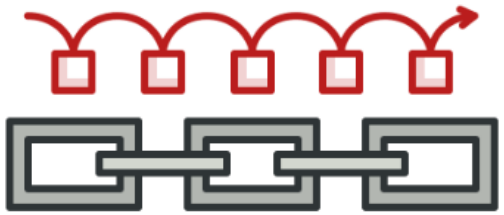
CSci 3081W: Program Design and Development

Lecture 10 – Behavioral Design Patterns
Strategy Pattern

Behavioral Design Patterns

Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects.

Behavioral Design Patterns



Chain of Responsibility

Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

We are not going over this one

Behavioral Design Patterns

Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

We are not going over this one

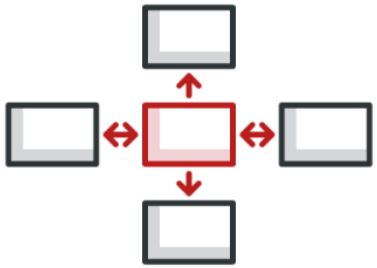
Behavioral Design Patterns

Iterator

Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

We are not going over this one

Behavioral Design Patterns

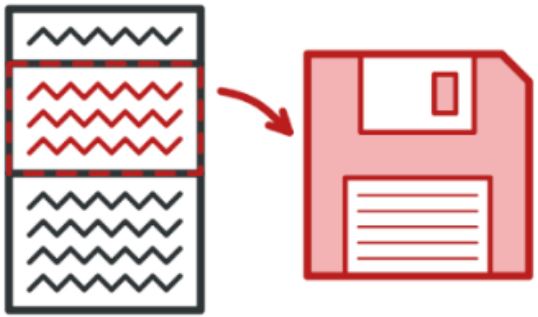


Mediator

Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

We are not going over this one

Behavioral Design Patterns

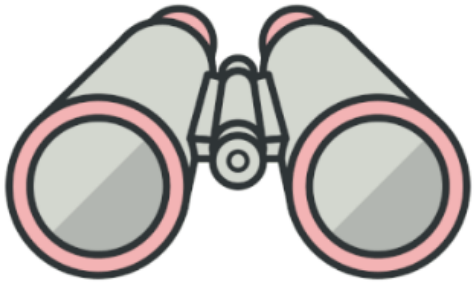


Memento

Lets you save and restore the previous state of an object without revealing the details of its implementation.

We are not going over this one

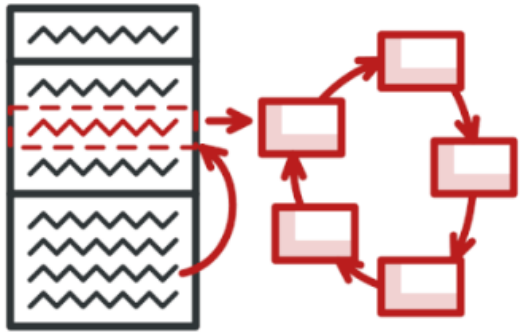
Behavioral Design Patterns



Observer

Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Behavioral Design Patterns

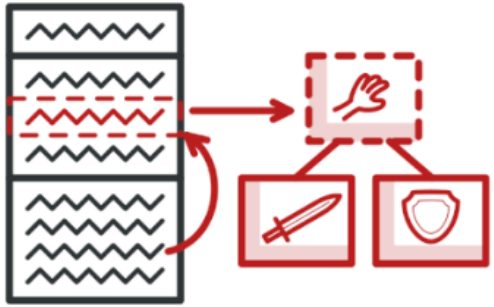


State

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

We are not going over this one

Behavioral Design Patterns



Strategy

Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

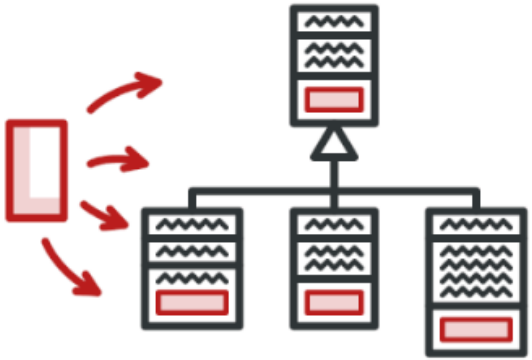
Behavioral Design Patterns

Template Method

Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

We are not going over this one

Behavioral Design Patterns



Visitor

Lets you separate algorithms from the objects on which they operate.

