Announcements

- GCP assignment and midterm survey due tonight
- Project 1 due Oct 8
  - Team presentations next Tuesday in class
  - All Project 1 deliverables due – presentation slides before class, report and github repository link at 11:59 PM
  - No class this Thursday – work on your project 1
- Project 1 design due Oct 13th (extended)
- Sit with your teammates today in class

Georgia Tech

CS3300 Introduction to Software Engineering

Project 1 Submission Guidelines

Dr. Nimisha Roy
- nroy9@gatech.edu

Submission Instructions

What is to be Submitted? – 1 presentation slide deck (ppt or pdf not SharePoint) and the **GitHub Link for your software.**

Software: The goal is to provide enough material, so that other people can use it and continue your work, if you are to open-source it --- in other words, you should make it easy and attractive for others to use your work.

Project 1 Rubric (17.5% of final grade)

Presentation & Demo in Class (9.7%)

Presentation (5 mins)

- Problem Statement and Frontend/Backend technologies described (~30 secs)
- AI tools incorporated, experiments conducted, results (~2 mins)
- Distribution of team effort described (~30 secs)
- Stay within Time limit
- Demo (~2 mins)
  - Landing page
  - Working Backend and Front End – All features
  - Cloud deployment

Report and Code (7.8%)

Details of the project in Report

Readme

- Release Notes - New features, Bug Fixes, Issues
- Readme
  - Install Guide
  - Pre-requisites:
  - Dependent libraries
  - Download instructions
  - Build instructions (if needed
  - Installation of actual application
  - Run instructions
  - Troubleshooting

Presentation guidelines

- Time allotted per group: 5 minutes + 0.5 minutes for Q/A
- Include the necessary details that you think are the most important and deserve to be discussed within the time constraints. *Consider extra credit requirements here*
- Include team effort distribution.
- Workflow- **Show a demo of the application**, include the deployed weblink for the demo in your ppt
- Submit your presentations on Canvas by 4:30 PM on 10/8
- Group Submission
- Come prepared/Be ready with HDMI connector, laptop etc. for demo. I will have all presentations on the desktop, so you shouldn’t need to connect to your laptop if you have the deployed GCP link

Software Guidelines

- Submit the GitHub Repository Link
- Include a README.txt with all details about the release notes and install guide.
- Group submission. One person per group will submit.

Georgia Tech

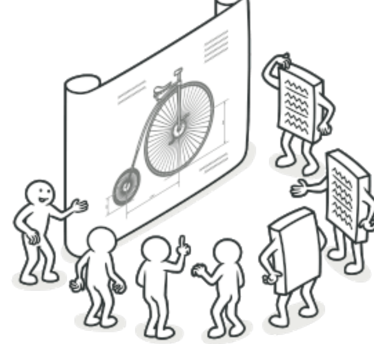
CS3300 Introduction to Software Engineering

Lecture 12: Design Patterns

Dr. Nimisha Roy ▸ nroy9@gatech.edu

Some slides adapted from refactoring.guru

What are Design Patterns?

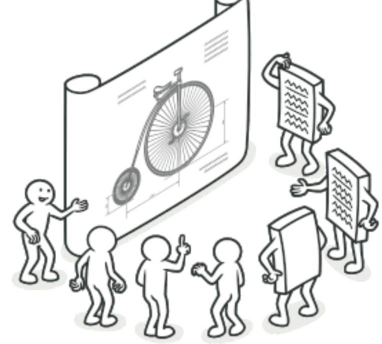


- **Typical solutions to common problems in software design**. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

- Patterns define industry standard strategies for solving common problems.

- By using design patterns, you can make your code more flexible, reusable, and easier to maintain.

What are Design Patterns?



- Design pattern is a problem & solution in context - solution/strategy reuse
- Goals:
  - To support reuse, of
    - Successful designs
    - Existing code (though less important)
  - To facilitate software evolution
    - Add new features easily, without breaking existing ones
  - Reduce implementation dependencies between elements of software system.

Design Patterns: Origin

Erich Gamma
Richard Hem
Ralph Johnson
John Vlissides
(gang of four)

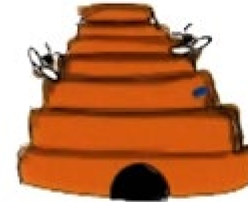
Book “Design Patterns: Elements of Reusable OO Software”

Patterns Catalogue

Fundamental Patterns
- Delegation pattern
- Interface pattern
- Proxy pattern
- …

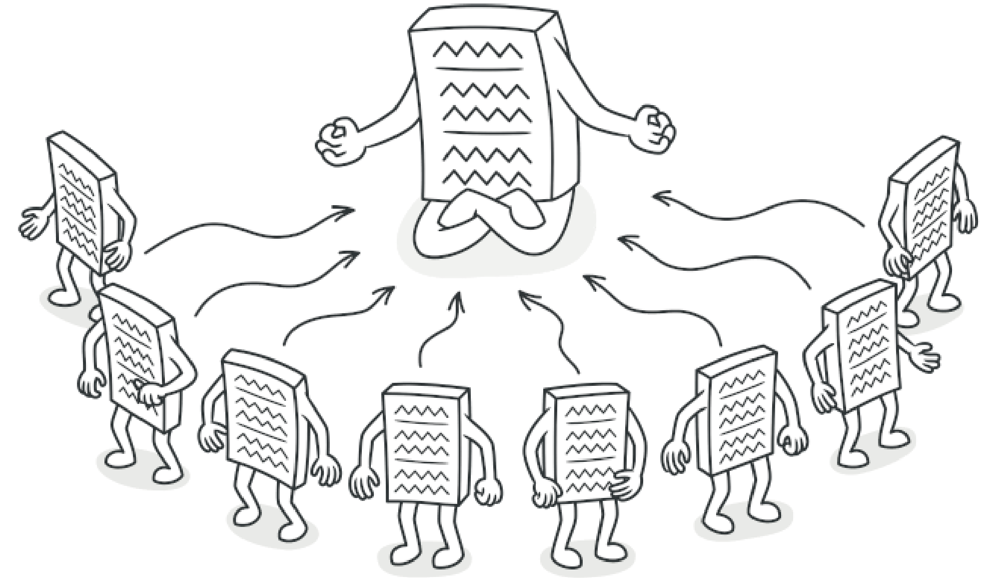
Creational Patterns
- Abstract Factory pattern
- Factory Method pattern
- Lazy Initialization pattern
- Singleton pattern
- …

Structural Patterns
- Adapter pattern
- Bridge pattern
- Decorator pattern
- …



Behavioral Patterns
- Chain of responsibility pattern
- Iterator pattern
- Observer pattern
- State Pattern
- Strategy pattern
- Visitor pattern
- …

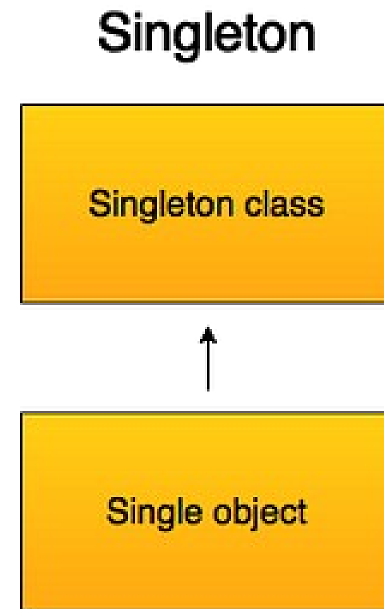
Concurrency Patterns
- Active object pattern
- Monitor object pattern
- Thread pool pattern
- …

