# Design Patterns with java

# What are Design Patterns?

- Design patterns are optimized, reusable solutions to the programming problems
- A design pattern is not a class or a library that we can simply plug into our system
- It is a template that has to be implemented in the correct situation

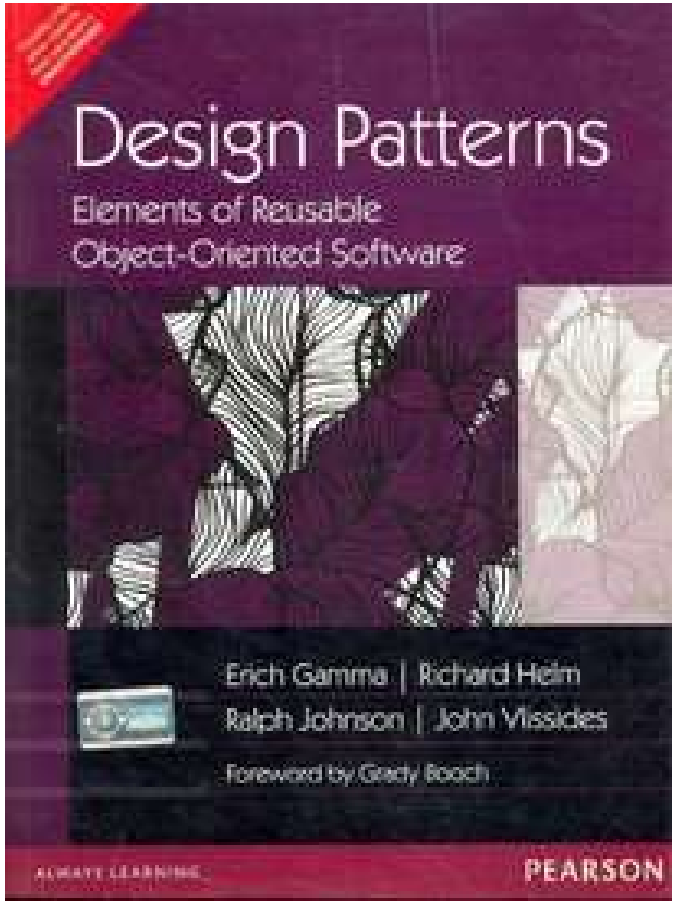
- It’s not language-specific either
- A good design pattern should be implementable in most of the languages, depending on the capabilities of the language

# What are Design Patterns?

- Most importantly, any design pattern can be a double-edged sword
- If implemented in the wrong place, it can be disastrous and create many problems
- However, implemented in the right place, at the right time, it can be a boon
- Design Pattern are proven solutions approaches to specific problems

# Gang of Four

- The *Gang of Four* **(GOF or GO4)** are the authors of the book, Design Patterns: Elements of Reusable Object-Oriented Software
- This book describes various development techniques and pitfalls in addition to providing twenty-three O O programming design patterns
- The book's authors are
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides



# Usage of Design Patterns

- **Common language for developers:**

  They provide developer a common language for certain problems. For example if a developer tells another developer that he is using a Singleton, the another developer (should) know exactly what this means

- **Capture best practices:**

  Design patterns capture solutions which have been applied to certain problems. By learning these patterns and the problem they are trying to solve a un-experienced developer can learn a lot about software design.

# Quick Review of Java (OOP)

- Classes and Objects
- Class Structure
- Communication between objects
- Why object creation is dynamic
- Inheritance (IS-A relationship)
- Polymorphism
- Composition (HAS-A relationship)
- Inheritance VS Composition
- Difference between abstract class and interface
- Interface based application design
- Multi layer applications

# Types of design Patterns

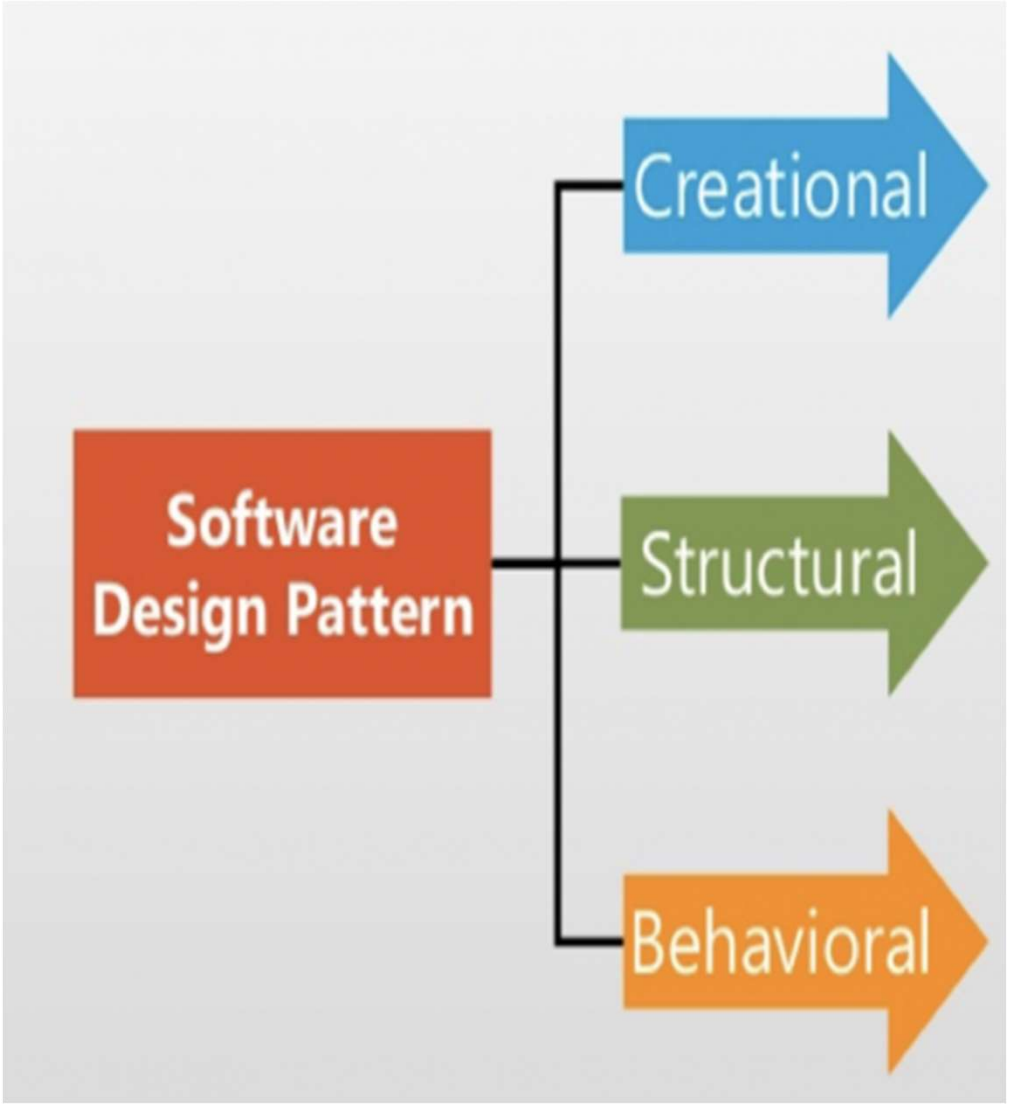
- There are three basic types of design patterns:
  - **Creational**
  - **Structural**
  - **Behavioral**

- **Creational** patterns provide ways to instantiate single objects or groups of related objects.

- **Structural** patterns provide manner to define relationships between objects, making it easier for these entities to work together

- **Behavioral** patterns define the way of communication between objects

# Types of design Patterns

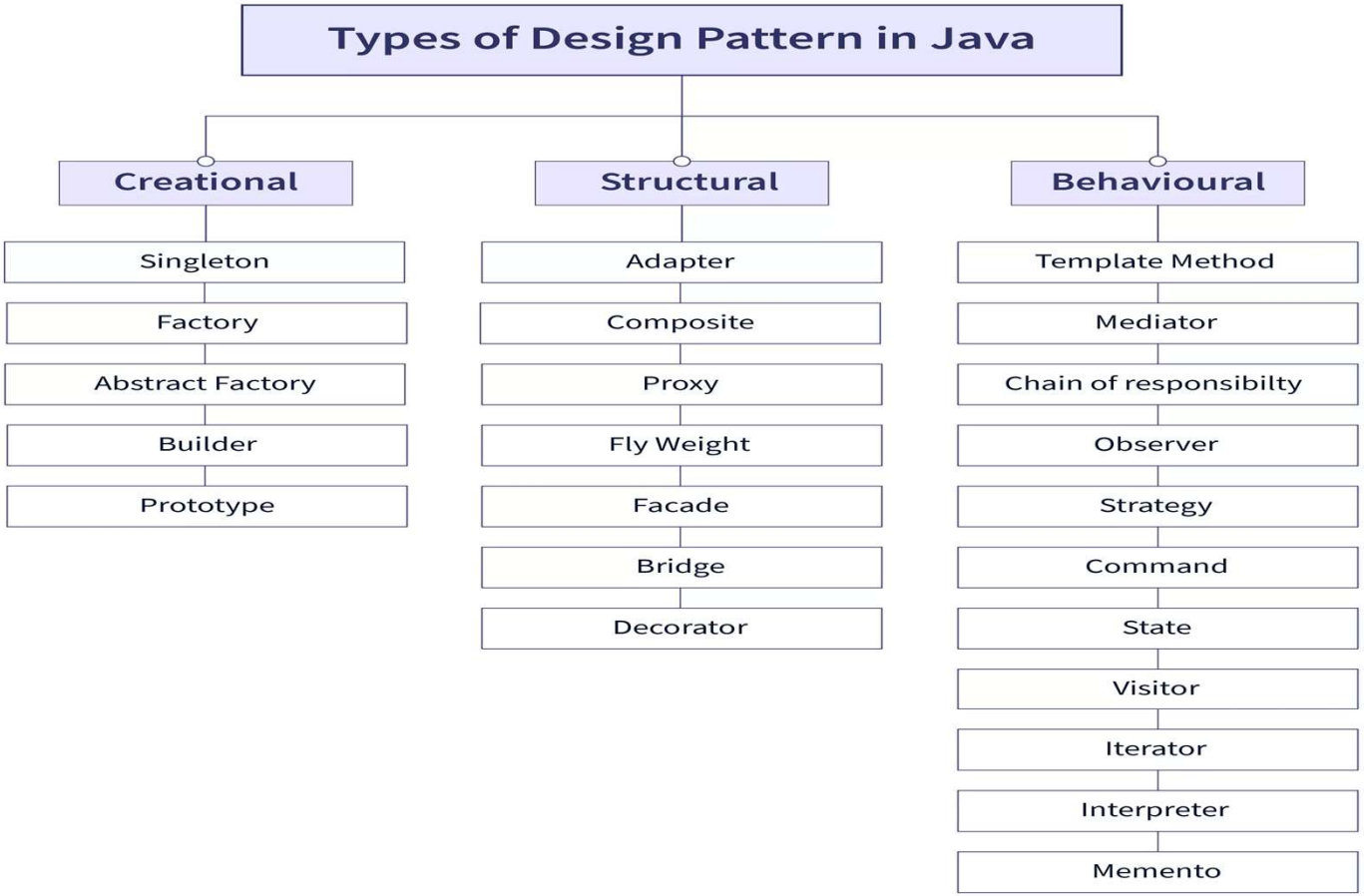



to create objects in a controlled manner suitable to the situation

to assemble objects and classes into larger structures while keeping these structures flexible and efficient

to ensure effective communication between components

# Types of design Patterns



# Creational Patterns

# Creational Patterns

- *Creational patterns prescribe the way the objects are created*
- *These patterns are used* when a decision must be made at the time a class is instantiated
- The details of the classes that are instantiated ( what exactly they are, how, and when they are created ) are hidden from the client class
- Client class knows only about the abstract class or the interface it implements
- The specific type of the concrete class is unknown to the client class

