# Summary of *Clean Code* by Robert C. Martin

Summary by Marius Shekow

## Introduction

Improving takes a lot of practice. On the one hand you need to acquire knowledge, OTOH you have to practice and apply it. Expect to fail applying the knowledge the first time. Just like it wouldn't work to learn riding a bicycle from reading the physics equations.
The book contains exercises to help the reader master their new skills.

## Chapter 1: Code

Code is the language that expresses the requirements of the program to such a detail that it can be understood by a machine.

### Bad code / technical debt

**Decline in productivity:** Even good programmers create technical debt, usually because of time pressure, which causes them to not take the time to clean up written code before moving to the next item on their list. As bad code accumulates, productivity of you and your team decreases. When managers add more staff to increase productivity again, this makes the situation worse, because the new team members don't understand the architecture (or don't take the time to understand it) and create code that waters down the architecture even more. Productivity keeps decreasing.

**Redesigns are hard:** If you already have a large bad code base and you are granted (by management) to do a re-design, this is often very hard, as the team will be split into 2 teams (one that does the redesign, the other maintains the existing system). These 2 teams are in a race against each other, and the re-design team also has to keep up with the new features that the system needs to have, which is very difficult.

**Good design turns into bad design quickly:** this can happen, e.g. because newer requirements don't match well with the existing architecture, or because of a harsh schedule. We could blame the customers or managers, but it's still our (the programmer's) fault, as they need to stand up for good code quality and demand for more time to keep the quality up.

### Clean code

There are many ways to describe features of clean code, here are some:

- Does *one* thing well
- Easy to read and understand (especially for others)
- Minimal to easy maintenance
- Have detailed error handling
- Is minimal (i.e. rather precise and short instead of long and verbose)

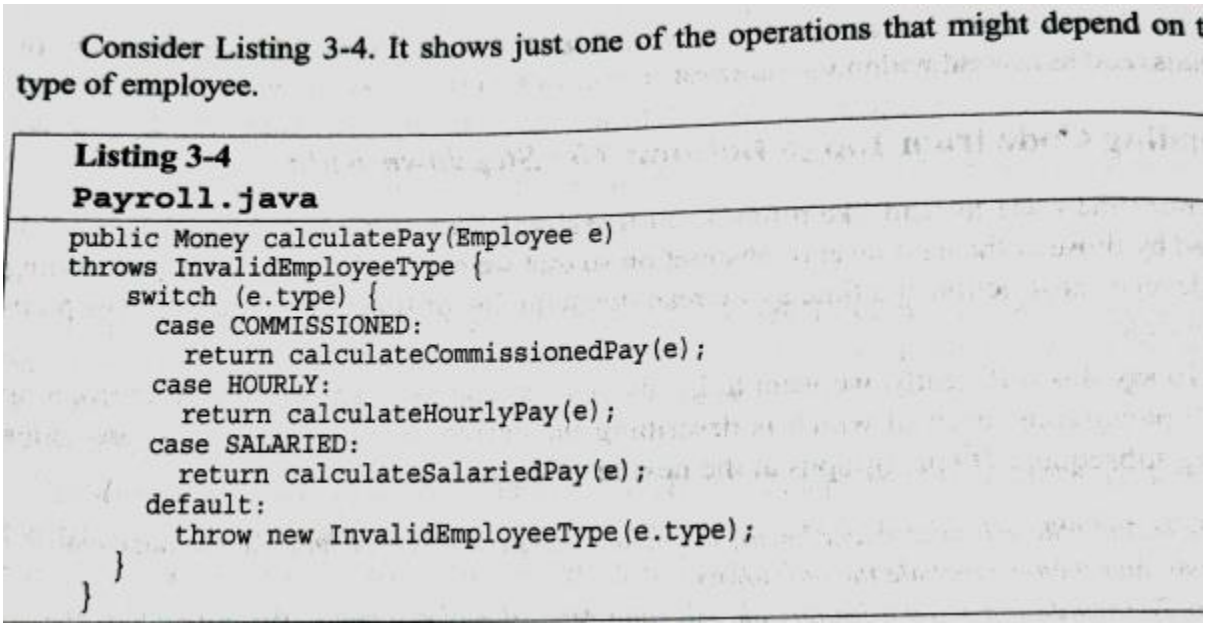
**Improving the code writing speed:** most of the time programming is spent on reading existing code, before writing new code (ration is ~10:1). You simply need the time to understand where to put new code, so that it is well integrated in the existing design and uses its pre-existing functionality. If you have spent the time to write clean, concise code, reading that code will also be faster, and you'll save time overall (even though writing clean code is generally taking a bit more time than writing sloppy code).

