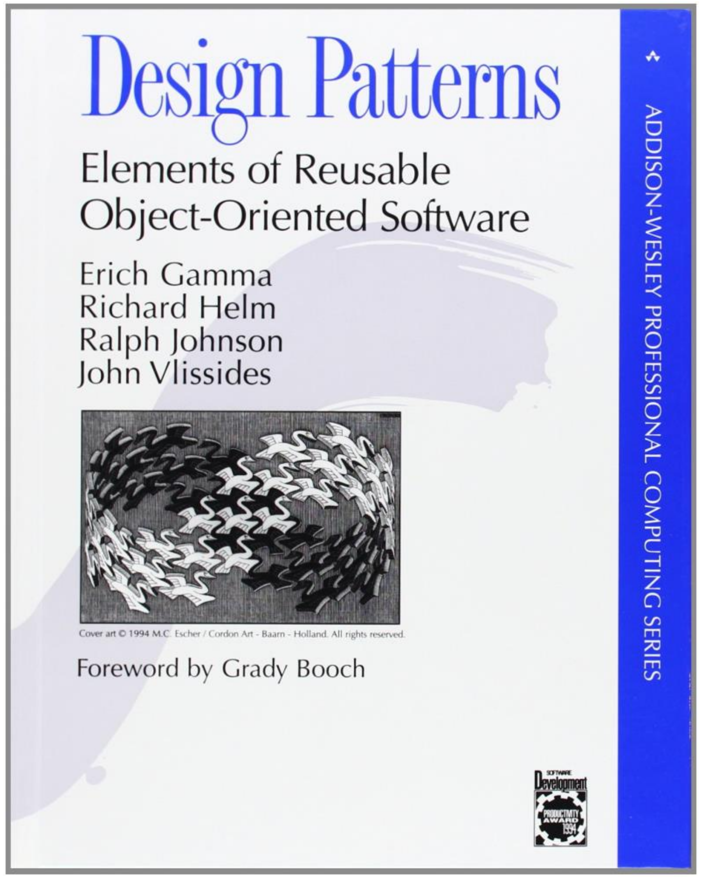
CSE 4513

Lec – 9

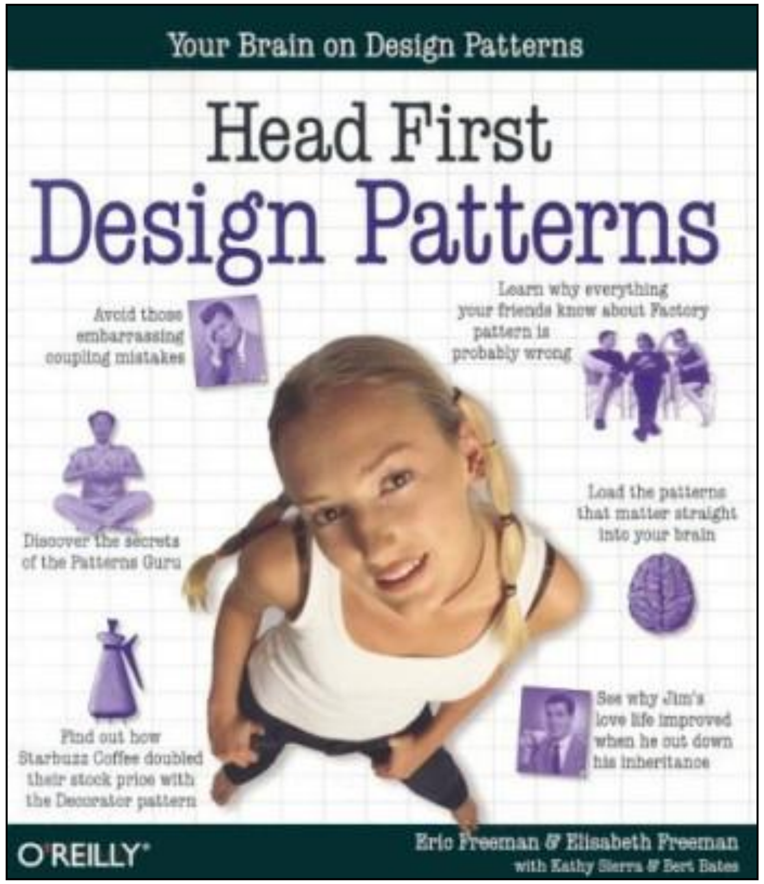
Design Pattern

Someone has already solved your problems

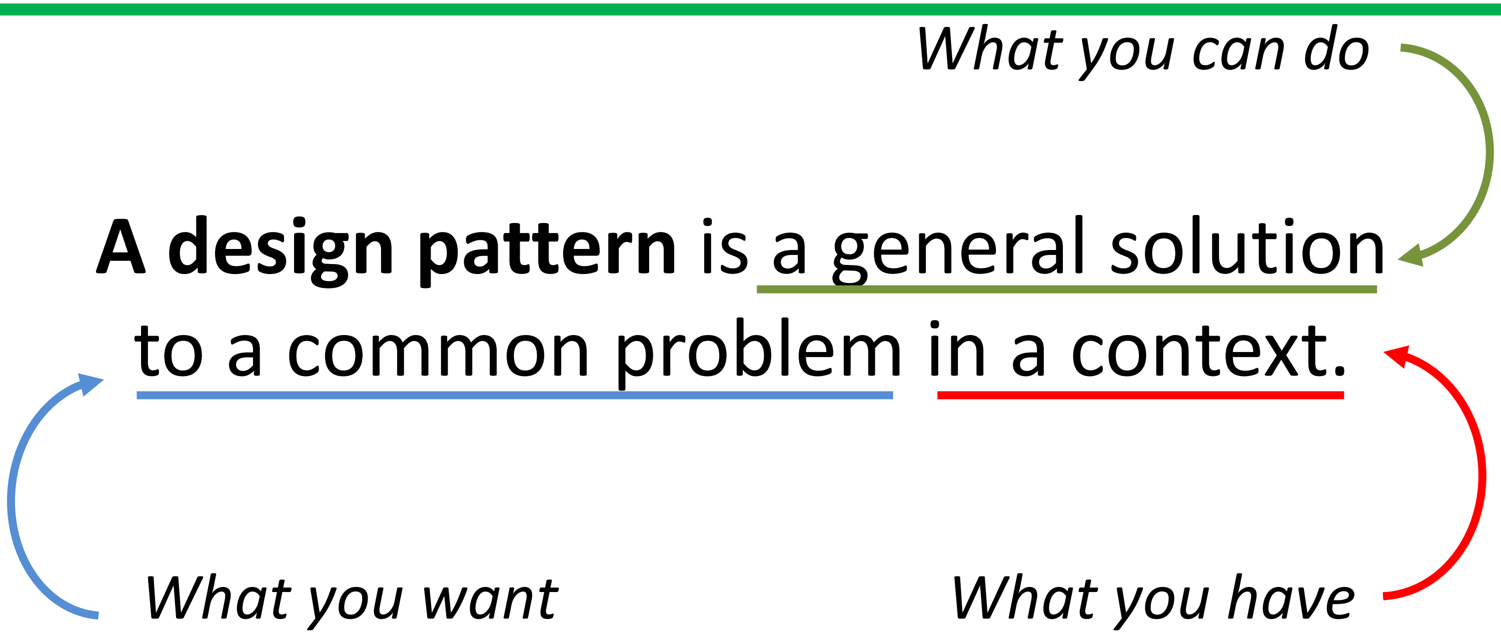
USEFUL BOOKS


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The "Gang of Four" book
1994


Your Brain on Design Patterns
Head First Design Patterns
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates
O'REILLY

Head First Design Patterns Book
2004



A design pattern is a general solution to a common problem in a context.

Each pattern Describes a problem which occurs over and over again in our environment ,and then describes the core of the problem

a design pattern is a general repeatable solution to a commonly occurring problem in software design”

Usage of Design Pattern

Design Patterns have two main usages in software development:

✓ **Common platform for developers**

Design patterns provide a standard terminology and are specific to particular scenario.

✓ **Best Practices**

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.

Elements of a Pattern

- ✓ four elements to describe a pattern
  - The name of the pattern
  - The purpose of the pattern: what problem it solves
  - How to solve the problem
  - The constraints we have to consider in our solution

TYPES OF DESIGN PATTERNS

There are about 26 Patterns currently discovered …
These 26 can be classified into 3 types:

1. **Creational:** These patterns are designed for class instantiation. They can be either class-creation patterns or object-creational patterns.

2. **Structural:** These patterns are designed with regard to a class's structure and composition. The main goal of most of these patterns is to increase the functionality of the class(es) involved, without changing much of its composition.

3. **Behavioral:** These patterns are designed depending on how one class communicates with others.

Design Pattern Index

Creational	Structural	Behavioural
Factory Method	Adapter	Template
Abstract Factory	Bridge	Strategy
Builder	Composite	Command
Singleton	Decorator	State
Multiton	Facade	Visitor
Object pool	Flyweight	Chain of Responsibility
Prototype	Front controller	Interpreter
	Proxy	Observer
		Iterator
		Mediator
		Memento

WARNING

OVERUSE OF DESIGN PATTERNS CAN LEAD TO CODE THAT IS DOWNRIGHT OVER-ENGINEERED.

ALWAYS GO WITH THE SIMPLEST SOLUTION THAT DOES THE JOB AND **INTRODUCE PATTERNS ONLY WHERE THE NEED EMERGES.**

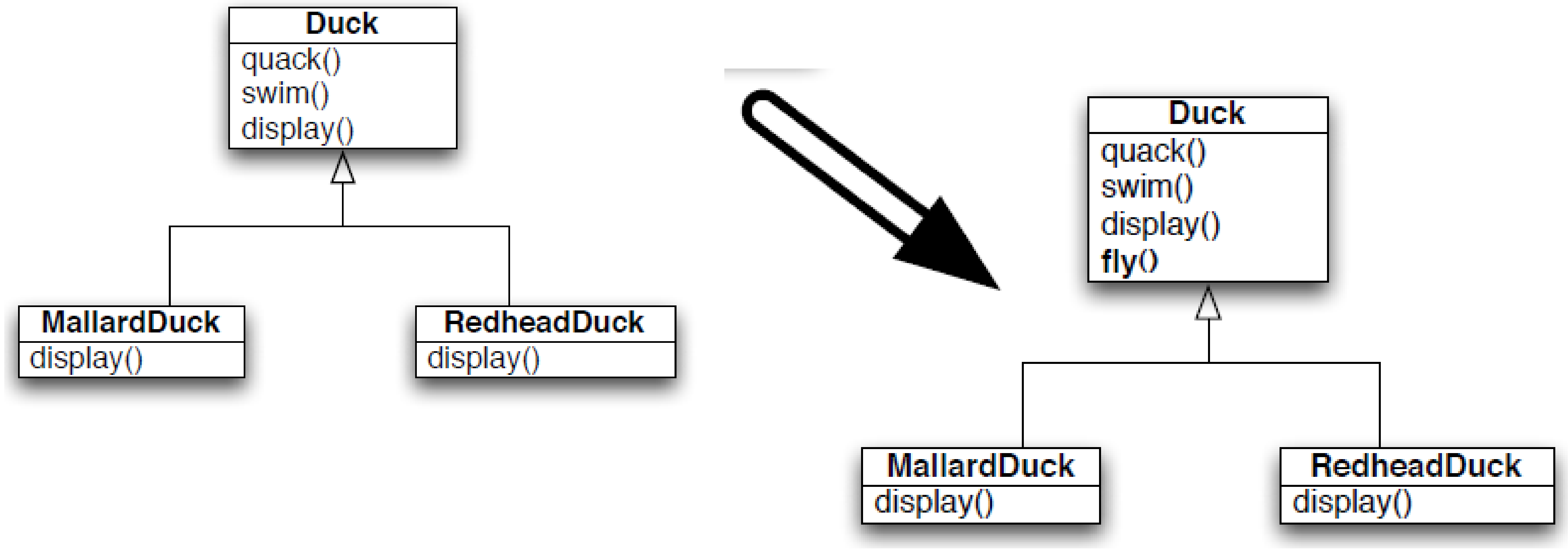
Strategy Pattern

Strategy Pattern

- ✓ it's actually probably the simplest pattern
- ✓ it's about using composition rather than inheritance
- ✓ it's about understanding that inheritance is not intended for code reuse

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

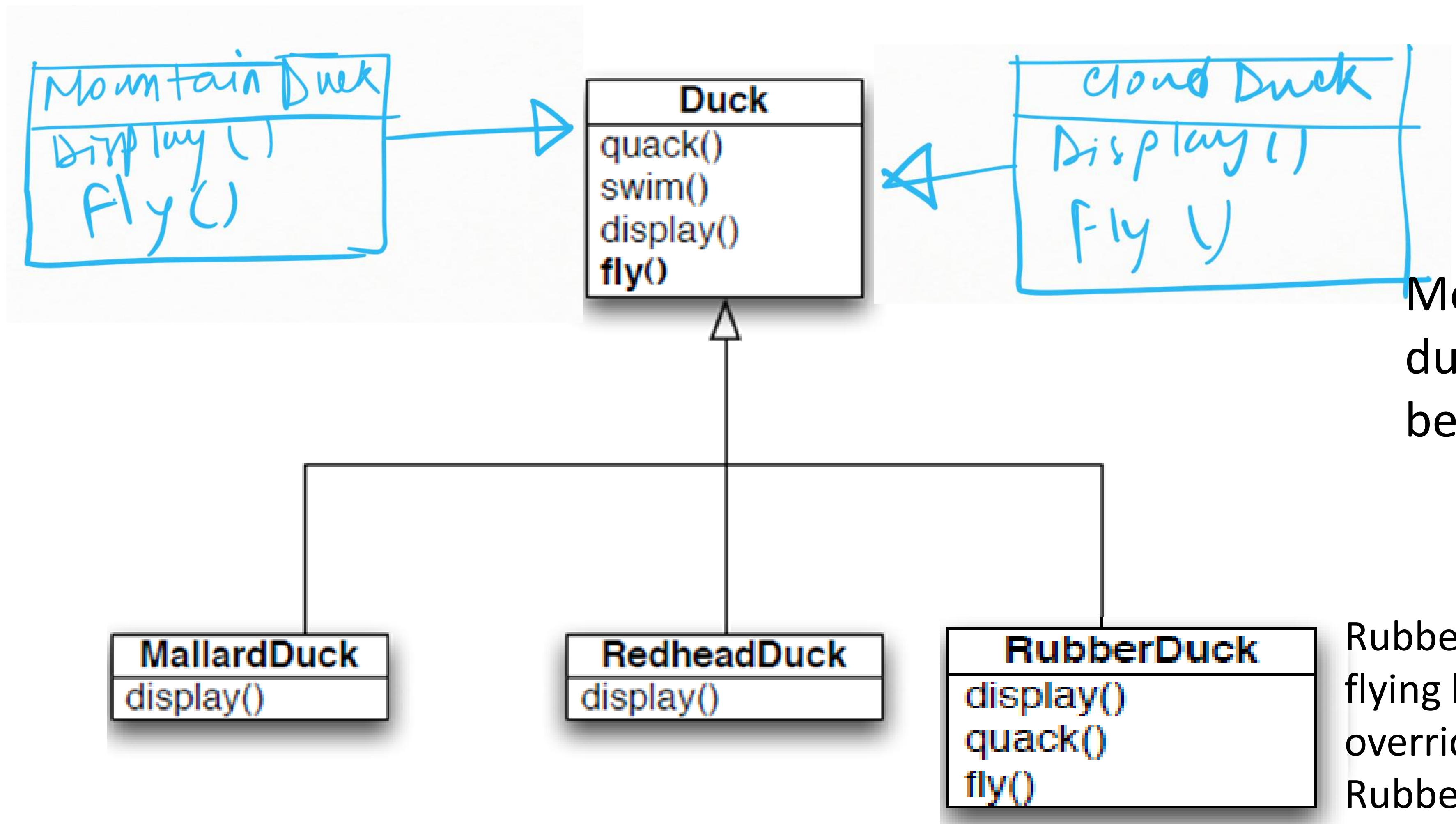
Strategy Pattern



But a request has arrived to allow ducks to also fly.

Strategy Pattern

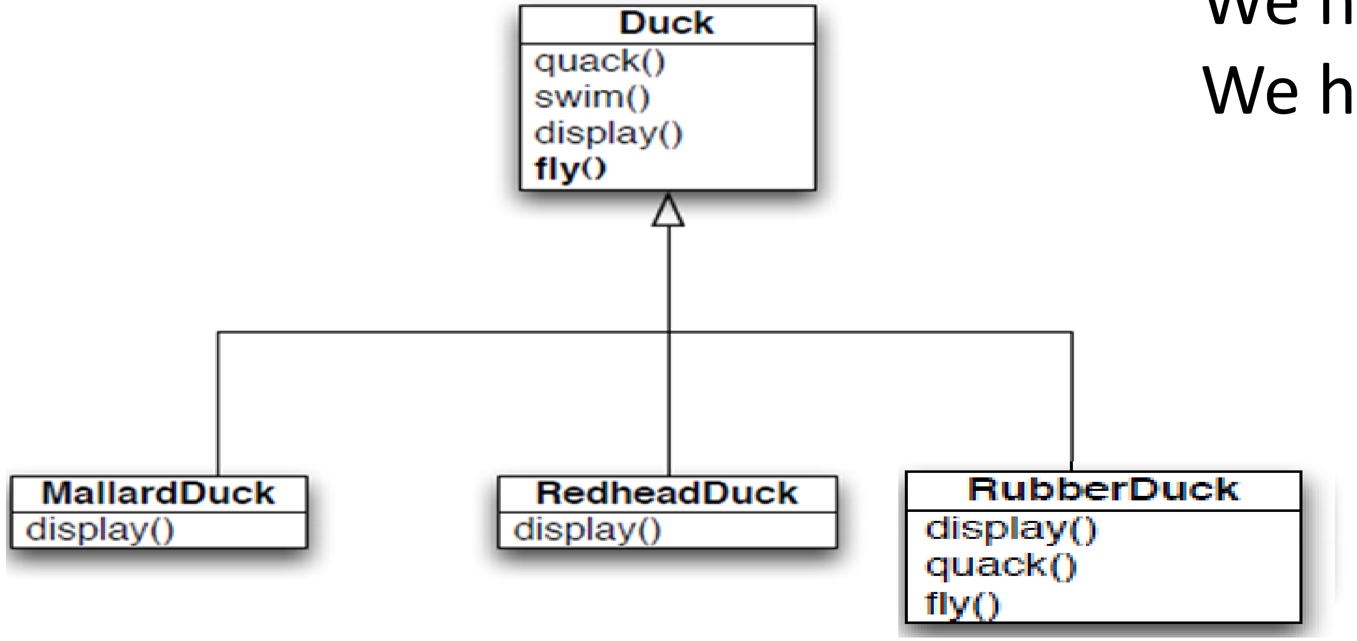
Now we have another duck for example rubber duck which is a fake duck..



Mountain and cloud duck has same flying behavior.

Rubber duck has different flying behavior . We could override fly() in RubberDuck to make it do Nothing..

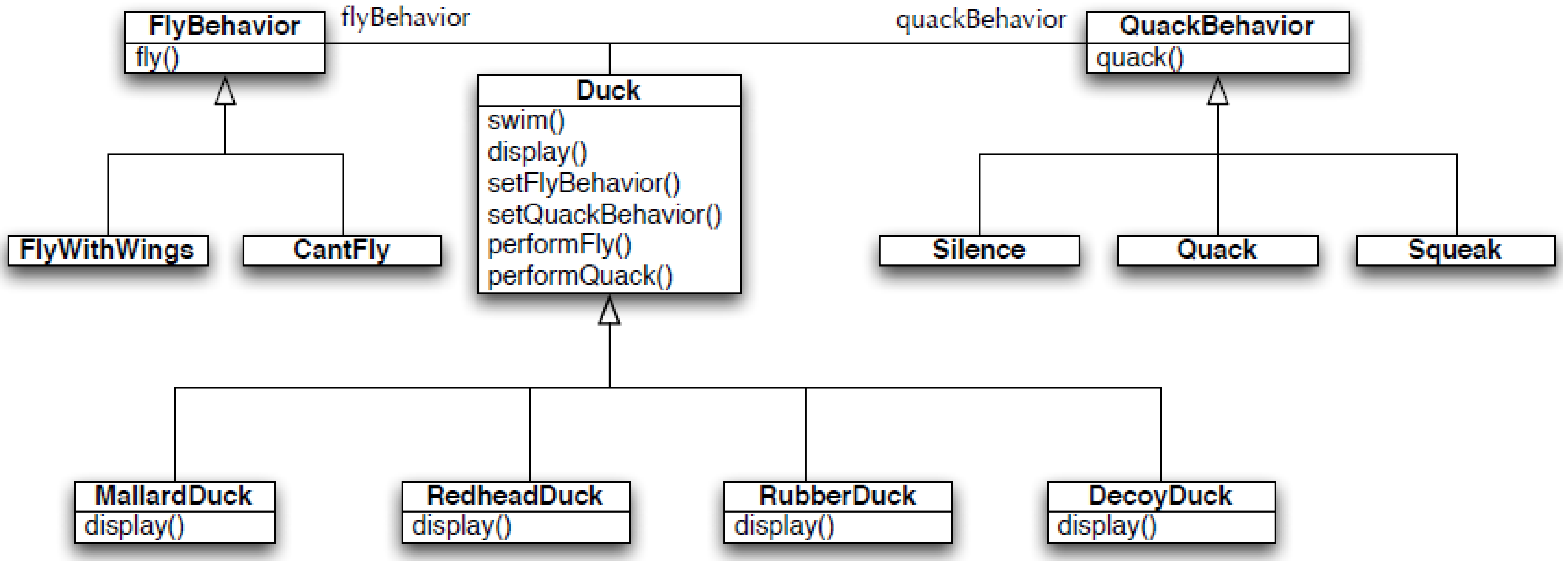
Strategy Pattern



We have algorithm for quacking
We have algorithm for flying…

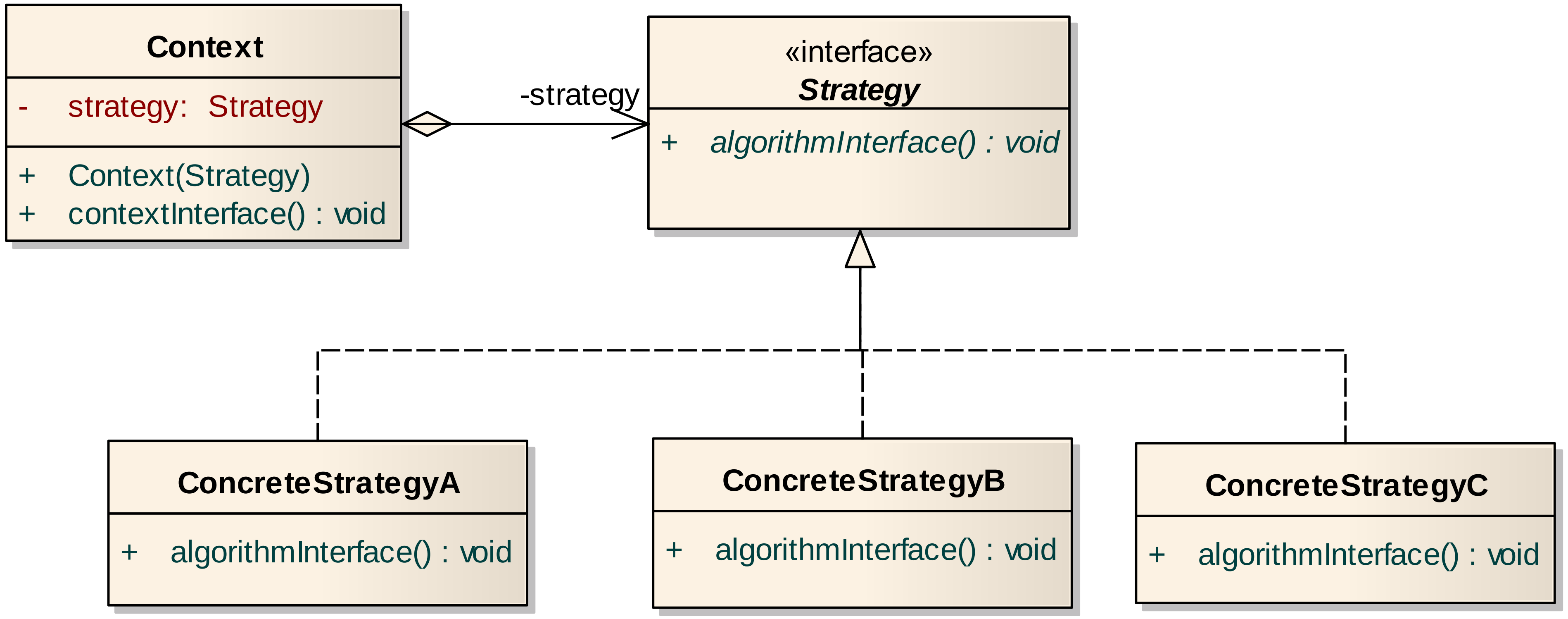
Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Strategy Pattern



FlyBehavior and QuackBehavior define a set of behaviors that provide behavior to Duck. Duck delegates to each set of behaviors and can switch among them dynamically, if needed.

Strategy Pattern



Context
- strategy: Strategy
+ Context(Strategy)
+ contextInterface() : void
-strategy
«interface»
Strategy
+ algorithmInterface() : void
ConcreteStrategyA
+ algorithmInterface() : void
ConcreteStrategyB
+ algorithmInterface() : void
ConcreteStrategyC
+ algorithmInterface() : void

Observer Pattern

Observer Pattern

observer: An object that "watches" the state of another object and takes action when the state changes in some way.

Problem: You have a model object with a complex state, and the state may change throughout the life of your program.
You want to update various other parts of the program when the object's state changes.

Solution: Make the complex model object observable.

observable object: An object that allows observers to examine it (notifies its observers when its state changes).
Permits customizable, extensible event-based behavior for data modeling and graphics.