# Creational Patterns

- **Creational Design Patterns**

1. Singleton Pattern
2. Factory Pattern
3. Abstract Factory Pattern
4. Prototype Pattern
5. Builder Pattern

# Singleton Pattern

- A singleton in is a class for which only one instance can be created and provides a global point of access this instance
- The singleton pattern describe how this can be archived
- Singletons are useful to provide a unique source of data or functionality to other Java Objects
- For example, you may use a singleton to access your data model from within your application or to define logger which the rest of the application can use

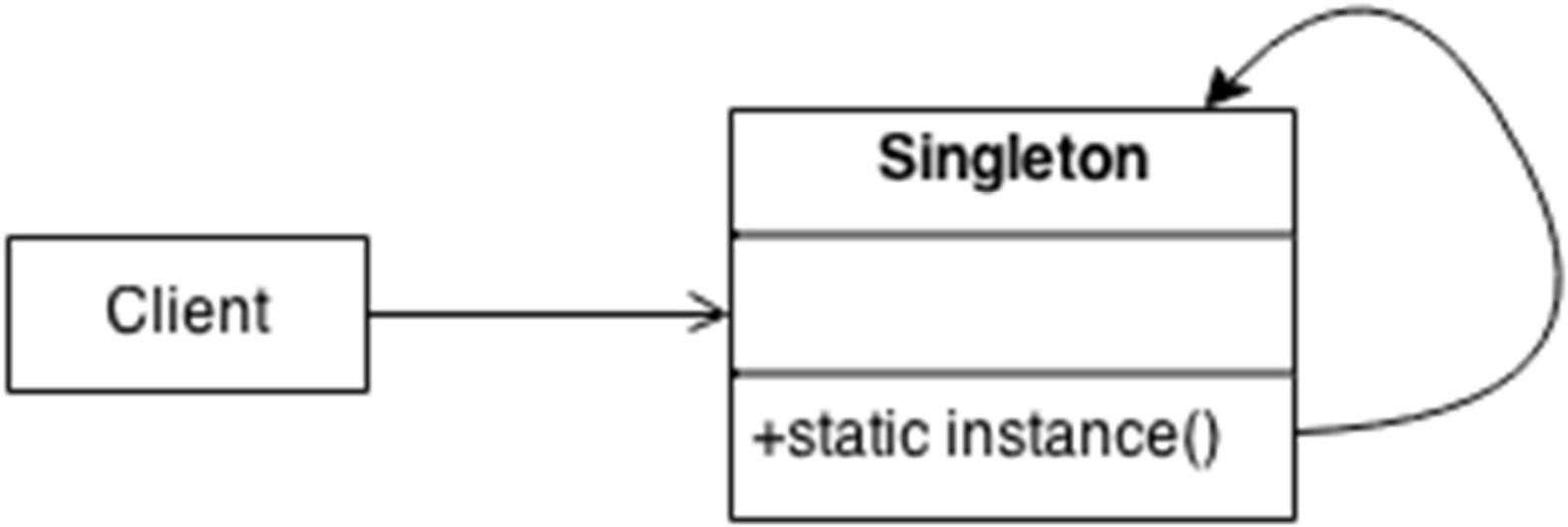
# Singleton Pattern

- A singleton is a class that is instantiated only once
- This is  accomplished by creating a static field in the class representing the class
- A static method exists on the class to obtain the instance of the class and is named such as getInstance()
- The creation of the object referenced by the static field can be done either when the class is initialized or the first time that getInstance() is called
- The singleton class typically has a private constructor to prevent from being instantiated via a constructor
- The instance of the singleton is obtained via the static getInstance() method.

# Singleton Pattern

1. Define a private static attribute in the "single instance" class.
2. Define a public static accessor function in the class.
3. Do "lazy initialization" (creation on first use) in the accessor function.
4. Define all constructors to be protected or private.
5. Clients may only use the accessor function to manipulate the Singleton.

# Singleton Pattern



# Singleton Pattern

Drawbacks

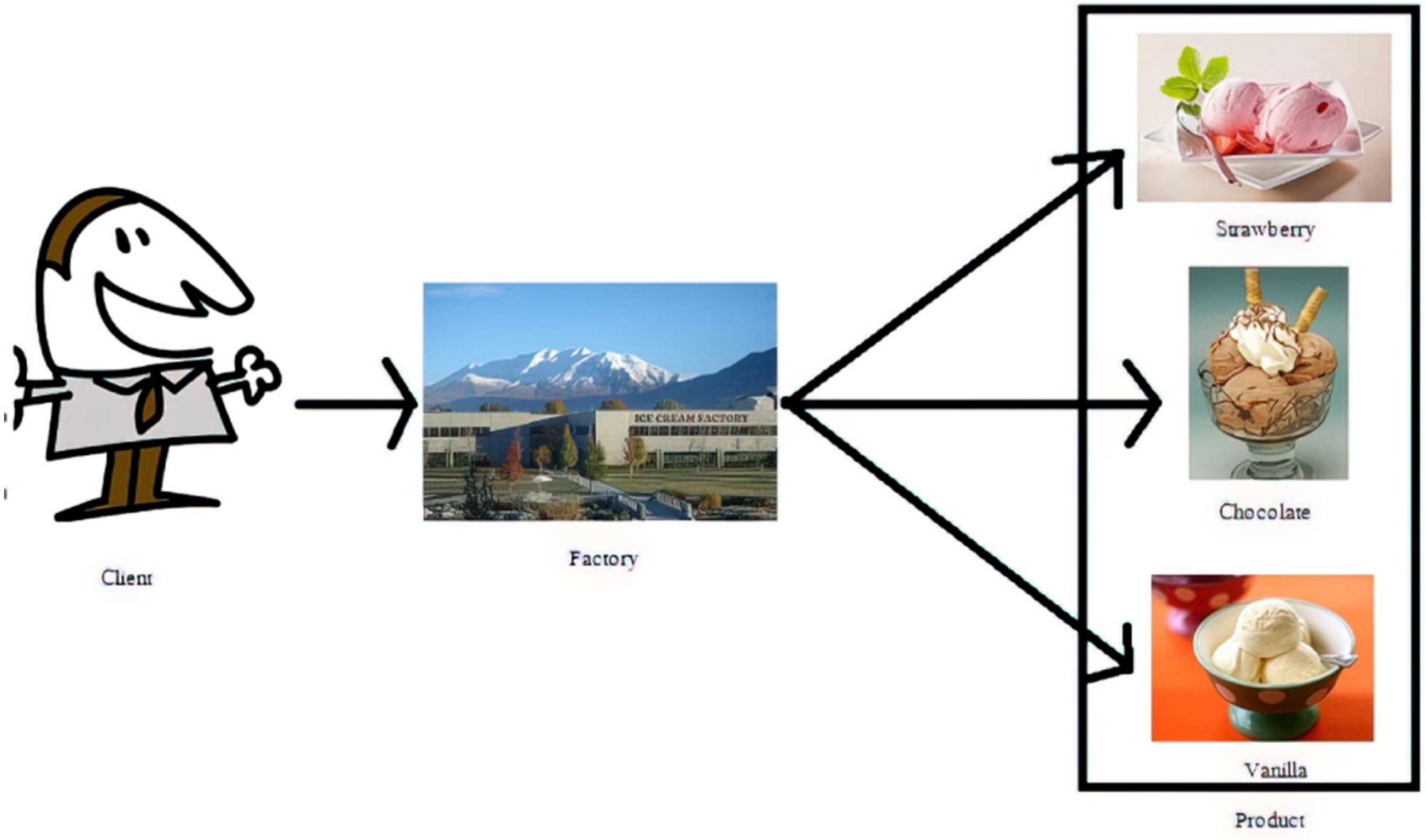
- This pattern reduces the potential for parallelism within a program, because access to the singleton in a multi-threaded context must be synchronized

- Advocates of dependency injection would regard this as an anti-pattern, mainly due to its use of private and static methods

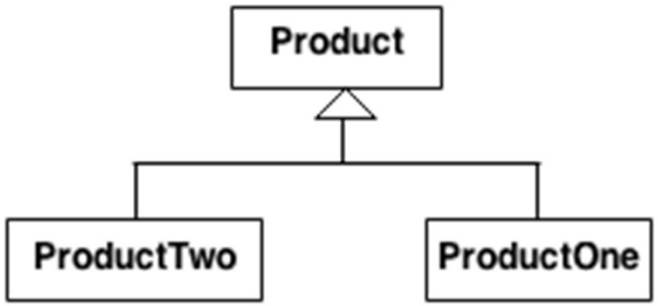
# Factory Pattern

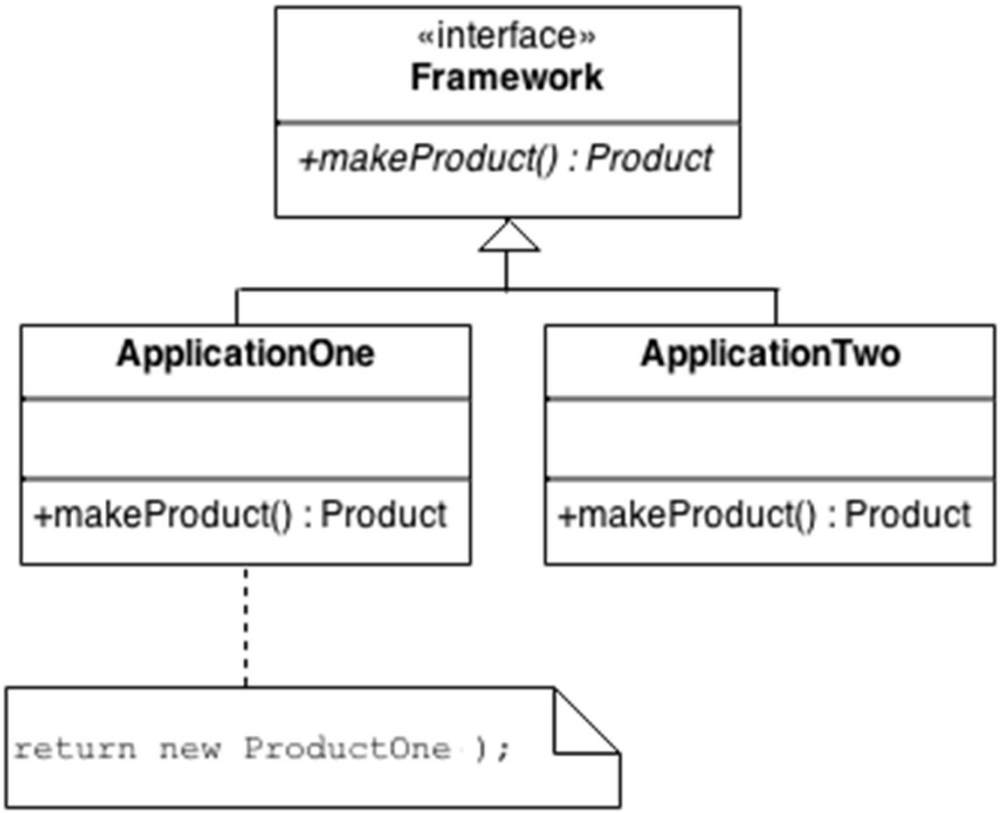
- The factory pattern (factory method pattern) is a creational design pattern
- A factory is a class that is used to encapsulate object creation code
- A factory class instantiates and returns a particular type of object based on data passed to the factory
- The different types of objects that are returned from a factory typically are subclasses of a common parent class
- The creational method is often called something such as *getInstance*
- The factory pattern is used to replace class constructors, abstracting the process of object generation so that the type of the object instantiated can be determined at run-time