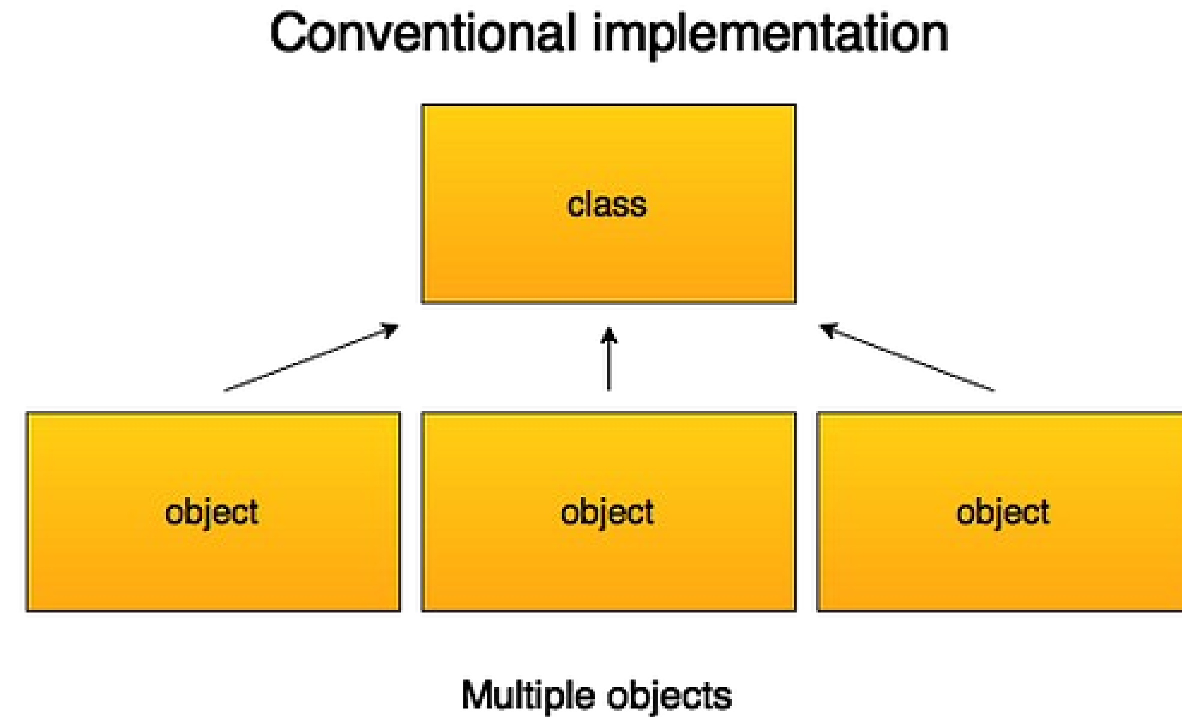


Singleton Design Pattern

Singleton Pattern

- A creational design pattern
- It is used to **ensure that only one instance of a particular class ever gets created** and that **there is just one (global) way to gain access to that instance**





https://medium.com/codex/guide-implementation-of-design-patterns-in-java-821611f15f64

Singleton Class - Structure

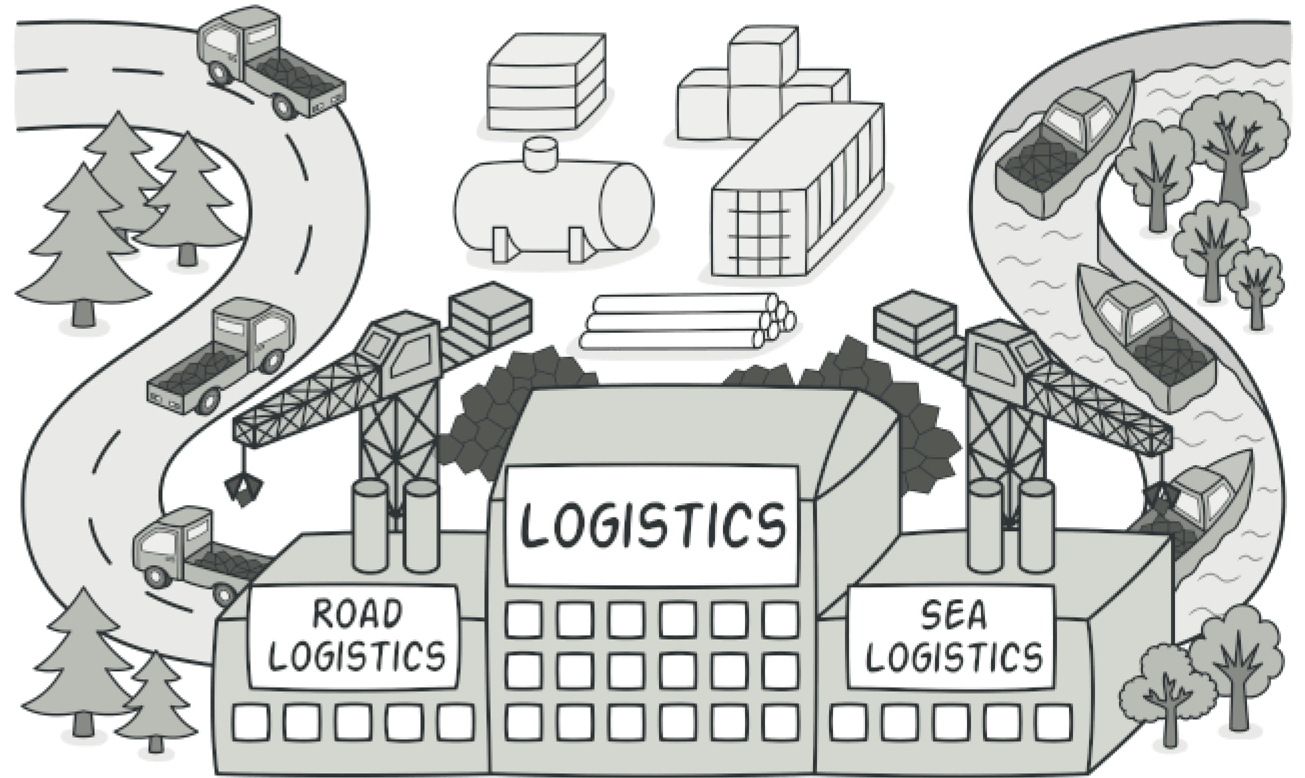
```
1   public class Singleton {
2
3       private static Singleton uniqueInstance;
4
5       private Singleton() {}
6
7       public static Singleton getInstance() {
8           if (uniqueInstance == null) {
9               uniqueInstance = new Singleton();
10          }
11          return uniqueInstance;
12      }
13  }
```

- **private constructor** (to prevent other classes in creating new instance)
- **private static instance variable** (to store one instance)
- **public static method** to gain access to instance
  - creates object if needed; returns it

Real World Examples of Singleton Class

- **Logging systems**: Singleton logger class ensures that all log entries are written to a single instance of the logger. It provides a central point of access for logging information from different parts of the application.
- **Database connections**: In applications that require database access, a Singleton database connection class can be used to manage a single connection instance throughout the system. This ensures that multiple connections are not unnecessarily established, improving performance and resource utilization.
- **Configuration managers**: Singleton pattern is often utilized in configuration management systems where global access to configuration settings is needed. A Singleton configuration manager class provides access to configuration parameters and ensures that the settings remain consistent across the application.

Factory Method Pattern



FACTORY METHOD

loosens the coupling of a given code by separating the product's construction code from the code that uses this product

creational design pattern

You have a burger restaurant, and you need to create a delivery application that delivers burgers

```java
public class Restaurant {
    public ???? orderBurger(String request) {
        if ("BEEF".equals(request)) {
            BeefBurger burger = new BeefBurger();
            burger.prepare();
            return burger;
        } else if ("VEGGIE".equals(request)) {
            VeggieBurger burger = new VeggieBurger();
            burger.prepare();
            return burger;
        }
    }
}
```

BeefBurger	VeggieBurger
- productId : int - angus : boolean - addOns : String	- productId : int - combo : boolean - addOns : String
+ prepare()	+ prepare()

Not closed for modification !