The chapter closes with the remark that this book is a school, just like e.g. the different martial art schools. Each one does things a bit differently. The authors of the book want you to read and acknowledge every described part, even if you don't agree with some of them (which is OK). They also want you to read other books about clean code, to broaden the horizon.

# Chapter 2: Meaningful names

- Reveal the purpose: what is the meaning of a variable? Unit of the variable (if applicable, e.g. seconds), avoid magic numbers, explain array/tuple index access (what does array[0] mean? Note that often Objects are better than arrays where each index has a different value with different meaning)
- Avoid misinformation: don't have variable names that are confusing because they are ambiguous, e.g. don't use "hp" for hypotenuse, as it could also mean "hewlett-packard"
- Use distinguishable names: e.g. copy(source, dest) is better than copy(a1, a2). Or don't create 3 classes such as Product, ProductData and ProductInfo, as a new reader will not know the difference between those 3 (doesn't matter if it's class name, variable name, function name).
- Use names you can can actually say out loud (especially abbreviations cannot be spoken out loud)
- Avoid encoding-conventions (e.g. ISomeInterface, AbstractFactory -> SomeInterface, Factory is sufficient)
- Use one word per concept: e.g. don't have mixed use of "retrieve_x()" and "get_x()" (just use either retrieve or get). Or don't have an Item*Manager* and List*Controller* (rather use: ItemManager and ListManager)
- Provide *context* for variable names: when reading a variable name for the first time, you should already understand what it is used for. E.g. the variable "state" could have many different meanings. The function in which it is declared *could* already make its context clear (e.g. consider a *state* variable in function evaluate_address(street, zip, ...)), but it might be better to either have a prefix (e.g. address_state) or, often even better, have a structure (such as a class Address) with the variables as members.
- Make variables names as short as possible (i.e. shorter is better than longer, but their meaning still has to be clear)

- Avoid large switch statements (or if-else blocks). Because if you have those, a function automatically has to do more than one thing. It's usually OK, however, to have such blocks, if they are very specific for the function they're in. But often they appear in the same structure in almost every function of a class, because what the function does depends on a common parameter or member variable. In this case it's better to use inheritance and its function-polymorphism feature and to have a factory that creates the correct sub-class object (using this switch-statement) in its makeObject() method. Here's an example:



Consider Listing 3-4. It shows just one of the operations that might depend on t type of employee.