# Factory Pattern



# Factory Pattern



# Factory Pattern

- A good example is when creating a connection to a data source if the type of the data source will be selected by the end-user
- In this case, an abstract class named "DataSource" may be created that defines the base functionality of all data sources
- Many concrete subclasses may be created, perhaps "SqlDataSource", “XmlDataSource", "CsvDataSource", etc, each with specific functionality for a different type of data
- At run-time, a factory class, perhaps named "DataSourceFactory", generates objects of the correct concrete class based upon a parameter passed to the factory method

# Factory Pattern

- It is common to pass data that determines the type of object to be created to the factory when the factory is created (using constructor)
- However, if multiple objects are being created by the factory, it may make sense to pass this data to the factory's creational method, since it might not make sense to create a new factory object each time we wanted to have the factory instantiate a new object.
- A factory may also be used in conjunction with the singleton pattern. It is common to have a singleton return a factory instance
- Replace

```
AnimalFactory animalFactory = new AnimalFactory();
```

- with

```
AnimalFactory animalFactory = AnimalFactory.getAnimalFactoryInstance();
```

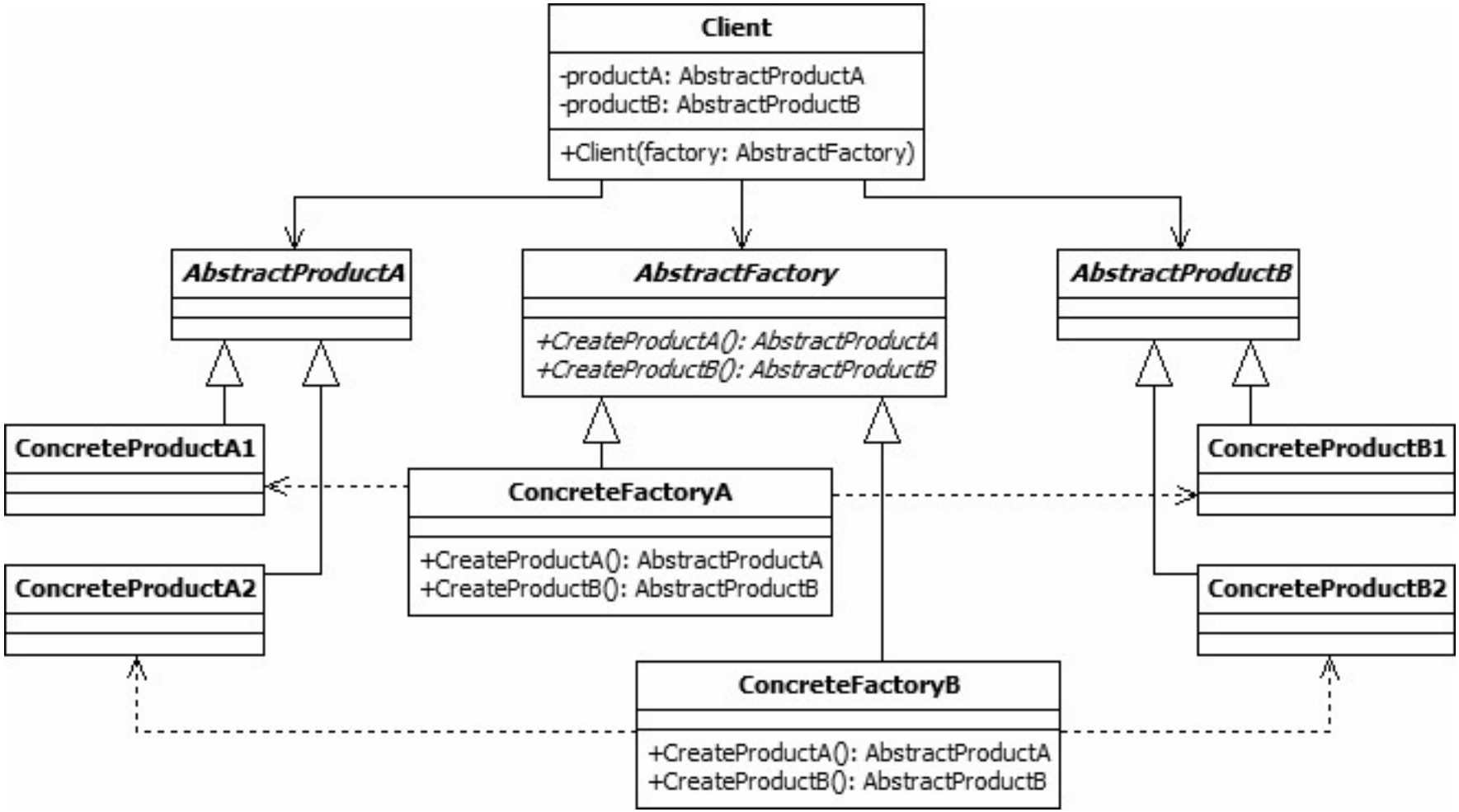
# Factory Pattern check list

- If you have an inheritance hierarchy that exercises polymorphism, consider adding a polymorphic creation capability by defining a static factory method in the base class
- Design the arguments to the factory method based on qualities or characteristics to identify the correct derived class to instantiate
- Consider designing an internal "object pool" that will allow objects to be reused instead of created from scratch
- Consider making all constructors **private** or **protected**

# Abstract Factory Pattern

- An abstract factory is a factory that returns factories
- A normal factory can be used to create sets of related objects. An abstract factory returns set of factories.
- Thus, an abstract factory is used to return factories that can be used to create sets of related objects
- Abstract Factory pattern is almost similar to **Factory_Pattern** and is considered as another layer of abstraction over factory pattern.
- Example:
  - you could have a Maruti factory that returns car part objects associated with a Maruti cars
  - You could also have a Skoda factory that returns car part objects associated with a Skoda cars

# Abstract Factory Pattern



# Abstract Factory Pattern

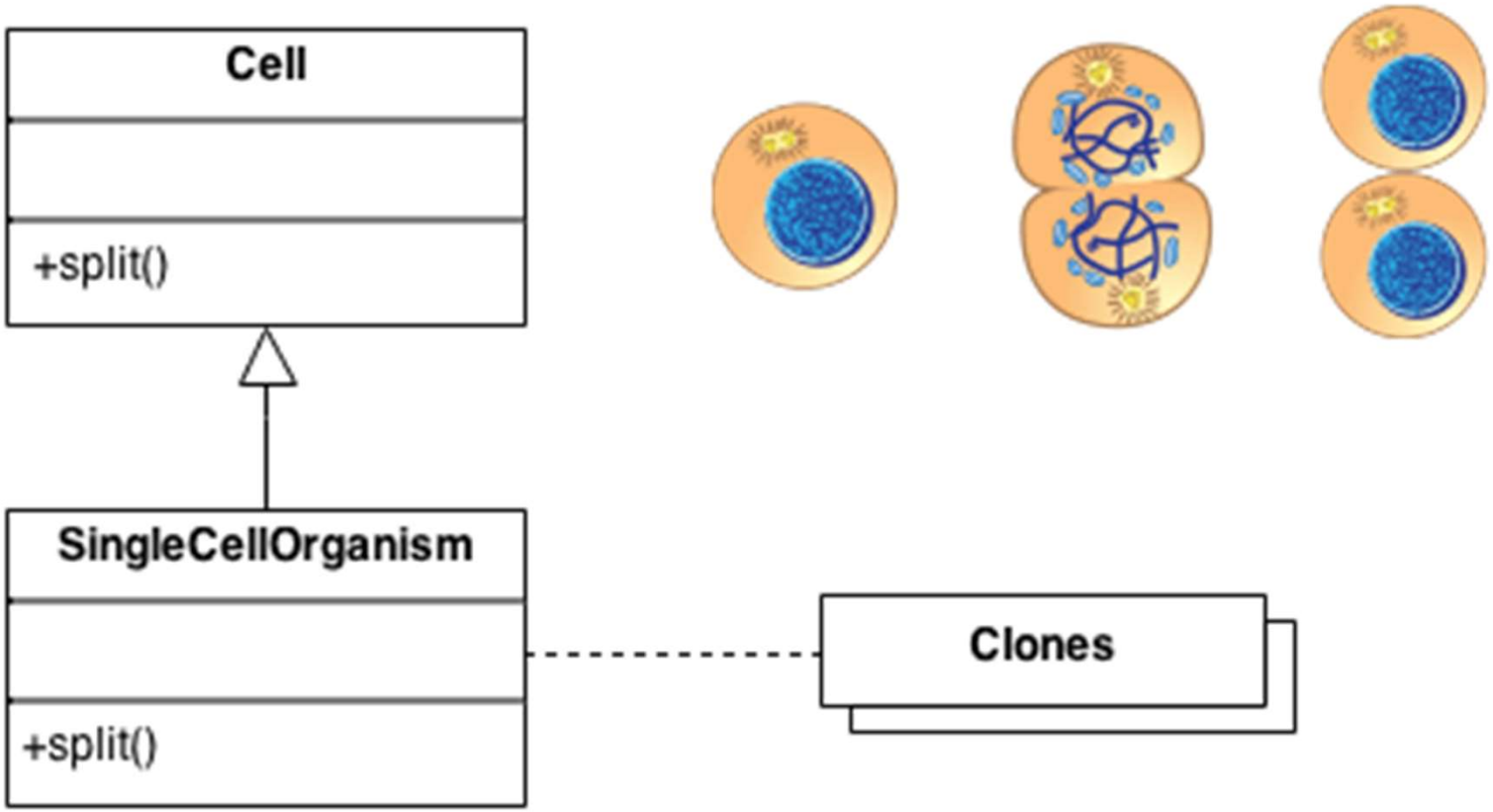
Components in AbstractFactory Pattern

1. Abstract Factory
2. Concrete Factories
3. Abstract Products
4. Concrete Products

# Prototype Pattern

- In the prototype pattern, a new object is created by cloning an existing object.
- In Java, the clone() method is an implementation of this design pattern.
- One example of how this can be useful is - if an original object is created with a resource such as a data stream that may not be available at the time that a clone of the object is needed
- Another example is - if the original object creation involves a significant time commitment, such as reading data from a database.
- Normally in Java, if you'd like to use cloning (ie, the prototype pattern), you can utilize the clone() method and the Cloneable interface.
- By default, clone() performs a shallow copy

# Prototype Pattern



# Structural Patterns

# Structural Patterns

- Structural patterns provide a manner to define relationships between classes or objects

- They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.

