Design Patterns

Strategy

COMP3607
Object Oriented Programming II

Week 8

Outline

- Design Patterns
  - Strategy
  - Template
- Code Smells
  - Object-Orientation Abusers

3 Strategy Design Pattern

The Strategy design pattern is an object behavioural pattern.

It defines a family of algorithms, encapsulates each one, and make them interchangeable.

Strategy lets the algorithm vary independently from the clients that use it.

This is achieved by capturing the abstraction in an interface, and burying implementation details in derived classes.

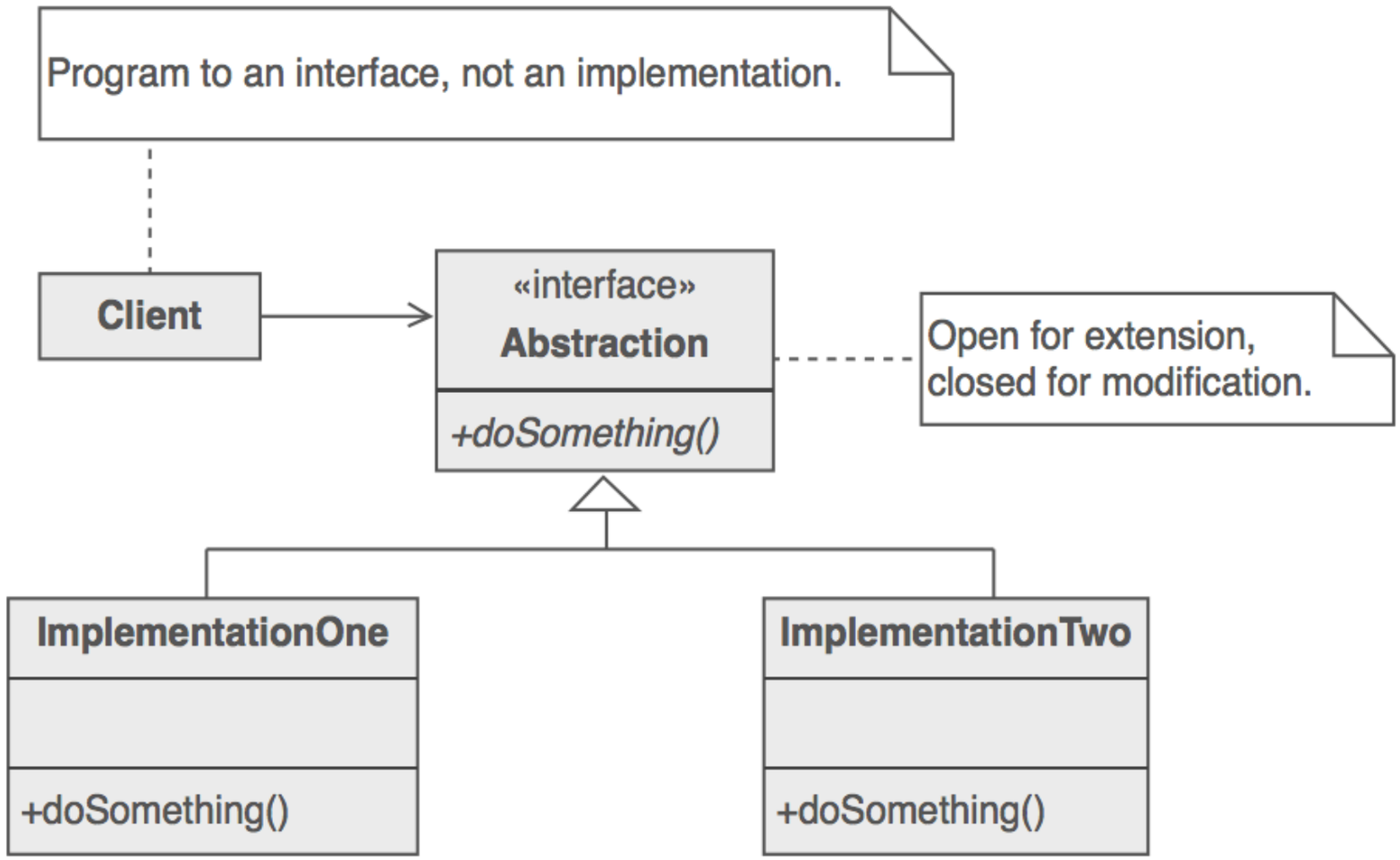
Abstract Coupling

One of the dominant object-oriented design principles is the "Open-Closed principle".