We need 1 interface/abstract class that creates objects of a type and creator subclasses that instantiate specific objects.

```
public abstract class Restaurant {
    public Burger orderBurger() {
        Burger burger = createBurger();
        burger.prepare();
        return burger;
    }

    public abstract Burger createBurger();
}
```

Factory method

```
public class BeefBurgerRestaurant extends Restaurant {
    @Override
    public Burger createBurger() {
        return new BeefBurger();
    }
}
```

```
public class VeggieBurgerRestaurant extends Restaurant {
    @Override
    public Burger createBurger() {
        return new VeggieBurger();
    }
}
```

```
public interface Burger {
    void prepare();
}
```

```
public class BeefBurger
        implements Burger {

    @Override
    void prepare() {
        // prepare beef
        // burger code
    }

}
```

```
public class VeggieBurger
        implements Burger {

    @Override
    void prepare() {
        // prepare veggie
        // burger code
    }

}
```

Users of the restaurant can now directly (dynamically) invoke the concrete restaurant class implementation they need and the correct prepared burger will be returned to them.

```java
public abstract class Restaurant {
    public Burger orderBurger() {
        Burger burger = createBurger();
        burger.prepare();
        return burger;
    }

    public abstract Burger createBurger();
}
```

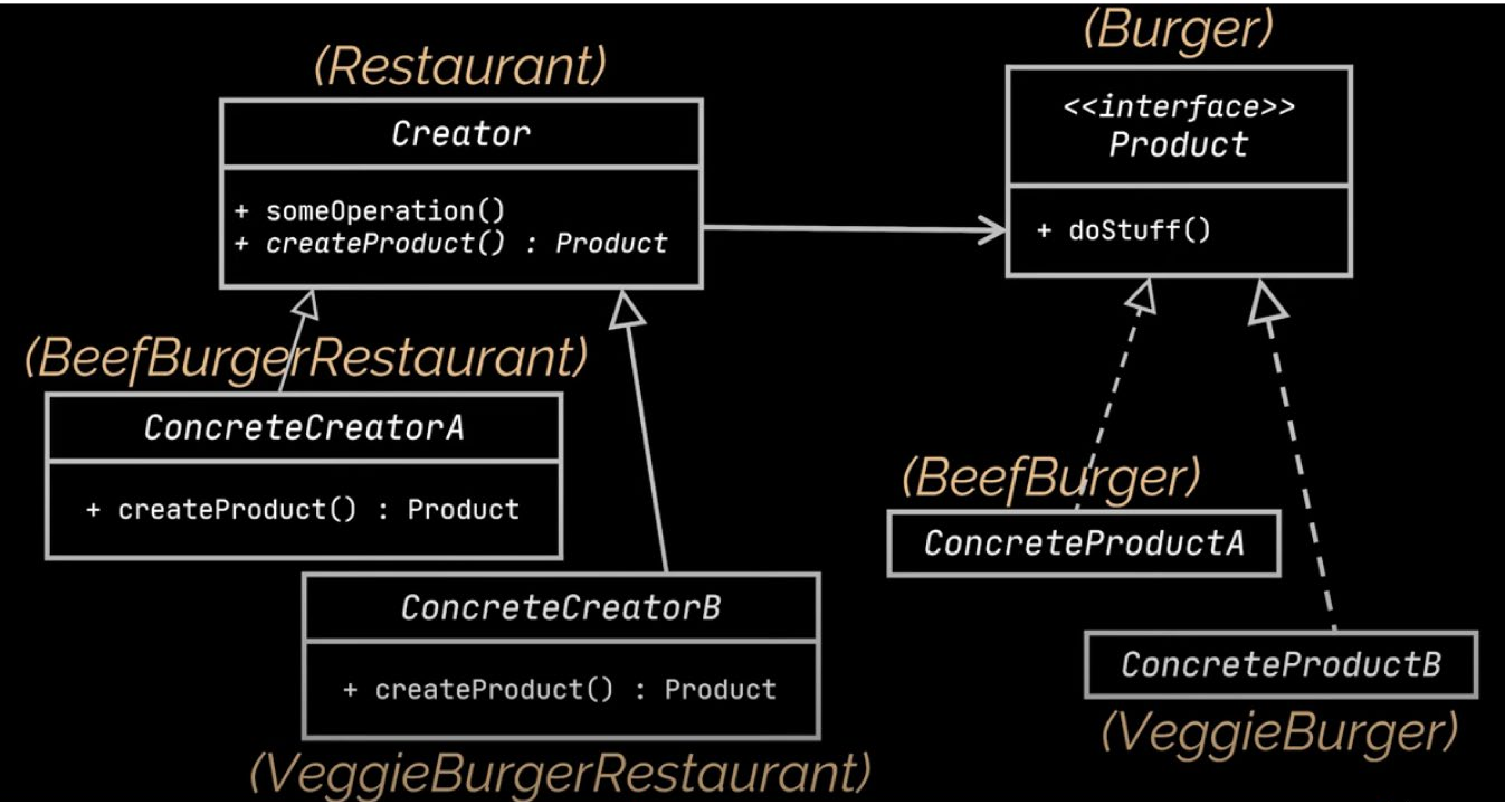
```java
public static void main(String[] args) {

    Restaurant beefResto = new BeefBurgerRestaurant();
    Burger beefBurger = beefResto.orderBurger();

    Restaurant veggieResto = new VeggieBurgerRestaurant();
    Burger veggieBurger = veggieResto.orderBurger();

}
```

Factory Method Pattern



When to use a factory method pattern

- When you don’t know ahead of time what class object you need to instantiate OR there is some logic associated to object instantiation
- When all of the potential classes are in the same subclass hierarchy
- To centralize class selection code
- To encapsulate object creation

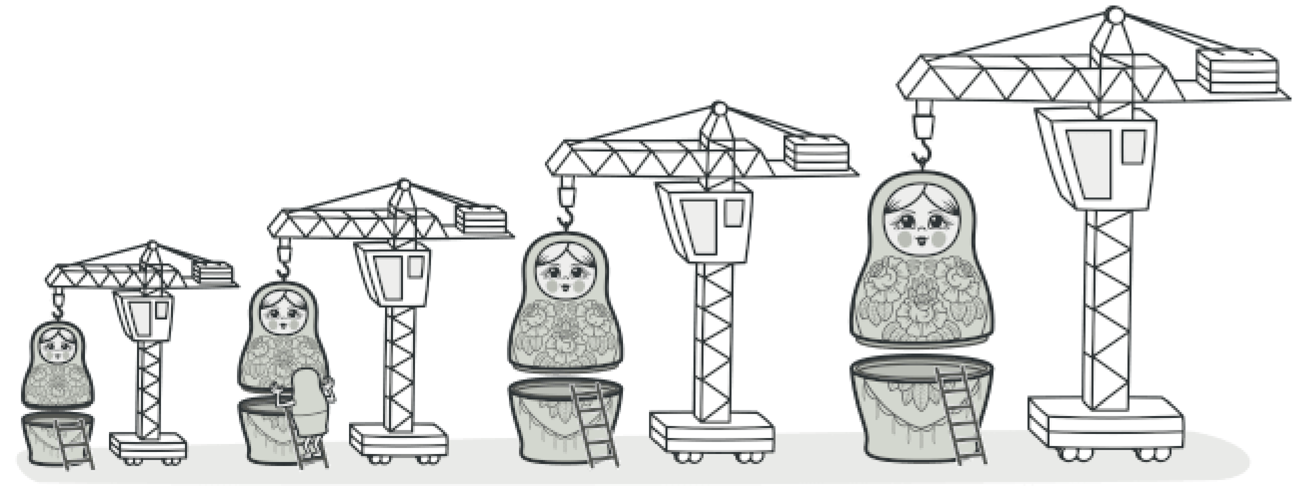
Factory Method

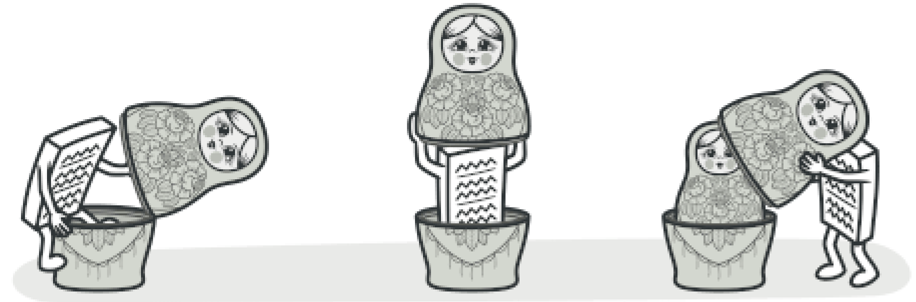
centralizes the product creation code in one place in the program

use it if you have no idea of the exact types of the objects your code will work with

allows introducing new products without breaking existing code

makes it easy to extend the product construction code independently from the rest of the application