We are not going over this one

Strategy Pattern

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Problem

First Iteration –

Shortest Path Algorithm

Beeline

This is just an example, only tangentially related to your project

Problem

Second Iteration –

Wind is too strong for Beeline

A* on the roads

This is just an example, only tangentially related to your project

Problem

Third Iteration –

A* is too expensive

Some other algorithm

This is just an example, only tangentially related to your project

Problem

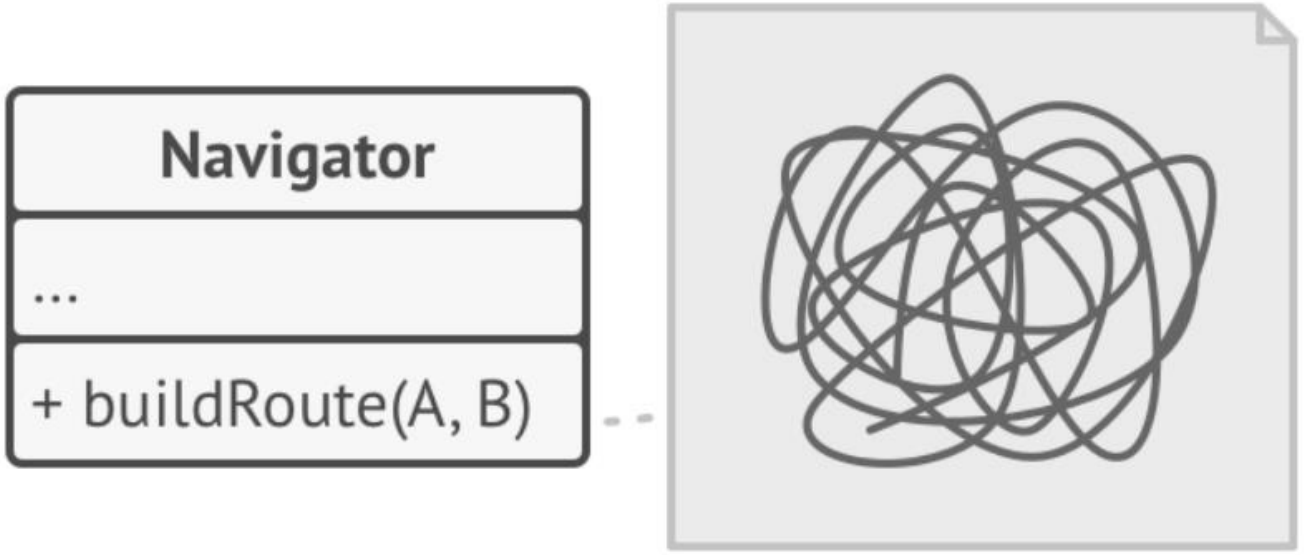
Nth Iteration –

Who knows

Algorithm N

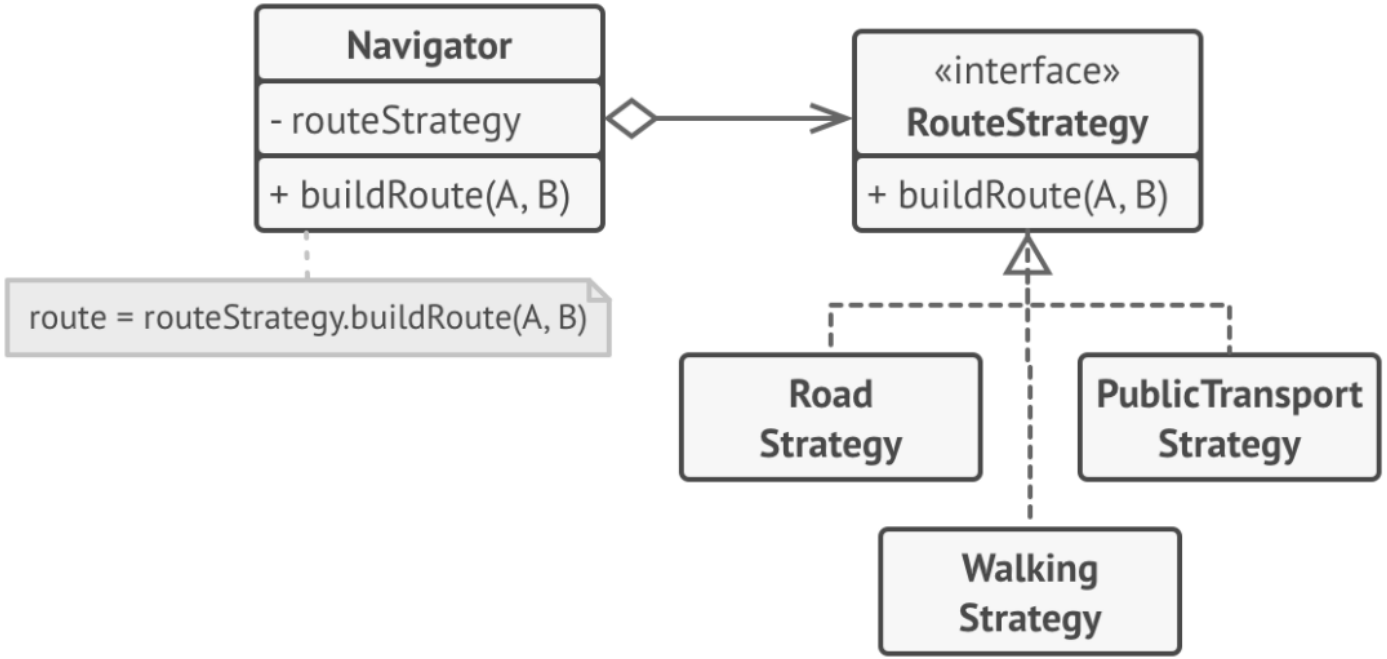
This is just an example, only tangentially related to your project

Problem



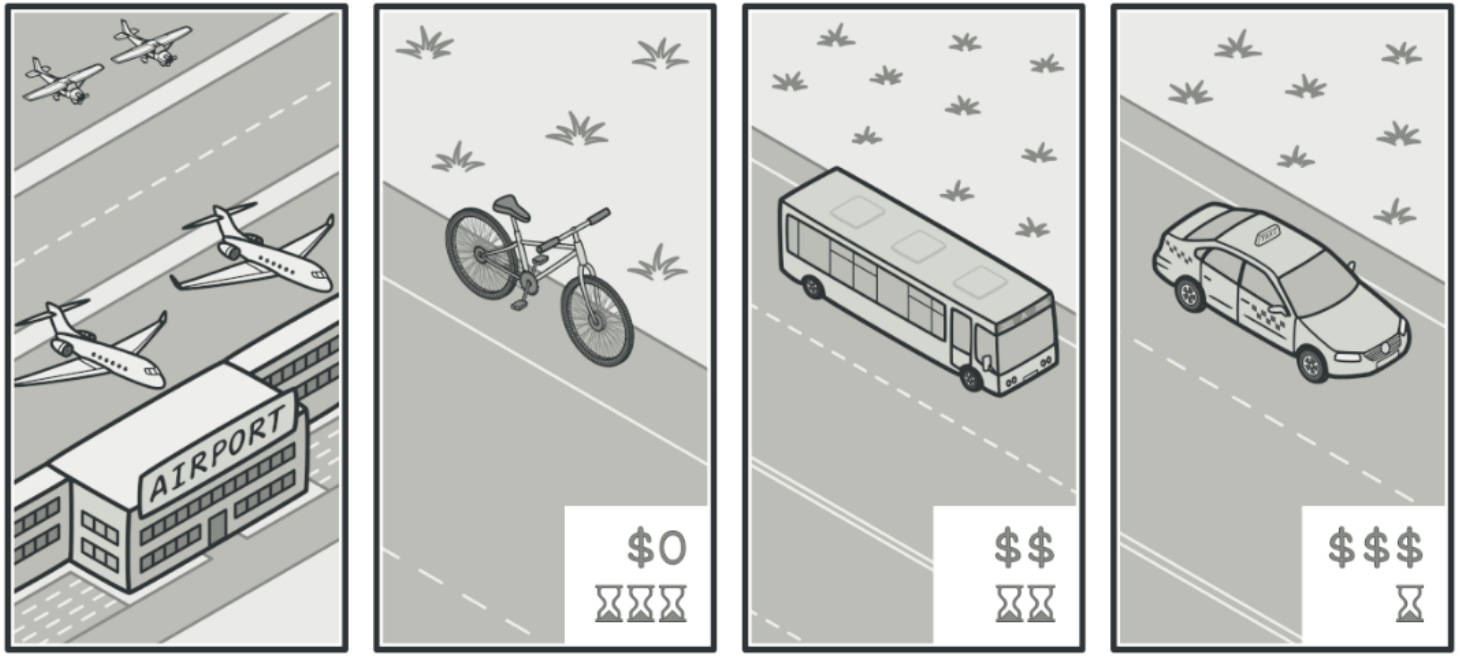
The code of the navigator became bloated.