**Listing 3-4**
**Payroll.java**

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
      case COMMISSIONED:
        return calculateCommissionedPay(e);
      case HOURLY:
        return calculateHourlyPay(e);
      case SALARIED:
        return calculateSalariedPay(e);
      default:
        throw new InvalidEmployeeType(e.type);
    }
  }
```

- Explain the intent of a conditional, i.e. the **…** when you write "if(**…**)". You can do this by extracting the intent into a function. E.g. if (this.accountHasExpired(user)) is better than if(user.isBlocked() || user.lastLoginTime() < currentTime() - 423423434)
    - Do this also with boundary conditions, e.g. a "currentIndex + 1" should be given a meaningful name.,

# Chapter 3: Functions

- Write *very short* functions. At most 20 lines. Better: just a few lines.
- Keep the indentation limit below 2.
- A function should do only one thing. This doesn't mean it should only have 1-2 lines, but it means it should include only one level of abstraction. Abstraction-levels are an intuitive concept, often indentations (due to loops and if-blocks) are an indication for a new level of abstraction. If a function includes multiple levels of abstractions, i.e. mixes high-level concepts with details, it is often hard for the reader to understand which parts are concepts and which ones are details. A function is doing more than one thing if you can extract another function from it, with a name that is not just restating its implementation.
- It's good practice to take time to choose a good name. Try several different variants, see which ones work best. Modern IDEs make it trivial to refactor function names. Precise

function names force you to implement the function to do exactly what its name says, thus improving quality.

- Be consistent in your naming. Use the same phrase structure, nouns and verbs where it makes sense.
- Think hard about its name: if you need to look at its documentation or implementation to understand what it does, you haven't chosen a good name!
- Parameters:
    - Have as little parameters as possible, at most 3. The more parameters you have, the harder it is for the reader to understand the function.
        - Try to **move some of the parameters to member variables** instead. Or create several functions, where each one has less parameters. Many parameters usually indicate that the function does more than one thing, which is bad (remember?). Especially **boolean-parameters are a good cause to be left out** (and the function be split into 2 functions).
        - Also you can often **summarize several parameters to an object** (create a class for it, or maybe one already exists?), **also leveling the abstraction level in the process** (e.g. make_circle(x, y, radius) becomes make_circle(point, radius) where point is a Point class with fields x and y).
        - In some cases it's better to have the functionality implemented in a member function of the class of one of the original function's arguments. e.g. the function Utils.append_to_buffer(MyBuffer, some_string) could be deleted, instead having MyBuffer.append(some_string).
        - Of course, **it can be OK to have more than 2 or 3 parameters, if the parameters are ordered components of a single value**. E.g. create_4d_vector(x, y, z, t) is perfectly fine. Once there are too many components, an Array as input may be more appropriate, though.
    - Make sure that all parameters have the same level of abstraction as the function itself.
    - Avoid output arguments (i.e. pure out- and in-out arguments). Use return values instead. If you must use in-out parameters, also return them, for the sake of consistency. Rather than having pure out-args, it's better to have outObject.change(...) rather than change(outObject, ...)
    - Choose the function names such that from reading them it becomes clear what is done with the arguments. e.g. the UnitTest function „assertEquals(expected, actual)“ would be better written as „assertExpectedEqualsActual(expected, actual)“
- A function should either *do* something (e.g. manipulate something), or *answer* something (e.g. read a value), but not both at the same time. If you violate it, not only does the function do two things, but it also won't be easy to understand from the function name (even if you choose a name that is as good as possible).
- Prefer throwing exceptions over returning status codes. There are 3 reasons. A) the function would both do something and answer something (violating the principle above), B) you're forced to check for the status codes in the code, creating poor quality nested if/else-statements. C) you cannot convey much information using an error code, whereas

with an exception, you have the possibility to attach a message, other data, and the stacktrace.

- Separate error handling from the actual code: try-catch blocks are confusing, because they mix error processing with normal processing. If possible, have minimal functions that have the form try-catch-finally, where in the try it just calls another (i.e. one) function which does all the normal processing (but doesn't do error handling). Keep in mind: error handling is “one thing”, normal processing another thing, so a function naturally shouldn't do both.
- Be aware of feature envy, while also being aware of the Single-Responsibility-Principle. This can be tricky, because you’ll sometimes have to apply feature envy in order to not violate the SRP. In other words, the SRP is more important than feature envy.

# Chapter 4: Comments

- Comments are generally considered bad, because it means that a concept or process is so complicatedly expressed in code that this code needs many explanatory words. So you should generally try to express yourself in good, understandable code. The main problem is that comments become out of sync with the code very soon, it's impossible even for good programmers to maintain their code. Once out of sync, they provide misinformation for the reader.
