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, 2025-2026

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. It is true that we will have programming languages with high abstraction levels and that we will use AI, but code will still be there.

- 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) should always be addressed.

- Start from the scratch will not fix the problem

  When bad code becames 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. According to Ward's principle, clean code always behaves as expected. 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 or imply data types.

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. 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, misleading or 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.

3. Test Driven Development

This chapter covers the concept of *Testing*. Given that the goal of a developper is achieving clean code that *works*, testing is a crucial part of the development process. There are however two main approaches to testing: *Debug-Later Development* and *Test Driven Development*. The first one is the most common, but the second one is the one that leads to better code quality. In this chapter, both approaches are explained, and the advantages of TDD over Debug-Later Development are shown.

3.1. Debug-Later Development

As the name suggests, Debug-Later Development is the approach where developers write code without testing it, and then debug it later when it doesn't work. The problem is that a new bug may be found at any time, forcing us to go back and forth between writing code and debugging it. In order to explain this concept, the following time definitions are used:

- Time until discovery (T_d): the time it takes for a bug to be discovered since it was introduced.
- Time until found (T_{find}): the time it takes for the reason for the bug to be found since it was discovered.
- Time until fixed (T_{fix}): the time it takes for the bug to be fixed since its reason was found.

This approach has several disadvantages:

- T_d and T_{find} are usually very long, because the code is not tested until it is complete, and the bug may be found at any time.
- The longer a bug is present in the code, the more likely it is to cause other bugs and the harder it is to fix it. Therefore, T_{fix} is also usually very long.
- Clean code is harder to achieve, because developers are more focused on making the code work than on making it clean.
- Testing is usually forgotten or neglected, as developpers think that their code works, and they don't want to spend time writing tests for it.

This is the reason why Debug-Later Development is not a good approach. Test Driven Development covers all these disadvantages.

3.2. Test Driven Development

Test Driven Development (TDD) is a development discipine (not a testing technique) where developers write tests before writing the code that makes the tests pass. It helps sequentially focus individually on working code and clean code, and it leads to better code quality.

3.2.1. The Three Laws of TDD

The Test Driven Development process is based on three laws, which garantee that the code is always tested:

- First Law: You are not allowed to write any production code unless it is to make a failing unit test pass.
- Second Law: You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

  This way, as soon as the new test fails, you must stop writing the test and start writing the production code to make it pass. In addition, as there is no production code yet, not compiling will happen very soon.
- Third Law: You may not write more production code than is sufficient to pass the currently failing test.

This leads to three main benefits. Firstly, only what is strictly necessary is coded. Secondly, bugs are found and fixed as soon as they are introduced (T_d, T_{find} and T_{fix} are reduced), which makes them easier to fix. Finally, tiny steps are always made towards the goal, which makes it easier to achieve clean code.

3.2.2. The TDD Microcycle

The TDD process can be summarized in a microcycle, which is a sequence of steps that are repeated every few minutes until the code is complete:

1. Write a small test (as small as possible).
2. Compile all the tests and see that the new test does not compile.
3. Write the production code to make the test compile.
4. Compile all the tests and see that the new test fails.

   We should not assume that the test fails, as it may pass for the wrong reason (for example, if the test is not written correctly). If we assume that the test fails, we may code with a false sense of security.
5. Write the production code to make the test pass.
6. Compile all the tests and see that they *all* pass. It is important to check that all the tests pass, because the new code may have broken some existing code.
7. Refactor the code in order to make it cleaner.

8. Compile all the tests again and see that they all pass.

Regarding refactoring, it is the process of changing the code without changing its behavior. It can be risky, as it can introduce new bugs (called regression bugs), so it should not be done until all the tests pass. Regression tests are tests that check that the behavior of the code has not changed after refactoring.

3.3. Testing

As explained before, testing is a crucial part of the development process, as they help us feel confident that our code works and it reduces the developers' stress. However, they are not easy at all, as they require a lot of practice and experience to be good at writing tests. In addition, it should be noted that tests *are also code*. They must be designed, written, refactored and maintained just like production code.

3.3.1. Red Bar Patterns

A red bar represents a failing test; so a red bar pattern is a pattern that leads to a failing test; which is what should be done according to the microcycle of TDD. There are some common red bar patterns that should be followed:

- List of Future Tests: a list of tests that should be written in the future should be written. It lets us focus on one test at a time, and it helps us keep track of the tests that we want to write. It should not be a fixed list, as it may change as we write the code and discover new things.

- Picking the Next Test: One that can easily be made to pass should be picked. This way, we can make progress quickly and feel good about it, which motivates us to keep going. If none applies, an existing bigger test can be split into smaller tests (which should also be added to the list of future tests).

  It is also important to note that a passing test provides an assertion. Therefore, tests that allow to assume things that will be useful for future tests should be picked, as they will make it easier to write the future tests.

- Adding Tests: Test should be added to the list of future tests for several reasons: when a potencial problem is introduced, when a bug is reported, when a big test is split into smaller tests... However, it should not be forgotten that the list may eventually be finished.

- Removing Tests: The list of future tests should be reviewed regularly, and tests that are no longer relevant should be removed, as more tests does not necessarily mean better testing.

  If a test can be made fail individually, it should be kept, as it tests something that is not tested by other tests. However, if a test cannot be made fail individually, it may be testing something that is already tested by other tests, and its deletion may be considered.

There are two more aspects that should be taken into account regarding red bar patterns: what to do when stuck, and how to end the programming session.

- When Stuck: if we are stuck, our first attempt should be to take a short break, as it may help us clear our mind and see the problem from a different perspective. If we are still lost after the break, throwing away the code and starting over may be a good idea, as it allows us to start fresh and avoid getting stuck in a dead end.

  Another good idea is to switch partners in pair programming, as it forces us to explain our code to the new partner, which may help us see the problem from a different perspective. In addition, the new partner is not emotionally invested in prior decisions, which may make it easier to throw away the code and start over if necessary.

- Ending the Programming Session: when it comes to ending the programming session:

  - If we program alone, we should end the session with the last test failing, so that we know where to start the next session.
  - If we program in teams, we should end the session with all the tests passing, so that we can leave the code in a good state for the next person.

3.3.2. Green Bar Patterns

A green bar represents a passing test; so a green bar pattern is a pattern that leads to a passing test. There are some common green bar patterns that should be followed:

- Obvious Implementation: the simplest implementation that makes the test pass should be written.

- Fake it ('til you make it): if the test is hard to make pass, a fake implementation (maybe returning a hardcoded value) can be written to make the test pass, and then it can be refactored to a real implementation.

  However, it is important to make sure that you have a test that will force you to look at the fake implementation again, if not, you should add it to your list of future tests.

- Triangulation: if you don't know how to code the implementation, wait until you have two or more tests. In that case, you should use the *Rule of Thumb*: fake it until it is more trouble to fake it than to make it. Therefore, when you can no longer fake it, you should code the implementation, as you will have enough information to do it.

3.3.3. Testing Patterns

Some good testing principles that should be followed are:

- Small Tests: tests should be small. A test should one fail for one reason, and it should be easy to understand why it fails.
- Always rerun all the tests: all the tests should be rerun after every change, as we may have broken something without realizing it.
- Tests should fail: if a bug is intentionally introduced, a test should fail.
- Tests should fail independently: ideally, each test should be able to fail independently of the others. If a lot of tests fail at the same time, it may be hard to understand why they fail and to fix the problem. It could also mean that the tests are not well designed, as they may be testing too much at the same time, or they may be redundant.
- Do not debug: in case of a failing test, if we can fix it without debugging, we should do it, as it is faster and it helps us keep the flow. However, if we cannot fix it directly, it is faster to go back (using VCS).
- Tests for external software: if we are using external software (for example, a database), we should write tests for it, as it may have bugs that affect our code.
- Test Private Methods through Public Methods: private methods should not be tested directly, as they are implementation details that may change. Instead, they should be tested through public methods that use them.
- Test Error Cases: tests should not only cover the correct cases, but also the errors. It should be checked that the errors are reported and handled correctly. With this aim, crash test dummies (explained in the following subsection) can be used.
- Test Should be Fast: tests should be fast, as they will be run frequently. If tests are slow, we may be tempted to run them less frequently (which is not desired). Posible ways to make tests faster are mocking external software (explained in the following subsection) or removing unnecessary tests.
- Test Unit Behavior: tests should not necessarily test the whole class, but they should test the behaviors of the unit. The tests' name should reflect the behavior that they are testing.
- Legacy Code: if we have legacy code (code without tests), there are two possible approaches to add tests to it:
  - If you can add unit tests directly, do it but just testing its current behavior (not caring about if it's correct or not).
  - If you cannot add unit tests directly, start from the outside to the inside: start from the outside (whole system tests), use them to write integration tests, and then use them to write unit tests.

  This way, you will have a safety net to refactor the code and add more tests later.

There is one more aspect that should be taken into account, mocking.

Mocking

Mocking is the process of replacing a real object with a fake one that simulates its behavior. It is useful when we want to test a unit that depends on an external software (for example, a database), or when we want to test a unit that is not yet implemented. Mocking is in fact a *slang* term for *test double*, which is a more general term that includes different types of fake objects:

- Test Dummy: an object that is passed around but never actually used (only used to satisfy the compiler).
- Test Stub: an object that provides predefined answers to method calls.
- Crash Test Dummy: special stub that return errors (throws exceptions) when called. It is mainly used to test error handling.
- Exploding Fake: an object that causes test failure if it is called. It is mainly used to test that a method is not called when it should not be.
- Test Spy: special stub that records information about how it was called, which functions were called, with what parameters, how many times...
- Mock Object: more complex spy that also includes assertions about how it should be called.
- Fake Object: an object that has a working implementation, but it is not suitable for production (for example, an in-memory database).

When injecting a test double, they should not be injected directly, as it may make the code more complex and less maintainable. Instead, they should be injected as a reference to the class. Therefore, the test double can be passed when testing, and the real object can be passed when running the code in production.

Mocking has several advantages, as it allows us to test units that depend on external software or that are not yet implemented. It also allows for faster tests. Therefore, a common mistake is thinking that every dependency should be mocked, but that can lead to slower tests and explosion of polymorphism (for dependency injection). Therefore, the "Uncle Bob's[1] Mocking Heuristic" should be followed: mock across architecturally significant boundaries, but not within those boundaries.

[1]Robert C. Martin, author of Clean Code, is known as Uncle Bob in the software development community.

4. Agile Development

Software Development has always been a field in constant evolution, with new methodologies and practices emerging to address the challenges of creating high-quality software efficiently. One of the most influential approaches in recent years has been Agile Development, which emphasizes flexibility, collaboration, and customer satisfaction. There are three main phases in the history of Software Development:

1. The Time of Legends: Pre-Agile Era:

   During the early days of software development, it was done in small controllable steps, with a focus on craftsmanship and quality. Projects were smaller, and developpers had more experience.

2. Industrialization: Scientific Management:

   From the 1970s, the most common approach was the Waterfall model, which is a linear and sequential process. Coding should be postponed until the design phase is complete, and everyone should follow the plan to the letter. This is ideal for projects with high cost of change and well defined problems, which is actually not really common.

3. Reformation: the comeback of Agile:

   It started in the 1990s with the emergence of new methodologies like Scrum and Extreme Programming (XP). In 2001, a group of software developers created the Agile Manifesto, which outlined the core values and principles of Agile Development.

In typical development process, after the common phases (requirements gathering, design, implementation...) usually a well-known phase arises: the *Death march phase*, where the project is in a critical state, with a lot of pressure to deliver, and the team is working overtime to meet the deadlines. This happens often because they have no real data that informs about the actual progress of the project, but just the *hope* to meet the deadlines. Agile provides data before the hope can kill the project.

In Figure 4.1 we can see the Iron Cross, which represents four qualities that software projects should have: *fast, cheap, good, done*. However, it is said that you can only have three of these qualities at the same time, and you have to sacrifice one of them. In order to correctly manage the iron cross, data is needed. The main core of Agile is to provide these data.