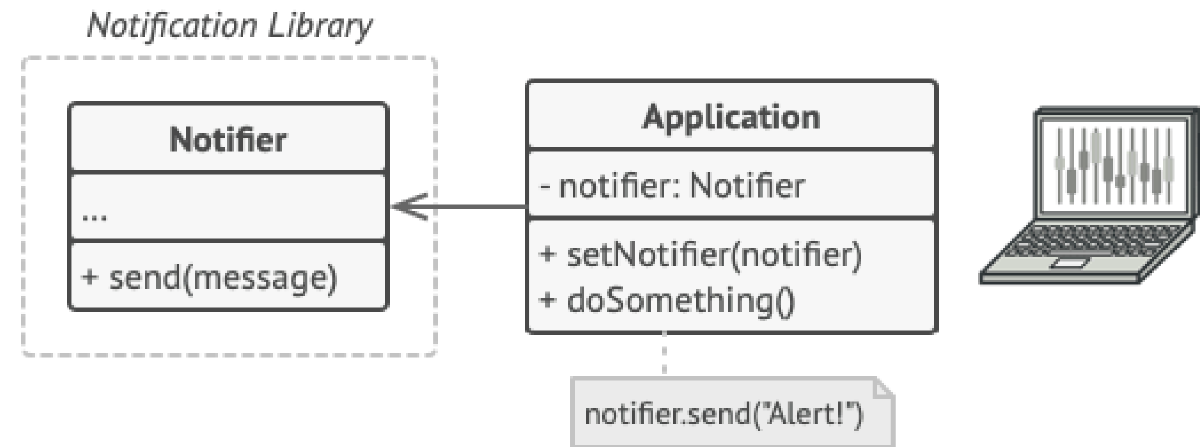
Decorator Pattern

Decorator Pattern

- Decorator is a **structural** design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.
- These new behaviors are added to the object dynamically using wrapping.
- Wrapping is just a fancy way of saying "delegation" but with the added twist that the delegator and the delegate both implement the same interface

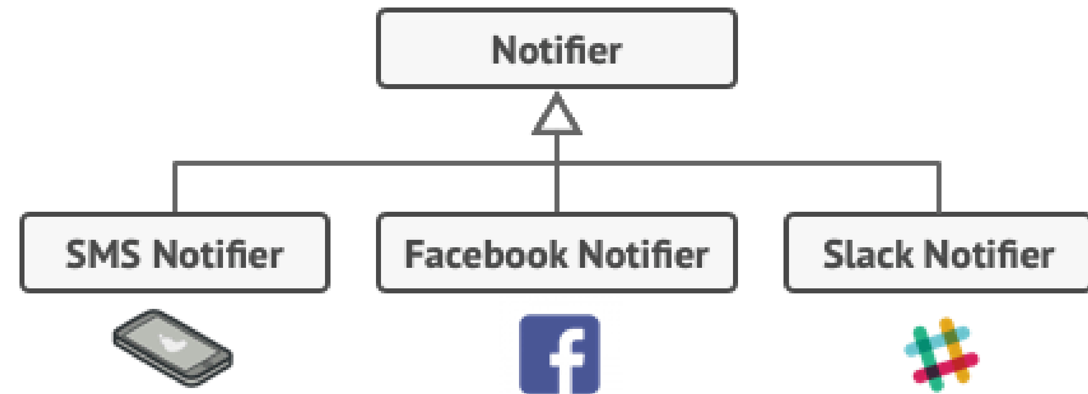
Problem

- Imagine that you're working on a notification library which lets other programs notify their users about important events.
- The initial version of the library was based on the Notifier class that had only a few fields, a constructor and a single send method.
- The method could accept a message argument from a client and send the message to a list of emails that were passed to the notifier via its constructor. A third-party app which acted as a client was supposed to create and configure the notifier object once, and then use it each time something important happened.



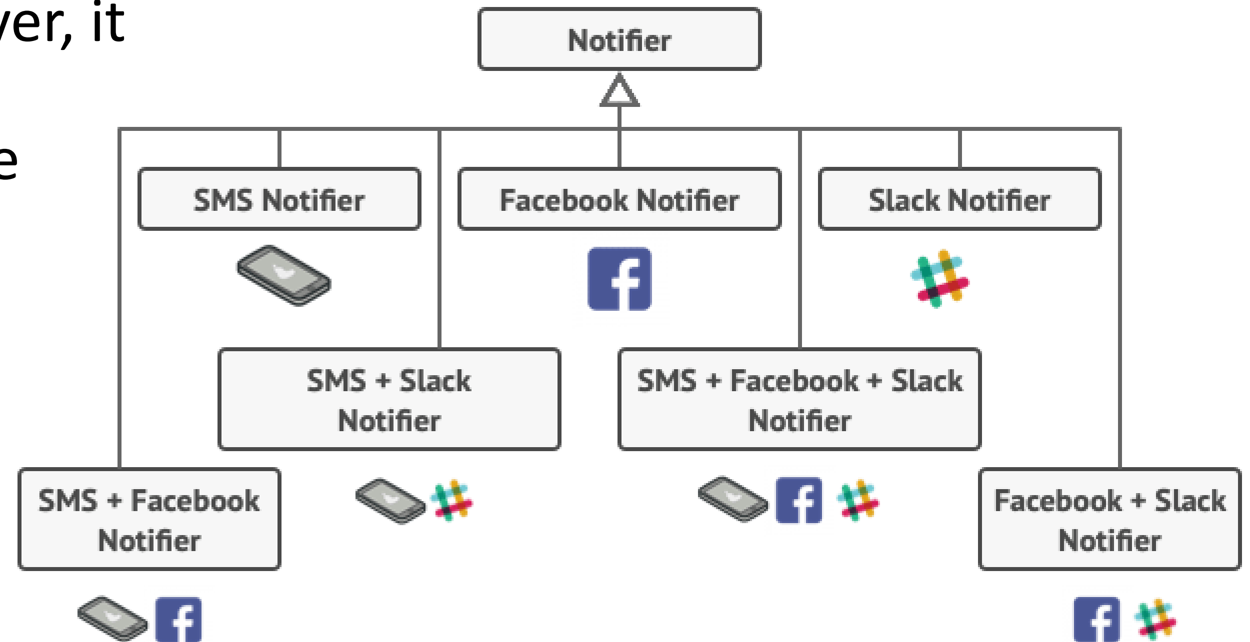
Problem

- You realize that users of the library expect more than just email notifications. Many of them would like to receive an SMS about critical issues. Others would like to be notified on Facebook and, of course, the corporate users would love to get Slack notifications.
- You extended the Notifier class and put the additional notification methods into new subclasses. Now the client was supposed to instantiate the desired notification class and use it for all further notifications.
- “Why can’t you use several notification types at once? If your house is on fire, you’d probably want to be informed through every channel.”