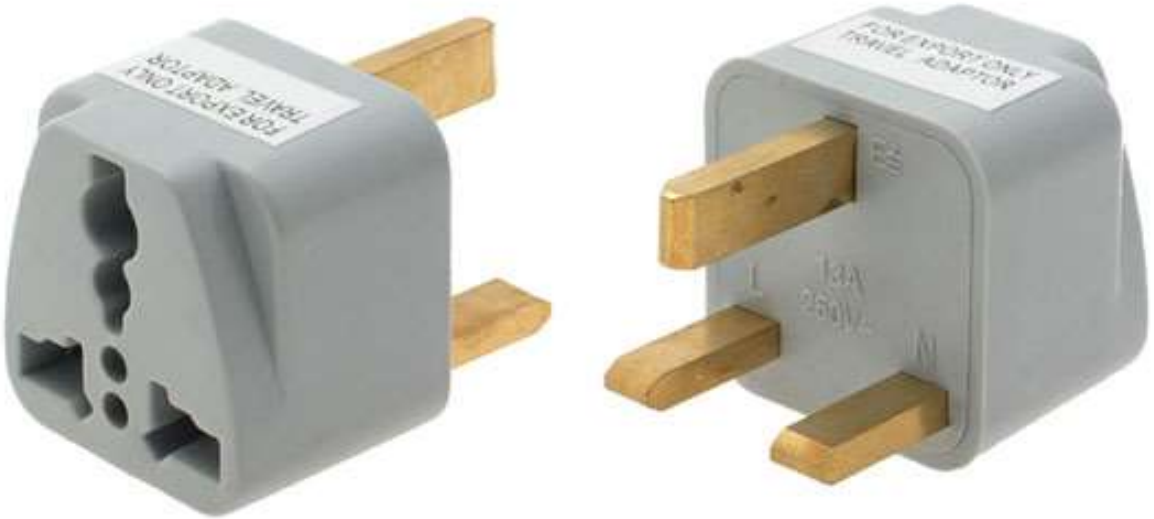
# Structural Patterns

1. Adapter Pattern
2. Composite Pattern
3. Proxy Pattern
4. Facade Pattern
5. Flyweight Pattern
6. Bridge Pattern
7. Decorator Pattern

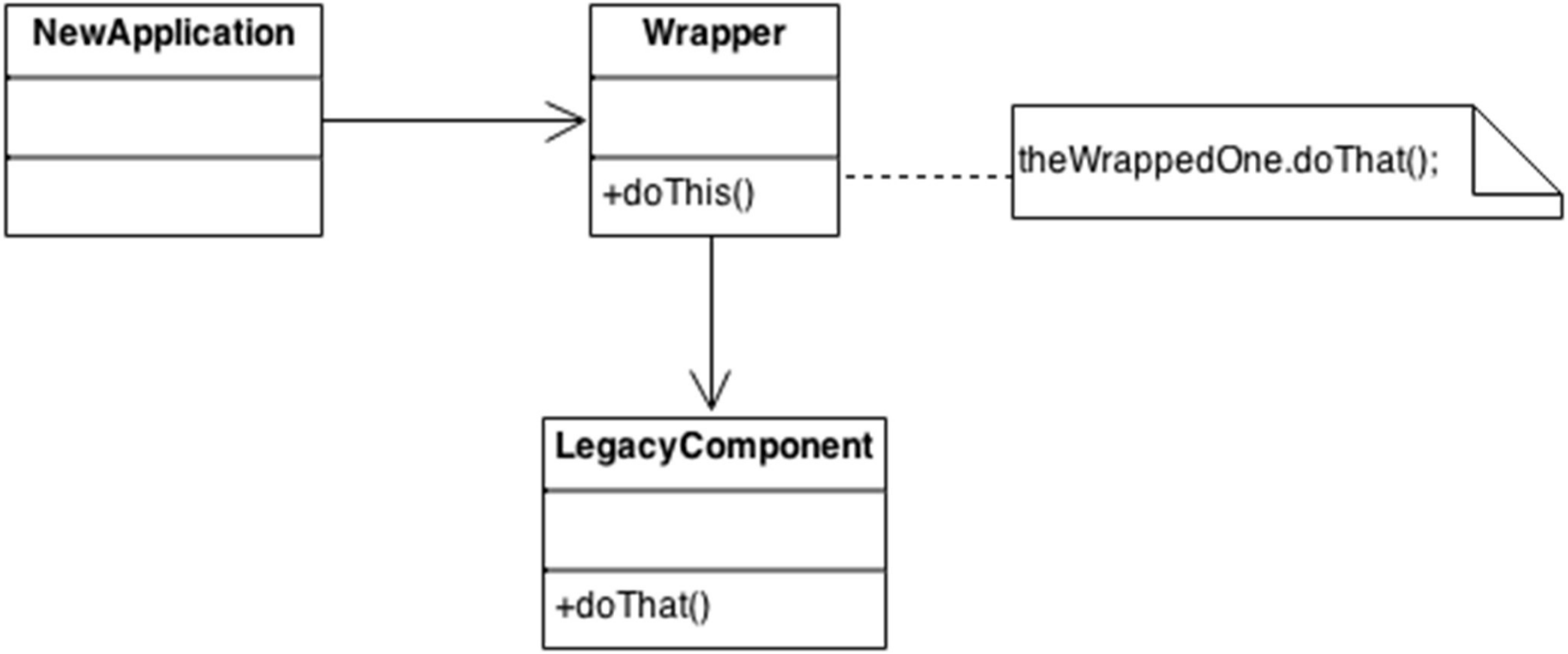
# Adapter Pattern

- In the adapter pattern, a wrapper class (ie, the adapter) is used translate requests from it to another class (ie, the adaptee).
- In effect, an adapter provides particular interactions with an adaptee that are not offered directly by the adaptee
- The adapter pattern takes two forms.
- In the first form, a "**class adapter**" utilizes inheritance.
- The class adapter extends the adaptee class and adds the desired methods to the adapter
- In the second form, an "**object adapter**" utilizes composition. The object adapter contains an adaptee and implements the target interface to interact with the adaptee.

# Adapter Pattern



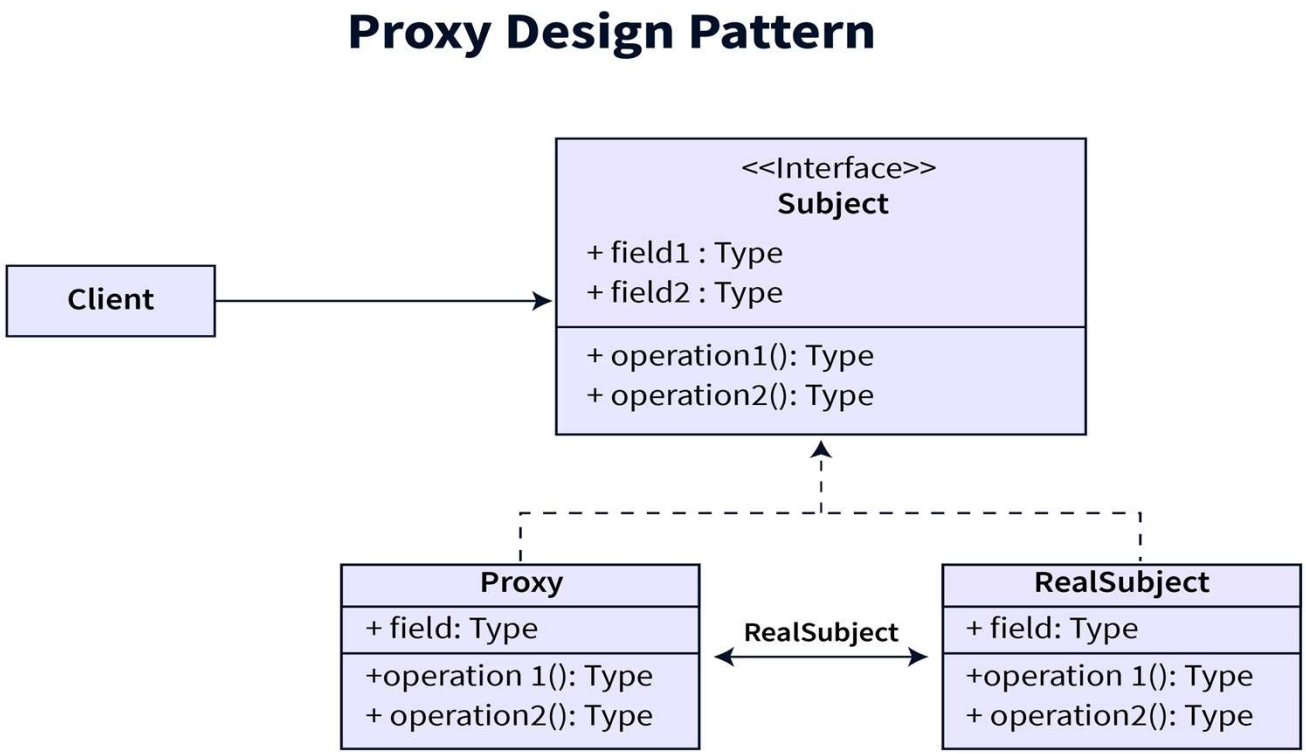
# Adapter Pattern



# Proxy Pattern

- In the proxy pattern, a proxy class object is used to control access to another class object.
- The reasons for this control can vary. As one example, a proxy may avoid instantiation of an object until the object is needed.
- This can be useful if the object requires a lot of time or resources to create.
- Another reason to use a proxy is to control access rights to an object.
- A client request may require certain credentials in order to access the object.

# Proxy Pattern



# Proxy Pattern

- Create Subject interface which is used by the client.
- It is the common **interface** for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- It defines the operations that are to be expected by the actual implementations.
- Real Subject provides the real implementation of the operations defined in the Subject Interface
- Proxy subject implements the Subject Interface to disguise itself as a Real Subject's object.
- It also maintains a reference to the RealSubject to provide actual functionality.
- It controls access to the RealSubject and may be responsible for its creation and deletion

# Proxy Pattern

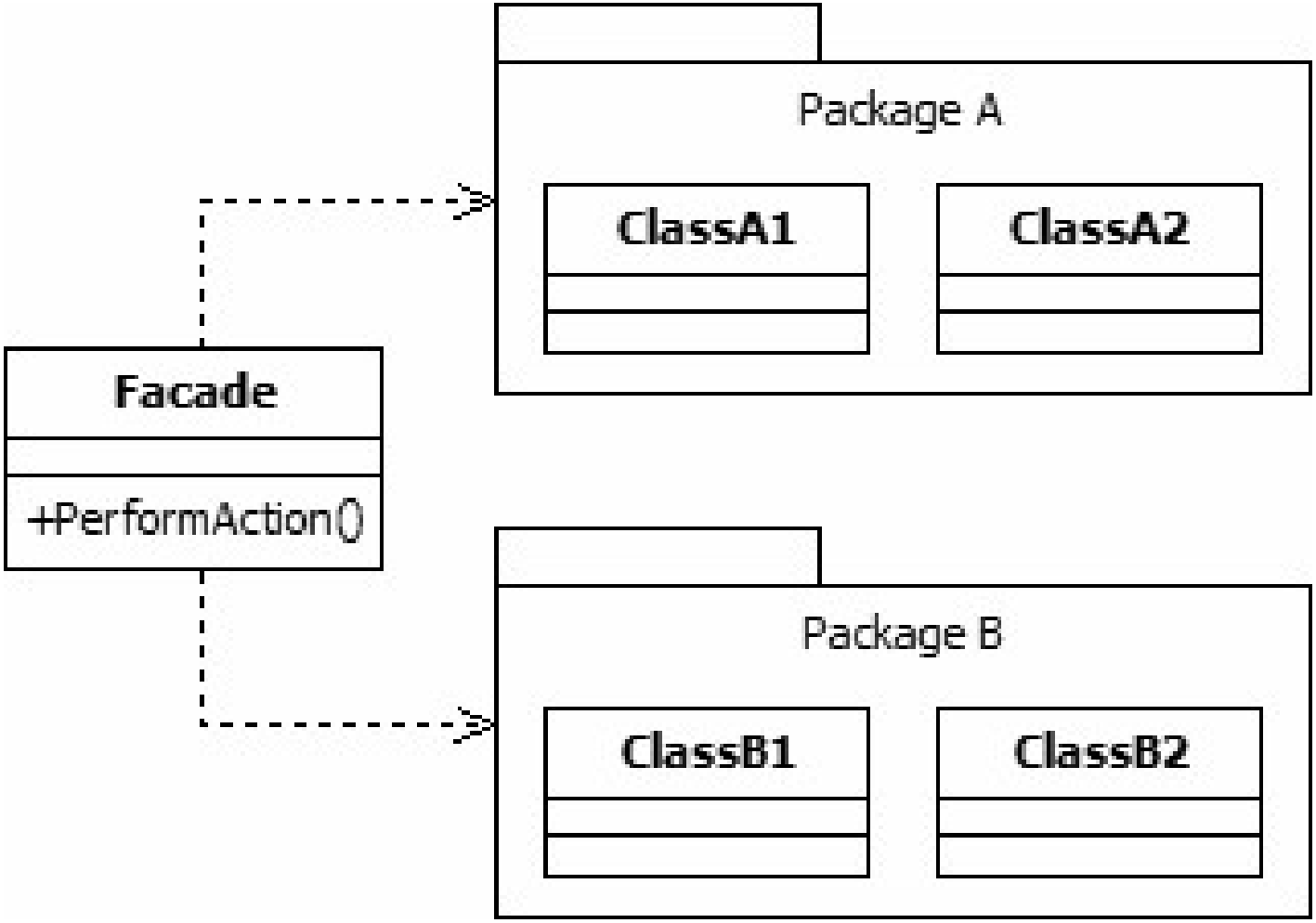
**common use cases in which the Proxy pattern**