Solution



Route planning strategies.

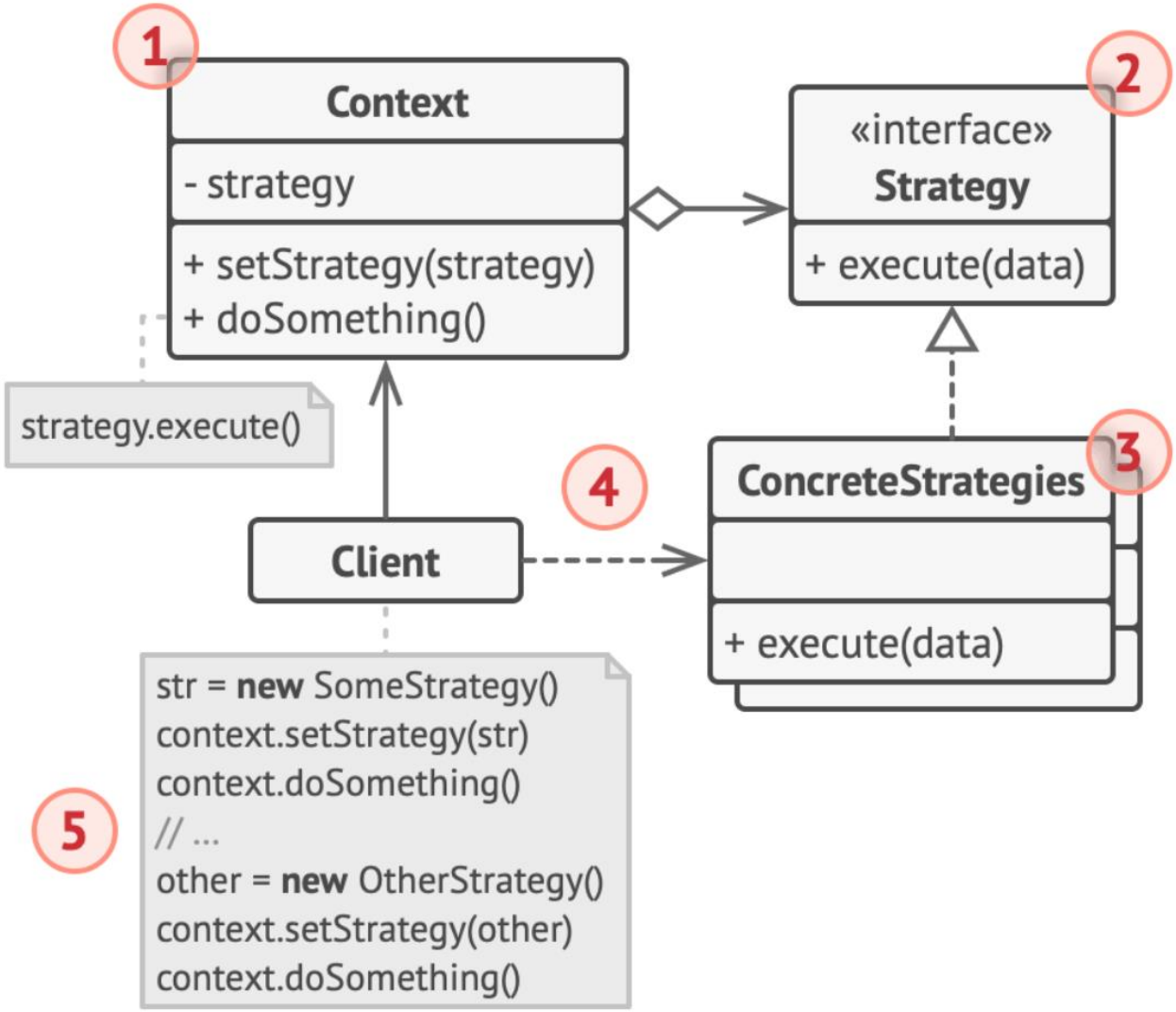
Another Example



Various strategies for getting to the airport.