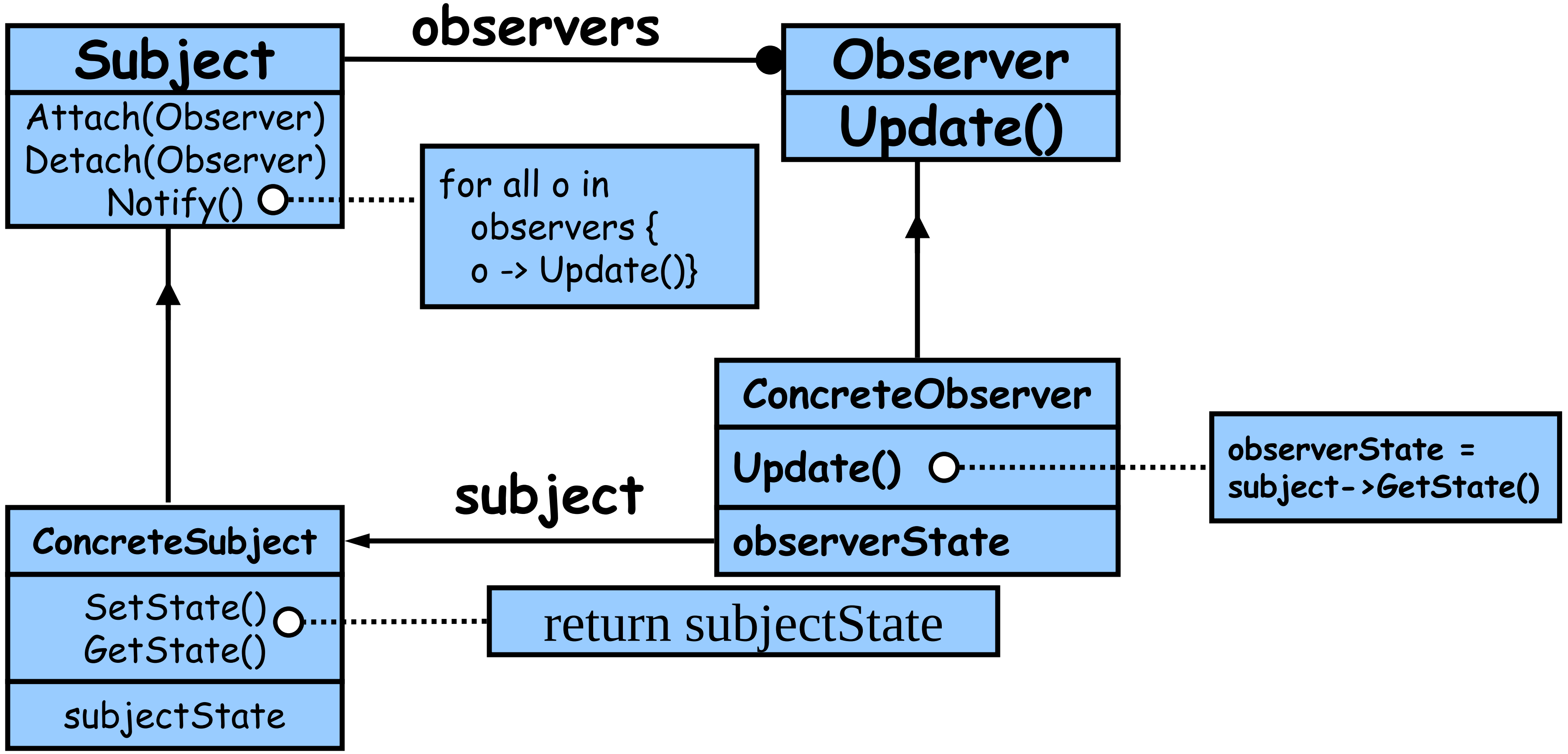
Observer Pattern

- Subject
  - has a list of observers
  - Interfaces for attaching/detaching an observer
- Observer
  - An updating interface for objects that gets notified of changes in a subject
- ConcreteSubject
  - Stores “state of interest” to observers
  - Sends notification when state changes
- ConcreteObserver
  - Implements updating interface

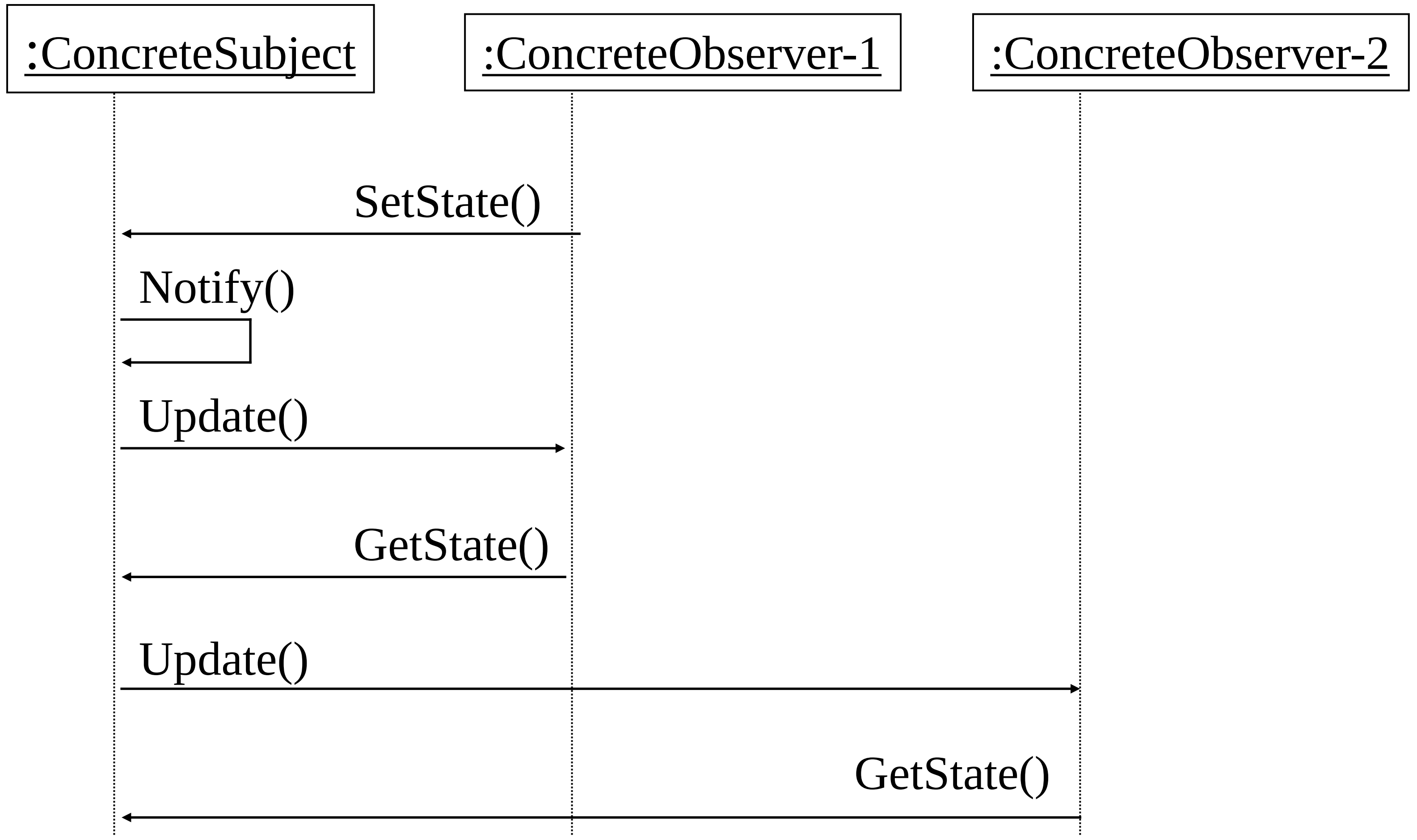
OBSERVER PATTERN

Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Observer Pattern



Observer Pattern



Decorator Pattern

DECORATOR PATTERN

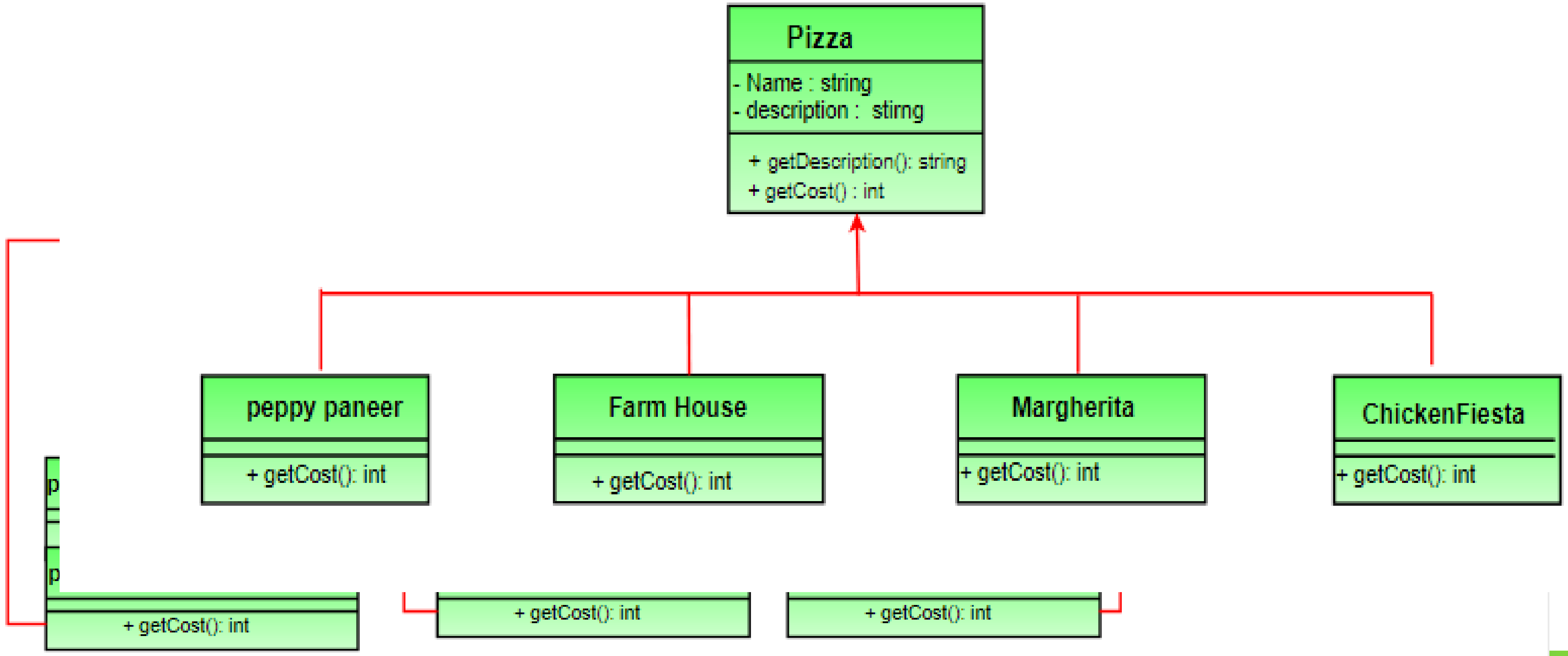
- ✓ The decorator pattern is a pattern where we write wrapper code to let us extend the core code.
- ✓ We just keep wrapping objects around objects that we want to use so that we keep extending the capabilities of the existing objects by defining new objects that have the capabilities of the existing object.

DECORATOR PATTERN

- ✓ how do we accommodate changes in the below classes so that customer can choose pizza with toppings and we get the total cost of pizza and toppings the customer chooses.

Option 1

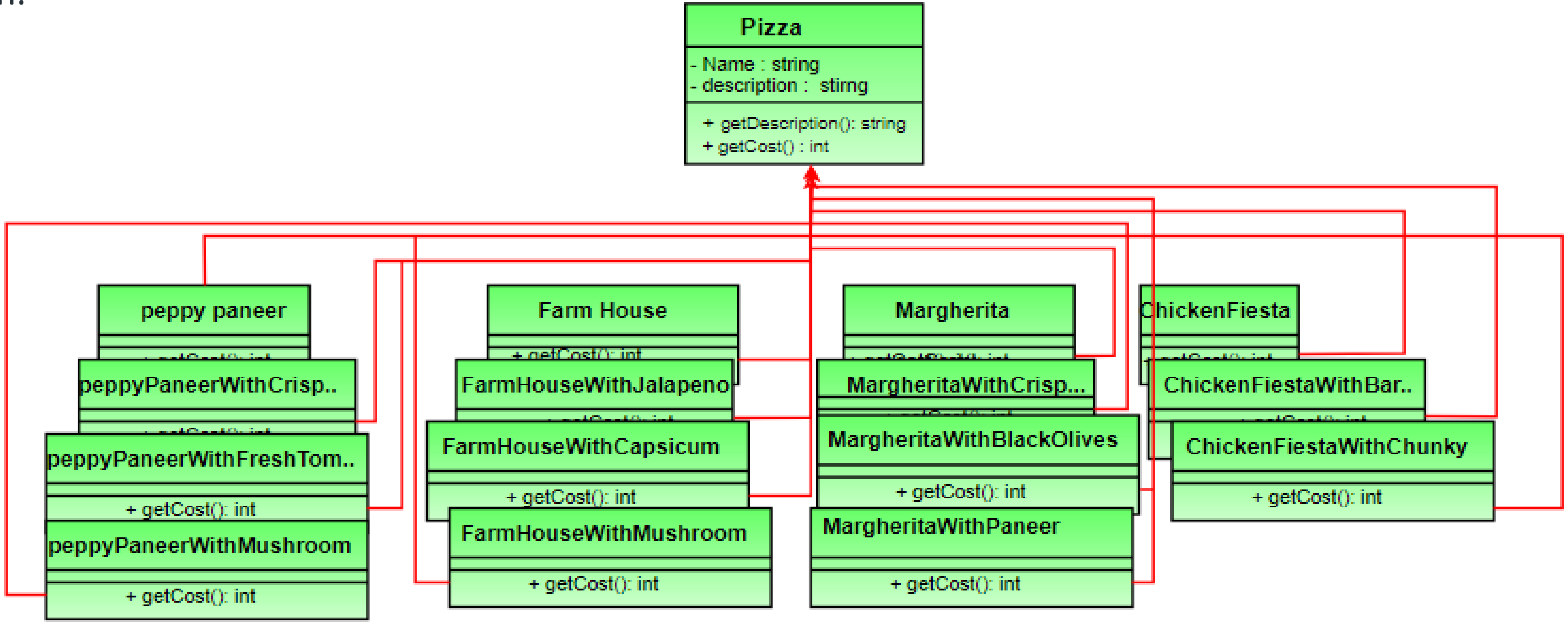
Create a new subclass for every topping with a pizza.



Decorator Pattern

Option 1

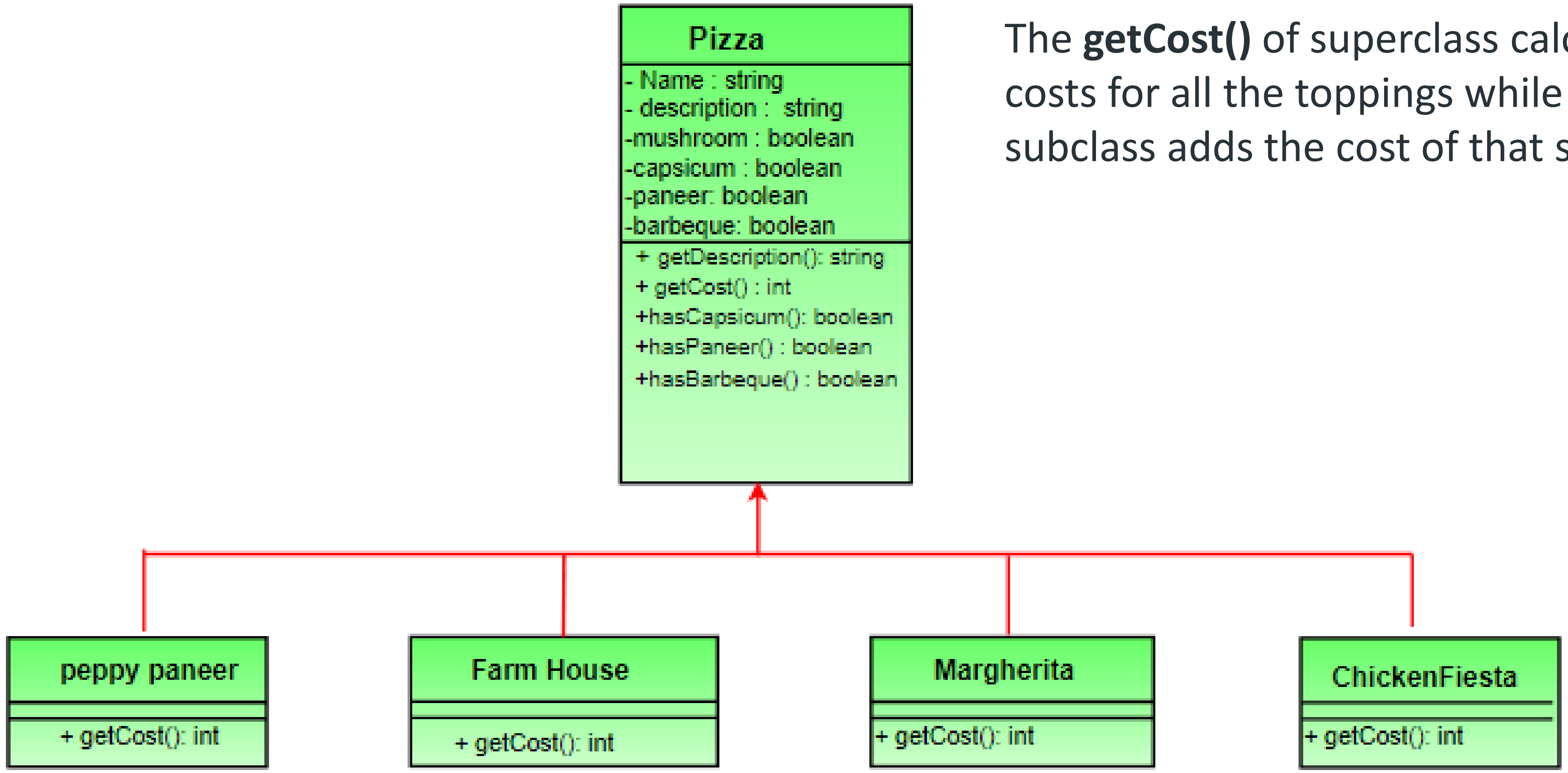
- ✓ This looks very complex.
- ✓ There are way too many classes and is a maintenance nightmare.
- ✓ Also if we want to add a new topping or pizza we have to add so many classes. This is obviously very bad design.



DECORATOR PATTERN

Option 2:

Let’s add instance variables to pizza base class to represent whether or not each pizza has a topping.



The **getCost()** of superclass calculates the costs for all the toppings while the one in the subclass adds the cost of that specific pizza.

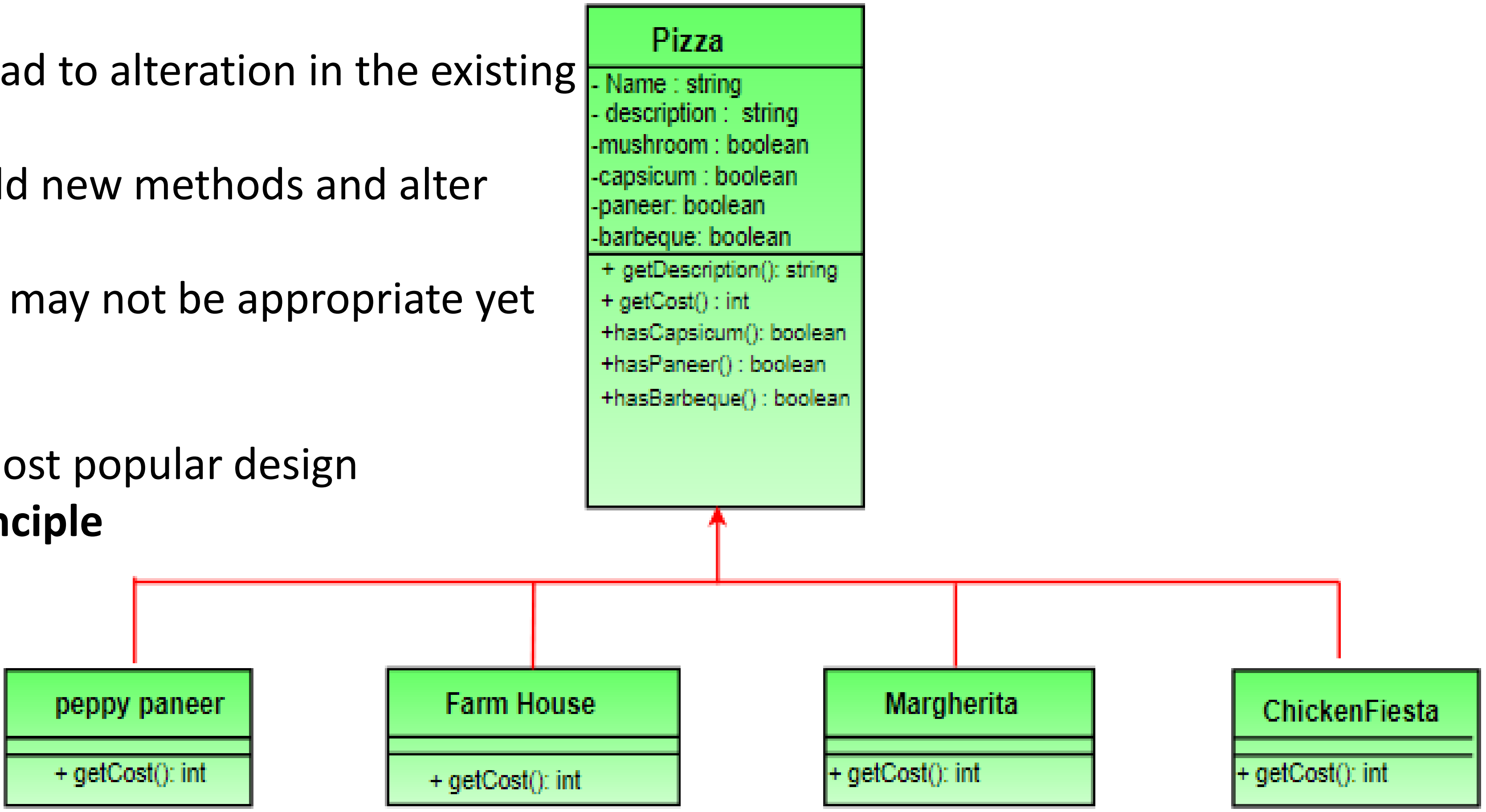
DECORATOR PATTERN

Option 2:

This design looks good at first but problems associated with it are:

- ✓ Price changes in toppings will lead to alteration in the existing code.
- ✓ New toppings will force us to add new methods and alter getCost() method in superclass.
- ✓ For some pizzas, some toppings may not be appropriate yet the subclass inherits them.

This design violates one of the most popular design principle – **The Open-Closed Principle**



Decorator Pattern

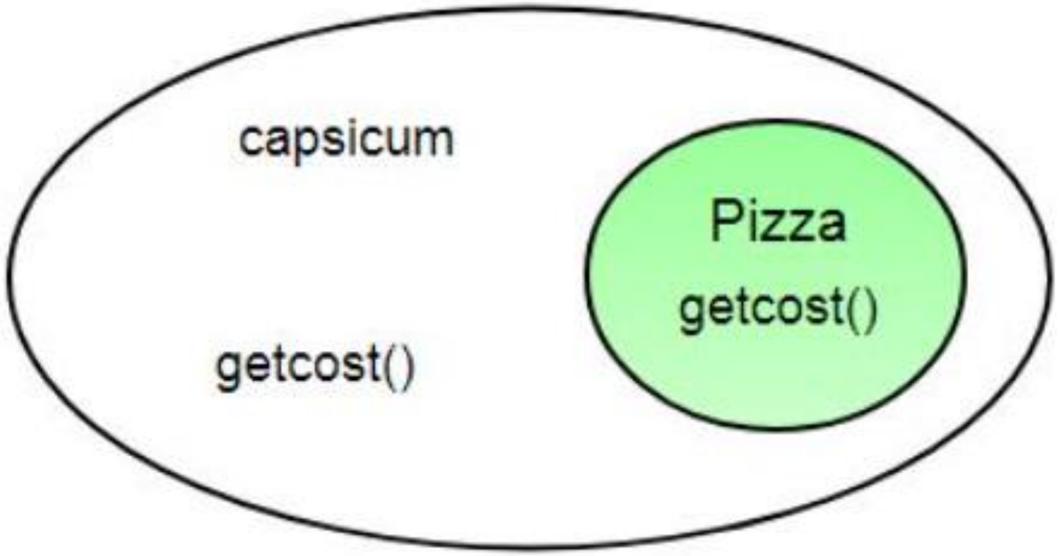
The decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

DECORATOR PATTERN

The solution:

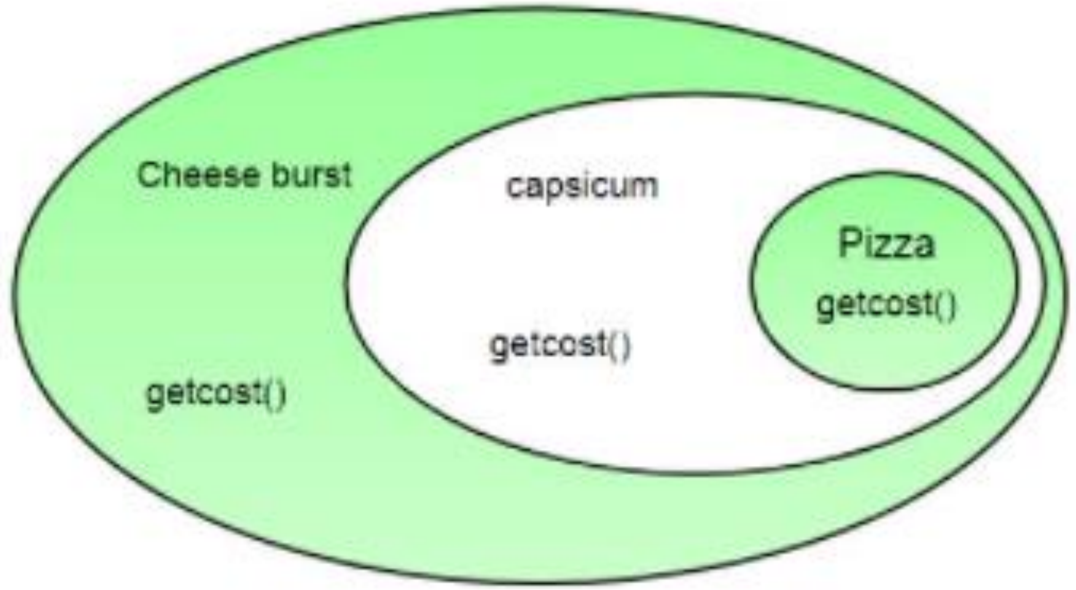
- ✓ Previous designs are not a good one
- ✓ Now, we will take a pizza and “decorate” it with toppings at runtime

1. Take a pizza object.
2. “Decorate” it with a Capsicum object.



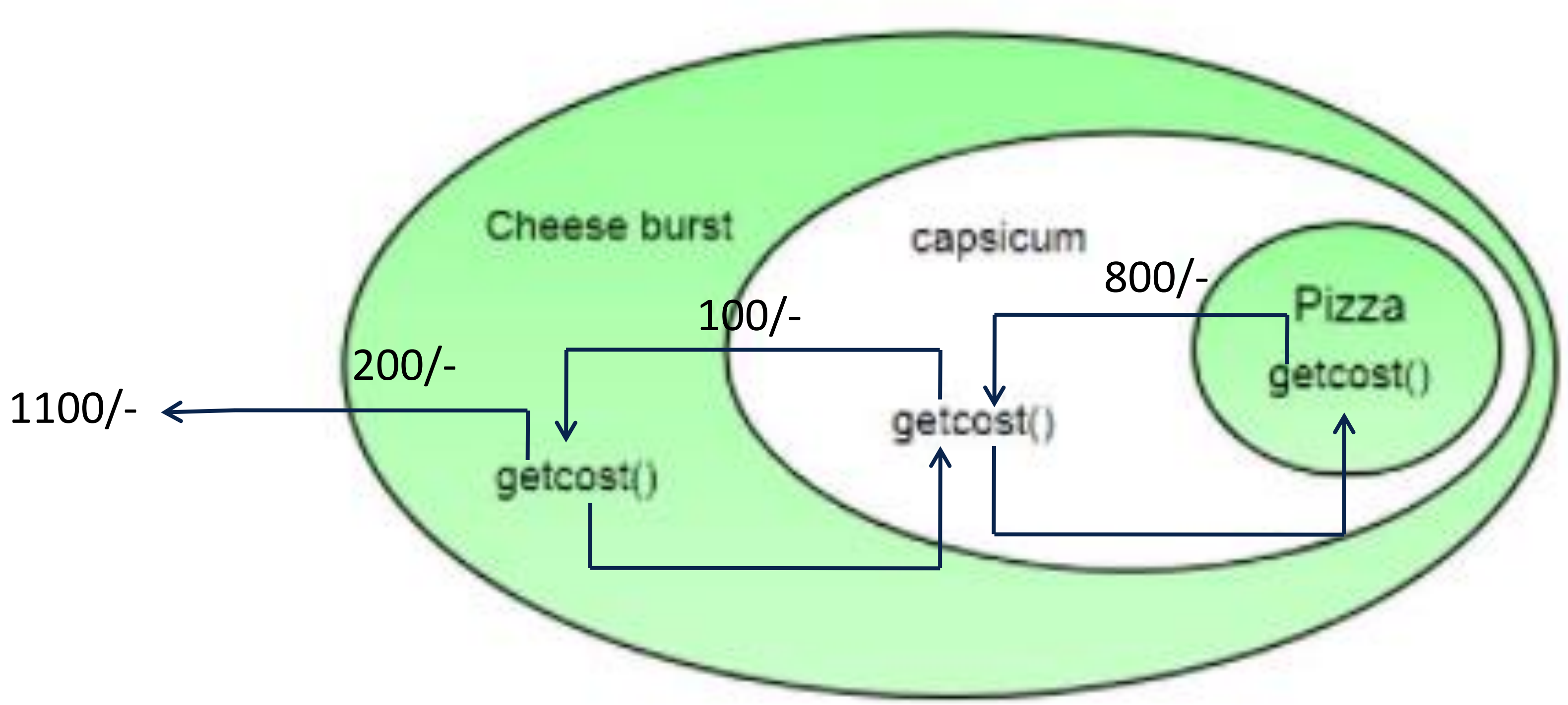
DECORATOR PATTERN

3. “Decorate” it with a CheeseBurst object.