- Concrete examples where comments are bad:
    - Enforcing that every function has to have a doc-block, just for convention's sake.
    - Restating the obvious
    - Making super-smart one-liners (in code) that need a comment for explanation → rather turn the statement into multiple lines that can be understood without any comments
    - Commented-out code: just delete the code, it's not gone as the SCM can still recover it
    - Comments that are directly related to the code beneath it, but there are still open questions (or the comments even raise new questions)
    - Comments with information that should better be stored elsewhere, e.g. changelogs of a class
- There are of course also good examples for comments. Here are some examples:
    - Doc blocks of abstract methods (so that the implementers know what to do) or public APIs
    - Explanation of a decision (i.e. when from the code it becomes clear **what and how** is being done, but the comment explains **why** that is being done). For example, often it seems that some code is inefficient or poorly structured (but it is done so for a good reason, e.g. because you use some library that has bugs and thus calls on it have to be done in a weird way). Comments can be left to avoid that the next programmer refactors/changes things.
    - However, always keep in mind that these comments can also become out of date
- Generally, take time to write good comments. Like for good code, it’s a good idea to “refactor” comments (i.e. rewrite them)

# Chapter 5: Size and formatting

- Regarding code style, your project should have an automatic tool with preset rules that ensures that the code is formatted appropriately. All team members must use these tools.
- Rules regarding lines of code: as a rule of thumb, have at most 500 lines of code per file. The longer a file (and, thus, usually, a module/class), the harder it becomes to understand it.
- Also, a good indicator that a class is too large is when you have too many member variables.
- Structure each file like a newspaper article: at the beginning you read the broad rough concepts, like a synopsis, followed by the details.
- If function A calls function B, have B declared after A.
- Constants: often constants are part of a high-level concept, rather than low-level concepts. They should generally be declared/written at the level of abstraction to which they belong (even if they are actually used/read in lower-level code → pass the default values down as parameters).
- Have those functions that do similar or very related things close to each other in the file.

# Chapter 6: Objects and data structures

- Data abstraction: make sure that you don't expose private member variables to the public. But also don't blindly add simple getters/setters. Instead, separate the high-level concept of data from the internal implementation and design public methods that provide an interface only for the high-level concepts, not their implementation.
- The rules are different for **data structures**, i.e. classes that have no behavior but just store data. It's not a bad idea to actually have the members to be public, rather than confuse things with getters, because getters blow up the number of lines of code and suggest that the members are internal and subject to change all the time, even though they might not.
- Adhere to the **Law of Demeter**: a function *f* inside a class *C* should only call methods on objects that are returned by any of the *allowed functions*. The *allowed functions* are: functions of C or functions of member variable objects of C, functions of an object *created* within *f*, functions of objects passed as parameters to *f*.
    - This principle is also referred to as "talk to friends, not to strangers".
    - Specifically this means you should not do stuff like someObj.someFunc().someOtherData() - this is also referred to as **train wreck**. It violates the Law of Demeter and is also incomprehensible (such statements

should be broken down into multiple lines so that each line gives a meaningful name to each of the returned objects of each function call).
    - Note that there is an exception where train wrecks can be OK: when the object is a data structure object.
    - A fix for the train wreck above could be to create a function like getSomeFuncsSomeOtherData() for the class of someObj. But that's not good as this can explode the number of methods in someObj. Often it's better to think about what is *being done* with the data that is actually returned by someOtherData() and add a method to someObj that already does this thing internally. This generally avoids **feature envy**.
- **Avoid object/data-structure hybrids**, i.e. classes that are a bit like data structures (because they have public members) and a bit like classes (because they have methods that do much). This indicates a muddled design.

# Chapter 7: Error handling

- You generally want to write code that is robust, i.e. has detailed error handling, but where the logic isn't tangled with error handling (i.e. you don't want error handling code to obscure your logic).
- One way is to have a higher-level method such as someOperation() which has a try-catch statement, in the try it calls "trySomeOperation()", which internally has reduced (or even no) error handling (because exceptions will interrupt the program flow anyway). The "try" in the function name already indicates that the function might not successfully complete this task.
