Design Pattern

Deepikkaa.S

Agenda

What is a Design Pattern?

GOF?

Types of Design Patterns?

Use cases.

What is a Design Pattern?

- ✓ The best practices followed by experienced software developers.
- ✓ GENERALIZED SOLUTIONS gave during the software development.
- ✓ Targeted to solve the problems of object creation and integration

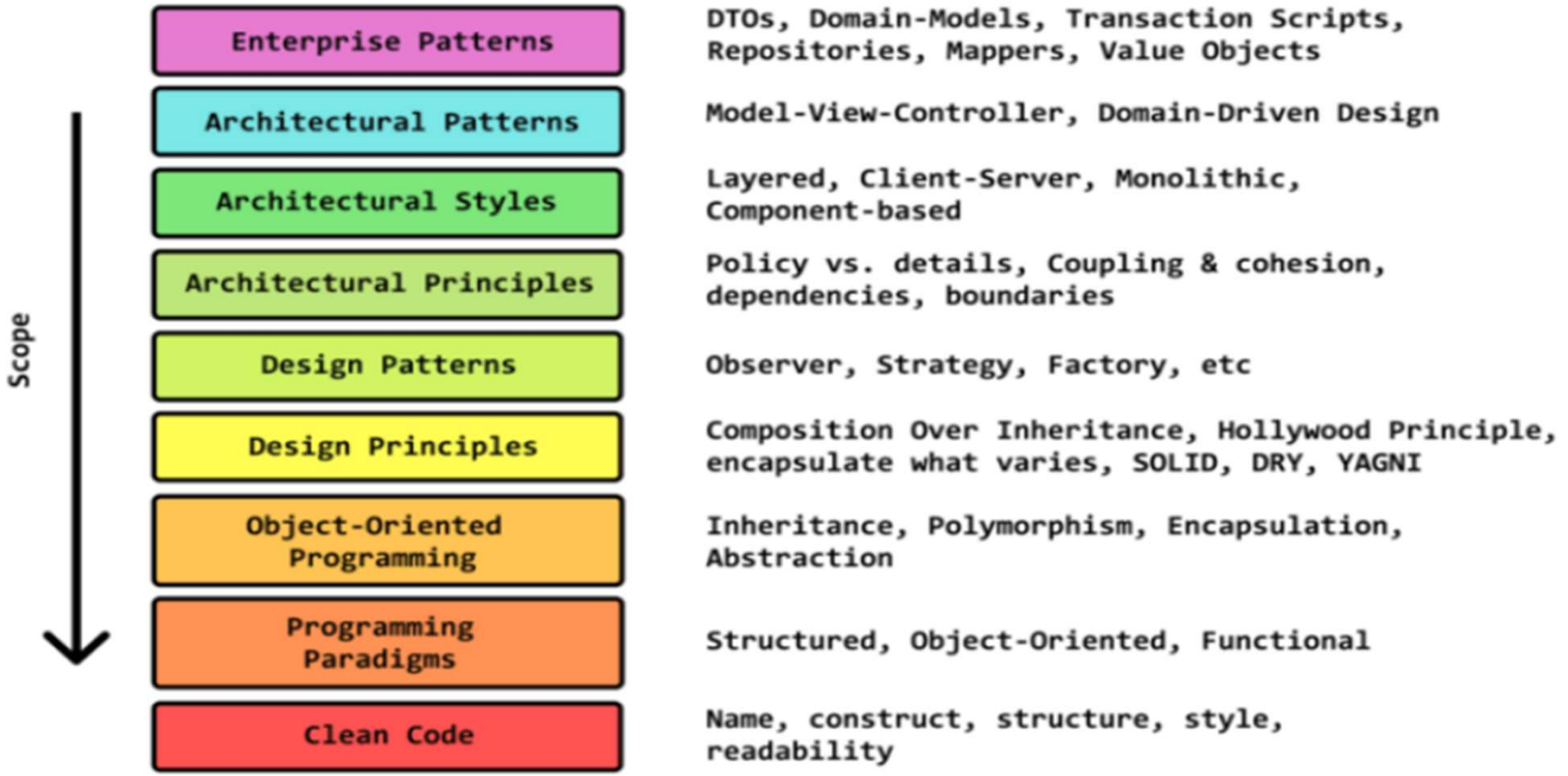
Design patterns are repeatable/reusable solutions to commonly occurring problems in a certain context in software design.

GOF – Gang Of Four

- ✓ Design Patterns are developed by
  - ✓ Erich Gamma
  - ✓ Richard Helm
  - ✓ Ralph Johnson
  - ✓ John Vlissides

- ✓ In the year – 1994, GOF published a book titled " **design patterns - elements of Reusable object-oriented software** " which encapsulated all the popular software design patterns.

The Software Design & Architecture Stack



Patterns and Frameworks

- Design patterns
  - design level constructs
  - small and easily implemented
  - the user must understand the details of the pattern to use it
  - used to help design frameworks
- Frameworks
  - includes both design and implementation
  - larger and more complex than design patterns
  - the user does not need to understand the framework completely

Two criteria of Design Pattern

- ✓Purpose: what a pattern does Creational: The process of object creation Structural: The composition of classes or objects Behavioral: Characterize the ways in which classes or objects interact and distribute responsibility
- ✓Scope: whether the pattern applies primarily to classes or to objects

Categories of Design Pattern

Creational

- Singleton
- Prototype
- Builder
- Factory
- Abstract Factory
- Object Pool
- Dependency injection
- Multiton

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral

- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- Chain of Responsibility
- State
- Strategy
- Visitor
- Null Object
- Template

Design Pattern Classification

- Along the scope dimension, patterns are classified into two types:
  1) Class patterns and
  2) Object patterns.

➢ Class patterns deal with the relationships between classes which is through inheritance. So, these class relationships are **static at compile time.**

➢ Object patterns deal with object relationships that **can be changed at run-time** and these are dynamic. Almost all patterns use inheritance to some degree.

Class Pattern Vs Object Patterns

	Class patterns	Object patterns
Creational	Can defer object creation to subclasses	Can defer object creation to another object
Structural	They focus on the composition of classes (primarily using the concept of inheritance).	They focus on the different ways to assemble objects.
Behavioral	Describes the algorithms and execution flows. They also use the inheritance mechanism.	Describes how different objects can work together and complete a task.

Organizing the Catalog

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Common Causes of Redesign

- Creating an object by specifying a class explicitly
- Dependence on specific operations
- Dependence on hardware and software platform
- Dependence on object representations or implementations
- Algorithmic dependencies
- Tight coupling
- Extending functionality by subclassing
- Inability to alter classes conveniently

Creational Design Pattern

- ✓ **Deal with Object creation mechanism**
- ✓ **Create objects based on a use case**
- ✓ **Reduce the complexity of object creation**
- ✓ **Define the best possible way to reuse it**
- ✓ Creational design patterns follow one of the essential principles of OOPs which is **abstraction**.
- ✓ They hide the internal logic of object creation.
- ✓ The primary principle of creational design patterns is program shouldn't depend on how objects are created.
- ✓ The clients can directly call some methods instead of using the "new" keyword. Example: getInstance().

Advantage of Creational Design Pattern

- ✓ It reduces the complexity of object creation. Creating and managing the objects is a tedious process that will be governed by these Design Patterns.
- ✓ It defines the best possible way to reuse the objects. We create the objects by using the number of objects like time and memory. If we use these patterns, we can efficiently reuse the already-created objects for our code.
- ✓ It saves a number of resources. By saving the number of resources, we can serve particular clients in a particular amount of time.

Creational Design Patterns

- Abstract Factory. Allows the creation of objects without specifying their concrete type.
- Builder. Uses to create complex objects.
- Factory Method. Creates objects without specifying the exact class to create.
- Prototype. Creates a new object from an existing object.
- Singleton. Ensures only one instance of an object is created.

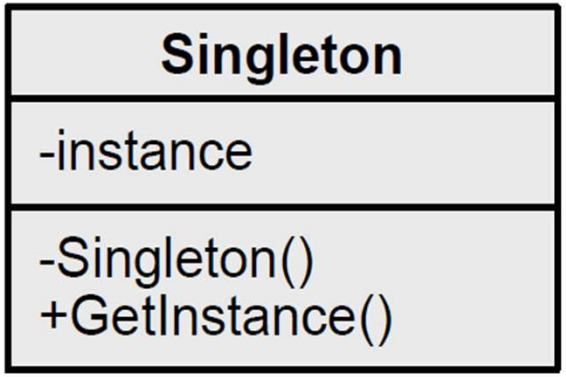
Singleton

- To create exactly one object of a class, and to provide global point of access to it.
- It does not allow the constructor to be invoked directly each time.