DECORATOR PATTERN

4. Call getCost() and use delegation instead of inheritance to calculate the toppings cost.



✓Now we get a pizza with cheeseburst and capsicum toppings.
✓Visualize the “decorator” objects like wrappers.

DECORATOR PATTERN

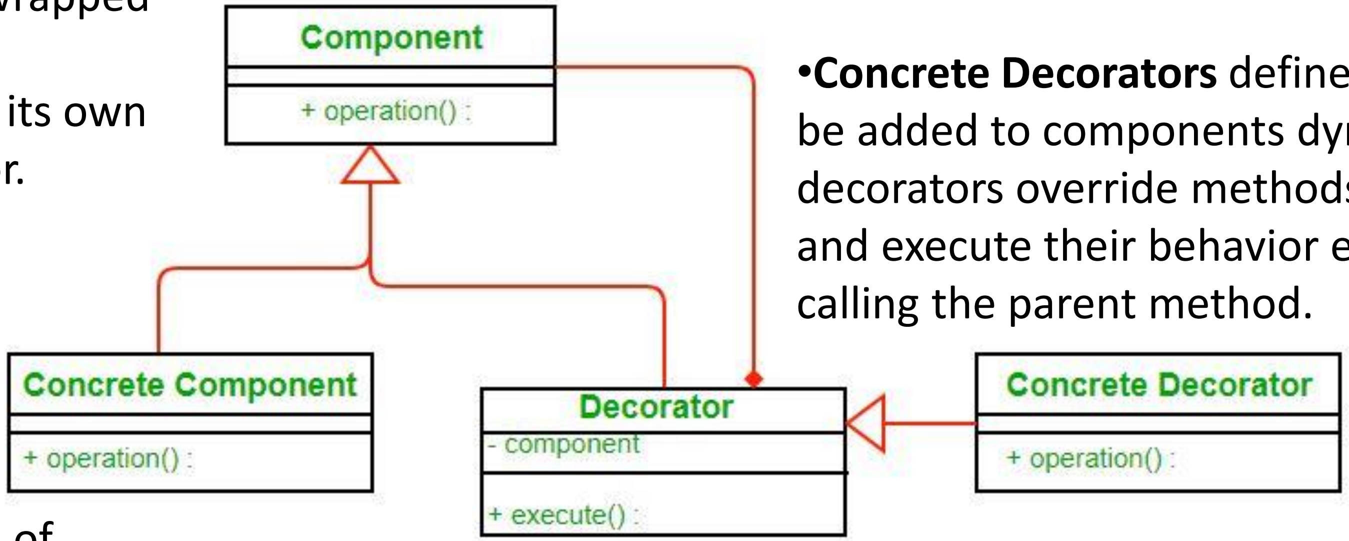
Here are some of the properties of decorators:

- ✓ Decorators have the same super type as the object they decorate.
- ✓ You can use multiple decorators to wrap an object.
- ✓ We can decorate objects at runtime.

Decorator Pattern

Structure:

•The **Component** declares the common interface for both wrappers and wrapped objects.
•Each component can be used on its own or may be wrapped by a decorator.



•**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.

•**Concrete Component** is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.
•The ConcreteComponent is the object we are going to dynamically decorate.

•The **Decorator** class has a field for referencing a wrapped object The decorator delegates all operations to the wrapped object.
•Each decorator has an instance variable that holds the reference to component it decorates(HAS-A relationship).

Decorator Pattern

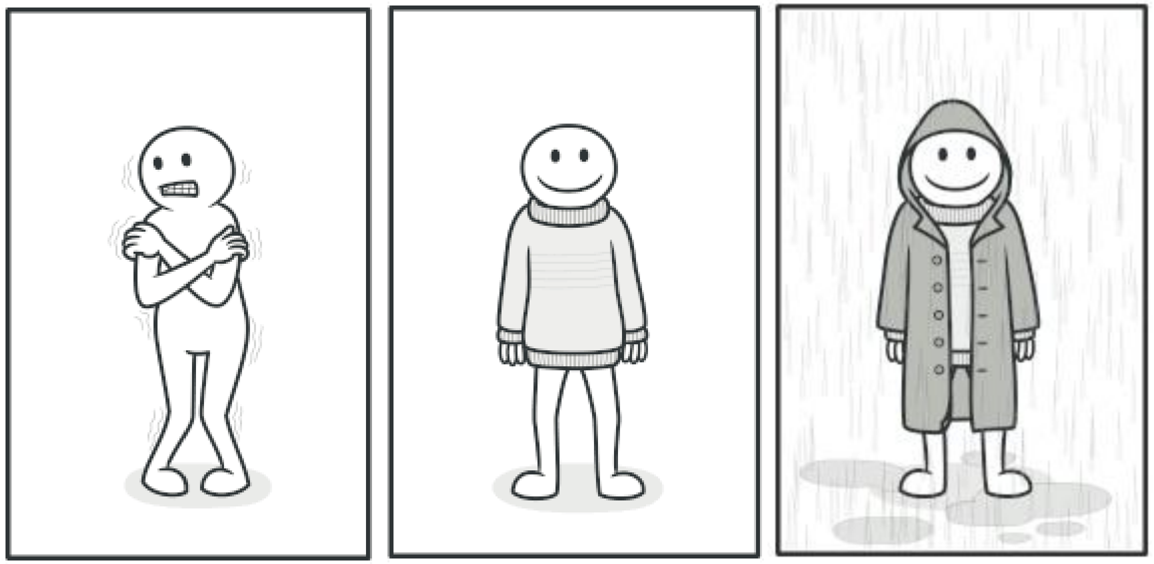
Advantages:

- ✓ The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime.

- ✓ The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects.

Decorator Pattern

Real-World Analogy

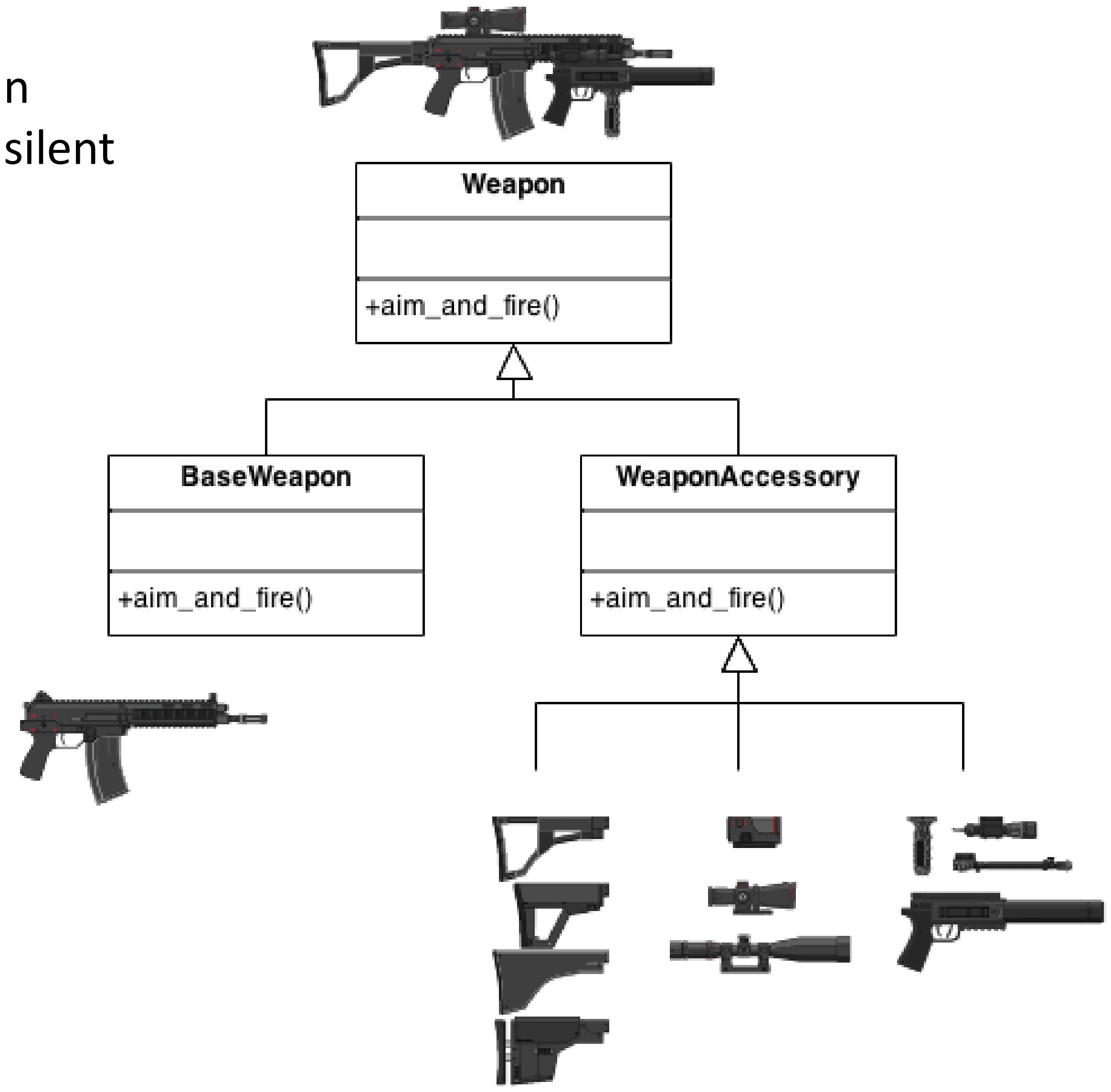


- ✓ Wearing clothes can be an example of using decorators.
- ✓ When you’re cold, you wrap yourself in a sweater.
- ✓ If you’re still cold with a sweater, you can wear a jacket on top.
- ✓ If it’s raining, you can put on a raincoat.
- ✓ All of these garments “extend” your basic behavior but aren’t part of you, and you can easily take off any piece of clothing whenever you don’t need it.

DECORATOR PATTERN

More scenario:

- ✓ assault gun is a deadly weapon on it's own. But you can apply certain "decorations" to make it more accurate, silent and devastating.



What is the diffence between decorator and strategy?
When we should use each of them?

Design Pattern:

Factory Method

Prepared by:
Tauseef Tajwar
180041109

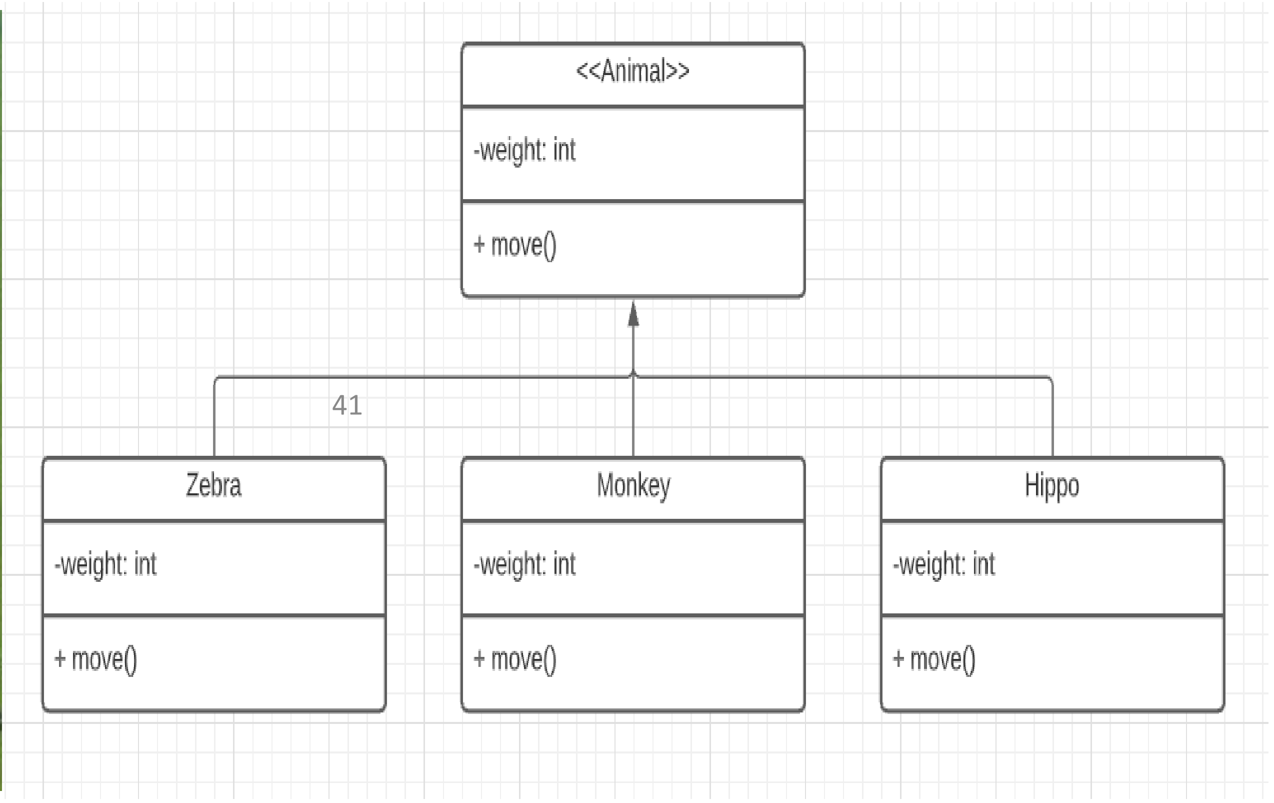
Definition

“The factory method pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiations to subclasses.”

This makes no sense at all.

Example

Let's say we are making a zoo simulator



Example

Basic Requirement: Spawn the animals in the zoo

How to do it: Create instances (or objects) of the animals

One way to do it: Hippo hippo = new Hippo()

Factory Method says: Wrap the instantiation

So,

```
Hippo hippo_creator()
{
    …
    Hippo hippo = new Hippo()
    …
    return hippo
}
```

Example

But why?

Two reasons:

1. Instantiation may be very complex. That is, the creation of the objects may depend on some logical calculations or business logic.

2. To achieve polymorphism. If the factory that creates the object is an instance itself, we can replace the instance with other instances at runtime. This is polymorphism.

Let’s expand our zoo simulator example.

Example

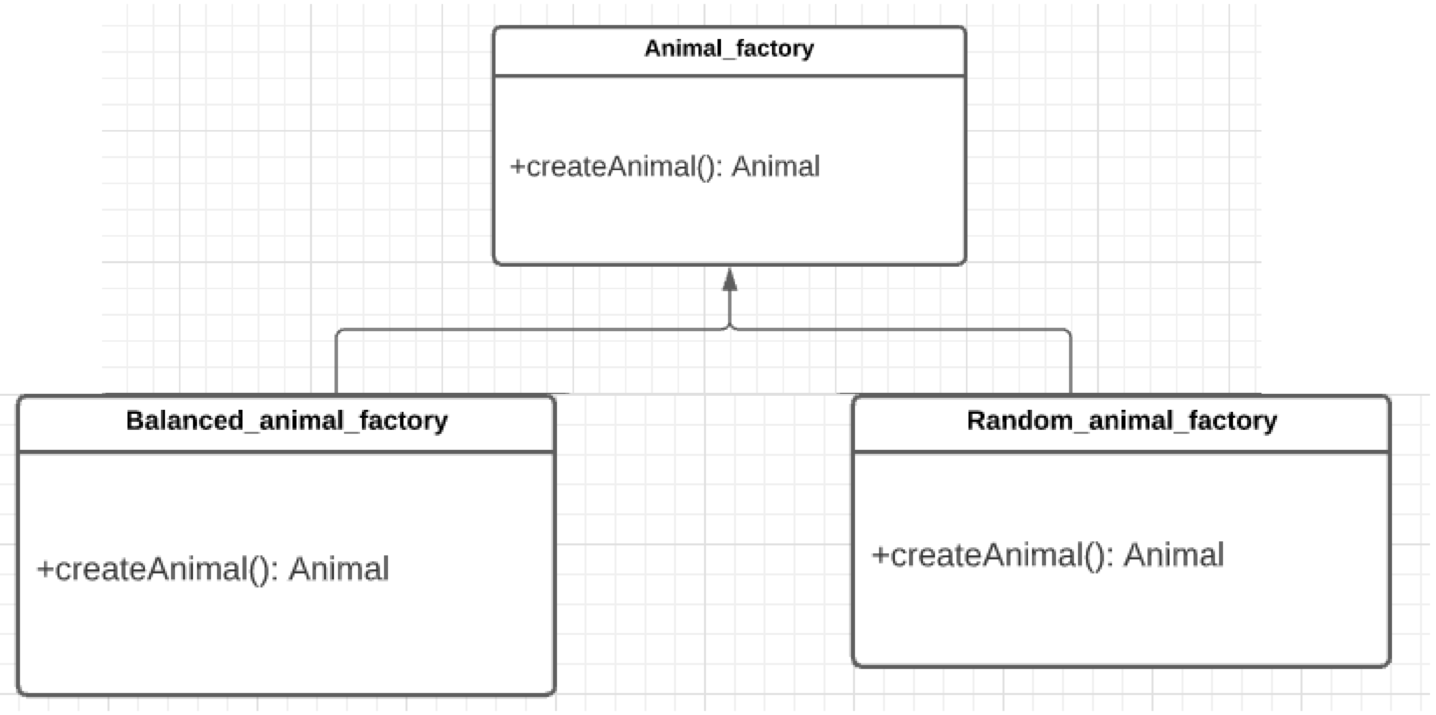
New requirements:

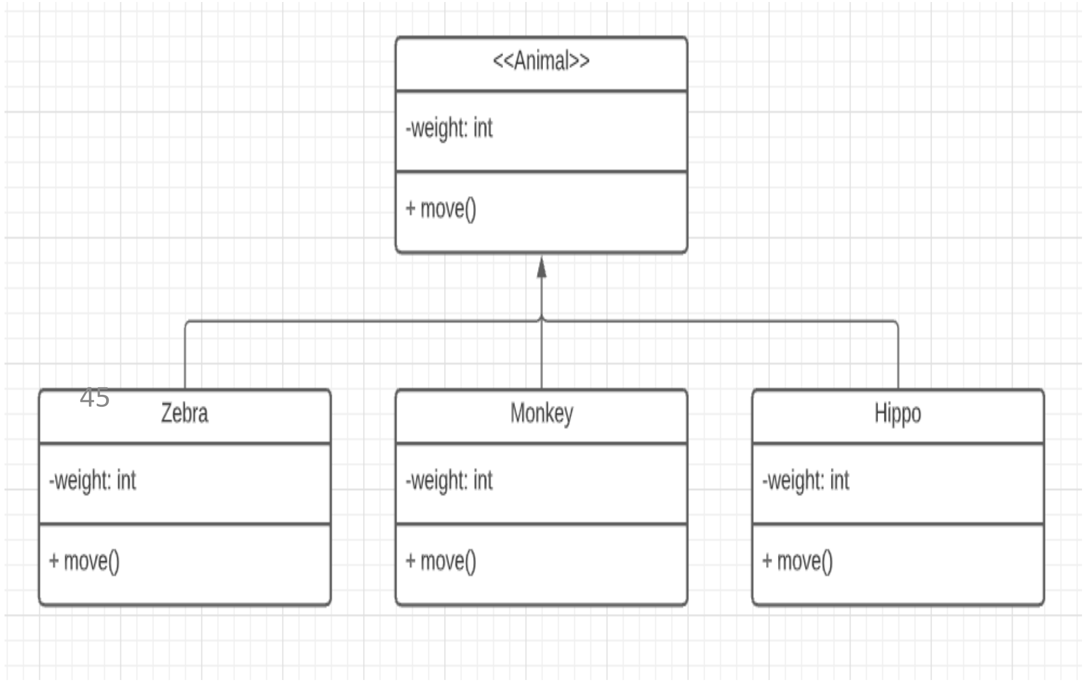
1. Animals should spawn completely randomly at some places.
2. At some other places there will be balanced randomness. That is, if too many hippos are randomly spawned somewhere, next few random spawns will not be that of hippos.

```
/* Some logic behind randomness */
      Hippo hippo = new Hippo()
```

```
/* Some logic behind balanced randomness */
      Hippo hippo = new Hippo()
```

Example

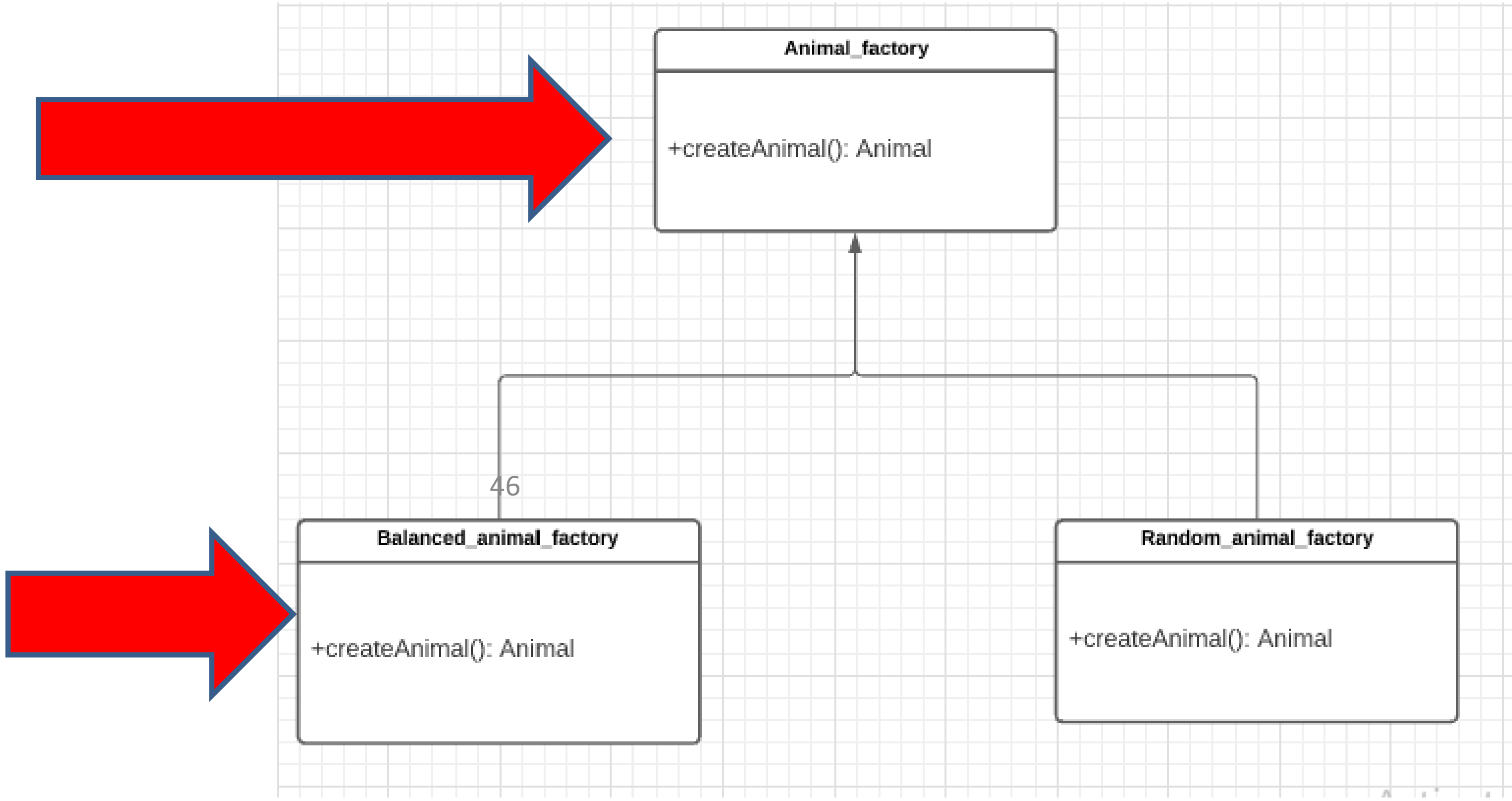

Animal_factory
+createAnimal(): Animal
Balanced_animal_factory
+createAnimal(): Animal
Random_animal_factory
+createAnimal(): Animal