Problem

- You tried to address that problem by creating special subclasses which combined several notification methods within one class. However, it quickly became apparent that this approach would bloat the code immensely, not only the library code but the client code as well.
- You have to find some other way to structure notifications classes



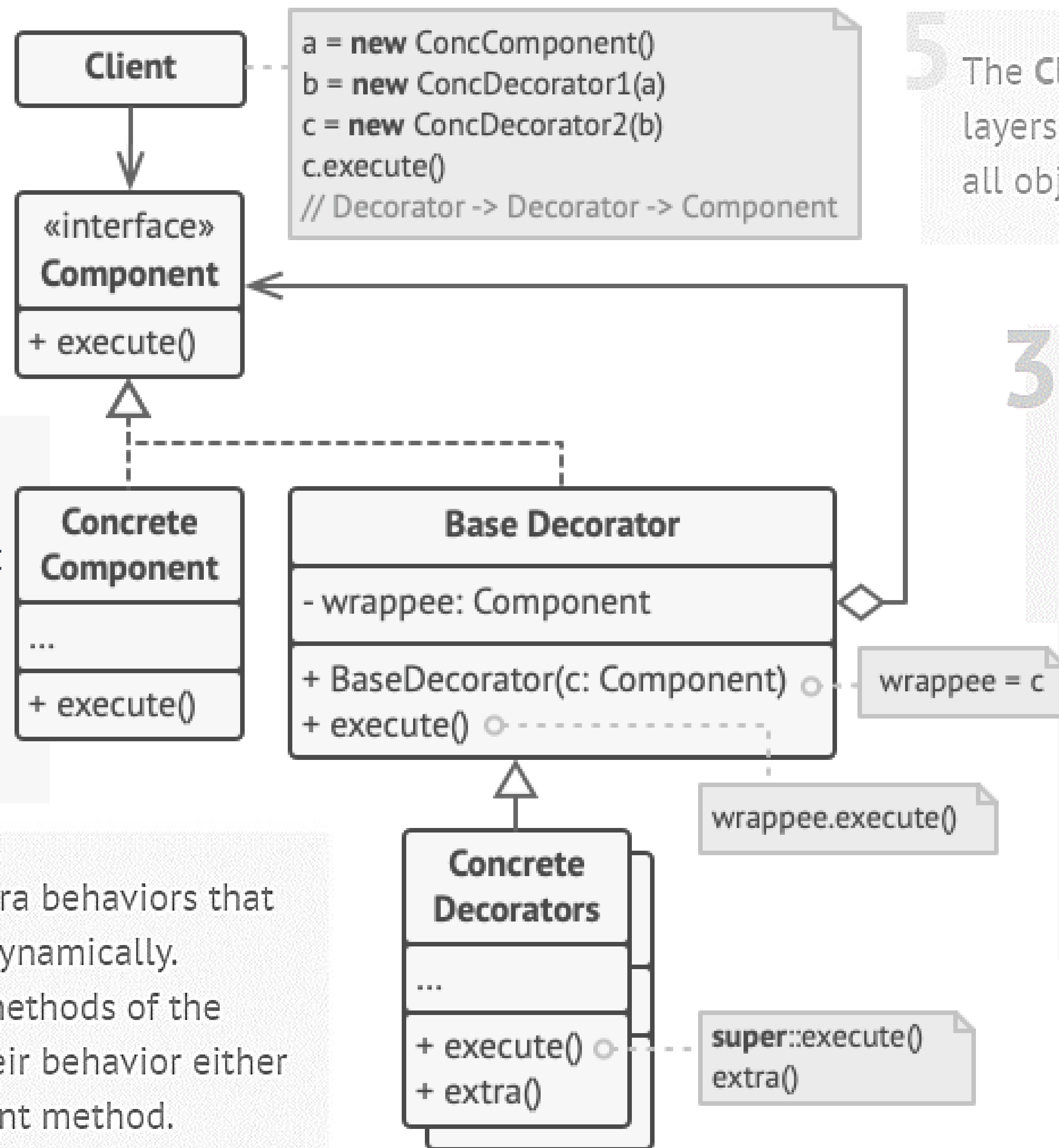
Problem with Inheritance

- Extending a class is the first thing that comes to mind when you need to alter an object's behavior. However, inheritance has several serious caveats that you need to be aware of.
  - **Inheritance is static**. You can't alter the behavior of an existing object at runtime. You can only replace the whole object with another one that's created from a different subclass.
  - **Subclasses can have just one parent class**. In most languages, inheritance doesn't let a class inherit behaviors of multiple classes at the same time.
- One of the ways to overcome these caveats is by using ***Aggregation or Composition*** instead of Inheritance.
  - key principle behind many design patterns, including Decorator.

Decorator Pattern

1 **Component**: This is an interface or abstract class that defines the common behavior of the objects that can be decorated. Concrete components and decorators will implement or extend this interface.

2 **Concrete Component:** Class that implements the Component interface or extends the abstract Component class. It represents the base object that can be decorated with additional behavior.

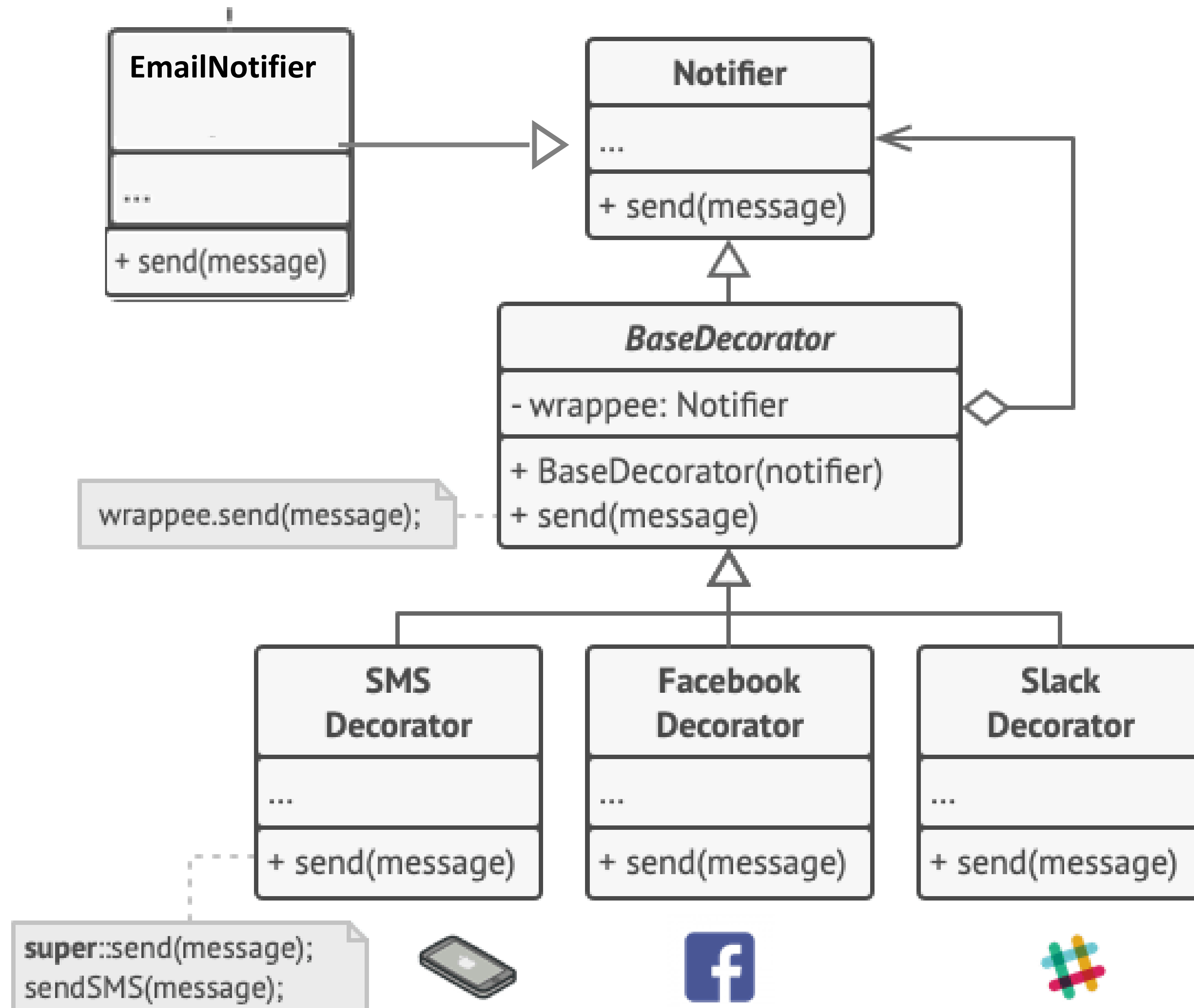


5 The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

3 **Decorator**: Abstract class that also implements the Component interface or extends the abstract Component class. It has a reference to a Component object, which represents the object it decorates. The Decorator class forwards requests to the Component it decorates and can add or modify behavior before or after forwarding the request.

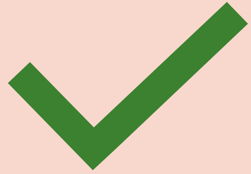
4 **Concrete Decorators** define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.