- A virtual proxy is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object
- A remote proxy provides a local representative for an object that resides in a different address space (stub in RMI)
- A protective proxy controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request
- A cache proxy that can preserve some values returned by the object earlier

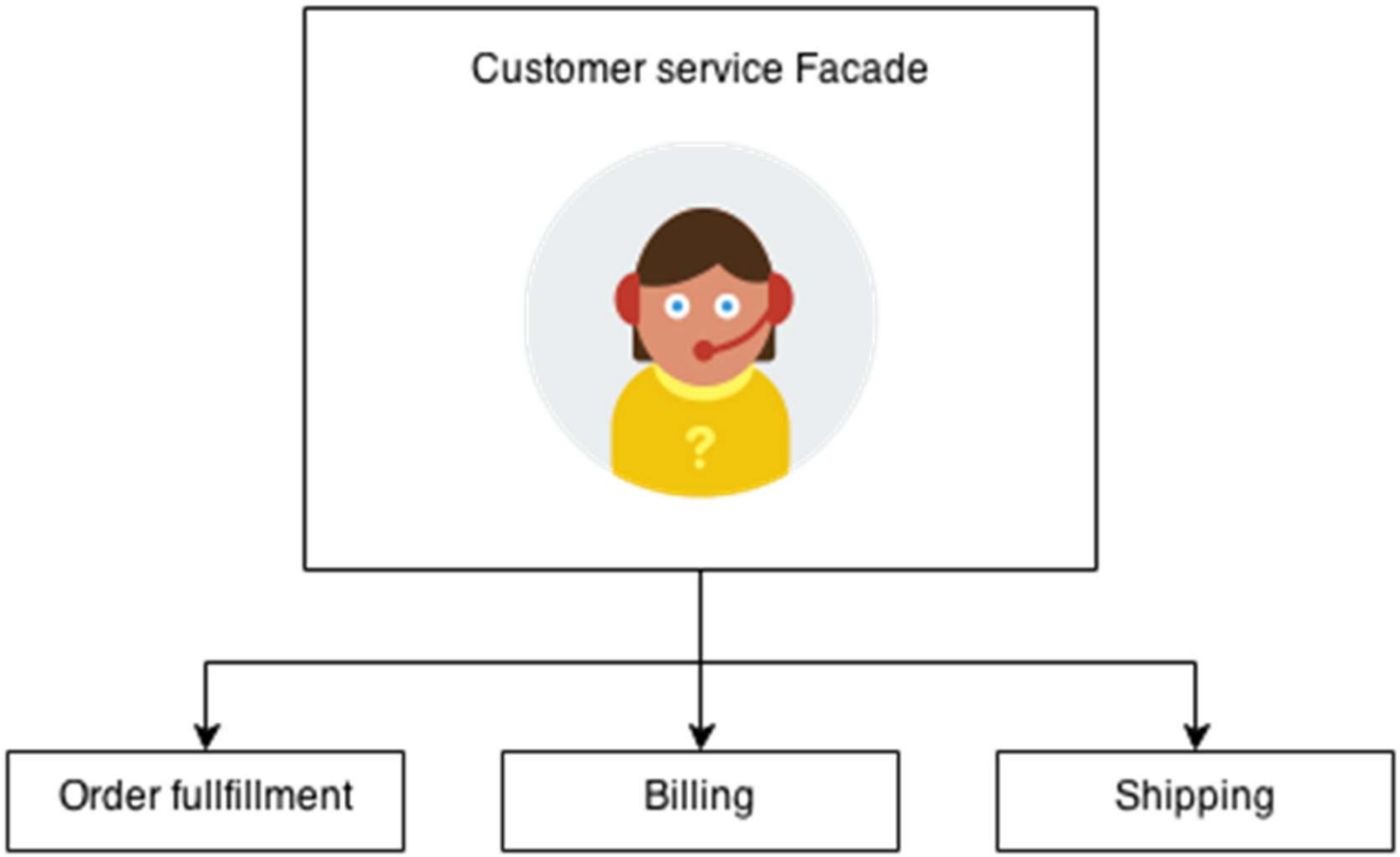
# Facade Pattern

- facade class is used to provide a single interface to set of classes.
- The facade simplifies a clients interaction with a complex system by localizing the interactions into a single interface.
- As a result, the client can interact with a single object rather than being required to interact directly in complicated ways with the objects that make up the subsystem