Class Diagram

Singleton

Ensure a class only has one instance and provide a global point of access to it



Real-time Example

- Instead of teaching every student individually, record videos only once, and it is repeatedly accessed by multiple users to learn.

Real-time Example in Application

- Singleton pattern can be used for developing in the following situations:
  - Logger classes
  - Database Connection
  - Configuration classes
  - Accessing resources in shared mode
  - Factories implemented as Singletons

- Example Singleton classes in java API:
  - java.lang.Runtime
  - java.awt.Desktop

  - Other patterns like Abstract Factory, Builder and Prototype can be implemented using the Singleton pattern

Lazy instantiation

```
public class Singleton

{
private static Singleton singleton;

/**
* Create private constructor
*/

private Singleton()
{
}

/**
* Create a static method to get instance.
*/

public static Singleton getInstance()
{
if (singleton == null)
{
singleton = new Singleton();
}
return singleton;
}

public void displayMessage()
{
System.out.println("I have called using singleton object");
}
}
```

Eager instantiation

```
public class Singleton
{
    private static Singleton singleton= new Singleton();
    /**
    * Create private constructor
    */
    private Singleton()
    {
    }

    /**
     * Create a static method to get instance.
     */

    public static Singleton getInstance()
    {
       return singleton;
    }

    public void displayMessage()
    {
        System.out.println("I have called using singleton object");
    }
}
```

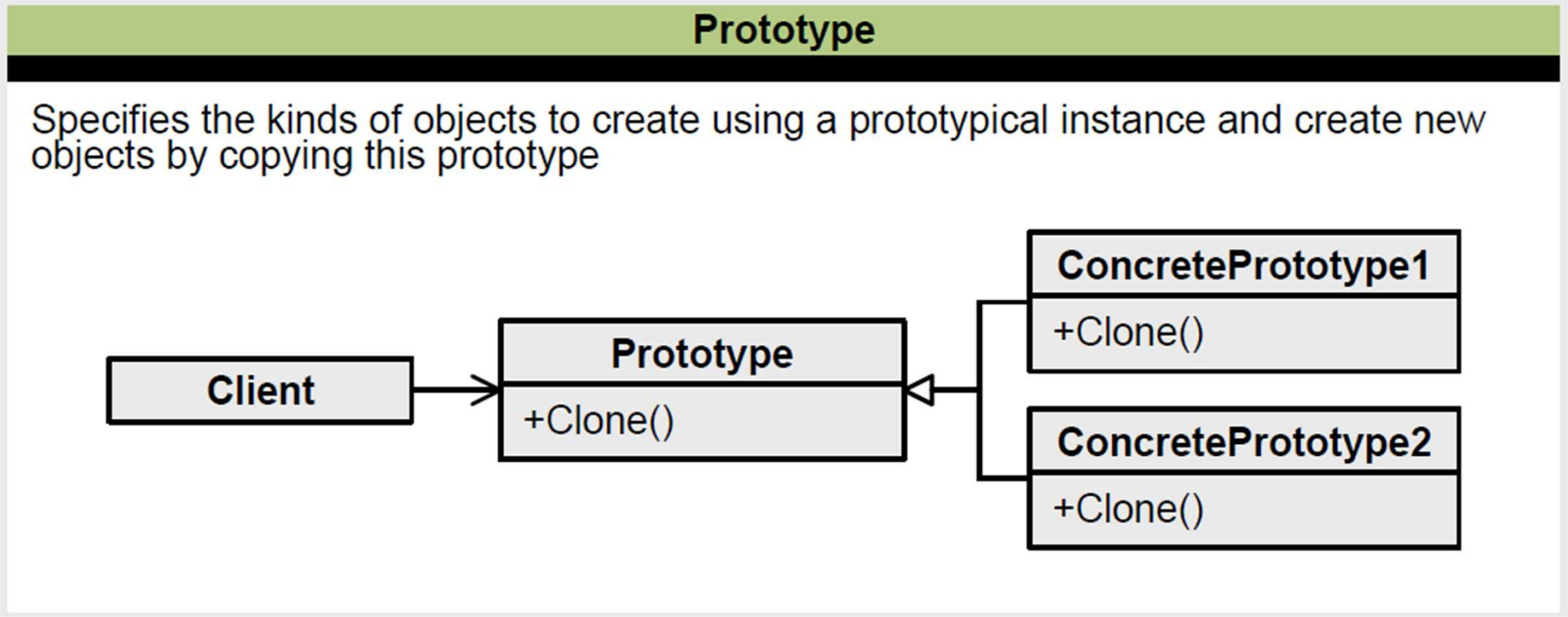
Synchronized Singleton Object – Thread Safe

```
public class Singleton

{
    private static Singleton singleton;

    /**
     * Create private constructor
     */

    private Singleton()
    {
    }

    /**
     * Create a static method to get instance.
     */

    public static synchronized Singleton getInstance()
    {
        if (singleton == null)
        {
            singleton = new Singleton();
        }
        return singleton;
    }

    public void displayMessage()
    {
        System.out.println("I have called using singleton object");
    }
}
```

Prototype

- The word “Prototype” itself conveys that a real object is accessed with the help of a virtual object.
- The main feature of the Prototype Design Pattern is that it creates objects by cloning or copying.
- If your application needs an N number of objects of the same type, then you’ll need to create the object for the first time, and you can copy or clone that object for the number of required times. So by using this pattern, you can save time, memory, and computational power.
- When to use?
  - When the object creation is a costly operation.
  - Example: An Object is to be created after reading data from a database. Reading data from the database is a costly operation.

Class Diagram



Real-time Example(Non-software Example)

- Let's suppose you are trying to download the movie "Mr. Bean". If the size of the film is 1 GB, it will take 30 minutes to download it. After watching that movie, you'll recommend it to your friends. If your friends ask for that movie file, you'll have to copy the file to a DVD or desktop rather than download it again.
- This is also called prototyping in which we are copying a source in many virtual things so that you can save memory, time, and data at the same time

Real-time Example in Application

- **Adding and deleting products in the middle of a game**–By registering a prototype instance with the client, you may easily integrate a new concrete product class into a system. Because a client can install and uninstall prototypes during runtime, this pattern is a little more flexible than other creational patterns.
- **Creating new objects by changing their values**–Object composition, rather than introducing new classes, allows you to design new behavior in highly dynamic systems by specifying values for an object's variables.
- **Creating new objects by changing their structure** – Many apps create things out of parts and subparts. Such apps frequently allow you to create sophisticated, user-defined structures to reuse a single sub-circuit.

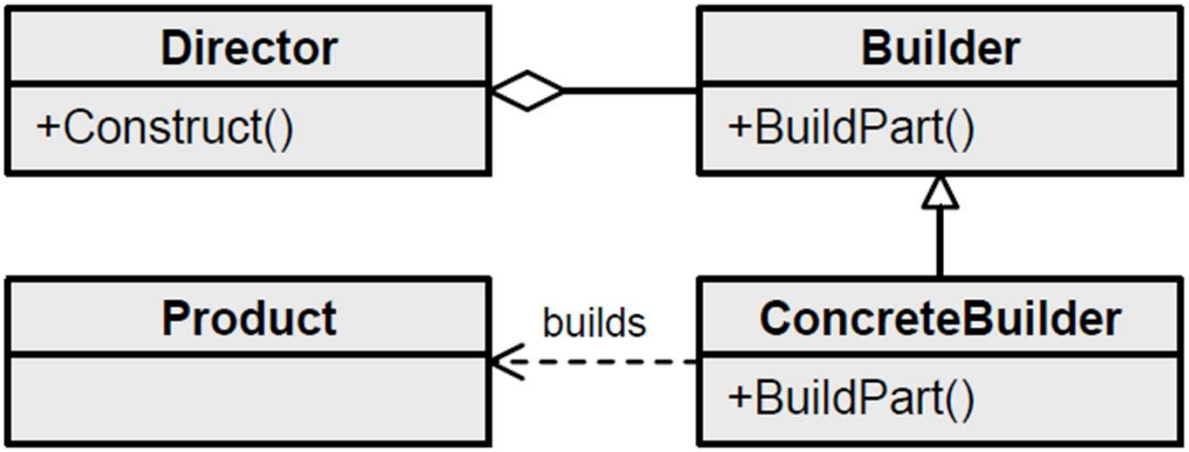
Builder

- To build complex objects using simpler ones.
- Use the Builder pattern when:
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
  - The construction process must allow different representations for the object that is constructed.
- The benefits and pitfalls of Builder pattern are:
  - It lets you vary the product's internal representation.
  - It isolates code for construction and representation.
  - It gives you finer control over the construction process.

