GoF Design Patterns

for ODDS

Your problem…

Somebody already solved it!

Software always Change

Would it be dreamy if there were a way to build software so that when we need to change it, we could do so with the least possible impact on the existing code?

Design Patterns

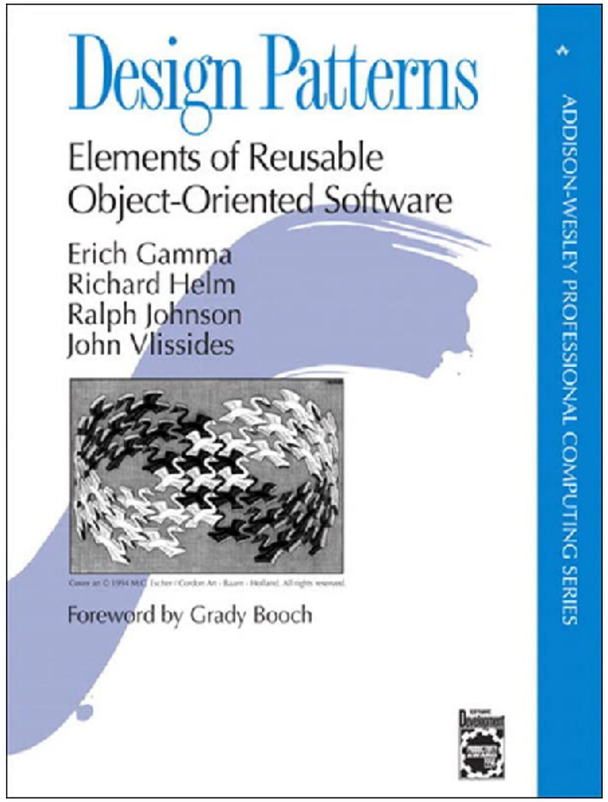
In software engineering

- Your problem, somebody (may) already solved it!
- A design pattern is a general repeatable solution to a commonly occurring problem in software design.
- A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

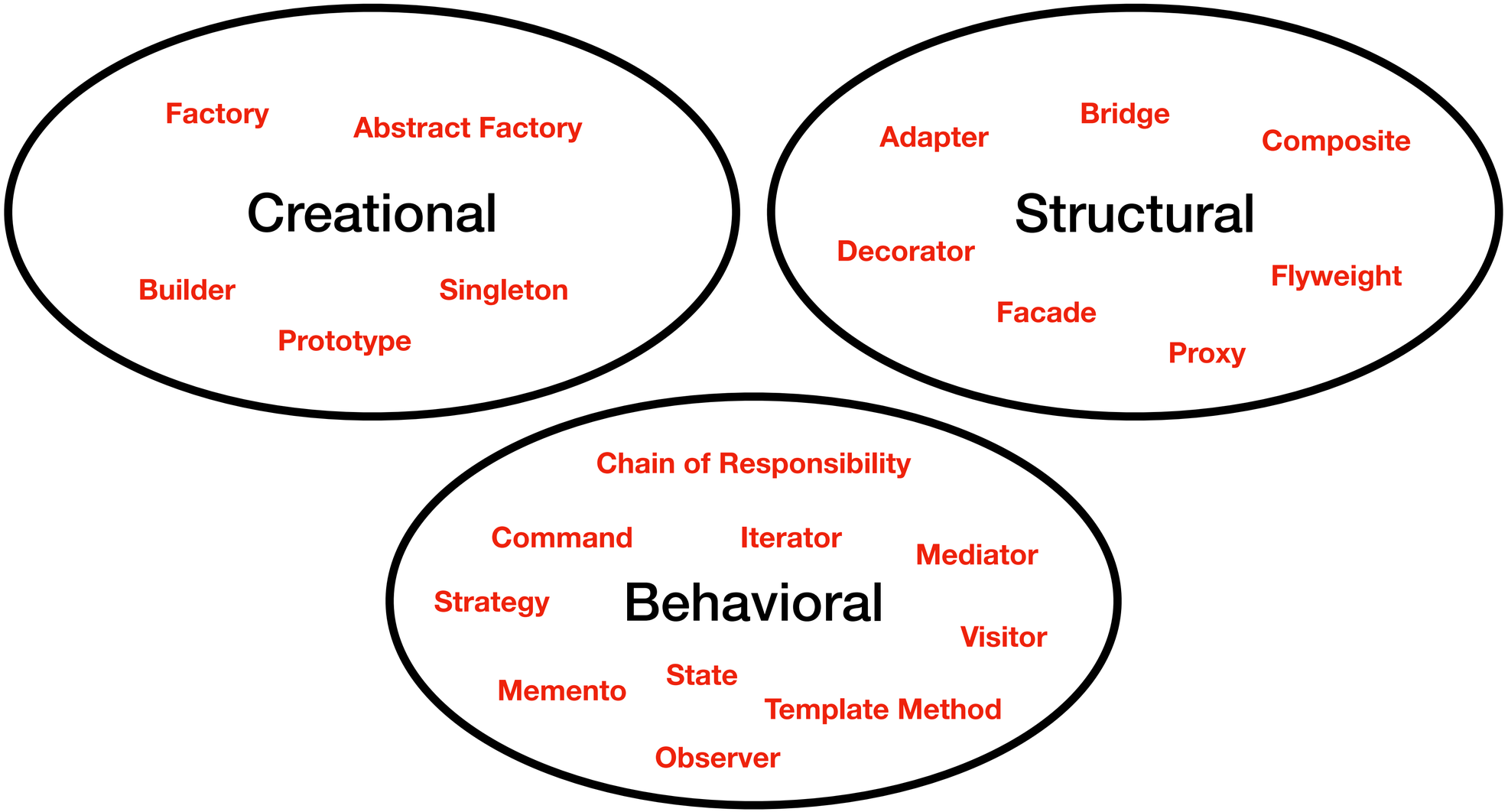
GoF Design Patterns

…stand for Gang of Four

- Almost 30 years ago (Oct 31, 1994) the iconic computer science book “Design Patterns: Elements of Reusable Object-Oriented Software” was first published.
- The four authors of the book: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, have since been dubbed “The Gang of Four”, or GoF.



Group of GoF Design Patterns



Use of Design Patterns

the benefits …?

- Design patterns can speed up the development process by providing tested, proven development paradigms.
- Effective software design requires considering issues that may not become visible until later in the implementation.
- Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.
- BTW. Adding patterns into software structure **increase complexity**. Use it as needed.

Terminology

Back to the basic of OO & UML

Object-Oriented Programming

let's recap

- Abstraction

  class, object, instance, attribute, method/operation, behaviour

- Encapsulation - information hiding, information protection

  private, protected, public, package

- Inheritance

  super class, sub class, abstract, interface

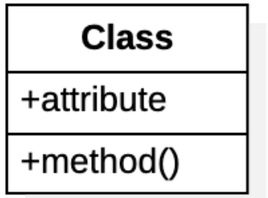
- Polymorphism

  override, overload

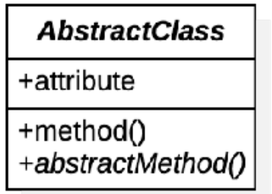
Some UML Class Diagram notations

good to know, but not require

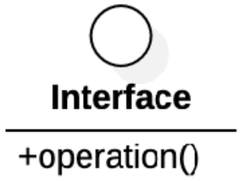
- Class

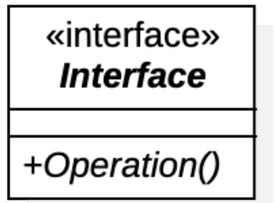


- Abstract class

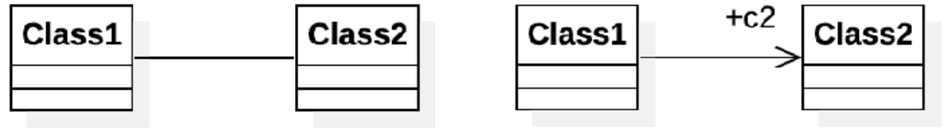


- Interface class

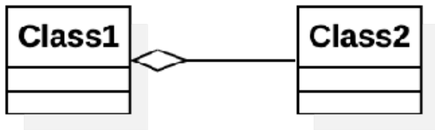




- Association

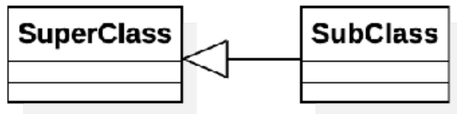


- Aggregation (share)



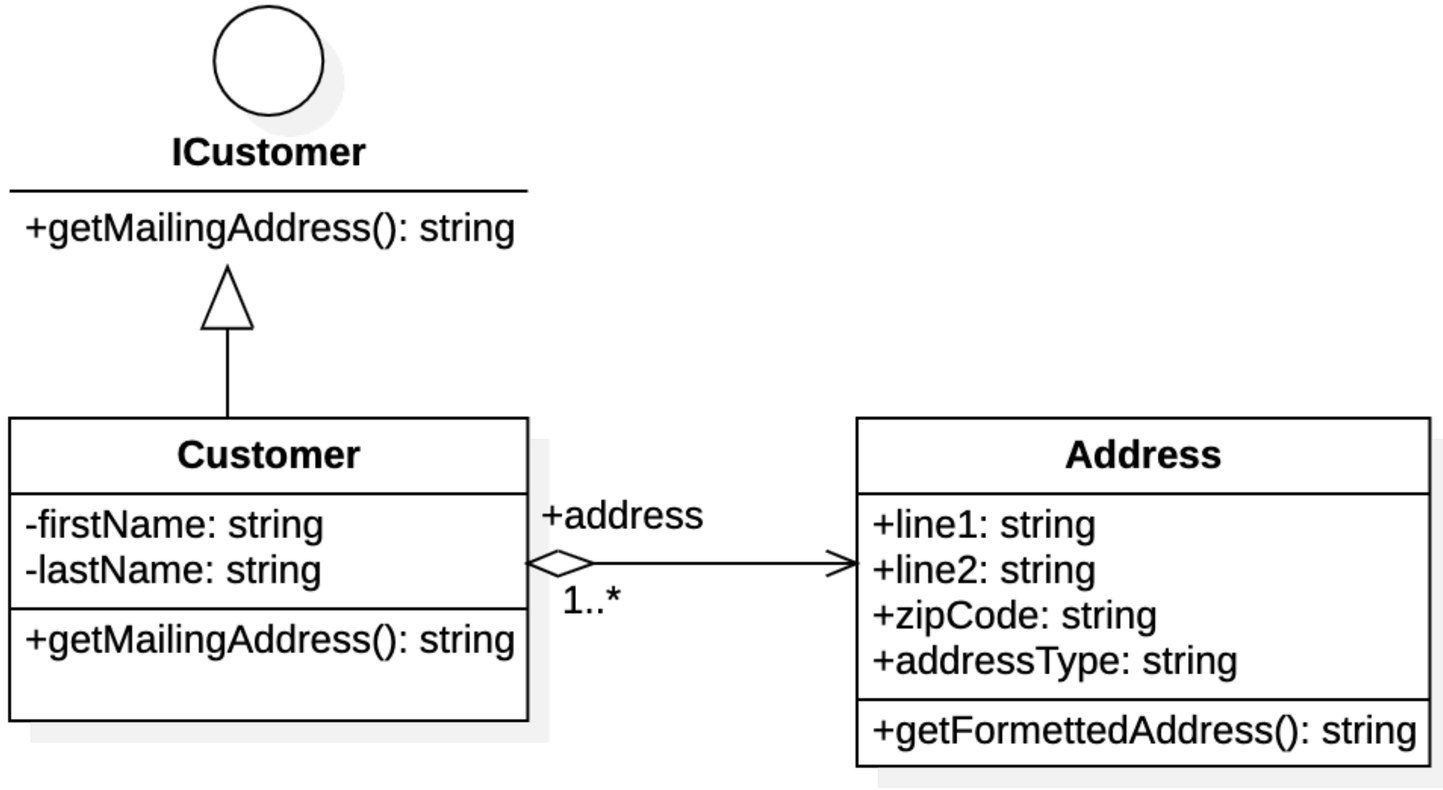
- Composition (own/component)

- Is-a relationship / Generalization (inherit, implement)



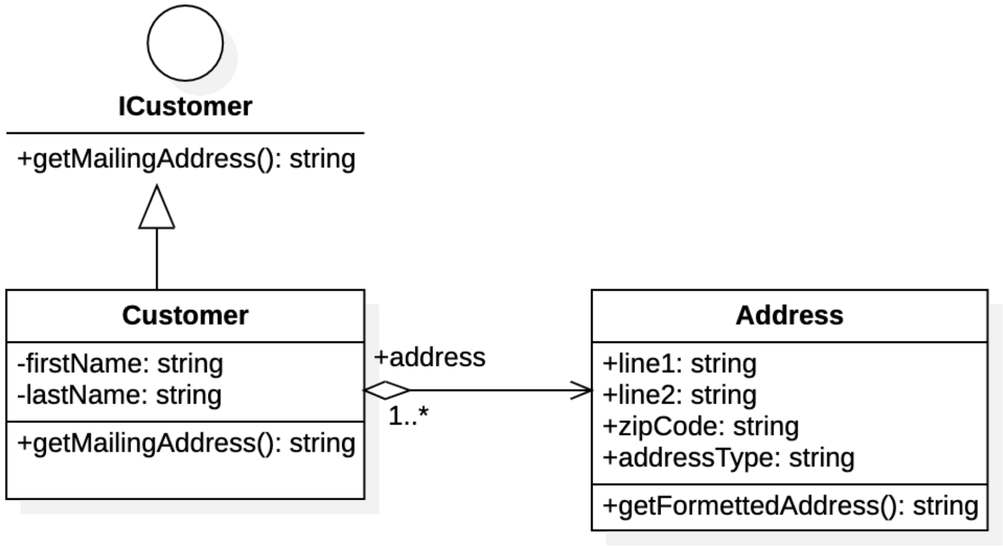
Class diagram exercise

one picture describe thousand words.



Class diagram

from diagram to code (and vice versa)




```
interface ICustomer {
    getMailingAddress(): string;
}

class Customer implements ICustomer {
    private firstName: string;
    private lastName: string;
    private addresses: Address[];

    constructor(firstName: string, lastName: string, address: Address[]) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.addresses = address;
    }
    getMailingAddress(): string {
        return this.firstName + " " + this.lastName + "\n" + this.addresses.map(address => address.getFormettedAddress()).join("\n");
    }
}

class Address {
    line1: string;
    line2: string;
    zipCode: string;
    addressType: string;

    getFormettedAddress(): string {
        return this.line1 + "\n" + this.line2 + "\n" + this.zipCode;
    }
}
```

Behavioral

Strategy Pattern

Setting behaviour dynamically

Strategy Pattern

Behavioral Patterns

Problem statement

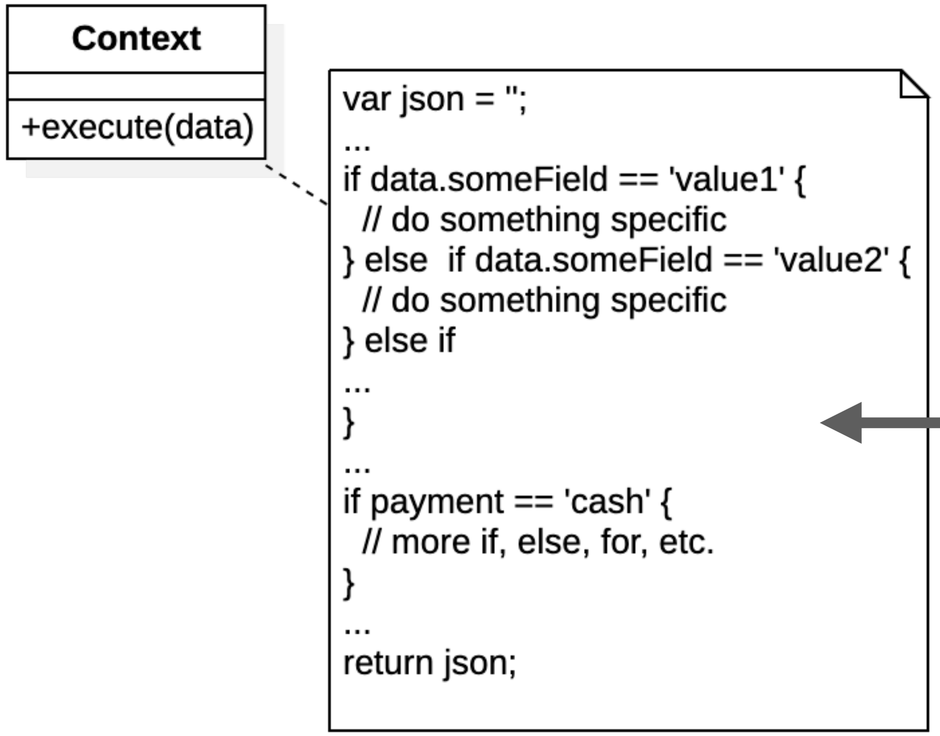
- Any change to one of the algorithms, whether it was a simple bug fix or a slight adjustment of the street score, affected the whole class, increasing the chance of creating an error in already-working code.

Solution

- The Strategy pattern suggests that
  - … you take a class that does something specific in a lot of different ways
  - … and extract all of these algorithms into separate classes. Called strategies.
- Strategy Pattern is a pattern for encapsulates interchangeable behaviors and uses delegation to decide which one to use.

Problem

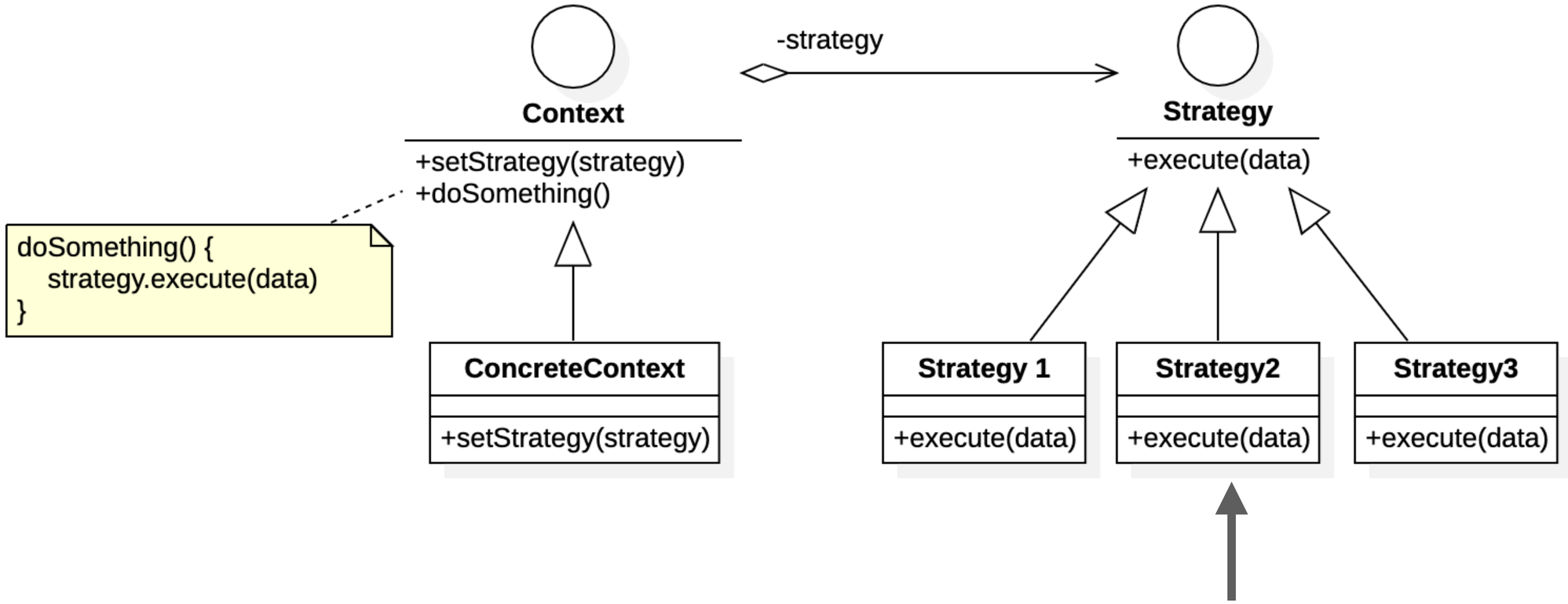
Class diagram



Any change to one of the algorithms, whether it was a simple bug fix or a slight adjustment of the street score, affected the whole class, increasing the chance of creating an error in already-working code.