# Facade Pattern



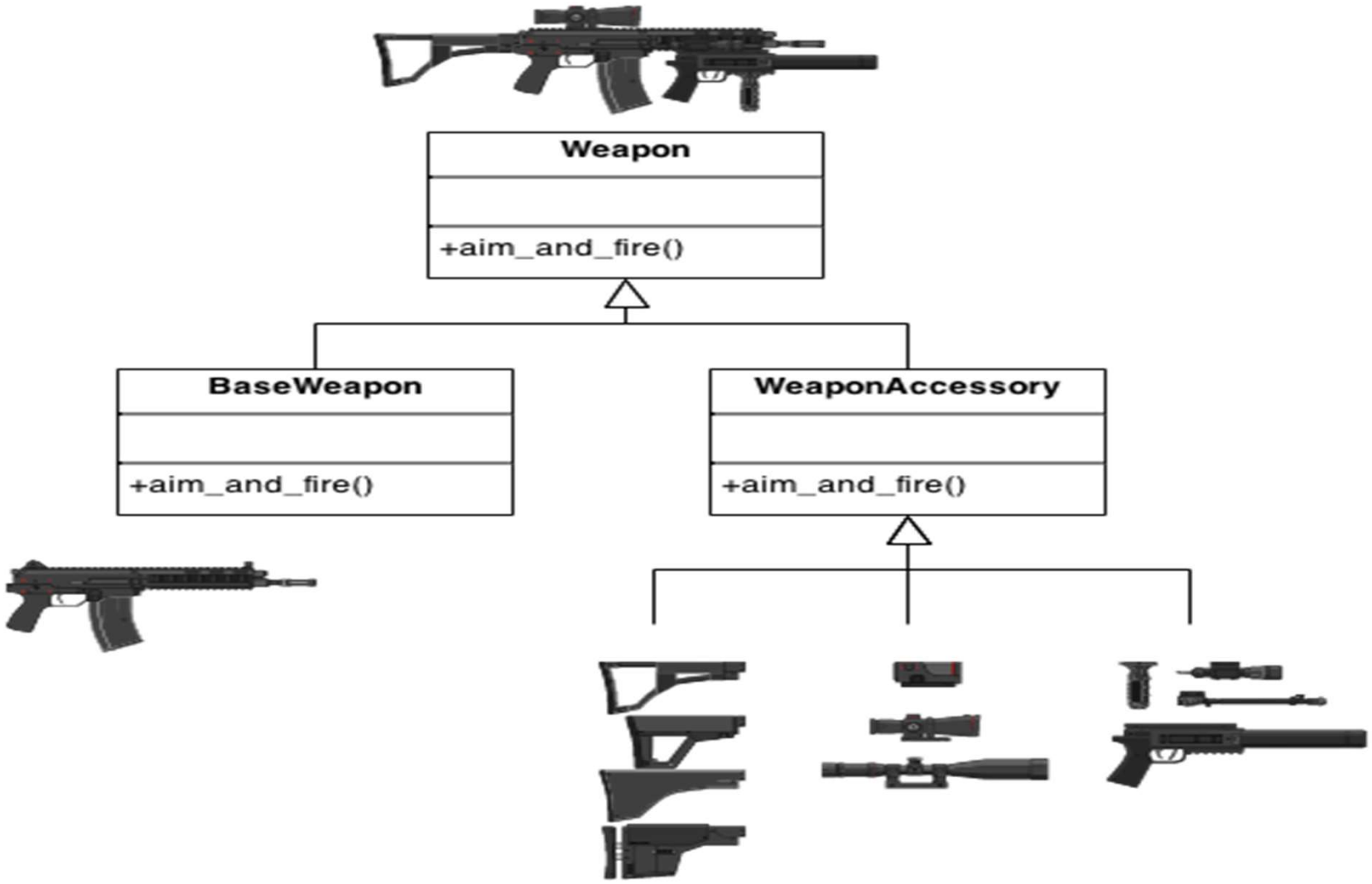
# Facade Pattern



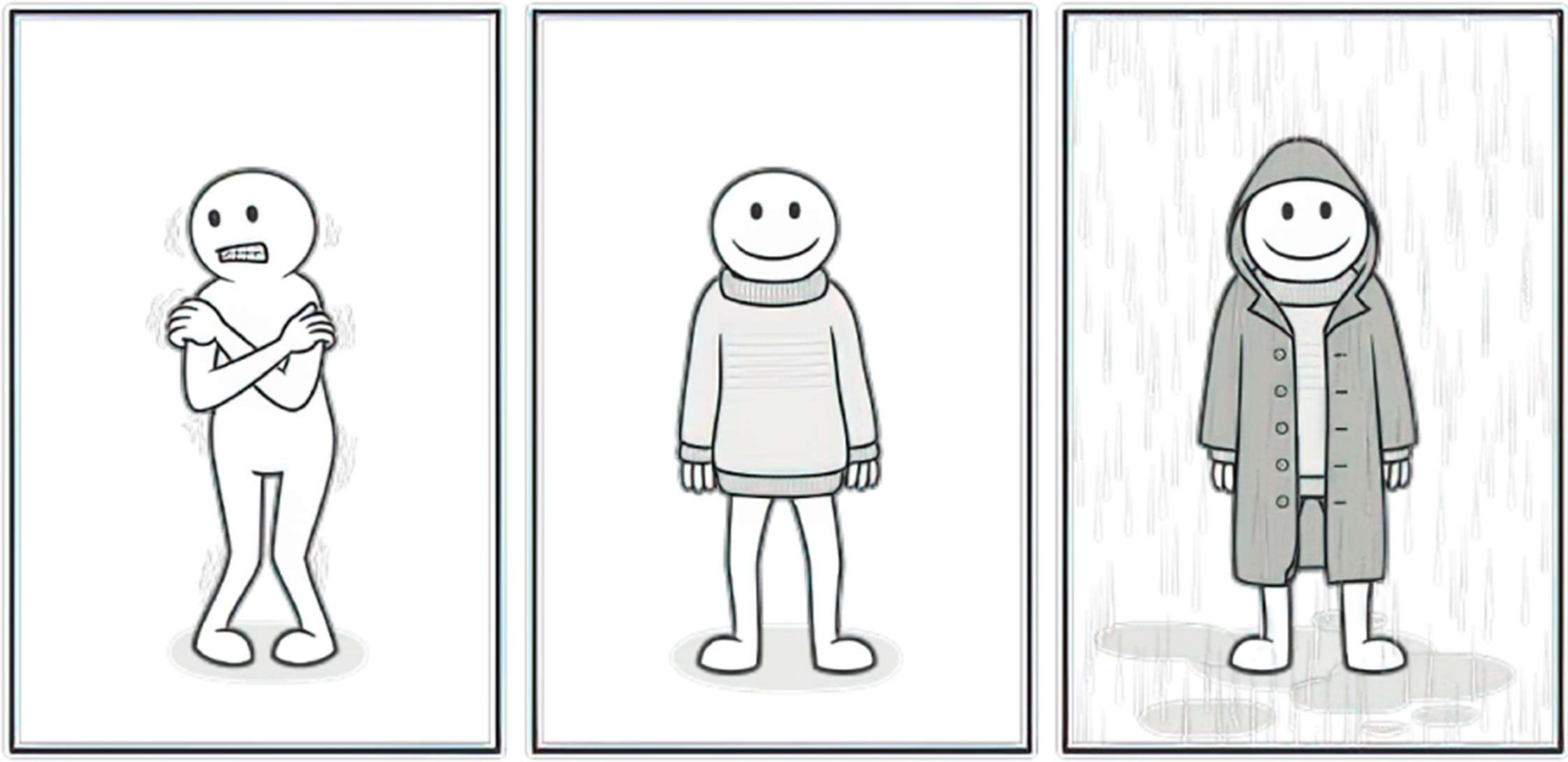
# Decorator Pattern

- Whereas inheritance adds functionality to classes, the decorator pattern adds functionality to objects by wrapping objects in other objects.
- Each time additional functionality is required, the object is wrapped in another object
- For the decorator pattern, we require a class that serves as the base object that we add functionality to.
- This is a Concrete Component, and it implements a Component interface.
- The Component interface declares the common operations that are to be performed by the concrete component and all the decorators that wrap the concrete component object.
- A Decorator is an abstract class that implements the Component interface and contains a reference to a Component. Concrete Decorators are classes that extend Decorator.

# Decorator Pattern



# Decorator Pattern



# Decorator Pattern VS Inheritance

- It appears decorator class simply adds new functionality to an existing class (similar to inheritance)
- Then why decorator if inheritance can be used?

- **There are two reasons for this**

- The decorator pattern allows you to add behavior **dynamically** at runtime to an instance of a particular class, whereas inheritance is compile time definition

- Adding new behavior to the base class by creating separate subclass for every possible combination could lead to a maintenance problem

# Behavioral Patterns

# Behavioral Patterns

- **Behavioral** patterns are used in communications between entities
- They make it easier and more flexible for these entities to communicate

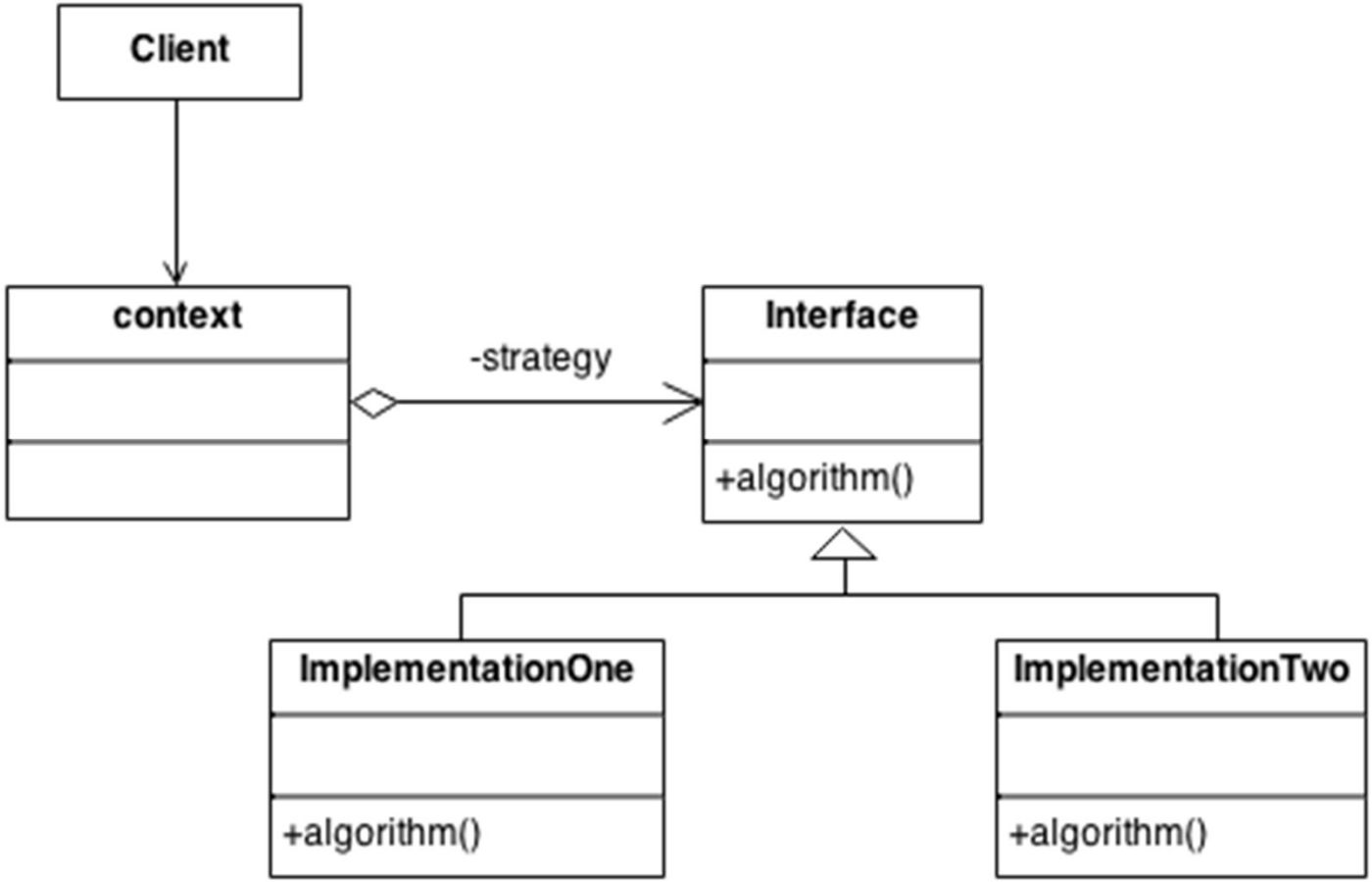
# Behavioral Patterns

1. Strategy Pattern
2. Chain of Responsibility Pattern
3. Observer Pattern
4. State Pattern
5. Command Pattern
6. Iterator Pattern
7. Interpreter Pattern
8. Template Method Pattern
9. Mediator Pattern
10. Visitor Pattern
11. Memento Pattern

# Strategy Pattern

- This pattern enables an object to choose from multiple algorithms and behaviors at runtime, rather than statically choosing a single one.
- It is based on the principle of composition over inheritance.
- It defines a family of algorithms, encapsulates each one, and makes them interchangeable at runtime.
- The core idea behind this pattern is to separate the algorithms from the main object
- This allows the object to delegate the algorithm's behavior to one of its contained strategies

# Strategy Pattern



# Strategy VS Inheritance

- The strategy pattern is one way that composition can be used as an alternative to subclassing.
- Rather than providing different behaviors via subclasses overriding methods in super classes, the strategy pattern allows different behaviors to be placed in Concrete Strategy classes which share the common Strategy interface
- In simpler terms, the Strategy Design Pattern provides a way to extract the behavior of an object into separate classes that can be swapped in and out at runtime

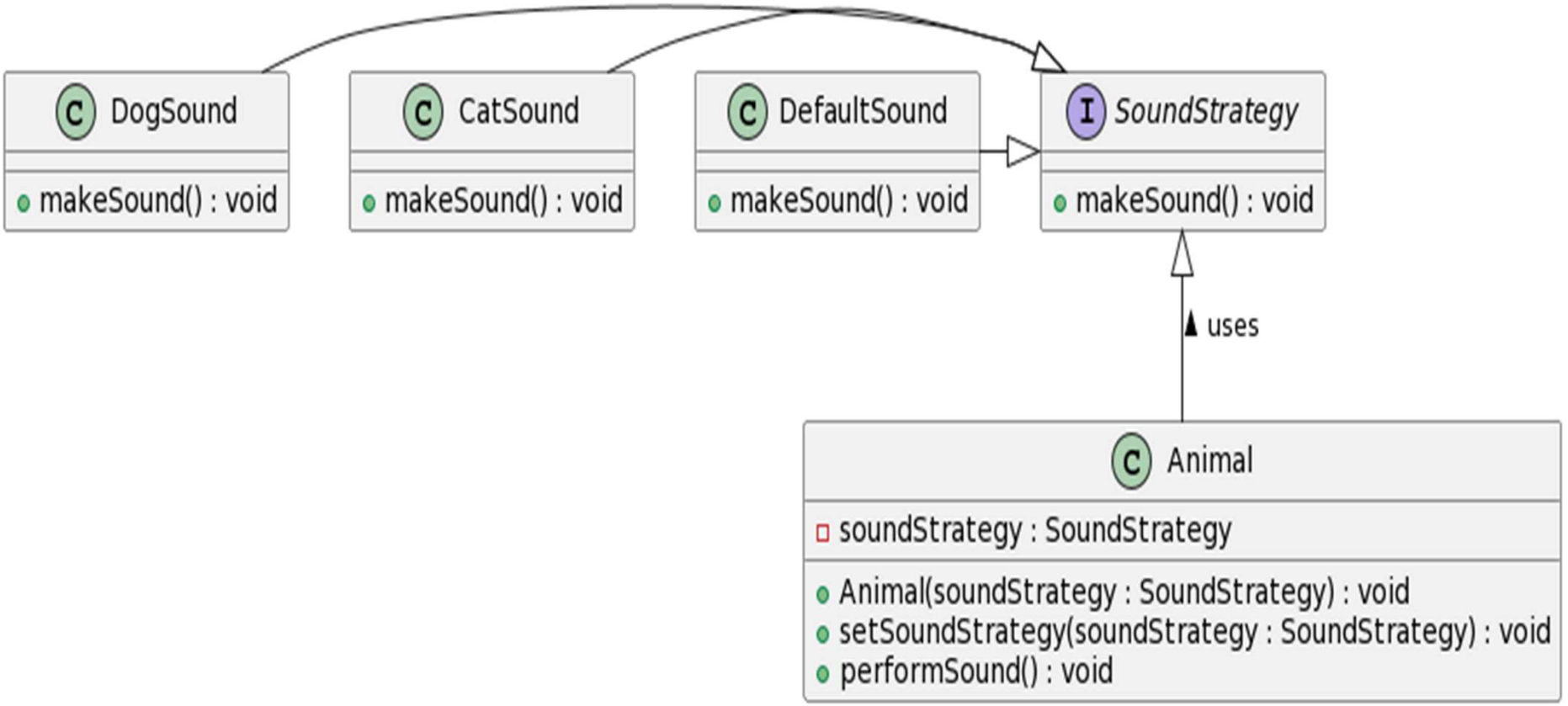
# Inheritance Solution

```
public class Animal {
    public void sound() {
        System.out.println("Animal makes a sound");
    }
}

public class Dog extends Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}

public class Cat extends Animal {
    public void sound() {
        System.out.println("Cat meows");
    }
}
```

3 fundamental questions

- What if Animal has additional methods
- What happens when new methods added to Animal
- What if you want multiple inheritance