Solution: Decorator Pattern

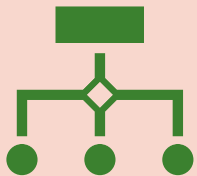


The Strategy Pattern

The Strategy Pattern

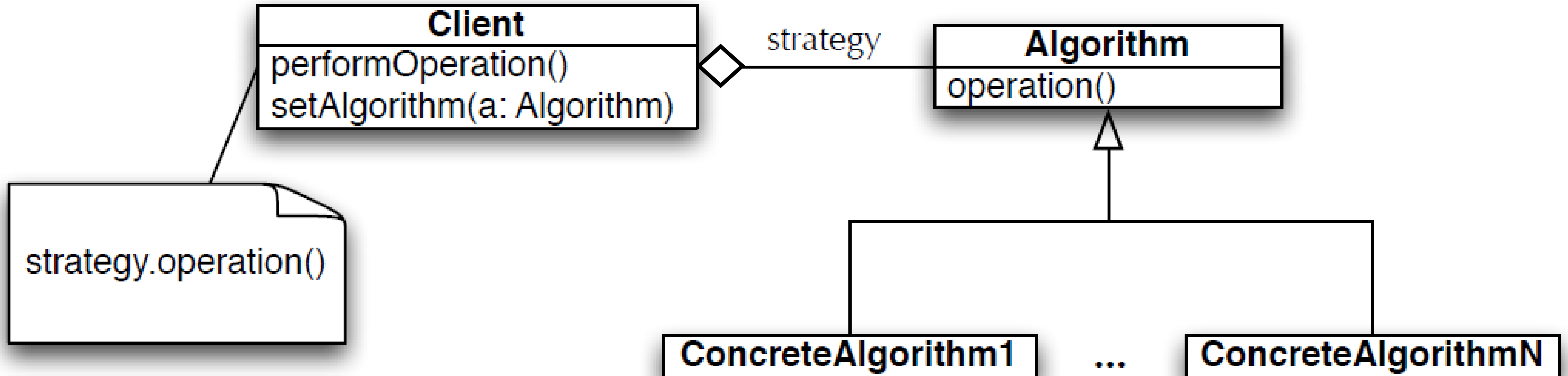


Allows for switching between different behaviors for accomplishing a task

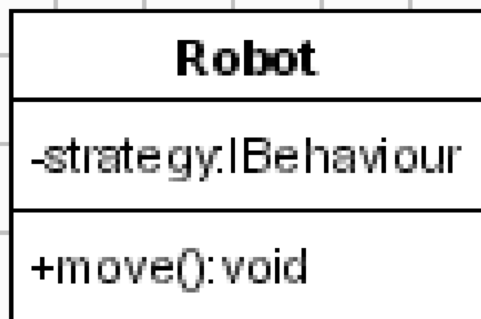


The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

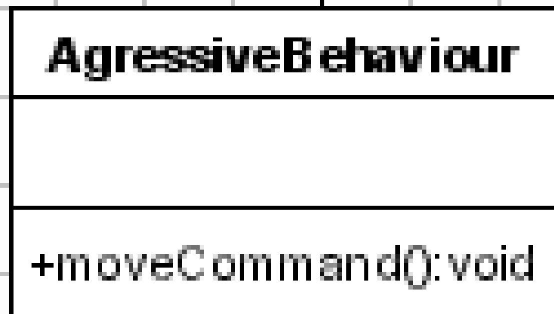
Structure of Strategy

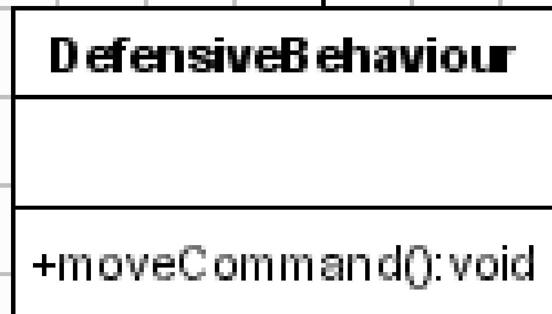


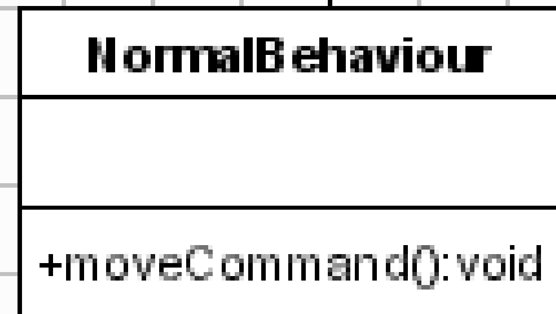
- Algorithm is pulled out of Client. Client only makes use of the public interface of Algorithm and is not tied to concrete subclasses.
- Client can change its behavior by switching among the various concrete algorithms


Robot
-strategy:IBehaviour
+move():void

<< interface >>
IBehaviour
+moveCommand():void


AgressiveBehaviour
+moveCommand():void


DefensiveBehaviour
+moveCommand():void


NormalBehaviour
+moveCommand():void

Strategy Pattern example in Game development

- **Character Movement**: Strategy – Movement. Concrete Strategies – walk, fly, swim, teleport….

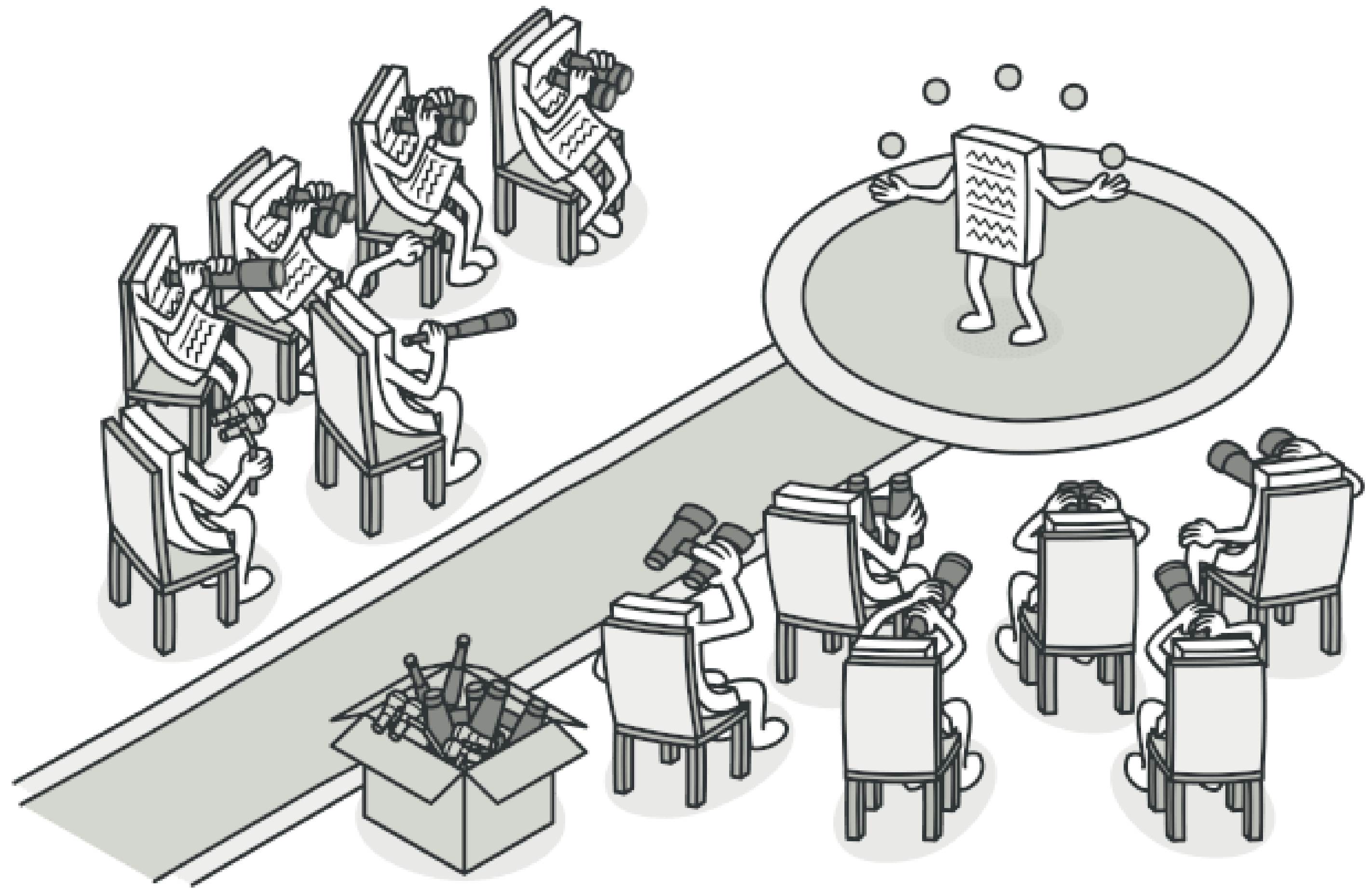
- **Weapon Fight Behavior**: Strategy – Fight. Concrete Strategies – Sword, Bow, Axe….