<<Animal>>
-weight: int
+ move()
Zebra
-weight: int
+ move()
Monkey
-weight: int
+ move()
Hippo
-weight: int
+ move()

Revisiting the Definition

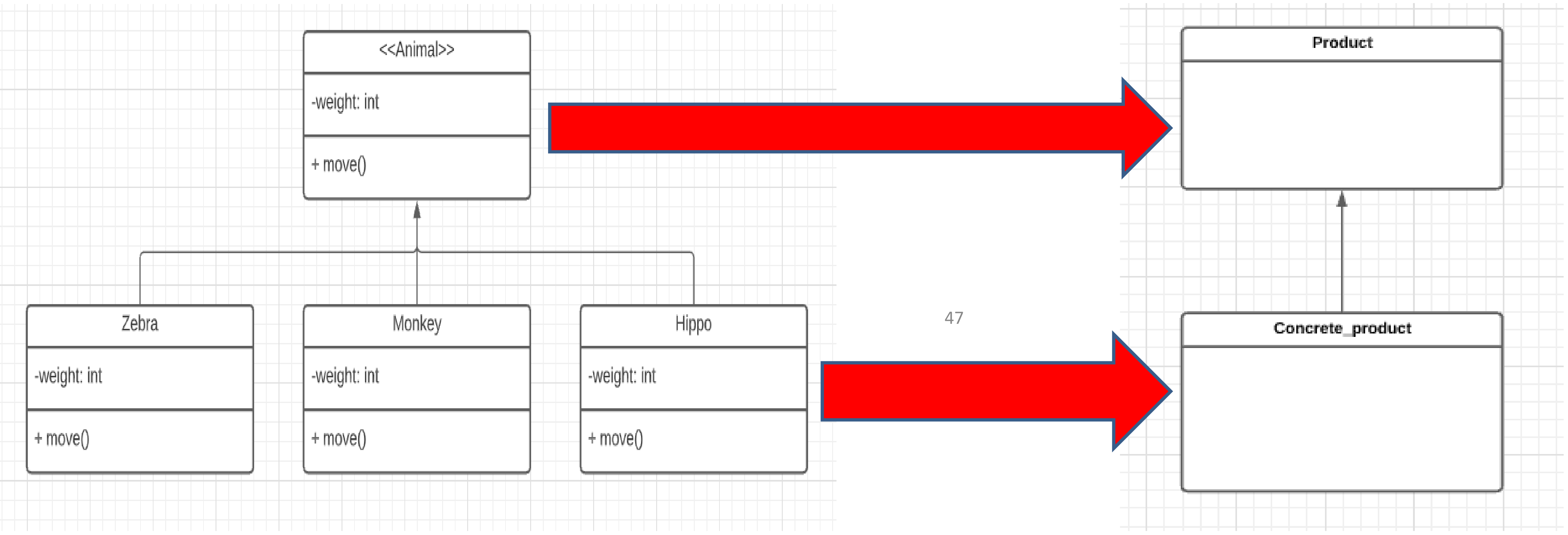
"The factory method pattern defines an interface for creating an object,

but lets subclasses decide which class to instantiate. Factory Method lets a class defer (yield to someone else's wish) instantiations to subclasses."

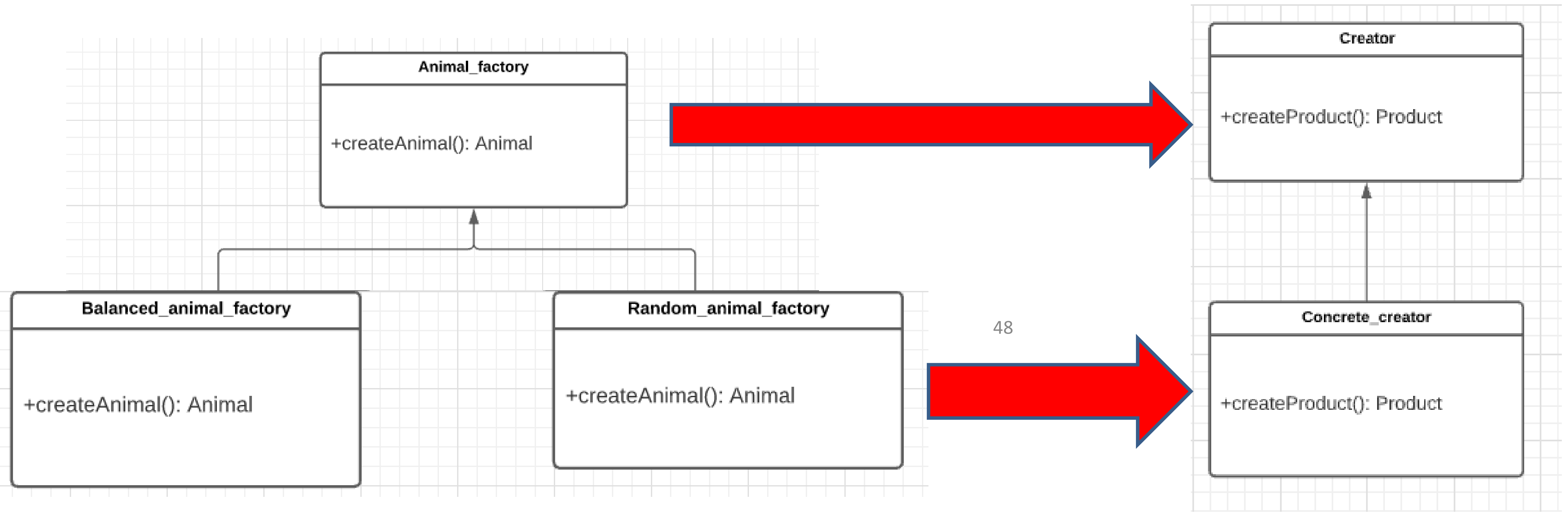


It all makes sense now!

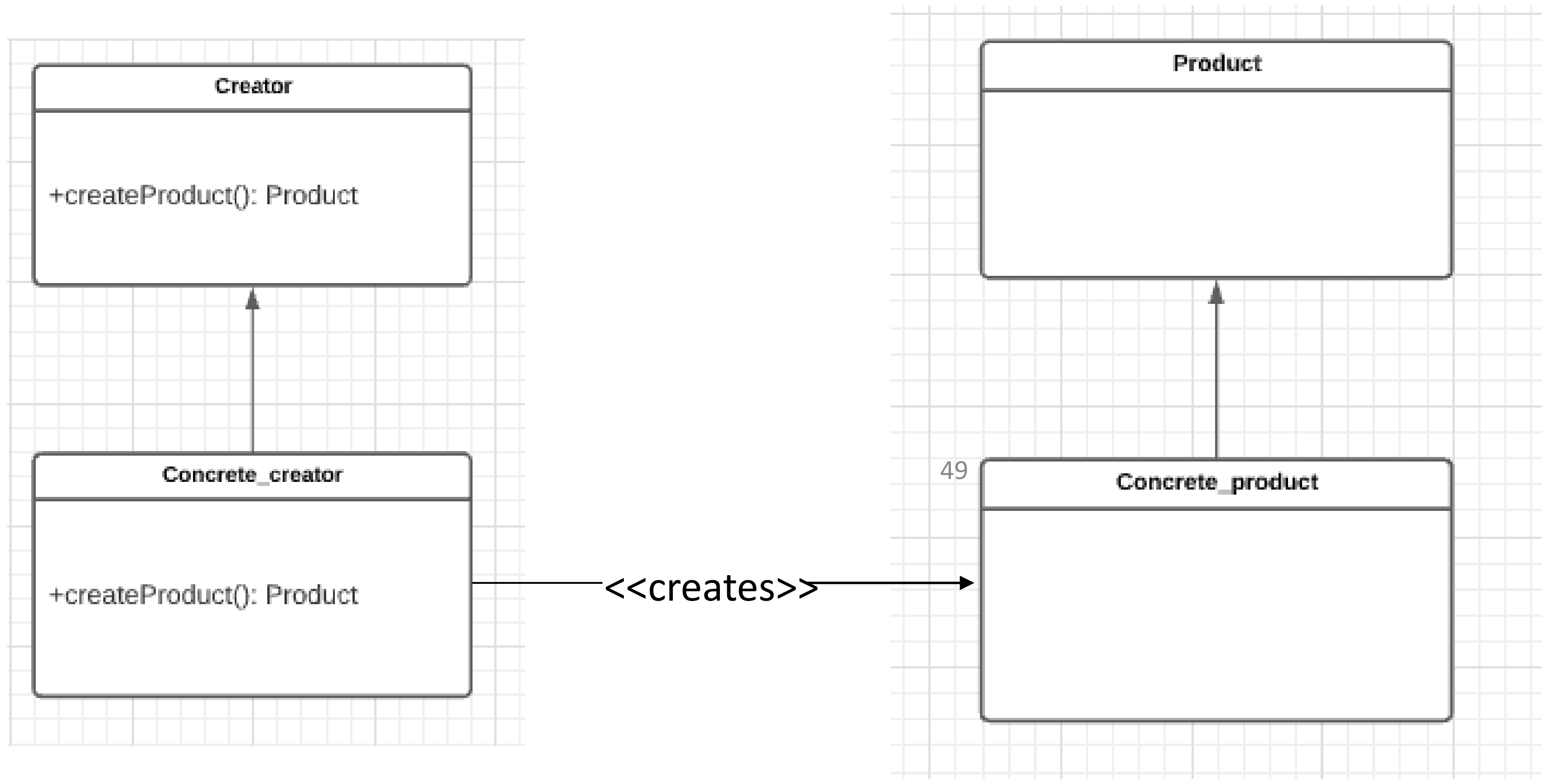
Some Notations



Some Notations



Structure


Creator
+createProduct(): Product
Concrete_creator
+createProduct(): Product
<<creates>>
Product
Concrete_product

To Summarize

1. Depending on the **business logic**, you may want to create different objects. This logic can be encapsulated by a factory.

2. A factory is a way to create the products. So, if you have two factories, you have two ways of instantiating the product. They both create the **same product**, but in **different ways**.

3. Not only do factories provide different ways to instantiate a product, it provides ways to instantiate **different subtypes** of the product.

Design Patterns

Command Pattern

[illegible]ash[illegible] Hoss[illegible]in 18[illegible]041128

Table of Contents

What *Is* the Command Pattern?

The Command Pattern *encapsulates* a request as an *object*, thereby letting you *parameterize* other objects with *different requests*, queue or log requests, and *support undoable operations*.

What *Is* the Command Pattern?

1. ***encapsulates*** a request as an *object*
1. ***parameterize*** other objects with ***different requests***,
1. queue or log requests, and
1. support ***undoable*** operations.

Example

- ❑ Every button **Does** something
- ❑ To and **Entity**
- ❑ When **Commanded** by an entity

Every Button Does Something

Doing something means Executing something

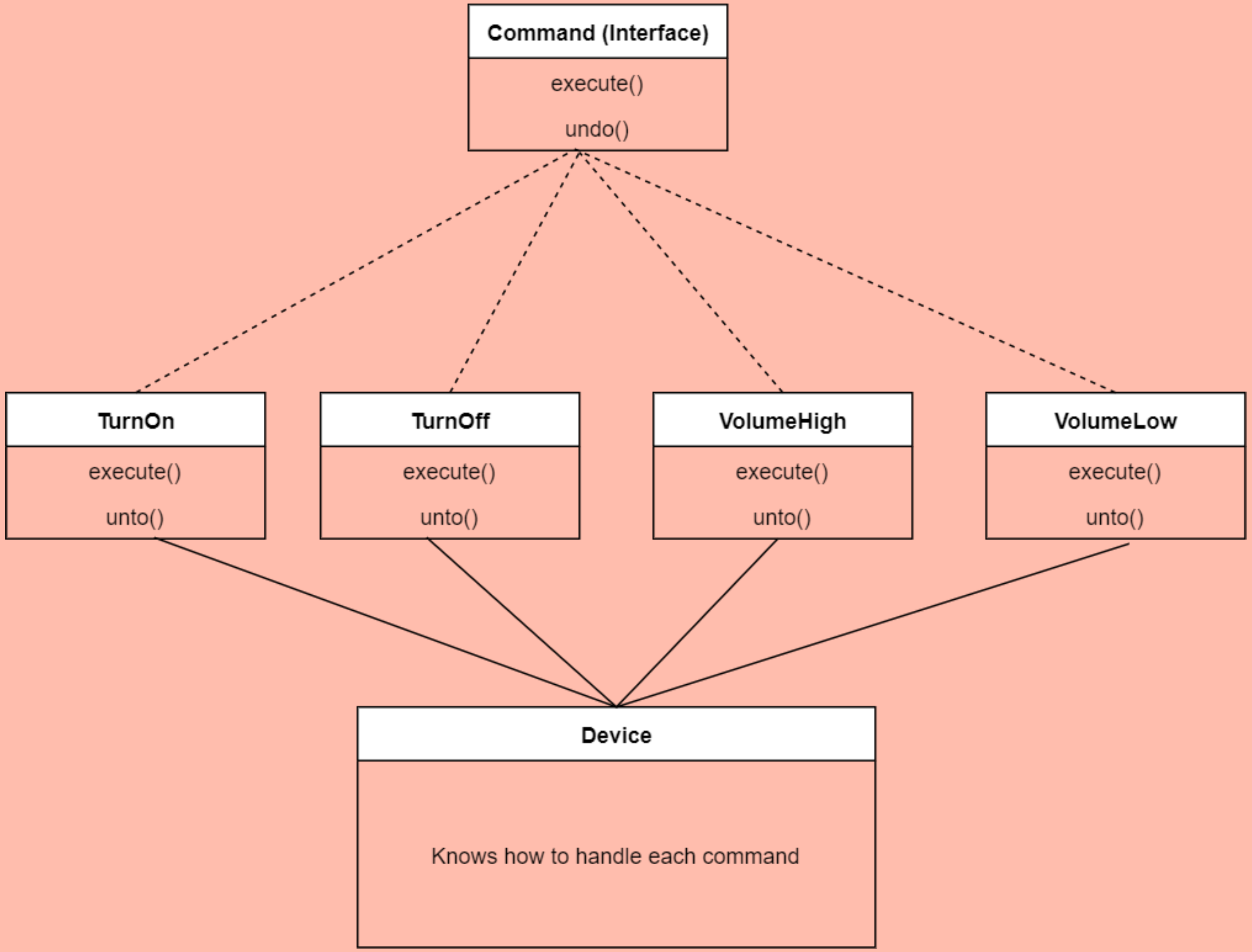
To an Entity

In this case, a Television

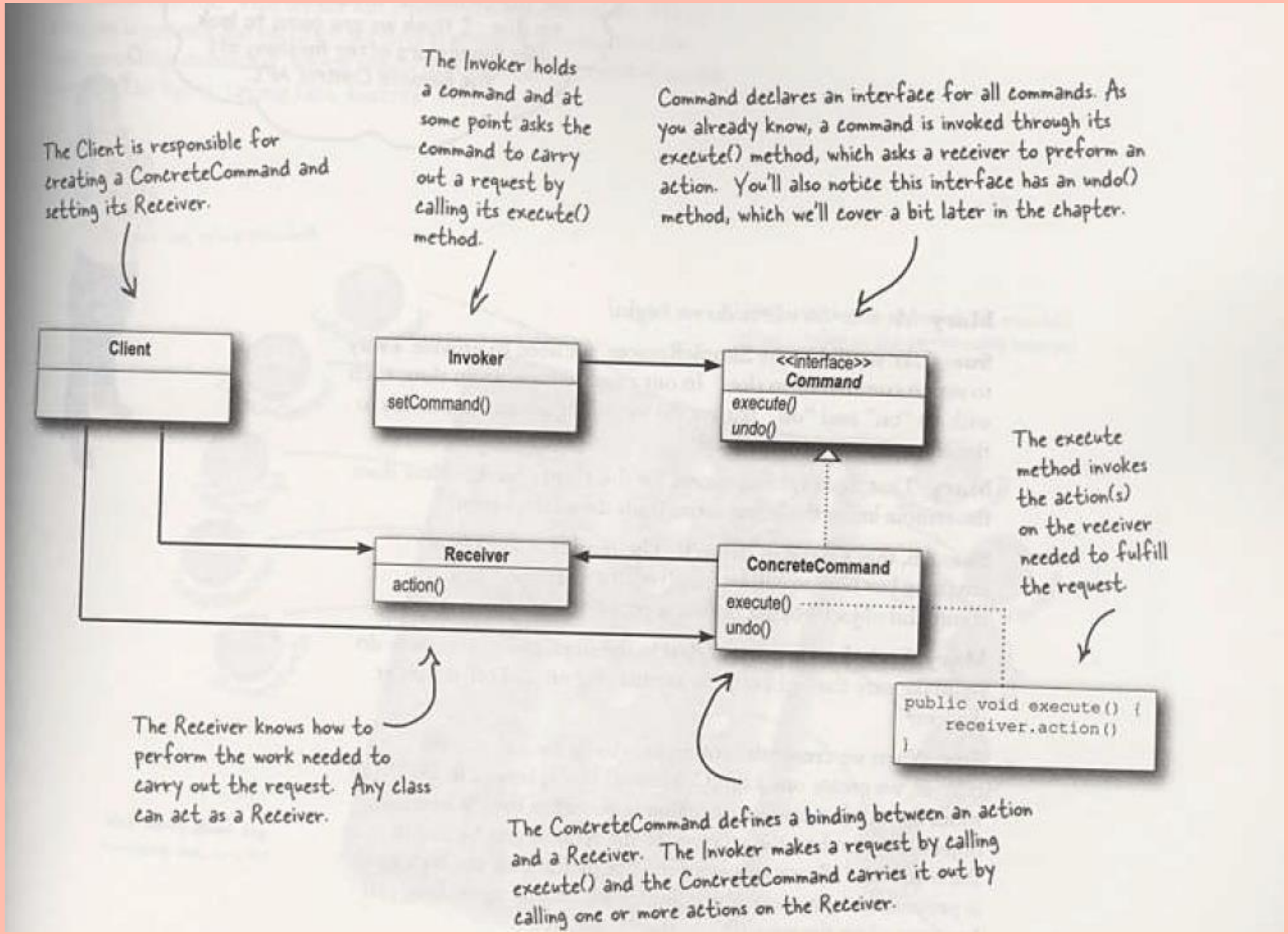
When Commanded by an Entity

In this case you/the viewer.

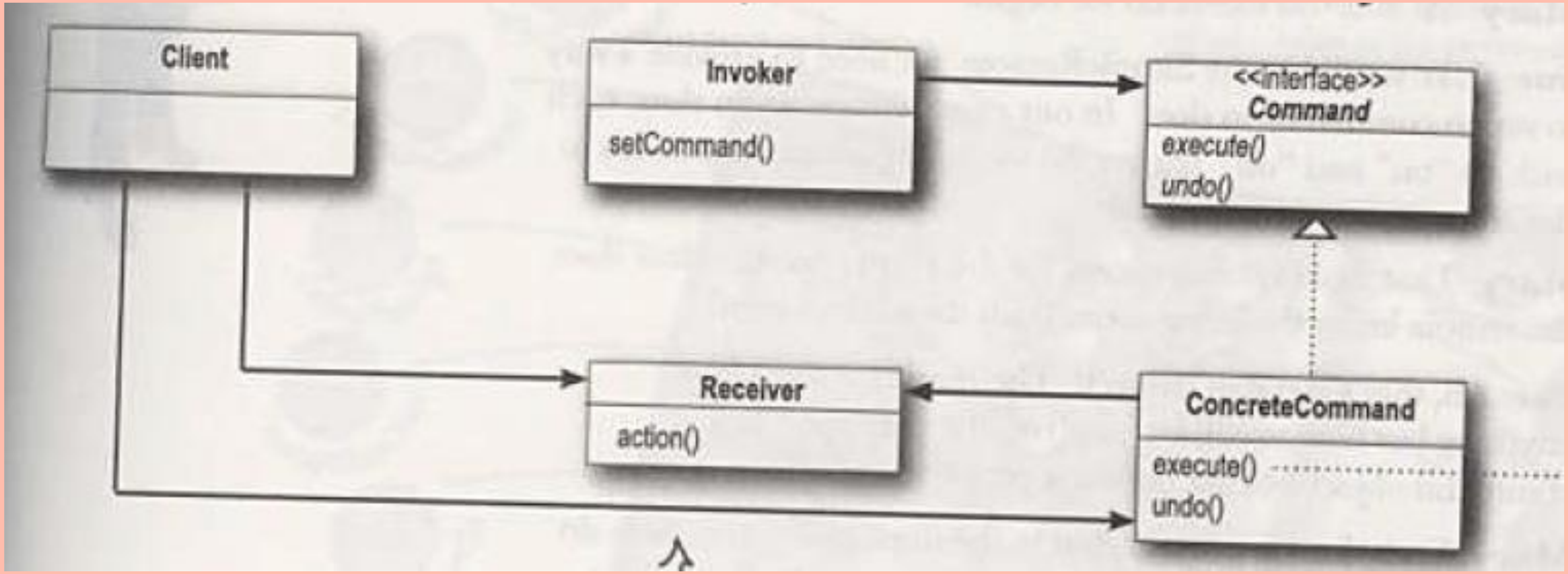
From the Command Pattern Paradigm



From the Command Pattern Paradigm


The Client is responsible for creating a ConcreteCommand and setting its Receiver.
The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.
Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to preform an action. You'll also notice this interface has an undo() method, which we'll cover a bit later in the chapter.
Client
Invoker
setCommand()
<<interface>>
Command
execute()
undo()
The execute method invokes the action(s) on the receiver needed to fulfill the request.
Receiver
action()
ConcreteCommand
execute()
undo()
public void execute() {
receiver.action()
}
The Receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.
The ConcreteCommand defines a binding between an action and a Receiver. The Invoker makes a request by calling execute() and the ConcreteCommand carries it out by calling one or more actions on the Receiver.

From the Command Pattern Paradigm



What *Is* the Command Pattern?

The Command Pattern *encapsulates* a request as an *object*, thereby letting you *parameterize* other objects with *different requests*, queue or log requests, and *support undoable operations.*

What *Is* the Command Pattern?

The Command Pattern *encapsulates* a request as an *object*, thereby letting you *parameterize* other objects with *different requests*, queue or log requests, and *support undoable operations*.

What *Is* the Command Pattern?

The Command Pattern *encapsulates* a request as an *object*, thereby letting you *parameterize* other objects with *different requests*, queue or log requests, and *support undoable operations*.

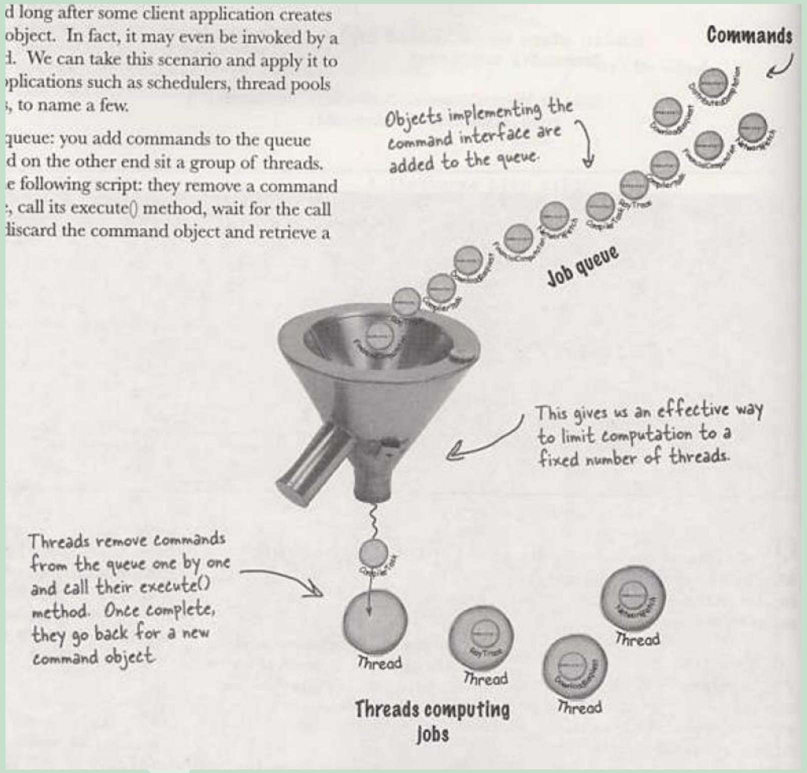
What *Is* the Command Pattern?

The Command Pattern *encapsulates* a request as an *object*, thereby letting you *parameterize* other objects with *different requests*, queue or log requests, and *support undoable operations*.

Queue and Logging!