Strategy Pattern

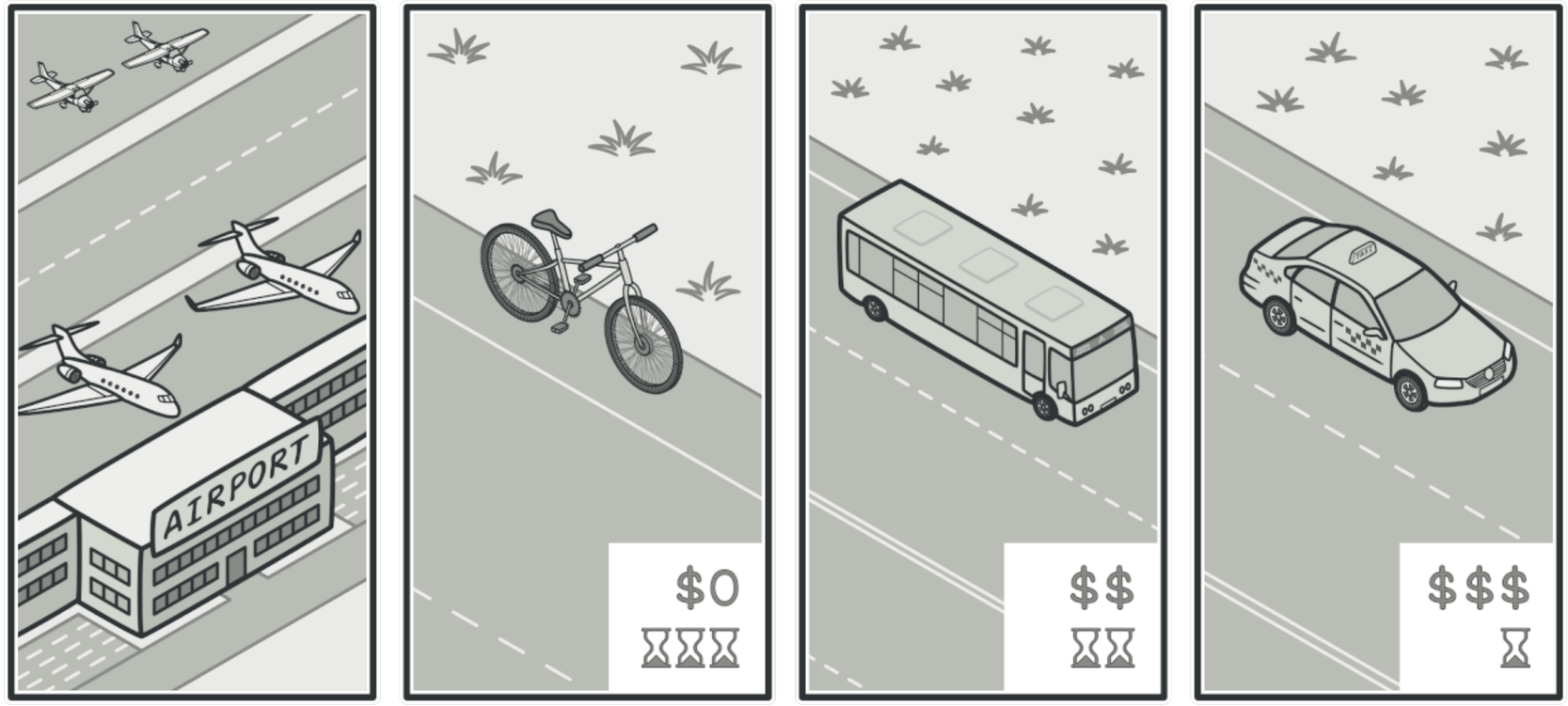
Class diagram



take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called *strategies*.

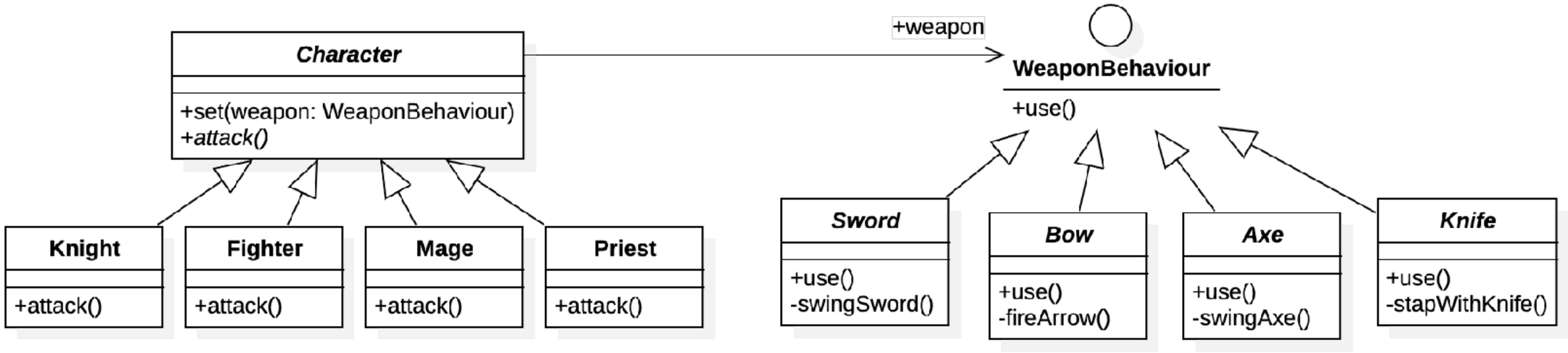
Strategy Pattern in Real Life

Travel



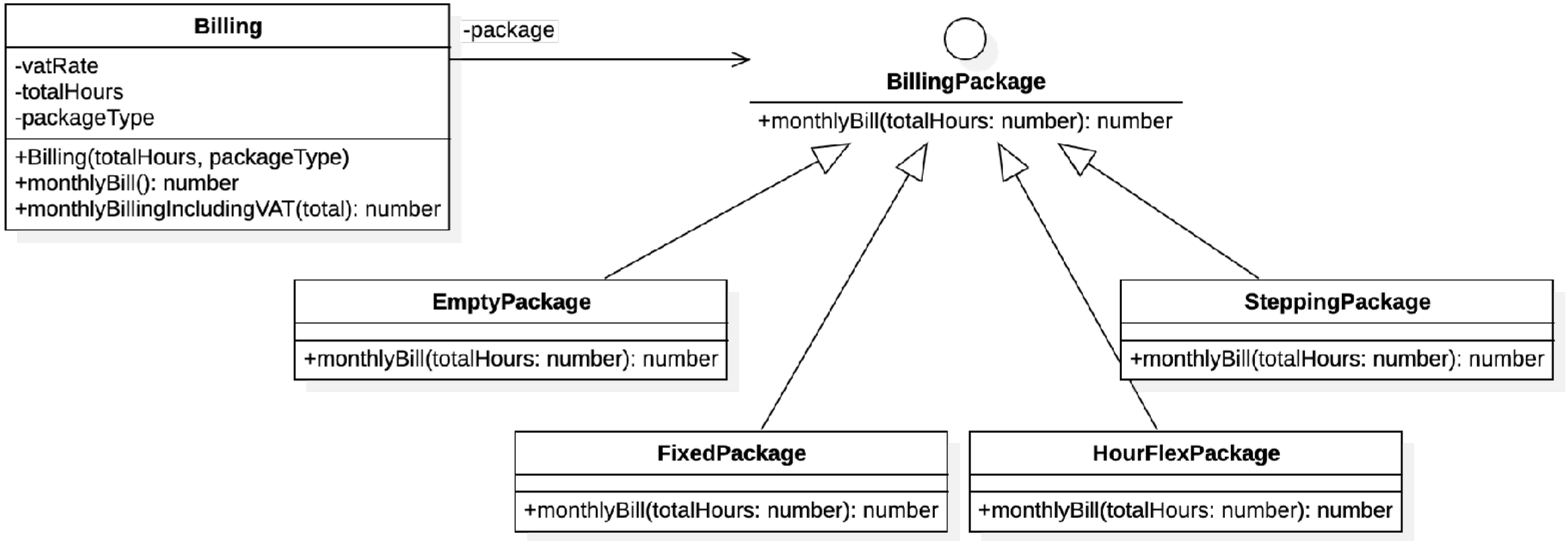
Wrapped Behaviour

example in Game



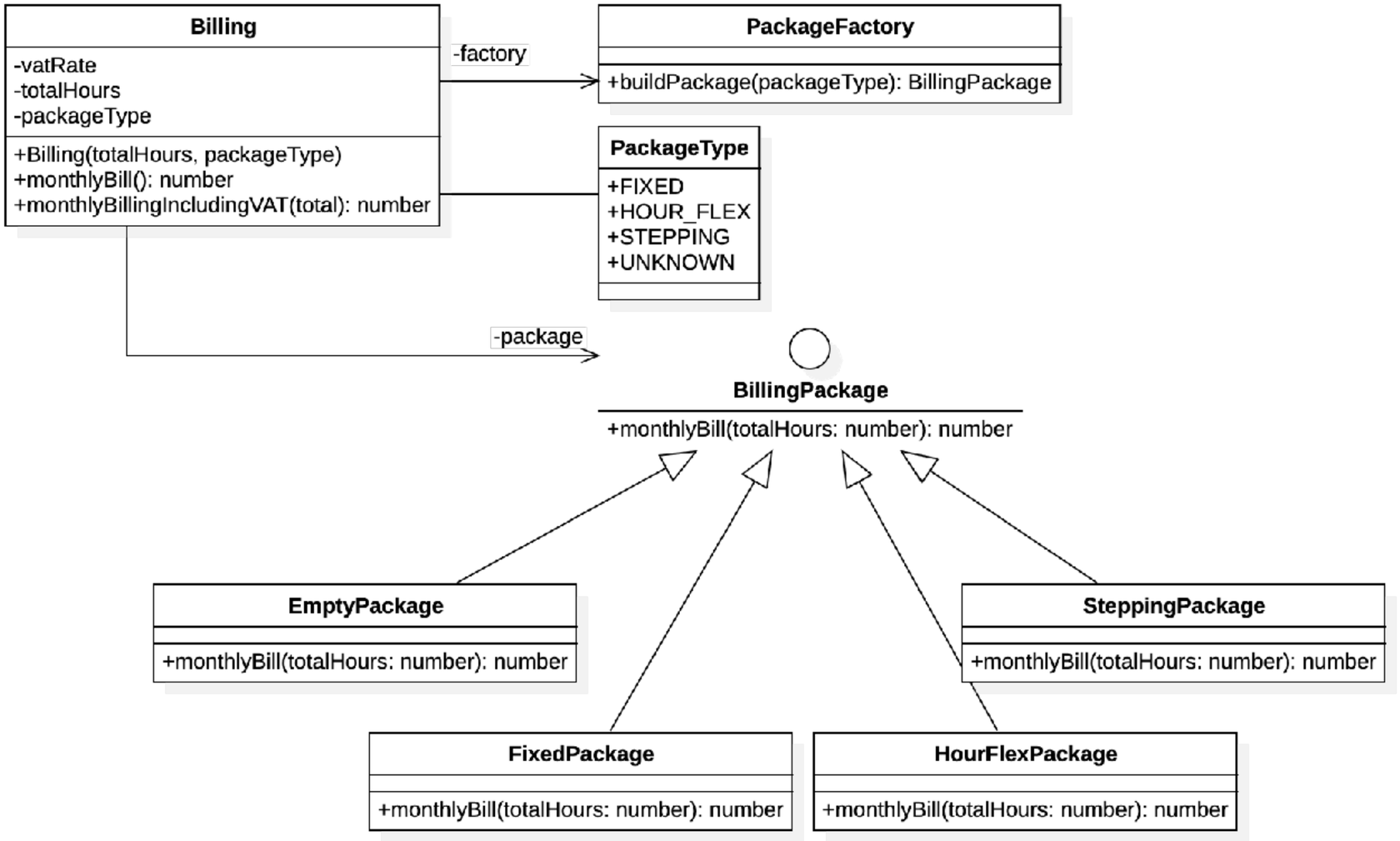
Lab

billing system



Lab

Strategy + Factory



Behavioral

Observer Pattern

Do you know ReactiveX observable?

Observer Pattern

Behavioral Patterns

Problem statement

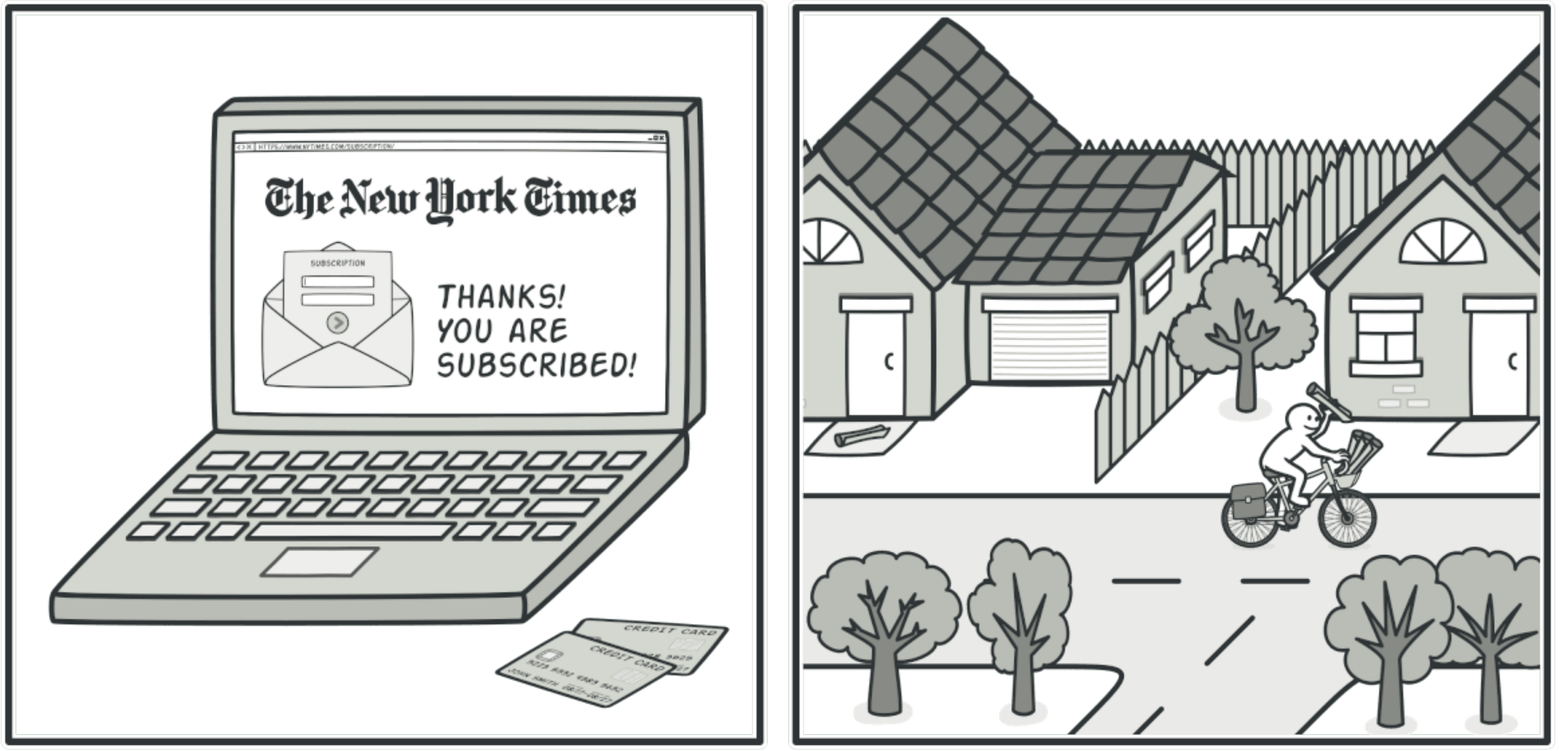
- Imagine that you have two types of objects: a **Customer** and a **Store**. The customer is very interested in a particular brand of product (say, it's a new model of the iPhone) which should become available in the store very soon.
- The customer could visit the store every day and check product availability. But while the product is still en route, most of these trips would be pointless.

Solution

- Allows a group of objects to be notified when some state changes.
- Publisher + Subscriber = Observer Pattern