- **AI Behavior**: Strategy – Action & Reaction. Concrete Strategies - attack aggressively, attack from range, avoid opponent….

When to use Strategy Pattern?

- The Strategy pattern can be used with classes that do something specific with different strategies
  - E.g., different attack behaviors, different robot functionalities
- It splits the different strategies (by using an interface) from the context
- The code in the context is unchanged if
  - A strategy contains a bug and is fixed
  - A new strategy is added

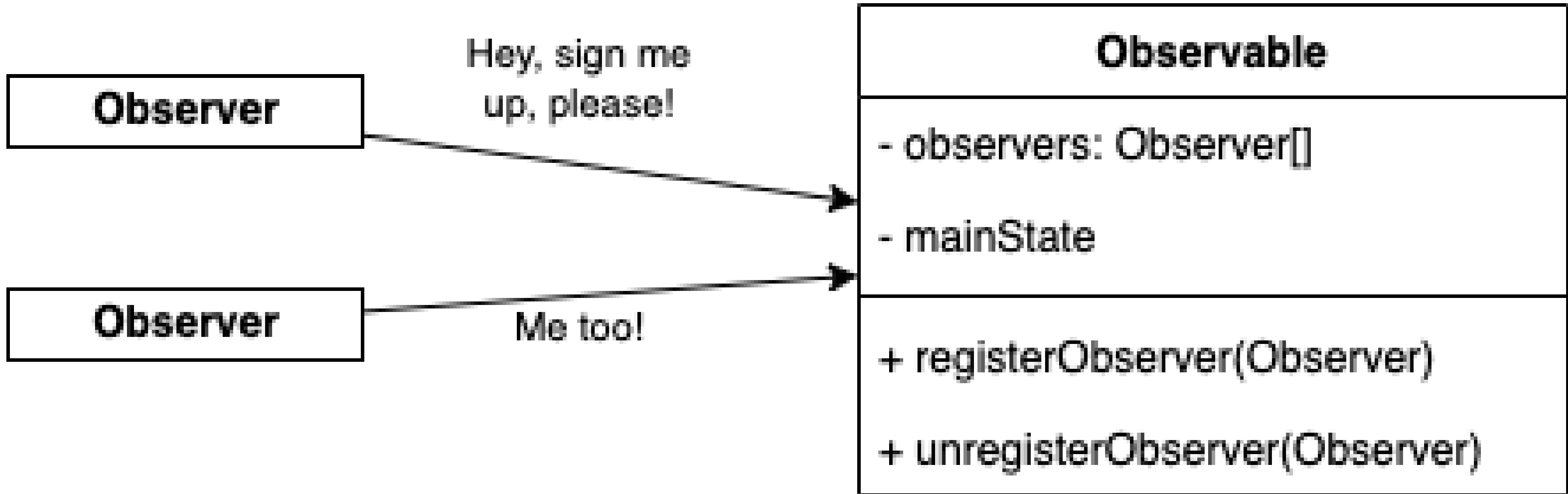
The Observer Pattern



Observer Pattern

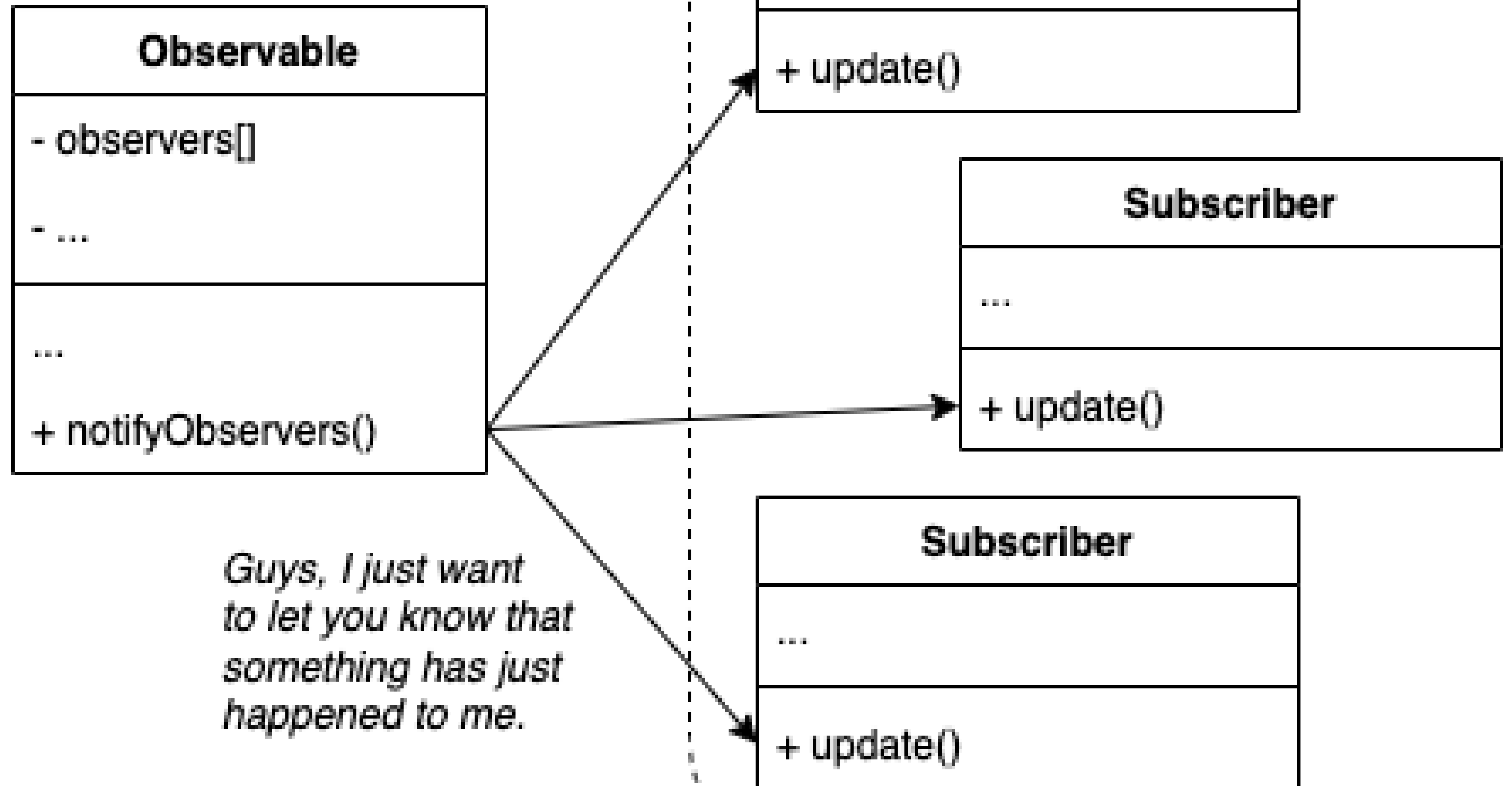
- Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing
- The observer pattern allows objects to keep other objects informed about events occurring within a software system (or across multiple systems).
- It’s dynamic in that an object can choose to receive or not receive notifications at run-time
- Observer happens to be one of the most heavily used patterns in the Java Development Kit

