to solve. Not one that is the closest to reality. The model has to cover the domain of interest. The field that you are working on. The model has to simplify the domain that you are working on to the bare minimum what you need to fulfill your programming task.

The domain-model is a high-level concept which has to be described. This can be done in several different ways. The most obvious description are UML diagrams. These are commonly used to show the relationship between different classes. However, UML diagrams are not always the ideal choice for describing code. UML has several deficiencies.

Documentation and planning

First, UML diagramms support only a somewhat limited amount of interactions between classes or class instances. There are frequently better ways to describe code than a class diagram. Maybe a piece of text will do, or a diagram that shows the temporal dependency of some process. It does not really matter how you represent the domain-model, as long as you understand it.

Second, one should always consider that UML diagrams should remain small. There are development teams that printed out their whole code base as a UML diagram, but this is pretty useless. There are way too many

objects with interactions between each other as if this graph could be useful. There were attempts to create a UML like programming language and they all failed for a reason. Graphical programming simply isn't any better than textual programming. Furthermore, a lot of information will get lost during the creation of the diagram. UML is not a complete programming language and it will never be. Keep UML diagrams small.

Instead you can use any kind of document you like. At times it is better to create a temporal order of a process than a class diagram. Or you create a diagram with class objects rather than classes. After all it's called Object Oriented programming, not Class Oriented programming.

As with all documents, the documentation of the domain core should be either kept up to date or archived. There is the danger that the documentation and the code diverge over time. Documentation has similar drawbacks as described in the chapter on comments. It takes a lot of efforts to keep a documentation up to date.

Though documentation has its merits. Code is often too detailed to really explain what it does. And there are plenty of things that code alone cannot explain. It has to be complemented either by comments or some additional documentation.

Make sure that design documents make ample usage of the Ubiquitous language. If the documentation does not use the same terms as defined in the Ubiquitious language, it is not useful. It doesn't help explaining what you are trying to implement. It only creates confusion.

Implementing a Model

There are cases where you can't implement a model you've 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 is able to 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 case where a developer can explain the code to the domain expert and the domain expert understands it. They would simply talk about the very same thing. In this case, the development of the code would feel very easy as everything just falls into place.

In reality, finding this optimal model is a really hard 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 how the optimal model should look like.

Decouple the domain-model code from your other code as explained in the section on The Abstraction Layers. This is important to keep the domain code clean and slim. Violating this rule would be also a violation of the SRP as the domain-model is located on a different abstraction level than, say, the database code. The domain model contains the actual conceptual complexity of the final software and thus it should not be cluttered with non-model related things like infrastructure or GUI code.

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 different domains in your project. For example the core domain. It is a first class citizen of the domains and has to be treated as such. The core domain is the most important domain of your project. The core domain is what your company makes money with, the thing that makes your company unique. The core domain has to be treated with special care. Try to keep it slim, only the most important things belong into the core domain. Your most experienced developers should be working on this topic.

// make some examples of domains? Are the exmaples fine?

Around the core domain you'll have several other domains. Each domain typically implements one class of features to support the core domain. For example an infrastructure domain taking care of the database, or the math library. Keeping the different domains appart is important as it prevents you from writing a Big Ball of Mud [https://de.wikipedia.org/wiki/Big_Ball_of_Mud].

Each domain corresponds to a piece of code, for example 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 that much resemblances between the different domains. For example the ubiquitous language does not have to be the same between different models. Rather the opposite. The ubiquitous language is expected to change between 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 be the time between take off and landing. But there may be also direct flights or flights with stop overs. This is an example where one expression may have different meanings, depending on what kind of model you are working in. Therefore it is always important to keep in mind what kind of domain model you are currently working in, what kind of flight you are talking about.

Refactoring toward deeper insight

// figure out what to write here exactly. Reread the corresponding chapter in DDD.

// This section is named after a chapter in the book Domain-Driven Design, p.322. It deals with a high-level point of view on refactoring, the domain level to be more precise.

- The design does not express the team’s current understanding of the domain;
- Important concepts are implicit in the design, though they should be explicit
- Important parts of the design can be made suppler

However, all the time you have to stay in close contact with a domain expert. Under no circumstances should you make changes that contradict what he says.

Deep domain models cannot be planned in a waterfall manner. They have to evolve over time. They only emerge from deeper insight that is gained over time. It has to be refactored toward deeper insight.

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. There is the desire to keep working on a unified model for the whole code base, but at the same time there are forces acting on the code to tear the model into smaller pieces. 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. The only question is: how do you deal with the different models?

Bounded Context

A bounded context is everything within a boundary. Typically a domain domain model corresponds to a bounded context, the boundary is its interface. The interface regulates what goes in and out of the bounded context. Bounded contexts are important as they separate some problems from the enterprise wide code base. One example of a bounded context is the math library. The things used in this library may be used elsewhere as well, but `sin`, `cos`, etc. have a very distinct and well defined meaning within this bounded context. These expressions are not to be reused within the math library. On the other hand, the expressions `sin` and `cos` may be used in other bounded contexts and have a completely different meaning.

Typically domain models consists of one bounded contexts. All the problems mentioned so far for the domain models are true for the bounded contexts as well.

Unified model

The attempt to keep the model unified is the most obvious one. Though it is hard to keep up the required level of communication to maintain this state. A good way to enforce this communication is Continuous Integration (CI). CI forces the team to merge often and early and therefore differences between the model and the actual code become apparent early. The automated tests enforce the behavior of the model and warn the developers if they are inadvertedly changing it.

On the other hand, working on a unified model is not always possible as for bigger project the forces tearing the single model apart become too big. In an entreprise scale software, it is simply not possible to work in a single model. The different requirements to the code become too big. In a single model, the `user` object for instance becomes too complex as it just keeps growing over time. At some point it is easier to deal with several different `user` objects where each one of them is smaller, making it easier to work with.

Context map

// this is really short. look at it again? Add a figure?

A context map is important when a model is split up in two parts. Both parts are now bounded contexts. They are individual domain models with a clearly defined boundary. 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 inteface the other one.

Shared kernel

Two bounded contexts may share a common sub-context. This is usually the domain core that is used by several different domain models. Having a shared domain core means that all involved models have to pay attention that the domain core is always in sync. This can be done by the CI. Additionally it takes quite some communication between the teams or else the core domain may get fragmented.

// The models involved with a shared kernel may be seen as ...?

Anticorruption layer

Separate ways

Sometimes the overhead of keeping models together simply becomes too big and it turns out that it is no longer useful to work together. It is no longer worth the effort. There is only very little overlap between the two models and cutting them apart is not such a big deal. If you need a feature from the other model, just

reimplement it. Having a little bit of redundancy between two models is still better than coupling them together. Of course it would probably be the best solution to break out the commonly used parts of the models into a third model that supplies infrastructure code for all other models. But maybe that's not worth it.

Conformist

Developer Client relationship

The model is split into two parts and one development team is relying on the model of the other team. 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 go into a developer client relationship where the downstream team can order features that the upstream team will implement. Whether this relationship works or not depends on politics and the goodwill of the downstream team inside the company.

Building blocks of DDD

// I really need to look at this again. Especially the part with all the IDs is not yet clear to me.
[https://youtu.be/jnutb5Z4wyg], see my question on stackoverflow:
[https://stackoverflow.com/questions/77425208/when-do-you-use-entities-value-objects-and-aggregates-ddd]

// what are entities, etc. used for?

In the book Domain-Driven Design, Eric Evans also introduced terms as entities, services, value objects and aggregates. These are different models to distinguish between different objects with different properties. Generally the building blocks of a domain-driven design are implemented in object oriented design. In most cases this is the easiest choice to model the functionality of the building blocks. However, other programming paradigms may be chosen as well.

I'd like to remark that you don't have to implement everything with entities, value objects, etc. the way it is explained here. It should be just regarded as some different way to think about how to structure your code.

Entities

// https://youtu.be/4rhzdZIDX_k

Entities are unique objects. Their lifetime typically spans over most of the code lifetime and they have unique properties and an ID. A very simple example are humans. Every human is unique and there are attempts to give every human some kind of ID. Though this is harder than it sounds. Obviously, names are not appropriate 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 other countries. For many websites, the email-address is used, at times also the phone number.

One example for entities are 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 there is exactly one seat reserved. Every seat has its unique ID. Two seats are only equal if their IDs are the same. Even if all other properties are the same, if the IDs are not the same the seats are not equal.

Now it is different if the tickets are not assigned to a specific seat (general admission). If the customers may sit on any seat available. Then the seats and customers are not entities anymore. They are just one object among

many. They become exchangable. They become value objects.

A tricky question is how to create a unique identifier for each entity. For example for a user of Netflix. A first attempt is to take the email address. This was done in many cases and is a good attempt, but it fails if the user wants to change the email address. Often it is better to create some kind of unique id, however this is not as easy as it sounds and is out of scope for this book.

When creating an entity it is important to stripe it down to the absolutely essential properties. // and ... ?