There are two main questions whose answer (based on data, not on hope) is the main goal for Agile:

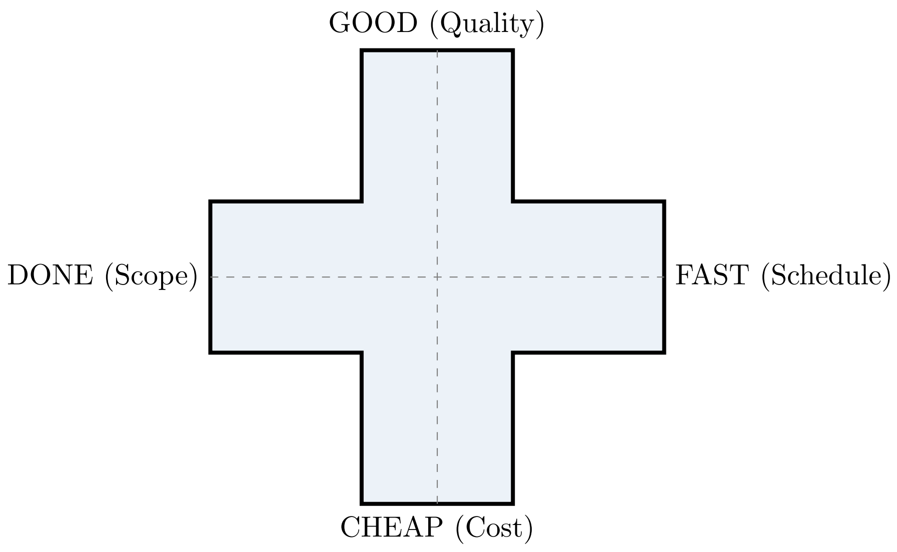

Figura 4.1: The Iron Cross of Software Development

- *Do we do the right thing?*: Fast iterations, early feedback and planning are used.
- *Can we still do it in time?*

Apart from that, some core Agile values are:

- Personal courage and risk taking.
- Intense communication and collaboration, ignoring barriers of hierarchy, location and time.
- Rapid feedback and learning.
- Simplicity of code, design and teams.
- Respect.

There are different Agile methodologies, such as Scrum, Extreme Programming (XP), Crystal... However, they all share the same core values and principles, and they all aim to provide data to manage the iron cross of software development.

4.1. Agile Methodologies

4.1.1. Scrum

Scrum is one of the most popular Agile methodologies, and it is based on the idea of iterative and incremental development. A depper understanding of the concepts behind Scrum can be found in the Chapter 1 of the contents of the Application Management course.

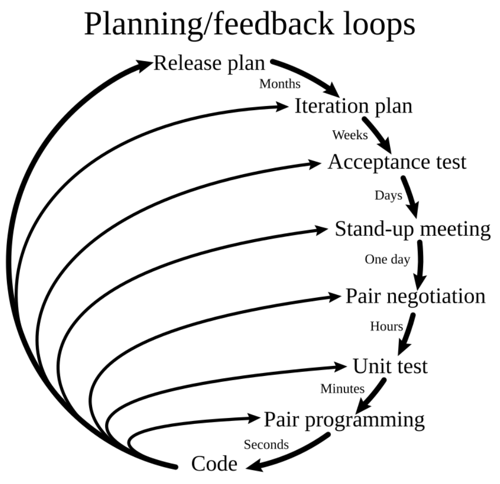

Figura 4.2: Planning and feedback loops in XP

4.1.2. XP (Extreme Programming)

XP is another popular Agile methodology, and it is based on the idea of continuous feedback and improvement. In the Figure 4.2 we can see that it uses a planning and feedback loop that represents how frequent are the feedbacks and iterations in XP.

This methodology is based on a concept named "Circle of Life", which represents the practices that should be taken, from a bigger aspect (the whole project) to a smaller aspect (the code itself). They will be convered in the following sections.

Business-facing practices

They are practices whose aim is to reduce the common gap between the business and the development team. The main goal here is planning correctly, and therefore a project should be divided into small pieces, where each piece should be estimated indivially. This estimation should be as accurate as possible but only as precise as needed. Some techniques for estimation are:

- Trivariate Analysis: Long-term estimation technique. The time to do a task is estimated based on three different scenarios:
  - *Worst case*: Time within which the task will be completed with a 95 % confidence.
  - *Nominal case*: Time within which the task will be completed with a 50 % confidence.
  - *Best case*: Time within which the task will be completed with a 5 % confidence.
- Story Points: Short-term estimation technique. The project is divided in stories.

- An user story is an abbreviated description of a feature from the perspective of the user. It typically follows the format: “As a [type of user], in order to [achieve a goal], I will [perform an action]”.

The stories should be made between the developpers and the stakeholders, and the details should be included when started working on the story, not before.

Each story should be estimated in story points.

- A story point is a unit of measure for expressing the overall effort (not time) required to implement a user story.

An initial story (called the “golden story”) of average complexity is chosen as a reference, and the other stories are estimated relative to it. For example, if the golden story is estimated to be 5 story points, and another story is estimated to be twice as complex, it would be assigned 10 story points. This technique allows for more accurate estimation and better planning of the project.

Then, the project evolves in iterations (each one finishing with a demo), which are organized in interation planning meetings.

- In iteration planning meetings, the stakeholders select a set of user stories to be implemented in the next iteration, based on their priority and the team’s *velocity* (the number of story points the team can complete in an iteration).

The initial velocity is an initial guess, and the velocity of next iterations is calculated based on the velocity of the previous iteration. Managers may have a velocity chart, which is a visual representation of the team’s velocity over time, to help track progress and make informed decisions about future iterations.

In addition, a *midpoint check* is usually done at the middle of the project, where the team reviews the story points completed so far, and compares it with the initial estimation. This allows to adjust the remaining story points asking the stakeholders to add/remove stories.

Team-facing practices

They are practices whose aim is to improve the communication and collaboration within the team. Some of these practices are:

- Metaphore / Ubiquitous language: A common language that is used by all members of the team, including developers, testers, and stakeholders.
- Sustainable pace
- Collective ownership
- CI
- Standup meetings

Technical practices

They are practices whose aim is to improve the quality of the code and the design of the software. Some of these practices are:

- TDD
- Pair programming: Two developers work together on the same code, sharing the workstation (it can also be done virtually). One developer writes the code, while the other reviews it in real-time. This practice promotes collaboration, knowledge sharing, helps to catch errors early in the development process, and also helps to train new team members. Pairing of larger teams can also be done, which is called *mob programming.*

  Some characteristics that pair programming should have are:

  - Optional
  - Intermittent
  - Unscheduled
  - Short-lived
- Refactoring
- Simple design

  Only the simplest design that works should be implemented. Some rules for simple design, in this priority order, are:

  1. Pass all tests.
  2. Reveal the intention of the code.
  3. Remove duplication.
  4. Decrease the number of elements (classes, methods, variables...).

5. Mid-level Software Design

In this course, our aim is to design good quality software, and to do that we need to know how to design it. The following are smells of bad design:

- Rigidity: the software is hard to change because every change affects too many other. Everything that *depends* on the changed thing needs to be changed too.
- Fragility: when you change something, it breaks something else that you didn't expect to break. There are hidden *dependencies*.
- Immobility: the software is hard to reuse because it's too entangled with the current application. The desired design is highly *dependent* on undesired parts of the software.

As the reader may have noticed, the common main root cause of all these smells is *dependencies*. Therefore, the solution to these problems is to design software with low dependencies. In order to do so, keeping a *simple* design is key. Only the code that is required should be written, with a structure that keeps it simplest, smallest and most expressive. In order to do so, the following rules should be followed (in this order):

1. Pass all the tests
2. Reveal the intent
3. Remove duplication
4. Decrease elements of the software

In this chapter, we will focus on the mid-level design of software. There are 4 levels:

1. Functions: already covered in Section 1.2.
2. Classes
3. Components
4. System or Architecture

Therefore, in this chapter we will cover the design of classes.

Definición 5.1 (Class)**.** Coupled grouping of functions and data.

Observación. Even though that is the definition, there are a lot of programming languages that call *classes* to things that are not classes; and also there are a lot of programming languages that give other names to things that are classes (for instance, *structs* in `C++`).

Regarding the dependencies mentioned before, and given that classes are connected to others, there are two types of dependencies:

- Compile-time dependencies: they are the ones that are created when the software is being compiled. They are produced when one class mentions another class (or its functions or data) in its code.
- Runtime dependencies: they are the ones that are created when the software is running. They are produced when one class transfers control to another's functions or reads/writes data from another class.

  One example of runtime dependency that is not a compile-time dependency is the case of *interfaces* (or *abstract classes*). A class A may have a compile-time dependency on an interface I, but then at runtime it may have a runtime dependency on a class B that implements I, without having a compile-time dependency on B.

Duting the following section, we will cover some important principles for designing good classes.

5.1. The SOLID Principles

The SOLID principles are a set of five design principles that are intended to make software more maintainable and flexible. Applying them too often/early can lead to over-engineering and unnecessary complexity, but they are very useful when applied in the right way.

5.1.1. Single-Responsibility Principle (SRP)

A class should have only one reason to change.

If a class has more than one responsibility (reason to change), they become coupled, so changes in one responsibility may affect the other. Ideally, each responsibility should be encapsulated in a separate class, but if not, at least they should be separated in different interfaces.

The SRP may lead to *temporary duplication*, as the same code may be needed in two different classes, but if it is due to different responsibilities (reasons to change), then this duplication may eventually disappear as the two classes evolve in different ways.

Another common symptom of a violation of the SRP is the presence of *merge conflicts*, as two different developers may be working on the same class for different reasons, and therefore merge conflicts may appear.

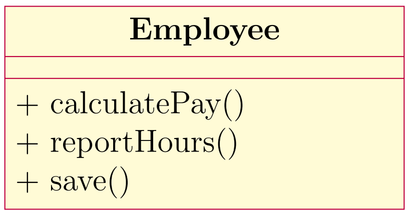

Figura 5.1: Example of a class that violates the Single-Responsibility Principle.

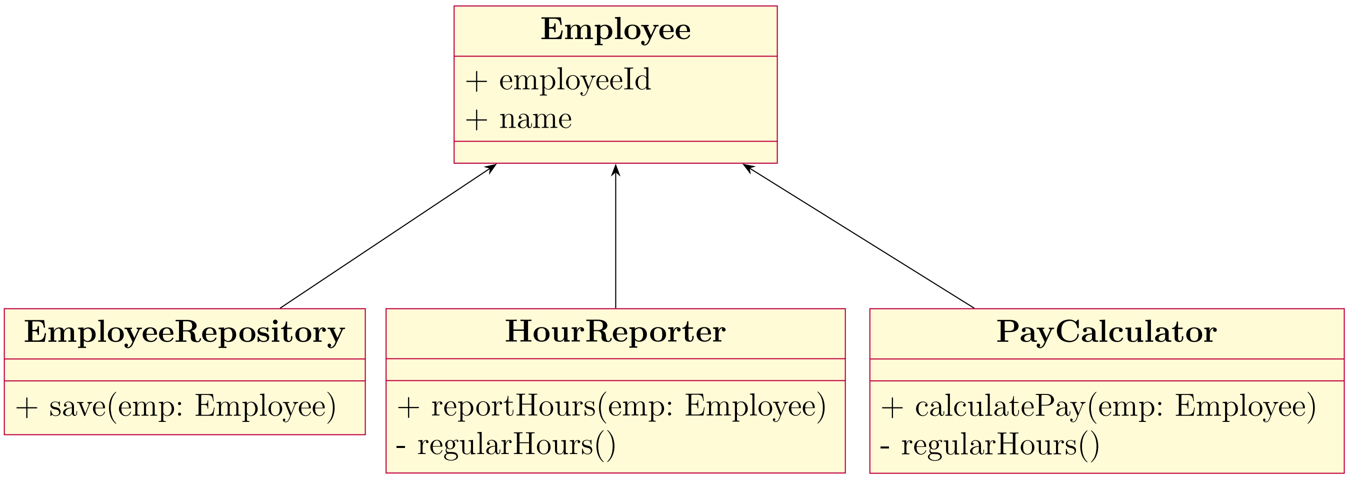

Figura 5.2: Example of a design that follows the Single-Responsibility Principle.