Class Diagram

Builder

Separates the construction of a complex object from its representation so that the same construction process can create different representations.



Builder Pattern

Product

```
class Pizza
{
        private String dough = "";
        private String sauce = "";
        private String topping = "";
        public void setDough(String dough)     { this.dough = dough; }
        public void setSauce(String sauce)     { this.sauce = sauce; }
        public void setTopping(String topping) { this.topping = topping; }
}
```

Builder

```
abstract class PizzaBuilder
{
        protected Pizza pizza;
        public Pizza getPizza() { return pizza; }
        public void createNewPizzaProduct() { pizza = new Pizza(); }
        public abstract void buildDough();
        public abstract void buildSauce();
        public abstract void buildTopping();
}
```

Builder Pattern

Concrete Builders

```
class HawaiianPizzaBuilder extends PizzaBuilder
{
        public void buildDough()   { pizza.setDough("cross"); }
        public void buildSauce()   { pizza.setSauce("mild"); }
        public void buildTopping() { pizza.setTopping("ham+pineapple"); }
}
class SpicyPizzaBuilder extends PizzaBuilder
{
        public void buildDough()   { pizza.setDough("pan baked"); }
        public void buildSauce()   { pizza.setSauce("hot"); }
        public void buildTopping() { pizza.setTopping("pepperoni+salami"); }
}
```

Builder Pattern

Director

```
class Waiter
{
        private PizzaBuilder pizzaBuilder;
        public void setPizzaBuilder(PizzaBuilder pb) { pizzaBuilder = pb; }
        public Pizza getPizza() { return pizzaBuilder.getPizza(); }
        public void constructPizza()
        {
                pizzaBuilder.createNewPizzaProduct();
                pizzaBuilder.buildDough();
                pizzaBuilder.buildSauce();
                pizzaBuilder.buildTopping();
        }
}
```

A customer ordering a pizza

```
class Client
{
        public static void main(String[] args)
        {
                Waiter waiter = new Waiter();
                PizzaBuilder hawaiian_pizzabuilder = new HawaiianPizzaBuilder();
                PizzaBuilder spicy_pizzabuilder = new SpicyPizzaBuilder();
                waiter.setPizzaBuilder( hawaiian_pizzabuilder );
                waiter.constructPizza();
                Pizza pizza = waiter.getPizza();
        }
}
```

Real-time Example

- If we consider a house, we need to go through a step by step procedure. First, we need to set the basement. Next comes the pillars. Then you'll need bricks and cement etc. For building up a house, you'll need a combination of stuff like bricks, rod, iron rod, cement, and many other things.
- For laptop manufacturing purpose, there are several steps included like setting the memory hard disk and then LCD skins and then the number of other steps. After completion of these steps, a laptop as a final object can be created.

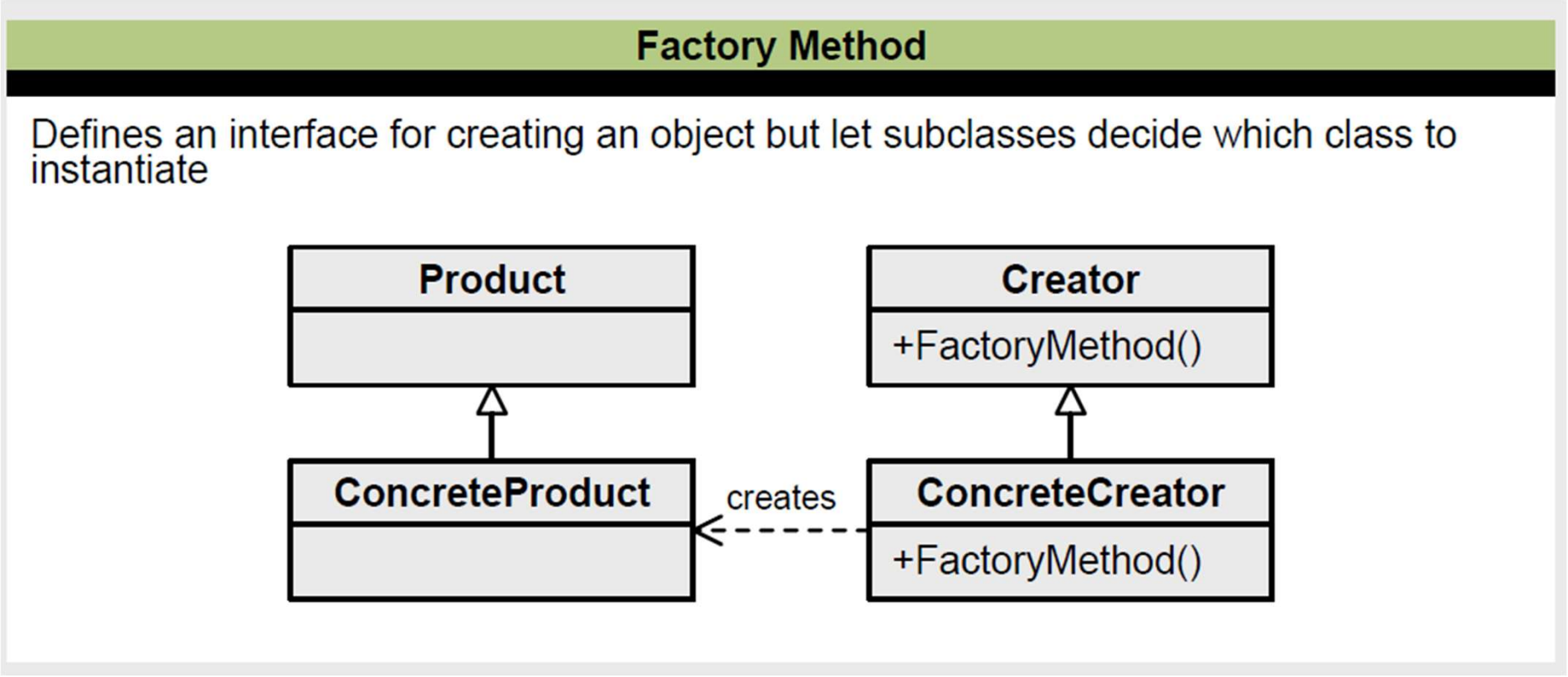
Real-time Example in Application

- Example Builder classes in java API:
  - java.lang.StringBuilder (unsynchronized)
  - java.lang.StringBuffer (synchronized)
  - java.nio.ByteBuffer (also CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)
  - javax.swing.GroupLayout.Group
  - All implementations of java.lang.Appendable

Factory Method

- Also known as – ***Virtual Constructor***
- Define an interface for creating an object, ***but let subclasses decide which class to instantiate***. Factory Method lets a class defer instantiation to subclasses
- Use the Factory Method pattern when:
  - Class cannot anticipate the class of objects it must create.
  - Class wants its subclasses to specify the objects it creates.
  - Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

Class Diagram



Real-time Example

- Let’s suppose, I am a customer of the shop, where I arrive to buy soap. I ask the shopkeeper for either a Hamam soap, a Rexona soap, or a Lux soap.
- The shopkeeper analyses my request and based on my request; he decides which soap needs to be returned.
- Here the shopkeeper is acting like a Factory.

Real-time Example in Application

- The following are the usage(s) of the Factory Method Pattern in JDK.
  - java.util.Calendar#getInstance()
  - java.util.ResourceBundle#getBundle()
  - java.text.NumberFormat#getInstance()
  - java.nio.charset.Charset#forName()
  - java.net.URLStreamHandlerFactory#createURLStreamHandler(String) (Returns singleton object per protocol)