Value Object

// https://youtu.be/P5CRea21R2E

Value objects are pretty much the opposite of entities. Value objects are only defined by their properties. They don’t have a unique ID. One example are apples in the super market. They might look slightly different, but all together they are indistinguishable. The only interesting traits are the flavor and the price of an apple. Other than that, they can be replaced at any time. Value objects are immutable. You can only change the properties of a value object at 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. For example, you may have an email object. As it is a value object, it may be set only once by the constructor. Thus you can also do the checking of the correctness of the address in the constructor. You won't have to deal with it anywhere else. Common value objects are small custom types, for example a price. A price is set in its constructor and cannot be changed anymore. Furthermore the constructor can ensure that the price is valid, for example it can't be negative. Similar things can be done, for example creating its own type for email addresses, rather than using plain strings.

Now there is the question left when an object should be an entity or a value object. As I already mentioned before, value objects are immutable. So if you have an object, like the apple above, that will never change it's 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 an entity. Generally you should have more values objects than entities in your code.

Services

Services are used for operations on value objects or entities. A good service has 3 properties //Quote DDD p. 105:

- The operation does not naturally fit into 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 generally represented by functions.

While entities and value objects are generally too fine grained to be reused, services are medium grained and thus appropriate for reuse.

Aggregates

// aggregates should be small. They are always a single transaction to the DB.

Aggregates are a combination of several other objects. They typically consist of some entities and value objects. An example of an aggregate is a car. The car has a global ID which is its root entity. A car consists of an engine, a chassis and tires. Let's say that the tires wear off and once in a while you have to change them. This makes them an entity of the car. Meanwhile the engine and the chassis never change their state. These are value objects. The whole car is the only thing that can be accessed from the outside. The engine, chassis and the tires can only be accessed from within the car object. Once the tires are worn off, they are disposed off at the recycling plant. The recycling plant is probably modeled by a different domain model than the car. At the recycling plant no one cares anymore about how worn off a single tire is. The recycling plant simply consists of one huge pile of old tires. The tire becomes a value object.

// DDD p.127?

Here is an example how the car entity could be modeled in code.

```
class Car():
    def __init__(self):
        self.ID = "123" # some unique id
        self._engine = Engine()
        self._chassis = Chassis()
        self._tires = [Tire() for _ in range(4)]

    def drive(self, distance):
        drive(self._tires)

    def __eq__(self, other):
        return self.ID == other.ID

def drive(tires)
    for tire in tires:
        if tire.distance_remaining < distance:
            tire = Tire()
        tire.drive(distance)

class Engine():
    pass

class Chassis():
    pass

class Tire():
    def __init__(self):
        self.distance_remaining = 1000

    def drive(self, distance):
        self.distance_remaining -= distance
```

You might have realized that `car` is a delegating class. Delegating classes generally fulfill the requirements of an aggregate. A delegating class hides all the functionality within the class such that it's only accessible through the class instance itself, whereas the class instance is accessible over the ID of the car. It is not

possible to change the instance of the `car` and its internals in any other way. It is not possible to violate the invariants of the car.

The class `Car` might have the following invariants. It needs:

- a unique identifier
- 1 chassis (value object)
- 1 engine (value object)
- 4 tires (entities) with `distance_remaining >= 0` These invariants are guaranteed by the constructor of the class (assuming that the `ID` was in fact unique). All the other functions acting on the car have to make sure these invariants are still valid. The method `car.drive()` replaces tires if they are worn down, but there are still 4 tires with positive `distance_remaining` after the replacement and driving some distance.

An aggregate also acts as a transactional boundary. Or as we called it in the section on classes, it is either a delegating class or a data class. Aggregates should always be dealt with as a whole to ensure they are in a valid state. They should always be dealt with as a whole object, for example saved to or loaded from a database. Aggregates are always completely within a domain of your code.

As the root entity is the only thing accessible from the outside, it is comparably simple to enforce the invariants of the aggregate. For example every car always has to have four wheels that are not yet worn down. All the accessor functions have to pass through the root entity. Thus this is the place where you can enforce the invariants. There you can define functions as `drive` that takes care at the same time that the wheels are still fine and replaces them otherwise.

Aggregate instances are frequently created by a factory or another of the creational design patterns. 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 then it is a noteworthy task and should be dealt with in a dedicated object. Furthermore the factory can also take care of invariants of the class instance at its creation.

Organizing aggregates

// see graphic p. 181, DDD

Is there some general rule how to organize the aggregates? Or just throw all the pieces on the table and see how they fit toegther?

// some old text

Make code explicit

See DDD p.205

Explicit logic is much easier to understand than implicit logic. The logic is usually only implicitly known, but the logic in the code has to be explicit.

// write some examples here?

32. 3rd party software

"Prefer visa over power shell" – some YouTube video

// where to write about integration? // In the google book they write a whole chapter about the problems on how to deal with 3rd party libraries. This really seems to be an issue.

There are thousands of companies selling parts for your software. For many problems there are also 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 his own database software. Even if they don’t like the products they can buy for whatever reason, they get something from the market. Everything else 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 found your niche elsewhere and you’re going to stay there unless there is a reason to change your business completely. You outsource everything you don’t really have to do yourself.

Of course, in software engineering there are not so many products around to fit all the possible problems. But still, there are quite some suppliers around that can help you solve parts of your problems.

Possibly you somehow have a bad feeling about this approach. You want to do everything by yourself. You don’t want to pay other companies for some small libraries. I can assure you that your feeling is natural. But you have to get over it. It’s just not worth it. You didn’t write your own operating system. You earn a good amount of money every year. And if you can save some time by outsourcing parts of the code, this is great. You also save the maintenance which is usually even more expensive than the actual development of the software.

Using 3rd party libraries or software is great. Most of the times. But sometimes it also has its issues. There are some companies who didn’t write their code up to the standards and now they are in trouble. And there are many more sources for trouble. Most famously all the customers using Oracle databases who didn't decouple the database from the rest of the code. They use Oracle database querries all over the code and are unable to change to a different database vendor. These companies now pay hefty licensing fees and don’t get away.

Another problem are libraries with comparably 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 pop up when using software that is not anymore supported. Though if you really need this software, you might consider becoming a contributor yourself.

Everything explained here is also true for IT services and infrastructure (GitLab, Amazon Web Services (AWS), github, google maps, ...). These services are great, but you should always be able to change your supplier. For example there are medium sized companies that need a lot of computing power. They started the company using AWS as everyone else does as well. It's just too convenient. But as soon as the company got bigger, the bill from AWS was reaching millions, so the company was migrating to their own server infrastructure. [https://youtu.be/XAbX62m4fhI]

To put it brief: Third party software and services are generally great. 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 3rd party library from your code. Write a thin interface between your code and the library. And if you don't, make sure that you really want to stick to this specific thrird 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 you can possibly use to support other databases as well. So you are flexible.

The very big question is always when you really need such an interface. Most of the time I'm too lazy as well for writing one. But you certainly need one when dealing with databases. Call all the database specific queries only within this thin layer. The whole rest of the code is database-syntax-free-zone. This makes it very simple to exchange the database. You only have to replace the wrapper. You might have to change some of the implementation as well as the functionality between databases differs slightly. But this is a small price to pay compared to the millions you payed to Oracle so far.

You should rethink using a 3rd party library if it has only few developers. If there is a reasonable alternative, you'd maybe better refrain from it. On the other hand, this code could be absolutely essential for your own software, then it would be a good idea to join the project and become a developer as well. In fact, pretty much all major software companies support the software projects they are relying on. Some projects got that much additional man power that they run 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

33. Working with Existing Projects

"Work on the assumption that code is a 'best guess'. It is probably wrong." - Dave Farley [https://youtu.be/gLYYXKL-Jug?t=760]

Up to this point everything was great. We had no restrictions what so ever. We assumed we worked on a so called green field project. I could tell you whatever I wanted. There were no restrictions due to the existing code base. "One beer please. Before I am forced to tell you how to wiggle around in an existing project."

Yes, working on existing projects can be hard. Sometimes the developers made some very obvious mistakes. But at the same time, it is really hard to keep everything in shape. In every software development there will be this point where you ask yourself: "Gosh, how did I screw up this code so badly?" Even if you follow all the advice this book gives. It will happen to everyone. So, if you start with your first job and the code looks nothing like what I explained so far, don't be disappointed. Don't be too harsh with your coworkers and your boss. Yes, it is not really motivating to work with bad code. But there is still a lot you can learn. And unless some extremely fundamental flaws were made it is very well possible to make improvements.

You might be motivated to suggest a complete rewrite of the code. You may do that, though I do not recommend it. A complete rewrite is hardly ever an option. It takes years, costs millions and very often the final code is not that much better. Generally, it is better to improve the existing code. You spot something you want to improve. 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's worth millions. You are not going to fix it in a few months.

There are some different stages of how bad the code can be. I'm trying to give you a short overview.

No Interfaces

This is probably the worst case. Without interfaces it is impossible to write tests. Without tests it is impossible to refactor and add interfaces. It's really bad, but it's not a lost cause. You can still try to refactor slowly. Though it will be painful and you constantly have to look out for bugs. A sales person will have to check your

work frequently. The whole refactoring will probably take years. Maybe a complete rewrite is indeed the better option. I hope you never end in this kind of position.