One example of a violation of the SRP is the class `Employee` described in Figure 5.1. This class has three responsibilities, which may lead to three different reasons to change. A possible solution to this problem is to split the class into three different classes, each one with a single responsibility, as shown in Figure 5.2. The three responsibilities are:

- Calculating the pay of the employee (possible used by the finance department).
- Reporting the hours worked by the employee (possible used by the HR department).
- Saving the employee data (possible used by the IT department).

5.1.2. Open-Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

They should open for extension, as new requirements may mean new behavior that needs to be added to the software. However, they should be closed for modification, as changing existing code may introduce bugs and break existing functionality.

An example of a violation of the OCP is the design shown in Figure 5.3, as trying to connect the `Client` to a new `NewServer` would require modifying the `Client` class. A possible solution to this problem is to introduce an interface `Connection`

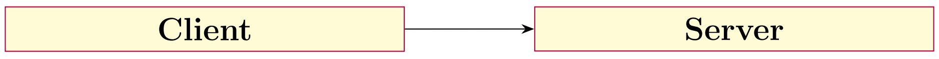

Figura 5.3: Example of a design that violates the Open-Closed Principle.

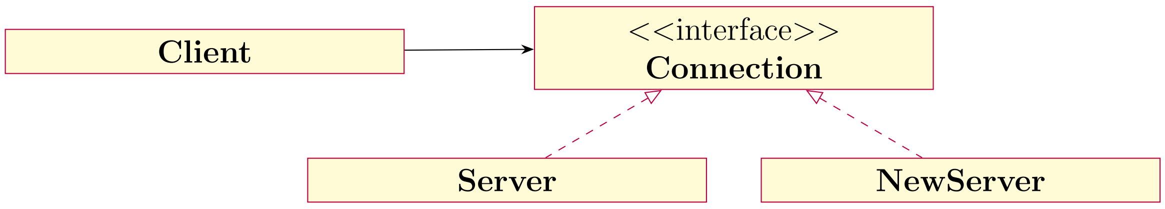

Figura 5.4: Example of a design that follows the Open-Closed Principle.

that both `Server` and `NewServer` implement, as shown in Figure 5.4. This way, the `Client` can connect to any class that implements the `Connection` interface without needing to be modified.

There will always some kind of change against which a module is not closed, so deciding which changes to be closed against is a matter of judgment. As many data as available should be collected (asking customers or domain experts, using personal experience, etc.) to make an educated guess. A common practice is considering that no changes will happen and, when a change happens:

1. We first implement the abstractions needed to make the change possible.
2. Then we implement the change itself.

5.1.3. Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types.

This principle is really important for the polymorphism. If a subtype violates this principle, then the code using the base type must be aware of the specific subtype it is using, which leads to code that is more difficult to maintain and extend. This principle enforces:

- Preconditions cannot be strengthened in a subtype.
- Postconditions cannot be weakened in a subtype.
- Invariants of the base type must be preserved in a subtype.
- A latter history contraint is also considered.

  Given that a subtype may have more methods than the base type, new changes of states may appear in the subtype that are not present in the base type. This should be avoided: only the changes of state allowed by the base type should be allowed in the subtype.

Examples of violations of the LSP also include throwing exceptions that the base type does not throw or degenerate functions (functions without any code).

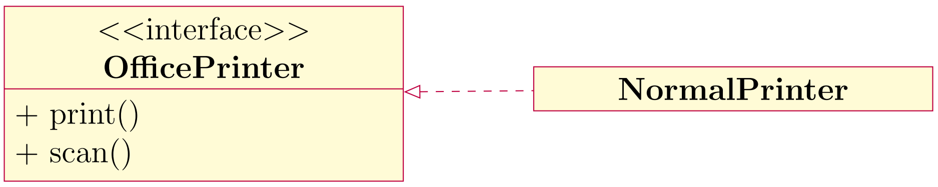

Figura 5.5: Example of a design that violates the Interface Segregation Principle.

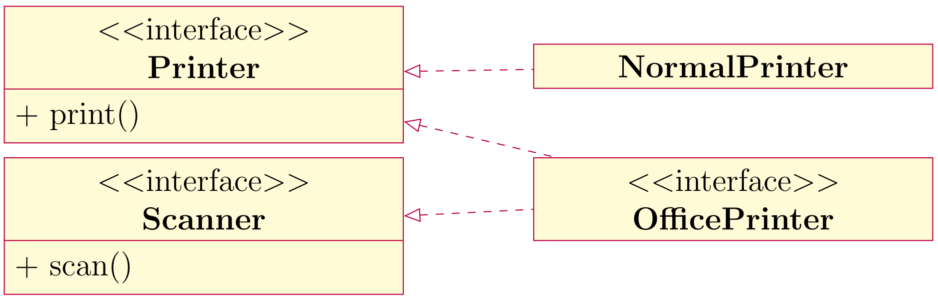

Figura 5.6: Example of a design that follows the Interface Segregation Principle.

LSP should not be confused with Strongly-typed OO. The latter force the *structure* to be consistent (resulting in a compile-time error if it is not), while LSP force the *behavior* to be consistent (resulting in a runtime error if it is not). Therefore, violating LSP may not be detected until runtime, making it much more dangerous.

5.1.4. Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use.

This principle is intended to prevent the creation of *fat interfaces*, which are interfaces that have more methods than the clients need. This leads to clients depending on methods that they do not use, which can lead to code that is more difficult to maintain and extend.

As always, over-engineering should be avoided in order to not have a class implementing dozens of interfaces. Interfaces grouping different interfaces together may help to avoid this problem. This principle should not be confused with the SRP, as the SRP is about the responsibilities of a class, while the ISP is about the responsibilities of an interface.

An example of a violation of the ISP is the design shown in Figure 5.5, where a normal printer would have to implement the `Office Printer` interface, which has methods that it does not need. A possible solution to this problem is shown in Figure 5.6, where two different interfaces are created, one for printing and one for scanning, and the `Office Printer` implements both interfaces, while the normal printer only implements the printing interface.

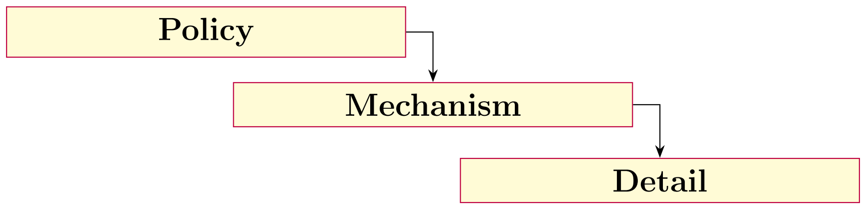

Figura 5.7: Example of a design that violates the Dependency Inversion Principle.

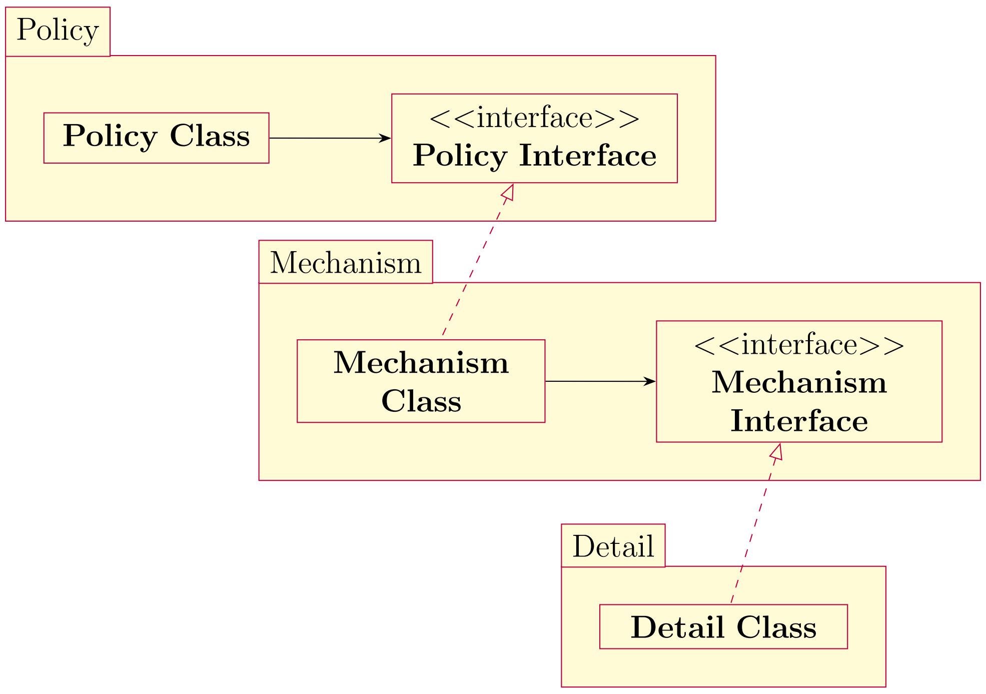

Figura 5.8: Example of a design that follows the Dependency Inversion Principle.

5.1.5. Dependency Inversion Principle (DIP)

The code that implements high-level policy should not depend on the code that implements low-level details. Rather, details should depend on policies.

In a software programm, usually there are high-level modules (policies), middle-level modules (mechanisms) and low-level modules (details). Usually, each level depends on the level below it, as we can see in Figure 5.7. The solution proposed by the DIP is to invert these dependencies, so that the high-level modules do not depend on the low-level modules, but on abstractions (interfaces) that the low-level modules implement. This way, the high-level modules are not affected by changes in the low-level modules. This approach is displayed in Figure 5.8.

A specific example of a violation of the DIP is the one shown in Figure 5.9, where the `UserService` (policy) depends on the `SQLDatabase` (detail), which is a low-level module. In case that another database is needed, the `UserService` would need to be modified, making it more difficult to maintain and extend. A possible solution to this problem is to introduce an interface `Database` that the `SQLDatabase` implements, and then the `UserService` depends on the `Database` interface instead of the `SQLDatabase` class. This way, the `UserService` is not affected by changes in

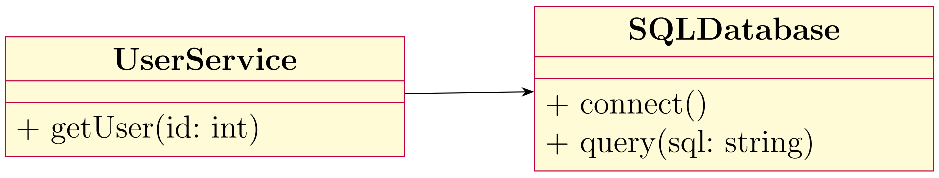

Figura 5.9: Specific example of a design that violates the DIP.