Queue and Logging

Queue

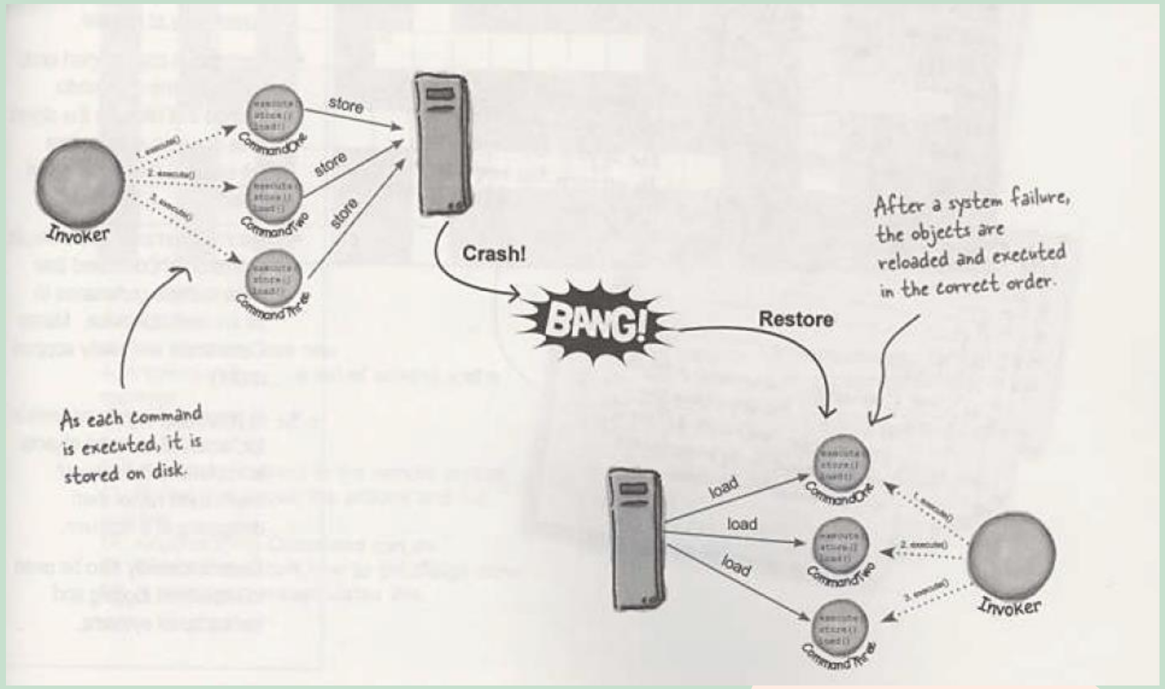


d long after some client application creates
object. In fact, it may even be invoked by a
d. We can take this scenario and apply it to
plications such as schedulers, thread pools
, to name a few.

queue: you add commands to the queue
d on the other end sit a group of threads.
e following script: they remove a command
, call its execute() method, wait for the call
discard the command object and retrieve a

Commands
Objects implementing the command interface are added to the queue.
Job queue
This gives us an effective way to limit computation to a fixed number of threads.
Threads remove commands from the queue one by one and call their execute() method. Once complete, they go back for a new command object.
Thread
Thread
Thread
Thread
Threads computing jobs

Logging



Invoker
1. execute()
2. execute()
3. execute()
CommandOne
CommandTwo
CommandThree
store
store
store
Crash!
BANG!
Restore
As each command is executed, it is stored on disk.
After a system failure, the objects are reloaded and executed in the correct order.
load
load
load
Invoker

Use Case

GOAL:

Decoupling

Bridge Design Pattern

Asir Saadat Nipun
180041127

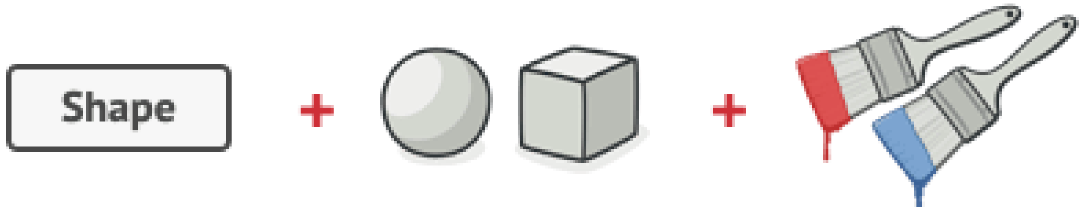
Definition:

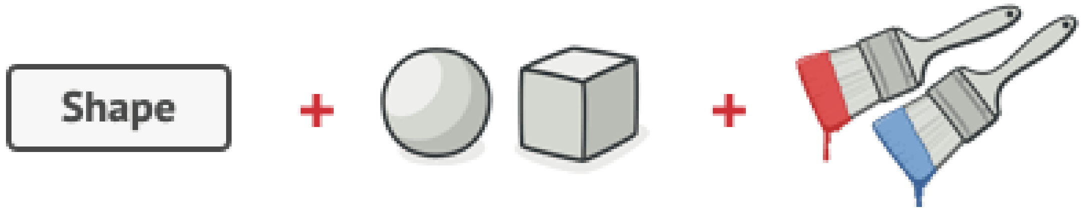
Decouple an abstraction from its implementation so that the two can vary independently

Abstraction

Abstraction is the concept of **object-oriented programming** that "shows" only essential attributes and "hides" unnecessary information.

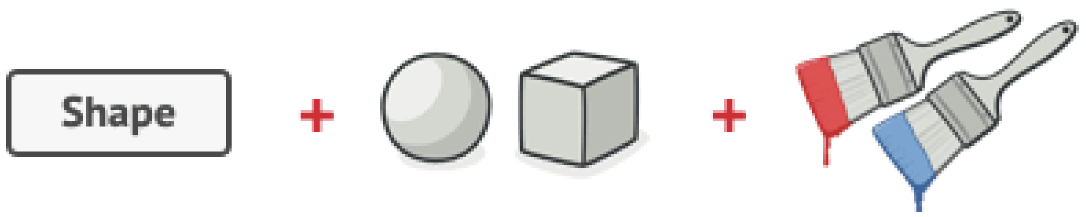
Now lets look at a simple example

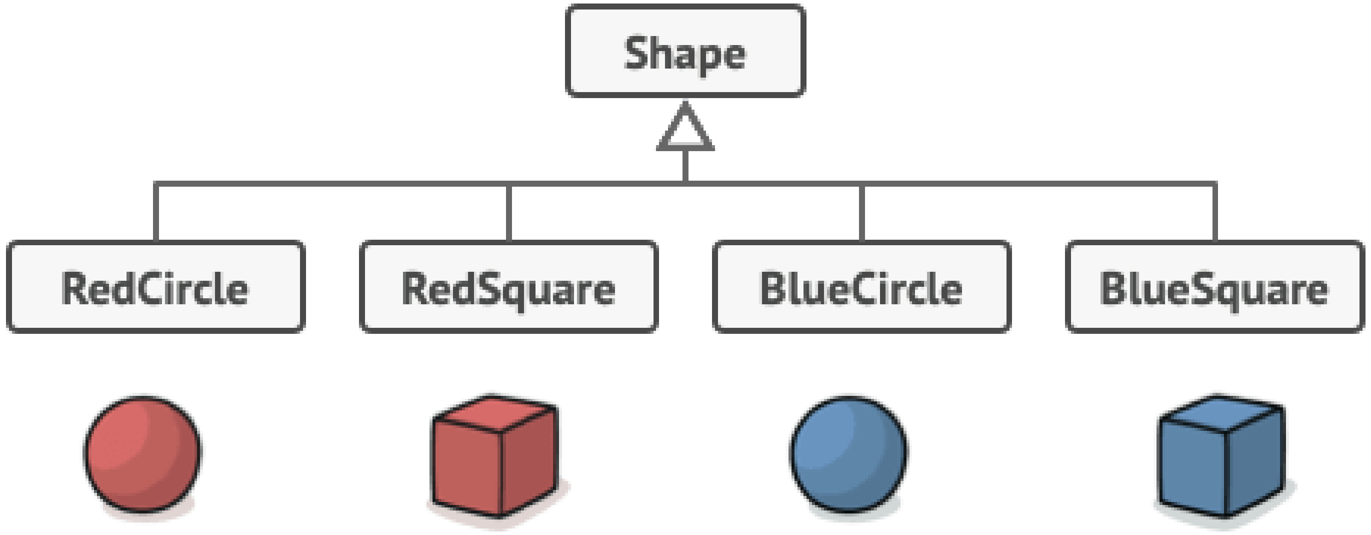

Shape
+
+



We want to implement two dimensions.

1. The shape
2. The color





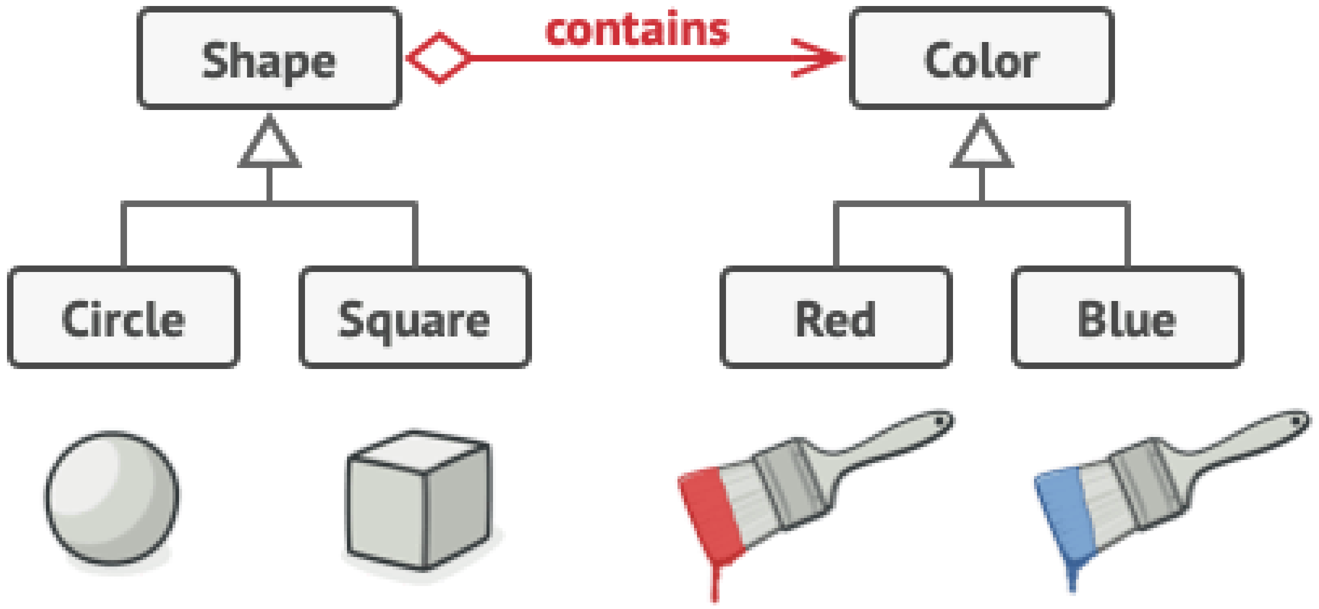
We want to implement two dimensions.

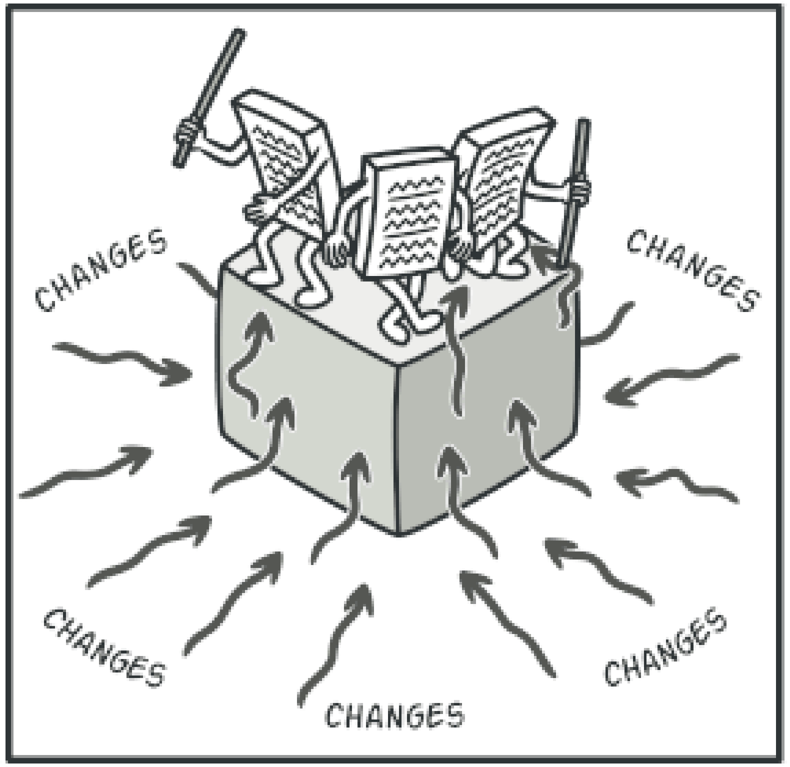
1. The shape
2. The color

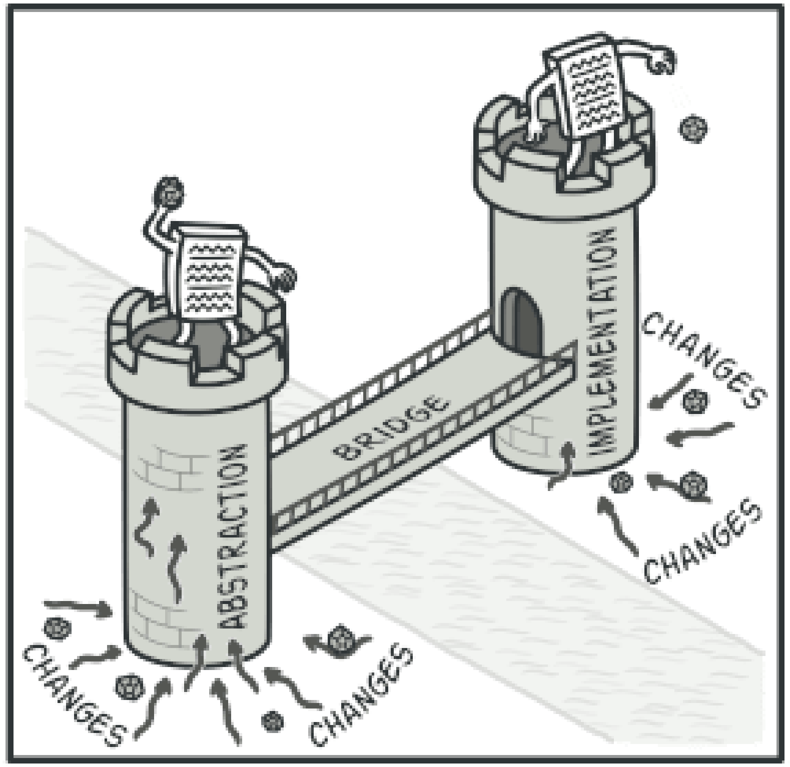
Solution:

The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition.

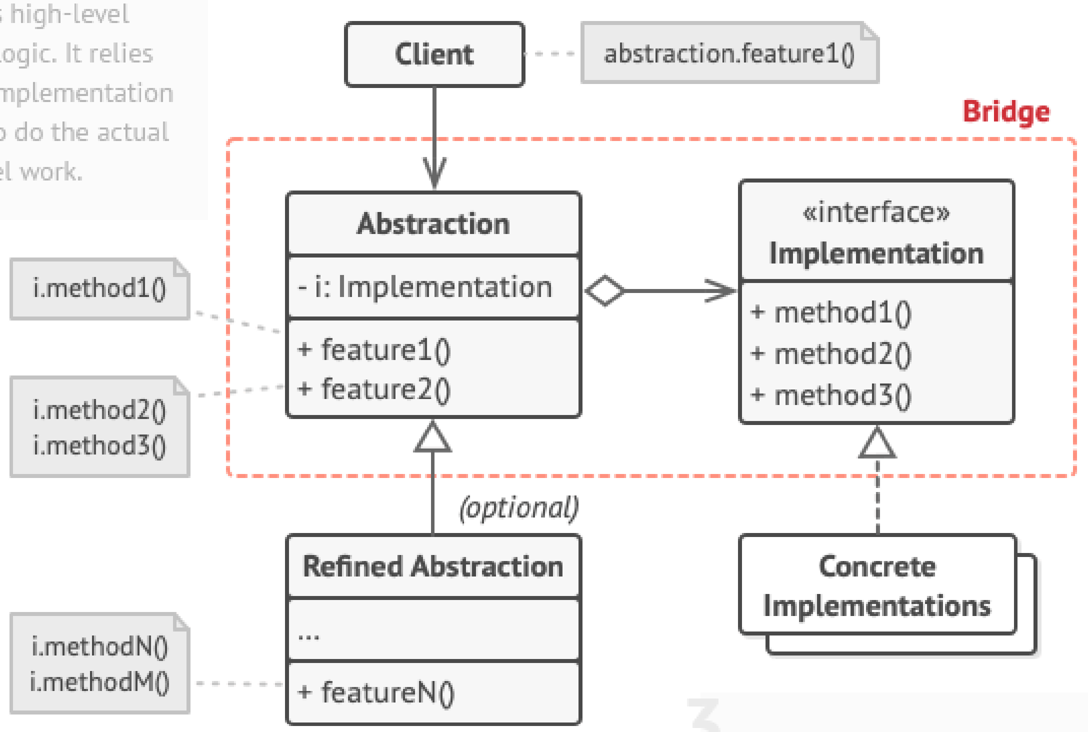
What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy.


Shape
contains
Color
Circle
Square
Red
Blue


CHANGES
CHANGES
CHANGES
CHANGES
CHANGES
CHANGES


IMPLEMENTATION
CHANGES
BRIDGE
ABSTRACTION
CHANGES
CHANGES
CHANGES

Structure



1 The **Abstraction** provides high-level control logic. It relies on the implementation object to do the actual low-level work.

2 The **Implementation** declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.

The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation.

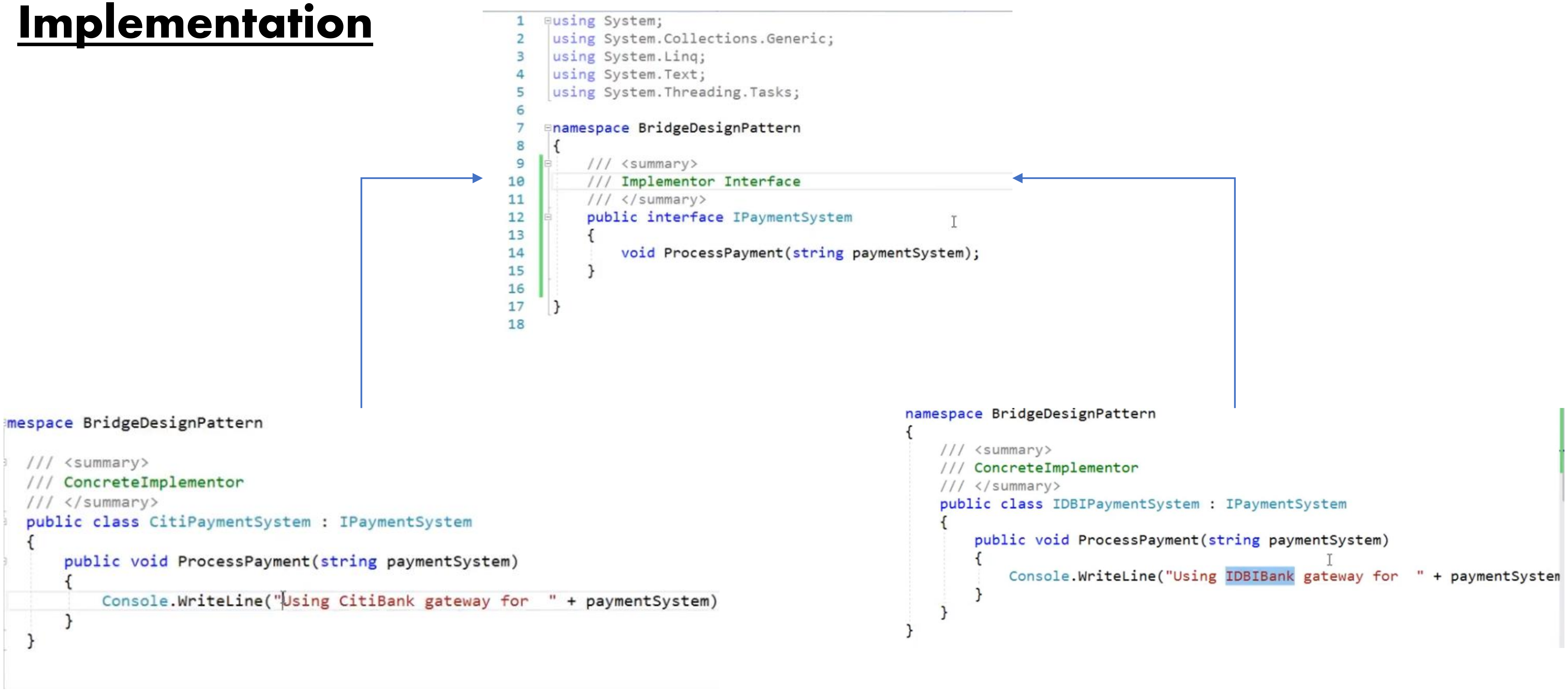
3 **Concrete Implementations** contain platform-specific code.

4 **Refined Abstractions** provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface.

5 Usually, the **Client** is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.

Code

Implementation

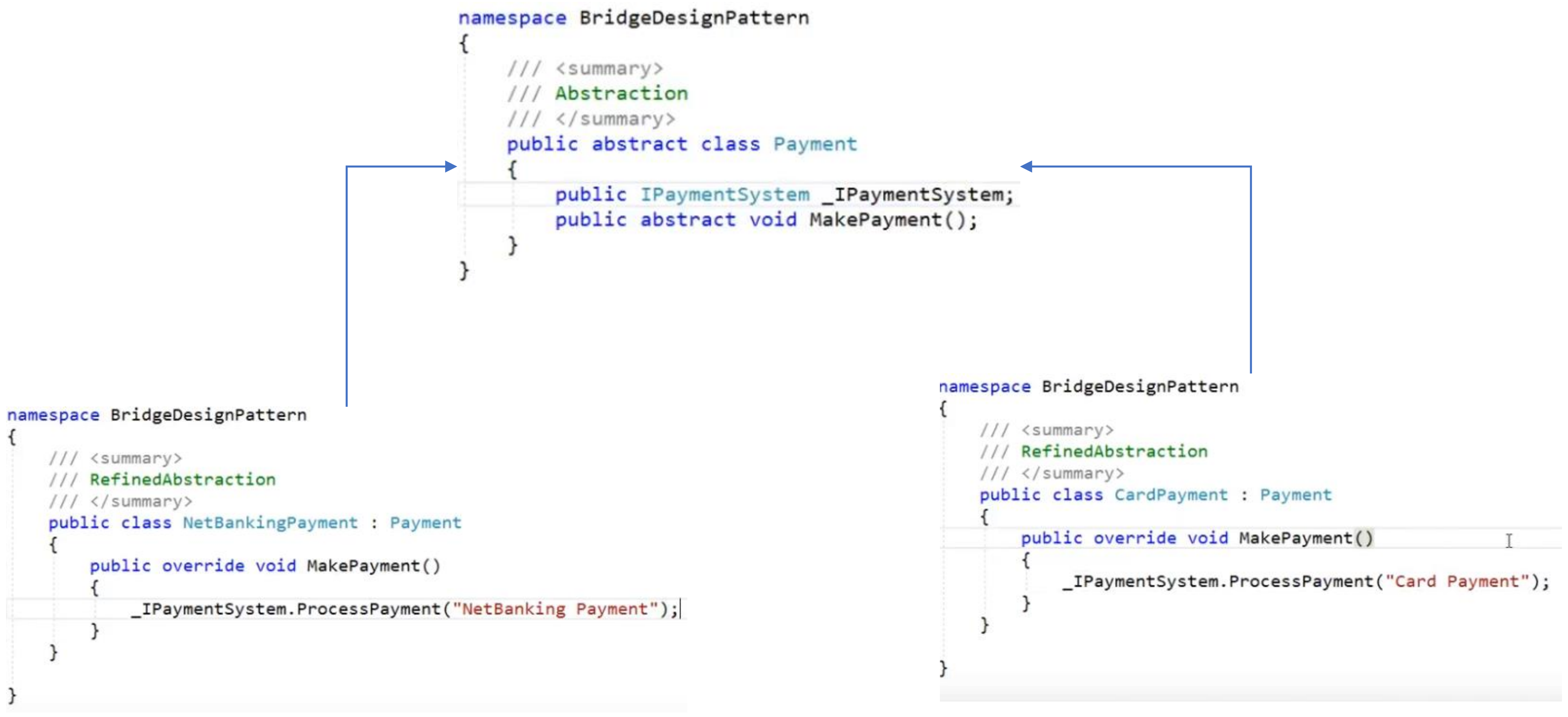


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BridgeDesignPattern
{
    /// <summary>
    /// Implementor Interface
    /// </summary>
    public interface IPaymentSystem
    {
        void ProcessPayment(string paymentSystem);
    }

}
```

```
mespace BridgeDesignPattern

 /// <summary>
 /// ConcreteImplementor
 /// </summary>
 public class CitiPaymentSystem : IPaymentSystem
 {
     public void ProcessPayment(string paymentSystem)
     {
         Console.WriteLine("Using CitiBank gateway for  " + paymentSystem)
     }
 }
```