This is routinely achieved by encapsulating interface details in a base class, and putting implementation details in derived classes.

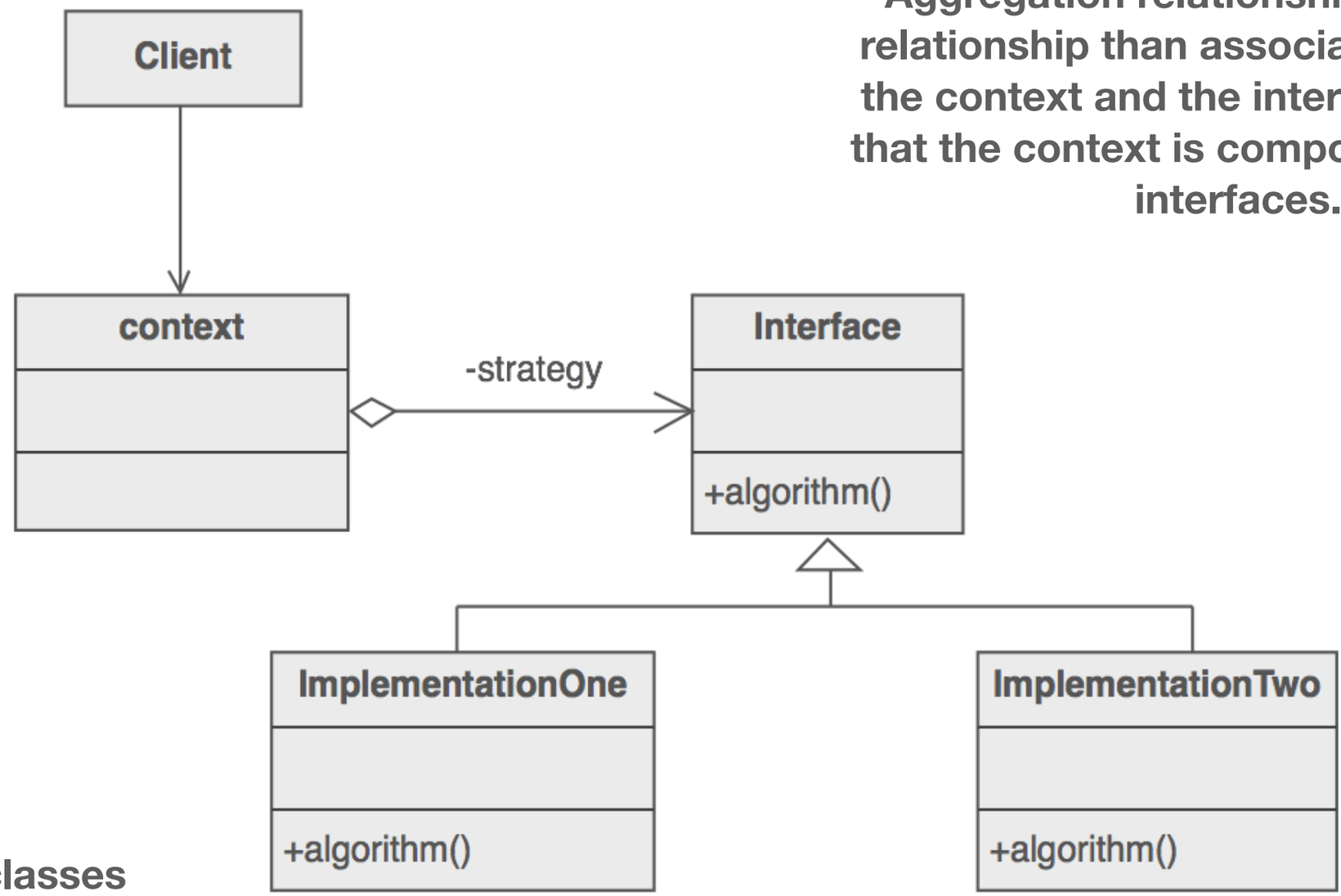
Clients can then couple themselves to an interface, and not have to experience the upheaval associated with change: no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes.