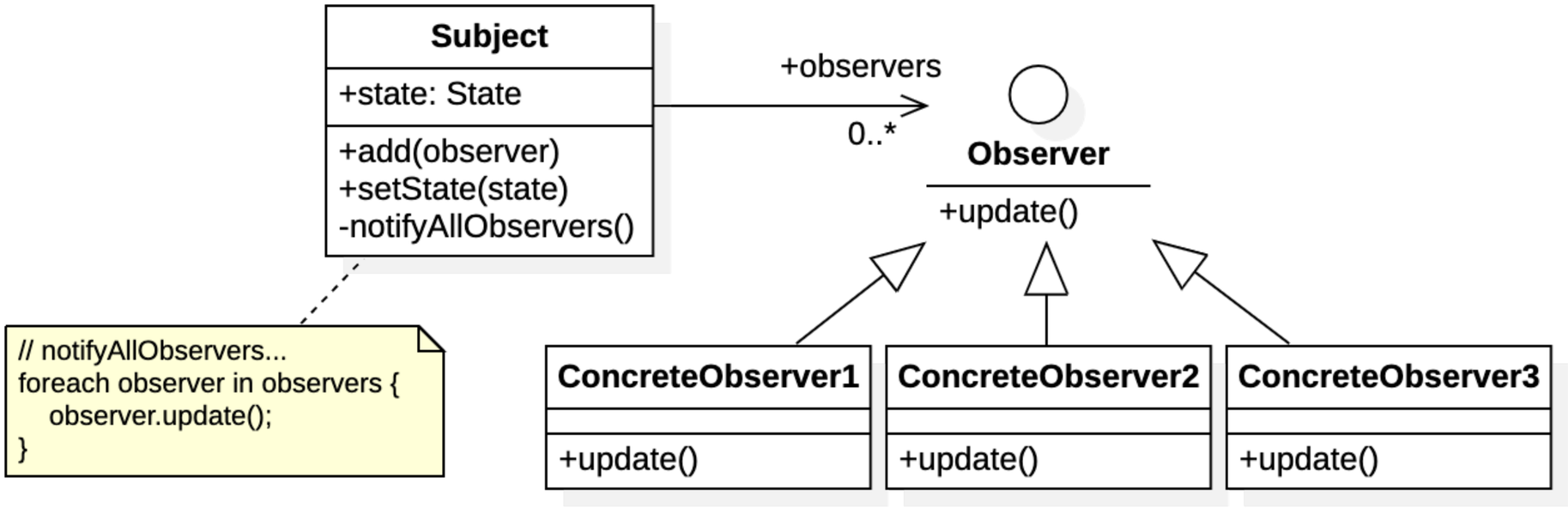
Observer Pattern in Real Life

Mailing list, Newspaper subscription

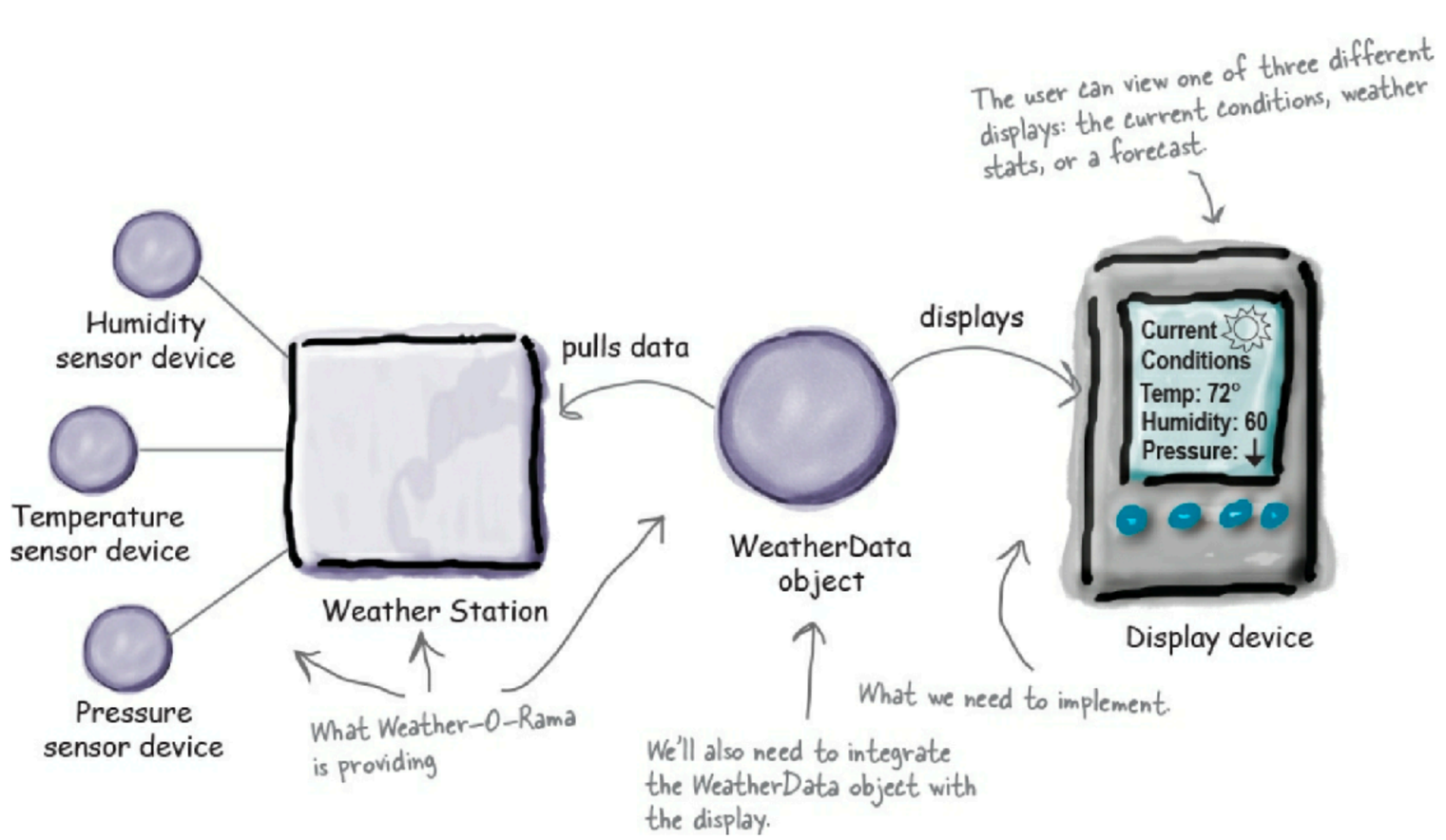


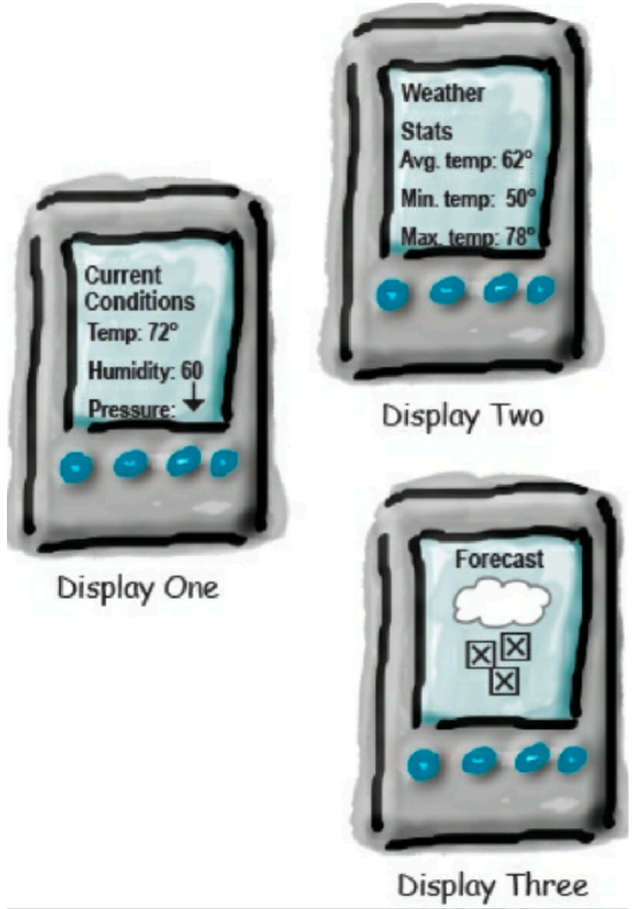
Observer Pattern

Class Diagram



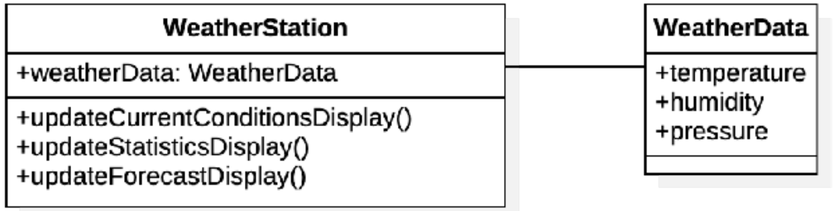
Lab


The user can view one of three different displays: the current conditions, weather stats, or a forecast.
Humidity sensor device
Temperature sensor device
Pressure sensor device
Weather Station
pulls data
WeatherData object
displays
Current Conditions
Temp: 72°
Humidity: 60
Pressure:
Display device
What Weather-O-Rama is providing
What we need to implement.
We'll also need to integrate the WeatherData object with the display.

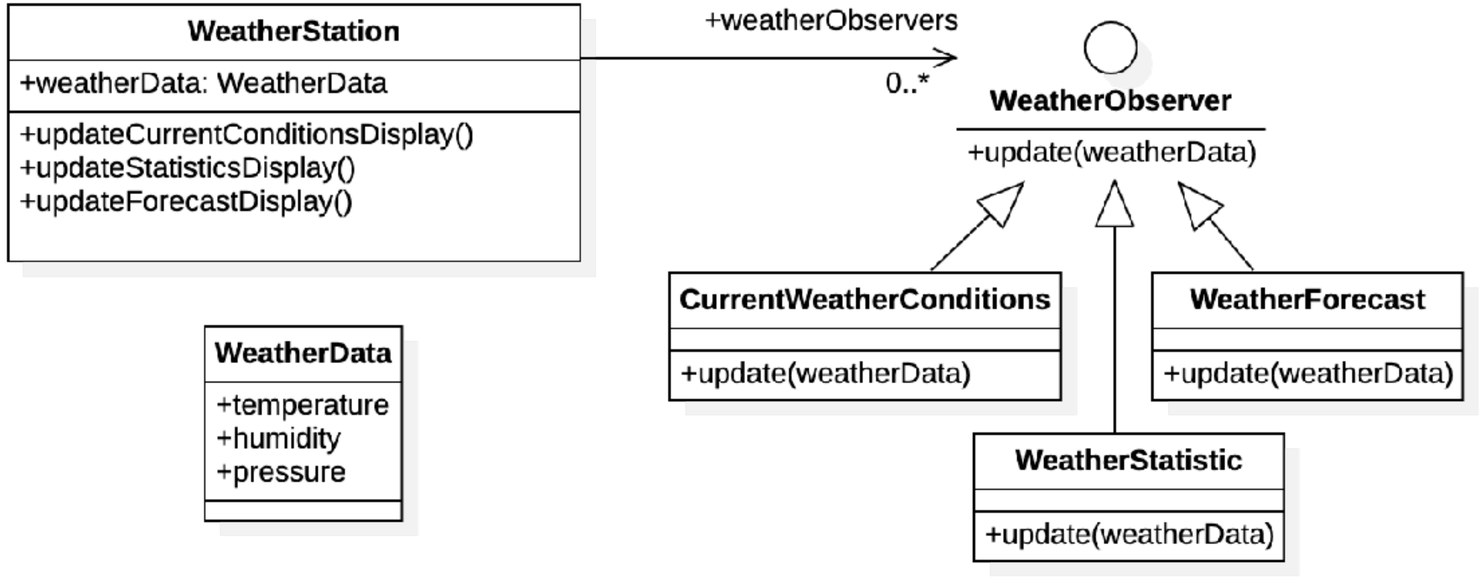

Current Conditions
Temp: 72°
Humidity: 60
Pressure:
Display One
Weather Stats
Avg. temp: 62°
Min. temp: 50°
Max. temp: 78°
Display Two
Forecast
Display Three

Lab

current version



final version



Creational

Builder Pattern

Dynamically re-order building methods.

Builder Pattern

Creational Patterns

Problem statement

- Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. Or even worse: scattered all over the client code.

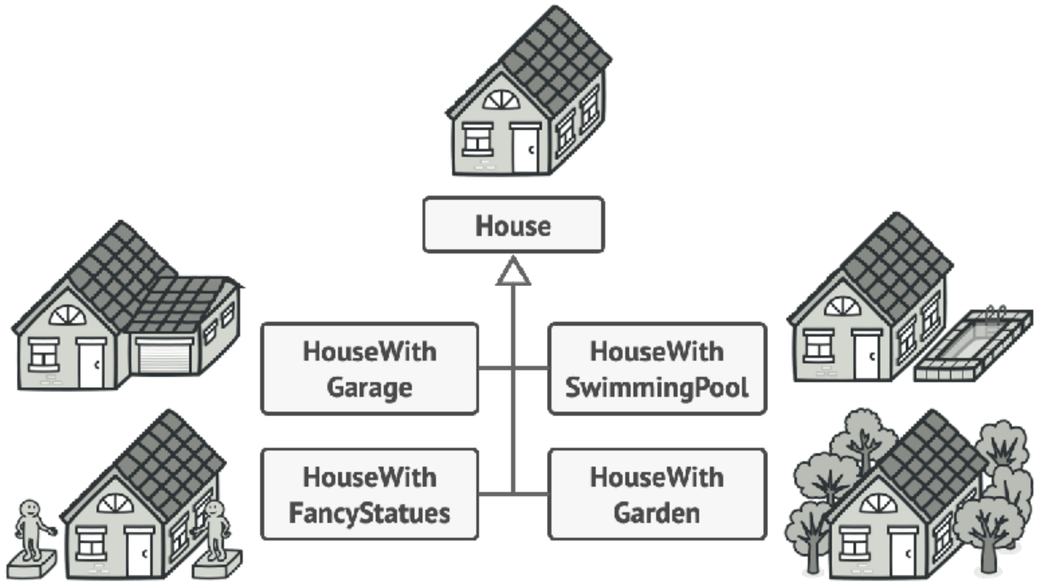
Solution

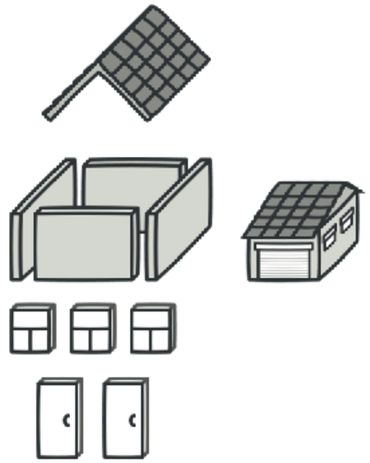
-

https://www.borntodev.com/2020/04/05/design-patterns-แบบพื้นฐานสุด-ๆ/

Builder Pattern in Real Life

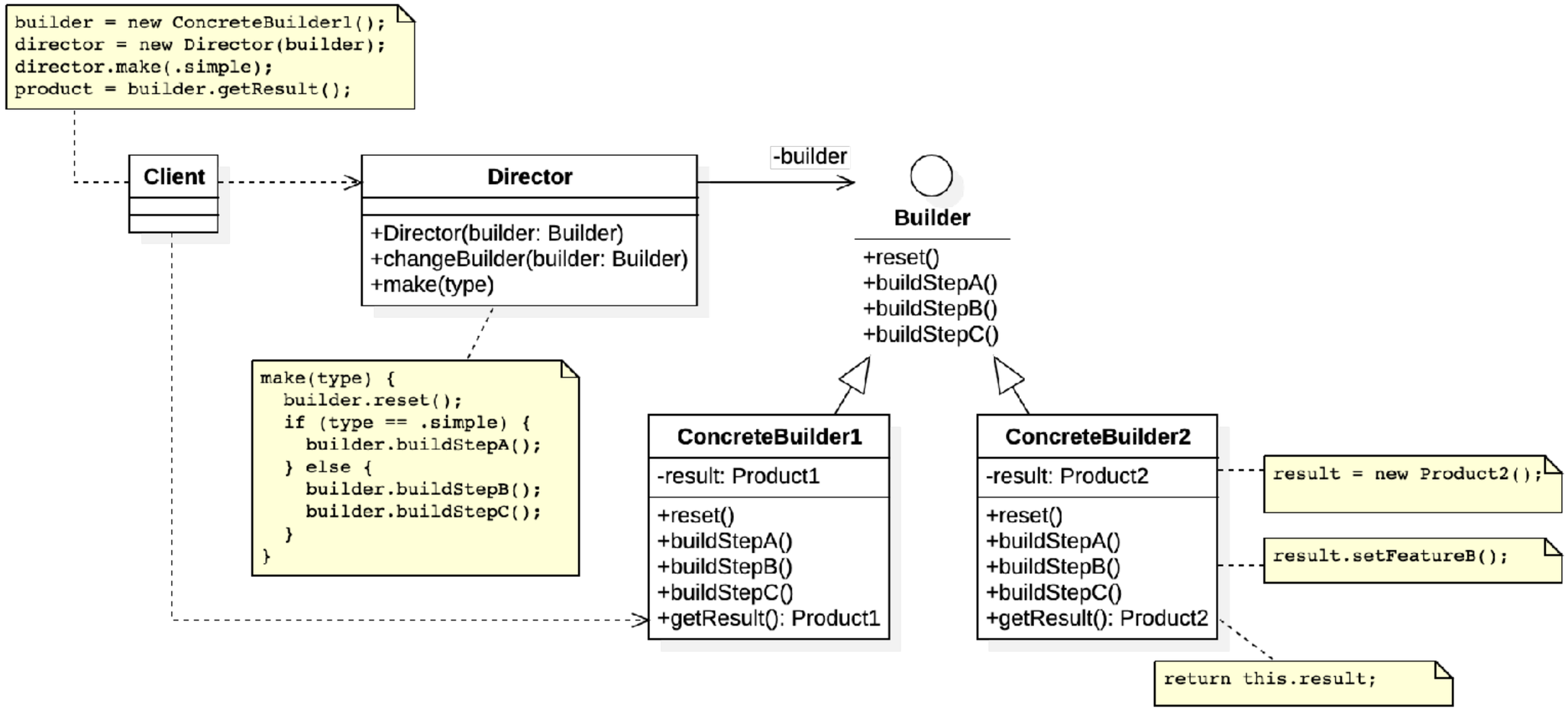
Build the house





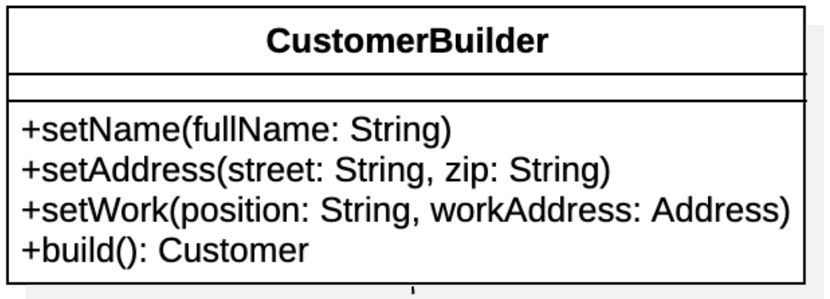
Builder Pattern

UML Diagram

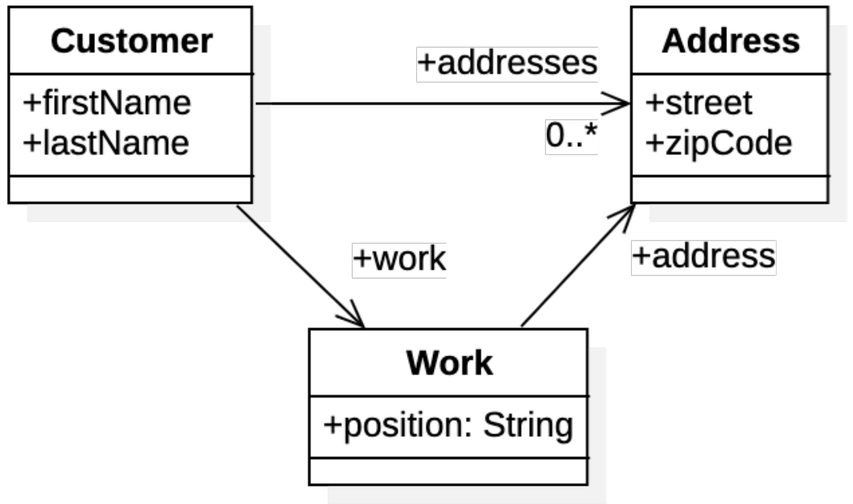


Builder Pattern

Example




```
const customer = CustomerBuilder()
    .setName('John Doe')
    .setAddress(new Address())
    .setWork('Dev', new Address())
    .build();
```



Structural

Decorator Pattern

Extension behavior at runtime, rather than compile time.

Decorator Pattern

Structural Patterns

Problem statement

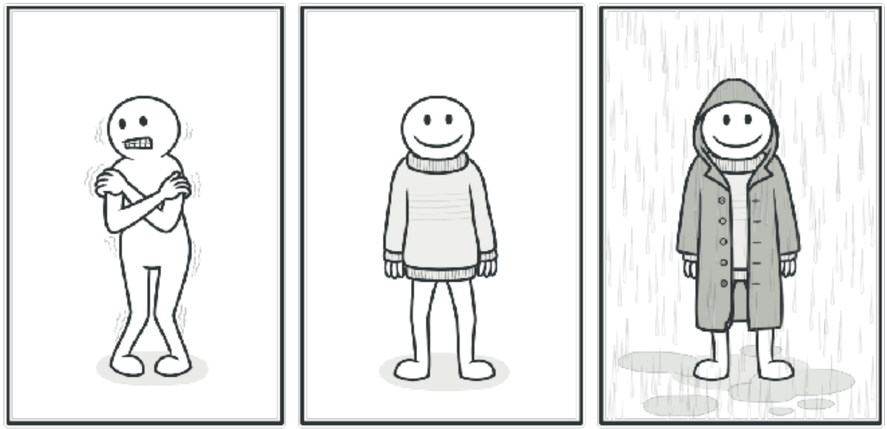
- We need to develop software that allows dynamic extension of a class's behavior at runtime, enabling one class to build upon another.

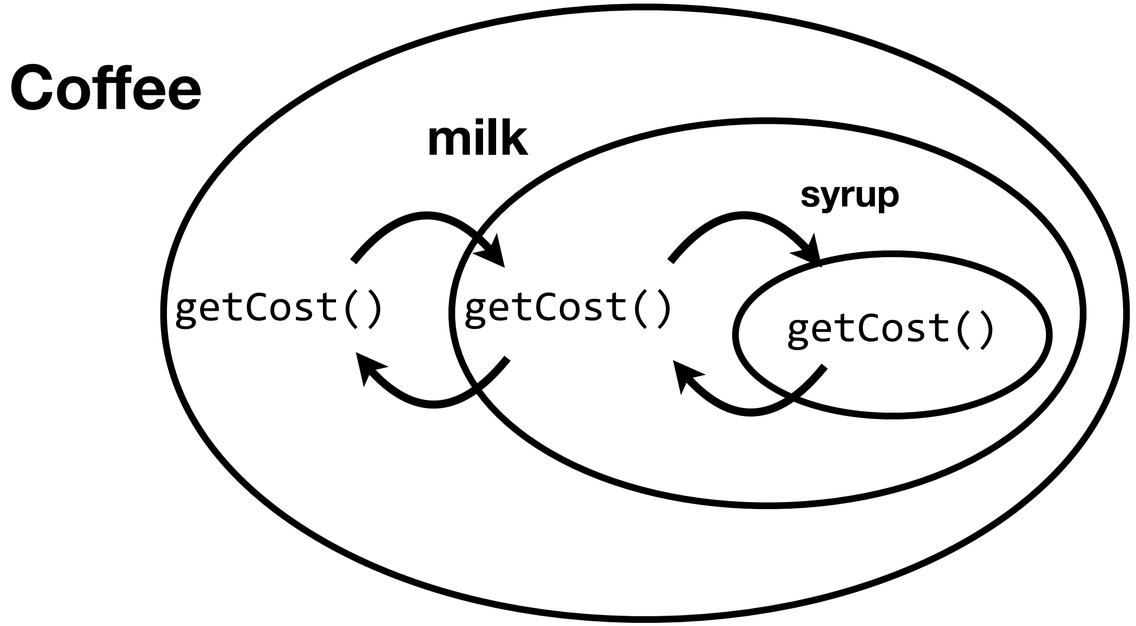
Solution

- Decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator Pattern in Real Life