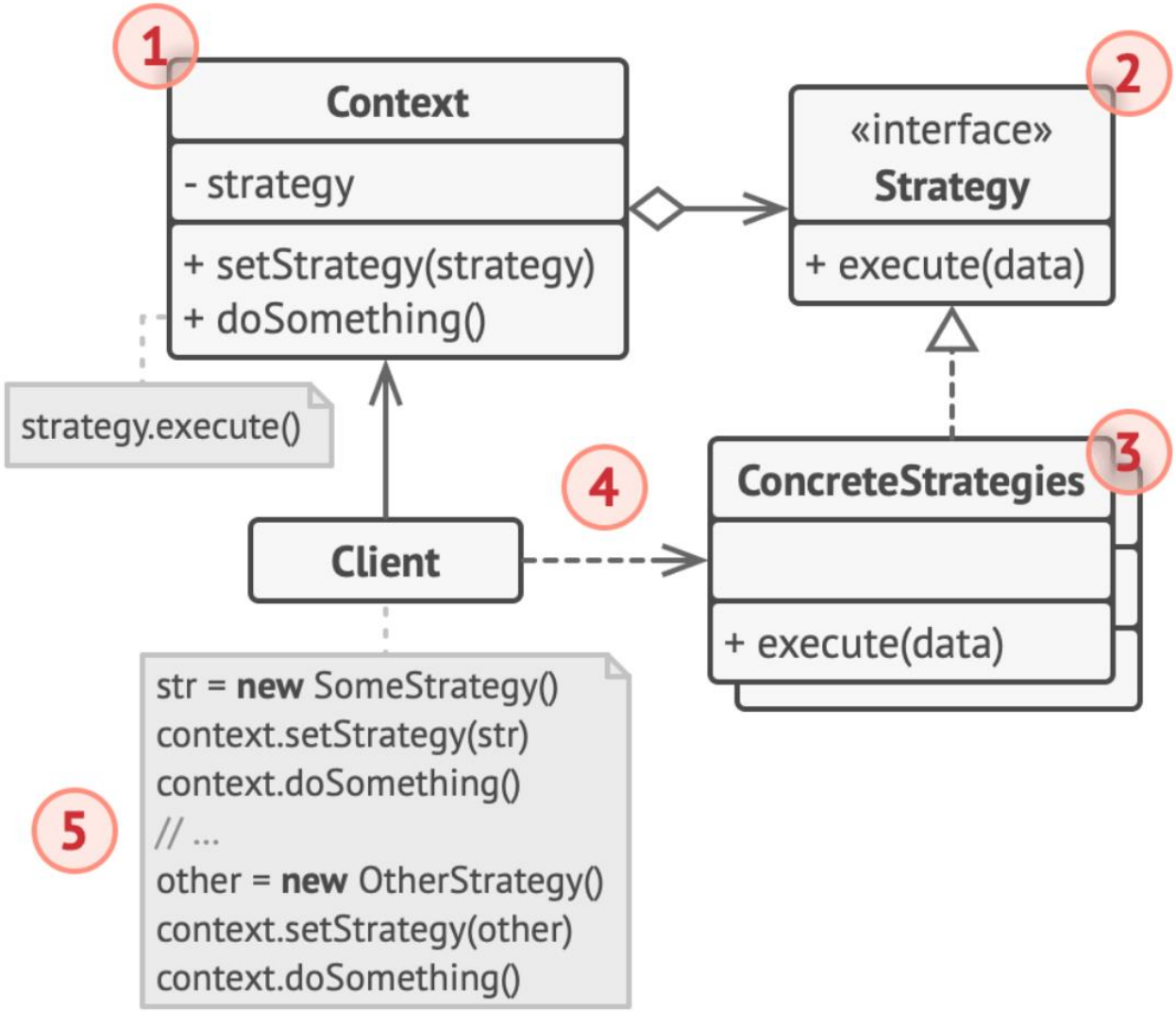
UML Class Diagram

1. The **Context** maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface.



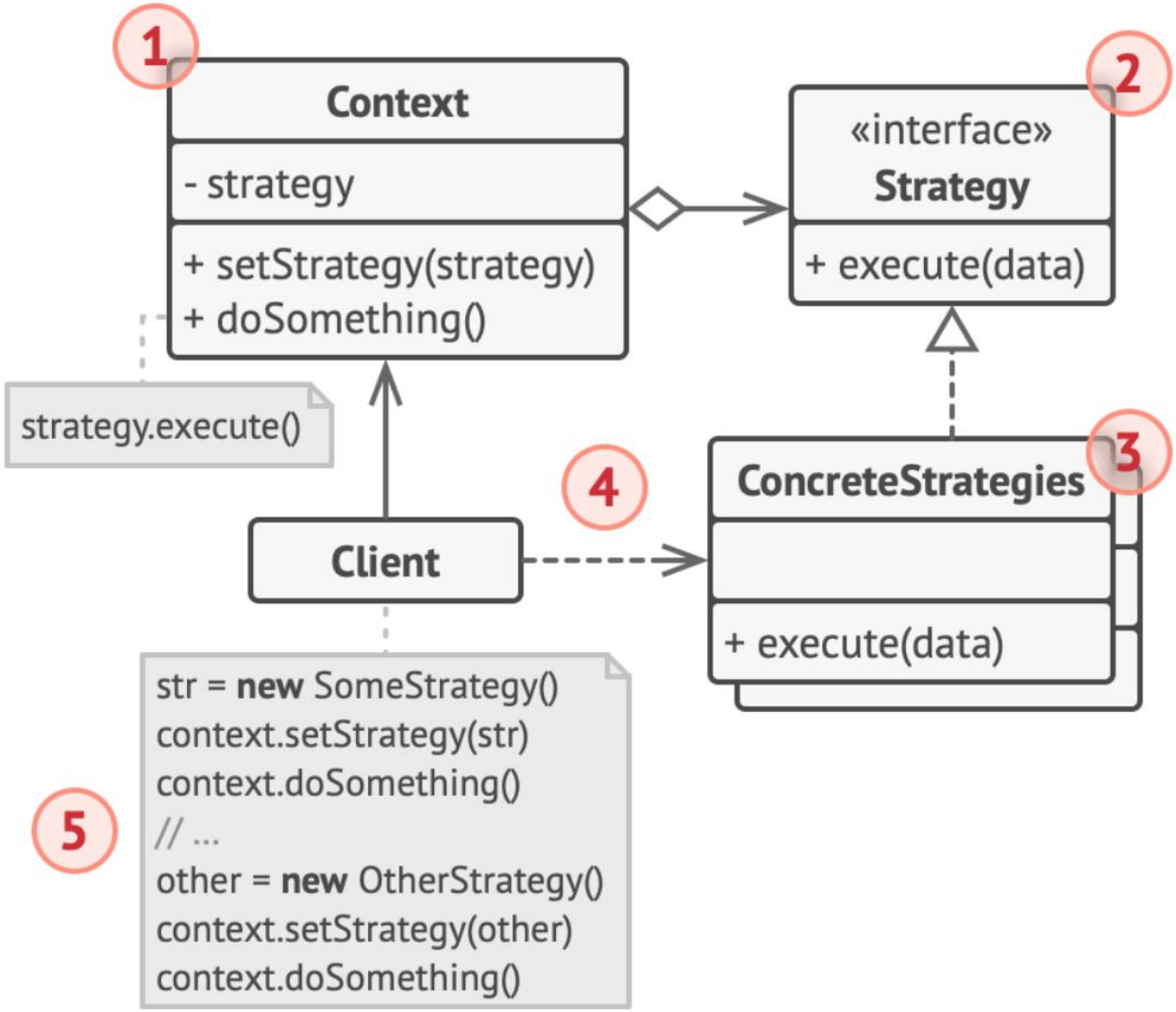
UML Class Diagram

2. The **Strategy** interface is common to all concrete strategies. It declares a method the context uses to execute a strategy.