```
namespace BridgeDesignPattern
{
    /// <summary>
    /// ConcreteImplementor
    /// </summary>
    public class IDBIPaymentSystem : IPaymentSystem
    {
        public void ProcessPayment(string paymentSystem)
        {
            Console.WriteLine("Using IDBIBank gateway for  " + paymentSysten
        }
    }
}
```

Abstraction

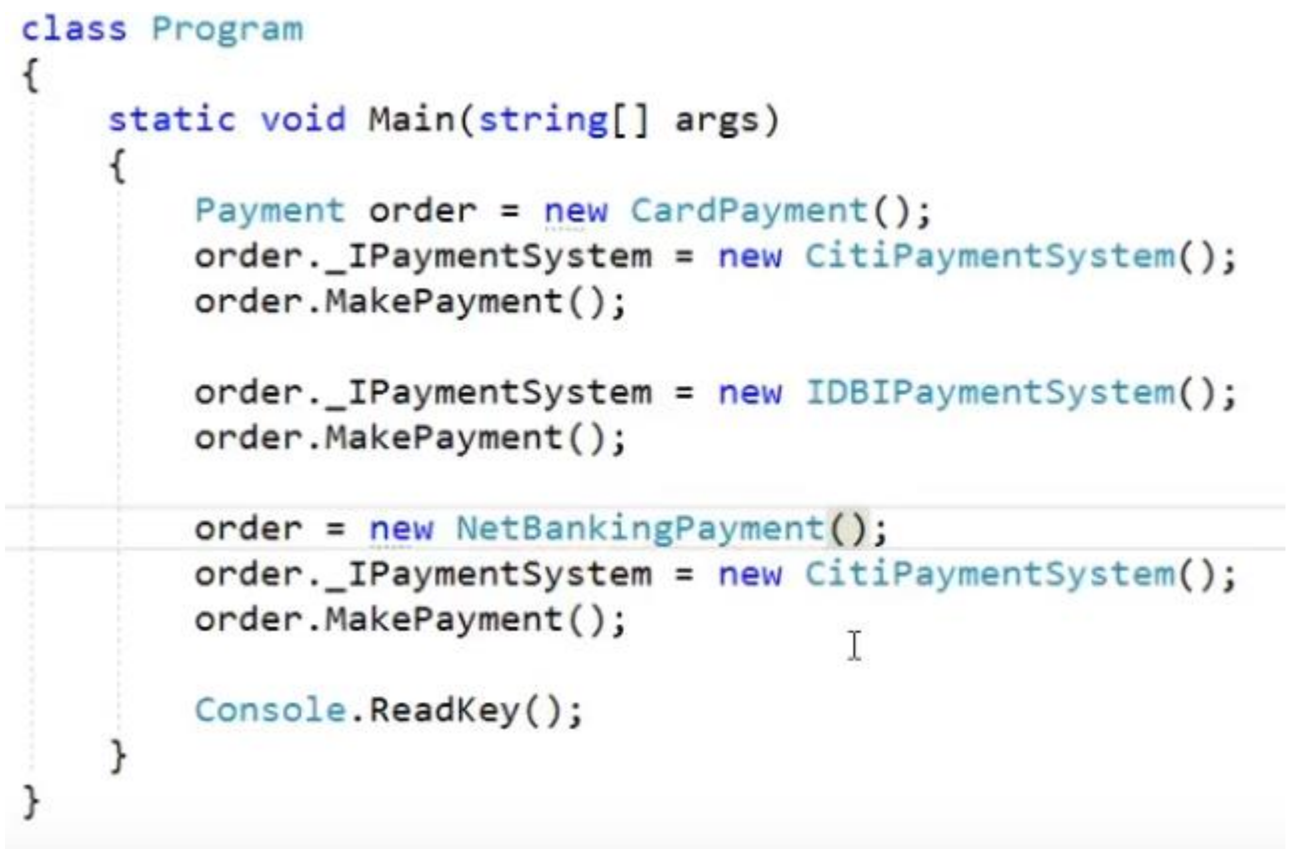


```
namespace BridgeDesignPattern
{
    /// <summary>
    /// Abstraction
    /// </summary>
    public abstract class Payment
    {
        public IPaymentSystem _IPaymentSystem;
        public abstract void MakePayment();
    }
}
```

```
namespace BridgeDesignPattern
{
    /// <summary>
    /// RefinedAbstraction
    /// </summary>
    public class NetBankingPayment : Payment
    {
        public override void MakePayment()
        {
            _IPaymentSystem.ProcessPayment("NetBanking Payment");
        }
    }

}
```

```
namespace BridgeDesignPattern
{
    /// <summary>
    /// RefinedAbstraction
    /// </summary>
    public class CardPayment : Payment
    {
        public override void MakePayment()
        {
            _IPaymentSystem.ProcessPayment("Card Payment");
        }
    }

}
```

Abstraction



```
class Program
{
    static void Main(string[] args)
    {
        Payment order = new CardPayment();
        order._IPaymentSystem = new CitiPaymentSystem();
        order.MakePayment();

        order._IPaymentSystem = new IDBIPaymentSystem();
        order.MakePayment();

        order = new NetBankingPayment();
        order._IPaymentSystem = new CitiPaymentSystem();
        order.MakePayment();

        Console.ReadKey();
    }
}
```

Singleton Principle

Presented by
M. K. Bashar : 180041238

Responsibility

Violating Single Responsibility Principle

1. Only one instance for a Singleton Class

1. The class has to provide global point of access to the unique instance

Use Cases

There are many objects we only need one of:

- Thread pools
- Caches
- Top level UI (Window, Frame)
- Objects that
  - Handle preferences and registry settings
  - Used for **logging**
  - Act as device drivers to devices like
    - Printers
    - Graphics cards.

Real-World Analogy

- A country can have only one official government.
- Regardless of the personal identities of the individuals who form governments, the title, “The Government of X”, is a global point of access that identifies the group of people in charge.

Class Structure

```
class Singleton {
        private static Singleton Instance;

        private Singleton ( ) {
        }

        public static getInstance ( ) {
                if (Instance == null) {
            Instance = new Singleton();
        }
                return Instance ;
        }
        // other useful methods here
}
```

SW E Presentation
on
Proxy design pattern

Presented By –

Nuzhat Nower

ID:180041116

Section : A

Course: CSE 4513

Definition

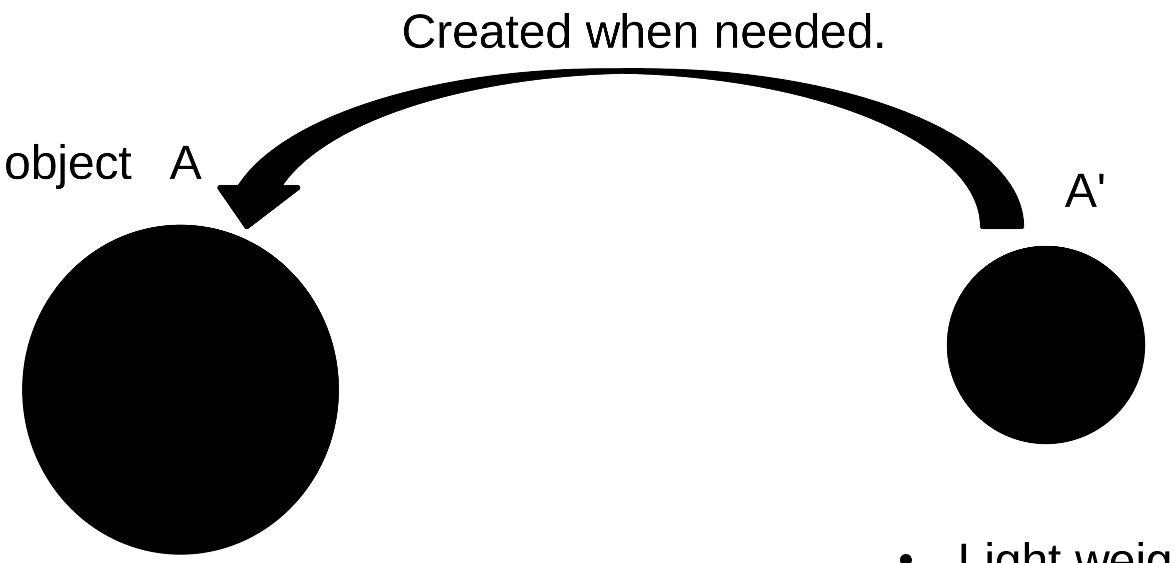
- In Proxy pattern,we create object having original object to interface it's functionality to outer world.
- Provide a suBSTITUTE or placeholder for another object.

Motivation and Intent

- **Proxy pattern comes into play when we have HeavyWeight objects**

- We want to implement a simpler version of a HeavyWeight object.
- We don't need the whole functionality of the HeavyWeight object.
- We want to limit the access to the HeavyWeight object.

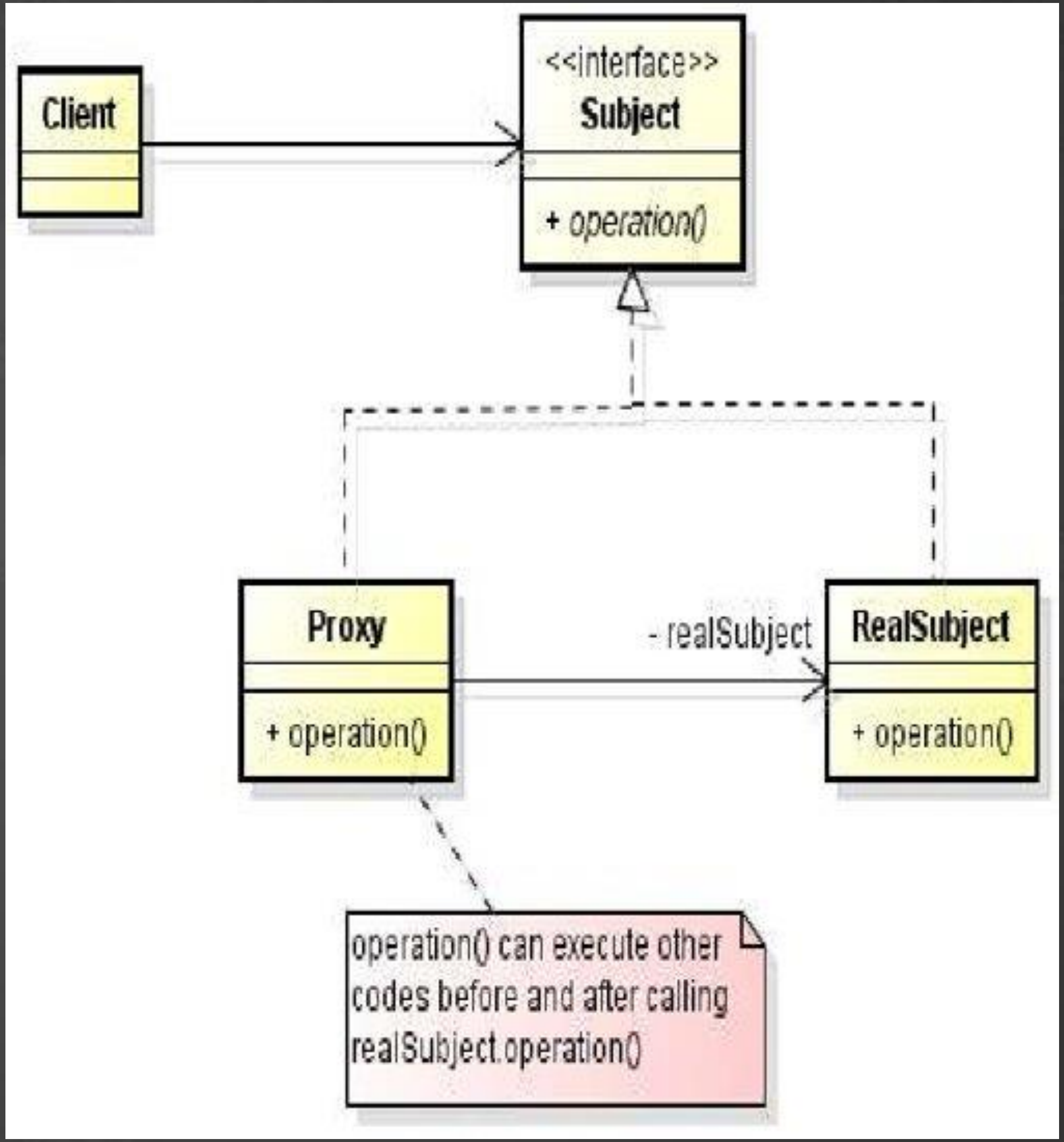
- **Because there may be a time-delay or complex mechanism in creating instances of HeavyWeight objects.**



- Heavy
- Needs many resources to create

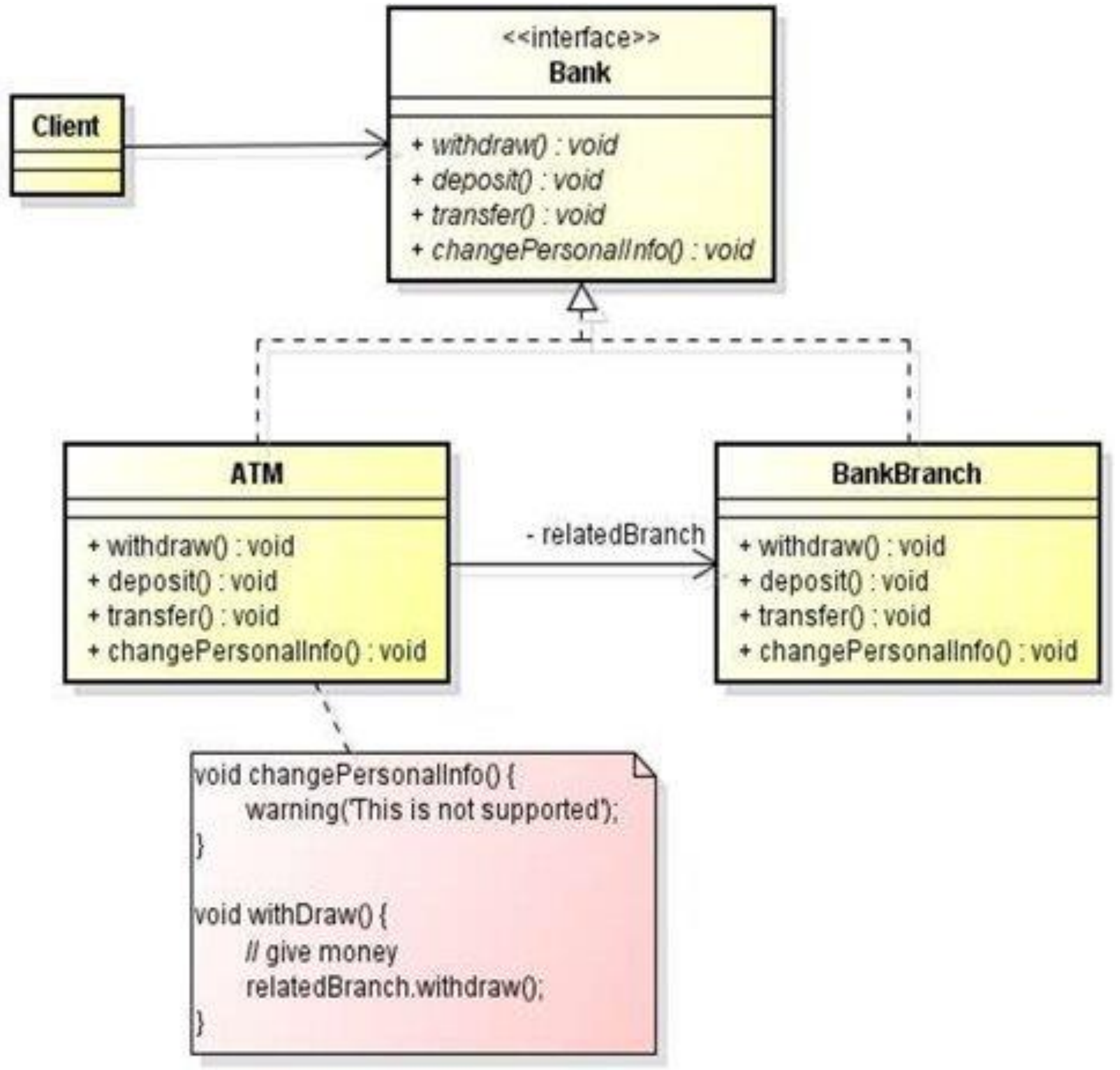
- Light weighted
- Able to do many works of A
- A' is proxy of A

Proxy Pattern Structure



- Drawback money from the bank!
- We will not go to a Bank. Just go to an ATM. Get the cash!
- So ATM is some kind of a bank. With reduced functionalities. ATM is also a proxy to the bank!

REAL LIFE SCENARIO



```
void changePersonalInfo() {
        warning('This is not supported');
}

void withDraw() {
        // give money
        relatedBranch.withdraw();
}
```

TYPES OF PROXY

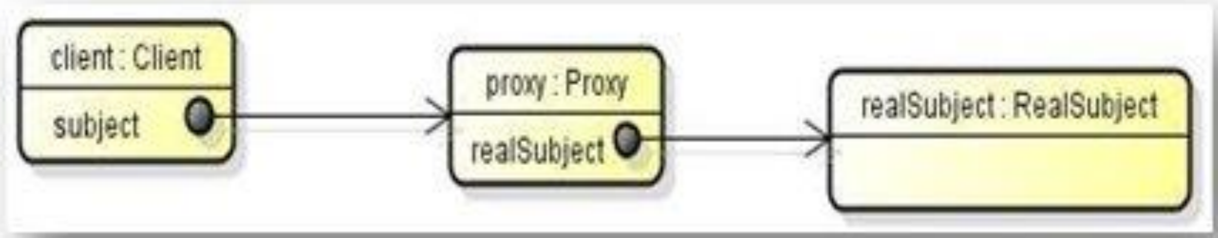
Remote Proxy : Provides a local represent of an object in different address space

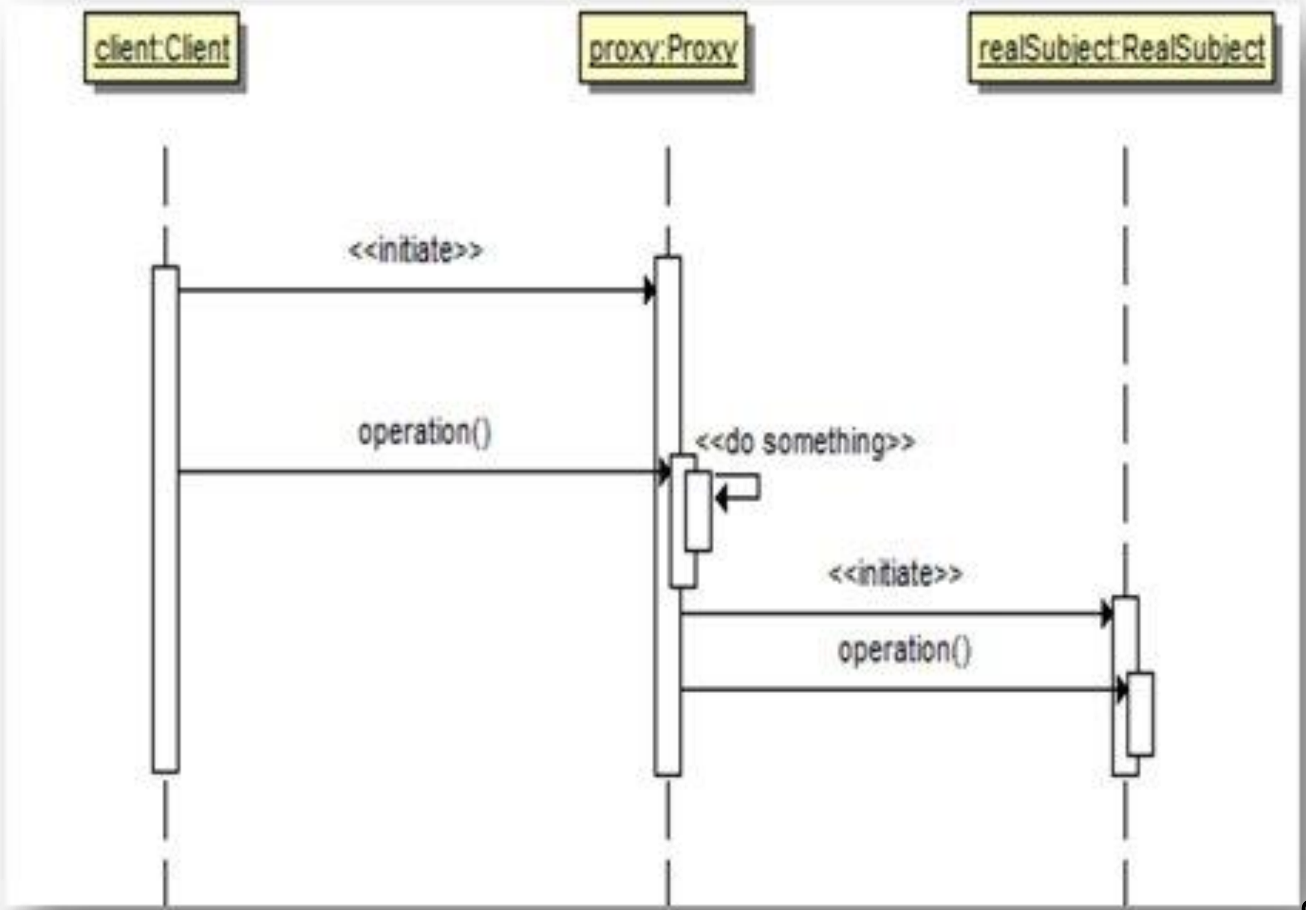
Virtual Proxy: Creates expensive objects on demand

Protection Proxy: Controls access to the original object

Smart Proxy: Replacement for a base pointer that performs additional actions.

OBJECT & SEQUENCE DIAGRAM





PROS

1) ACCESS OF MAIN OBJECT

2) LESS MEMORY

3) FAST

CONS

- Identity Comparison
- Inappropriate change of response
- Obstruction in the identification of overload

Facade Design Pattern

Prepared by,
Morsalina
Id : 180041103

What is Design Pattern ?

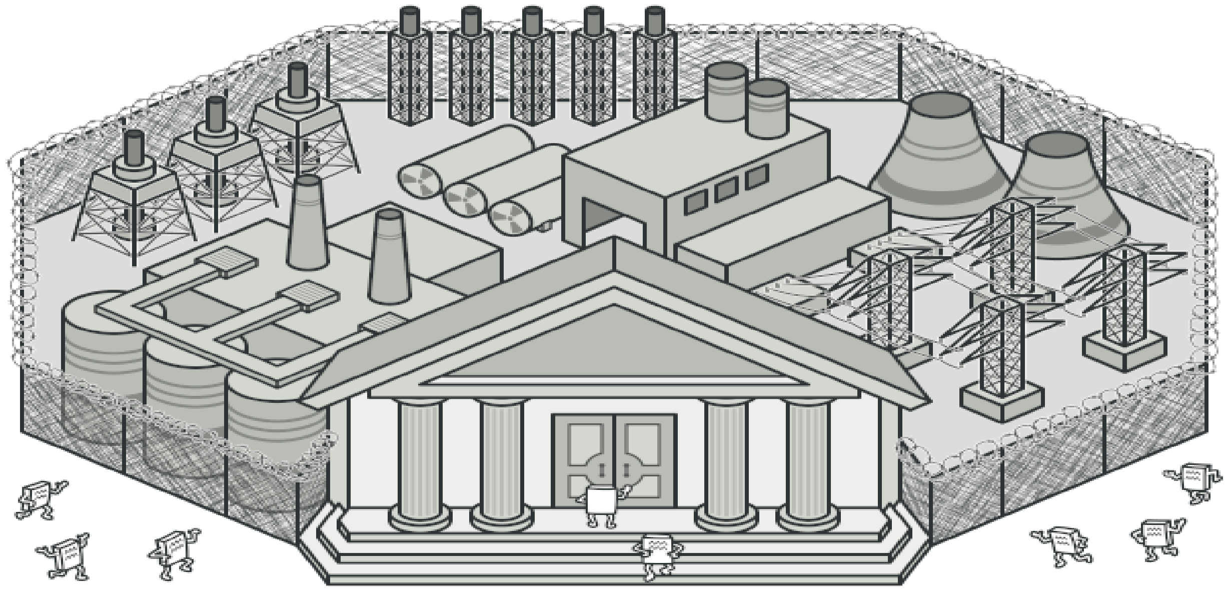
- Design patterns are *typical solution* to some common problems in software design.
- It is like a blueprint.

Why do we need Design Pattern ?

- Patterns are like toolkit of solution to common problems in software design.
- Helps a team to communicate more efficiently.

Facade - Introduction

Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.



Problem

- Imagine that you must make your code work with a broad set of objects that belong to a sophisticated library or framework

- 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

- ➔ Facade provides a simpler interface to a complex subsystem.
- ➔ It includes only those features that the clients are interested in.
- ➔ Facade is handy when you need small number of features.

Let’s consider ,

An app that uploads funny videos on social media. It is an example of Facade design pattern.

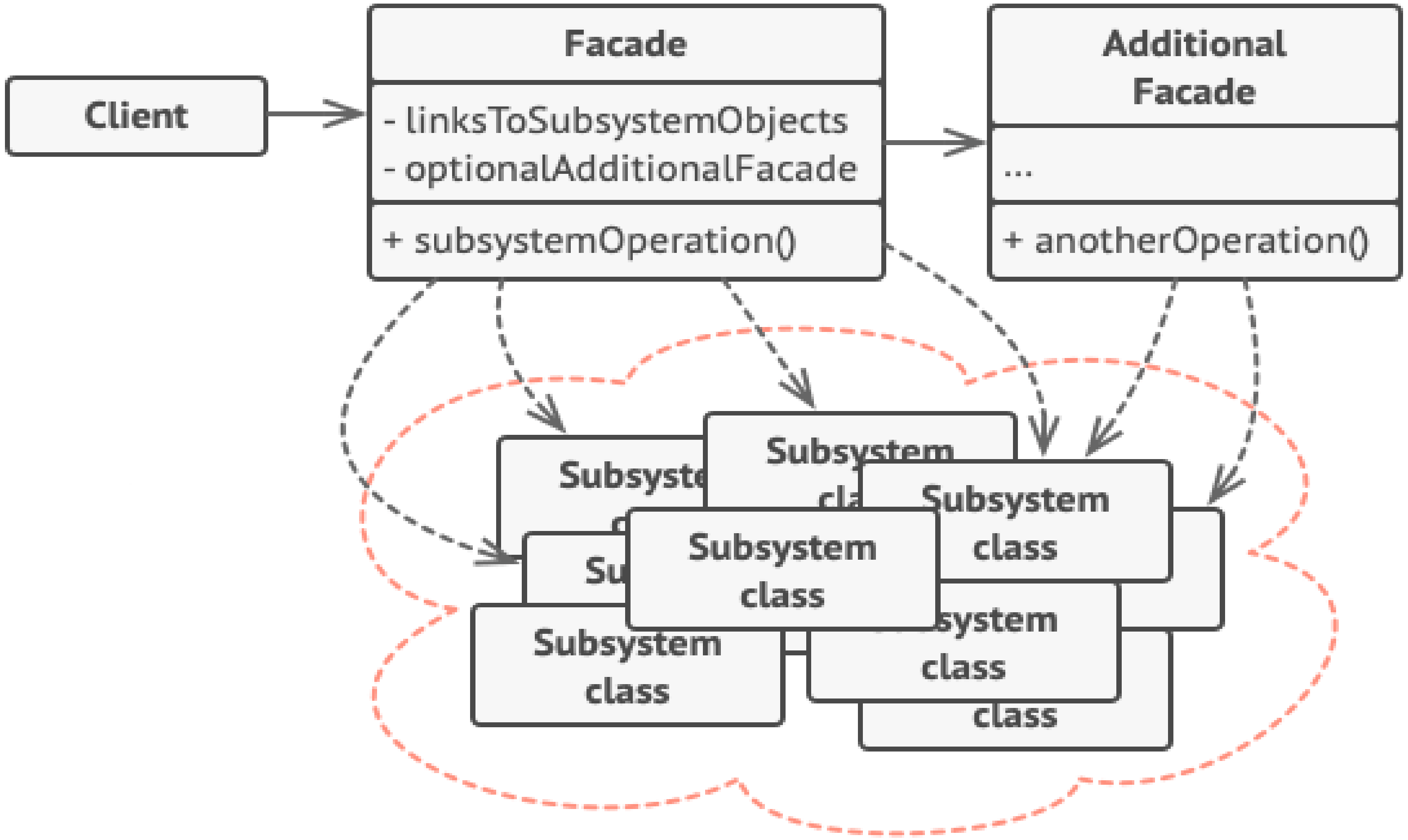
but, **HOW ??**

Real-life Analogy

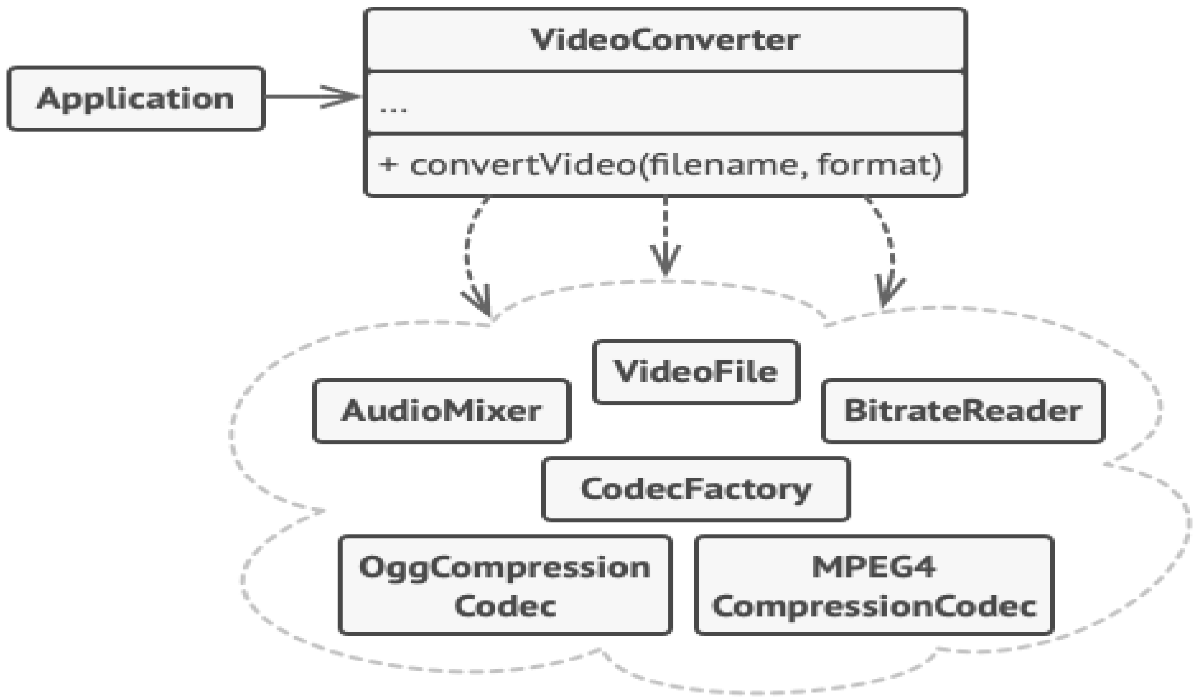
Suppose, you have gone to a restaurant and ordered a cup of coffee. After a few moment, you get the ordered coffee.

In this scenario, we can consider you as a client who wants a feature that can be compared to a cup of coffee. But preparing a cup of coffee requires some more sub tasks and the clients are not concerned about this.

Structure


Client
Facade
- linksToSubsystemObjects
- optionalAdditionalFacade
+ subsystemOperation()
Additional Facade
...
+ anotherOperation()
Subsystem class
Subsystem class
Subsystem class
Subsystem class
Subsystem class
Subsystem class

Example and Pseudocode



An example of isolating multiple dependencies within a single facade class.