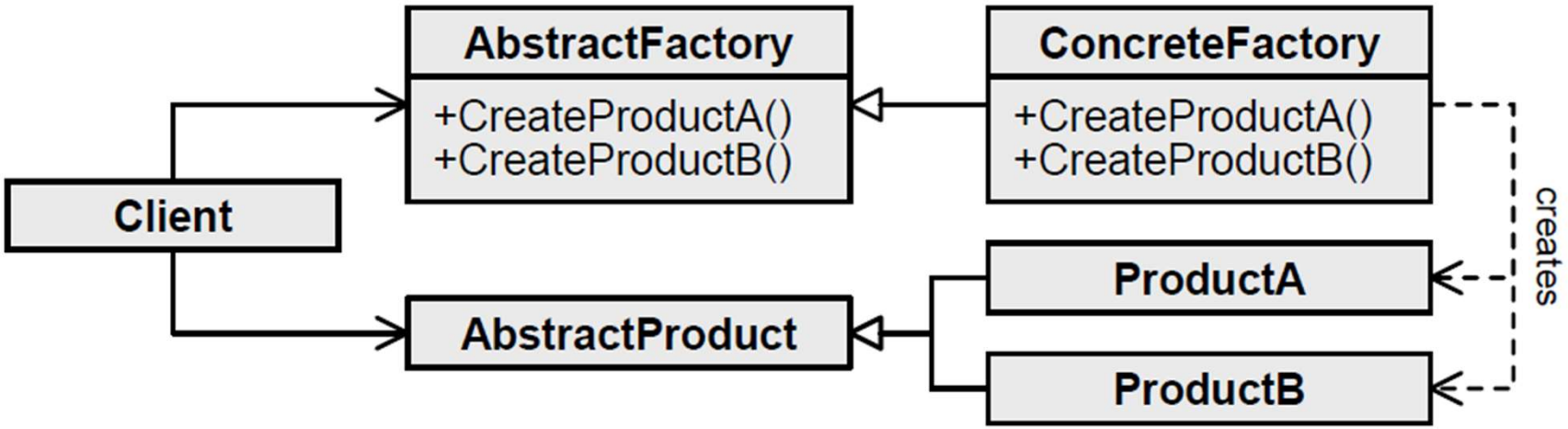
Abstract Factory

- Also known as – **Kit**
- It provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- The Abstract Factory pattern takes the concept of the Factory Method Pattern to the next level.
- An abstract factory is a class that provides an interface to produce a family of objects.

Class Diagram

Abstract Factory

Provides an interface for creating families of related or dependent objects without specifying their concrete classes



Real-time Example

- Suppose you are decorating your room with two different types of tables; one is made of wood and the other one of steel.
- For the wooden type, you need to visit a carpenter, and

  for the steel type, you may need to go to a metal shop.
- Both are table factories. So, based on demand, you decide what kind of factory you need.

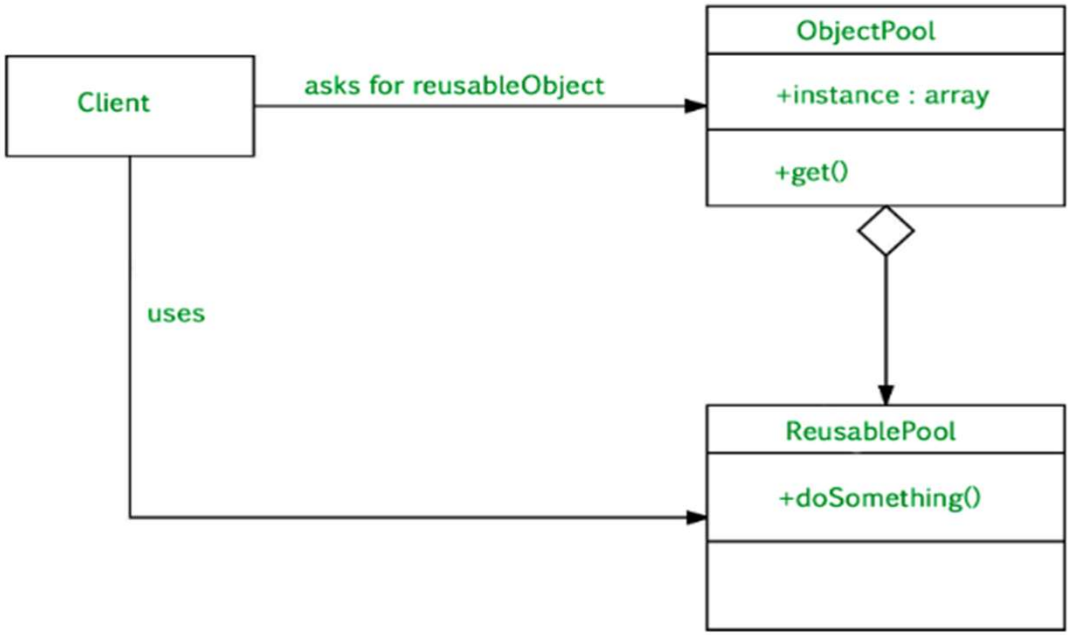
Real-time Example in Application

- Abstract Factory Pattern in JDK
  - java.util.Calendar#getInstance()
  - java.util.Arrays#asList()
  - java.util.ResourceBundle#getBundle()
  - java.sql.DriverManager#getConnection()
  - java.sql.Connection#createStatement()
  - java.sql.Statement#executeQuery()
  - java.text.NumberFormat#getInstance()
  - javax.xml.transform.TransformerFactory#newInstance()

Object Pool

- When objects are expensive to create and they are needed only for short periods of time it is advantageous to utilize the Object Pool pattern.
- The Object Pool provides a cache for instantiated objects tracking which ones are in use and which are available
- Objects in the pool have a lifecycle:
  - Creation
  - Validation
  - Destroy.

Class Diagram



Real-time Example in Application

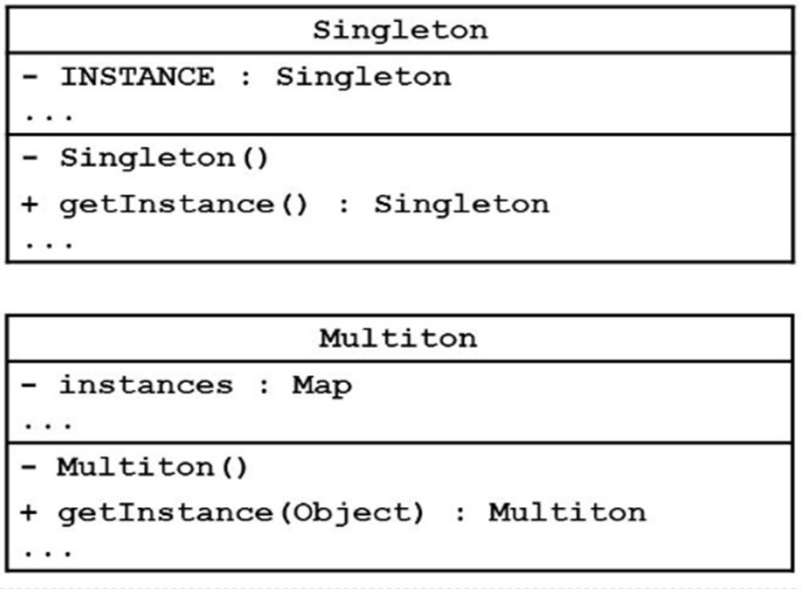
- Let’s take the example of database connections. It’s obvious that opening too many connections might affect the performance for several reasons:
  - Creating a connection is an expensive operation.
  - When there are too many connections opened it takes longer to create a new one and the database server will become overloaded.

Multiton

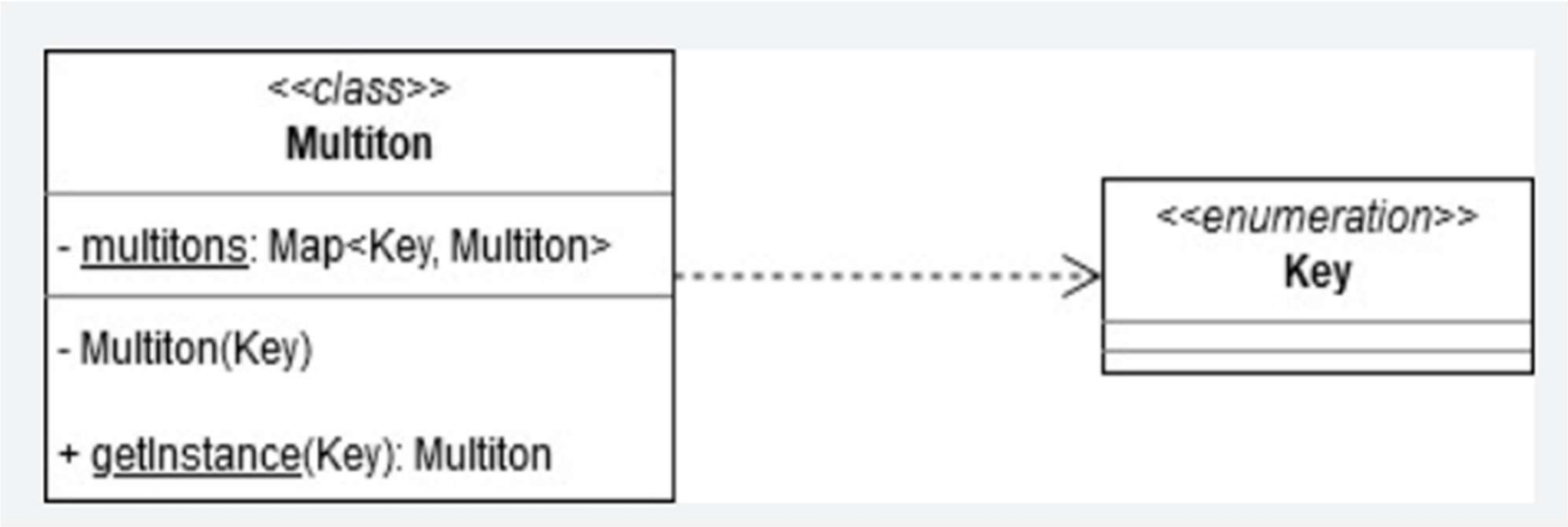
- Ensure a class only has a limited number of instances and provide a global point of access to them.
- Multiton pattern ensures there are a predefined amount of instances available globally.
- It can help to enforce the single responsibility principle, because it allows you to create separate instances of a class for different tasks or responsibilities. This can make it easier to design and maintain large, complex systems.
- Multitons, however, are **NOT** collected properly by conventional garbage collectors: this is harmless for applications that create a small number of mulitons but poses a scaling problem when Multitons are used to represent a large number of objects of fine granularity