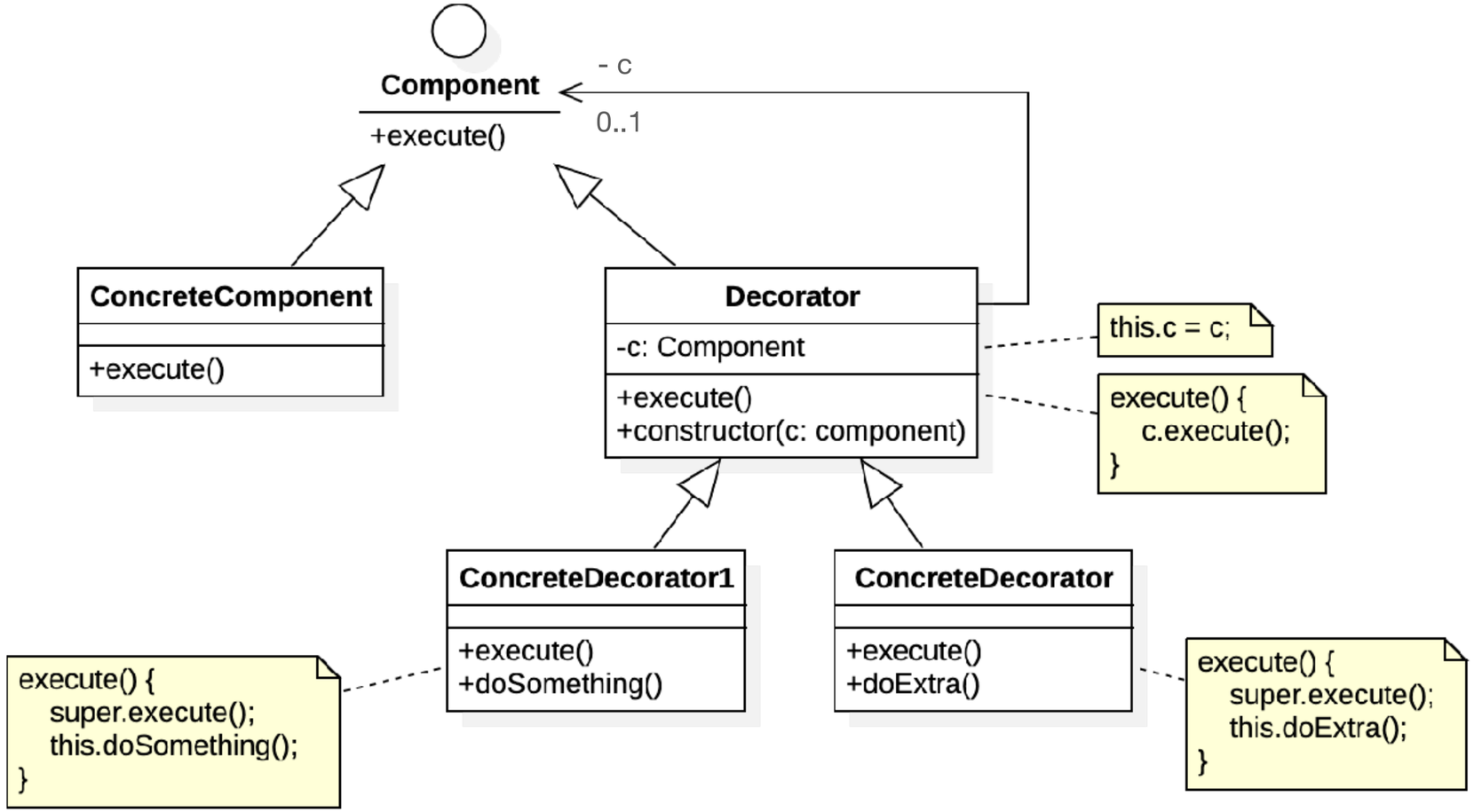
The visualization





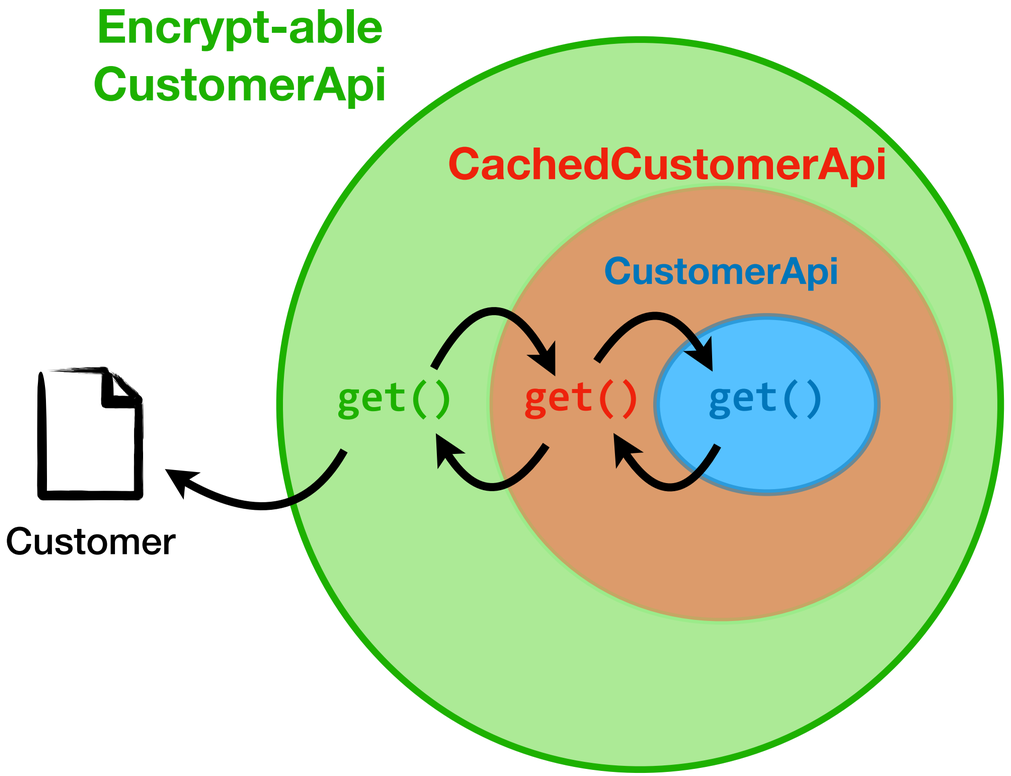
Decorator Pattern

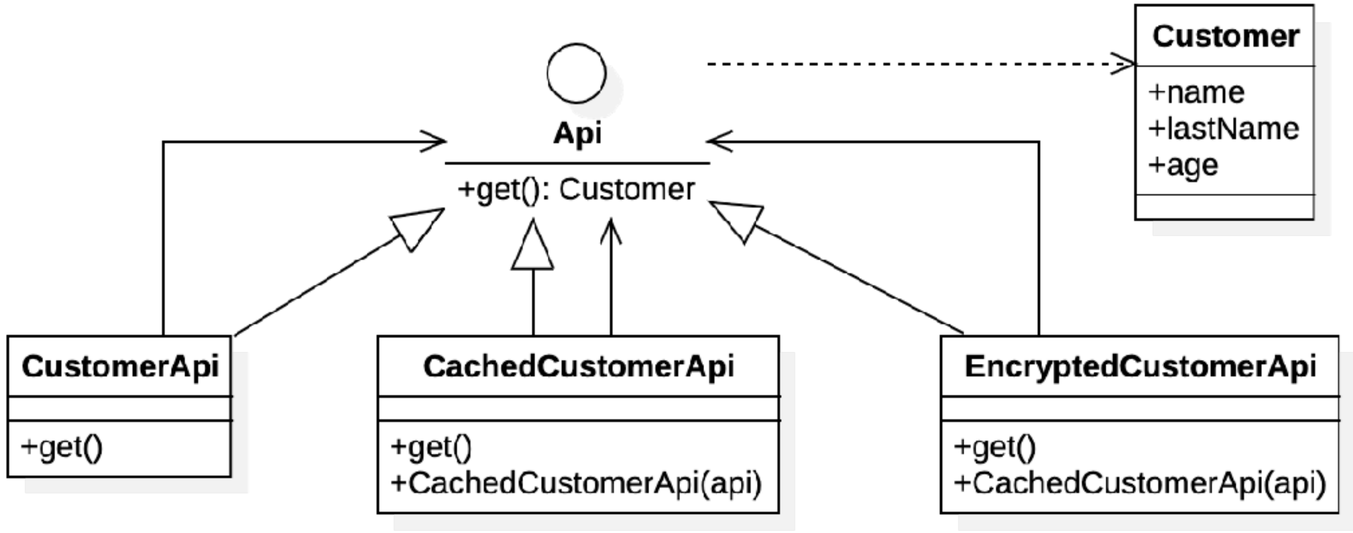
UML Diagram



Lab

JsonBuilder + Wrapped + Encrypted





Creational

Factory Pattern

Baking with OO Goodness

Factory Pattern

Creating objects as Template Method is to implementing an algorithm.

Problem statement

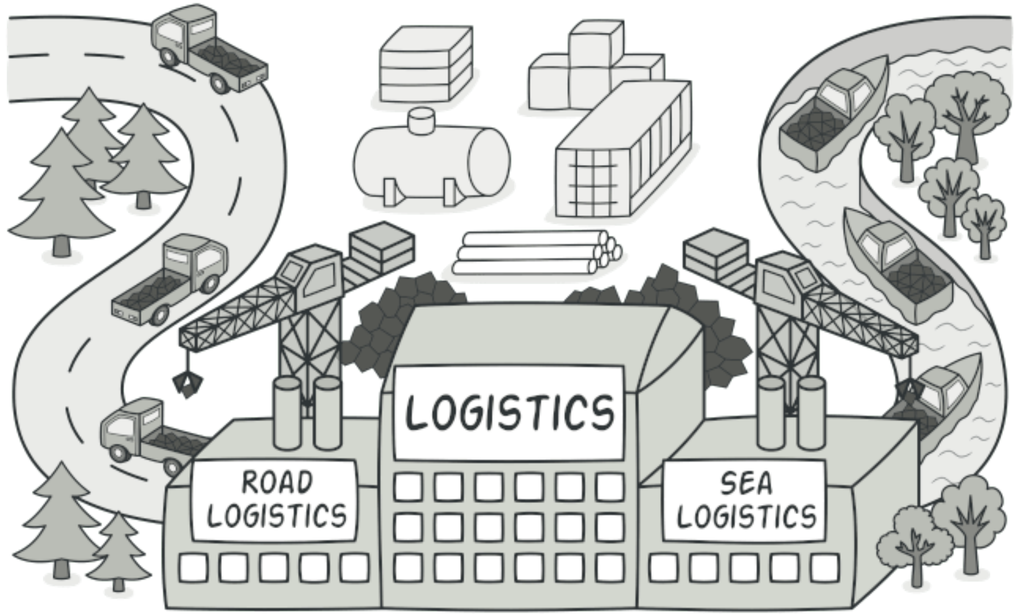
- A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

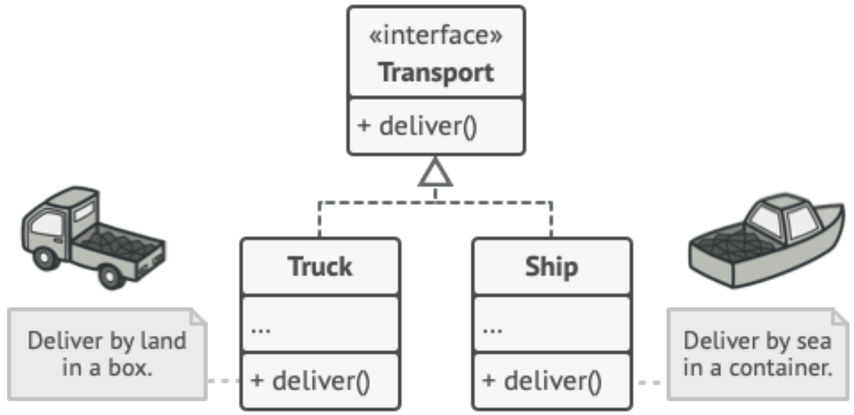
Solution

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Defining a "virtual" constructor.
- The new operator considered harmful.

Factory Pattern in real life

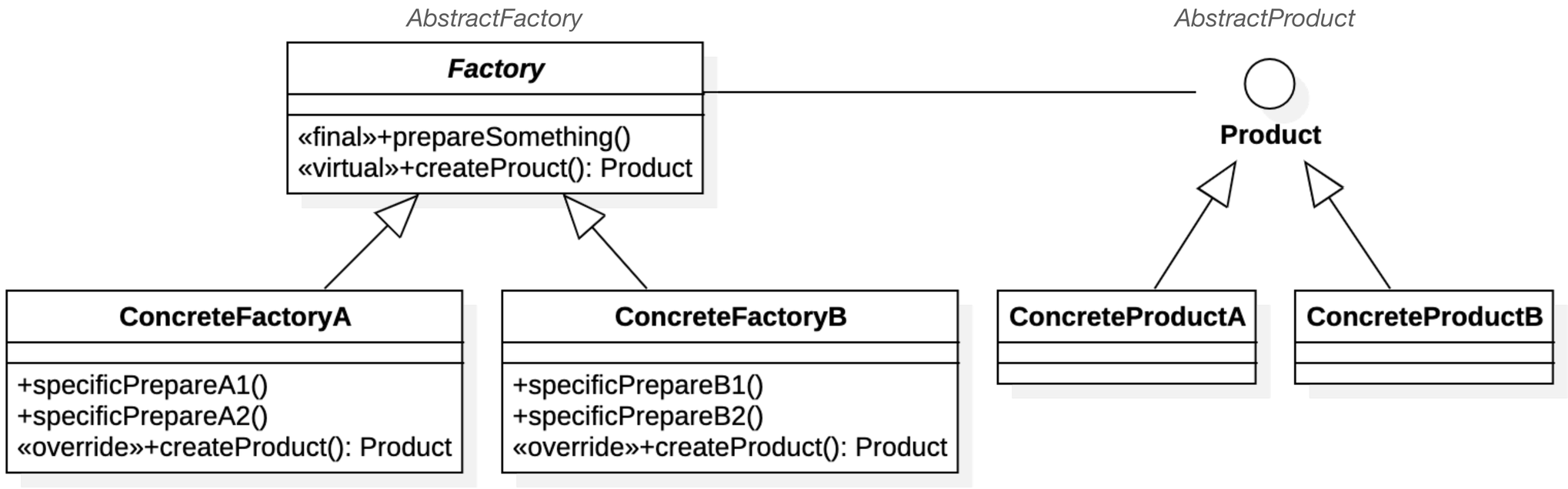
The visualization





Factory Pattern

Class Diagram



Creational

Singleton Pattern

One of a kind

Singleton Pattern

Creating objects as Template Method is to implementing an algorithm.

Problem statement

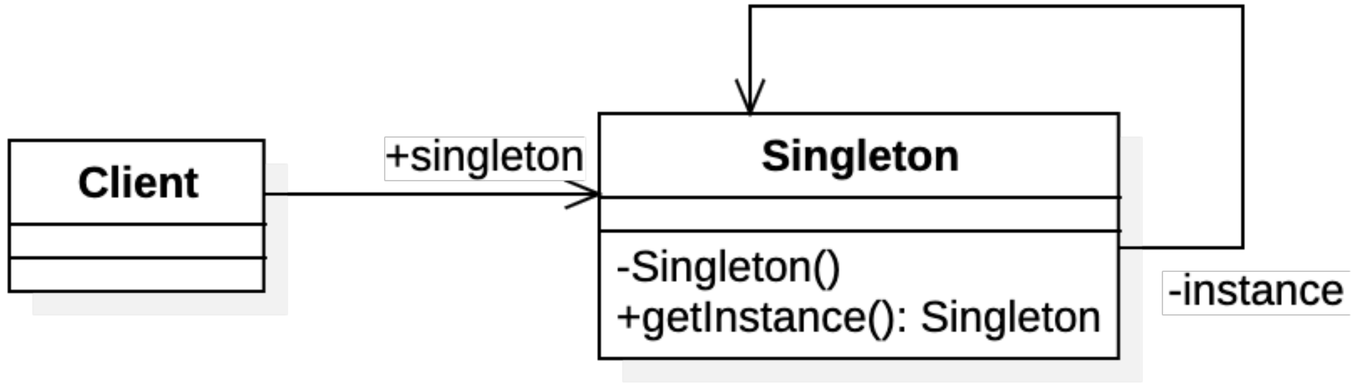
- We prefer an object's method to manage shared resources rather than a static class method. This object should be **unique and singular**, one and only one object.
- Provide a global access point to that instance.

Solution

- Make the default constructor **private**.
- Create a static creation method that acts as a constructor.

Singleton Pattern

Class Diagram



Singleton Pattern

The dark-side

- Violates the *Single Responsibility Principle*. The pattern solves two problems at the time.
- The Singleton pattern *can mask bad design*, for instance, when the components of the program know too much about each other.
- The pattern requires special treatment in a multi-threaded environment so that multiple threads won’t create a singleton object several times.
- It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance *when producing mock objects*. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don’t write the tests. Or don’t use the Singleton pattern.
- So, *don’t use Singleton* until you know what you’re doing and you know to test it.

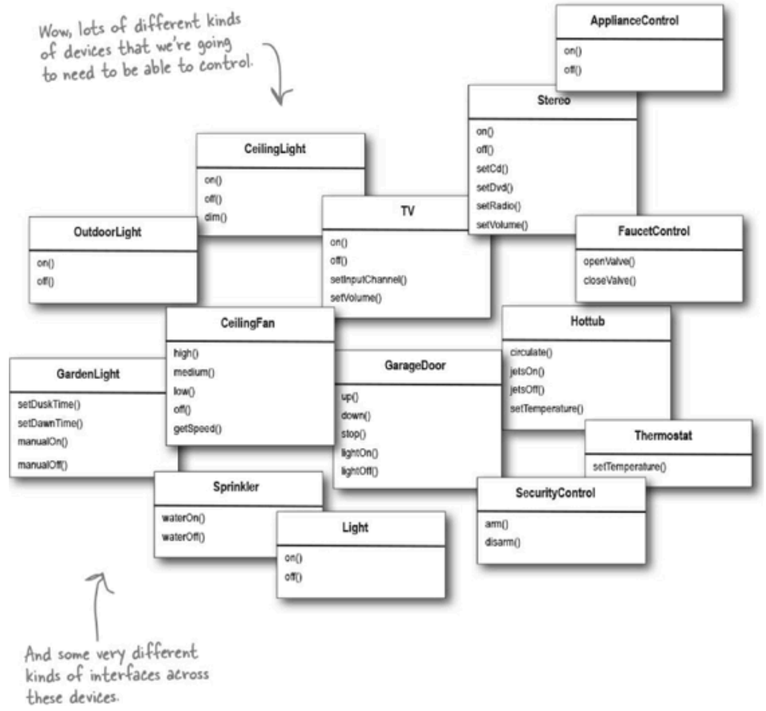
Behavioral

Command Pattern

Encapsulating Invocation

Command Pattern

Creating class that encapsulate methods invocation.



Problem statement

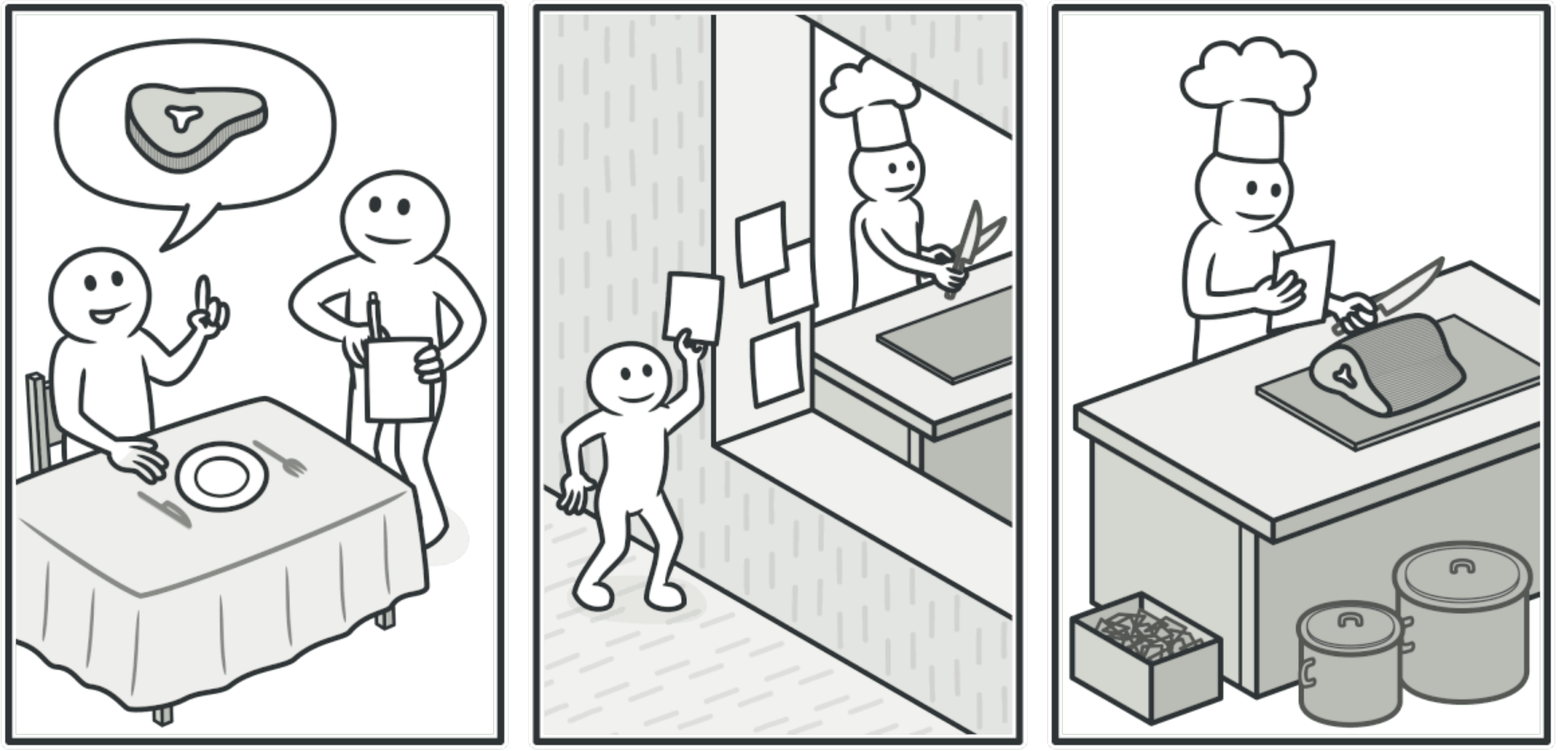
- Imagine that you're working on a new app that controls smart devices from many partners. They provide their device specifications and libraries to control their smart devices. Your job is to design an app that supports and controls all of those devices.

