Inhaltsverzeichnis

1. OOP Design Prinzipien

1.1. Single Responsibility Principle

The **Single Responsibility Principle** (SRP) states that each class should have a single responsibility and that responsibility should be encapsulated by the class. This means that classes should not be responsible for more than "one" thing, as this can lead to code that is difficult to maintain and debug. By adhering to SRP, developers are able to create code that is easier to understand and modify.

1.2. Open/Closed Principle

The **Open-Closed Principle** (OCP) states that classes should be open for extension but closed for modification. This means that once a class has been written, it should not need to be modified in order to add new features or functionality. Instead, additional features should be added through inheritance or composition. Adhering to OCP helps to ensure that existing code remains unchanged while still allowing for flexibility when adding new features.

1.3. Liskov’s Substitution Principle

The **Liskov Substitution Principle** (LSP) states that derived classes should be substitutable for their base classes. This means that any object of a derived class should be able to be used wherever an object of its base class is expected without causing any errors. Adhering to LSP helps to ensure that objects behave as expected and reduces the risk of unexpected behavior.

1.4. Interface Segregation Principle

The **Interface Segregation Principle** (ISP) states that clients should not be forced to depend on methods they do not use. This means that interfaces should be broken down into smaller, more specific pieces so that only the necessary methods are exposed. Adhering to ISP helps to reduce complexity and makes it easier to extend and maintain code.

1.5. Dependency Inversion Principle

The **Dependency Inversion Principle** (DIP) states that high-level modules should not depend on low-level modules, but rather both should depend on abstractions. This means that instead of having concrete implementations of classes, developers should use abstractions such as interfaces or abstract classes. Adhering to DIP helps to decouple components from each other, making them easier to test and maintain.

1.6. Don’t Repeat Yourself Principle (DRY)

DRY is a programming principle that encourages developers to reduce **repetition** of code, and instead use abstraction and modularization to make the code more maintainable. This means that rather than writing multiple lines of code for similar tasks, you can create functions or classes that can be reused throughout your program. By following this principle, you can save time and effort when coding, as well as making it easier to debug and modify existing code.

1.7. Keep it simple and stupid (KISS)

A phrase just saying that when writing an application, implementation should start **small, easy and lightweight**, and everything, that is not needed (for the moment) shall be basically ignored and implemented later. But, that does not protect developers from "considering" future requirements that maybe affect the overall architecture.

Approaches like "Test Driven Development" support this principle strongly, because tests can be written with a "fail fast" approach and can be extended step by step.

1.8. Composition Over Inheritance Principle

Composition is a type of relationship between two objects where one object contains the other. This means that an object can be composed of multiple parts, each with its own behavior and attributes. The advantage of composition over inheritance is that it allows for greater flexibility in how objects are structured.

On the other hand, **inheritance** is a type of relationship between two objects where one object inherits the properties and behaviors of another. Inheritance is useful when you want to reuse code from an existing class without having to rewrite it.

1.9. Loose coupling between components

Loose coupling is a principle that helps to reduce the complexity of code by decoupling components from each other. This means that changes made to one component will not affect any other components, which makes it easier to maintain and debug the code.

1.10. Immutability

Passing immutable data between objects (or systems) is one of the most common, but "boring" tasks in many Java applications. Prior to Java 14, this required the creation of a class with boilerplate fields and methods. With the release of Java 14, **records** were introduced. In many cases where simple data shall be transferred, such data shall be immutable, since immutability ensures the validity of the data without synchronization.

To accomplish immutability, class should implement the following aspects:

- private, final field for each piece of data
- getter for each field
- public constructor with a corresponding argument for each field
- equals method that returns true for objects of the same class when all fields match
- hashCode method that returns the same value when all fields match
- toString method that includes the name of the class and the name of each field and its corresponding value

Records replace these repetitious data classes. They are "naturally" immutable data classes that require only the type and name of fields.

The equals, hashCode, and toString methods, as well as the private, final fields and public constructor, are generated by the Java compiler.

1.11. Document your public API

see → `Course 6`

2. OOP Best Practices & Code Smells

A selection of best practices:

Meaningful Names

Very important. Variable names should express precisely their meaning.

Fewer Arguments

More than 3 method parameter or constructor arguments indicate a so-called code smell. Such a finding can indicate the violation of the principle "single responsibility" as well as the KISS principle.

Verify Arguments

This is a good practice anyway. Validation subject data in methods or algorithms is always recommended. These checks (assertions) can be functional (e.g. checks for *value ranges*) as well as technical (e.g. checks for *null*).

Avoid using constructors extensively

Constructors a good to create instances in a clear and understandable way. But, together with the above-mentioned "fewer arguments" recommendation, the opposite can happen, that is complex instanciation with a lot of prerequisites. There is a pattern that addresses this smell named *factory pattern*, which is used to create instances of a class via a factory class.

Interfaces instead of abstract classes

Interfaces are more flexible than abstract classes because they allow for multiple inheritance. Interfaces also provide better abstraction than abstract classes. An interface defines the behavior of an object without specifying how it should be implemented, and does it not itself. Interfaces also make it easier to test code. Since the interface does not specify any implementation details, it's easy to mock out objects in unit tests. This makes it much simpler to write automated tests for code that uses interfaces.

Use specific (checked) exceptions judiciously

See also → `Course 6`

Reducing Conditional statements (Zyklomatische Komplexität)

Cyclomatic complexity is a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code. It was developed by Thomas J. McCabe, Sr. in 1976.

Every code shows an *impression* of its complexity, and the calculable metric can be checked, in real life projects most often via **"Code Metric Tools"**.

An example:

```
// Summe aller "Bedingungen" = 6
// +1
public boolean monitor(Train train, MonitoringSubject subject) {

    // +1
    if (monitoringIsEnabled) {

        String number = train.number();
        // +2
        if (isTrainObservable(number) && butIgnoreIf(train)) {

            switch(subject) {
                // +1
                case Speed -> {
                    observeSpeed(train);
                    return true;
                }
```

```
                // +1
                case Delay -> {
                    observeDelay(train);
                    return true;
                }
                default ->
                        throw new IllegalArgumentException(
                                "Invalid enum value: " + subject);
            }

        } else {
            return false;
        }
    }
    return false;
}
```

Polymorphism over switch statements (Strategy pattern)

Polymorphism is a powerful tool in OOP that allows for the same code to be used with different types of objects. This means that instead of writing separate switch statements for each type of object, you can write one piece of code that works with all of them. This makes your code more efficient and easier to maintain since you don't have to keep track of multiple switch statements.

A small example:

```
Printer printer = new Printer();
Context context = new PrintContext();
printer.print(Format.Xml, context);
```

Implementation:

```
public void print(Format format, Context what) {
  switch (format) {
    case Xml  -> printXml(what);
    case Json -> printJson(what);
    default   -> printPlain(what);
  }
}
```

See also → `module-pattern` (Strategy Pattern)

Favor generics, lambdas over proprietary implementation

Use modern language features if applicable. They are most often better in general, better maintainable and more efficient than self-made implementations, just like Lambda statements that help to replace loops and list queries.

Apply design patterns to known problems

Design patterns are reusable solutions to common programming problems, and they provide a way for developers to structure their code in an organized and efficient manner. By using design patterns, developers can create code that is easier to read, understand, debug, and maintain. Design patterns also help to ensure that the code is more consistent across different applications and platforms, which makes it easier to reuse components and libraries.

See also → `module-pattern`