- Use unchecked exceptions! Checked exceptions are bad, because imagine you have a method that throws a checked exception, but error handling is done in a method that is 3 stack-levels above that one. All intermediate level methods (which don't handle that exception) consequently have to also check-throw this exception. If you change the checked exception in the low-level method, you also need to change the signatures of all intermediate level methods (and recompile/redeploy them) which is very bad!
- Provide context information as part of your exceptions. What went wrong? Where did it go wrong? Is sufficient information for logging available?
- When using 3rd party libraries (which defines its own exceptions), it's a good idea to have a wrapper class around it. This one not only allows you to create your own API (making it possible to swap the concrete 3rd party libs in the future without having to rewrite the code that uses it), but you would also wrap 3rd party lib exceptions in your own exceptions. In the process, you can also improve the exception structures (and their member variables).

- When you have to decide between an exception hierarchy or just maybe an enum field that details the type of exception, then use a hierarchy only if you want one exception while allowing the other(s) to pass through.
- Improve special-case treatment (Fowler's *special case pattern*):
    - Bad practice: sometimes you will call functions and you know beforehand that this function might fail (raise an exception or return null for example), and in this case, you also know what to do instead (e.g. calling another function for this special case). This special treatment is sometimes not unique, i.e. is done in many different places → you have duplicated functionality in different places
    - Better practice: have the function that does the normal case already handle the exception/returning null *internally*, so that it then also does the special case for you.
- Avoid returning null: because it forces the calling code to check for null, which makes its code bad. Instead, consider to either return a special-case object instead, or throw an exception.
    - If null is returned by 3rd party code, consider wrapping it with a function that throws an exception or returns a special-case object.

# Chapter 8: Boundaries

- Boundaries exist between your code and 3rd party code that it uses.
- It's generally a good idea to create wrappers around 3rd party code. Sometimes that 3rd party code is not an external dependency, but part of the core library of the programming language you're using, in which case it may also make sense to wrap it. The reason is that 3rd party code tends to be generic and has a lot of features (to be as generically usable as possible), many of which our code doesn't need (or it would even be bad if every place in the code had access to this functionality).
- As it is often hard to figure out how a 3rd party lib works, it may be beneficial to create so-called *learning tests*, i.e., unit tests that encapsulate your lessons learned about the behavior of those library functions you will actually use in your code. These tests also discover once the behavior changed. You can simply rerun them whenever there is a new version of the library.
- Sometimes you want to use code that doesn't exist yet. Or several different alternatives might exist and you still need to pick one. That is, the direction is flipped: not from implementation to high-level API, but from API to implementation. What you can do then is to create your self-defined API and then have an adapter/bridge that implements that interface and uses that code, at a later point of time.

# Chapter 9: Unit tests

- Some thoughts about TDD:
    - The idea of TDD is to write tests timely, i.e. before / along-side the actual code. This forces you to design the production code so that it is testable. If you only

write testing code after the production code is written, you might be lazy and decide that some production code is too hard to test.
    - It is correct that TDD will result in very many tests, and that maintaining these tests will be as much work as maintaining the actual code. But in the mindset of a TDD-team, test-code is just as important as production code. Its great advantage is that you can be sure that changes to your code base work as expected. You will be sure that changing one part of the system won't break another part. Nothing stops you anymore from readily changing production code, because you can be sure that bugs will be caught. Maintaining testing code is a lot of work, but once you let it rot, your actual code will also soon start to rot.
- Test cases have to be just as readable as your actual code, so that they can be understood (and modified) quickly. Just like with production code, you usually won't come up with nicely readable testing code right away. Instead, you first write code that gets the work done, and then you refactor it.
- It's fine if the test code isn't very run-time efficient, because CPU and memory is usually not restricted in a testing environment. You should prefer code that is easy to read over worrying about performance.
- Make sure to test only one concept per test. It's not a good idea to test two or more different things in a testAandB() function, as it reduces readability.
- Tests have to be able to run independently, in any order, and should not depend on each other. Otherwise, when the first test fails, also all other dependent tests fail.
- Testing becomes difficult when there is tight coupling between the component you want to test, and other components that this component needs. Therefore it's a good idea to reduce coupling by using some kind of dependency injection mechanism for the tested component (e.g. replacing the coupled components with Mock components)
- When you've finished writing functionality, check again that its tests cover every *boundary* case.
- Don't simply disable non-running tests. Tests are a safety mechanism, and you shouldn't disable them. You risk forgetting to enable the tests again.