No Tests

Code without tests is one thing. One can still write them later on, even though it takes much more efforts. The real issue is probably the low quality of the code. It has some interfaces but the classes are way too big and the objects are hard to create. This is one of the few cases where you are officially allowed to cheat. You may make private methods public in order to test them. Once the refactoring is done, you should make it private again. In a year or two. Once you have some test coverage, you can break the classes into smaller ones.

If you work on an existing project, there might be no or only an insufficient number of tests. This is a serious issue. Not only from a technical point of view, but also a political one. Due to the bad test coverage, one might introduce bugs when refactoring. And the last person to touch the code is becoming 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 wanted. You only wanted to improve it, not own it. Ultimately, people are afraid of refactoring the code because they'll become responsible for it and not so much, because it would be hard. Therefore, the developers stop refactoring and the code decays even faster than it did before.

Extremely long functions

Let's be honest. A function, or even worse a method, of about a thousand lines is an absolute nightmare. No one will ever understand it with all its corner cases. It is absolutely impossible. No one is ever going to touch it. You might be able to make some small changes, but you are not fixing it fundamentally. The only way to really change it is a complete rewrite. The hardest part about it is getting the specification what the function actually did so far. If bugs are absolutely not allowed, you'd better just leave the function as is.

34. Refactoring Fundamentals

"If you wait until you can make a complete justification for a change, you've waited too long." – Eric Evans

There will be change

If code lives long enough (which is usually sooner rather than later), it will have to adapt to change. The build system might change, the database changes, and you'll have to adapt your code to the new environment. This is almost inevitable. Only if you write extremely low level code with hardly any dependencies you might be safe. Or if you write mobile apps that are guaranteed 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 stay flexible. You have to keep it in shape. Make sure you can react to change.

Keeping code in shape

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 possible source for deteriorating the code quality. You may add duplication, increase the class size or disrupt the order of logic in your code. Brief, the code becomes dirty and you have to clean it 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 efforts as explained in the section on entropy [chapter Physical Laws of Code].

I guess everybody reading (or writing) this book knows some of the problems why code rots. The very first example is copy paste code. Copy paste code should be banned altogether. Instead of rewriting a function to fit its new needs, one just copies it and changes a line or two in its new location. Another issue is adding more and more features to an existing class. Also the features that you add are in hindsight at the wrong place in your code and have to be moved to the correct location. These are some of the reasons why code rots and needs regular refactoring.

Refactoring and automated tests

Refactoring means to change the code without changing its functionality. This is what people didn't do in very old code. They were afraid that they would break existing functionality. They would introduce bugs. It's like 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 say what exactly. Long story short, the next person had to cook in a dirty kitchen. And at some point, there were so many dirty dishes in the kitchen they didn't even see the bugs anymore 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 were left with bugs anyway. They didn't clean up the kitchen nor refactor the code. They started adding many more bugs further down the road, because the whole code base 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 take you your fear from refactoring, your fear from introducing bugs. You need a safety net. Something that automatically tells you when you introduced 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 to highlight the lines of your code covered by tests. Or you can also change one line of code and see whether one of the tests fails, though this is not a very productive solution.

If you are confident about the test coverage you can do pretty much anything you want. Whatever code you don't like, just throw it out and rewrite it from scratch. Or even better, use a third-party library if available. As long as the tests pass you are quite certainly fine.

Keep refactorings small

Most refactoring is fairly small. Renaming a variable. Breaking up a class into two new classes. Removing duplicate code. Extracting functions. Rewrites of complete features are comparably rare. The biggest mistake one can make with refactoring is waiting for too long. If you have the gut feeling your fundamental data structure could be an obstacle you should act right away. Discuss with your work colleagues whether this is really the correct choice and what other options you would have. 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 as soon as there is some code 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 the tests before the code, as explained in the section on Test Driven Development [chapter Writing better Code with tests]. This pattern is great because you can really do

one thing at the time. You can write mediocre code to start with. Maybe you don't know yet how a variable should be named or you tend once again to write a class that is too big. Maybe there's even duplicated code. Certainly, it would be better to write perfect code right from the beginning. But you cannot multitask. You cannot develop code and make it perfect at the same time. You're not perfect. Learn dealing with your imperfections and refactor your imperfect code.

Then you write the tests. Some tests may fail as your imperfect code might contain bugs. You fix the bugs and the code becomes even more ugly. 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 missed 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 in order to end up with good code.

Finally, you refactor. You look at all the code that you wrote since you refactored the last time. Possibly also at code that existed for a long time and could be merged with your new code because it's very similar. The code will probably look more complicated than you would expect it to be. You try to rethink the logic of the problem you just solved. Can you change the algorithm somehow that you can drop all the `if` statements for the corner cases? Or do you have to sort the data differently to make the code better?

There are hundreds of things you could do for improving the quality of the code. Look at the code and figure out what the most important things are you want to change. Try to write good code and follow your gut feeling. But make sure you also get some real work done between the refactoring sessions. The code will never be perfect. But it will be good enough. Move on once this is the case. Don't be stuck with endless discussions on the name of a variable. Go on and write some new code again.

Levels of Refactoring

Maybe you came up with a simple question: On what level should you refactor? Should you refactor only the small things, or should you dig down to the core of your software?

Let me make another small example. Let's say you are going to build a house and you like cooking. So you make sure in the kitchen is ample space for all your equipment. You are very pleased. This is the equivalent to a first draft of your code. Everything looks perfect.

Yet dishes get dirty you still have to clean up the kitchen every day. Otherwise you'd be in no time left with a huge mess. This corresponds to the everyday refactoring of a software engineer. Make sure you remove code duplications, name all variables properly and clean up everything along the way you don't like.

Once in a while you buy an additional kitchen device and over time, you start running out of space. You have to sort out all the old devices you don't need anymore, and make use of your Tetris skills to fit everything back into the shelves in an ordered fashion. Such that you still find your belongings. This is an intermediate refactoring.

At some point you buy another device and you realize there is not enough space for your equipment anymore. There is only one solution. You need a bigger kitchen. You have to plan how much additional space you need for the next few years and either tear out some walls or expand your house. Now this will be a very demanding and expensive refactoring.

I hope you got the memo. Small refactoring should be done all the time. Every few lines of code. The costs are low and it keeps your work space clean. Intermediate refactoring costs more and effects a fair amount of your code base. It should be discussed with your work colleagues during the coffee break and may be done

together. Big refactoring is really labor intensive. It’s done only every few months and takes good planning and dedicated meetings as there is a lot at stake.

Refactoring is dynamic

Waterfall refactoring is bound to fail the same as most waterfall projects are. Refactoring is concrete. Just as normal coding, it consists of a learning process while you are doing it. It’s a feedback loop. It usually has 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 even have to drop these grand plans all together 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 how refactoring is done.

// Make circle graphic: changes to be made, make changes, more changes to be made

A refactoring certainly has the highest impact if you have some new understanding of the problem you try to solve. This allows you to rewrite a complete piece of code at once and make significant progress with your code quality. Eric Evans calls this "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 instance, working with code containing global variables, many dependencies, huge classes etc. is always a pain, no matter if you are writing new code, tests or doing some 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 becomes a nightmare because you have a circle of doom. You start postponing your refactoring because it’s hard to understand bad code. But over time, this will only make it worse and worse and worse. Until you reach the point where refactoring is essentially impossible and you are paralyzed. You’d have to refactor your code because it’s bad, but you can't because it's gotten too bad to make it any better.

Don’t slack off refactoring. You’d pay the price rather sooner than later. Make sure you always keep the code in shape, this makes your life much easier.

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 rethink fixing it as the users may rely on that bug.

When to Refactor

It is generally a good idea to do refactoring. Most developers do rather too little refactoring than too much. Still, there are some general recommendations when to refactor or not.

Every few lines of code you wrote, you should consider refactoring them. It is not always necessary, tough it is by far the best moment. You still know what you just programmed and you might have an idea what there is left to improve. Maybe you just introdced some code duplication? Additionally, you are always working on a tidy workplace which increases your productivity. Code that’s well taken care of is much easier to modify.

As already mentioned above, you should always refactor the code you just wrote. This is the number one rule. Furthermore, you should stick to the boy scout rule: Leave the camp ground tidier than you found it. Always refactor a little bit more than you should have. This helps fighting the code entropy.

Refactor when you found a bug. Don't just add a patch that might resolve the issue superficially. Search for the real source of the problem. Then consider if there is some redundant code that might need fixing, or better refactoring, as well. Find a good fix for the bug, possibly including some refactoring.

If you add a feature, it may not really fit into the code. Most likely, because the code has not been cleaned up, or the other authors simply didn't know how the code should look in the future. Thus, the code has a different structure than you would need for this new feature. But now, as you're adding this new feature, you're smarter. You might have an idea how the code should really look like for the feature to fit in. Now don't squeeze the feature into the existing code base. Refactor instead and make sure the new feature fits in smoothly. Maybe transform a datastructure as explained in the section on Orthogonality. Altogether, this is less work. And especially the code will ultimately be in a much better condition. This was explained in the section on orthogonality. Write an adapter to make the feature fit neatly into the code base.

Also, during code review you can do refactoring. Team up with the author of the code and do some pair programming. This is much more motivating than a normal review as there is better knowledge exchange and the output of the review is significantly increased.

