Software Craftmanship

Los Del DGIIM, `losdeldgiim.github.io`

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Software Craftmanship

Los Del DGIIM, `losdeldgiim.github.io`

Arturo Olivares Martos

Granada, 2025

Índice general

1. Clean Code

We will focus on code, as we don't only want to write good code, but also tell the difference between good and bad code. However, we should firstly assume some claims:

- There will always be code

  In a system, there will always be requirements (as with no requirements, the system would not be needed). Code is simply a high specification of those requirements. In addition, given that we will have programming languages with high abstraction levels and that we will use AI, code will not disappear.

- Code is written for humans

  Code is almost never just written once and then forgotten. We should always assume that code will be read by other people, and thus, we should write it in a way that is easy to understand.

- Bad code will bring your company down

  If bad code is written, the cost of solving bugs will increase, and thus, the company will lose money. Code smells, which are indicators of bad code (they may not always be bugs), may include duplicated or commented code, long methods, or large classes.

- Start from the scratch will not fix the problem

  When bad code becames so unmanageable, it may be tempting to start from scratch. However, this is not a good idea, as it will take a lot of time and resources, it may not solve the problem, and it may even lead to two systems developped in parallel, which will be a nightmare to maintain.

  In order to solve the problem, code should always be kept clean, following the *Boyscout Rule*: "Always leave the code cleaner than you found it".

Once we have assumed these claims, we can start talking about clean code. Even though there is no single definition of clean code, it should just be as easy, minimal and *simple* as possible. It should clearly express its intent. We will focus on some points that will help us to write clean code.

1.1. Meaningful names

We name a lot of components in our code, such as variables, functions, classes, etc. We should always try to give them meaningful names, as they will help us to understand the code.

1. Use intention-revealing names.

   `int x;` is not a good name, as it does not reveal the intent of the variable.

2. Avoid disinformation.

   We should not misuse known abbreviations, imply data types, or use similar names for different things.

3. Make meaningful distinctions.

   We should not use the same name for different things, as it will be confusing. What is the difference between `a1` and `a2`? We should use names that clearly distinguish between different things.

4. Avoid encodings

   Even though it was done in the past, type of scope information should not be encoded in the name, as it should nowadays be provided by the IDE.

5. Regarding classes:

   Nouns should be used for classes, while verbs should be used for methods. Accessors (`getters`), mutators (`setters`), and predicates should be named accordingly, as they are very common and should be easily identifiable.

6. Solution vs Problem Domain names

   Code that is related to the problem domain should be named accordingly, while code that is related to the solution domain should be named in a way that reflects its purpose.

7. Use short, pronounceable names.

   A name should be as short as possible, as long as it provides enough context.

Finding good names is not easy, but it is worth the effort, as it will make our code much easier to understand and maintain.

1.2. Functions

Functions are one of the most important components of our code, as they are the building blocks of our programs. We should always try to write functions that are clean and easy to understand. Some guidelines for writing clean functions are:

1. Small functions.

   Functions should be small (ideally, 3 lines or less), as they are easier to understand and maintain. If a function is too long, it may be a sign that it is doing too much and should be split into smaller functions.

   This may lead to a lot of functions, aspect that may be criticized and that may even impact performance. Therefore, each one should find the right balance between the number of functions and their size.

2. Do one thing.

   A function should do one thing and do it well (Error handling is needed and is not considered as doing more than one thing). If the developper is tempted to write a comment before a block of code, it may be a sign that the function is doing too much and should be split into smaller functions.

3. One Level of Abstraction per Function.

   A function should not mix different levels of abstraction. Each time a function is extracted, it should be at a lower level of abstraction than the one that calls it.

4. Regarding arguments:

   a) Low number of arguments.
   Functions should have a low number of arguments (ideally, 0 or 1), as they could be messed up when calling the function. If too many arguments are needed, maybe they should be encapsulated in an object.

   b) Output arguments should be avoided.
   It is assumed that arguments flow in the function, not out of it.

   c) Flag arguments should be avoided.

5. They should have no side effects.

6. Command / Query Separation.

   Functions should either do something (command) or answer something (query), but not both. If a function does both, it may be a sign that it is doing too much and should be split into smaller functions.

7. DRY (Don't Repeat Yourself).

8. Exceptions should be used instead of return codes.

9. Multiple `return` statements are acceptable.

   Functions should ideally follow a Strict-Structured Programming style, which means that they should have a single entry and a single exit point (only one `return` statement, no `break`, `continue`, or `goto` statements). However, in some cases, it may be acceptable to have multiple return statements, as long as they are used in a way that does not make the code harder to understand.

1.3. Comments

Comments are a necessary evil. They are really extendedly used, but they are not always good. They may be outdated, they may be misleading, and they may even be used to explain bad code. Bad comments lead to people ignoring them, and thus, they may even hide important information.