# Strategy Solution



# Strategy Solution

```
public interface SoundStrategy {
    void makeSound();
}

public class DogSound implements SoundStrategy {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

public class CatSound implements SoundStrategy {
    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }
}
```

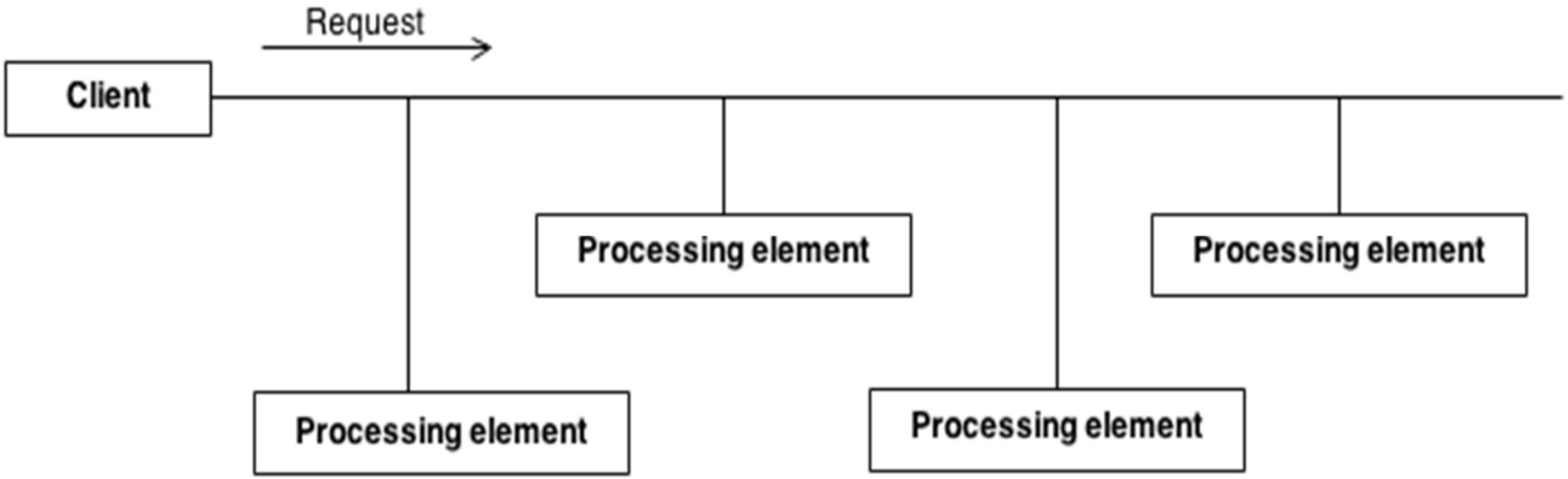
```
public class Animal {
    private SoundStrategy soundStrategy;

    public Animal(SoundStrategy soundStrategy) {
        this.soundStrategy = soundStrategy;
    }

    public void setSoundStrategy(SoundStrategy
soundStrategy) {
        this.soundStrategy = soundStrategy;
    }

    public void performSound() {
        soundStrategy.makeSound();
    }
}
```

separate *Dog* and *Cat* classes are no longer necessary

# Chain of Responsibility Pattern

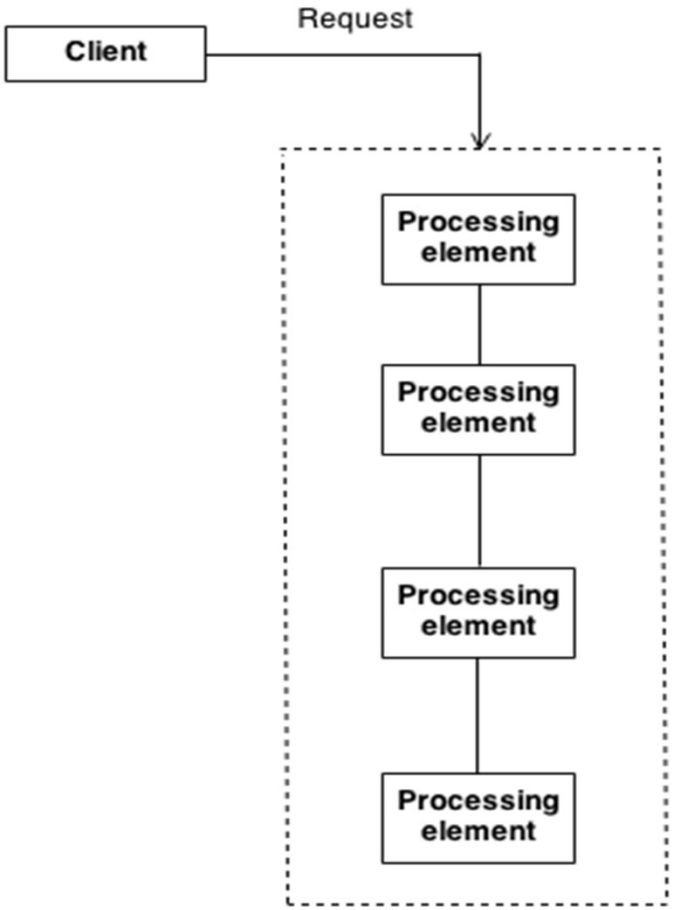
- a series of handler objects are chained together to handle a request made by a client object.
- If the first handler can't handle the request, the request is forwarded to the next handler, and it is passed down the chain until the request reaches a handler that can handle the request or the chain ends.
- In this pattern, the client is decoupled from the actual handling of the request, since it does not know what class will actually handle the request
- Handler is an interface for handling a request and accessing a handler's successor.
- A Handler is implemented by a Concrete Handler. The Concrete Handler will handle the request or pass it on to the next Concrete Handler.
- A Client makes the request to the start of the handler chain

# Chain of Responsibility Pattern



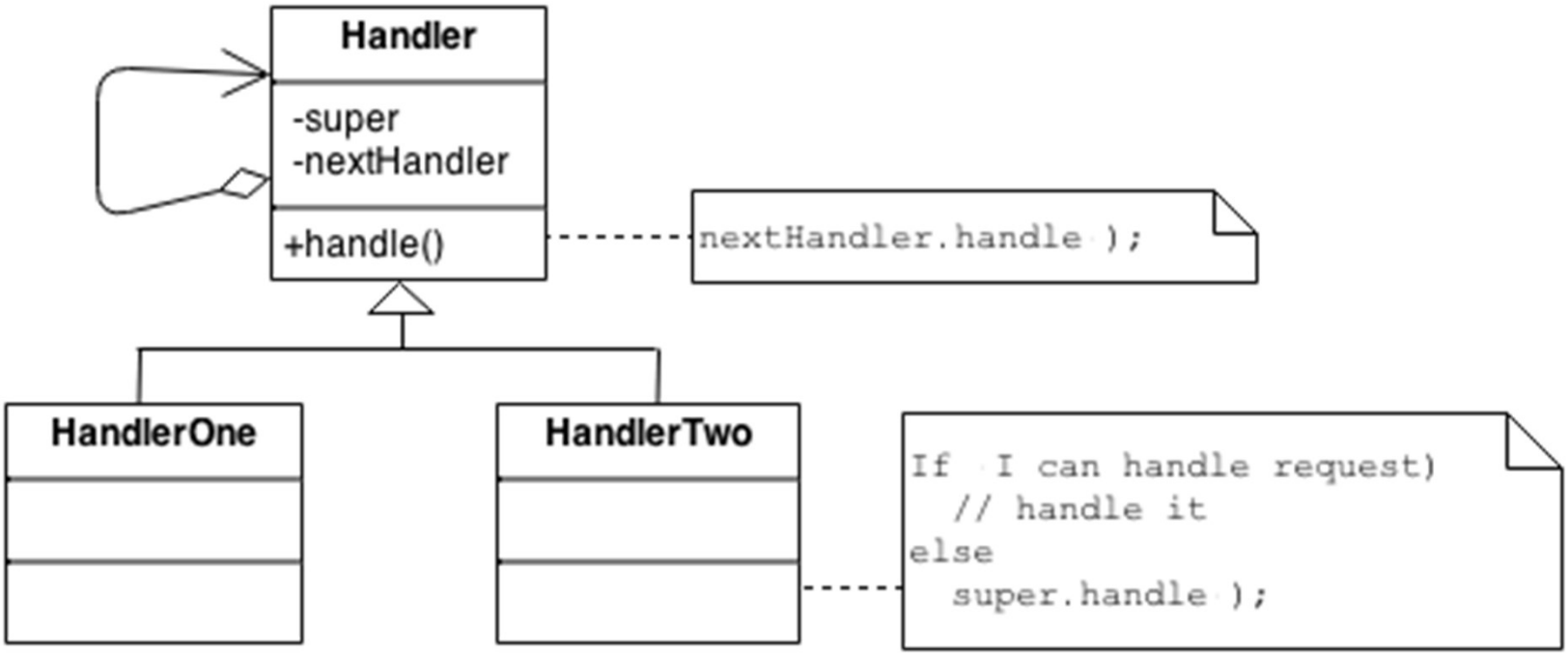
# Chain of Responsibility Pattern

- Encapsulate the processing elements inside a "pipeline" abstraction; and have clients "launch and leave" their requests at the entrance to the pipeline
- 2140




# Chain of Responsibility Pattern

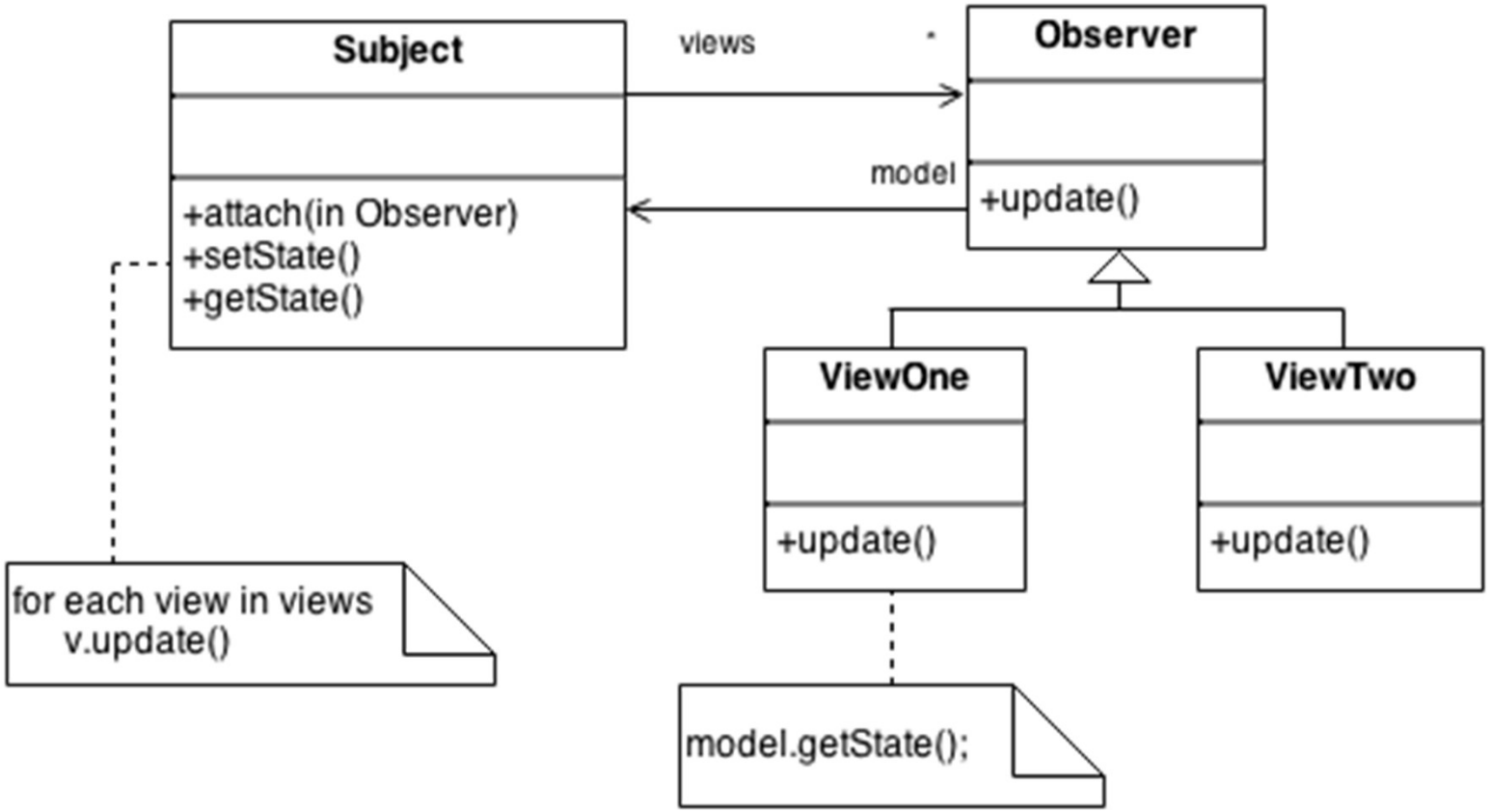
- The derived classes know how to satisfy Client requests. If the "current" object is not available or sufficient, then it delegates to the base class, which delegates to the "next" object, and the circle of life continues