Generally, you should refactor code that you work with. In some cases, you may refactor code that you just walked by, but this should not be the rule. If there is no reason for you to touch that code at the time being, you shouldn't refactor it. It is important in software engineering to know when to postpone some work. And this is one of the cases. If no one works with some piece of code at the moment, then there is no need to refactor it right now.

Once in a while, you have to do a bigger refactoring. One that you don't just do between writing a few lines of code, but it will take considerable efforts to get it done. You should probably discuss this topic with your work colleagues, opposite to the smaller refactorings that you just do by yourself.

Last but not least it is your code. You are responsible. You are the one to decide it's time for a 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 that I also use when writing this book here. I first started with writing down the basic ideas. Some rough drafts of what I wanted to have in this book. Some ideas I had for a long time, others I got while reading other books. Then I was reading the text over and over again and reworked it several times, clarifying something here and there, removing redundant parts, moving chapters around and adding some more explanations where needed. Every time I started to understand my text better and could further improve it. Until I was roughly at the point where the text said what I wanted it to. Until I had put all my knowledge from my head into the text and sorted it out into a human readable piece of text. Or as Ward Cunningham had put it: "By refactoring I move the understanding from my head into the code."

// add the graph from p.193, DDD. Refactoring is a non-linear process.

Refactoring, just as writing code, is a highly non-linear process. It cannot be planned too well because it is a creative process. And knowledge gains may come out of the blue. All of a sudden you understand the problem much better and the code can be improved accordingly.

35. Refactoring techniques

"To me, legacy code is code without tests" - Michael Feathers [WELC]

// WIP: This needs some more work. Read WELC and Refactoring and add some more examples.

The techniques explained here mostly require an existing set of automated tests as changes to the code may introduce bugs otherwise. Refactoring can be done also without tests, though in most cases, it is a very dangerous game to play. Even if some techniques seem save 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 overriden functions it becomes tricky. Refactoring code in compiled languages is a little bit easier than for interpreted languages as the compiler does valuable checking of names, functions, types, etc.

There is a plethora of concrete refactoring techniques to be applied in specific cases. I will only briefly explain some of them. Most originate from the book Refactoring of Martin Fowler [Refactoring, Addison Wesley, 2019]. In the following I will group these techniques into two categories: one category mostly explained in [Refactoring, Fowler] for good code and the one categoriy from [WELC, Feathers] for bad legacy code with global variables, inheritance, no tests etc.

Note that some of the techniques explained in the section on good code, for example renaming, can also be applied for legacy code. Meanwhile the techniques explained for legacy code don't make much sense for good code.

Refactoring good code

When following the rules taught in this book, you should be writing good code. It is well tested, contains clear interfaces, no global variables, no side effects, etc. Still, you have to refactor once in a while. But it's comparably easy because you can focus on the refactoring part. The tests are already in place. In this section, you will learn some techniques that you can apply.

Renaming

Even though renaming hardly changes the shape of the code, it should be done extremely often. Not only for good, but also for legacy code. Finding good names is one of the hardest tasks in programming as judging the quality of names is very difficult. There are some general rules how naming should be done, yet still it’s 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 you just happen to know a better name. Then rename this object. This is the only way names get better over time. Don’t assume the author of the code knew it better. You have much more information now 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 gotten accustomed to the whole 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 in the beginning of the development and not change them anymore later on.

One possibility is to use only mediocre names to beginn with and search better names only at the end of programming a few lines. Then also Copilot can help you find better names.

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 as too long functions are an extremely prevalent phenomenon and extracting functions is the main mechanism to get it under control. If there are not too many variables involved, the technique is fairly simple. The biggest difficulty is finding good names for the newly created functions.

Let's say we have this very simple code here. We already saw that this is violating the SRP as printing a string and calling a function are 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 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 in order to create the new function as 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 use Copilot to extract this function. Just write the command "move the print statements into a dedicated function" and Copilot will do the rest for you. Though as always, you should pay attention that the solution is correct. In this case it happened to me that Copilot suggested an incorrect solution.

There is not that much more to know about extracting functions than what I just showed here. It is a really simple refactoring technique, yet it is very important. This is probably the most used refactoring technique besides renaming. The only thing you have to watch out for are the variables used by the newly created function. If the code is inside the class you might decide to make the function a member function of the class as well, because otherwise you might have to pass too many arguments to the function. Though this would be a sign of bad 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 is the opposite process of what we just saw and used rarely. Take a function call and replace it with the function body. Apparently, this makes the surrounding function longer as soon as the copied function body has more than one line. This is generally not desirable as 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 you are planning to split up the old function. One advantage of inlining function 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.

Scratch refactoring [Feathers p. 212]

In chess there is a rule of thumb that you should quietly talk with your pieces during your opponents turn. You should ask them where they would like to be and thus get a feeling for the position. In programming there is something quite similar. Scratch refactoring is not about improving code, it is only about getting an idea how the code could look like. Just refactor as you like without caring about bugs or similar issues. Figure out how the code should look like in a dream world. But also try to implement some of the edge cases that will make your life harder to get a feel for the drawbacks of your dream implementation. I like this concept of scratch refactoring very much as it gives you an idea how the code could look like instead.

Once you're done refactoring, discard everything and do a normal refactoring, trying to apply the ideas you just got. Pay attention that you don't just lightly reimplement the code you dreamed of before, you might have missed some technical details and the solution from the scratch refactoring might not work out the way you did it. After all, the scratch refactoring was just a dream...

Refactoring legacy code [WELC]

// WIP

// Stuff here is from Feathers book. Read it again and add some more stuff here. // It's not really about refactoring anymore. The refactoring has been treated in the previous section. This is more about the problems you face when refactoring.

So far we only refactored code covered with tests. Refactoring code without tests would be too dangerous. But unfortunately, this is exactly the problem with so many projects. There are so many of projects out there without tests. And because of global variables, functions with side effects, complicated constructors, etc. it is very hard to write tests for them. In these cases you start getting afraid to changes the code as you are supposed to do in a refactoring. There's just too much that can break without test. This is apparently a really bad thing. No one likes to life in fear. In your own code you can prevent this situation by meticulously testing all the code you write, but if you work on an existing project, you will have to face the demons.

Refactoring untested code is usually a very hard task, there are whole books about it. And if the code is already pretty bad, refactoring becomes even harder. The most common issues on the macro level are

1. No tests
2. Obscure code
3. No time (or budget) to fix it

And on the micro level we have a few more indications that things will get tough:

1. No interfaces
2. Functions with side effects and global variables
3. Huge classes and functions
4. Objects that are hard to construct
5. Inheritance chains

Let’s say you want to break a class into pieces, but it's really big. It has no tests and you are uncertain of the side effects it might have. This is bad as functional changes introduced are bugs. The only way to prevent these changes is having plenty of regression tests.

How do you refactor legacy code?

First of all, you have to change as little code as possible to get tests in place.

1. Identify change points ("Seams")
2. Break dependencies
3. Write the tests
4. Make your changes
5. Refactor

The difficult points are number 1 and 2. The rest is text book refactoring.

Seams

Writing tests would be a very noble thing to do, but it is not always that easy. As I explained before, how easily you can write tests depends highly on the quality of your code. In order to write tests, you need something you can get a hold on. Michael Feathers calls this a "seam". "A seam is a place where you can alter behavior in your program without editing in that place." [WELC] Vice versa, you can edit it elsewhere, in the so-called enabling point.

There are several different ways to implement seams. The best seams are interfaces and dependency injection. They are very easy to deal with and resemble normal code. Just create a new implementation of the interface or inject it and you are done.

Some of the seams explained in [Working Effectively with Legacy Code] change the behavior on the compiler level, either by the linker or the preprocessor. Needless to say that implementing such kind of fancy seams is a fairly desperate measure. Such techniques resemble strongly black magic and should be avoided.

The most common seam is simply function arguments. It is not mentioned in Working Effectively with Legacy Code and the following code is just a strictly worse version of using dependency injection, but it is still 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:

```
def create_reader(debug):
    if(debug):
        return DebugReader()
```

```
    else:
        return Reader()

def main(debug=False):
    reader = create_reader(debug)
    reader.read()
```

Usually just passing a number or a string is not sufficient for implementing a seam as changing their values does not alter the behavior of the function significantly. It only yields a different result.

The piece of code you hold in your hands between two seams may be way too big and you have no idea what you should test exactly. In the extreme case the only tests you can write are functional tests. And if you don’t have any useful API you can write your tests with, you might be completely screwed. I'm sorry, there's no other way to say it.

And no, I'm not exagerating. Spaghetti code without tests can be an enormous issue and there really seems to be no solution. A friend of mine was developing gas turbines. And there was one person who developed a complete software that took some parameters and created a complete CAD model of a turbine. Now the problem was that this person got retired and the code was a 15'000 line long mess. The company payed millions in a desperate attempt to refactor the code, but failed. In the end they just worte a wrapper around this piece of code and left it as is.

Sketches

Making sketches and diagrams may help you finding 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 some kind of temporal behavior or what Feathers called a "scratch refactoring". Basically, a draft code that shows how the final code could roughly look like without considering all the details that make real refactoring so hard. These are all tools that help you understand your code better and make it easier to write the actual refactoring code.

// Add the temporal graph from Evans? which one? nonlinear growth?