The programmer should explain himself through the code, not through comments. Some bad examples are:

- `int d; // elapsed time in days`
- Noise/Redundant Comments, as

  `int calculatePay() { // calculate the pay`
- Using a difficult condition and then explaining it with a comment, instead of just using a bool variable with a meaningful name or a function that encapsulated the condition.
- Comments that try to explain bad code, instead of just refactoring it to make it easier to understand.
- Mandatory comments, which are required by the company or by the project, but that do not provide any useful information.
- Non-Local information

  Comments should not provide information that is not local to the code they are commenting, as it may be easily forgotten and thus, lead to confusion.
- Comments of different sections of a function

  If a function is too long and it is divided into different sections, it may be a sign that the function is doing too much and should be split into smaller functions.
- Closing brace comments

  When a function is too long, it may be difficult to know which closing brace corresponds to which opening brace. However, if the function is too long, it should be split into smaller functions, and thus, closing brace comments should not be needed.
- Journal comments & Attributions

  The VCS should be used to track changes and authorship, not comments.
- Commented-out code

  Again, the VCS should be used to track changes and to recover old code if needed, not comments.

However, there are some good comments, such as:

- Legal comments, which are required by law or by the company.
- Explanation of intention

  Sometimes, a decision is made in a way that is not obvious, and it may be worth explaining the intention behind it. People may not agree with the decision, but at least they will understand it.
- Warning of consequences

- Amplification of a point in the code that is not obvious.

  People may not understand why a certain point in the code is important, and it may be worth amplifying it with a comment. If not done, another developper may change that point in the code, not understanding its importance, and thus, breaking the code.

- Comments to create Public API documentation (e.g., Javadoc).

Lastly, there are some comments that are not good, but that may be necessary, such as:

- Clarification of a point in the code that is not obvious.

  Sometimes, a point in the code is not obvious, and it may be worth clarifying it with a comment. For instance, when a function returns a value that is not obvious, or when a complex regular expression is used, it may be worth clarifying it with a comment. However, it is risky because if the code is changed, the comment may become outdated and thus, misleading.

- TODO comments, which indicate that something needs to be done in the future.

  They may be used to ask a colleague to work on something, a reminder to change a part that depends on something else... but never to clean up bad code later.

  Modern IDEs have tools to track TODO comments, but they should be used with caution, as they may be forgotten and thus, lead to technical debt. The Boyscout Rule should be followed.

On conclusion, comments should be used with caution, as they may be outdated, misleading, and even hide important information. The code should explain itself, and if a comment is needed to explain a point in the code, it may be a sign that the code is not clean and should be refactored to make it easier to understand.

2. VCS With Git

In this chapter, we will focus on the insights of using Git as a Version Control System (VCS). However, a depper understanding of the concepts behind Git can be found in the Chapter 2 of the contents of the Application Management course.

Definición 2.1 (Version Control System)**.** A Version Control System (VCS) is a software tool that helps manage changes to source code over time. It allows multiple developers to collaborate on a project, track changes, and maintain a history of modifications.

Git is a distributed version control system that provides a powerful and flexible way to manage code changes. There is a lot of reasons why Git is widely used in the software development industry, including its speed, efficiency, and support for branching and merging. Even when someone works alone on a project, using Git can provide benefits such as keeping a history of changes, allowing to revert to previous versions, and facilitating the integration of new features or bug fixes.

2.1. Git Data Model

Git is based on three different types of objects: blobs, trees, and commits. All of these objects are identified by a unique SHA-1 hash, which is generated based on the content of the object, and saved in the `.git/objects` folder.

1. Blobs: A blob (binary large object) is a file that contains the contents of a file in the repository. It does not contain any metadata, such as the file name or permissions.

2. Trees: A tree is a directory that contains references to blobs and other trees. It represents the structure of the repository at a given point in time. It also contains metadata, such as the file name and permissions.

3. Commits: A commit is a snapshot of the repository at a specific point in time. It contains a reference to a tree, which represents the state of the repository at the time of the commit. It also contains metadata, such as the author, date, and commit message.

   In order to refer commits in a more human-readable way, Git uses references, such as branches and tags. The most important ones are:

   - `HEAD`: A reference to the current commit that the user is working on.

- `master` or `main`: The default branch in Git, which typically represents the main development line.

Commits always have a comment. There are good and bad practices for writing commit messages. A good commit message should be concise and should describe the changes made in the commit.

Another important concept in Git is commit granularity. Although some people have a different opinion on this topic, it is generally recommended to have small and focused commits that represent a single logical change. This makes it easier to understand the history of the repository and to revert changes if necessary.

In order to save the history of changes, Git uses a DAG (Directed Acyclic Graph) data structure. Each commit points to its parent commit(s), creating a chain of commits that represents the history of the repository. This allows Git to efficiently manage and track changes over time, enabling features like branching and merging.

Lastly, the idea of branches in Git is fundamental to its workflow. A branch is a pointer to a specific commit, allowing developers to work on different features or bug fixes without affecting the main codebase. Branches can be easily created, merged, and deleted, making it a powerful tool for managing parallel development efforts.

2.2. Working with Others

Git is a distributed VCS, so it uses a centralized collaboration. This means that there is a central repository that serves as the main source of truth for the project, and developers can clone this repository to their local machines, make changes, and then push those changes back to the central repository. This central repository can be hosted on platforms like GitHub, GitLab, or Bitbucket, which provide additional features for collaboration, such as pull requests, code reviews, and issue tracking.