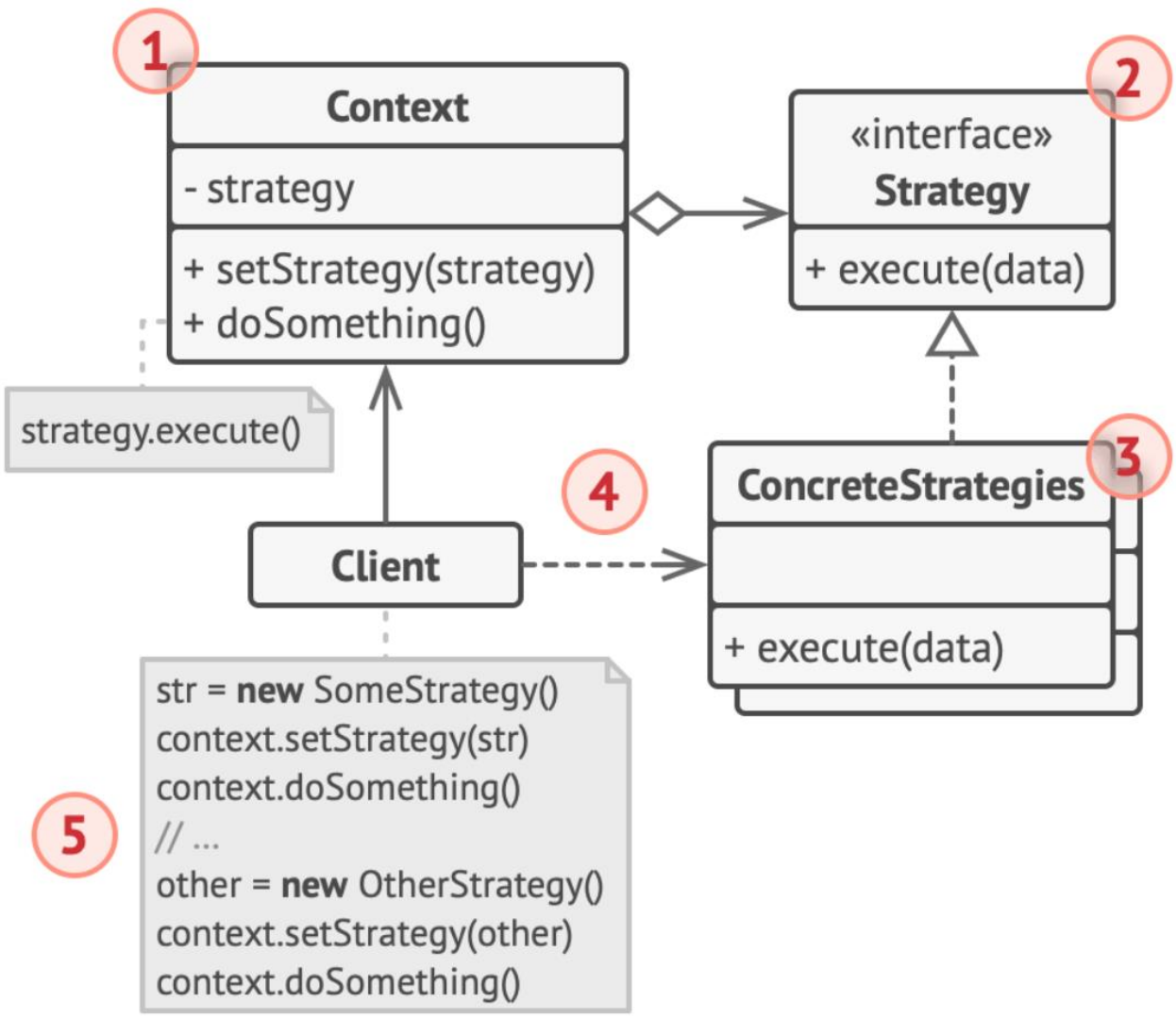
UML Class Diagram

3. **Concrete Strategies** implement different variations of an algorithm the context uses.