```
class VideoConverter is
    method convert(filename, format):File is
        file = new VideoFile(filename)
        sourceCodec = new CodecFactory.extract(file)
        if (format == "mp4")
            destinationCodec = new MPEG4CompressionCodec()
        else
            destinationCodec = new OggCompressionCodec()
        buffer = BitrateReader.read(filename, sourceCodec)
        result = BitrateReader.convert(buffer, destinationCodec)
        result = (new AudioMixer()).fix(result)
        return new File(result)

// Application classes don't depend on a billion classes
// provided by the complex framework. Also, if you decide to
// switch frameworks, you only need to rewrite the facade class.
class Application is
    method main() is
        convertor = new VideoConverter()
        mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
        mp4.save()
```

Pros:

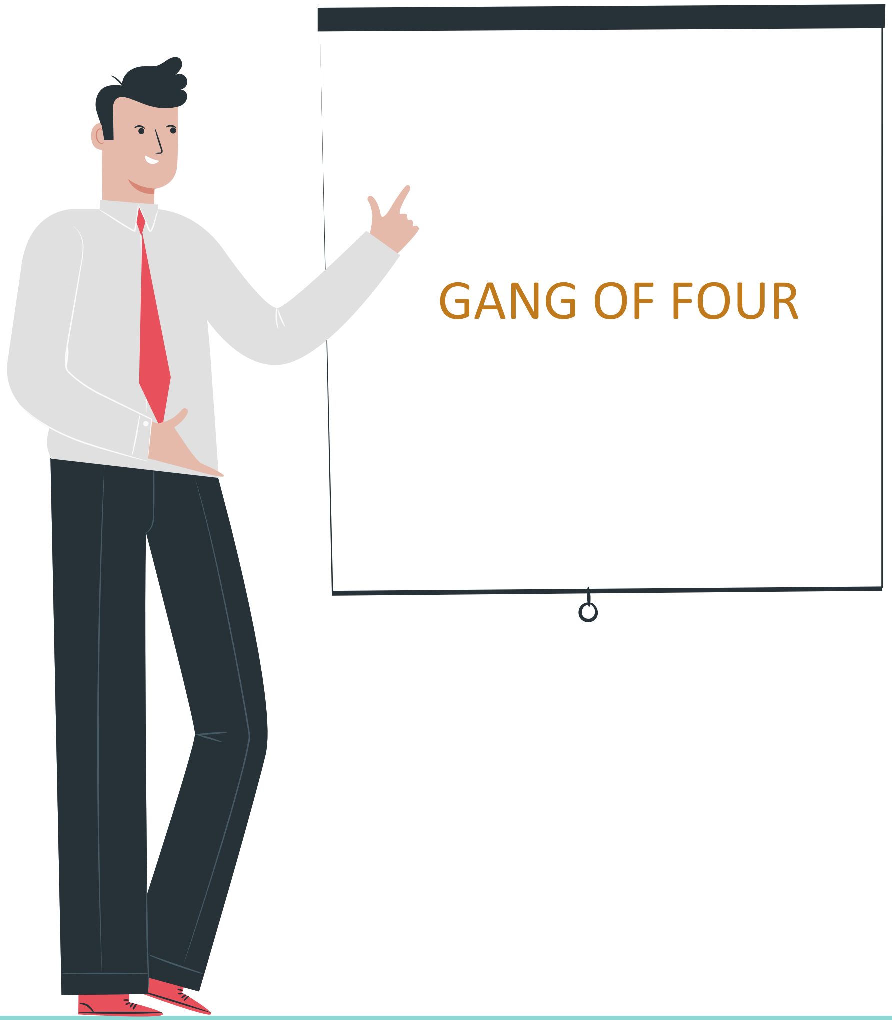
- You can isolate your code from the complexity of a subsystem
- Makes the interface simple to clients.

Cons:

- A facade can become **a god object** coupled to all classes of an app.

ADAPTER PATTERN

Shams Tanveer Jim
180041107

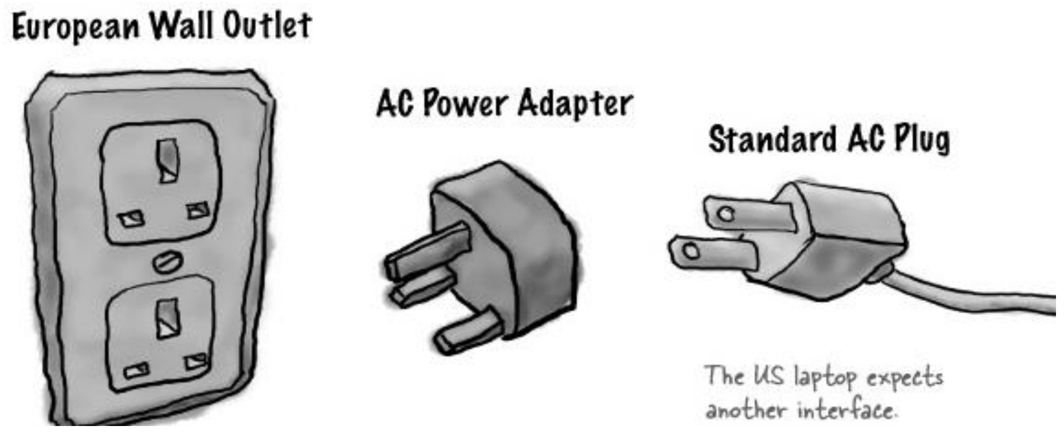


Intent of Adapter

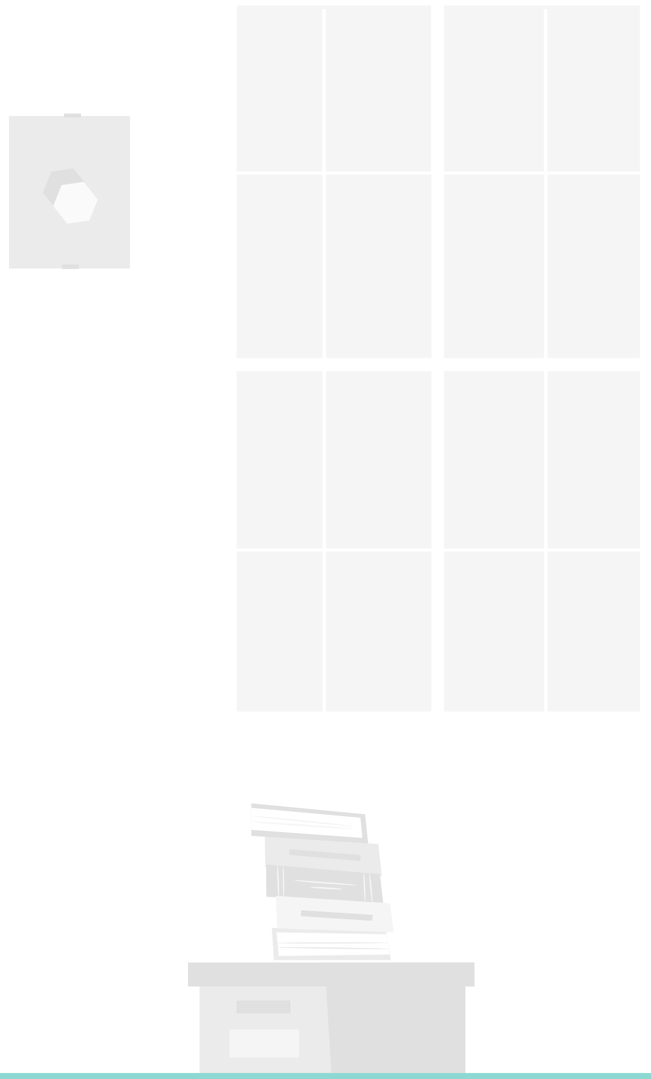
Convert the interface of a class into another interface that the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

WHAT IS ADAPTER PATTERN

Adapter is a structural design pattern, which acts as a wrapper between two incompatible objects, to make them compatible.



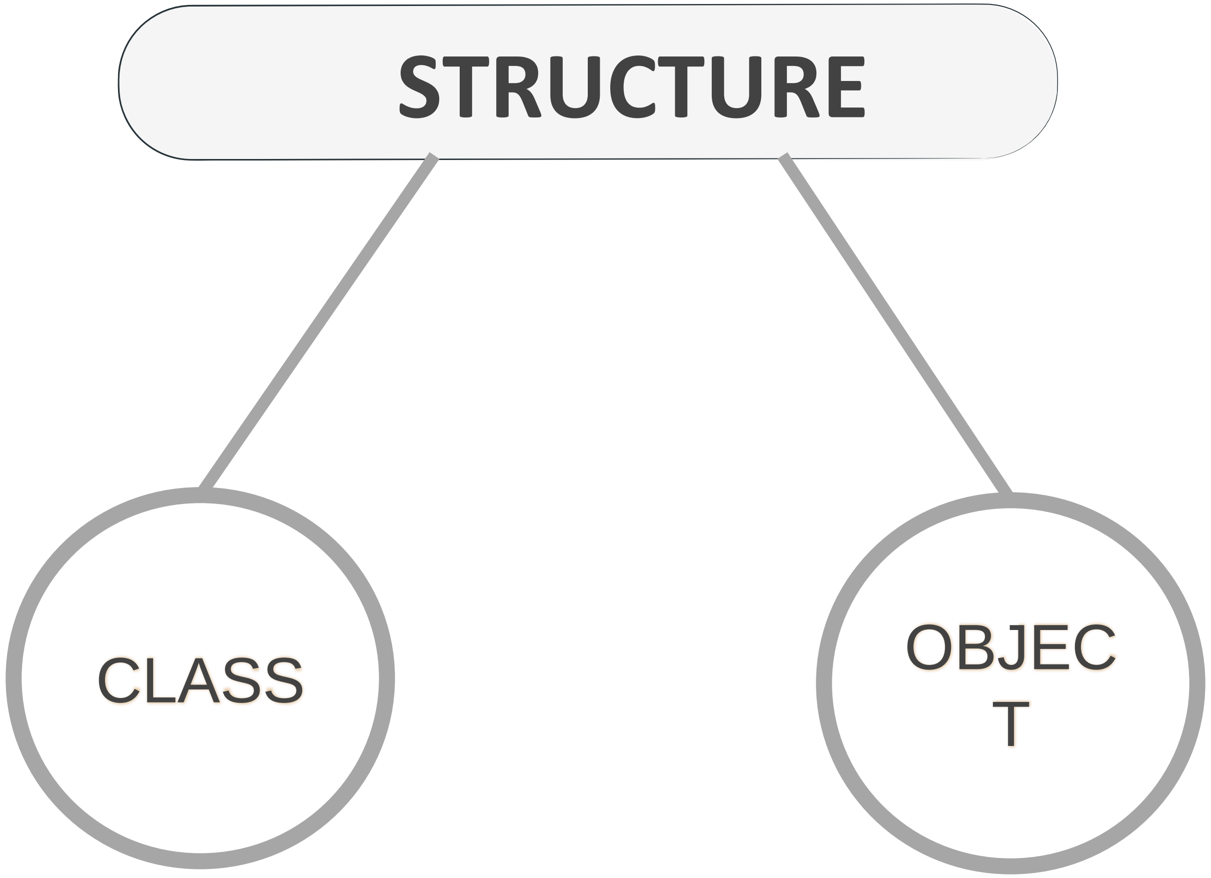
APPLICABILITY



01 Need to use an existing class but it’s interface does not match with what is needed.

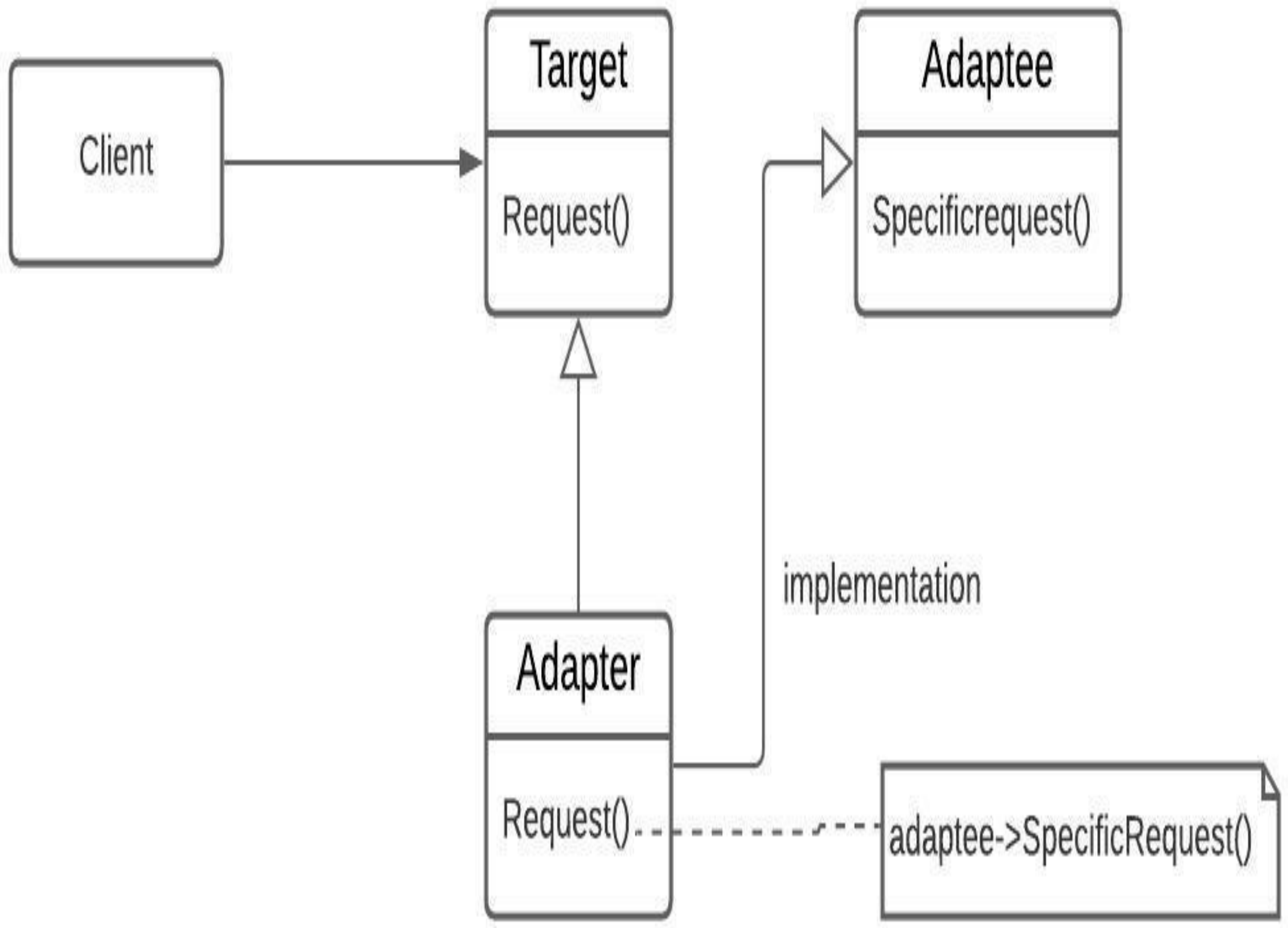
02 Need to create a reusable class that cooperates with unrelated classes with incompatible interfaces

03 Need to convert the programming interface of one class into that of another.


STRUCTURE
CLASS
OBJECT

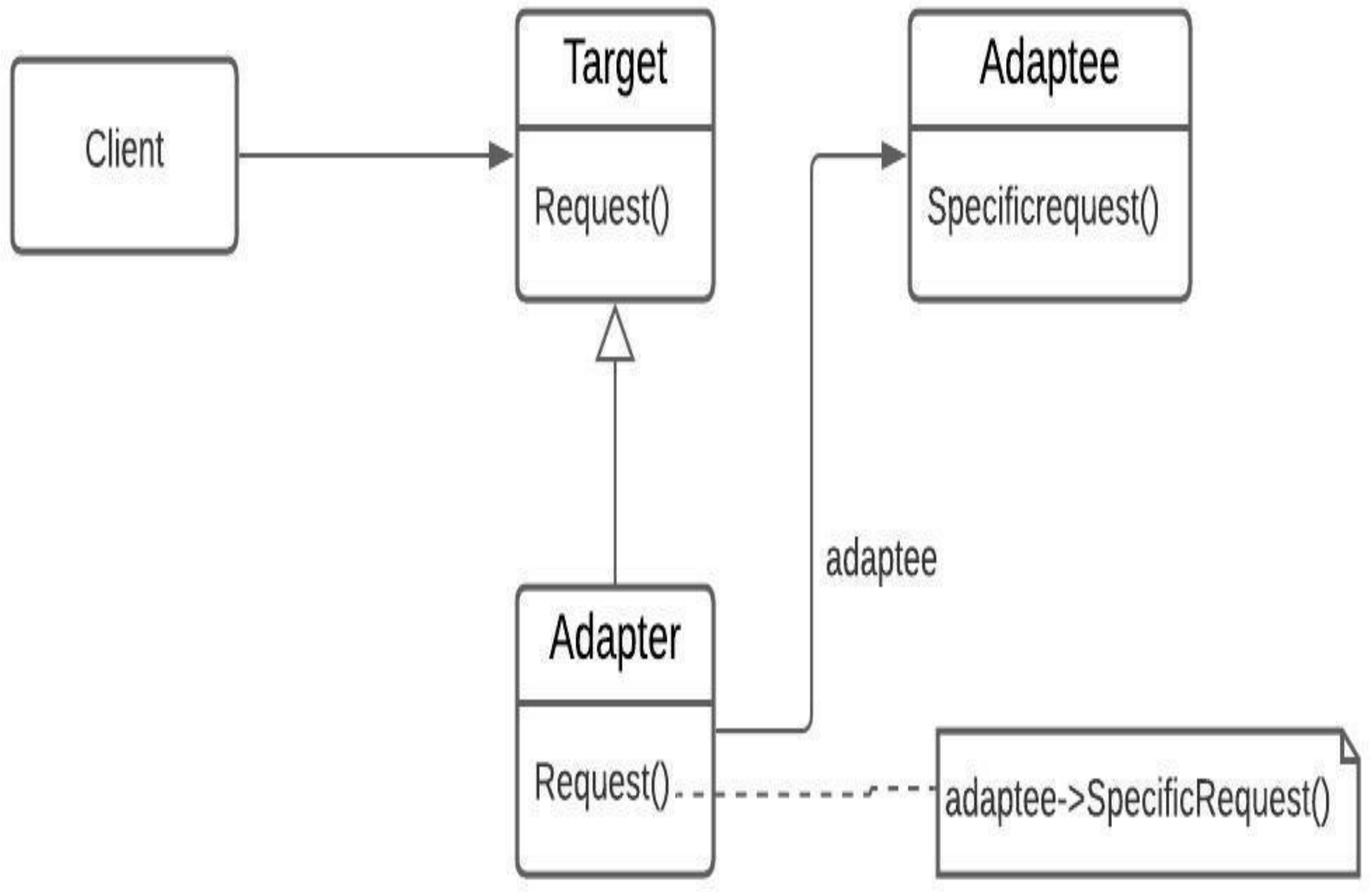
CLASS ADAPTER

- Use multiple inheritance to adapt one interface to another
- Adapter inherits the interface of the client's target and the interface of the adaptee as well.
- Easy to implement in those programming languages which supports multiple inheritance such as C++. For Java, the target or adaptee interfaces could be Java Interfaces.



OBJECT ADAPTER

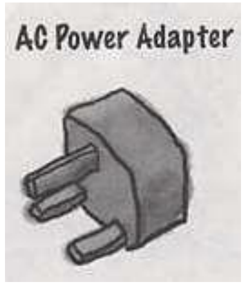
- Use compositional technique to adapt one interface to another
- Adapter inherits the target interface that the client expects to see, while it holds an instance of the adaptee.
- Easy to implement in all popular programming languages



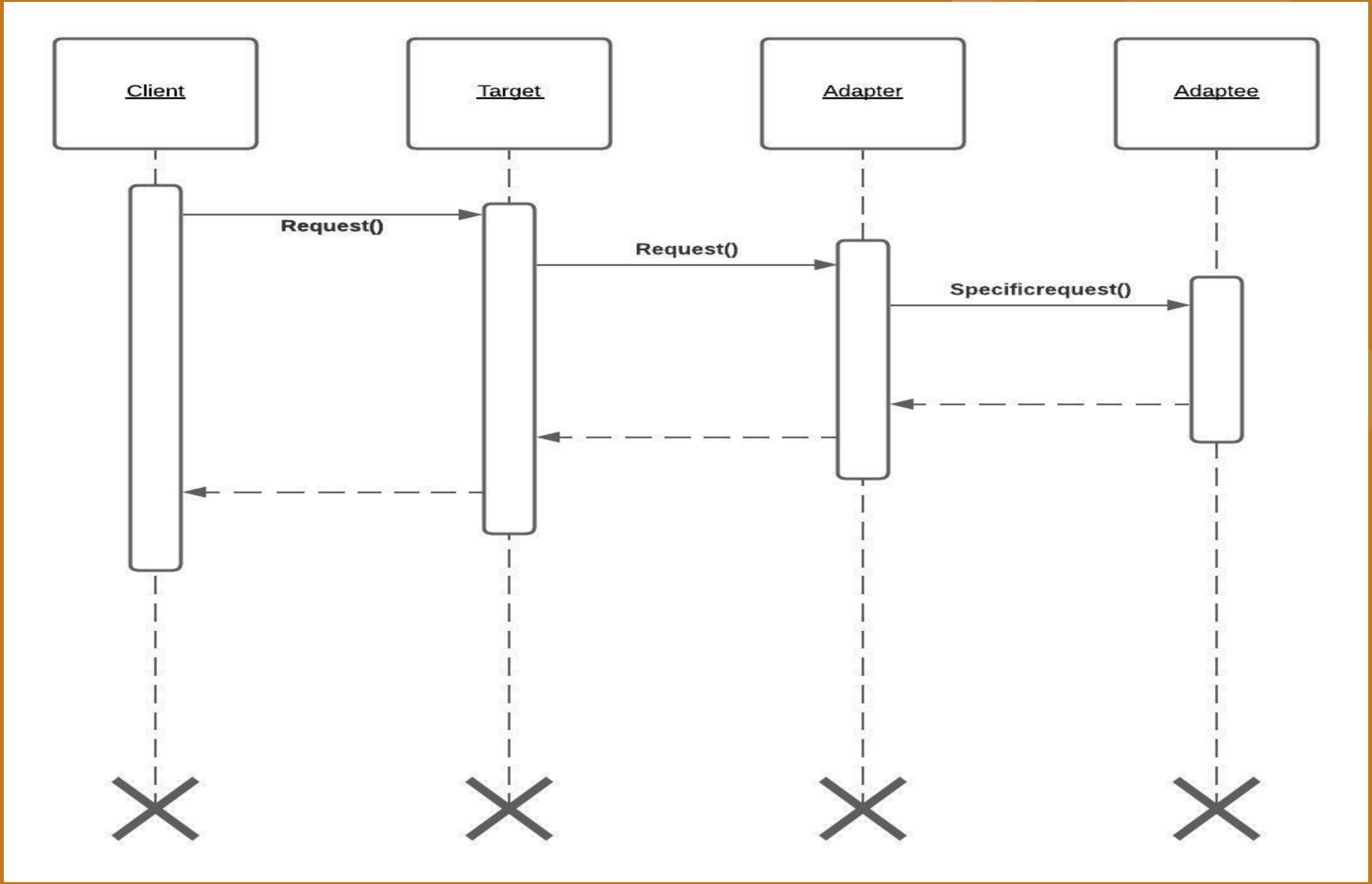
PSEUDOCODE