Solution

- Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

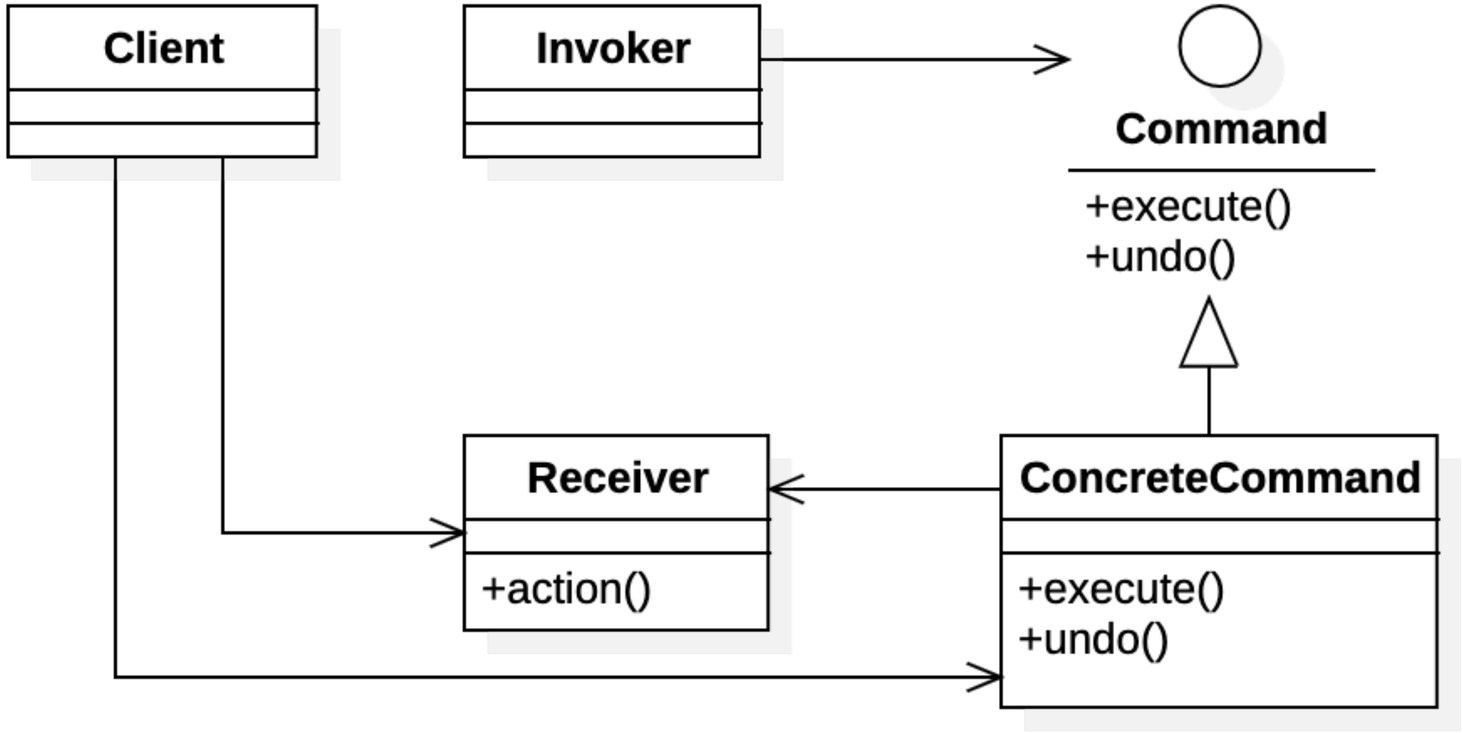
Command Pattern in Real Life

Order a meal in restaurant



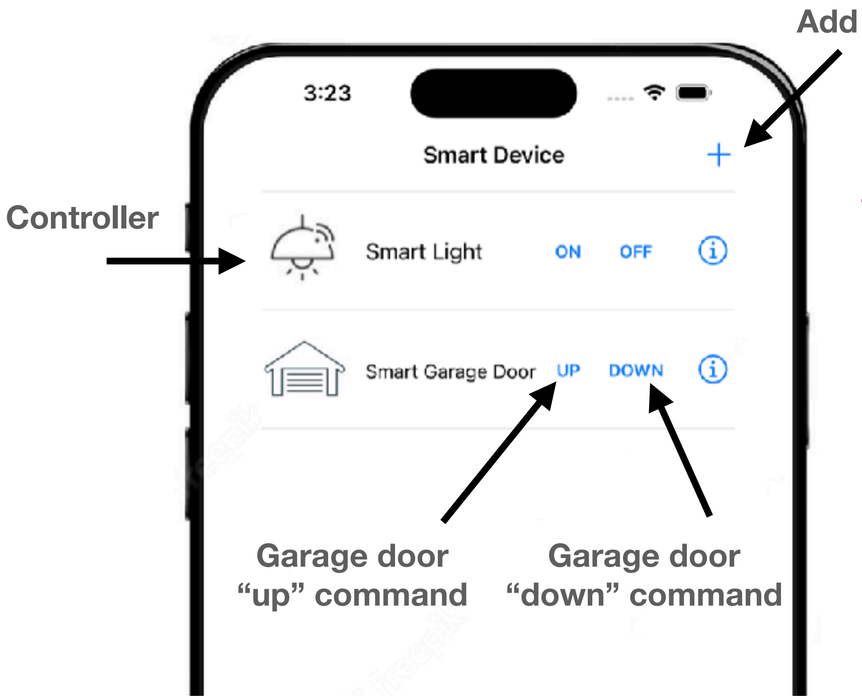
Command Pattern

Class Diagram



Example

Smart Device Controller App




```
let controllers: [Controller] = [
        Controller(
            icon: UIImage(named: "light")!,
            name: "Smart Light",
            commands: [
                LightOnCommand(),
                LightOffCommand()
            ]
        ),
        Controller(
            ....
    ]
```

```
struct Controller {
    let icon: UIImage
    let name: String
    var commands: [Command] = []
}
```

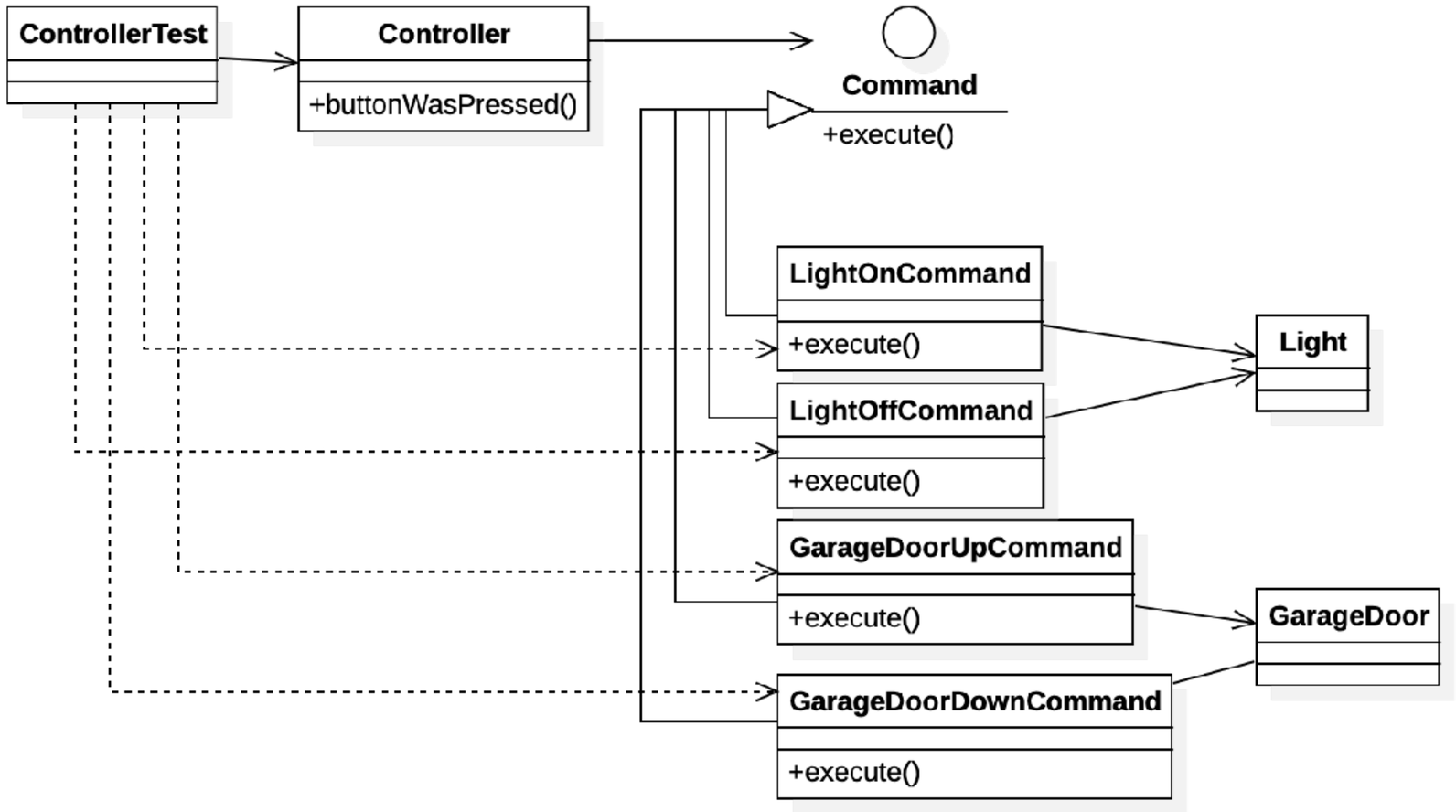
```
protocol Command {
    var title: String { get set }
    func execute()
}

struct GarageDoorUpCommand: Command {
    var title: String = "up"

    func execute() {
        print("Garage door is up!")
    }
}

struct GarageDoorDownCommand: Command {
    var title: String = "down"

    func execute() {
        print("Garage door is down!")
    }
}
```

```
struct LightOnCommand: Command {
    var title: String = "ON"
    func execute() {
        print("Light on is press!")
    }
}
```

```
struct LightOffCommand: Command {
    var title: String = "OFF"
    func execute() {
        print("Light off is press!")
    }
}
```

Remote Control

Class Diagram



Structural

Adapter Pattern

Convert one Interface to another

Adapter & Facade Pattern

Convert one Interface to another.

Problem statement

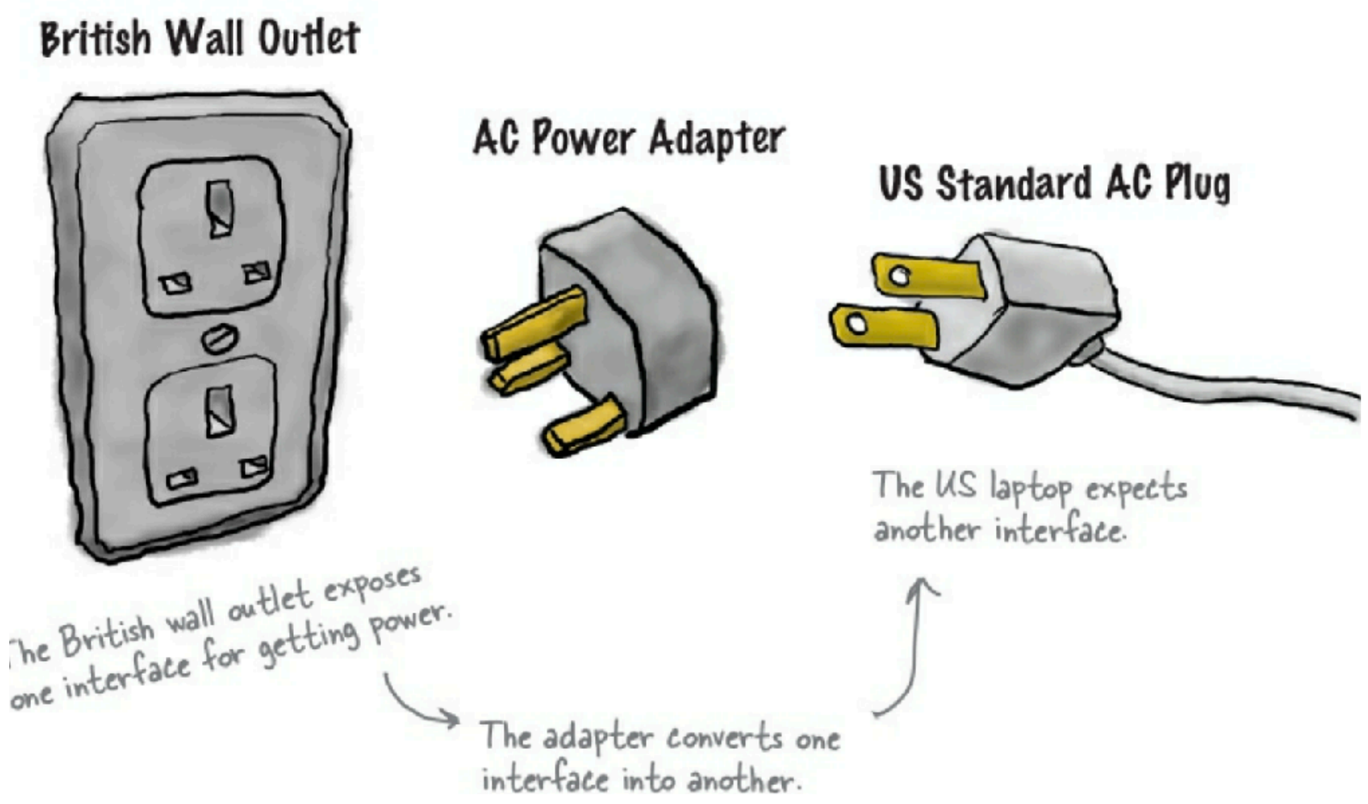
- When you need to use an existing class and interface is not the one you need.

Solution

- Converts one interface to another that client expected. Adapter pattern lets incompatible interface classes work together.

Adapter all around us.

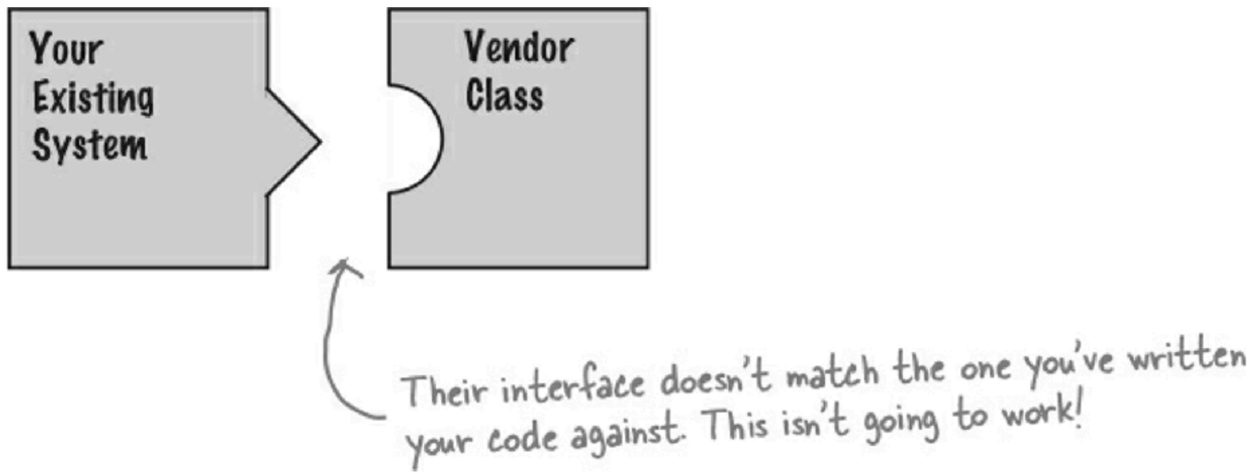
in real life

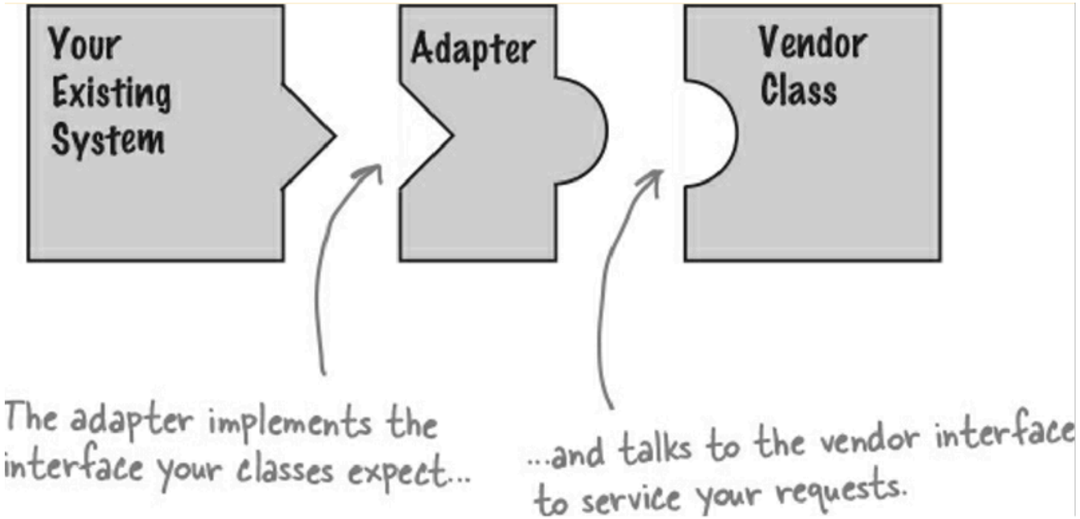


Head First Design Patterns, Chapter 7: Being Adaptive.

Using Adapter

Use it when you have an existing system structure but it incompatible with the vendor interfaces.

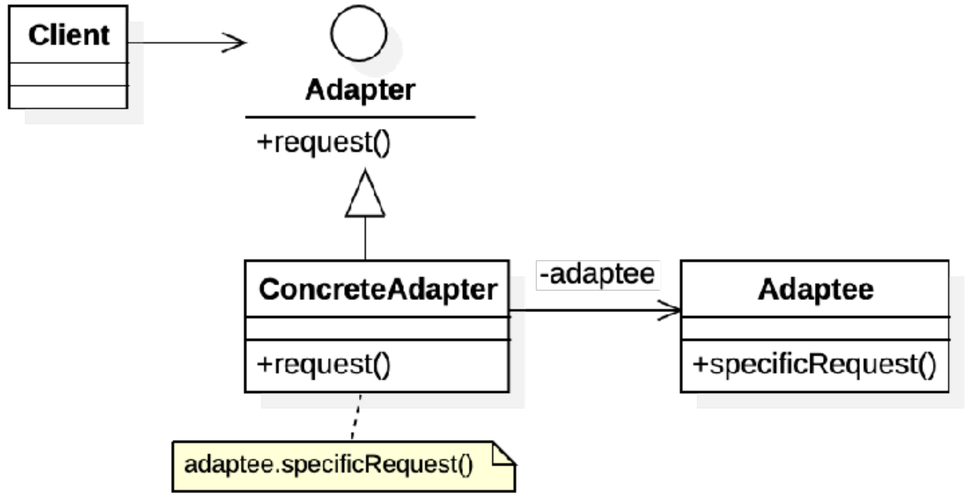


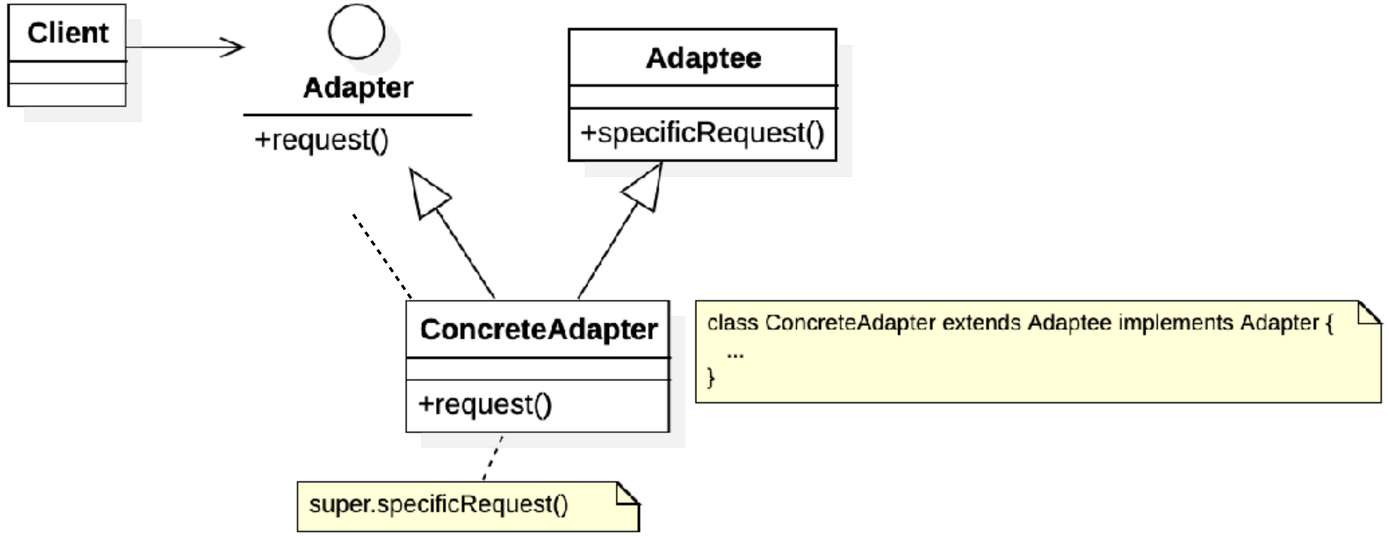


Doesn't alter the interface, but adds responsibility !!!