[WELC p.200(?)]

How do I get the code under test?

What tests should I write?

Sprout method [WELC p. 58]

Let's say you have some method or function that you can't test but you have to add some functionality. How do you do that without deterioating the code quality any further? The solution is adding a new function or method that you can test and call it from the old function. This is called a sprout method (or function).

Assume we have the following code that we can't test for whatever reason. In reality it would of course be much more complicated. I just simplified it enough for the sake of making a readable example.

```
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.

```
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 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
```

```
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)
    for entry in valid_entries:
        entry.post()
    transactions.get_current().add(valid_entries)
```

So we managed to add only one additional line of code to the original function. All the other code went into the `get_valid_entries` function. This new function is now also unit tested.

Sprout class [WELC p. 62]

// WIP

If you have a class that is getting too big, you can extract some of the functionality into a new class. This is called sprouting a class. The new class is usually a member of the old class. This is a very simple refactoring technique. Just make sure that the new class is only loosely coupled to the old class. Otherwise you might

have to pass too many arguments to the new class. If there are too many arguments you have to hand over, you should maybe reconsider your class design and rewrite it such that it has less coupling.

// I think there is still quite something to write here. Maybe add some more examples?

// "When you use Sprout Method, you are clearly separating new code from old code. Even if you can't get the old code under test immediately, you can at least see your changes separately and have a clean interface between the new code and the old code. You see all of the variables affected, and this can make it easier to determine whether the code is right in context." - Michael Feathers [WELC]

// make an example [https://www.codewithjason.com/taming-legacy-code-using-sprout-method-technique/]

1. Write a test around the buggy area—expiration date validation—and watch it fail
2. Extract the expiration date code into its own method so we can isolate the incorrect behavior
3. Fix the bug and watch our test pass

Copilot

It seems as if Copilot sometimes struggles with refactoring code. At times it doesn't find a reasonable refactoring suggestion. Though with playing around a little and giving Copilot 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:

```
    # 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.

```
    # 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:

```
    # 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. Performance Optimization

"Premature optimization is the root of all evil" - Donald Knuth

No optimization needed

One of the most overestimated topics in programming is performance. This has historic reasons. Computers used to be extremely slow and expensive. Thus, it was worth spending a lot of time improving every bit of your algorithm. Back in the days, low level languages like Fortran or even Assembler allowed you to do so. But the performance of computers had been growing exponentially for the last 50 years, while the price of computers dropped considerably. Modern programming languages like Python are not focusing on performance anymore. But rather on usability. Simply because it is more important to write readable code, rather than fast code.

As we have learned the main goals of a software engineer are creating value for the customer, writing code that is easy to understand, correct and well covered with tests. Performance is not a main goal. 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 make most standard programs run at a reasonable speed without optimizing them.

I'd like to remark that the way I recommend to write code does not result in fast code. I didn't care about speed so far. Instead I was coding for readability and reusability. The problem is that all this polymorphism that I recommended requires look ups at the so called v-table and this is slow. There are youtube videos [https://youtu.be/tD5NrevFtbU] that explain these things in great detail. So yes, the code I recommend you to write is comparably slow. But it does not matter. When do you need millions of function calls to 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.

Optimization might be needed

Still, let’s say you start writing one of the few applications that you assume needs performance. You’re a bit lost at which point in time you should start optimizing the code. Right from the beginning? Should you plan your algorithms such that they will 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 performance topic for the time being. Write your code with the usual 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 had this feeling that you had 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 have to run your code only once and it takes 2 days, run it over the weekend. Spending hours for optimization would be wasted 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. Pretty much all code that you’ll ever see has very few bottlenecks. Usually it’s some fancy calculation on a huge data structure that scales worse than O(N*log(N)) [https://en.wikipedia.org/wiki/Big_O_notation]. 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 find this bottleneck using a profiler. For example, it turns out to be self written Fourier Transformation operating on a list with 10’000 elements. So, as you start reading through that code, you realize that the algorithm you have implemented scales with O(N^2). Such bad scaling is usually unacceptable. You ask the internet for advice. You find Fourier transform libraries that scale with O(N*log(N)). As your code is well structured you can just remove your own Fourier transform

function call, tweak your data structure a little and use the library you found. Now your code runs within seconds. Done. You won't have to care about anything else.

Optimizing from scratch

Finally, there are indeed some cases where you have to plan the software from scratch and focus on optimization. But these are very rare. These are mostly simulation software, games, websites containing a lot of data, or infrastructure code for huge server farms where not only performance but also energy consumption it a major concern. If the code can be parallelized, it will become much more complicated as this is an additional complexity when designing data structures and algorithms. As a very rough rule of thumb, it takes twice the amount of time to write parallel (or distributed) code compared to linear code. There is a lot to learn if you want to write high performance code. But you won't be alone. You'll be likely working in a team where every single team member knows way more about parallel programming than I do.

There are many small things you can do for optimizing your code like manual loop unrolling. Keep your hands away! The performance gains are negligible. And if you are working with a compiled language, the compiler can optimize such things much better than you do. Only improve major algorithms. Especially those that scale better.

Always keep in mind: code that was written with performance in mind, rather than readability, is always very hard to maintain!

Part 8: Miscelaneous

37. 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*.

Comments are a very double-edged sword. While they may be useful at times, they are also a liability. You always have to make sure you keep them up to date as you have to any piece of documentation. Additionally comments tend to be a remedy to fix bad code. And this is certainly not what comments are supposed to do.

Bad comments

"Comments? Don't."

"Why?"

```
def add(a,b):
    # This function returns the sum of the two arguments
    return a + b
```

Of course, I exaggerated in this example. I just wanted to make a point. But there are programmers out there who think that this comment here is justified.

I do not share this opinion at all. In my opinion this comment is just a useless boilerplate comment. Read the function name. It explains exactly what the function does. And if you are not sure, take a look at the implementation. This is exactly what makes code good. You read a function name and you know what it does. Good code is self-documenting. There is barely any need for additional comments. This comment here is a violation of the SRP.

"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 claims something 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 can't enforce that a comment stays at its correct location. You will eventually end up having a comment that is plain wrong. It will confuse everyone who works on this code. It will cost time. It will cause bugs.

Not convinced? You think you won't have these issues because you work carefully?

"Ha ha. NO!"

Now you're certainly wrong this time. By now you should know better. This is exactly what I'm trying to teach you throughout this whole book. You are human. Every human makes mistakes. I make mistakes, you make mistakes. It's inevitable. Accept your faith and deal with it. Code is good if you can make as few mistakes as possible. Removing useless comments is a must. They are an unnecessary source for bugs.

You want to become a software engineer. So stop using the English language and start reading code instead. 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: Simple and Practical Techniques for Writing Better Code, Boswell & Foucher]. The original code was written in C++, I translated it to Python.

```
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)
```

Undoubtadly, this code is bad. It is very hard to read this code. There is too much code without any structure.

The authors of this book formated the code a little and ended up with something like this:

```
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 certainly became much more readable. But adding these comments doesn't solve the fundamental issue: This class should be broken down into 3 sub classes and one dataclass as a parent 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 own suggestion 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, including the user code of it. But it is both much better structured and there is no need for any comments. Note how we were also able to simplify some parts of the code. For instance we now write just `view` instead of `view_profile`. The profile part of the function name is now already clear due to the context inside the `Profile` class.

Here is another example from the same book. It suffers from a similar problem: The authors tried to improve the code by adding comments instead of improving the code itself.

This is the original code. Needless to say that it is not very readable. It lacks any visible structure.

```
# 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 but refactored.

```
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.

```
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)

    items = create_dict(user, friends, suggested_friends)
    return render("suggested_friends.html", items)

def get_friends_emails_of(user):
    return set(f.email for f in user.friends())

def import_email_addresses_from(user, email_password):
    contacts = import_contacts(user.email, email_password)
    return set(c.email for c 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 became only quite little longer compared to other refactoring examples. But at the same time it is so much more readable. You understand what it does by just looking at the top level function `suggest_new_friends`. You don't have to read the details of the function. You can just read the function names and you know what it does. This is what makes code readable. Not the comments.