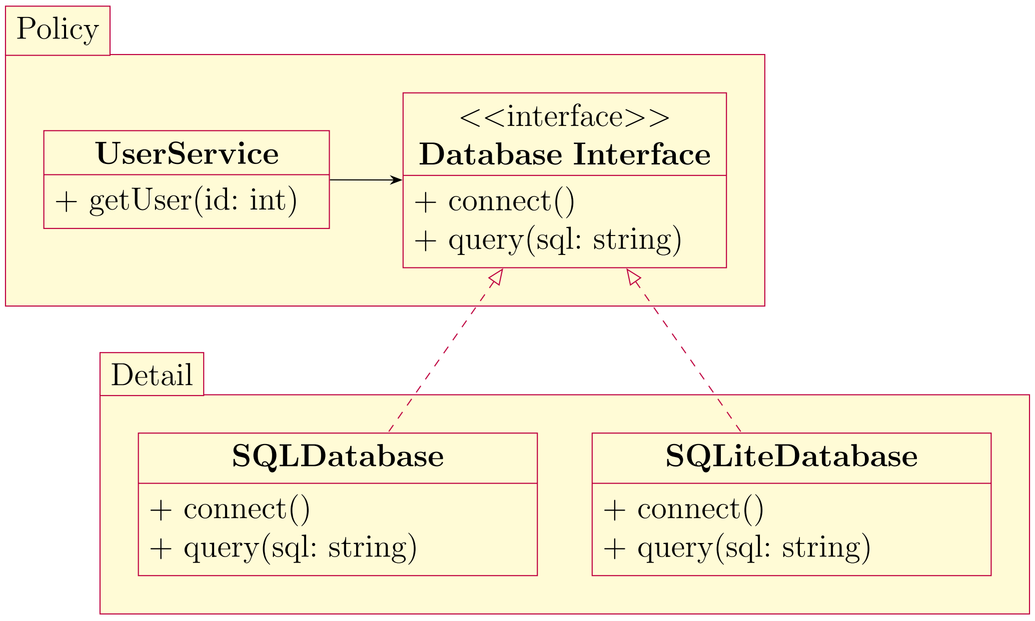

Figura 5.10: Specific example of a design that follows the DIP.

the database implementation, and it can work with any database that implements the `Database` interface. This approach is shown in Figure 5.10.

5.2. Further Principles

There are much more principles that can be applied to the design of software, but they are not as widely known as the SOLID principles. Some of them are:

- KISS (Keep It Simple, Stupid): the design should be as simple as possible. Complexity should be avoided, as it makes the software more difficult to understand, maintain and extend.

- YAGNI (You Ain't Gonna Need It): only the code that is required should be written. Adding code that is not currently needed may lead to over-engineering and unnecessary complexity.

- DRY (Don't Repeat Yourself): duplication should be avoided, as it makes the software more difficult to maintain and extend.

- SoC (Separation of Concerns): different concerns should be separated, as it makes the software more modular and easier to maintain and extend.

- APO (Avoid Premature Optimization): optimization should be avoided until it is necessary, as it may lead to over-engineering and unnecessary complexity.
- LoD (Law of Demeter): a class should only depend on its direct collaborators, and not on the collaborators of its collaborators. This principle is intended to reduce the coupling between classes and to make the software more maintainable and extendable.

6. Design Patterns

A design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into code, but rather a template for how to solve a problem that can be used in many different situations.

The learning cycle of design patterns usually has two stages:

1. The Phase of Enthusiasm: In this phase, developers are excited about the design patterns learnt and try to squeeze as many patterns as possible into their code, even when they are not necessary.

2. The Phase of Disillusionment: In this phase, developers realize that not all design patterns are suitable for every situation and that overusing them can lead to unnecessarily complex code. They learn to apply design patterns judiciously, using them only when they provide a clear benefit.

Therefore, it is important to understand the context in which a design pattern is applicable and to use it appropriately to solve specific problems in software design. Their bad consequences should also be taken into account.

The most famous patterns are the ones described in the book Design Patterns: Elements of Reusable Object-Oriented Software, which are known as the *Gang of Four* (GoF) design patterns (the authors of the book). However, there are many other design patterns that have been developed since then, and new ones are still being created as software design evolves. In the following sections, we will cover some of the most commonly used design patterns (esentially the GoF patterns, but also some others). They are also explained in detail in this website.

6.1. Creational Patterns

Creational patterns are concerned with the process of object creation.

6.1.1. Dependency Injection

Dependency Injection is sometimes considered a design pattern. It allows a class to receive its dependencies from an external source rather than creating them itself. This promotes loose coupling and makes the code more modular and testable. For instance, instead of a class creating an instance of a dependency, it can receive it through its constructor or through a setter method. This approach is shown in Listing 1.

```java
public class Client {
    private Connection connection;
    public Client(Connection connection) {
        this.connection = connection;
    }
}
```

Código fuente 1: Dependency Injection example.

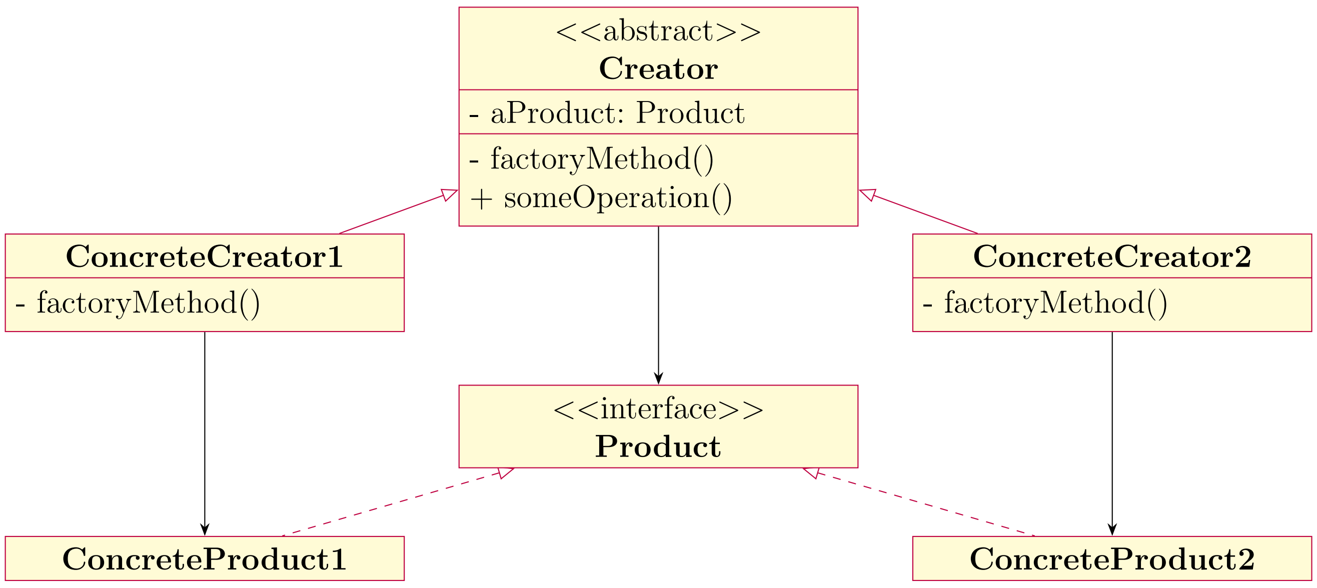

Figura 6.1: Factory Method pattern.

However, if multiple new objects were needed (maybe iteraing over a `for` loop), then the dependency injection would not be suitable. We would need to actually create the objects, which would lead to a tight coupling between the classes. This is where the following creational patterns come into play.

6.1.2. Factory Method

This pattern is used when a class needs to create objects, but does not know the exact class of the object that will be created. It uses inheritance to allow subclasses to decide which class to instantiate.

The main application is usually called the `Creator` class, which has to use a product, which is any class that implements a common interface `Product`. The `Creator` is a abstract class that defines the factory method, which is responsible for creating the product. The concrete subclasses of `Creator` implement the factory method to create specific products. This pattern is shown in Figure 6.1.

Observación. It should be noted that this can also be used to better name the constructors of the products, since they can be named according to the product they create. The name `factoryMethod` can be replaced by a more descriptive name, such as `createProduct1` or `createProduct2`, which can make the code more readable and easier to understand.

A possible implementation for this is when an application needs to notify users about certain events, but the way to notify them can vary (e.g., email, SMS, push notifications). The `Creator` class would be the application, the `Product` interface would define the method for sending notifications, the concrete products would be the different notifiers (e.g., `EmailNotifier`, `SMSNotifier`, `PushNotifier`), and the concrete creators would be the classes responsible for creating the specific notifiers based on the user's preferences or the context of the notification. However, almost the whole implementation would be the in the `Creator` class.

6.1.3. Abstract Factory

This pattern is used when a class needs to create families of related or dependent objects without specifying their concrete classes. Two ideas should appear:

- There are different products, and each product has different subtypes. In the Figure 6.2, the products and their subtypes are represented using letters.
- The subtypes are organized into families. In the Figure 6.2, the families are represented using numbers.

The main application has to use all the products, but each one of them is an abstract class from which all subtypes inherit. In order to use all the subtypes from the same family, the main application also has to use an abstract factory (an abstract class) for creating the products. The concrete factories (concrete classes) implement the abstract factory to create specific products from the same family. This pattern is shown in Figure 6.2.

The idea may be better understood with an example. Consider the client to be a software application that needs to create user interfaces for different platforms (e.g., Windows, macOS, Linux). The products would be the different UI components (e.g., buttons, text fields, checkboxes), with its subtypes being the specific implementations for each platform (e.g., `WindowsButton`, `MacOSButton`, `LinuxButton` for the button product). The families would be the different platforms, with the concrete factories being the classes responsible for creating the specific UI components for each platform (e.g., `WindowsFactory`, `MacOSFactory`, `LinuxFactory`). The client application would use the abstract factory to create the UI components without needing to know the specific classes of the products it is using.

In this pattern, concrete classes names only appear in the concrete factories, which are the only ones that know about the specific products they create. This leads to the following advantages:

- Easy to add new varieties of products without changing the client code (just adding a new family).
- Easy to assure consistent usage of products from the same family (the client only uses one factory).
- Easy to exchange whole families of products (the client can use a different factory).

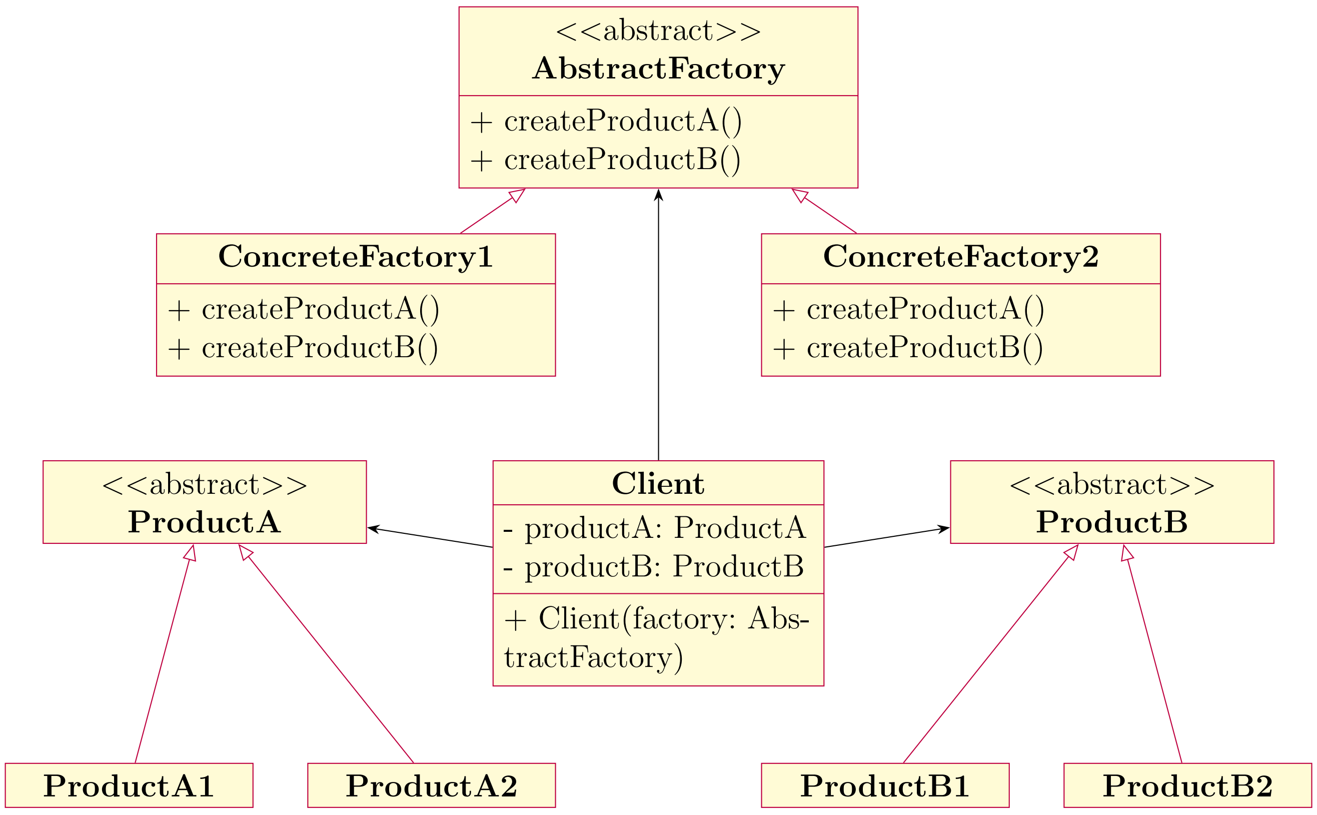

Figura 6.2: Abstract Factory pattern.