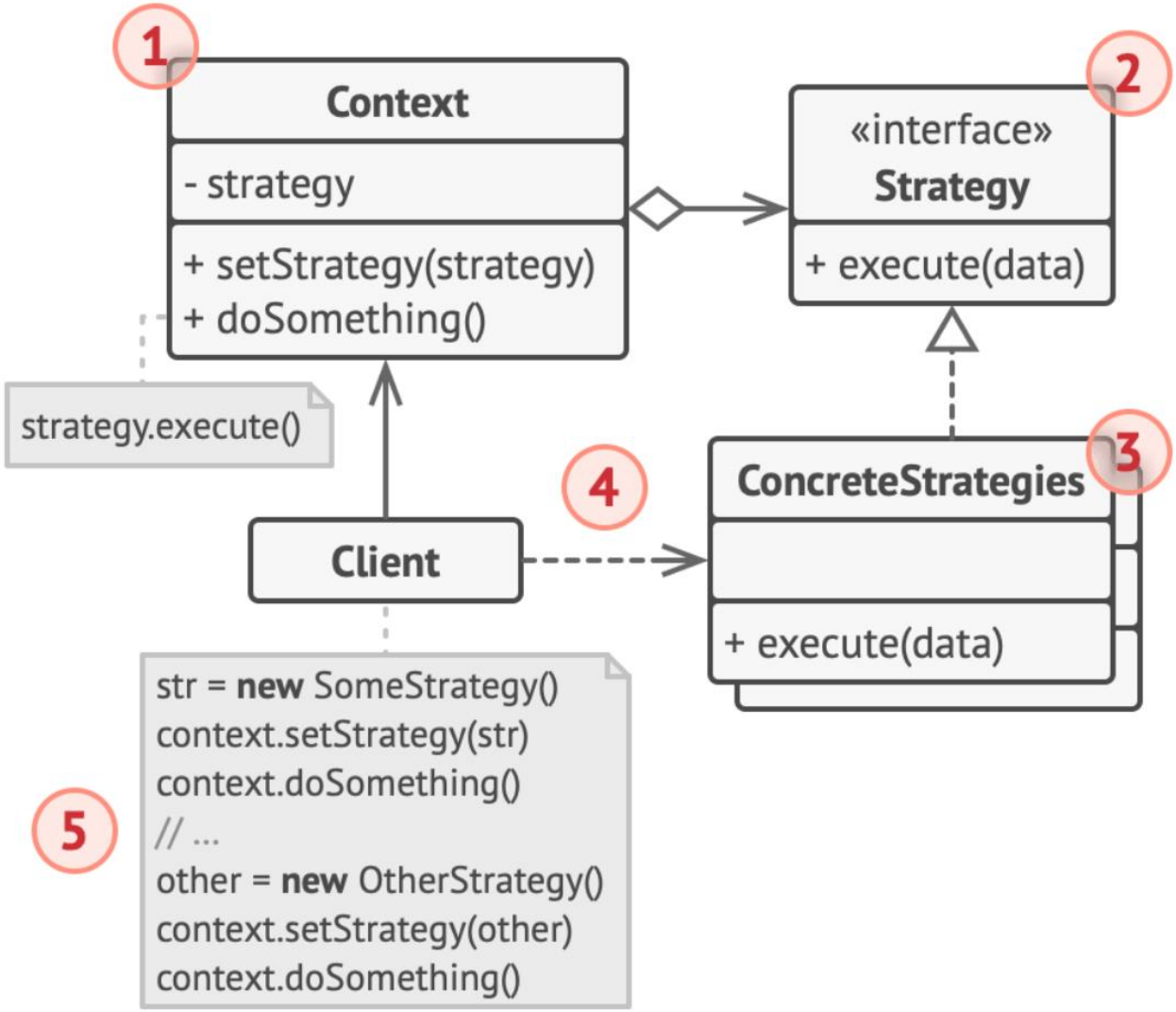
UML Class Diagram

4. The context calls the execution method on the linked strategy object each time it needs to run the algorithm. The context doesn't know what type of strategy it works with or how the algorithm is executed.



UML Class Diagram

5. The **Client** creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.



Applicability

When you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another

When you have a lot of similar classes that only differ in the way they execute some behavior

To isolate the business logic of a class from the implementation details of algorithms

when your class has a massive conditional statement for switching between variants of the same algorithm

The big pros

Swap algorithms used inside an object

Isolate implementation of an algorithm from the code that uses it

Replace inheritance with composition

O from SOLID

Duck code example

HW 3 Intro