At times it is very difficult to explain code with code alone. So there is of course the temptation to use a comment to make it clearer. As in the folowing example, also from the book "The Art or Readable Code" (I would like to mention that I really like the book, but I don't agree with all the examples they use):

```
// 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 do have an issue with this comment. It is very hard to understand. And as always, having a comment to explain code is always suboptimal. Now the first problem I see with this function is that it does two things at the same time. It orders the elements of the vector and it returns the index of the last element that is smaller than the pivot. It has a mutable argument and a return value at the same time. This is a violation of the SRP. The function should be split into two parts.

Additionally there is something else that can explain code: unit tests. The test cases act as examples how the code is supposed to be used. This is frequently a better help than some 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 more guessing that writing structured code. He knew his work was really bad. Therefore, he decided to leave the old code in the repo 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? Everybody reads it. Nobody knows how to deal with it. It's just causing confusion and wastes 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 show this book as a proof if needed.

TODO comments

Another bad habit is TODO comments. When you implement a feature, you are responsible that the implementation is ready to be merged into master. It's ready to be merged when there is nothing important to be done anymore that would justify a TODO comment. Make sure you never merge any TODOs into master. They only cause confusion and there is never time to do them. You will never implement a feature without a ticket and for refactoring the code you don't need a TODO comment. Therefore again: make sure you never merge any TODO comments into master.

At the same time it is fine if you use TODO comments during the development of a feature. It might help you to organize your work. Just make sure to remove all the TODO comments before merging your changes into master.

Comments replacing code

Introducing a lot of tiny functions hurts readability to some degree. It takes keeping track of and jumping around the function calls. Though this cost is very low if the functions are named properly. If all the functions do what they say, you can just read the function names and you know what the code does. This is what makes code readable. Not the comments.

As a summary I can say: yes, all the small functions have 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 talk about the cases where using comments is fully legitimate.

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 you cannot express in code. For example, you can add links to the source of a code fragment, library or the explanation of an algorithm. It may also be useful to use comments on the interface of a library or API used be the documentation software. And of course comments are used at the beginning of the file for the boilerplate copyright statement.

Requirements

A very legitimate use of comments are requirements. A comment that the code has to be this way because of some requirement. Requirements are something you cannot explain in code. They are usually written in a natural language. Yet 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. 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 enough. They are not visible in the code. You have to search for them. And you don't know which acceptance test belongs exactly to which line of code. Therefore, comments are the only thing I could think of that can link the code to the requirements.

How to write comments

Just as code, comments should be as short and pregnant as possible. In the following example we have the opposite. What does "it" in the following sentence mean? Don't write such 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, like sphynx in Python, for automatically generated documentation. However, docstrings should only be used as an external documentation. Never use docstrings for internal purpose. Why should you read a docstring documentation if you can read the source code and all its comments?

// should I add some more points when comments are allowed?

Commenting magic numbers

// Move the following example to the chapter on comments?

Here we have an example of bad code, for once it's C++. I found it in "The Art of Readable Code" [The Art of Readable Code]. The authors correctly state that this code is hard to understand. But unfortunately they failed to explain why exactly. Note that this example is in C++ because the suggested solution does not work in Python.

```
connect(10, false);
```

This code is obviously 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.

There are two better solutions to this problem. In Python, C++20 and most other modern programming language, keyword arguments are supported.

```
// check that this code really works!
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, see chapter on Naming.

```
int timeout_ms = 10;
bool use_encryption = false;
connect(timeout_ms, use_encryption);
```

Here I didn't even have to type anything as Copilot was able to suggest the correct solution.

Summary

Use comments only for things that cannot be made apparent by the code itself, yet you think it’s still very important.

Copilot

Copilot is not yet able to write more than boilerplate comments. The following comment was created by the document function of Copilot Labs.

```
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
```

38. Logging

// this chapter is very short. merge it with another chapter?

The basic idea of logging is to have a feedback what kind of steps your software executed. It might help you finding bugs. Now this sounds great, but in reality, there are several things to consider.

- The most obvious drawback is that logging needs time to be implemented. It's not a huge amount, yet it may add up.
- Logging polutes your code, similar to comments.
- Logging is usually not needed. Most code is deterministic. If you run the same code twice it will do the exact same thing, down to some 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 finding your bugs, you should rather improve the quality of your code. Simplify its structure and write unit tests. You will have less bugs and they are easier to find.

At the same time there are some cases where you can consider using a logger.

- For non-deterministic software it may be useful. For example, if you have several programs that communicate asynchronously with each other, as in microservices. All kind of race conditions may occur that you didn't think of. Depending on the temporal order of the messages being sent. A logger may help you to trace back the source of a bug. Though finding such bugs is hard, even with the best logger. You may be just overwhelmed by the amount 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 finds a bug.
- And finally, a logger may be helpful for the user to send in auto created error reports if something went wrong. He can just click a button to send in an error report with all relevant data and doesn't have to bother writing such a report by himself. 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.

39. Data files

There are several file formats to save data or use them as an interface. A lot of people apparently don't even know the most important once of them so I would like to give you a very short introduction.

The file formats that I used so far are CSV, json, XML, hdf5 and databases. Along with some custom file formats. There are of course many more as yaml, toml, etc. But for the sake of brevity I won't explain those.

The file formats mentioned here are sufficient to get your work done and it won't take much efforts to learn the other file formats if needed.

CSV

Comma Separated Values (CSV) is probably the simplest and one of the most common file format. You save numbers and separate them by commas or whatever other character you feel like. It won't get any easier. But this is also one of the weaknesses of CSV. In some natural languages, German for example, the comma character is used for the decimal separator. Thus you cannot use comma for separating different values as well. This overload of the comma character would cause serious problems when reading a CSV file.

There is no CSV file standard so you can do whatever you want. And that's at the same time the down side of it. People do whatever they want and for every file you have to write a new bit of code to read out the data. Saving auxiliary data is pretty much impossible in CSV files. CSV saves only plain, unstructured lists.

Long story short: CSV is the file format everyone uses who doesn't know anything better, like json for example. Or who works with Excel.

Here is some example code how to read a CSV file having to columns separated by commas and lines starting with # are comments:

```
def read_csv(filename):
    with open(filename) as file:
        x = []
        y = []
        for line in file:
            if line.startswith("#"):
                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 format for everyday purposes. Its libraries are very simple to use. It can save any data structure you want and is extremely wide spread and thus supported. There are libraries to automate the parsing of json files for all major programming languages. The output data structure is a mixture of nested maps and arrays. It won't get any easier to read a file into data.

Once you use json on a more serious project, you might want to use a schema to check your files for correctness. You may use different schema for different versions of your interface. And before you write a schema by hand, there are tools around to do it. You only have to make sure your json file contains all possible fields in order to get a complete schema.

Thanks to schemas, json is also a meta language. It is possible to define a general pattern of how the json file should look like. This defines a standard which enables easy file exchange between different projects.

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.

```
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

```
a = [1,2,3]
b = [4,5,6]
# write a and b to a json file called 'data.json'
```

Copilot makes the following suggestion:

```
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

```
# read the json file into a and b
```

Copilot makes the following suggestion:

```
with open('data.json', 'r') as f:
    data = json.load(f)
    a = data['a']
    b = data['b']
```

Thus one can say that Copilot allows us to save some time writing code and saving some brain memory cells 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 nicely as json. Otherwise there are only minor differences between the two formats. One thing people might miss in json is the possibility 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 like the XML supporting comments and namespaces.

Generally I prefer json over XML. If I have to read out an XML file, I use tools that convert the data structure into the json object and work with it the way I'm used to. Like this I don't have to learn something new.

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:

```
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.

```
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.