However, the complexity of the code increases and adding just one new product requires creating one version of the product for each family, which can lead to a large number of classes.

6.1.4. Prototype

This pattern is used when a class needs to create new objects by copying existing ones, rather than creating new instances from scratch. It allows for the creation of new objects based on a prototype instance, which can be cloned to create new objects with the same properties and behavior. There is a `Prototype` interface that defines the method for cloning itself, and concrete classes that implement this interface to create specific prototypes. It it really important that the cloning method creates a *deep copy* of the object, meaning that all the properties and references of the object are also copied, rather than just copying the reference to the original object. This ensures that the cloned object is independent of the original object and can be modified without affecting it. This pattern is shown in Figure 6.3.

This approach has different advantages, because it is simpler than factories and allows getting rid of repeated initialization code. It is also useful when the creation of an object is expensive (e.g., it requires a lot of resources or time) and we want to avoid creating new instances from scratch. However, it can be difficult to implement correctly, especially when the objects being cloned have complex structures or contain references to other objects.

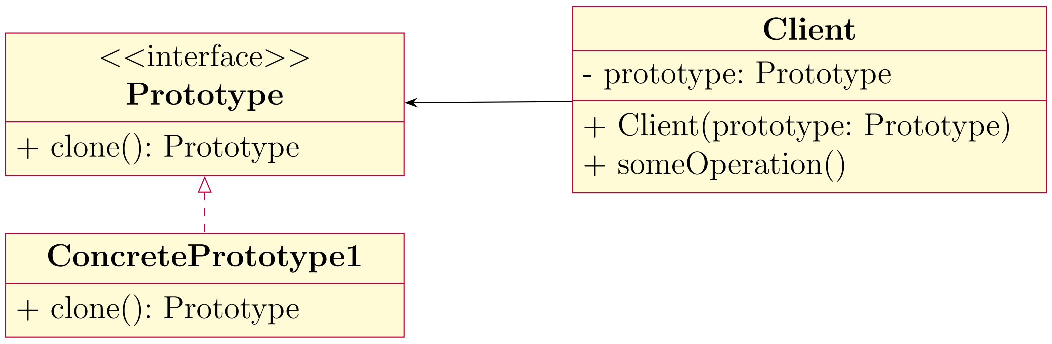

Figura 6.3: Prototype pattern.

```java
public class Computer {
    private String CPU;
    private String GPU;
    private int RAM;
    private int storage;
    private String operatingSystem;
    public Computer(String CPU, String GPU, int RAM, int storage, String
    operatingSystem);
}
```

Código fuente 2: Example of a class with a constructor that has too many parameters.

6.1.5. Builder

This pattern is used when a class needs to create complex objects step by step, rather than creating them in a single step. It allows separating the construction process from the representation of the object. The code shown in Listing 2 is an example of the necessity of this pattern, because the constructor of the `Computer` class has too many parameters, which can lead to confusion and errors when creating instances of the class.

As we saw, injecting so many dependencies through the constructor leads to a confusing and error-prone code. In order to avoid it, an additional class called `Builder` can be created, which is responsible for constructing the complex object step by step. The `Builder` class has methods for setting each property of the complex object, and a method for building the final object. This pattern would be used as shown in Listing 3. It is really important that each setter method in the `Builder` class returns the builder itself, which allows for method chaining and makes the code more readable and easier to understand. This approach also allows for more flexibility in the construction process, as it allows for optional parameters and different configurations of the complex object to be created without needing to create multiple constructors with different parameter combinations.

Using a Dictator

The general builder pattern includes also the idea of a *dictator*, which is a class that controls the construction process and ensures that the complex object is built

```java
public class Builder {
    private Computer computer;
    public Builder() {
        computer = new Computer();
    }
    public Builder setCPU(String CPU) {
        computer.setCPU(CPU);
        return this;
    }
    // Other setter methods for RAM, storage, and operatingSystem
    public Computer build() {
        return computer;
    }
}

public static void main(String[] args) {
    Computer computer = new Builder()
        .setCPU("Intel Core i7")
        .setGPU("NVIDIA GeForce RTX 3080")
        .setRAM(16)
        .setStorage(512)
        .setOperatingSystem("Windows 10")
        .build();
}
```

Código fuente 3: Example of using the Builder pattern to create a complex object.

correctly. This class contains the extact knowledge about the steps (the order of the setter methods) that need to be followed to create a valid complex object.

The first apprach is the shown in Listing 4, where the client code just uses the director to create the complex object. The director knows the exact steps that need to be followed to create a valid complex object (whether because they are hard coded, or just retrieved from a configuration file or database). This approach has the advantage of keeping the client code simple and easy to understand, as it only needs to interact with the director to create the complex object.

However, in this approach, the director class has too much control over the construction process, which can lead to a rigid and inflexible design, and may be violating the OCP (in case we want to modify the specifications of the complex objects being created). An alternative approach is shown in Figure 6.4, where all the exact information is stored in concrete builder classes (all of them implementing a common builder interface), and the director just uses the builder interface to create the complex objects. This allows for a more flexible and extensible design, as new complex objects can be created by simply adding new concrete builder classes that implement the builder interface, without needing to modify the director class. In addition, and given that the product obtained is only managed by the concrete builders, the director class does not need to know about the specific products being created and those products do not need to implement any common interface (as opposed to the abstract factory approach), which promotes loose coupling and makes the code more modular and easier to maintain. For instance, one concrete builder could return a `Computer` object, while another concrete builder could return

```java
public class Director {
    private Builder builder;
    public Director() {
        builder = new Builder();
    }

    public Computer constructGamerComputer() {
        return builder
            .setCPU("Intel Core i7")
            .setGPU("NVIDIA GeForce RTX 3080")
            .setRAM(16)
            .setStorage(512)
            .setOperatingSystem("Windows 10")
            .build();
    }

    public Computer constructOfficeComputer() {
        return builder
            .setCPU("Intel Core i5")
            .setRAM(8)
            .setStorage(256)
            .setOperatingSystem("Windows 10")
            .build();
            // No GPU for office computer
    }
}
```

Código fuente 4: Example of using a Director class to control the construction process of complex objects.

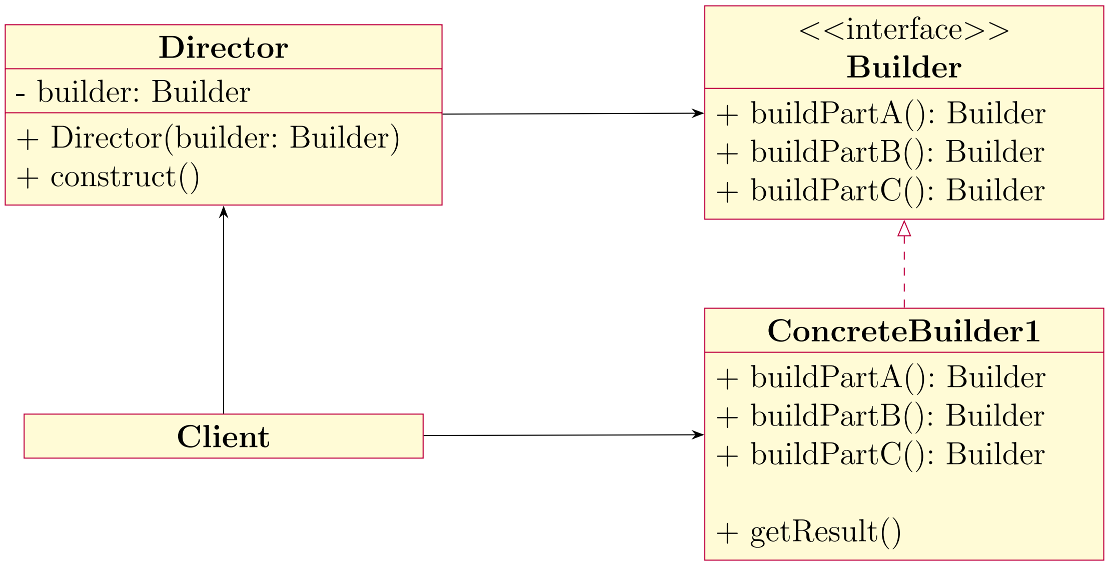

Figura 6.4: Example of using a Director class with a Builder interface to control the construction process of complex objects.

a `String` object representing the specifications of a computer.

Some of the advantages of this approach are the construction of different complex objects using the same construction process (the director class), and the ability to create complex objects with different representations (the concrete builder classes can create different products). However, it can lead to a large number of classes and must handle cases where the construction process is not finished (calling the `getResult` method too soon).

6.1.6. Singleton

This pattern is used when a class needs to ensure that only one instance of itself is created and provides a global point of access to that instance. It is often used for managing shared resources, such as database connections or configuration settings. The singleton class has a private constructor to prevent other classes from creating instances of it, and a static method that returns the single instance of the class. This pattern is shown in Listing 5.

6.1.7. Monostate object

This pattern is used when a class needs to have only one possible state, but multiple instances of the class can exist. It is similar to the singleton pattern, but instead of ensuring that only one instance of the class is created, it allows for multiple instances while ensuring that they all share the same state. The monostate class uses static variables to store the shared state, and all instances of the class access and modify this shared state.

```java
public class Singleton {
    private static Singleton instance;
    private Singleton() {
        // Private constructor to prevent instantiation
    }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Código fuente 5: Example of a Singleton class.

6.2. Structural Patterns

Structural patterns are concerned with the composition of classes and objects, and how to combine objects into bigger parts. They provide a way to create relationships between classes and objects to form larger structures while keeping the code flexible and maintainable.

6.2.1. Adapter

Also known as Wrapper, this pattern is used when a class needs to work with another class that has an incompatible interface. It allows for the conversion of one interface into another, so that the two classes can work together. A main class (the client) uses a target interface, but the adaptee class (the one that should be used) cannot implement the target interface directly. Therefore, an adapter class is created, which implements the target interface and contains an instance of the adaptee class. The adapter class translates the requests (both the data and the methods) from the client into calls to the adaptee class, allowing them to work together. This pattern is shown in Listing 6.

This should be avoided when possible, as making the interfaces compatible can lead to a better design and a more maintainable code. However, there are cases where the adapter pattern can be useful, such as when working with third-party libraries that cannot be modified to fit the desired interface. For instance, the client could be a shopping cart application, and the interface could define a method for making a payment. Different payment gateways (e.g., PayPal, Stripe, Square) may have different APIs for processing payments, and each one may need an adapter class to translate the client's payment requests into the specific API calls required by each payment gateway.

6.2.2. Decorator

This pattern, also known as Wrapper[1], is used when a class needs to add new functionality to an existing class without modifying its structure. It allows for the dy-

[1]The name Wrapper is also used for the Adapter pattern, they should not be confused.