2 kinds of Adapters

UML class Diagram





Composition (Object Adapter)

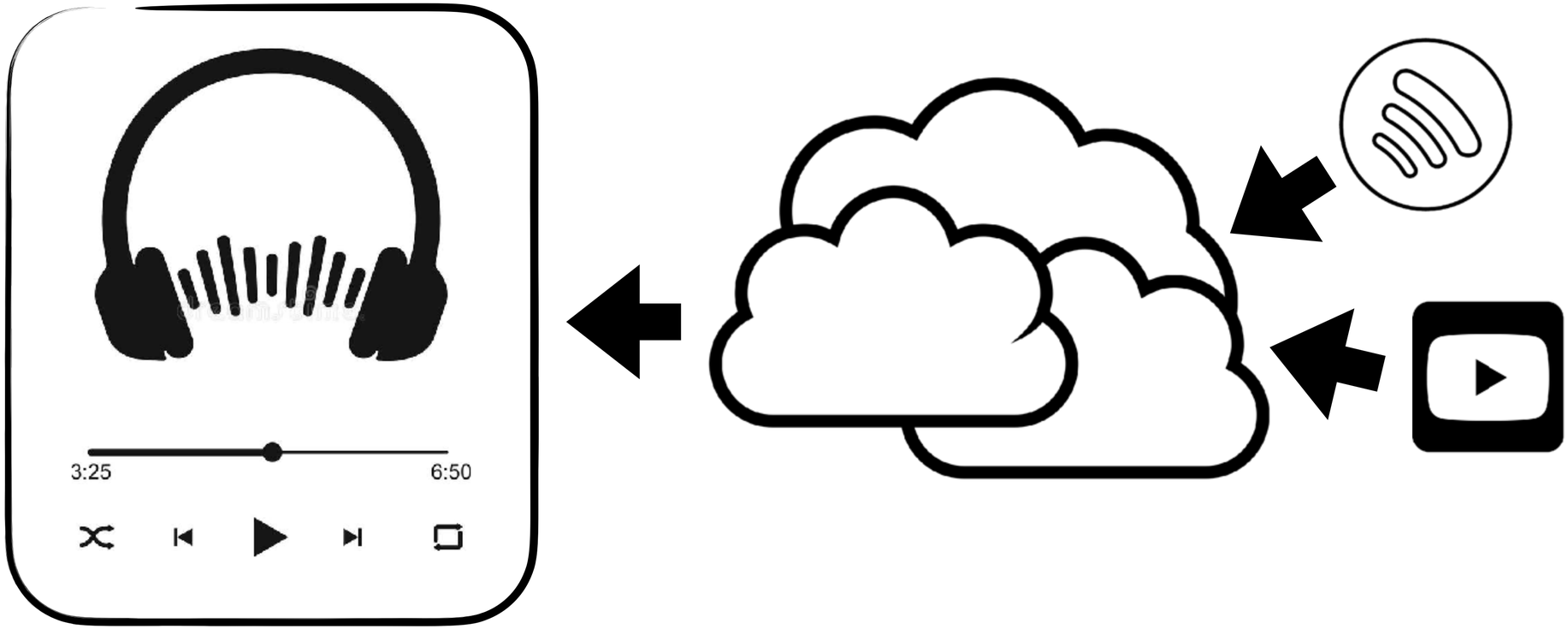
- Can work with many Adaptee sub-class.
- Re-implement the wrapped methods.
- More flexible.

Inheritance (Class Adapter)

- Committed to one subclass (with many interfaces).
- Reuse implemented logic from super class.
- More reusable.

Lab: Media Player

listen from Spotify and more…



Structural

Façade Pattern

Decouples a client from a complex subsystem

Facade Pattern

Creating objects as Template Method is to implementing an algorithm.

Problem statement

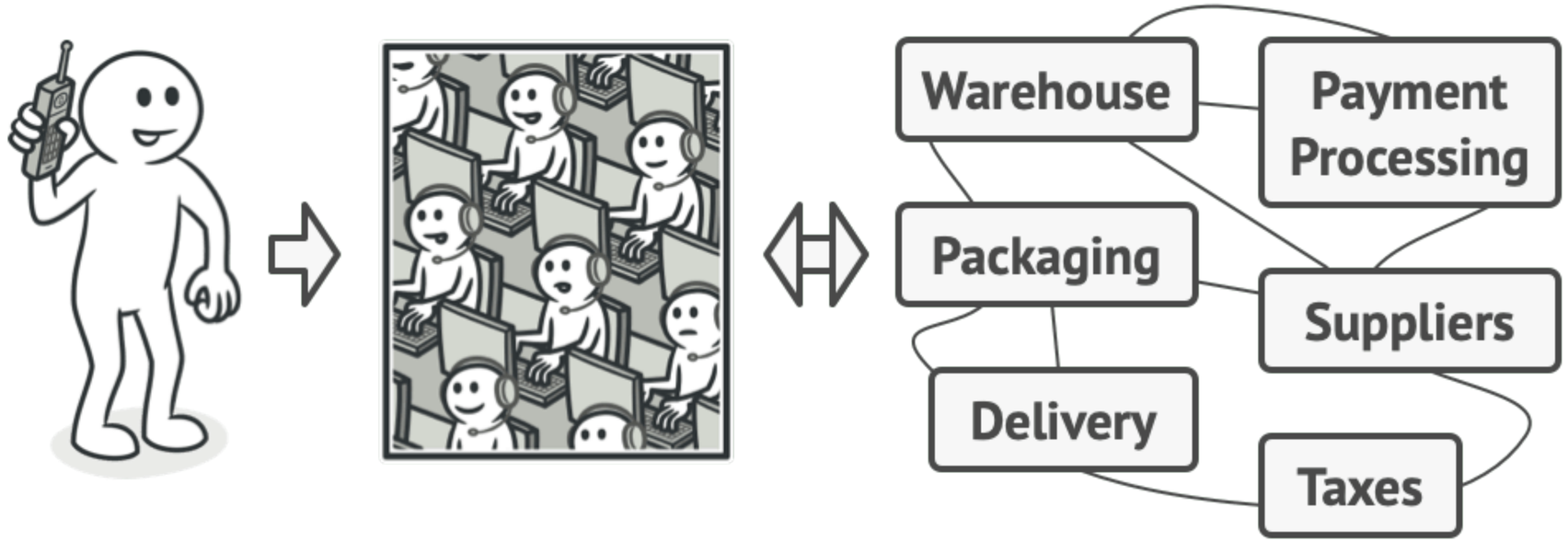
- Imagine that you must make your code work with a *broad set of objects* that belong to a sophisticated library or framework. Ordinarily, you'd need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on.
- As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.

Solution

- The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Make an interface simpler. Define a higher-level interface that makes the subsystem easier to use.

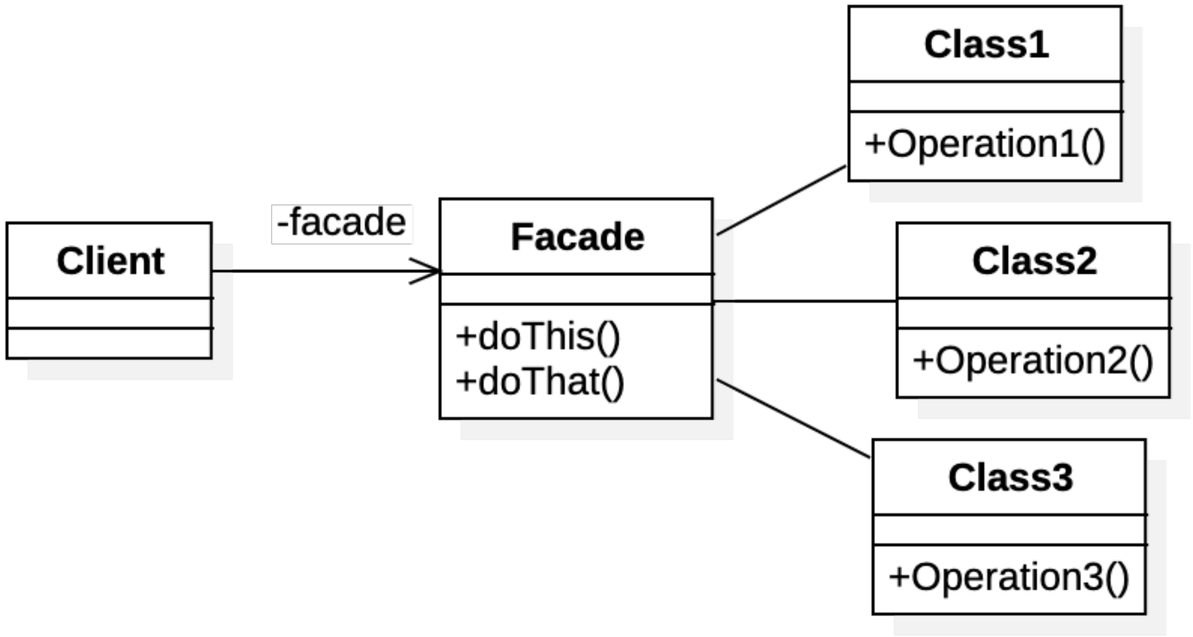
Facade

real world analogy



Facade Pattern

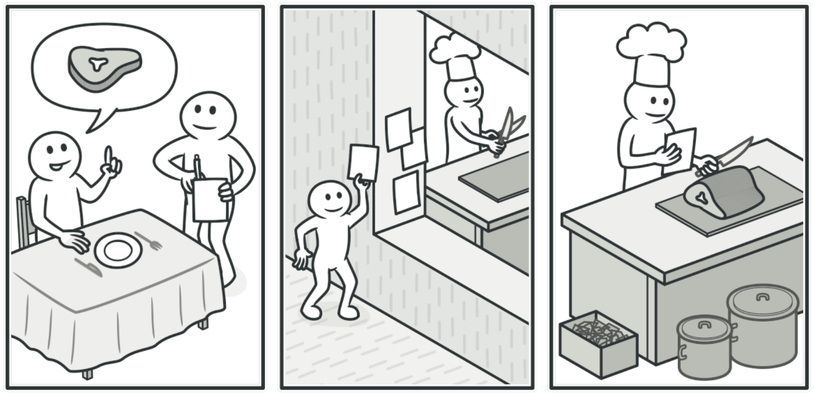
UML Class Diagram



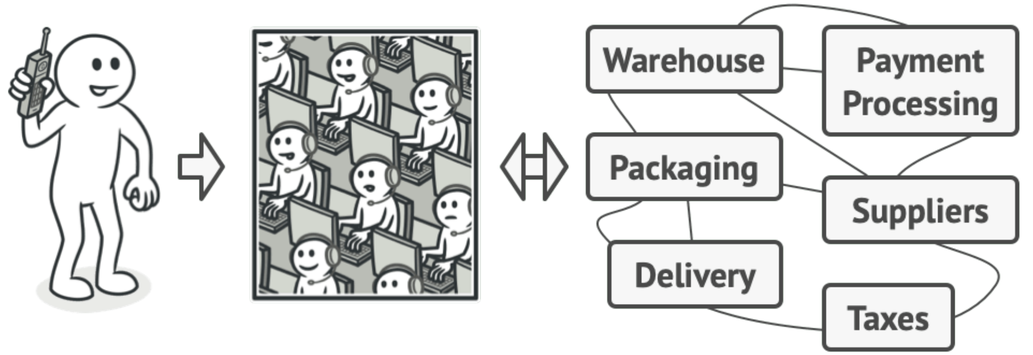
Questions?

Group discussion

- What different between Command pattern and Facade Pattern?



Command Pattern

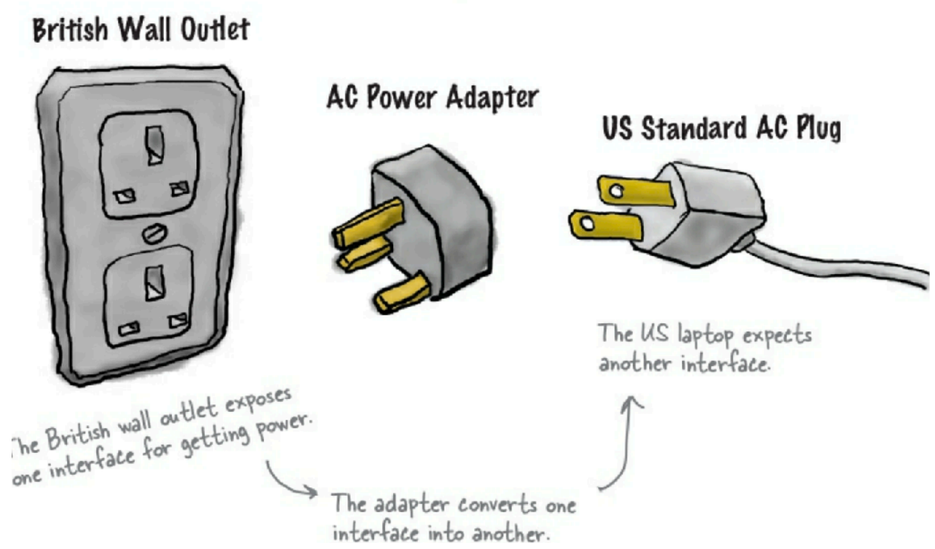


Facade Pattern

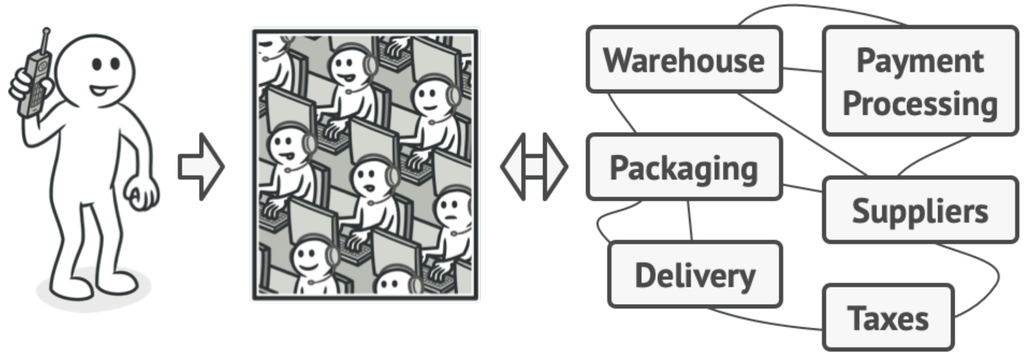
Questions?

Group discussion

- What different between Adapter pattern and Facade Pattern?



Adapter Pattern



Facade Pattern

Behavioral

State Pattern

handle state & flow without if, then, else

State Pattern

Creating objects as Template Method is to implementing an algorithm.

Problem statement

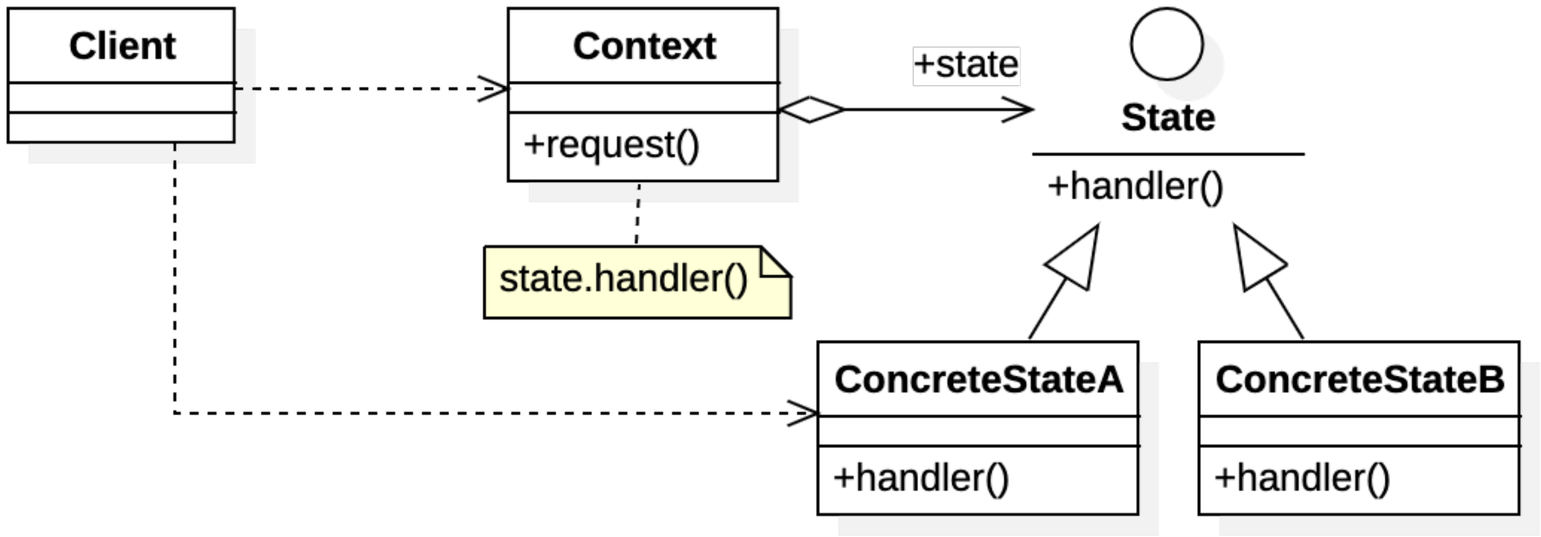
- When apply state machine diagram into code. Its usually implement with lots of conditional statements (if-else, switch/case) that select the appropriate behavior depending on the current state of the object.
- Lead to difficult to add or change when need to add state or change state transition behaviour.