Observer Pattern



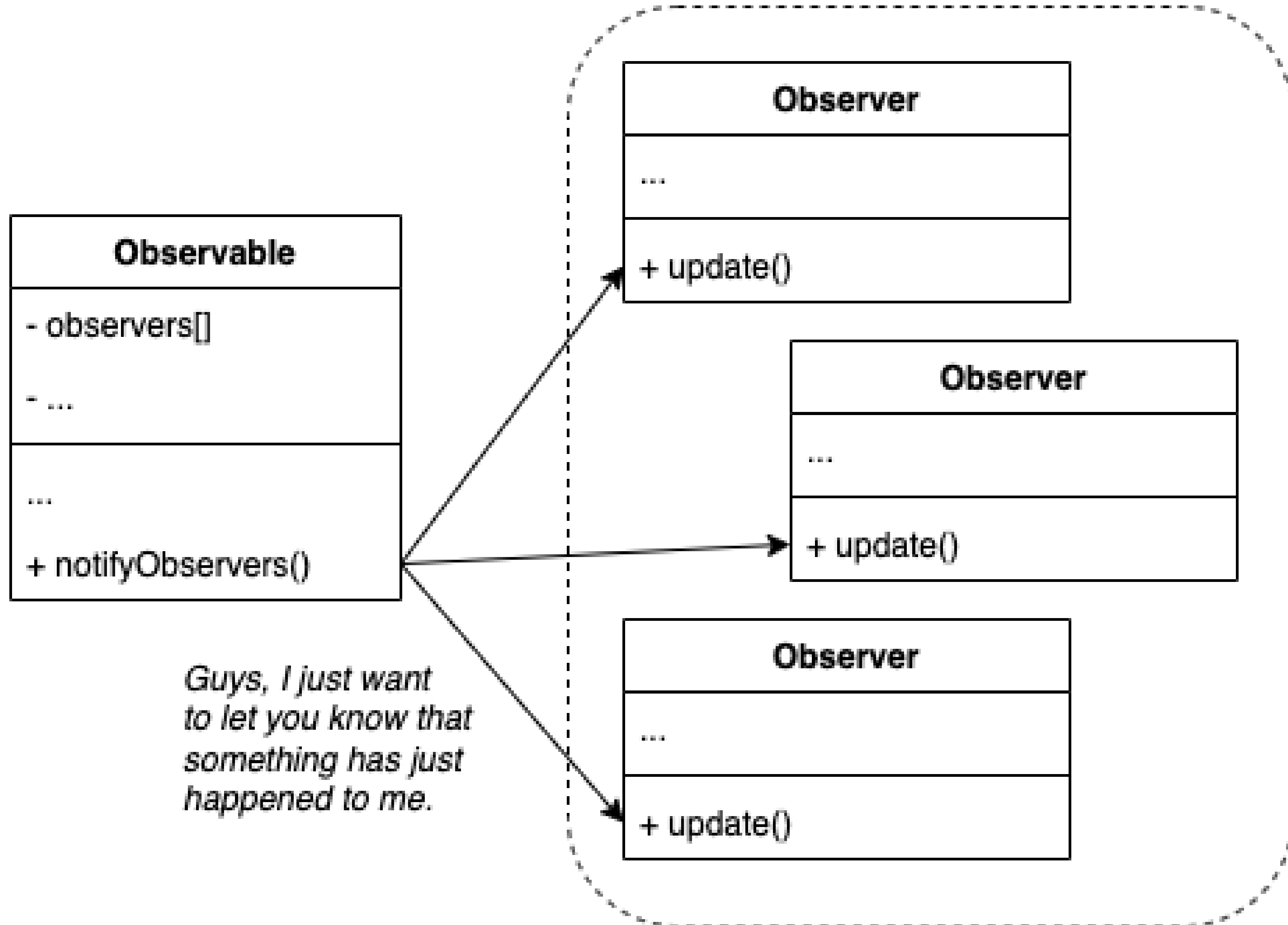
Observable/Publisher consists of 1) an array field for storing a list of references to observer/subscriber objects and 2) several public methods which allow adding observers to and removing them from that list.

Observer Pattern



Observer Pattern

- You wouldn't want to couple the observable to all those observer classes
- It is crucial that all observers implement the same interface, and that the observable communicates with them via that interface

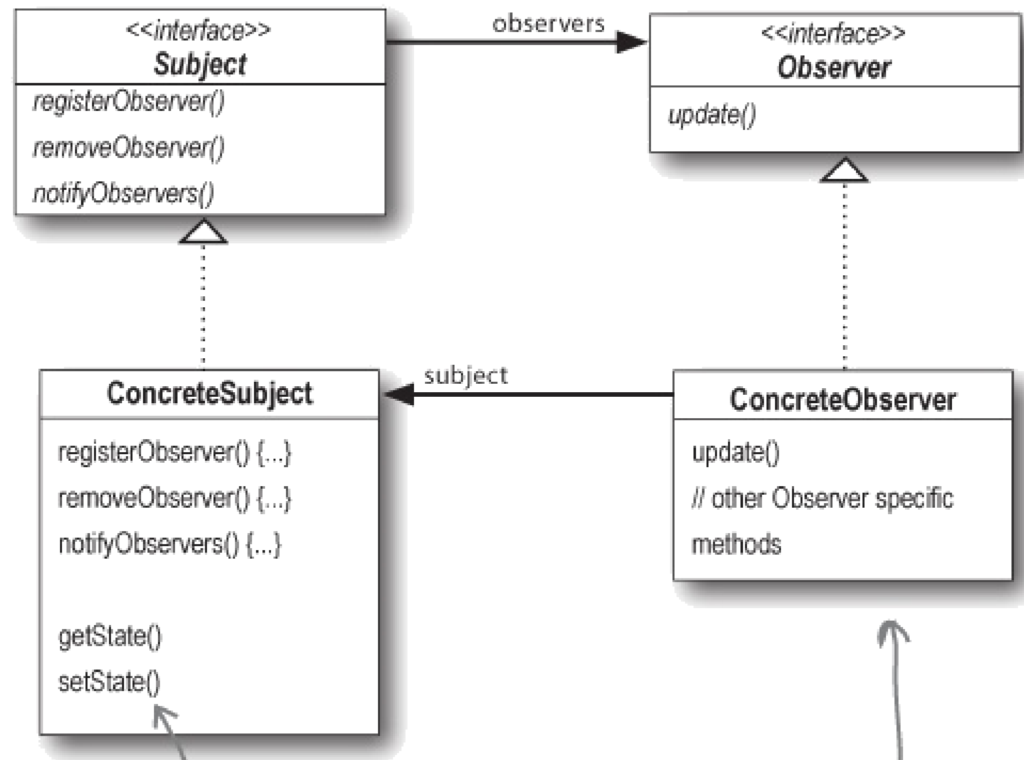


Observer Pattern

Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.

Each subject can have many observers.

All potential observers need to implement the Observer interface. This interface just has one method, update(), that gets called when the Subject's state changes.



A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.

The concrete subject may also have methods for setting and getting its state (more about this later).

Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

Applicability

- When changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically
- When some objects in your app must observe others, but only for a limited time or in specific cases

Choosing a Pattern

Approach

- Understand your design context
- Examine the patterns catalogue
- Identify and study related patterns
- Apply suitable pattern

Pitfalls

- Selecting wrong patterns
- Abusing patterns

You will implement Project 2 incorporating atleast 3 design patterns

AI with design patterns – code maintainability

- E-commerce Project Example - The project simulates an e-commerce order processing system. It includes classes for handling products, orders, payment processing, notifications, and discounts. However, the current implementation is tightly coupled, hard to extend, and lacks modularity due to the absence of several design patterns and presence of design flaws.
- Let’s see the example Demo and how can AI help in detection and refactoring

AI with design patterns – code maintainability

- Inspect the project as a group for 5 mins and list code maintainability issues
- List design patterns that could help make maintainability better