```java
public interface Target {
    void operation(data);
}
public class Adaptee {
    public void specificOperation(specialData) {
        // Implementation of the specific operation
    }
}
public class Adapter implements Target {
    private Adaptee adaptee;
    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }
    @Override
    public void operation(data) {
        // Translate the data and call the specific operation of the adaptee
        specialData = translateData(data);
        adaptee.specificOperation(specialData);
    }

    private SpecialData translateData(Data data) {
        // Implementation of the data translation
    }
}
```

Código fuente 6: Example of the Adapter pattern.

namic addition of behavior to an object, not affecting the behavior of other objects of the same class. The object that is being decorated is called the `ConcreteComponent`, which is wrapped by an interface with the same methods as the component, called `Component`. The `Decorator` class also implements the `Component` interface and contains an instance of the `Component` that it decorates. All the methods of the decorator class call the corresponding methods of the decorated component. However, different concrete decorator classes inherit from the decorator class and can add new behavior before or after calling the methods of the decorated component. This pattern is shown in Figure 6.5.

This way, new behavior can be added to an object dynamically at runtime, without affecting the behavior of other objects of the same class. Multiple decorators can also be combined to add multiple layers of behavior to an object, using for instance `new ConcreteDecoratorA(new ConcreteDecoratorB(component))`. Removing also the last decorator from the stack of decorators is also possible just returning the decorated component. However, it can lead to a large number of classes and the big limitation is the order of the decorator sequence, as decorators should not depend on the order in which they are applied to the component.

6.2.3. Proxy

This pattern is used when a class needs to control access to another class, which is called the `RealSubject`. The `RealSubject` is wrapped by an interface called `Subject` with the same methods as the real subject. Lastly, a proxy class implements

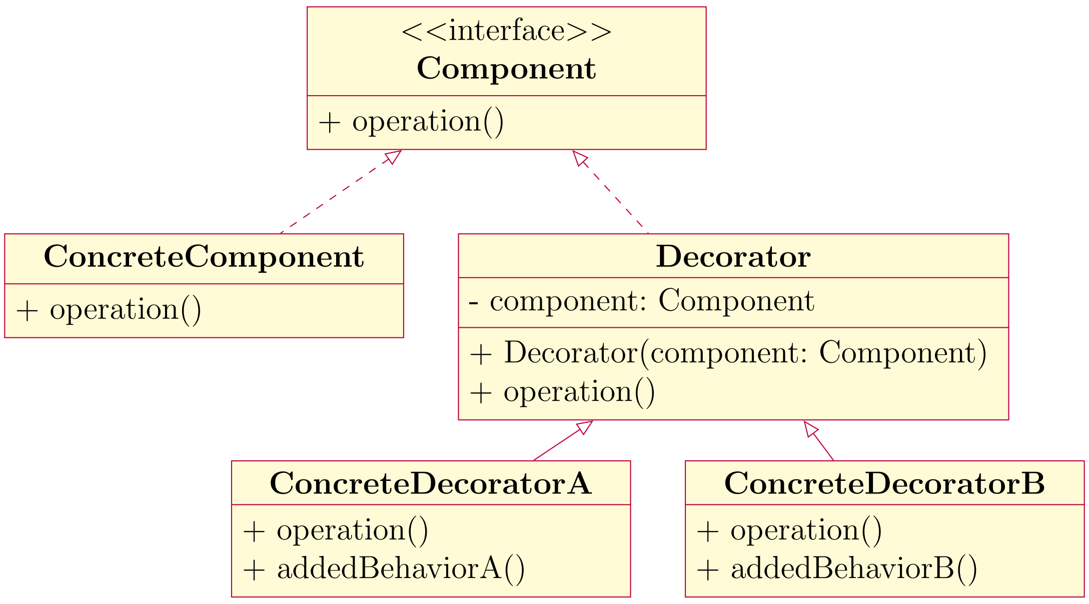

Figura 6.5: Decorator pattern.

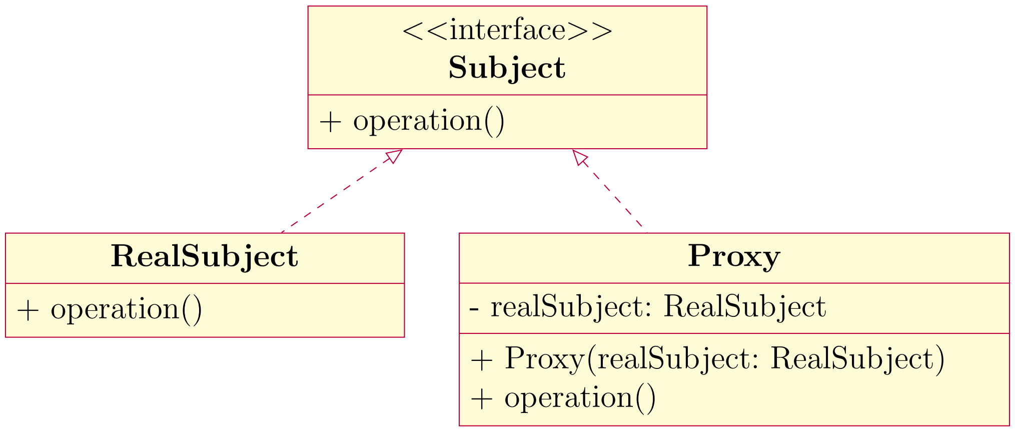

Figura 6.6: Proxy pattern.

the `Subject` interface and contains an instance of the real subject. The proxy class controls access to the real subject by implementing the same methods as the real subject, but adding additional behavior before or after calling the corresponding methods of the real subject. This pattern is shown in Figure 6.6.

This pattern can be used for various purposes, such as:

- Controlling access to a resource (Protection Proxy).

  This can be used to restrict access to a resource (database, file, etc.) based on certain conditions, such as user permissions or network availability. The proxy can check the conditions before allowing access to the real subject, and can deny access if the conditions are not met.

- Lazy initialization (Virtual Proxy)

  In this case, the proxy is used to delay the creation of the real subject until it is actually needed, improving performance and reducing resource usage. If an object is only used in certain conditions, the proxy can create the real subject

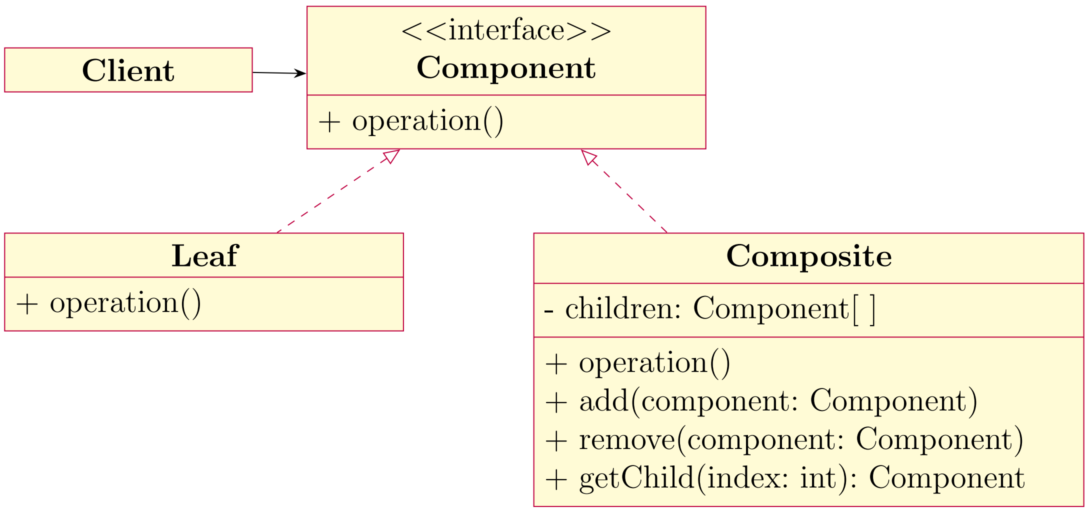

Figura 6.7: Composite pattern.

only when those conditions are met, rather than creating it at the time of the proxy's instantiation.

- Logging or monitoring (Logging Proxy)
- Caching request results (Caching Proxy)

  This can be used to cache the results of expensive operations, so that subsequent calls to the same operation return the cached result instead of re-executing the operation.

- Reference management

  The proxy can keep track of clients that obtained a reference to a heavyweight object or its results. From time to time, the proxy may go over the clients and check whether they are still active. If the client list gets empty, the proxy might dismiss the service object and free the underlying system resources.

This helps separating the concerns of the real subject and the additional behavior added by the proxy, making the code more modular and easier to maintain. However, latency is possibly added to the calls, and when the subject is not reachable should be taken into account.

6.2.4. Composite

This pattern is used when a class needs to represent a part-whole hierarchy, where individual objects and compositions of objects are treated uniformly. It allows for the creation of complex structures by combining simple objects into tree-like structures. The main component of this pattern is the `Component` interface, which defines the common methods for both individual objects (called `Leaf`) and compositions of objects (called `Composite`). Both classes implement the `Component` interface, allowing them to be treated uniformly by clients. The methods in the `Composite` class typically recursively run the method on all its children, which can be either `Leaf` or other `Composite` objects. This pattern is shown in Figure 6.7.

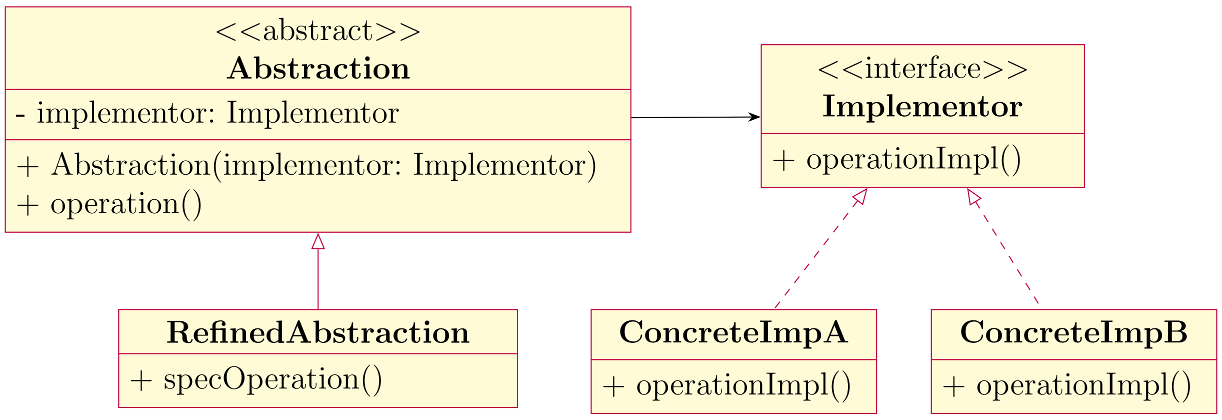

Figura 6.8: Bridge pattern.

Regarding where are the methods for adding, removing, and getting children defined, there are two common approaches:

- Design for transparency: These methods are defined in the `Component` interface, which means that both `Leaf` and `Composite` classes must implement them. Transparency here is respected, as clients can treat both individual objects and compositions of objects uniformly, without needing to know whether they are dealing with a leaf or a composite. However, this approach leads to LSP violation and possible ISP violation, as the `Leaf` class may not need to implement these methods and may throw exceptions if they are called.

- Design for safety: These methods are defined only in the `Composite` class, which means that only composite objects can have children. This approach respects LSP and ISP, but the client will have to check whether it is dealing with a leaf or a composite before calling these methods, which can lead to more complex and less maintainable code.

6.2.5. Bridge

This pattern is used when a main class needs to be developped in two independent dimensions[2], and each dimension has different variations. usually, there is a main dimension (called the abstraction) and a secondary dimension (called the implementation). In a single hierarchy with n abstractions and m implementations, there would be $n \times m$ classes. However, by using the bridge pattern, we can reduce this to one abstract main class (the abstraction), one interface (the implementor), n concrete abstractions, and m concrete implementors, resulting in a total of $n+m+2$ classes. The abstraction class contains a reference to the implementor interface, and the concrete abstractions and concrete implementors can be developed independently of each other. This pattern is shown in Figure 6.8.

A possible example of this pattern is when we have a software application that needs to support multiple platforms (e.g., Windows, macOS, Linux) and multiple user interfaces (e.g., graphical, command-line). The abstraction would be the user

[2] A higher number of dimensions is also possible just adding more bridges, but the most common case is when there are 2 dimensions.

interface, with its concrete abstractions being the different types of user interfaces (e.g., `GraphicalUI`, `CommandLineUI`), and the implementor would be the platform, with its concrete implementors being the different platforms (e.g., `WindowsPlatform`, `MacOSPlatform`, `LinuxPlatform`). This way, we can develop the user interfaces and the platform implementations independently of each other, and easily add new user interfaces or platforms without needing to modify existing code.