Solution

- Encapsulate state-based behaviour and delegate behaviour to the current state.
- Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

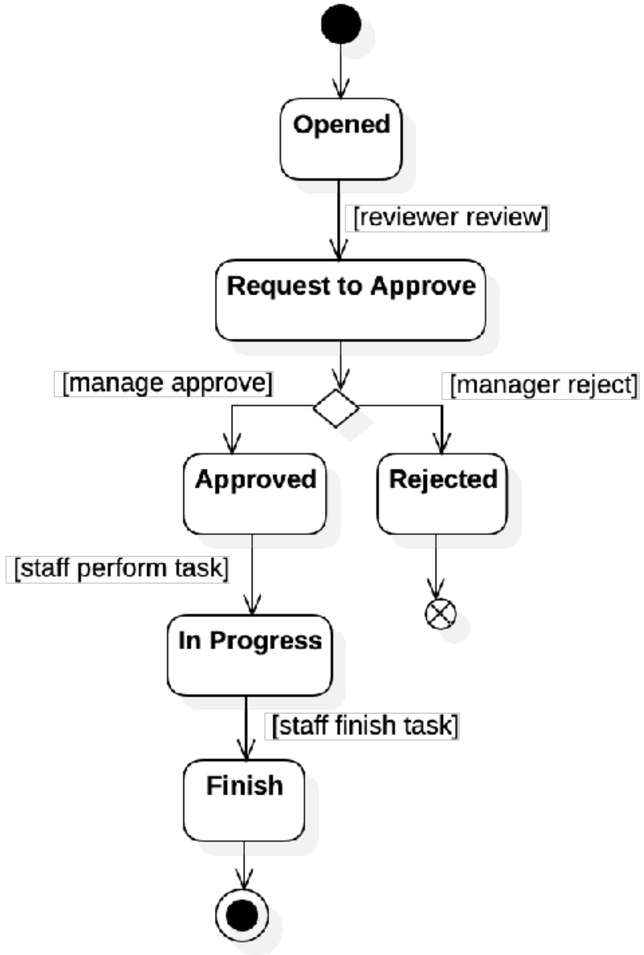
State Pattern

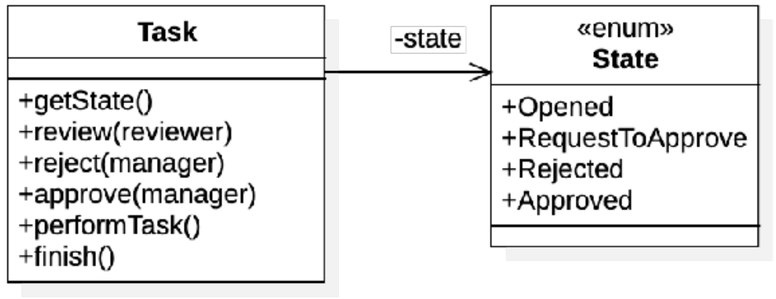
class diagram



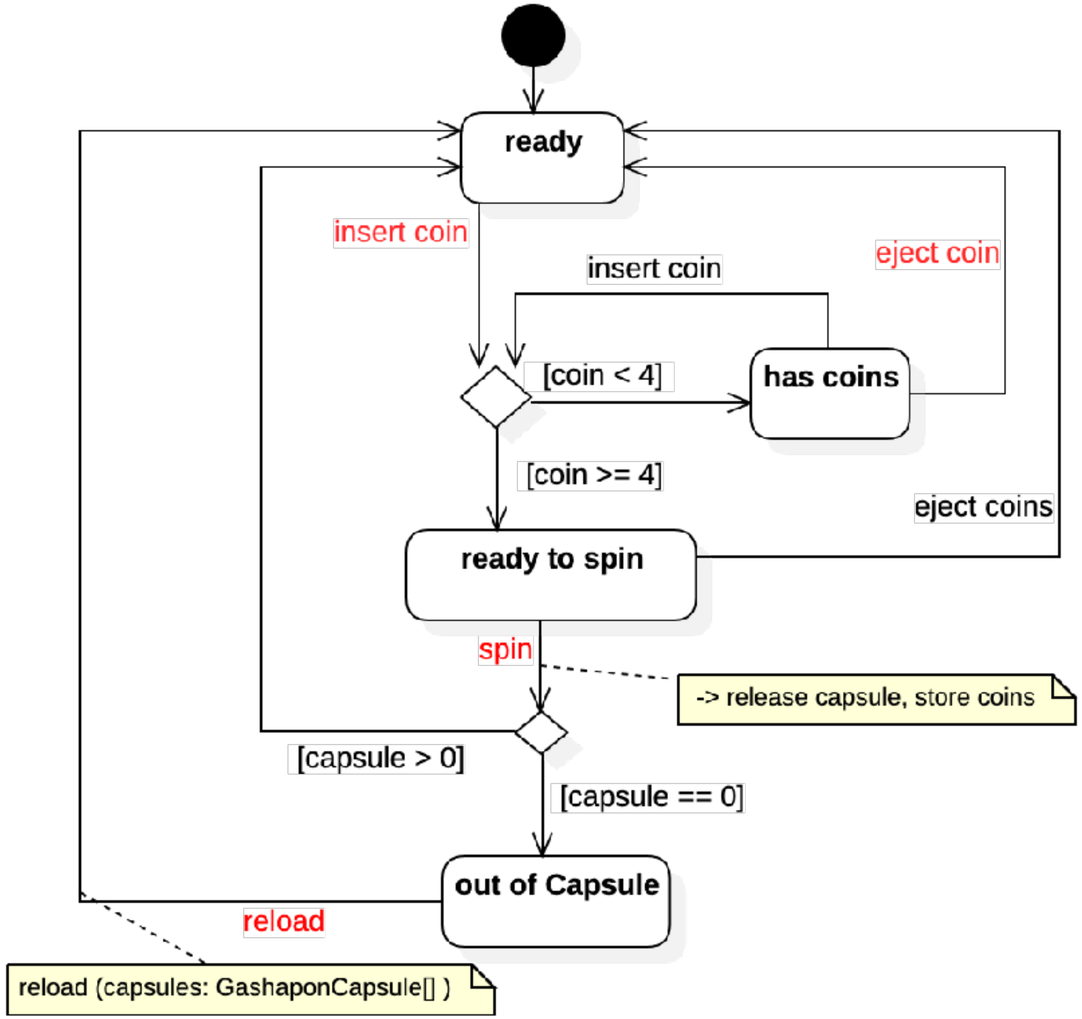
UML State Diagram Revisit

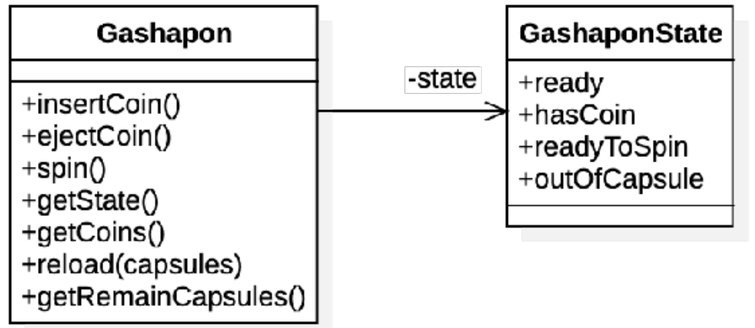
or State-Machine, State chart Diagram



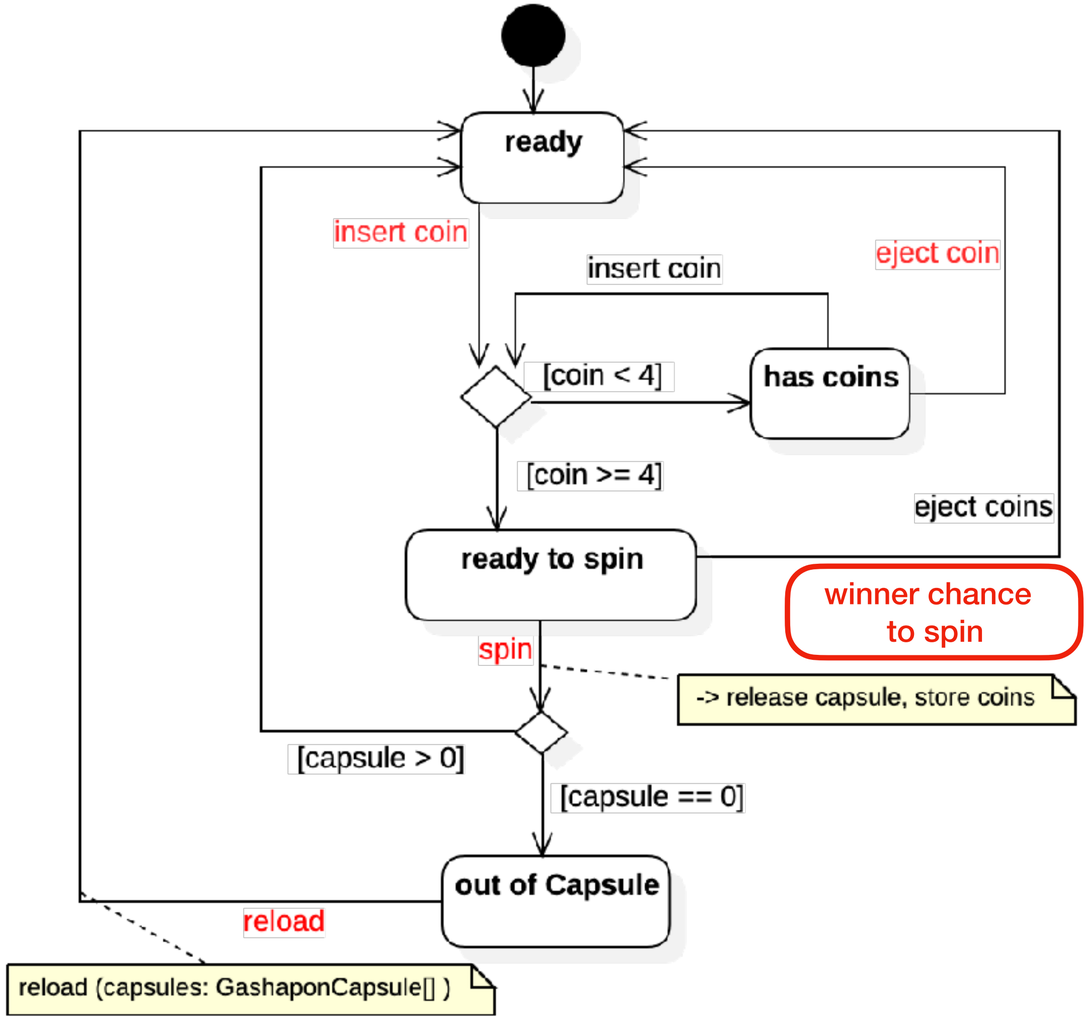


Lab: Gashapon Smart Machine

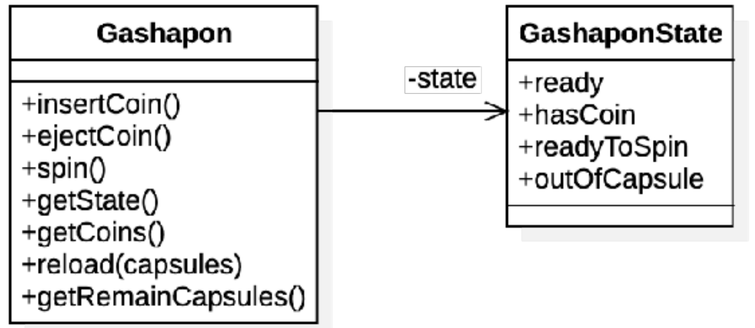




Requirement Changed!!!



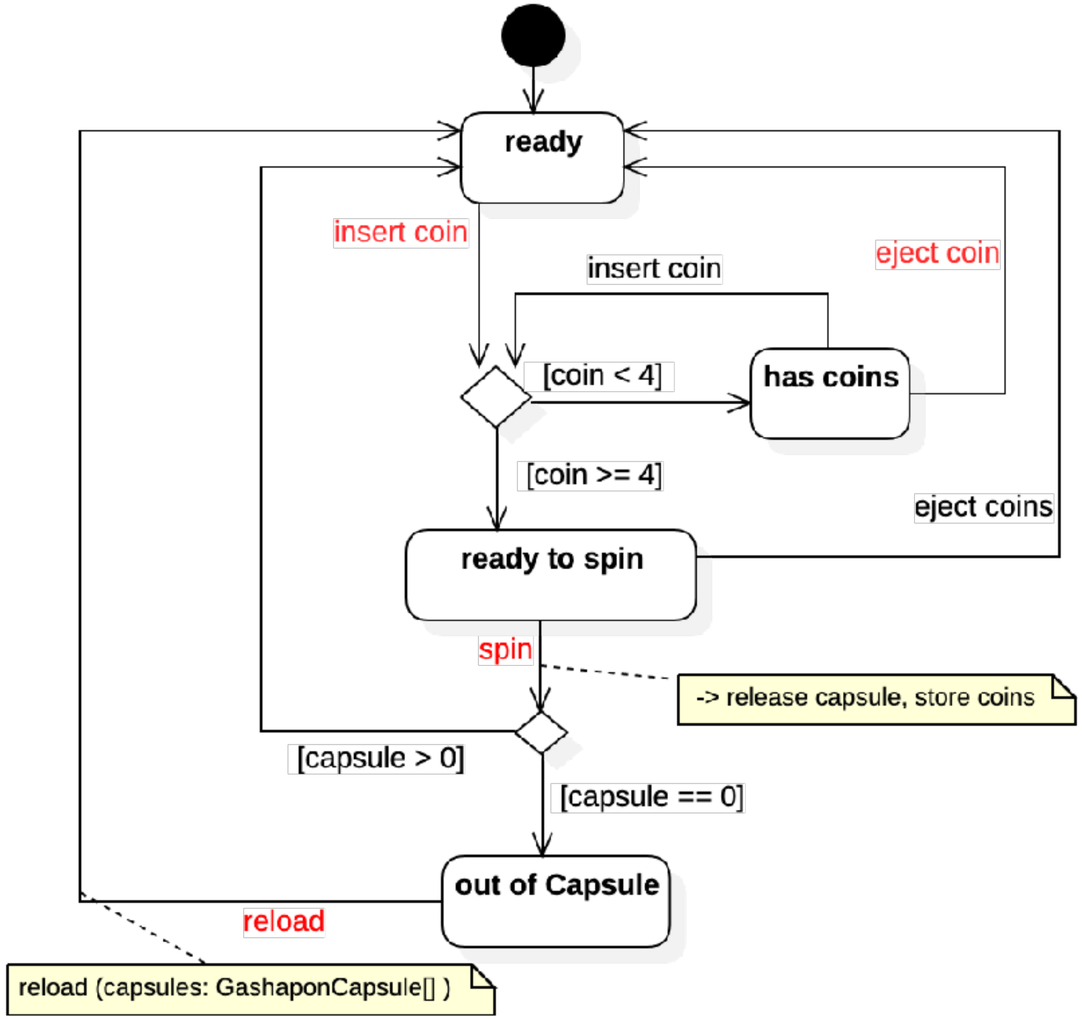
… add somewhere here?



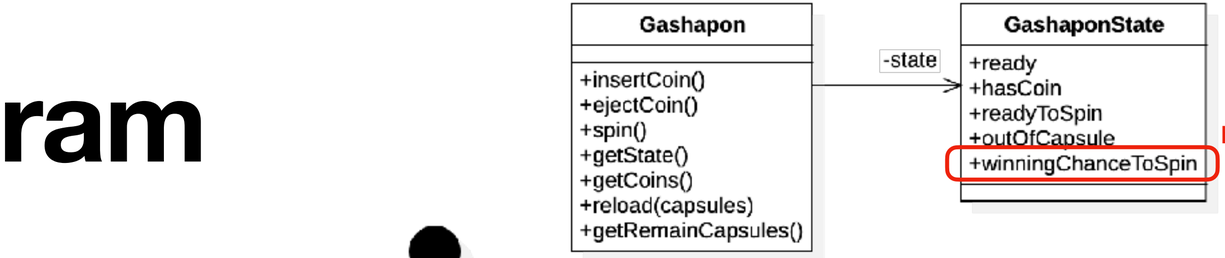
Be a Winner!!!

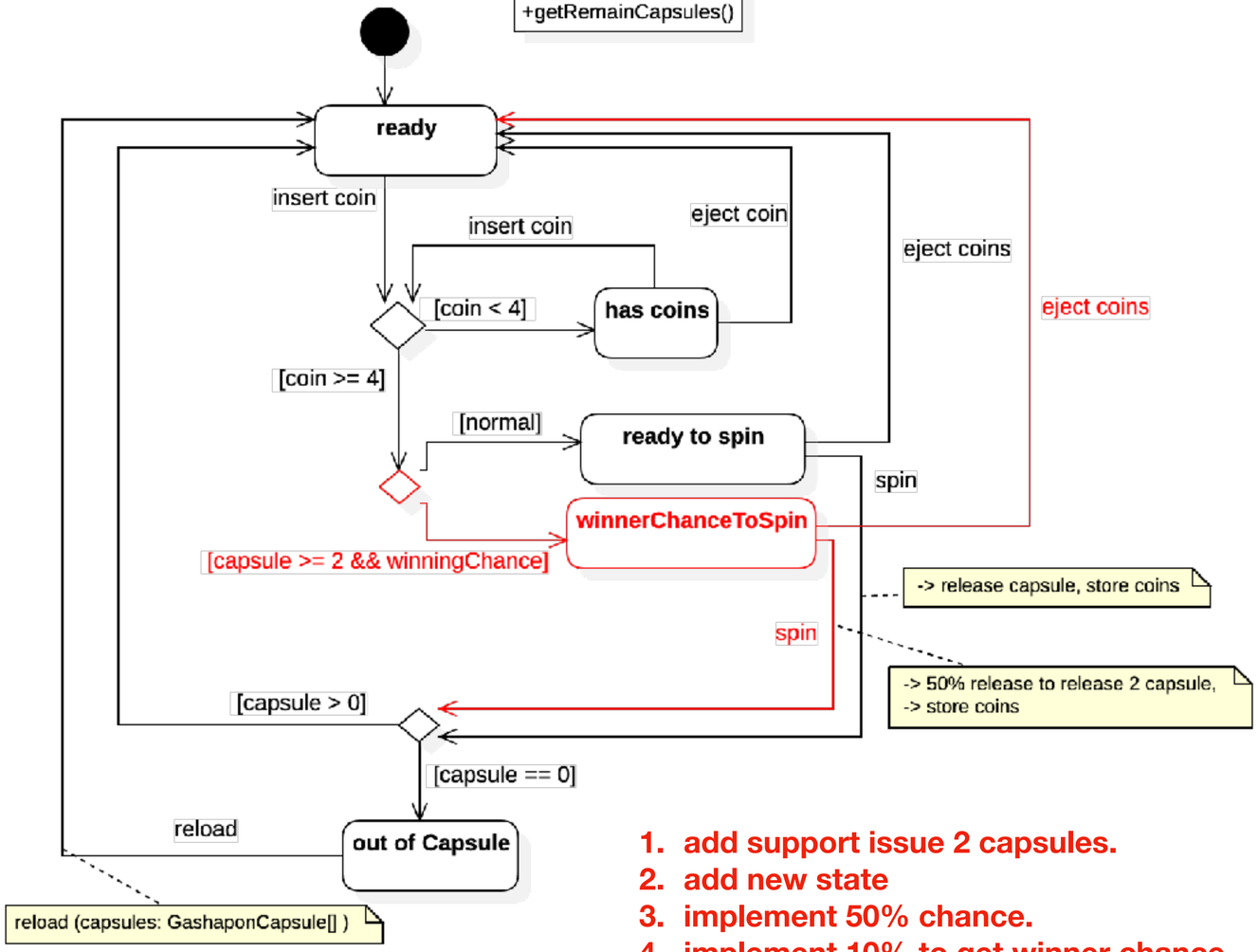
10% chance to get 2 Gashapons when WinnerChanceSpin alert!

Compare State Diagram



Version 1





Version 2

1. add support issue 2 capsules.
2. add new state
3. implement 50% chance.
4. implement 10% to get winner chance.

Adding new state is challenging!

```
export enum GashaponState {
    ready = 'ready',
    hasCoin = 'hasCoin',
    readyToSpin = 'readyToSpin',
    outOfCapsule = 'outOfCapsule',
}
```

```
winnnerChanceSpin = ‘winnnerChanceSpin’
```

```
insertCoin() {
    if (this.state === GashaponState.readyToSpin) {
        throw new Error('Cannot insert coin when ready to spin');
    }
    if (this.state === GashaponState.outOfCapsule) {
        throw new Error('Cannot insert coin when out of capsule');
    }
    if (this.state == GashaponState.ready || this.state == GashaponState.hasCoin) {
        this.coins += 1;
        if (this.coins < this.needs) {
            this.state = GashaponState.hasCoin;
        }
        if (this.coins == this.needs) {
            this.state = GashaponState.readyToSpin;
        }
    }
}
```

```
ejectCoins(): number {
    if (this.state === GashaponState.ready) {
        throw new Error("You haven't insert any coin");
    }
    if (this.state === GashaponState.outOfCapsule) {
        throw new Error("You haven't insert any coin");
    }
    if (this.state === GashaponState.hasCoin || this.state === GashaponState.readyToSpin) {
        const coinToReturn = this.coins;
        this.coins = 0;
        this.state = GashaponState.ready;
        return coinToReturn;
    }
    return 0;
}
```

Structural

Proxy Pattern

Control & Manage Access

Proxy Pattern

Problem statement

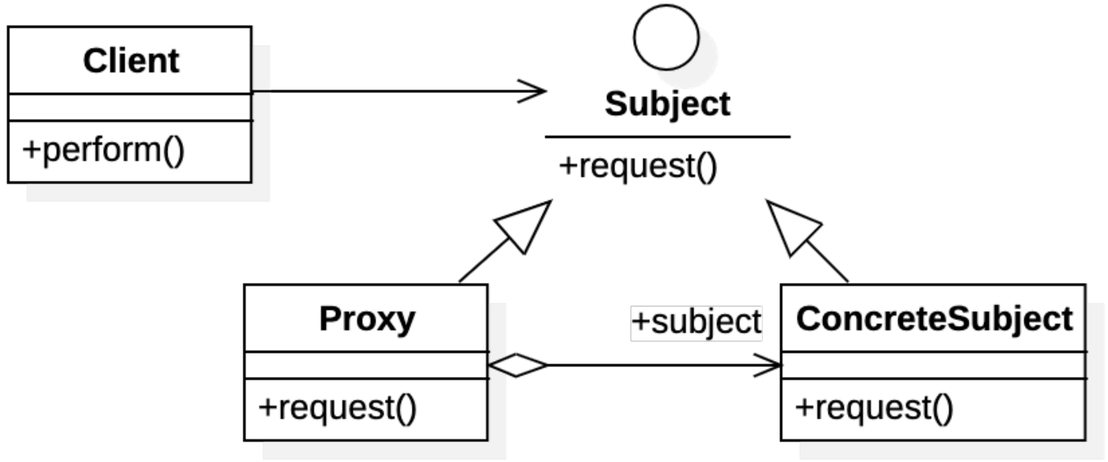
- The client object use other uncontrollable behavior object. You need something to control it’s behaviour but the client shouldn’t notice.