Maximise Cohesion, Minimise Coupling



https://sourcemaking.com/design_patterns/strategy

Strategy (UML) Structure



Aggregation relationship (a stronger relationship than association) between the context and the interface indicates that the context is composed of several interfaces.

Concrete classes implement the interface

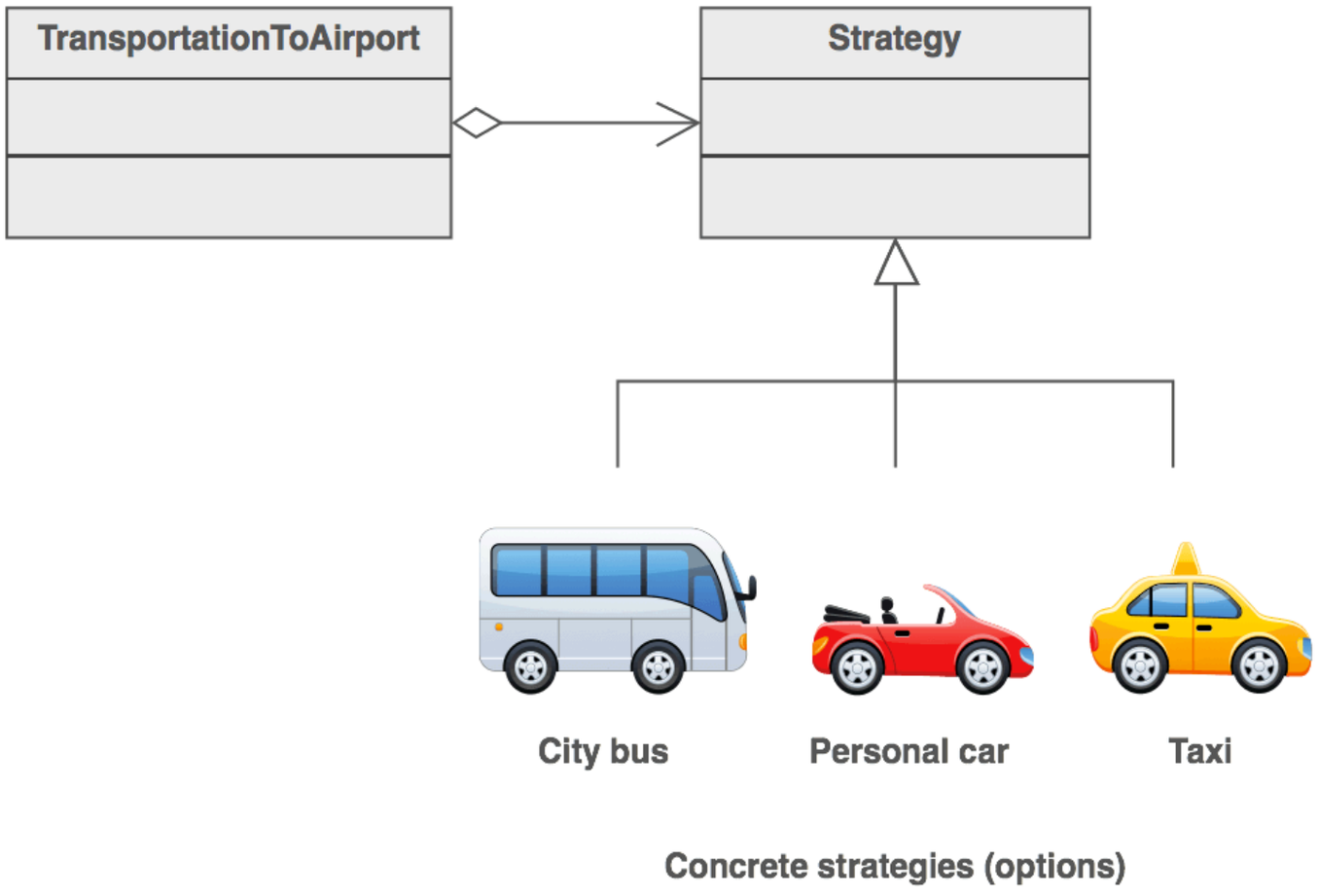
https://sourcemaking.com/design_patterns/strategy

Strategy Design Pattern
-Steps-

1. Define the interface of an interchangeable family of algorithms
2. Bury algorithm implementation details in derived classes
3. Clients of the algorithm couple themselves strictly to the interface

Example

Modes of transportation to an airport is an example of a Strategy.