Some of the advantages of this pattern are the separation of concerns between the abstraction and the implementation, the ease of adding new abstractions and implementations without modifying existing code, and the ability to change the implementation at runtime by changing the reference to the implementor in the abstraction.

6.2.6. Facade

A facade is a design pattern that provides a simplified interface to a complex system (usually a library or a set of classes[3]). It acts as a front-facing interface that hides the complexities of the underlying system and provides a more user-friendly interface for clients to interact with. The facade class typically contains methods that wrap the functionality of the underlying system, allowing clients to access the features of the system without needing to understand its internal workings. This pattern is useful for reducing coupling between clients and complex systems, and for providing a clear and concise interface for clients to use.

6.2.7. Flyweight

This pattern is used when saving RAM is esential, and it is done by sharing common parts of the state between multiple objects, rather than storing the same data in each object. In order to do so, the common parts of the state are separated from the unique parts of the state:

- The common parts of the state are called the *intrinsic state*, and they are stored in a shared object called the `Flyweight`.
- The unique parts of the state are called the *extrinsic state*, and they are stored in the client objects that use the flyweight objects. They are passed to the flyweight objects when they are needed, rather than being stored in the flyweight objects themselves.

This is really used in text editors, where each character can be represented as a flyweight object that contains the intrinsic state (e.g., the character code, font, size) and the extrinsic state (e.g., the position of the character in the document) is stored in the client objects that use the flyweight objects. This way, we can save a lot of memory by sharing the intrinsic state between multiple characters, while still allowing for unique extrinsic state for each character. However, it is really complex to implement correctly, as it requires careful management of the shared state and the extrinsic state, and it can lead to performance issues if not used properly.

[3]Usually not a replacement for a single class. That would be more similar to an adapter.

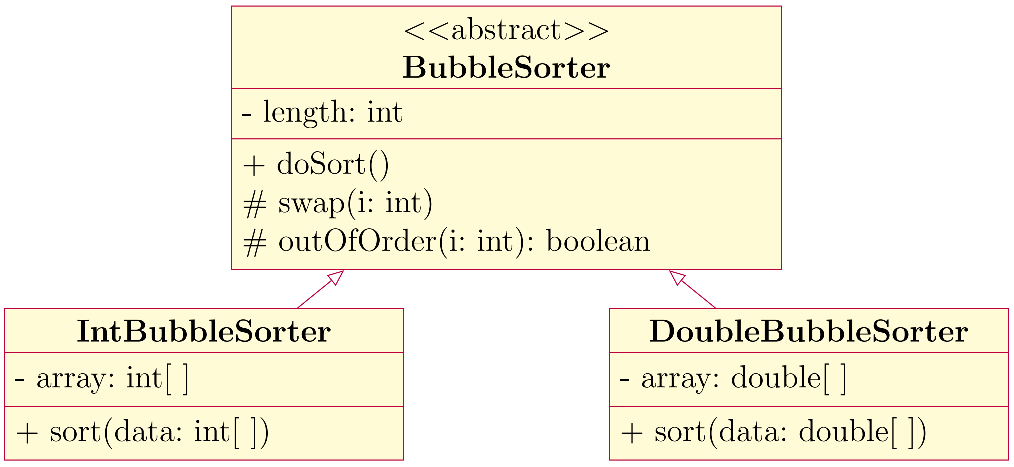

Figura 6.9: Example of the Template Method pattern.

6.3. Behavioral Patterns

Behavioral patterns are concerned with the control and the adaptation of the behavior of objects.

6.3.1. Template method

This pattern is used to separate a generic algorithm from its detailed context using *inheritance*. The generic algorithm is defined in a method called the `template method`, which is implemented in an abstract class. The detailed context is provided by concrete subclasses that implement the specific steps of the algorithm. This pattern allows for the reuse of the generic algorithm while allowing for flexibility in the specific steps of the algorithm, as different concrete subclasses can provide different implementations for those steps. A concrete example of this pattern is shown in Figure 6.9.

6.3.2. Strategy

This pattern is used to solve the same problem as the template method pattern, but using *composition* instead of inheritance. The main class has a dependency injection to an interface (called handler) that contains the details. An example is shown in Figure 6.10.

Although both patterns solve the same problem, the strategy pattern is more flexible and promotes better code reuse, as it allows to reuse the same handler for other generic algorithms (in this case, it could be used in other sorting algorithms), whereas the template method pattern binds the specific implementation to the generic algorithm, which can lead to code duplication. In addition, the strategy pattern allows for changing the behavior of the algorithm at runtime by changing the handler, while the template method pattern requires creating new subclasses to change the behavior of the algorithm. However, the template method pattern can be simpler to implement and understand in some cases.

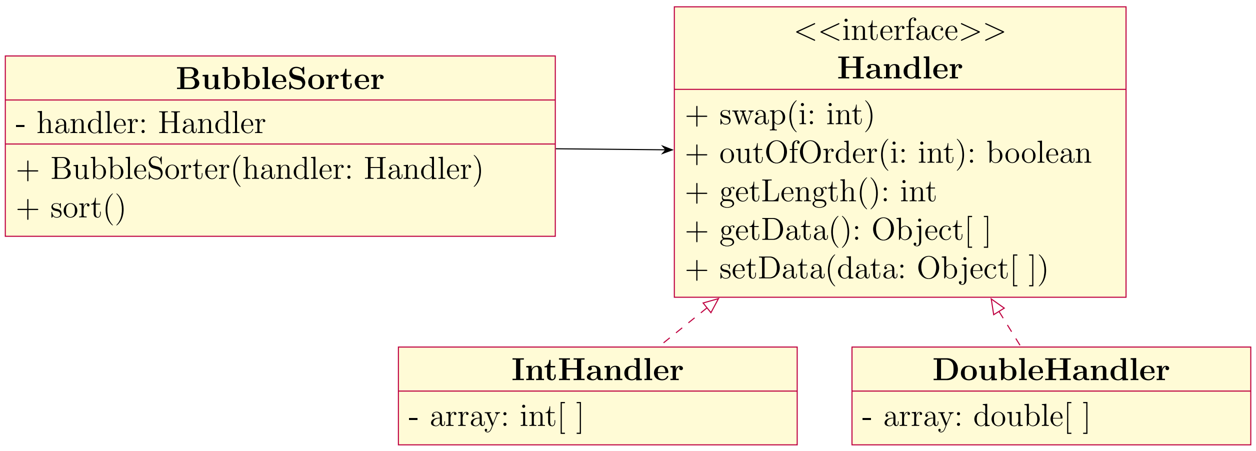

Figura 6.10: Example of the Strategy pattern.

6.3.3. Null object

This pattern is used to hide the absence of an object without using null references (which can lead to multiple checks for null and code more difficult to understand). With this aim, a null object is created, which is an instance of a class that implements the same interface as the real object, but with empty or default behavior. The client code can then use the null object instead of checking for null references, leading to cleaner and more maintainable code. However, it can ease the hiding of errors when they should be exposed.

6.3.4. Command

This pattern is used to decouple initiating an action from executing it (for instance, to queue a request or execute it remotely). It also allows to redo or undo an action. The main components of this pattern are:

- The `Invoker` class, which is responsible for initiating the request (not the action). They typically have a method for setting the command to be executed and a method for executing the command. For instance, they could represent buttons in a UI.

- The `Command` interface, which defines the method for executing the command. If undo functionality is needed, it can also store the state of the system before executing the command and define a method for undoing the command.

- The `ConcreteCommand` classes, which implement the `Command` interface and define the specific actions to be executed. It does not execute the action itself, but rather calls the corresponding methods on the receiver object to perform the action. In case that the command needs parameters, they should be configured by the client and stored here.

- The `Receiver` class, which is the object that performs the actual action when the command is executed. It typically has methods that correspond to the actions that can be performed.

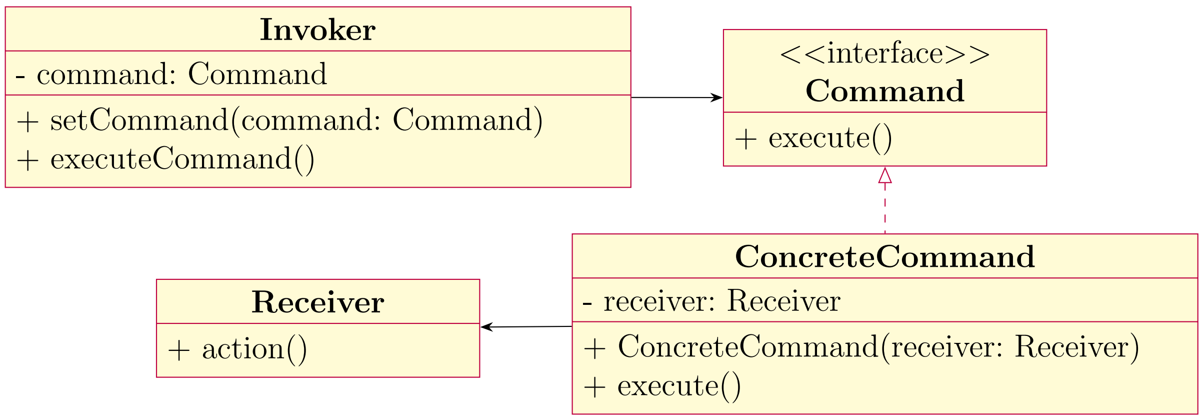

Figura 6.11: Command pattern.

As already explained, this pattern can be used in several ways, given that there is an object (`Command`) that represents the action itself that should be taken.

- In order to queue requests: a list can be stored with the commands to be executed later.
- In order to execute requests remotely: the command object can be serialized and sent over the network to be executed on a different machine.
- In order to implement undo functionality: the command object can store the state of the system before executing the command, and define a method for undoing the command that restores the previous state.
- In order to correctly implement undo/redo functionality: an stack can be used to keep track of the executed commands, allowing for multiple levels of undo and redo.
- In order to implement logging functionality: the command object can log the details of the command being executed, such as the parameters and the time of execution, which can be useful for debugging and auditing purposes.
- In order to implement macros: a macro command can be created, which is a command that contains a list of other commands to be executed in sequence. This allows for the creation of complex commands that can be executed with a single action.