```
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.

```
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 deal with terabytes of data and optimized for high throughput. Pretty much all research facilities and companies dealing with huge amounts of data use this file format. It supports structured and auxiliary data. For looking at the data you either have to use the HDF5 library in your programming language of choice or download the free GUI software. Use HDF5 if you want to save several gigabytes of numeric data.

Working with HDF5 is in my opinion a tad less intuitive than working with json files. This is because HDF5 uses datasets that have to be created instead of just accepting a dict.

The following code saves a list of values inside 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 big amount of data that you want to analyze but doesn't fit into memory. Databases have a whole lot of different functionality that improves searching and manipulating data within the database. There are several vendors and 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 such that you can easily replace the DB by another one, or you'll be stuck paying hefty annual fees.

Also make sure a database is not the heart of your software. It's just a place to save and access data. It can be replaced by a text file if needed!

sqlite is probably the easiest to use database. That's why I make a short example here. Compared to other databases, creating or migrating an sqlite database is trivial.

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 format

Similar to the CSV file you can also define your own file format for other things than only numbers. You can define your own file with structured data. You can even define your own programming language like 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 want to be a serious software engineer you have to gain value for the customer. You have to use json or write a library for a normal programming language. There’s no reason to define custom file formats.

40. 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 with writing code right away when they have some task to do. And they postpone the whole infrastructure work for as long as they can. They keep compiling code with the command line for as long as they can. They don't use git. And they certainly don't use a Continuous Integration (CI) tool. This is dreadful. Set up these things right 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 used to it. But it is worth it. The very first reason why it is worth it is DRY. If you have to type in the compilation command to the terminal over and over again, you are repeating yourself time and time again. This is going to slow down the development process. This is way 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 need hardly any time. Already after the first time introducing some hard to track down bug you'll be glad having version control and being able to simply revert your last changes. The same holds for the build process. Typing in many long commands not only takes time, it 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 once you have to cooporate with other developers, there is no way around a proper version control software and a build tool.

Similarly for Continuous Integration (CI). It will take some time setting 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 master has been compiled and the tests pass without any errors.

So yes, setting up the infrastructure of a project may need some time. But it is certainly time well spent. There are so many advantages having a properly set up infrastructure:

- You anyway have to learn how to use git, cmake and all the other tools. So it's good practice to get started with them as soon as possible.

- You will save a lot of time down the road. This will outweigh the time needed now to set everything up.
- Having properly set up tools makes it easier for new team members to get started. They only have to clone the repo and run the build tool and they can get started.

Project folder

// Add a plot with the folder structure

Code is mostly a collection of text files. One question is: how do you deal with them?

The very first thing is the length of each file. Try to keep them short. About 100 lines per file would be great, a few hundred are kind of acceptable. Having many fairly small files improves the overview. Generally, one file contains either a class or a bunch of similar functions. Classes that have more than 1000 lines should have been broken into pieces a long time ago. For this reason, files should never have more than 1000 lines. In fact, files should usually be much smaller than this.

The way to arrange the files in folders depends on the programming language. The code is located inside the src folder, sorted by further subfolders if necessary. Generally, each subfolder corresponds to a library of the project. Make sure there is only your own code and, depending on the programming language, your tests in there. Nothing else. Do never ever allow any auto generated files inside your src folder. Auto generated files should never make it into the version control. They just polute it!

Generated files belong into the build folder. Like this cleaning up the build is quite simple. Just delete the build folder and all build files are gone. It also makes version control fairly simple. Add the build folder to the .gitignore file to make sure that generated files never make it into the 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 into the lib folder. They are not part of the git project, therefore the lib folder should be on the .gitignore file. You need some other way to manage them. If you use few libraries just manage them manually. In Python you can use the package management software pip. Togheter 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 by 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 for installation and build of the project. Getting the project, downloading the 3rd party libraries, building the project, running the tests and executing the project should all require only one single command.
2. The readme.md file shown on the front page of the git project. It usually contains installation instructions and a short description of the project. In fact, this book was also written as a readme.md file in a git project.
3. .gitignore is related with 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 miscelaneous files.

There are a few pitfalls how to arrange the files and folder of your project. But as long as you follow the general best advice you should be fine. Consult the wisdom of the internet for your programming language of choice.

41. Tools

"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'." - Linus Thorwalds

// I think this chapter needs some reworking. Or remove it completely?

There is a fair amount of software that is supposed to help you writing more or better software. Here is a list of the most important classes of tools I 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 Thorwalds programmed because all the alternatives were too slow for managing the Linux kernel or had other drawbacks like licencing 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. Start learning it as soon as possible. Every programmer has to be able to work with it. The only difference between companies is the way how they use git exactly. For example there are different ways how to deal with branches when merging them into master.

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. But also other pure text files are worth controlling with git. For example if you do research and have some files with measurement results. This could also be version controlled. The price you pay is negligible compared to what you gain by controlling all your files with git.

Or if you write a book like this one. It is written in Markdown and version controlled with git. This makes it easy to cooperate with reviewers and at the same time I always have a safety net when I screwed up some of my text.

I won't go further into details about git. There are plenty of tutorials in the web that teach you how to use git. They will teach you how to make commits, use branches and do merges. And remember as I told you in the chapter on unit tests: unit test are great if you have merge conflicts. The unit tests will tell you immediately if you 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 military knife of software development. It is the glue that connects all the different tools together. It enables us to automate all the build processes. For this reason, the command line tools are generally to be preferred over GUI based tools. GUI based tools like the file browser are great for getting started with some smaller projects, however you'll quickly reach some limits as they don't scale up on bigger projects.

The shell is an extremely powerful and versatile tool for executing other programs and running scripts for running all kind of commands dealing with configuration settings, the filesystem, networking, etc. It is certainly worth learning at least some of the basic functionality once you have the opportunity of automating 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 allows you to ask for specific changes to the suggestion made.

IDE

The Integrated Development Environment (IDE) is a class of software used for writing code. It's like Microsoft Word adapted to programmers. There are dozens of different IDEs available, both proprietary and freely available. I never cared about the IDEs. I just use what my work colleagues showed me: VS code. I don't think it's worth spending too much time here by yourself figuring out the details of one specific IDE. So I recommend you to do the same as I did. Just ask your friends.

In all up to date IDEs, there are plug-ins for most of the tools mentioned above. Ask your work colleague which ones you need. Spend a few hours with watching your colleague working in his IDE of choice to get an idea what the plug ins are used for. This is not wasted time. You also learn something about the code he writes during that time.

It is worthwhile learning some of the shortcuts in your IDE that allow you to modify code in different files faster. This is useful as it improves your work flow. But don’t overdo it at the beginning, you may be wasting too much time here. You can still learn more once you know how your personal work flow looks like. And if you really like to push the shortcuts to the limit, you’ll have to learn the VIM text editor which is operated by keyboard only.

Continuous Integration

"Continuous integration (CI) is the practice of merging all developers' working copies to a shared mainline several times a day." [https://en.wikipedia.org/wiki/Continuous_integration]

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 for Continuous Integration (CI) software. I don’t know the precise differences and probably you won’t have to neither. You don’t need this if you work alone and in any serious 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 at 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. But 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. Mostly setting break points, navigating through the code and looking at the stack trace. But generally, it’s a sign of bad code if you have to use a debugger too often. Write small classes and functions where you can tell exactly what they should do. Along with plenty of unit tests. Depending on the error you should be able to pin point the source to a certain class or area of the code without using a debugger. Anyway. Feel free to use a debugger, for example if you work with legacy code. But always keep the code quality high and make sure you need the debugger as little as possible. It's a useful tool, but it's a bad sign if you need it.

Profiler

A profiler lets you check the time required for executing each part of the code. Depending what kind of programming you do, chances are high you will never need one. Only run the profiler if the program is slower than it should be, which is probably not too often. Thus, the profiler is not a software you have to get acquainted with at the very beginning of your programming career.

Formatter

Pretty much all companies have a fixed ruleset 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 afterwards. If the formatter follows the wrong rule set you will have formatting changes in your merge requests. Which is absolutely terrible, because it’s hiding 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 kind 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 out dated 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 to be preferred. If you change the formatting rule set, run the formatter on all code and create a dedicated MR.

There are also some companies where every employee can use any formatting style he likes. Once an employee creates an MR, the official formatter runs over the code as a first step of the merge request. This offers the best of both worlds: the users are free to use whatever formatter they like but there is still some default formatting that won't affect the merge requests. The only cost is the work of the DevOps developer who has 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 Thorwalds (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 recommends also a line length of 80 characters, though it is less strict.

Code quality checker

There are different programs available that check your code on 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. Tough this metric shouldn’t be abused as a business metric. Use it to check 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 in the beginning but you get used to it and it will make your code better and prevent bugs. There is a reason why the compiler warns you about something and he does a better job with finding unassigned variables and other problems than you ever could. Why should you search for potential bugs yourself if the compiler can do the job automatically?

Pip, cmake

Some people may argue that pip and cmake don’t belong into this list. Of course, they are right. But I'm still feeling like mentioning a sentence or two about them.

Pip is the Python package management software [https://pypi.org/project/pip/]. As a Python developer it’s mandatory to know pip. It’s very easy to use. The command line pip install numpy will install the numpy library. Done. A little bit trickier is the handling of the virtual environment (venv) that allows you to install all the packages in a separate environment [https://docs.Python.org/3/library/venv.html]. This is useful if you have different projects that require different versions of the same library.

Cmake is the most commonly used build tool for C++. Meson is a more modern alternative you can use for new projects. Make is outdated. And don’t ever 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-man project as it helps organizing the work. And it doesn't even have to be a software project. You can also use it for all other kind of projects. If you don't work for a company there are also free alternatives. But I've never used any of them.

Wiki

Most companies use confluence as the knowledge base. Like Jira this is an industry standard. Write general thoughts and high level documentation here. However, things will go out of date quickly so be careful.

You may also write some high-level documentation of your code. But don't go too much into details. Low level details change too often and will get outdated and they should better be looked up in the code.

Again, there are free alternatives around, though you are unlikely to see any alternatives in professional environments.

Docstring

The docstring software auto creates a documentation depending on the comments in the code. It sounds like a nice idea, though it should be used scarcely. There is very little use of using docstrings for internal documentation as you can also look at the code instead. Instead docstrings should be used as a documentation for external APIs used by your customers.