Multiton Vs Singleton

- The multiton pattern is a design pattern which generalizes the singleton pattern. Whereas the singleton allows only one instance of a class to be created, the multiton pattern allows for the controlled creation of multiple instances, which it manages through the use of a map.



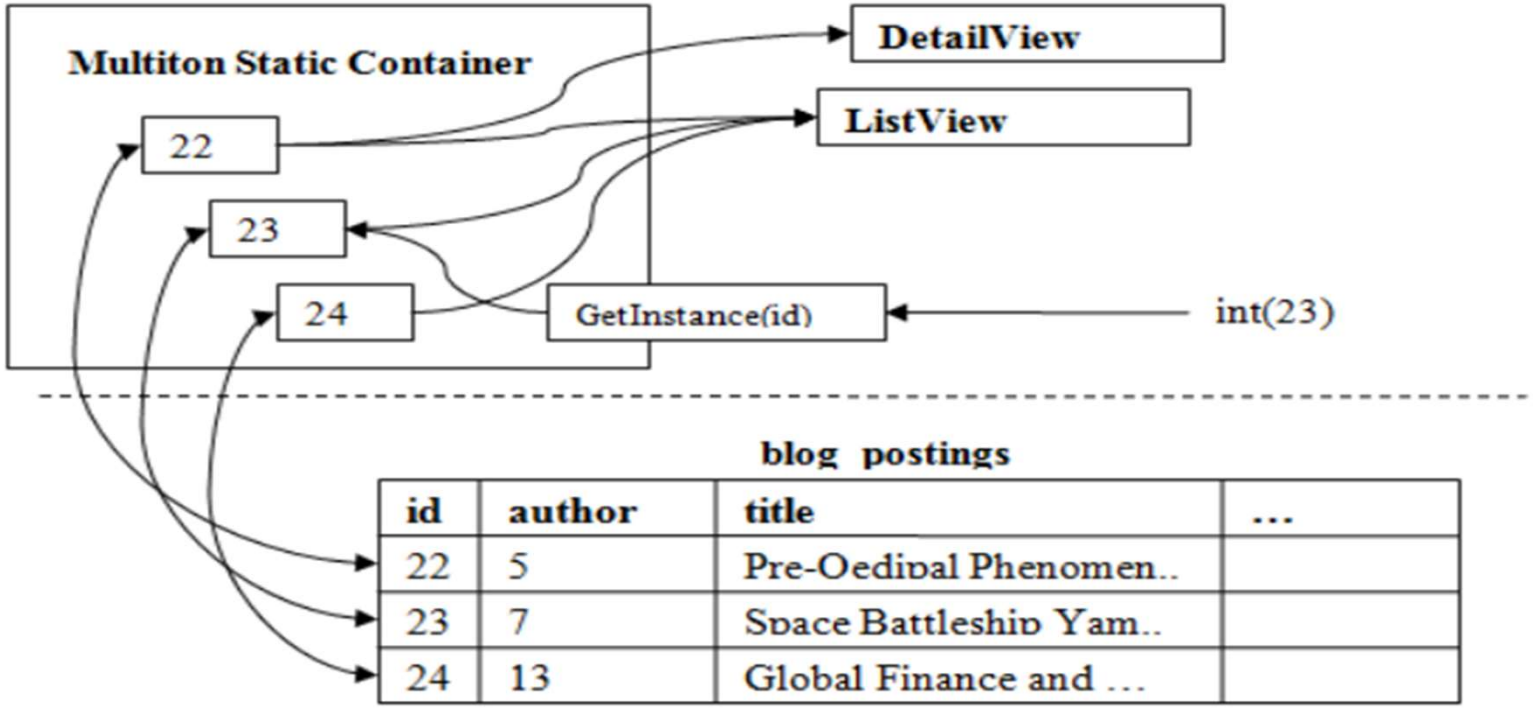
Class Diagram



Real-time Example in Application

- The Multiton pattern can be used in systems that store persistent data in a back-end store, such as a relational databases. The Multiton pattern can be used to maintain a set of objects are mapped to objects (rows) in a persistent store: it applies obviously to object-relational mapping systems, and is also useful in asynchronous RIA's, which need to keep track of user interface elements that are interested in information from the server.

Multiton


Multiton Static Container
22
23
24
GetInstance(id)
DetailView
ListView
int(23)
blog_postings
id
author
title
...
22
5
Pre-Oedipal Phenomen..
23
7
Space Battleship Yam..
24
13
Global Finance and ...

Structural Design Pattern

- ✓ Help to identify a simple and best way to realize relationships between entities.
- ✓ Focuses on how classes and objects can be composed to form a relatively large structure.
- ✓ Use inheritance or composition to group different interfaces or implementations.
- ✓ Your choice of composition over inheritance (and vice versa) can affect the flexibility of your software
- ✓ We need to loosely couple the interface and implementation.
- ✓ Advantages:
  - ✓ The efficiency of the application increases.
  - ✓ Reusability of the fragment of code also increases
  - ✓ The structure of the applications becomes clean and simpler.

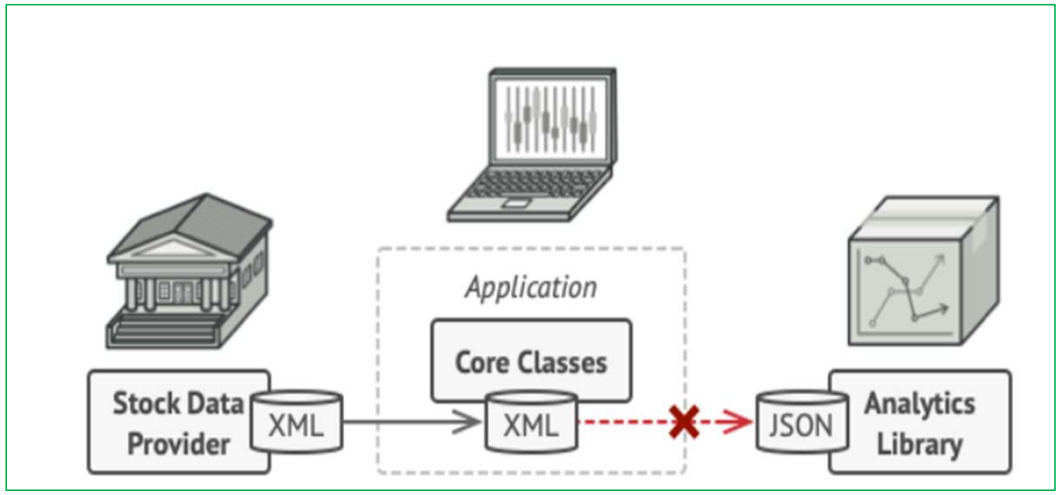
Structural Design Patterns

- Adapter. Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.
- Bridge. Decouples an abstraction so two classes can vary independently.
- Composite. Takes a group of objects into a single object.
- Decorator. Allows for an object's behavior to be extended dynamically at run time.
- Facade. Provides a simple interface to a more complex underlying object.
- Flyweight. Reduces the cost of complex object models.
- Proxy. Provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity.