This pattern is shown in Figure 6.11. An important disavantage of this pattern is that it may be difficult to detect who and why started an action. However, it allows an elegant approach to different problems, al already explained. In addition, the OCP is respected by simply adding new command classes, and the SRP is also respected decoupling classes that invoke the action from the classes that actually perform them.

6.3.5. Observer

This pattern is used when a class (called the `Subject`) needs to notify other classes (called `Observers`) about changes in its state. This is for example used in

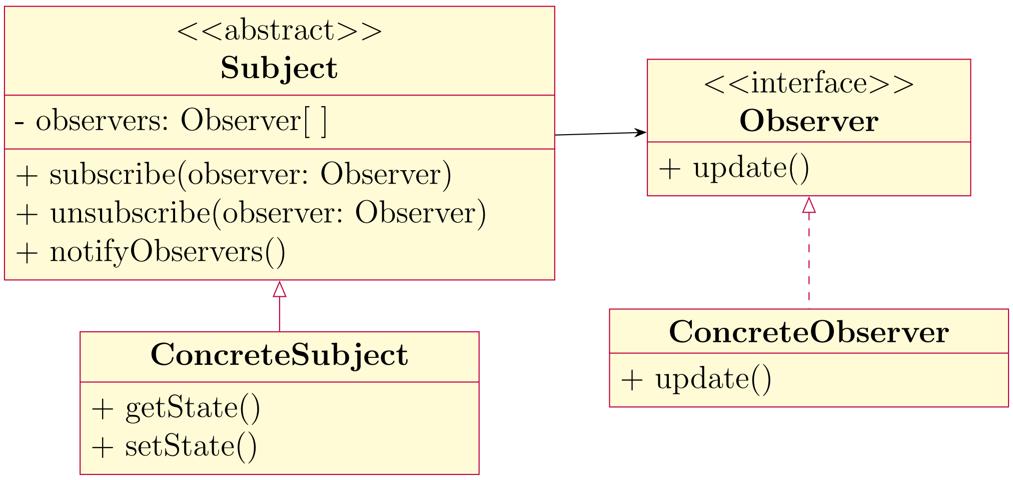

Figura 6.12: Observer pattern.

the MVC pattern by the `Model` class, which notifies the `View` classes when its state changes. The main components of this pattern are:

- The `Subject` class (also called `Publisher`), which maintains a list of its observers and provides methods for attaching, detaching, and notifying observers. This is usually an abstract class, as the specific implementation of the subject may vary.

- The `Observer` class (also called `Subscriber`), which defines the method for receiving notifications from the subject. This is usually an interface.

It also has an alternative approach: if the subject needs to notify the observers really often, it can be more efficient to use a polling mechanism, where the observers periodically check the state of the subject for changes, rather than the subject actively notifying the observers. This approach can reduce the overhead of notifications, but it can also lead to increased latency in detecting changes and may not be suitable for all use cases.

This pattern is shown in Figure 6.12. Some of its advantages are that the OCP is respected, as dinamically changing the observers is allowed. However, the notification order is not guaranteed, which may lead to problems if observers need to consume any data.

6.3.6. Visitor

This pattern is used to separate an algorithm from the objects on which it operates. It allows to perform an operation on all element of a collection (for example, a tree) but want to separate the operation from the elements. For instance, in a Composite Tree (where not all the nodes have the same type), we may want to perform an operation on all the nodes, but we want to separate the operation from the nodes. The main components of this pattern are:

- The `Visitor` interface defines a `visit` method for each type of element in the object structure. This allows the visitor to perform different operations

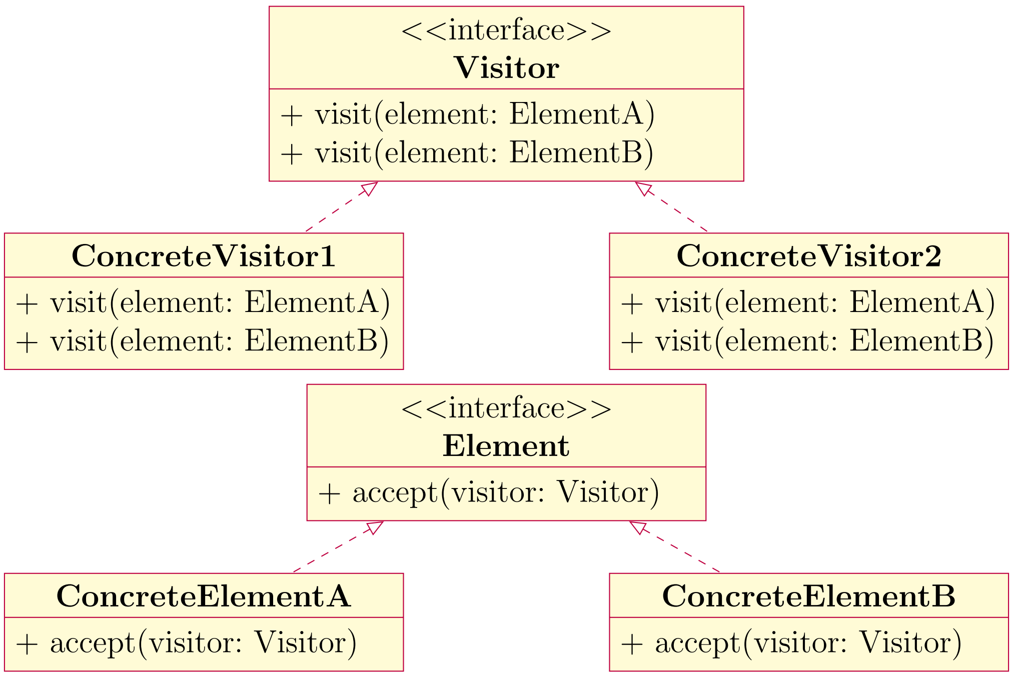
<<interface>>
Visitor
+ visit(element: ElementA)
+ visit(element: ElementB)
ConcreteVisitor1
+ visit(element: ElementA)
+ visit(element: ElementB)
ConcreteVisitor2
+ visit(element: ElementA)
+ visit(element: ElementB)
<<interface>>
Element
+ accept(visitor: Visitor)
ConcreteElementA
+ accept(visitor: Visitor)
ConcreteElementB
+ accept(visitor: Visitor)

Figura 6.13: Visitor pattern.

on different types of elements. If the programming language does not support method overloading, different methods can be defined for each type of element.

- Each concrete visitor class represents a specific operation to be performed on the elements of the object structure. It implements the `Visitor` interface and provides the implementation for each `visit` method, defining the specific behavior for each type of element.
- The `Element` interface defines an `accept` method that takes a visitor as an argument.
- Each concrete element class implements the `Element` interface and provides the implementation for the `accept` method. The `accept` method typically calls the corresponding `visit` method on the visitor, passing itself as an argument. This allows the visitor to perform the operation on the corresponding element.

This pattern is shown in Figure 6.13. Some of its advantages are that it allows to add new operations without modifying the existing element classes, which respects the OCP. However, every operation needs to know about every data structure, which can lead to a data structure difficult to extend. In addition, visitors typically need a lot of information about visited objects, possibly breaking encapsulation.

6.3.7. State

This pattern is used to separate the behavior of an object from its state. It allows an object to change its behavior when its internal state changes, without needing to modify the object's class. This avoids case distinction in the code, which leads to cleaner code.

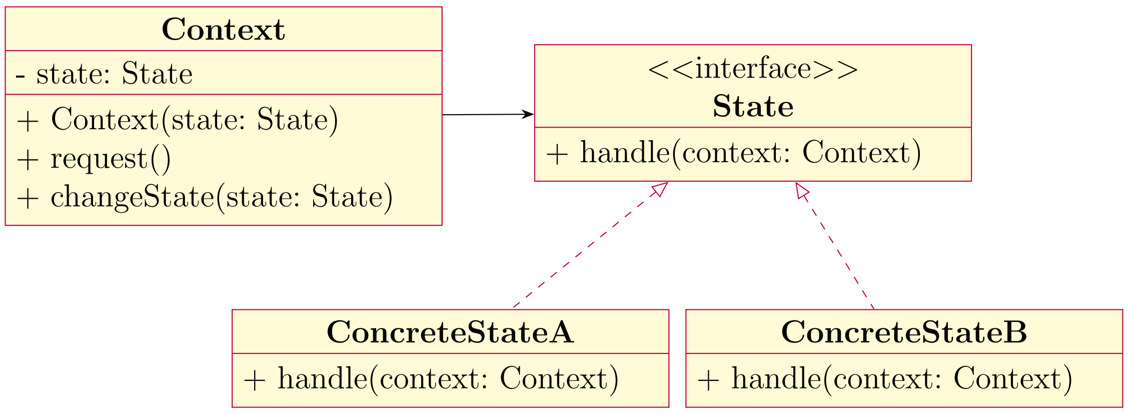
Context
- state: State
+ Context(state: State)
+ request()
+ changeState(state: State)
<<interface>>
State
+ handle(context: Context)
ConcreteStateA
+ handle(context: Context)
ConcreteStateB
+ handle(context: Context)

Figura 6.14: State pattern.

The main class is usually called `Context`, and it has a reference to an interface called `State`, which represents the different states of the object. The `Context` class is programmed as usual, and the methods that depend on the state are added to the `State` interface. Each concrete state should implement the `State` interface and provide the specific behavior for that state. Two aspects should be taken into account:

- Changing the state should be possible. Whether the state change is triggered by the context or by the state itself, the context should have a method for changing the current state.
- The context is usually passed to the state' methods so that they can access variables with lifetimes longer than state objects.

This pattern is shown in Figure 6.14. Some of its advantages are that it allows to add new states without modifying the existing code, which respects the OCP. In addition, it can lead to cleaner code by avoiding case distinction based on the state; and state transitions are explicit.

6.3.8. Iterator

The iterator pattern is used to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It allows for the traversal of a collection of objects without needing to know the details of how the collection is implemented.

An interface called `Iterable` defines a method for creating an iterator object, which is responsible for traversing the collection. Every collection class that implements the `Iterable` interface must provide an implementation for this method, which typically returns an instance of a class that implements the `Iterator` interface. The `Iterator` interface defines methods for checking if there are more elements to traverse (e.g., `hasNext()`) and for retrieving the next element in the collection (e.g., `next()`). This pattern allows for a consistent way to traverse different types of collections, and it promotes encapsulation by hiding the internal structure of the collection from the client code. This pattern is shown in Figure 6.15.

Some aspects that should be taken into account are:

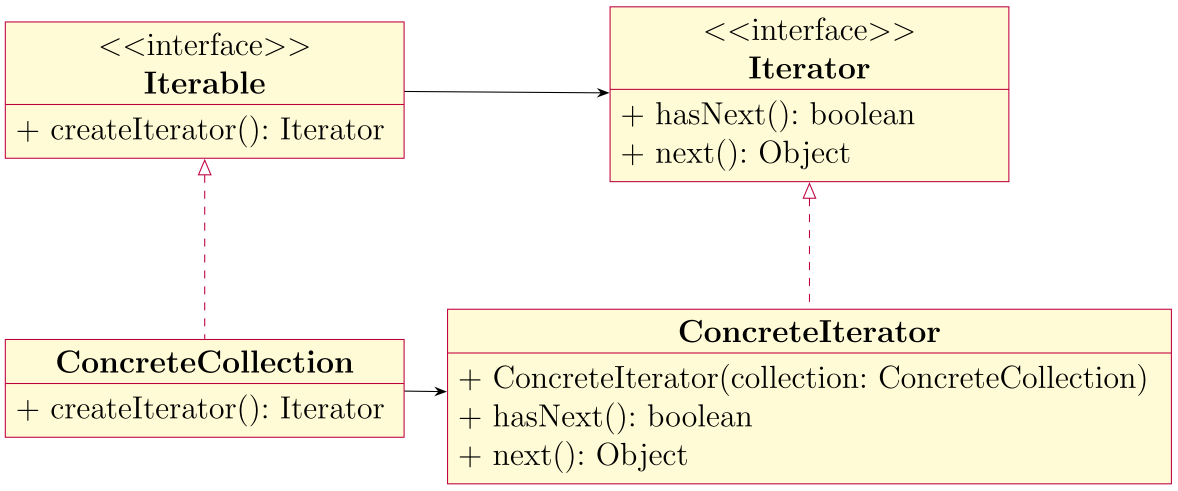

Figura 6.15: Iterator pattern.

- Using proxies, a Lazy Evaluation is possible by creating an iterator that only retrieves the next element when the `next()` method is called, rather than retrieving all elements at once. This can be useful for large collections or for collections that are expensive to retrieve.
- When a Null Object is needed to denote that there are no elements in a collection, it can be implemented as a special case of the iterator that always returns false for `hasNext()` and throws an exception for `next()`.

Some advantages of this pattern are that it promotes encapsulation by hiding the internal structure of the collection from the client code, and it allows for a consistent way to traverse different types of collections. However, depending on the programming language it can be difficult to implement correctly.

6.3.9. Other behavioral patterns

Other behavioral patterns include:

- Chain of Responsibility
- Interpreter
- Mediator
- Memento