- Use a coverage tool, e.g. in your IDE, to make sure every line has been covered.

# Chapter 10: Classes

- Classes should be small, however, don't only use lines of code as measure, but also the number of *responsibilities*. Another definition for responsibility is *"reason to change [the module]"*.
    - Actually, the class name, or a short description (e.g. 25 word) of the class is a good indicator for this: if you can find a concise name for a class and a short description for it, where both don't contain "and"s or "or"s, you're on the right track.
- You should prefer many smaller classes than a few larger ones. Each small class has one responsibility and collaborates with a few other classes to achieve this task.
- Another great indicator for when to split a class into several smaller classes is *in-class cohesion*. You have a high in-class cohesion if every member variable is used by *every*

method of the class (this is not possible in practice, it's enough if the member variables are used "just" by *many* of the methods). Classes with a *low* cohesion often have a structure where a *subset* of methods uses a certain subset of member variables. This is a good indicator to split the class, putting these subsets of member variables and methods into their own class.
    - A real-world example where this often applies is when you want to break down a huge method. This method will have many stack variables that are used all over the place in that method. When creating multiple smaller functions, you would not pass these variables as parameters (it would look messy) but you would turn these stack variables into member variables, so that the parameter-signature of all those small methods is concise. But then you'll end up with a subset of methods (those you just created) using a subset of member variables → consider creating a new class for the job.
- Be aware of object-oriented-design principles such as OCP (Open-Closed-Principle) or the Dependency-Inversion-Principle). These are explained better in other books such as https://www.amazon.de/Software-Development-Principles-Patterns-Practices/dp/0135974445
- Maintain a proper separation of the levels of abstraction, also at class-level. That means to split the classes into abstract classes (or interfaces) and implementations, where appropriate. When doing so, check again that you didn't accidentally include methods or constants into a high-level class but should actually belong into an implementation.
- Avoid clutter, e.g. unused variables or functions or empty default constructors.
- Avoid coupling where it is not needed: e.g. often you declare enums, inner classes, variables or constants inside classes where it is convenient to declare them, because you're in a hurry. They are coupled to the class in which they are declared. Often, they don't really belong there, but should be declared somewhere else, independently.
- Use enums instead of a collection of static (integer) variables. Since enums are classes in most programming languages, you can also extend them with some simple behavior. E.g. imagine an Enum "Pressure" with values "HIGH" and "LOW". Then you could also have a method float getPressureLevel() that returns a value for each.

# Chapter 11: Systems

- The system level is the highest level of abstraction of your system. At this level (and also lower levels) you have to make sure that you have distinct components whose *concerns are separated*.
- Don't expect to be able to achieve a perfect design initially. BDUF (Big Design UpFront) is not a good way to go. A) because you can never anticipate all aspects of the system perfectly anyway, and B) because it reduces your willingness to change the design once it exists, because you've already put so much work into it.
- One important question is whether it's actually feasible to slowly adapt/change the overall architecture of a system over time, or whether you always need complete rewrites. The authors claim that slow adoptions are feasible, given that you have well separated the concerns within your system.

- One concern that exists in every system is its initialization. A kind-of bad example are methods like
  ```
  Service getServer() {
    if (null == service) service = MyServiceImpl(some, params);
   return service;
  }
  ```
  Because we have mixed the concerns *runtime operation* with *initialization*. The method actually has two responsibilities (choosing which specific service to construct, and returning the service). Usually, such methods, i.e. the place where objects are created, are scattered all over the system, and therefore also initialization is scattered.
    - One way to fix this is to have your system entry point (e.g. **main()**) use Builders to create configured objects or configured factories, and then hand these to the run() method that actually performs the execution of the system. Configured factories are a nice thing, because they make it possible that the runtime-code decides *when* to create objects, but how these are built is already pre-determined.
- Consider using AOP (Aspect Oriented Programming) if your programming language supports it. It enables you to have a separation of *cross-cutting* concerns, such as logging, security, caching, fail-over/retry, etc. in a central place, rather than spreading them all over your modules.

# Chapter 12: Emergence of simple design

- The goal: achieve and keep simple/good design over time. According to Kent Beck's book "Extreme programming explained" there are 4 rules to follow to achieve this (rule 1 is most important, rule 4 least important).
- Rule 1: Use testing, as it makes sure your system runs as intended when applying the other rules (where we change code and we need to be sure the changes don't break the system)