Adapter

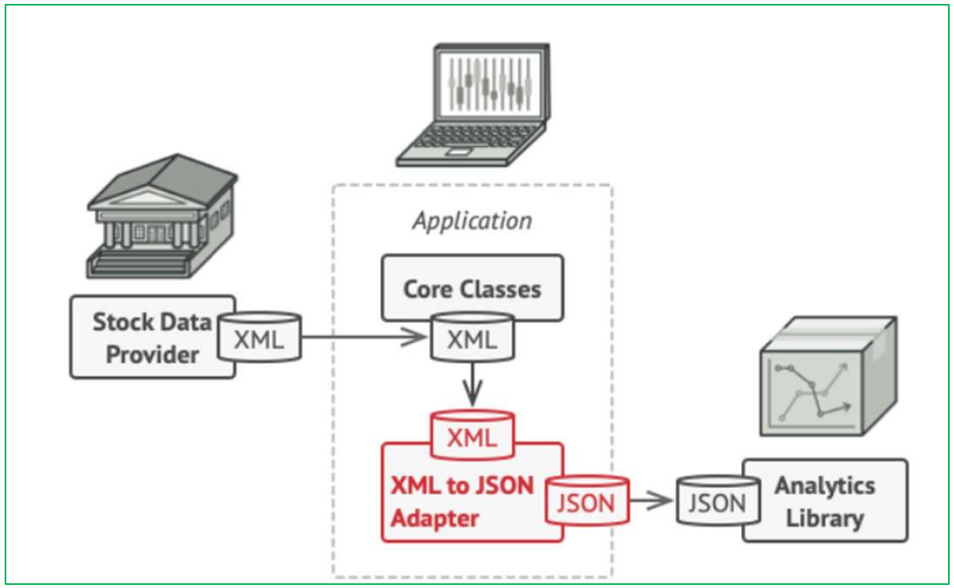
- Also known as –Wrapper
- The adapter pattern is an adaptation between classes and objects.
- There are two types of adapters, the object adapter, and the class adapter. So far, we have seen the example of the **object adapter which use object's composition**, whereas, the **class adapter relies on multiple inheritance** to adapt one interface to another.
- As Java does not support multiple inheritance, we cannot show you an example of multiple inheritance, but you can keep this in mind and may implement it in one of your favorite Object Oriented Languages like c++ which supports multiple inheritances.
- To implement a class adapter, an adapter would inherit publicly from Target and privately from Adaptee. As the result, the adapter would be a subtype of Target, but not for Adaptee

Adapter Pattern

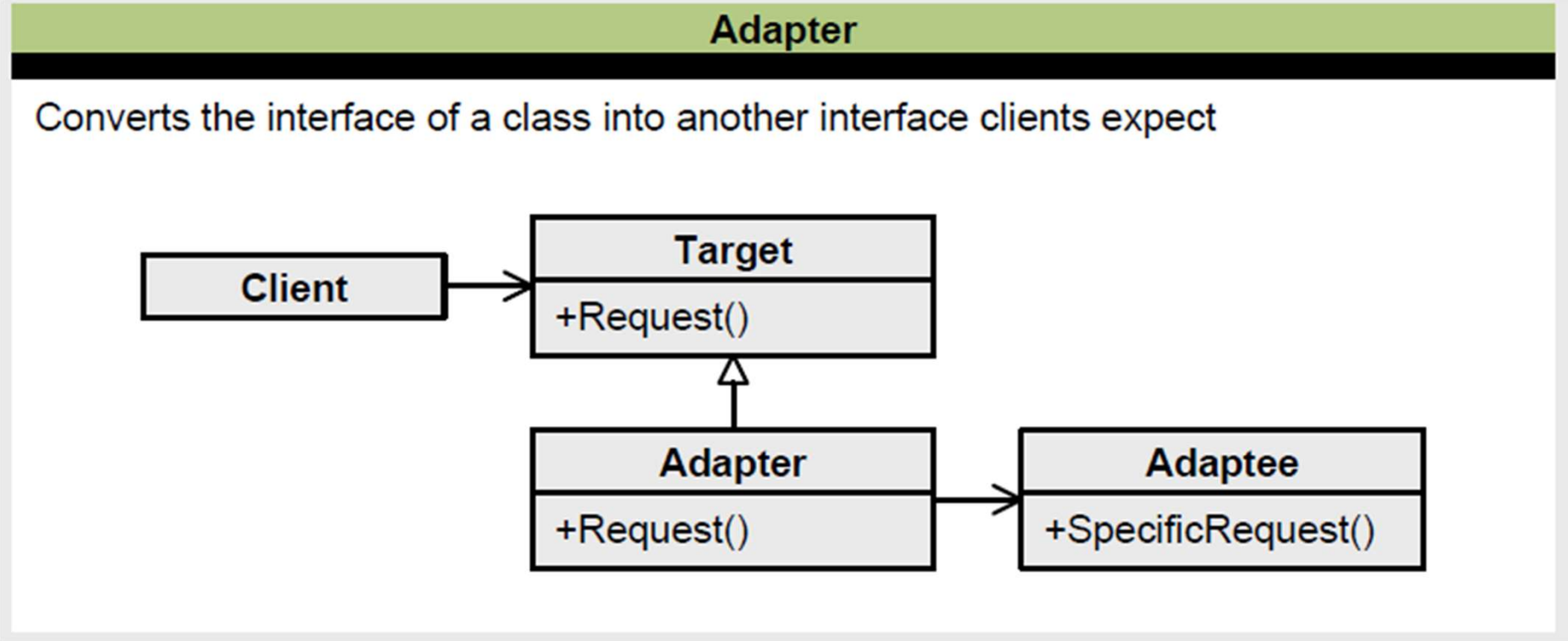


Existing System

Expected New System



Class Diagram - Adapter



Real-time Example

- For charging your phone, you need to connect your phone with a phone charging adapter. In India, 220 V is the power output for household purposes. So, the phone charging adapter will cover the 220 V into 9 V for phone applications.