Every programming language has one docstring tool. For Python it's Sphynx, for C++ it's doxygen.

Part 9 Collaborating

42. Working in teams

// https://github.com/97-things/97-things-every-programmer-should-know/tree/master/en/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. A great manager considers what is to be done and leaves the rest to his employees. He has faith in them. This is much more motivating than someone yelling around.

The times of the lone wolf programmers is over. Instead, you will spend most of your career working in teams. The story of the lone prodigy programmer in the basement is a myth. Modern programming is done in teams. Not only do programmers work with other programmers, but you'll also have to work with people from marketing and sales as well as customers.

Cooporating with other programmers has its advantages and drawbacks at the same time. Comparing programming in teams with a lone programmer is like comparing a parliament with a dictator. A parliament

requires more time to come to some conclusion, yet the solution is generally better than the 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 parliament.

Team structure

In most projects a team consists of roughly 4-12 software engineers, one project owner and one project manager. The software engineers are doing the real work. They work their ass off writing awesome code while everyone else is just slacking off. Just kidding. All other employees have work to do as well, it even if it might not be that apparent to the developers.

The project owner (PO) takes care of the tickets. He is at the interface between the project manager and the developers. The project manager (PM) is ... managing the whole project. He has to talk to customers and get an idea what they want. Or rather, what they need. These are not always the same thing. Nobody wanted an iPhone before it was released, yet still everybody needed one. And it's the PM's job to figure out such things.

It is important that everybody in the team talks to each other. Software engineers talk a lot about their code. But quite frequently they have questions about the ticket that the PM has to answer. Vice versa the PM wants to know the state of each feature for estimating the progress of the software.

The bus factor

The bus factor [https://en.wikipedia.org/wiki/Bus_factor] says how many team members have 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. People can get sick or they quit their job for whatever reason. And for a low bus factor, this may put the whole project at risk.

Make sure the bus factor in your project is as high as possible. Ensure that there is a good amount of knowledge exchange between the team members. Such that everyone knows something about everything. That the whole project does not stall only because a single person is ill.

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 hungry software engineers can do the job. That's at least their point of view. Now let's get a little bit more serious.

Software engineers have several tasks. The most obvious one is, of course, talking to each other. This is why any modern software company has a coffee machine with free coffee. You also have to discuss the tickets, you have to discuss how the fundamental structure of the code should look like and you have to talk during pair programming or code review sessions. Though the talking during code reviews is not absolutely necessary. Smaller MRs can be done with written comments, but when in doubt, it’s always better to talk to each other. Especially during times like Corona, the human touch got lost and things sometimes got hairy.

You also have to talk during pair programming. Both participants discuss together how the code should look like. This creates important knowledge transfer, especially if both participants are experts in different areas of the code. Both programmers learn from each other and the code quality improves. Code review in the MR is

no longer required. All together pair programming takes some more time than working alone but frequently this time is well worth it. Because remember: the most important resource in your company isn't code, but knowledge. And knowledge is gained by talking to each other.

Another fairly big job is going through Merge Requests (MRs), also refered to as Pull Requests. Everyone has to do it, no one really likes it. But it has to be. Nobody likes to wait a day until his code is approved and merged. Therefore reviewing MRs has to be done quickly. So get up, open the browser and select the first MR. And now... doing code review is a tricky business. You can somehow tell that the code is not good but it’s so... elusive. It’s your job to bring it to the point without being too picky. Furthermore, you see some code that had been there before and now it’s duplicated. It should be refactored. And a dozen of other things. Time to give the author of this MR a call. This can’t be resolved by writing comments.

Communication

As mentioned above, teamwork is a key element in modern software engineering. Without good communication skills, team work is not possible. So it is important to learn how to talk to other people. To learn about the flaws of humans and how to deal with 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. And if you think communicating with a computer is hard, think twice. For a computer you can just google what to do and most of the time it works. But with other humans it may take significantly more efforts for a good communication.

- Make sure you know what you are talking about and know how to talk to the specific audience.
- Keep communicating. Ask questions. Let the other person talk. Once you don’t get replies anymore you have an issue.
- Make sure the other person really understood what you were talking about.

There are probably hundreds of other rules, but these few here are the ones I know. Even though I’m not that good at applying them. As a developer you don’t have to know all these things. But if you want to manage people you have to get a feel for how to talk to others. Get some books or seminars about it, this is not the place to go into details.

Humans are all inherently flawed. They are insecure and try to hide themselves. They don't like to be criticized. They are scared because they are not a genius. But they don't have to be. Hardly anyone is a genius and most work is done by good, but not outstanding programmers. This fear, however, makes things worse. Because it is important to talk to other developers. Your team is much more productive if the team members talk to each other. If they are able to criticize each other in a constructive way.

It is ok to fail. Fail early, fail fast, fail often. Get feedback as early as possible and improve. This is the only way to make progress. Vice versa, you always have something to teach. Discuss with your colleagues and you can learn from each other. Don't be affraid, no body is perfect.

Don't come up with claims like "this is bad". Critisism is has to be constructive or else it is useless. Even worse, it can be regarded as a flat out assault. You have to be able to explain why something is bad such that the original author has a chance to improve.

In order to excel, humans need psychological safety. This requires 3 things: humility, respect and trust. Effective team work is not possible without these things. Discussions only work if all parties involved are treated equally.

Working with customers

// https://github.com/97-things/97-things-every-programmer-should-know/tree/master/en/thing_97

Customers are only humans. Quite frequently they don't say what they mean because they don't know it any better. Keep your vocabulary changing to figure out what certain words actually mean in the view of the customer. At times this reveals some misguided view. For instance if customer and client have some completely different meaning. Do not expect the discussion on requirements to be over after one meeting with the customer. You have to stay in touch in order to get constant feedback to make sure you implement what the customer wants and not what he says.

Frequently customers don't know what is important. Or at least things are important to customers that are not 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 really badly designed, the customer refuses to adapt. You really have to come up with some significant improvement that your version will be accepted.

43. 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 to improve the quality of the code. This does not work without some criticism, so it needs a little bit of intuition to know how to criticize the code without insulting the author. Most important of all, you have to critisize the code, rather than the author of it. But let's first have a look at how the whole code review process got started.

A long time ago, in a kingdom far away, software developers started cooperating. They shared their code. They started working on the same code. At the same time. And problems started creeping up. They needed some software to control the different versions of the code.

After some mediocre attempts to fix this issue there was our savior. Linus Thorwalds, the hero of every fanatic Linux developer, saved us by developing git. This solved the problem of version control software once and for all.

Unfortunately, git was not yet the final solution. It was still possible to write crappy code and merge it into the master. There was no other solution than firing this malicious developer.

But now comes the real solution: Merge Requests (MR) and code reviews. No user is able anymore to make changes on master all by himself. Before he can merge his changes into master, he needs to create a public request and wait for someone else to accept it. And thus the other developer allows the changes to be merged into master.

Now there are a few things to consider regarding code reviews. First of all, code reviews are great not only to keep the code quality up to date, but it also helps improving the programming skills of all developers. They are a great opportunity for knowledge exchange. Developers are obliged to look at each-others code and thereby learn a lot.

Drawbacks

However, there are some downsides as well. They can be severe enough that teams even stopped using MRs altogether. Most importantly, everyone has to stick to the rules. There is no way to prevent foul play by the developers sabotaging the system in a way that will render the MRs useless or even counterproductive.

The first problem is people just accepting merge requests without commenting anything, maybe even without looking at the MR. Either because they don't understand it, because they are lazy, they don't have time, or to make the author of the MR a favor. One would be better off not using MRs at all.

The second problem is speed. Speed is crucial. It is of utmost importance to check MRs as quickly as possible. Too long idle times for MRs lead to a very significant drop in the developers' productivity. Additionally, it is highly frustrating waiting for an MR to be looked at and not being able to continue working.

Another very serious problem are too long MRs. It is impossible to judge the quality of a change of a thousand, even a hundred lines of code. You should keep the tickets small. You should keep the commits small. And you should keep the MRs small. Huge MRs are a waste of time as no one understands what's going on. If a ticket turns out to be too long, split up the code in several MRs and make sure the tickets become smaller in the future.

There is a wide spread and very fundamental misunderstanding regarding MRs. Don't expect the referee to find bugs. This is in 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 most certainly free of bugs. MRs are more about the general structure of the code. And they are about knowledge exchange. The referee can only check that there is a reasonable amount of test coverage.

Always be polite. An MR is like criticizing someones code by email. This is a highly delicate thing to do. Stay professional and make sure you only comment the code and not its author. Once people start YELLING at each other in MRs it is high time to quit the job. Now things certainly deteriorated during the Corona virus pandemic when most developers had to work in home office. It takes some good team spirit in order to deal with written comments on MRs.

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 do pair programming, you may skip the code review phase all together as there were already two developers in agreement that the code is fine. Pair programming also allows your team to exchange knowlegde that would have to be done in the code review. This is one of the reasons why pair programming does not require twice the amount of time. The code review would take a considerable amount of time that will be saved with pair programming.

For teams with little experience, I think it's still important to make merge requests. The advantages outweigh the drawbacks in my opinion. However, only if everyone plays by the rules and gives fast feedback.

With very experienced programmers, on the other hand, one can skip the code reviews and just do some high-level discussion of the code instead. This is faster and usually does the job as well. Very experienced programmers only have to coordinate the high-level abstractions and don't have to review the low-level details. I hope your team gets to this state quickly as you can be so much more productive.

I generally recommend doing code reviews. But if code reviews become a nuisance, which they easily can you have to rethink the way you work and possibly find alternative ways to share knowledge about your code. Just don't forget that sharing knowledge is very important, but unfortunately also very expensive.

44. 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?

// Volker: kanban worked much better with real paper. Software has all kind of drawbacks.

// add the INVENT points from clean agile

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 all the complexity top down 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.

// write something about these values?

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. // add the bill of rights for the customers??

- 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.

Write something about the bill of rights?

Work 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.

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

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.

45. Requirements Engineering

Written together with Felix Gähler

46. 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.

47. 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.

48. 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.

49. 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.

50. 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 Dad():
    def eat(food):
        if food.name == "apple_pie" and food.flavor == Falvor.VERY_CREAMY
            print("I'm so happy")

class Flavor(Enum):
    VERY_CREAMY = 1
    SALTY = 2
```

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).

```
def create(food_name):
    food_dict = {"apple_pie" : ApplePie()}
    return food_dict[food_name]

class ApplePie():
    def __init__(self):
        self.flavor = Flavor.VERY_CREAMY
        self.name = "apple_pie"
```

Paint

Evans 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.

51. 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.

```
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.

```
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.

52. 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,

53. 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.

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.

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 they write tests before accepting 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

54. 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