- Rule 2: Avoid duplication. Regularly try to spot duplication, i.e. equal or similar lines of code that appear multiple times. The **template method pattern** is one of the examples of how this can be achieved. I.e., you have an abstract base-class with an implemented method (the *template* method) which knows the algorithm and its steps (abstract or implemented functions, depending whether they're always the same for all subtypes or not), but (at least some of) the sub-functions are implemented in sub-classes.
- Rule 3: be expressive. This basically means the accumulation of the tips above, mostly chapter 2+3. The reader shouldn't be surprised when reading class/method names and discovering what their responsibilities are. Also, when using design patterns, you can sometimes use the pattern name in the class/methods that implement it, e.g. XyzVisitor.
- Rule 4: Don't blow up classes and methods: be cautious when splitting functionality into classes and methods. The system will blow up if you e.g. create interfaces for almost every class just for the eventuality of future extensibility, or having very many small methods unnecessarily.

- Rules 2-4 are achieved through regular refactoring. Whenever you've written some code, immediately after, ask yourself whether it has degraded the existing design and whether the test code is sufficient.

# Chapter 13: Concurrency

- Concurrency decouples *what* is done from *when* it is done.
- Keep concurrency-related code separate from the other code. I.e., have classes and methods that are oblivious of threading, and then wrapping classes that deal with all the concurrency stuff. This also makes it easier to test each aspect in isolation, and to replace e.g. just how the concurrency/threading stuff is done with a different implementation (e.g. replacing the simple thread-creation scheduler with a thread pool executor scheduler).
- Reduce the number of concurrently manipulated/accessed variables to a minimum. You'll often forget to properly protect the read/write access, or have it in too many places (--> duplication) for good measure.
    - Solution 1: use strong data encapsulation, single out the points where read/write protection takes place. This is also called *server-based* locking.
    - Solution 2: create copies of data: often the costs of creating a copy is smaller than the cost of e.g. the mutex/lock mechanism.
    - Solution 3: don't even have variables that are shared among multiple threads, but try that each thread has its own set of data to operate on. As with solution 2, this may require copying of data.
- Know your runtime library / language: find out what mechanisms regarding concurrency the language (or 3rd party libraries) offers. For instance, what multi-threading/processing options are there? What thread/process-synchronization methods exist? Are there thread-safe data structures, and which ones aren't thread-safe?
- Be aware of the 3 most common concurrency patterns and know their solutions (often several solutions exist for each case, with different advantages/disadvantages)
    - Producer-Consumer pattern: 1 or more producers produce tasks and put it in a queue (unless that queue is full, then they wait), 1 or more consumers consume and work on tasks from that queue (unless it's empty, then they wait). Signalling is used to wake up the respective other party.
    - Readers-Writers pattern: One shared resource is concurrently used by writers (modifying it, takes time) and readers (reading from it, which also takes time). Just using a single mutex for all readers and writers is inefficient, because readers should be able to read in parallel. More involved solutions exist, some are in favor of the writers (w.r.t. speed and avoiding starvation), some in favor of the readers, and some try to balance starvation among readers and writers.
    - Dining philosophers problem: When each thread needs not just one but multiple shared resources. In such a case, deadlocks occur, thus you need to implement a strategy to avoid that this happens.
- Specific bugs that often occur
    - You assume some operation is atomic, although it's not (e.g. **x++**)

    - 2-stage-access: you modify a resource x by reading from x and y (the individual read operations are already guarded with a mutex). But then x or y change between the 2 reads and the modification/write. → make sure the read**+write** operations are both in **one** critical section.
    - Initialization sleeps: some component sleeps for a randomly chosen amount of time (because it needs to wait for some other component to become ready) → you should rather rely on *signaling*.
    - Losing notify: the notify() is “lost” because it occurs before the thread performs the wait() call → fix by: the notifying code needs e.g. a method condition_was_met() and the waiting code should have a block like while(not obj.condition_was_met()) obj.wait(some_max_time)
    - Indefinitely blocking critical sections: e.g. if you call your own (or 3rd party code) within a critical section, these calls may never terminate, blocking the critical section indefinitely.
- Testing concurrent code:
    - Make sure to repeat tests (e.g. also have the same test run in a loop many times)
    - Run tests on different platforms and under different system loads
    - Treat “spurious” test failures (e.g. if a test fails in 1 of 100 or more executions) as very serious - they **are** bugs after all! Don’t just treat them as “one offs”.
    - Create tests for your non-threaded code first, i.e. test the functions that are called by a single thread anyway.
    - Make threaded code pluggable, e.g. allow to specify the number of used threads, or have pluggable mock objects (used within the threaded code) where you can configure the speed of execution (e.g. simulating a very slow database).