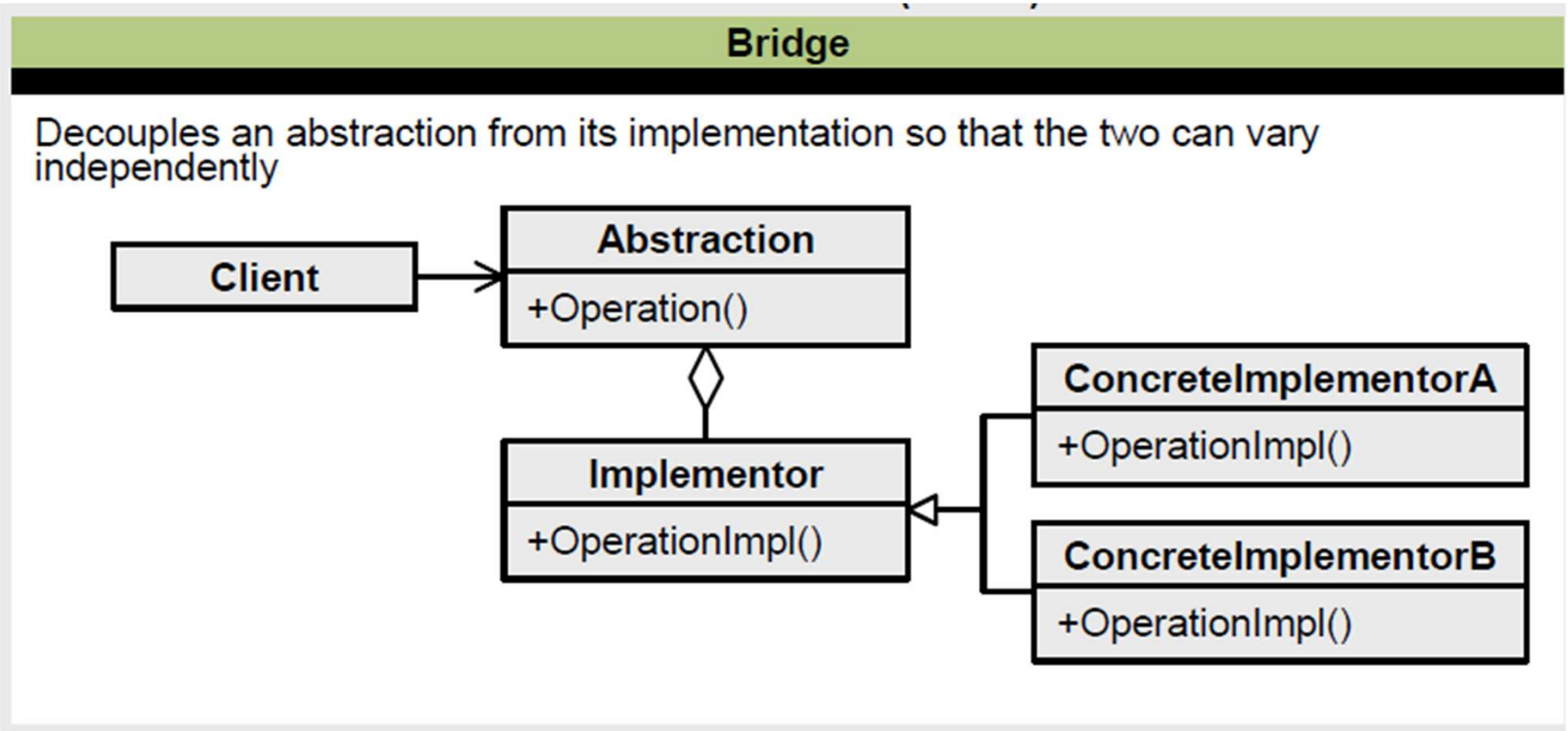
Real-time Example in Application

- People who write Java applications are not aware of the databases behind them. It may be “Oracle”, “MySQL” or “Sybase”.
- If a new vendor is providing a better database, then the database behind that would change. So, we aren’t aware of the database running behind. We need to write an application, using some commands and statements. The adapter class would handle those problems.
- Let’s suppose we are trying to migrate from “Oracle” to “Sybase”, then the adapter class will be changed. So this would be handled by this adapter class.
- To integrate a legacy code with a new code, or when changing a 3rd party API in the code which is due to incompatible interfaces of the two objects which do not fit together.
- Changing the payment gatway, converting xml data to json data.
- Following are the examples in Java API where adapter pattern is used:
  - **java.util.Arrays#asList()**
  - **java.io.InputStreamReader(InputStream) (returns a Reader)**
  - **java.io.OutputStreamWriter(OutputStream) (returns a Writer)**
  - **javax.xml.bind.annotation.adapters.XmlAdapter #marshal() and #unmarshal()**

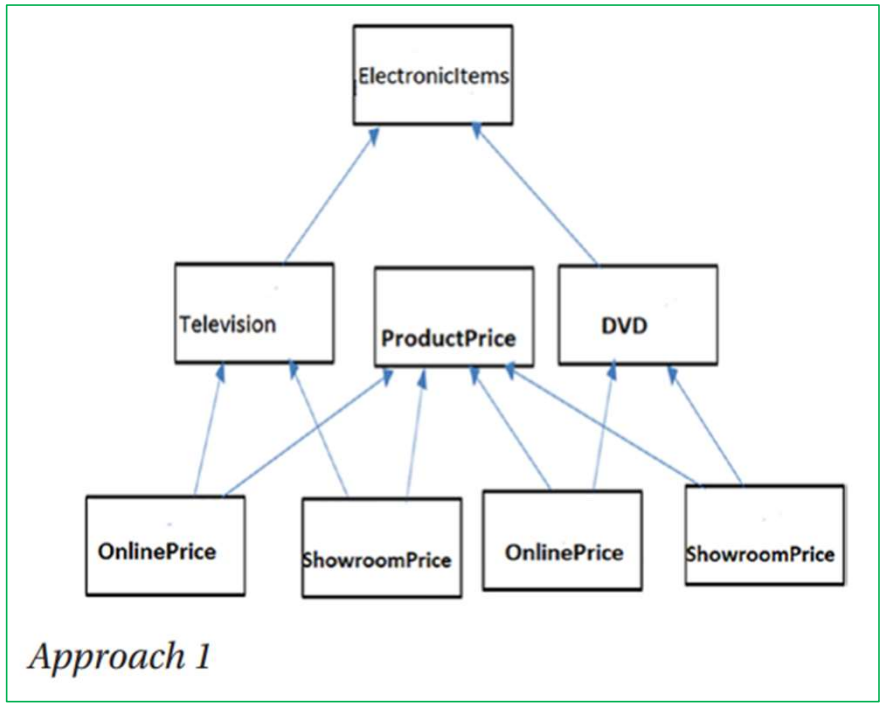
Bridge Pattern

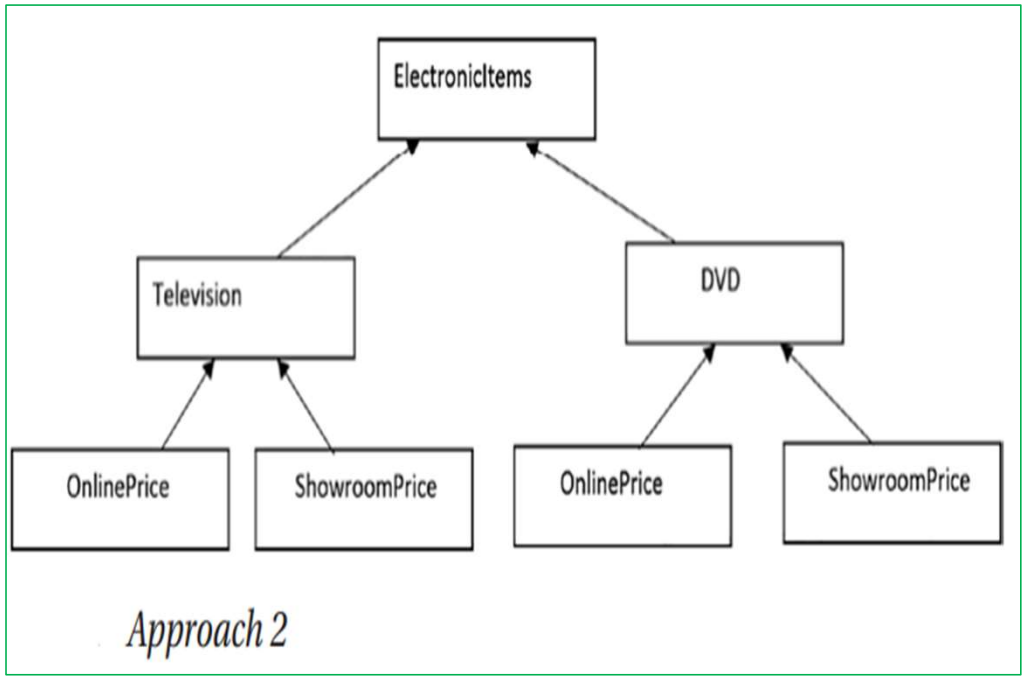
- Also known as – Handle/Body pattern
- It decouples an abstraction from its implementation so that the two can vary independently.
- You can maintain two different inheritance hierarchies. They can grow or shrink without affecting each other

Class Diagram

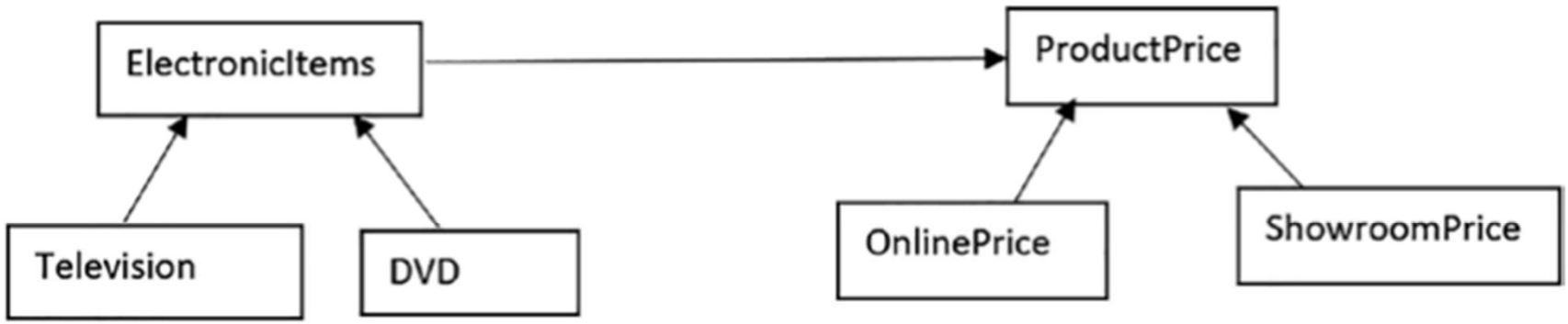


Bridge Pattern





Bridge Pattern



Maintaining two separate hierarchies using the Bridge pattern

Real-time Example

- In a software product development company, the development team and the marketing team both play crucial roles. Normally, the marketing team does a market survey and gathers customer requirements. The development team implements those requirements in the product to fulfill customer needs.
- Any change (say, in the operational strategy) in one team should not have a direct impact on the other team. In this case, you can think of the marketing team as playing the role of the bridge between the clients of the product and the development team of the software organization