Solution

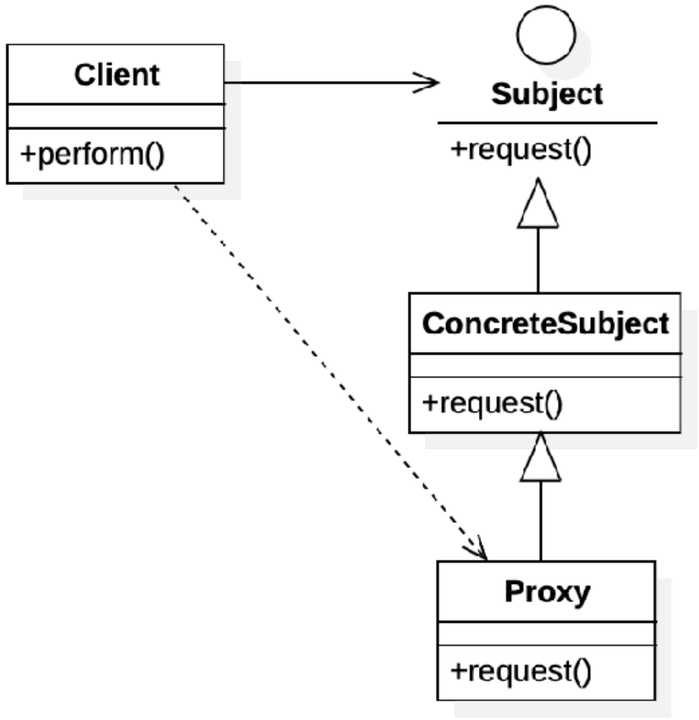
- The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the original object’s clients.
- Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.

Proxy Pattern

UML class diagram

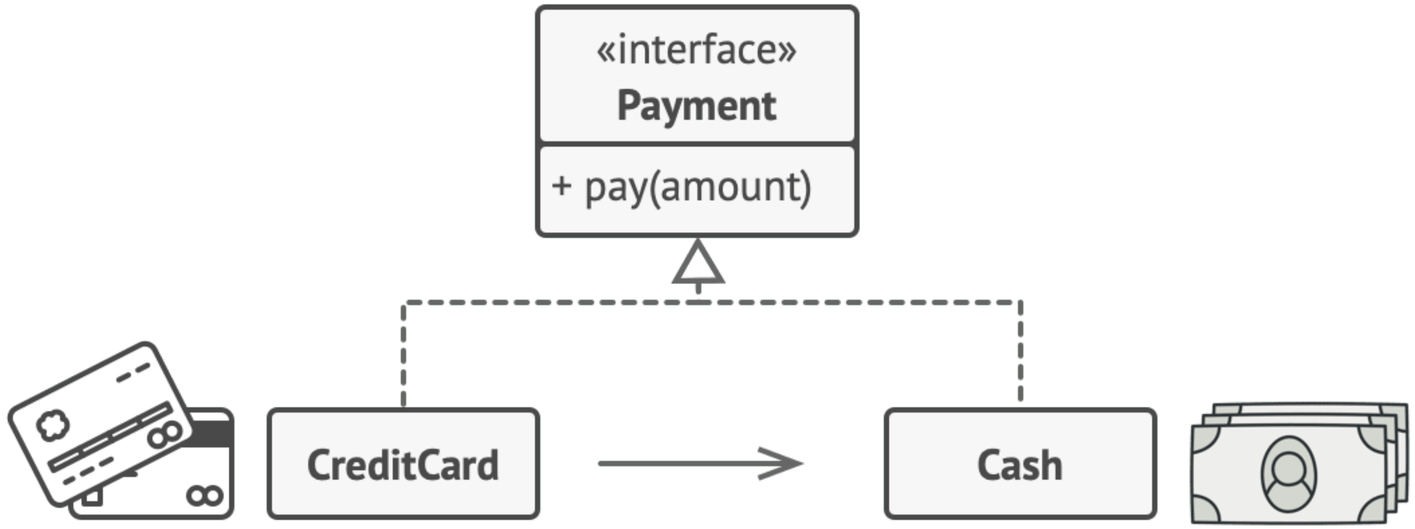


Association / Delegation



Inheritance

Real world Example



Structural

Template Method Pattern

Encapsulating pieces of Algorithm

Template Method Pattern

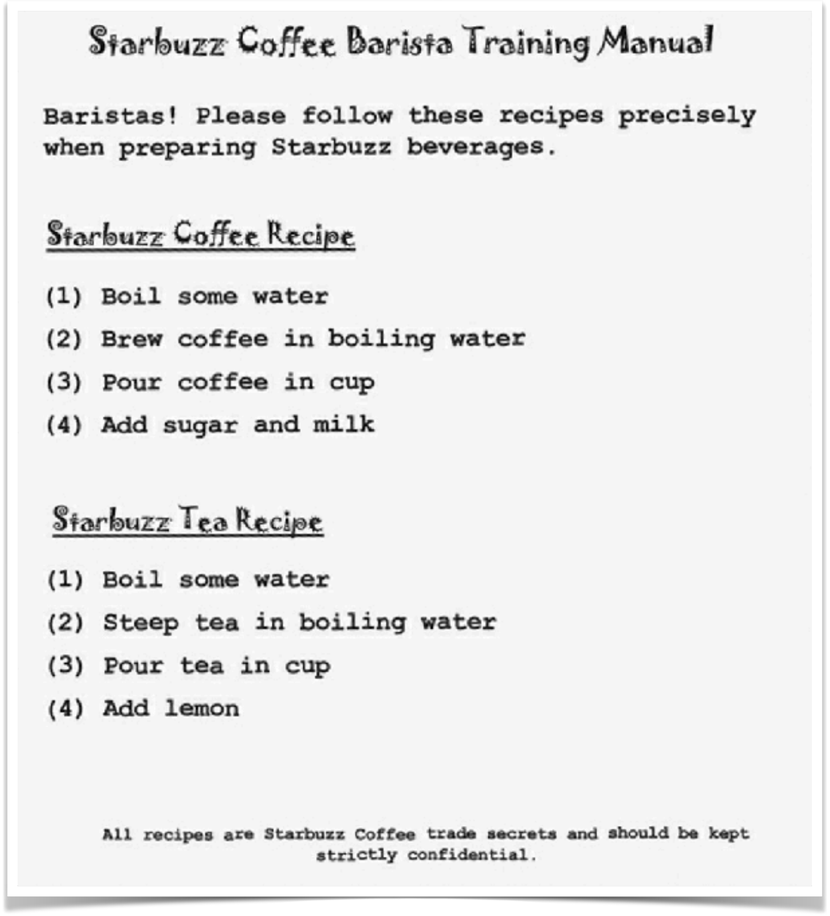
Problem statement

- You are creating an app that needs to use the same algorithm (or steps) in many different implementation, but you want to avoid duplicating the code.

Solution

- The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses.
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

Real world example



Starbuzz Coffee Barista Training Manual

Baristas! Please follow these recipes precisely when preparing Starbuzz beverages.

Starbuzz Coffee Recipe

(1) Boil some water
(2) Brew coffee in boiling water
(3) Pour coffee in cup
(4) Add sugar and milk

Starbuzz Tea Recipe

(1) Boil some water
(2) Steep tea in boiling water
(3) Pour tea in cup
(4) Add lemon

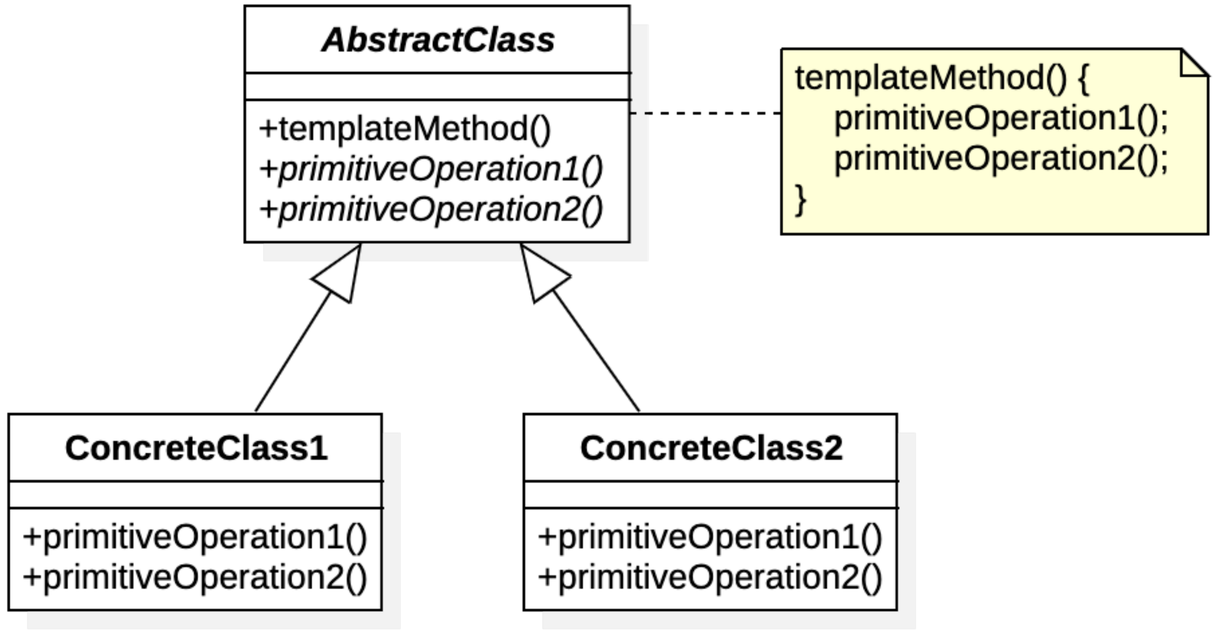
All recipes are Starbuzz Coffee trade secrets and should be kept strictly confidential.

```
export class CoffeeMachine {

    private waterIsBoilded = false;
    private CoffeeGrindsIsBrewed = false;
    private pouredInCup = false;
    private sugarAndMilkAdded = false;

    public brew(): string {
        this.boildWater();
        this.brewCoffeeGrinds();
        this.pourInCup();
        this.addSugarAndMilk();

        return 'Here is your coffee!';
    }

    private brewCoffeeGrinds() {
        this.CoffeeGrindsIsBrewed = true;

    }

    private boildWater() {
        this.waterIsBoilded = true;
    }

    private pourInCup() {
        this.pouredInCup = true;
    }

    private addSugarAndMilk() {
        this.sugarAndMilkAdded = true;
    }
}
```

```
export class TeaMachine {

    private waterIsBoilded = false;
    private TeaGrindsIsBrewed = false;
    private pouredInCup = false;
    private sugarAdded = false;

    public brew(): string {
        this.boildWater();
        this.brewTeaGrinds();
        this.pourInCup();
        this.addSugar();

        return 'Here is your tea!';
    }

    private brewTeaGrinds() {
        this.TeaGrindsIsBrewed = true;

    }

    private boildWater() {
        this.waterIsBoilded = true;
    }

    private pourInCup() {
        this.pouredInCup = true;
    }

    private addSugar() {
        this.sugarAdded = true;
    }
}
```

Template Method Pattern

UML class diagram



In Summary…

Design Patterns

the definition

- A Pattern is a **solution** to a **problem** in a **context**.
  - The **context** is the situation in which the pattern applies. This should be a recurring situation.
  - the **problem** refers to goal you're trying to achieve in this context, but it also refers to any constraints that occur in the context.
  - The solution is what you're after: a general design that anyone can apply that resolves the goal and set of constraints.

Let say …

"If you find yourself in a context with a problem that has a goal that is affected by a set of constraints, then you can apply a design that resolves the goal and constraints and leads to a solution."

Anyway, this is not a complete definition of a pattern. We haven't even giving a name.

Pattern Catalogs

name its, group its…

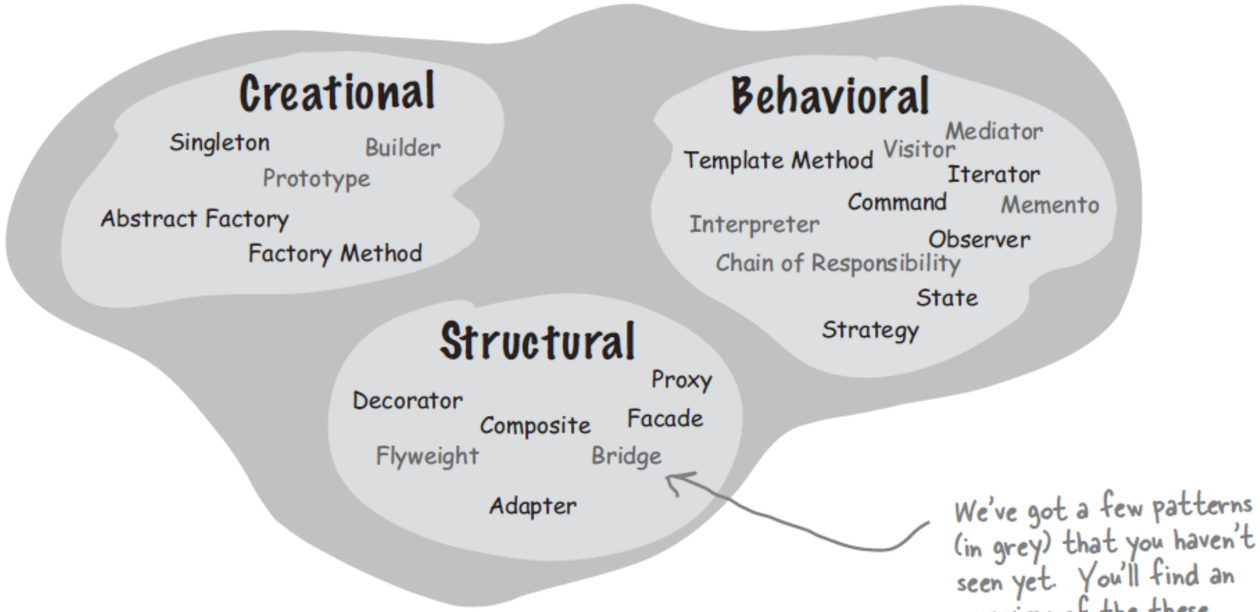
- Without a name, the pattern doesn't become part of **vocabulary** that can shared with other developers.
- That way, other developers can **quickly recognize** the patterns you're using when you documented in your design. So that they will know which parts are using patterns and which parts are just classic design.
- The way to describe patterns and collecting them together is put its into **patterns catalogs**.
- It help us to think of the items at the more abstraction level.
- BTW. Many developers confusing about the categories where the patterns belong. For example :- why Decorator belong to structural not creational. Just focus on how you compose the objects **dynamically to gain functionality** to fit the problem rather than focus on why it be here, not there.

The Classic GoF Pattern Catalogs

and much more…

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.

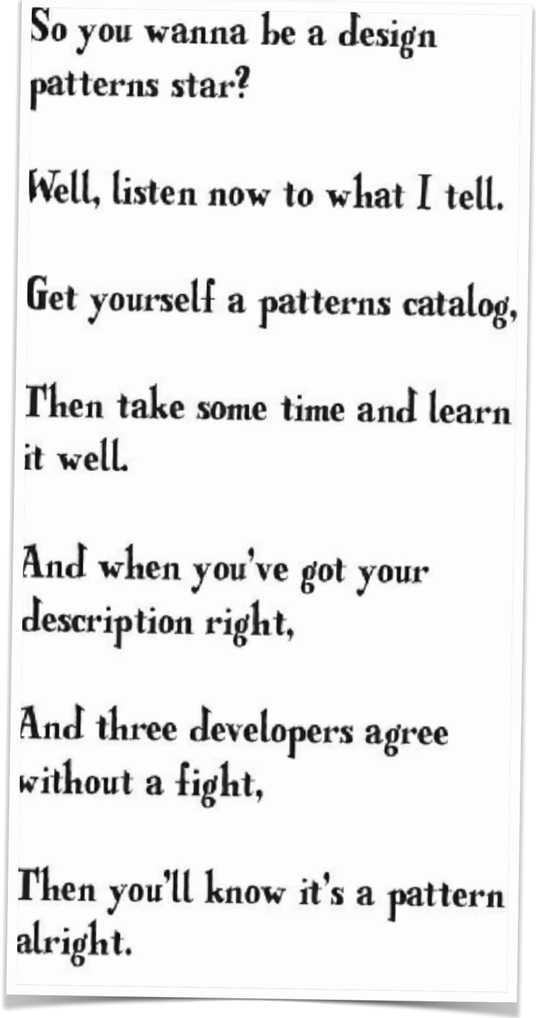


Structural patterns let you compose classes or objects into larger structures.

Picture from Head First Design Patterns 2nd edition book.
Design Patterns: Element of reusable Object-Oriented Software (GoF book)

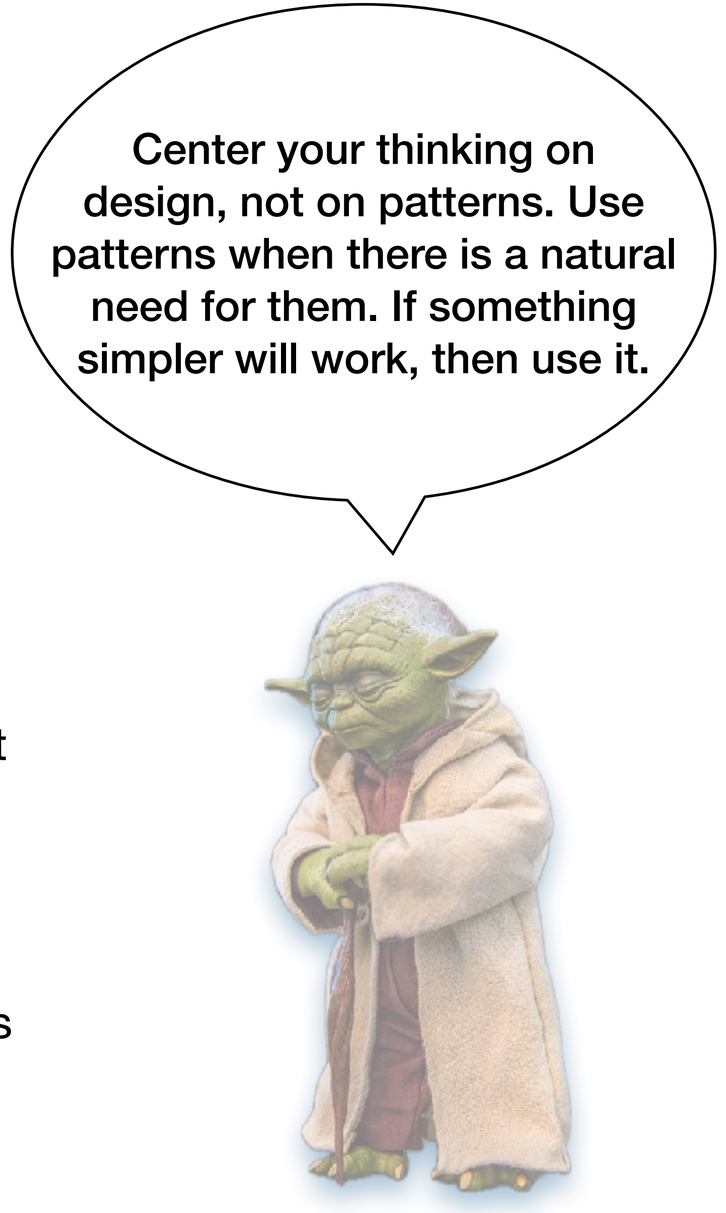
Is it possible to create my own pattern?

- The patterns are discovered, not created.
- It’s not easy task and doesn’t happen quickly, not often. Being a “patterns writer” is taken commitment.
- You don’t have a pattern until ti passes the “Rule of Three”. This rule states that a pattern can be called a pattern only if it has been applied in a real-world solution at lease three times.
- BTW. Pattern document will not cover in this class.



Patterns mind

- Pattern can introduce complexity. Keep it simple.
- Your goal is to solve the problem, not “how can I apply the pattern to this problem”.
- Patterns aren’t magic bullet. You can’t plug one in, compile, and them problem solved. You need to think through the consequence for the rest of your design.
- You know you need the pattern when you make sure that simple solution will not meet your needs. Give the consideration before you commit to using a pattern.
- Refactoring time is a patterns time.
- Take out what you don’t really need. Don’t be afraid to remove a Design Patterns from your design.
- If you don’t need it now. You may don’t need to do it now.



Your Mind on Patterns

beginner, intermediate, Master

The beginner use patterns everywhere.

The intermediate mind starts to see
where are needed and where they aren’t.

The master mind is able to see patterns
where they fit naturally.

One more thing. Anti-Patterns

An anti-pattern tells you how to go from a problem to a BAD solution.

- Anti-pattern is a recurring bad solution to a common problem.
- By document it. We can prevent other developers from making the same mistake. After all avoiding bad solutions can be just as valuable as finding good ones.
- An anti-pattern tells you why a bad solution is attractive.
- An anti-pattern tells you why that solution is bad in the long term.
- An anti-pattern suggests other applicable patterns that may provide good solution.

An anti-pattern always looks like a good solution but then turns out to be a bad solution when it is applied.

By documenting anti-patterns we help others to recognize bad solutions before they implement them.

Like patterns, there are many types of anti-patterns including development, OO, organizational, and domain-specific anti-patterns.

References

- Refactoring Guru - Design Pattern
  https://refactoring.guru/design-patterns
- Design Patterns: Elements of Reusable Object-Oriented Software
  https://shorturl.at/muFZ4
- Head First Design Patterns: 2nd edition
  https://shorturl.at/vDHJX
- Code on Github
  https://github.com/olarn/GoF-Design-Pattern