```
 interface Itarget{
  public void Request
 }
class Client implements Itarget{
 @override
public void Request(){
 code segment
}
 class Adaptee{
  public void Specificrequest()
     Code Segment
 }
 }
```

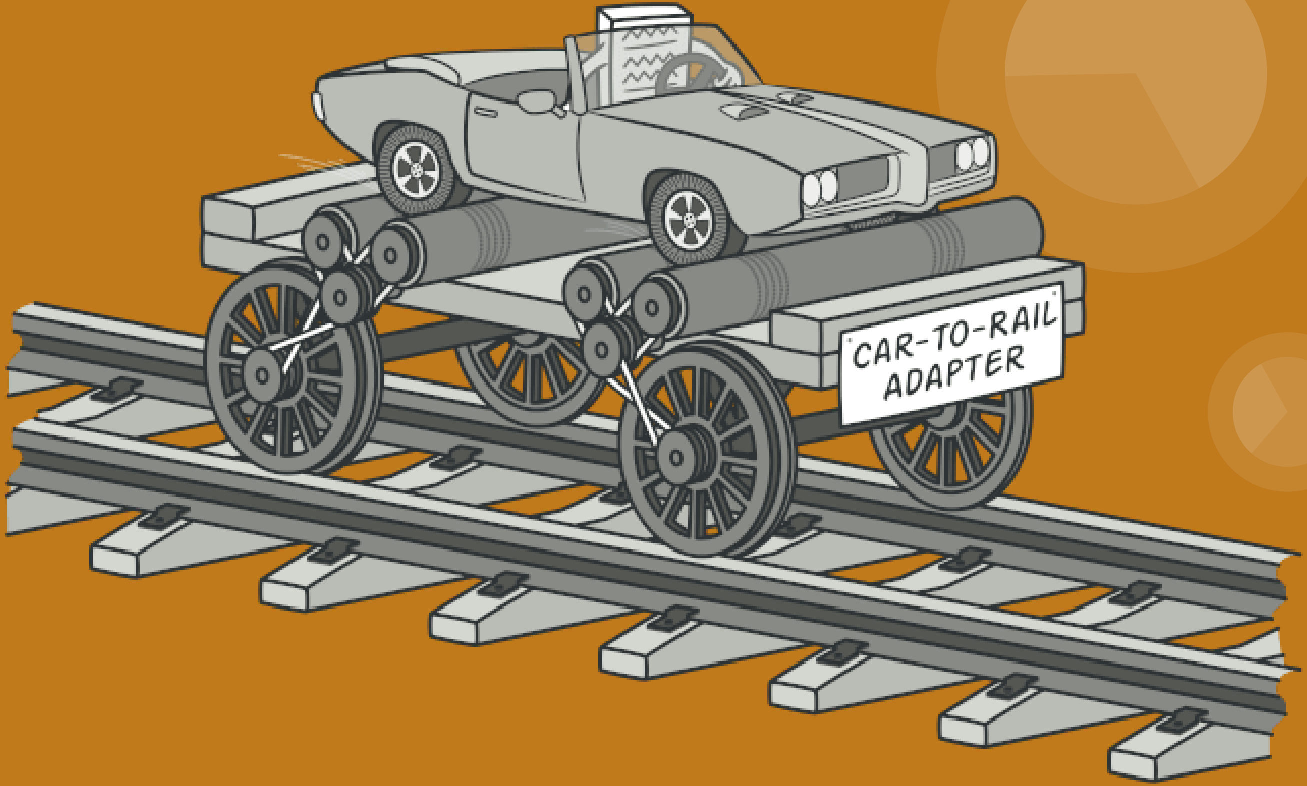
```
 class Adapter implements
Itarget{
  Adaptee adaptee;
 public Adapter(Adaptee
adptee){
  this.adaptee= adptee;
}
@override
 public void Request(){
 adaptee.Specificrequest();
}
```



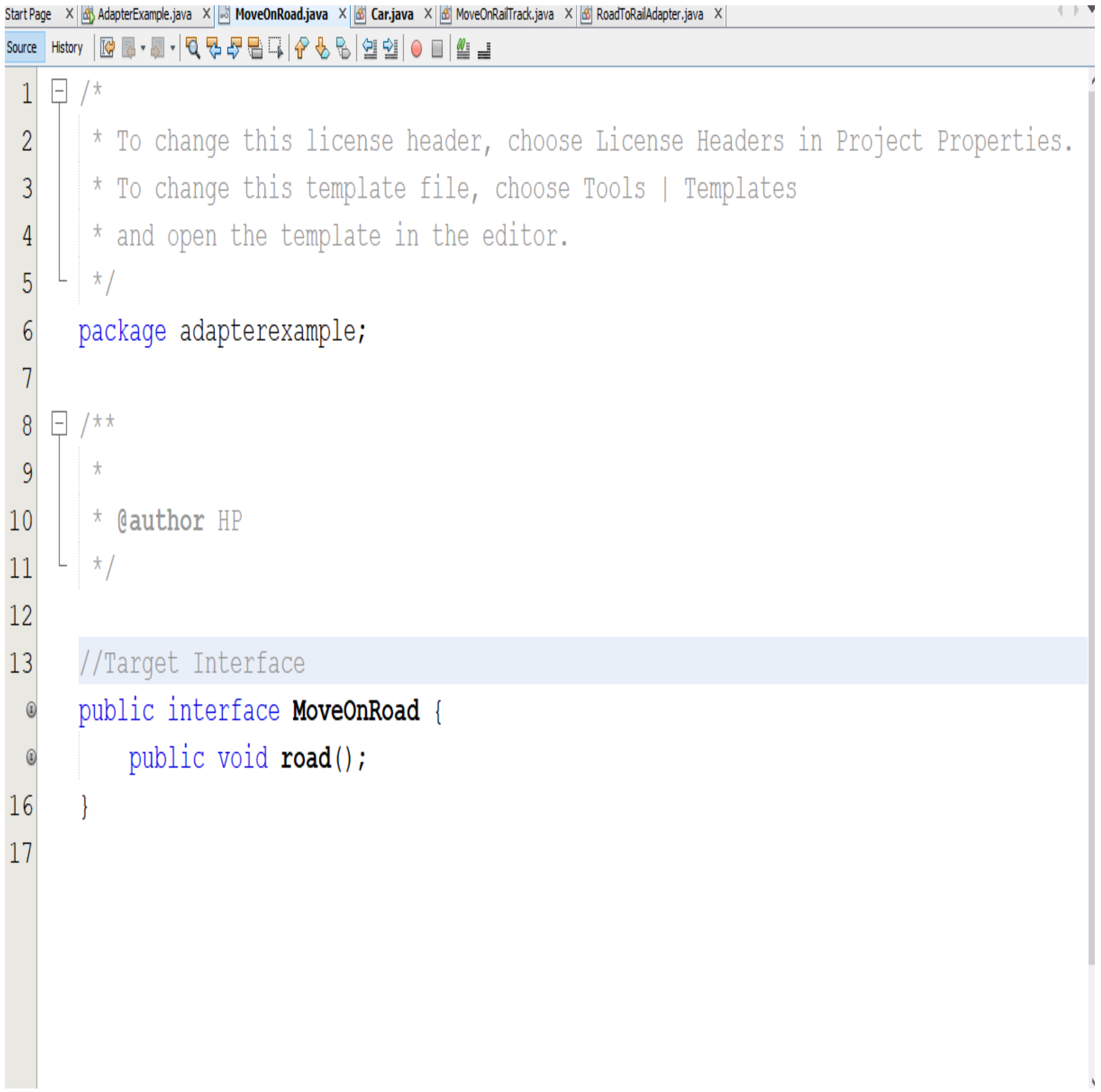
SEQUENCE DIAGRAM



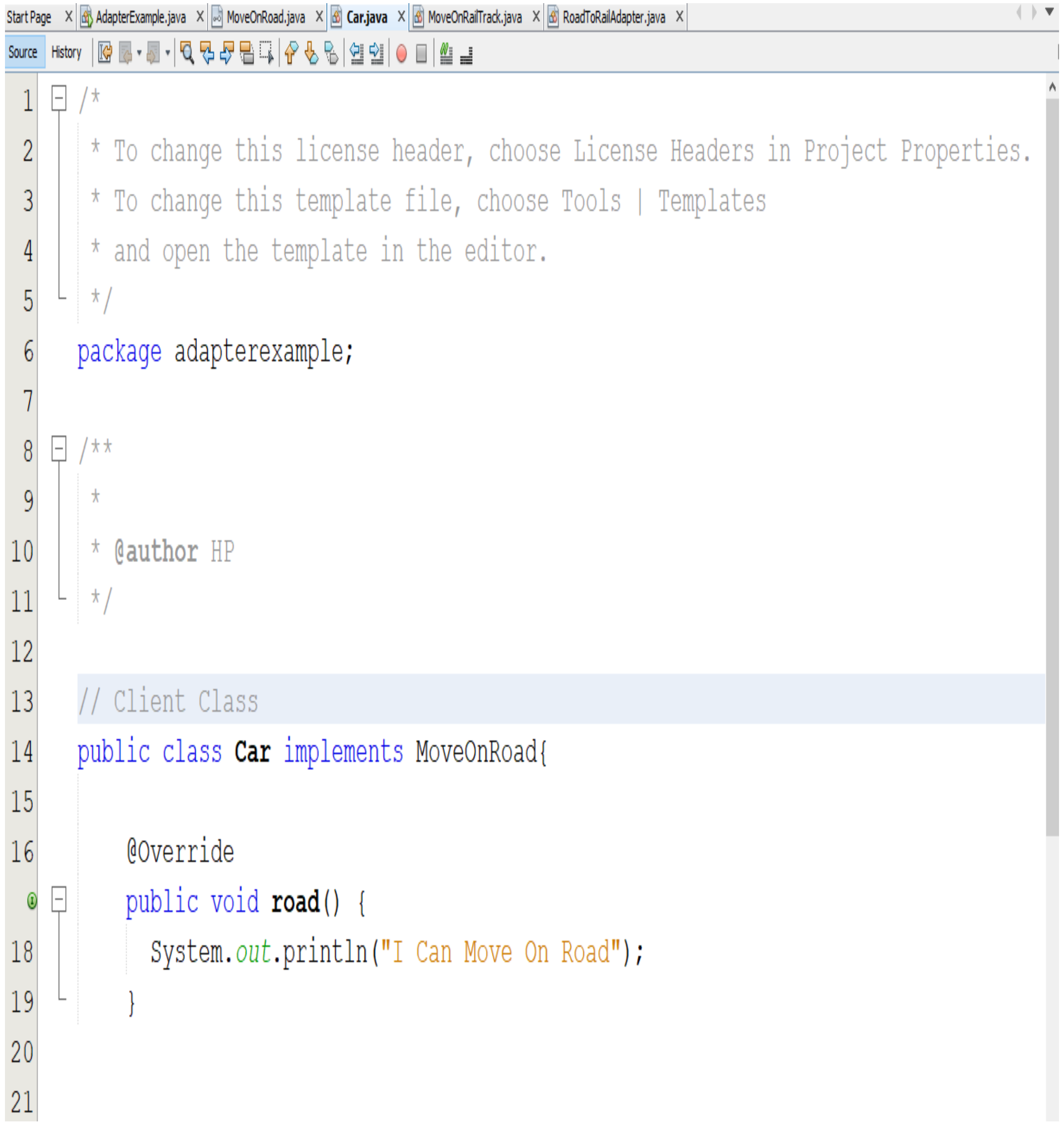
EXAMPLE



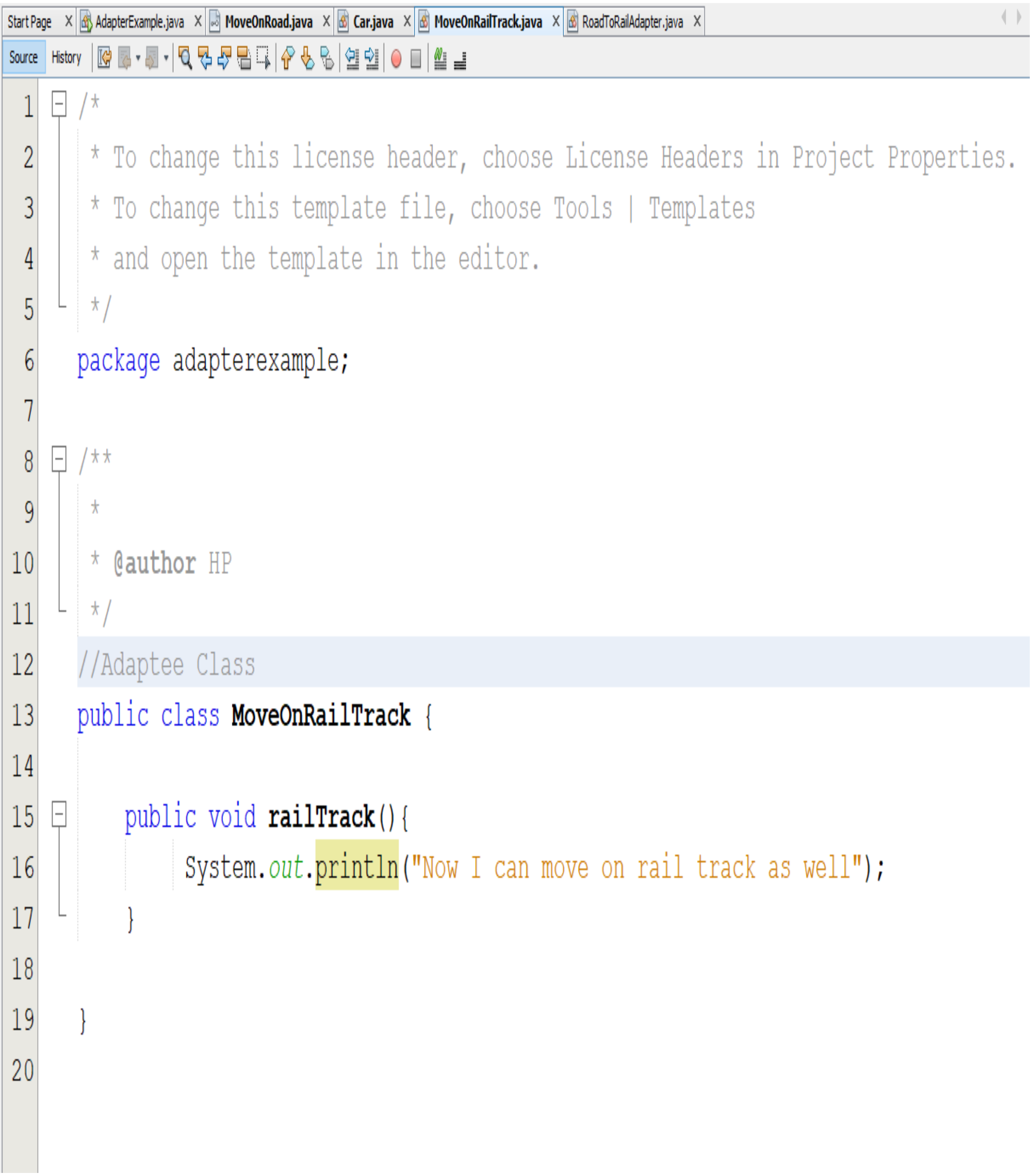
Example In Java



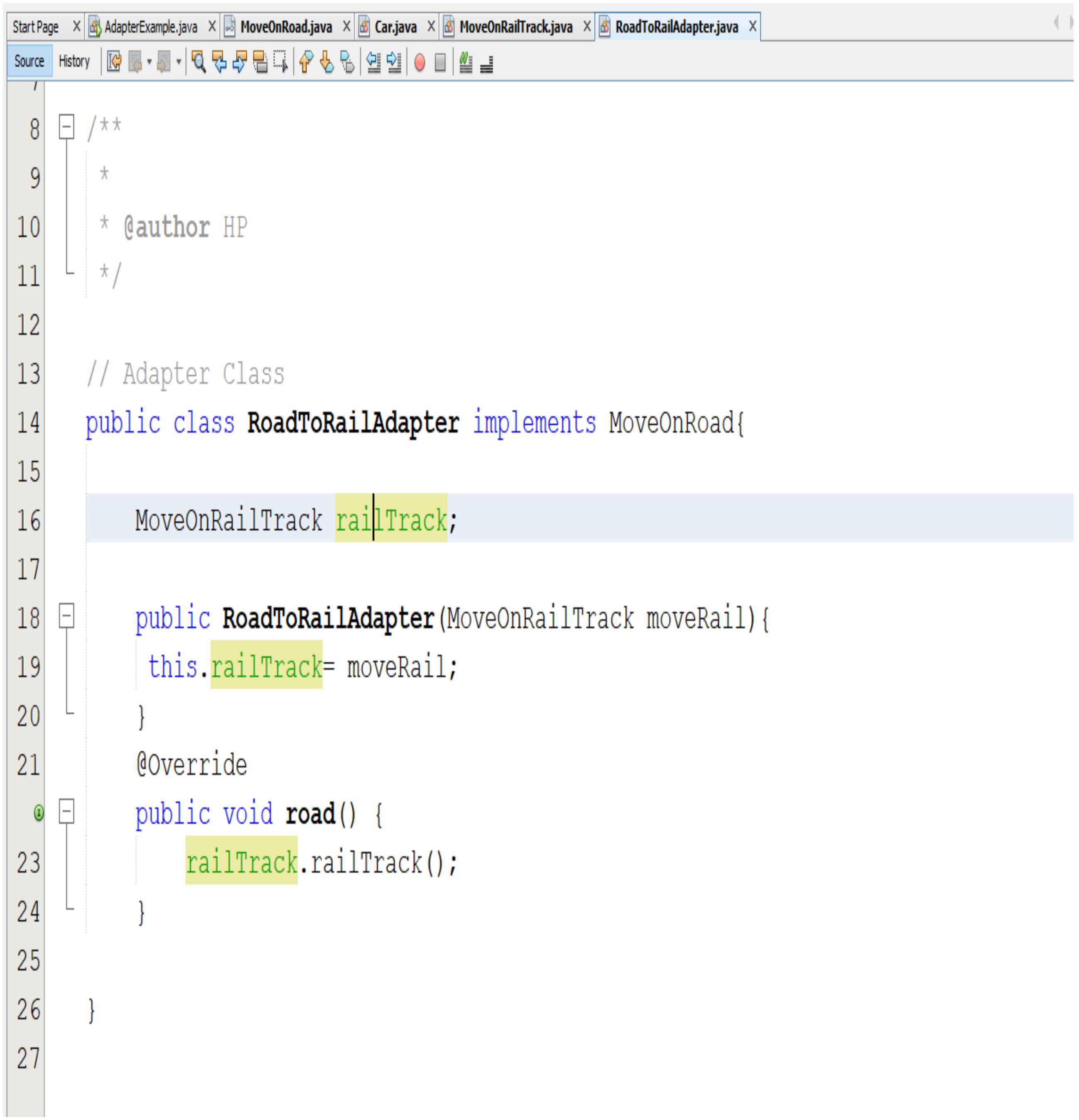
```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package adapterexample;

/**
 *
 * @author HP
 */

//Target Interface
public interface MoveOnRoad {
    public void road();
}
```



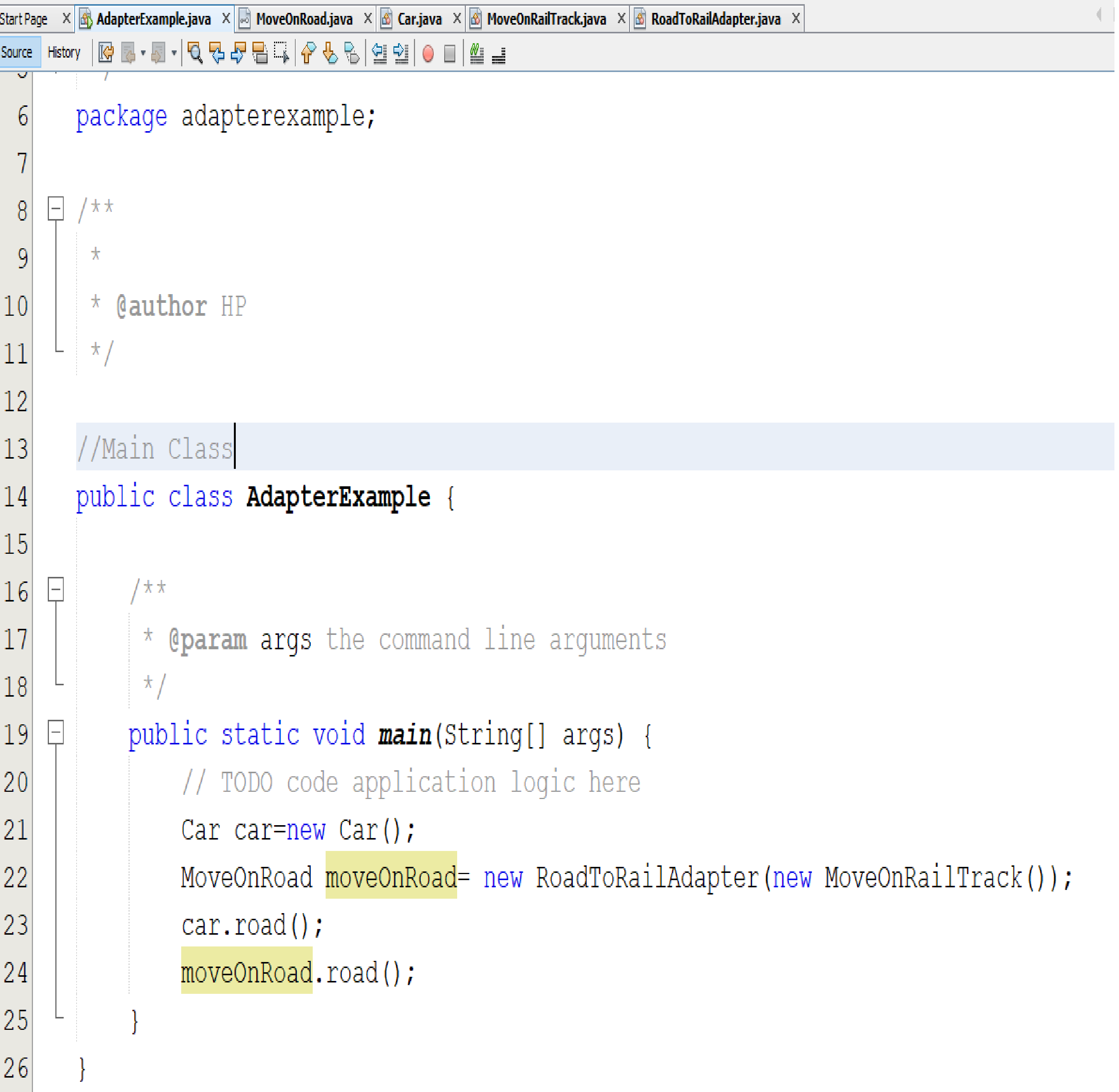
```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package adapterexample;

/**
 *
 * @author HP
 */

// Client Class
public class Car implements MoveOnRoad{

    @Override
    public void road() {
      System.out.println("I Can Move On Road");
    }


```



```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package adapterexample;

/**
 *
 * @author HP
 */
//Adaptee Class
public class MoveOnRailTrack {

    public void railTrack(){
         System.out.println("Now I can move on rail track as well");
    }

}
```

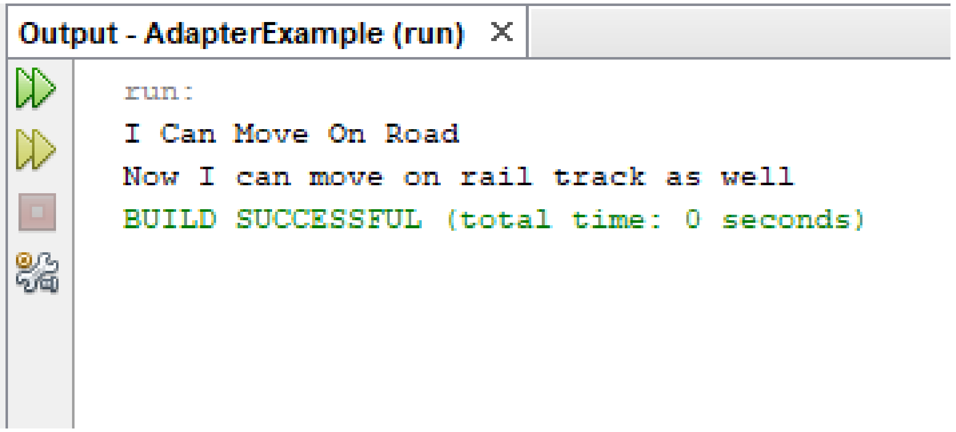


```java
/**
 *
 * @author HP
 */

// Adapter Class
public class RoadToRailAdapter implements MoveOnRoad{

    MoveOnRailTrack railTrack;

    public RoadToRailAdapter(MoveOnRailTrack moveRail){
     this.railTrack= moveRail;
    }
    @Override
    public void road() {
        railTrack.railTrack();
    }

}
```


Start Page | AdapterExample.java | MoveOnRoad.java | Car.java | MoveOnRailTrack.java | RoadToRailAdapter.java

Source History

```java
package adapterexample;

/**
 *
 * @author HP
 */

//Main Class
public class AdapterExample {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        Car car=new Car();
        MoveOnRoad moveOnRoad= new RoadToRailAdapter(new MoveOnRailTrack());
        car.road();
        moveOnRoad.road();
    }
}
```


Output - AdapterExample (run)

```
run:
I Can Move On Road
Now I can move on rail track as well
BUILD SUCCESSFUL (total time: 0 seconds)
```