# Observer Pattern

- Define an object that is the "keeper" of the data model and/or business logic (the Subject).
- Delegate all "view" functionality to decoupled and distinct Observer objects.
- Observers register themselves with the Subject as they are created.
- Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

# Observer Pattern



# State Pattern

- The state pattern allows an object to completely change its behavior depending upon its current internal state

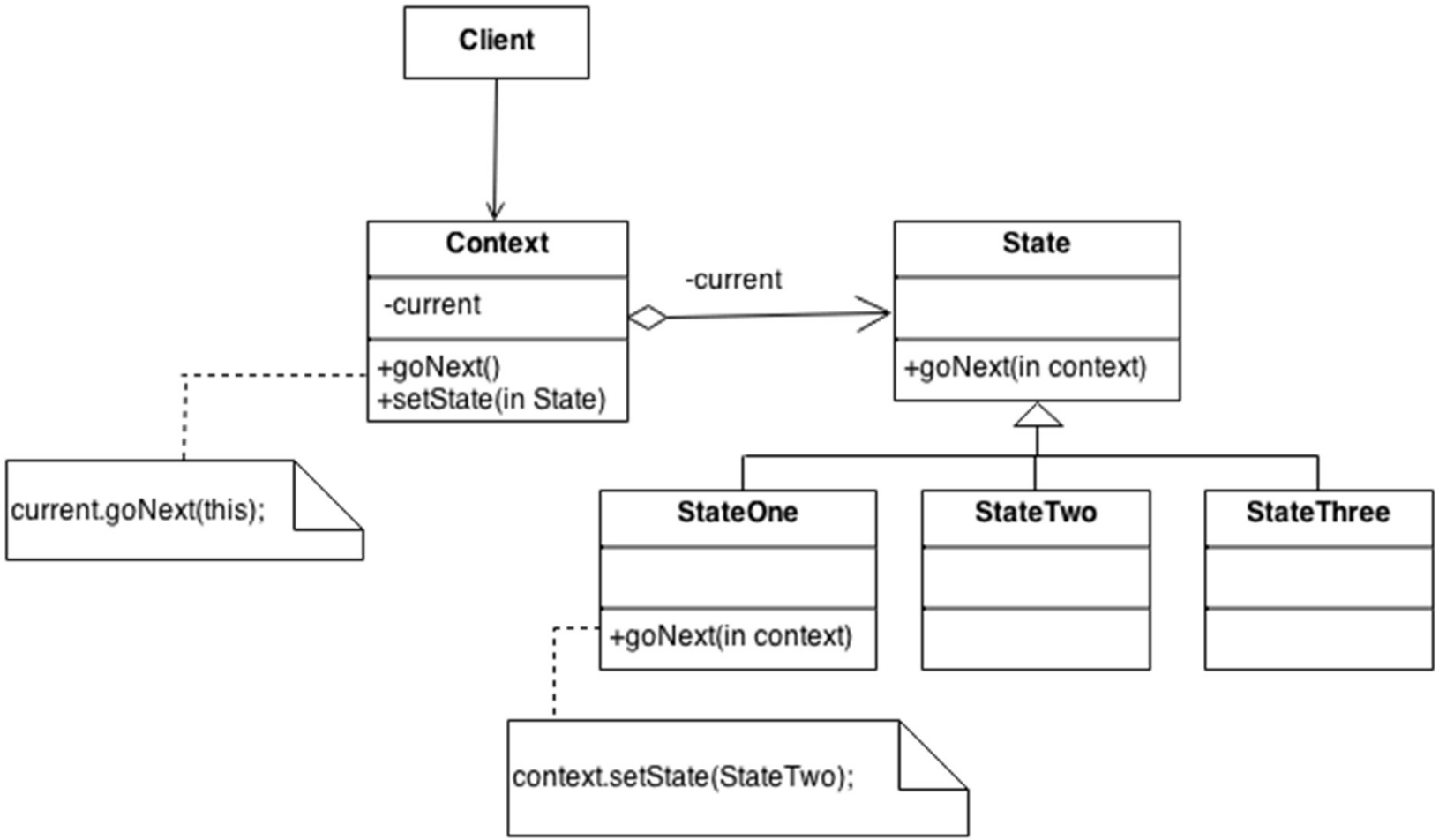
- The simplest approach would be to add some boolean flags and apply simple if/else statements within each of our methods
- This works in a simple scenario.
- However, it might complicate and pollute the code with too many states

- The State interface declares particular methods that represent the behaviors of a particular state.
- Concrete States implement these behaviors.
- By changing a Context's Concrete State, we change its behavior.

# State Pattern

- Define a "context" class to present a single interface to the outside world.
- Define a State abstract base class.
- Represent the different "states" of the state machine as derived classes of the State base class.
- Define state-specific behavior in the appropriate State derived classes.
- Maintain a pointer to the current "state" in the "context" class.
- To change the state of the state machine, change the current "state" pointer.

# State Pattern



# State vs Strategy

- Both design patterns are very similar, but with the idea behind them slightly different.
- First, the **strategy pattern** defines a family of interchangeable algorithms. Generally, they achieve the same goal, but with a different implementation, for example, sorting or rendering algorithms.
- In state pattern, the behavior might change completely, based on actual state

- Next, in strategy, the client has to be aware of the possible strategies to use and change them explicitly
- Whereas in state pattern, each state may be linked to another and create the flow as in Finite State Machine

# Command Pattern

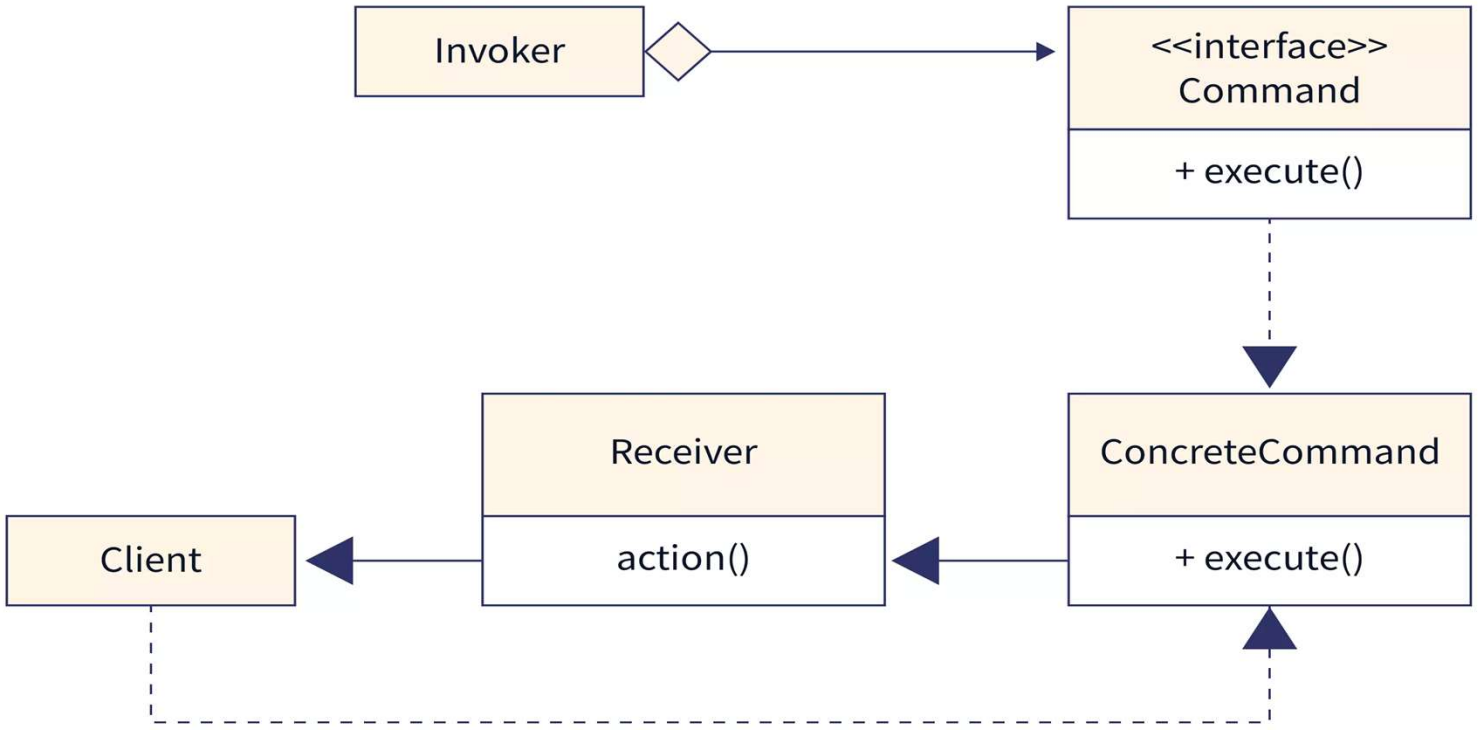
- In command pattern an object is used to encapsulate all information needed to perform an action
- This information includes the method name, the object that owns the method and values for the method parameters.

- Four terms always associated with the command pattern are command, receiver, invoker and client.
- A command object knows about receiver and invokes a method of the receiver.
- Values for parameters of the receiver method are stored in the command
- The receiver object to execute these methods is also stored in the command object

# Command Pattern (contd)

- The invoker does not know anything about a concrete command, it knows only about the command interface
- Invoker object(s), command objects and receiver objects are held by the client
- The client decides which receiver objects it assigns to the command objects, and which commands it assigns to the invoker
- The client decides which commands to execute at which points
- To execute a command, it passes the command object to the invoker object.

# Command Pattern



# Command Pattern - process

1. Define a Command interface with a method signature like execute().

2. Create one or more derived classes that encapsulate a "receiver" object, the method to invoke, the arguments to pass

3. Instantiate a Command object for each execution request

4. Pass the Command object to the invoker (receiver)

5. The invoker decides when to execute the command by calling execute() on the command object

# Learn More

Explore links below for other patterns and good examples:

1. https://www.blackwasp.co.uk/GofPatterns.aspx
2. https://sourcemaking.com/design_patterns
3. https://www.avajava.com/tutorials/categories/design-patterns