Several options exist such as:

- driving one's own car
- taking a taxi
- an airport shuttle
- a city bus
- a limousine service.
- subways (some airports)
- helicopters (some airports)

Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably.

The traveler must chose the Strategy based on trade-offs between cost, convenience, and time.

3

Other Examples

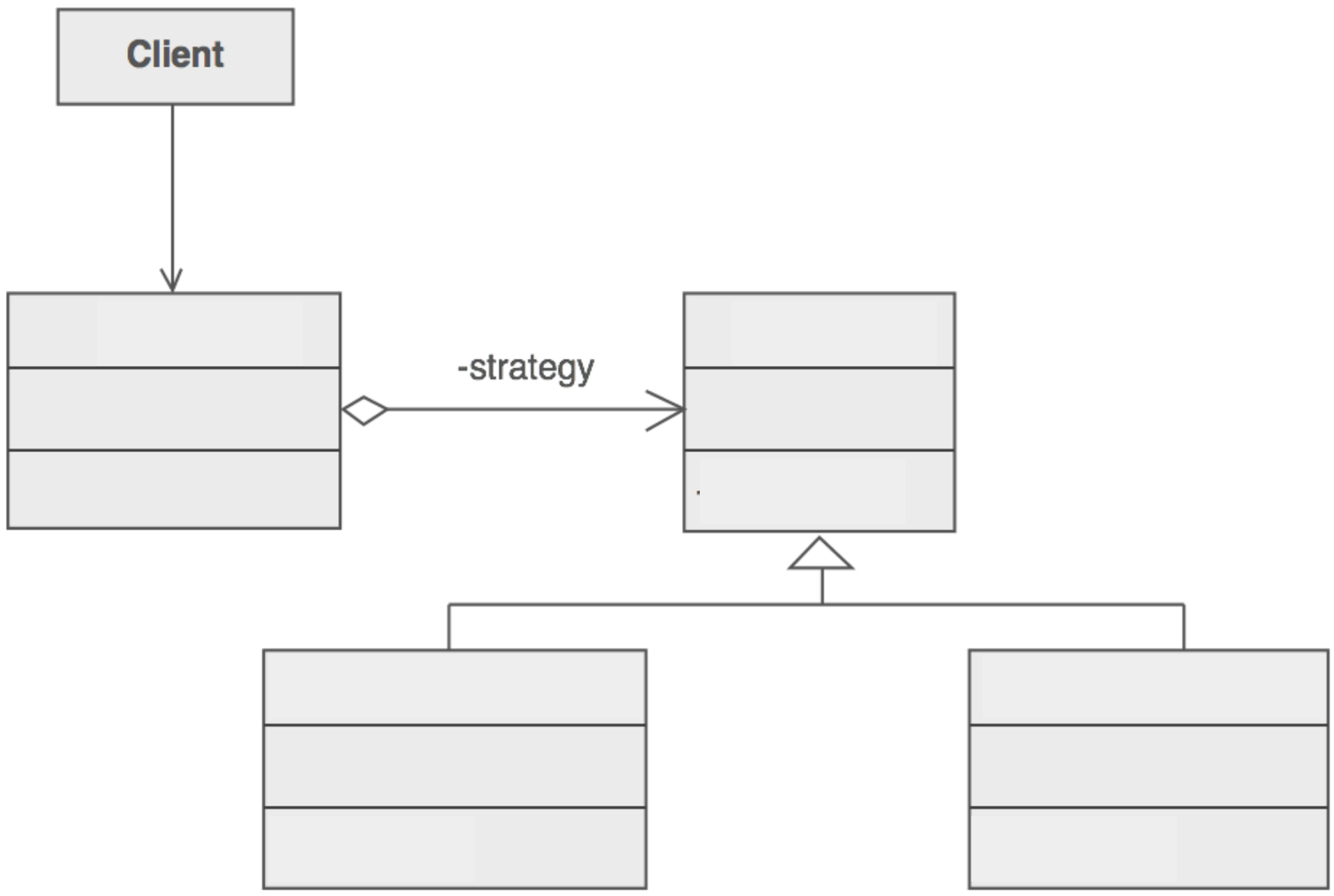
Other examples that use the Strategy pattern would be:

- saving files in different formats
- running various sorting algorithms
- different algorithms for file compression

Exercise

5 mins

Exercise: Use the Strategy Design Pattern to fill in the diagram for a file compression application where we can create either zip or rar files



https://dzone.com/articles/design-patterns-strategy

3

Code Example

```
//Strategy Interface
public interface CompressionStrategy {
  public void compressFiles(ArrayList<File> files);
}
```

```
public class ZipCompressionStrategy implements CompressionStrategy {
  public void compressFiles(ArrayList<File> files) {
    //using ZIP approach
  }
}
```

```
public class RarCompressionStrategy implements CompressionStrategy {
  public void compressFiles(ArrayList<File> files) {
    //using RAR approach
  }
}
```

https://dzone.com/articles/design-patterns-strategy

Code Example

```
1  public class CompressionContext {
2    private CompressionStrategy strategy;
3    //this can be set at runtime by the application preferences
4    public void setCompressionStrategy(CompressionStrategy strategy) {
5      this.strategy = strategy;
6    }
7
8    //use the strategy
9    public void createArchive(ArrayList<File> files) {
10     strategy.compressFiles(files);
11   }
12 }
```

```
1  public class Client {
2    public static void main(String[] args) {
3      CompressionContext ctx = new CompressionContext();
4      //we could assume context is already set by preferences
5      ctx.setCompressionStrategy(new ZipCompressionStrategy());
6      //get a list of files...
7      ctx.createArchive(fileList);
8    }
9  }
```

https://dzone.com/articles/design-patterns-strategy

Applicability

Use the Strategy pattern when:

- many related classes differ only in their behaviours
- you need different variants of an algorithm
- an algorithm uses data that clients shouldn't know about
- a class defines many behaviours and these appear to be multiple conditional statements in its operations.

Consequences: Strategy

Benefits and Liabilities:

- Families of related algorithms
- An alternative to subclassing
- Strategies eliminate conditional statements
- A choice of implementations

- Clients must be aware of different strategies
- Communication overhead between Strategy and Content
- Increased number of objects

Code Smells

“A surface indication that usually corresponds to a deeper problem in the software system” - Martin Fowler

Object-Orientation Abusers

All these smells are incomplete or incorrect application of object-oriented programming principles.

For example:

- Switch Statements
- Temporary Field
- Refused Bequest
- Alternative Classes with Different Interfaces

Switch Statements

You have a complex **`switch`** operator or sequence of **`if`** statements.

Switch Statements (or for that matter the if-else conditions) tend to mix logic and data together, which isn't exactly the best way to do it.

Each time the developer needs to change the logic or the data, he has a mess in his hands.

Classes should be Open for extensions, but Closed for modification (Open Closed Principle)

Temporary Field

There are times when a programmer decides to introduce fields in a class which are used only by one method.

Instead of passing it as method parameter, this is done to avoid the Long Parameter List (smell).

These fields are used only when the particular method is use under certain circumstances.

Outside of these circumstances, the fields are empty and sit idle in the class most of the time.

Refused Bequest

If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter.

The unneeded methods may simply go unused or be redefined and give off exceptions.

The subclass ignores the functionalities of parent class (refuses to implement parent behaviour) and overrides it.

Furthermore, this will also violate the Liskov Substitution Principle as the subclass class cannot replace the parent in code without affecting the functionality.

Alternative Classes with Different Interfaces

Two different classes perform identical functions but have different interfaces (different method names, signature etc).

Such situation arises mostly when there is lack of communication between team members who end up developing similar classes, or when the developer fails to check the available classes.

References

- Design Patterns: online reading resources
  - https://sourcemaking.com/design_patterns/strategy
  - https://dzone.com/articles/design-patterns-strategy
- Refactoring: Code Smells
  - https://refactoring.guru/refactoring/smells/oo-abusers\