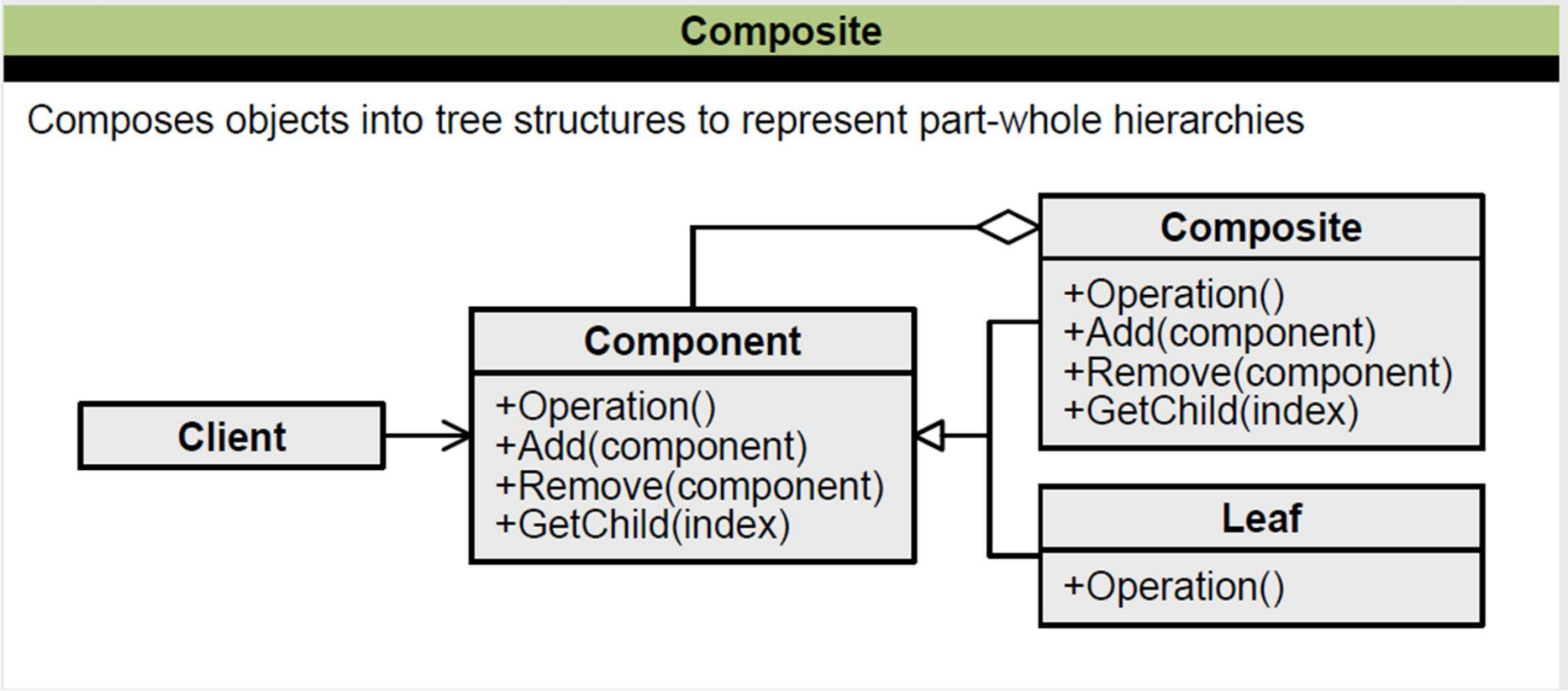
Real-time Example in Application

- In Java, you may notice the use of JDBC, which provides a bridge between your application and a particular database. For example, the java.sql.DriverManager class and the java.sql.Driver interface can form a bridge pattern where the first one plays the role of abstraction and the second one plays the role of implementors. There are many concrete implementors such as com.mysql.cj.jdbc.Driver and oracle.jdbc. driver.OracleDriver.

Composite Design Pattern

- Also known as –"has-a" relationship among objects
- It composes objects into tree structures to represent part-whole hierarchies. The Composite pattern lets clients treat individual objects and compositions of objects uniformly.
- If you are familiar with a tree data structure, you will know a tree has parents and their children. There can be multiple children to a parent, but only one parent per child.
- In Composite Pattern, elements **with** children are called as **Nodes**, and elements **without** children are called as **Leafs**.
- Adv: can easily add a new component to the architecture or delete an existing component from the architecture

Class Diagram



Real-time Example

- Apart from the previous example, you can also think of an organization that consists of many departments. In general, an organization has many employees. Some of these employees are grouped to form a department and those departments can be further grouped to build the final structure of the organization.

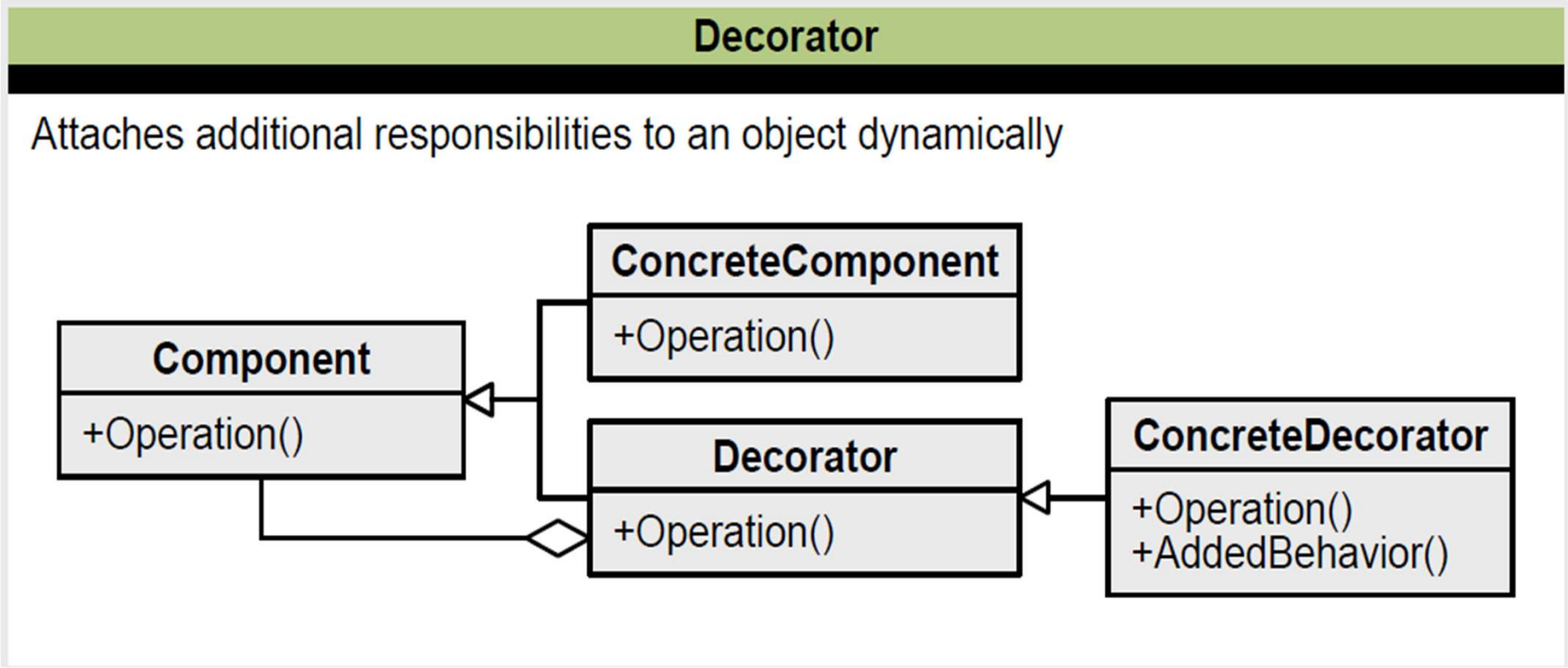
Real-time Example in Application

- XML files are very common examples where you notice such tree structures.
- In Java, note the use of the generic Abstract Window Toolkit (AWT) container object. It is a component that can contain other AWT components. For example, in the java.awt.Container class (which extends java.awt.Component) you can see overloaded versions of the add (...) method. These methods accept another Component object to proceed further.

Decorator Pattern

- Also known as –
- It attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Class Diagram



Real-time Example

- If we take a plain pizza as an example and a customer orders for chicken pizza, so a chef will add the additional responsibility on the plain pizza which is a chicken layer in this case.
  So, when the object should be treated dynamically at the run time based on the criteria, then we should choose Decorator Design Pattern.

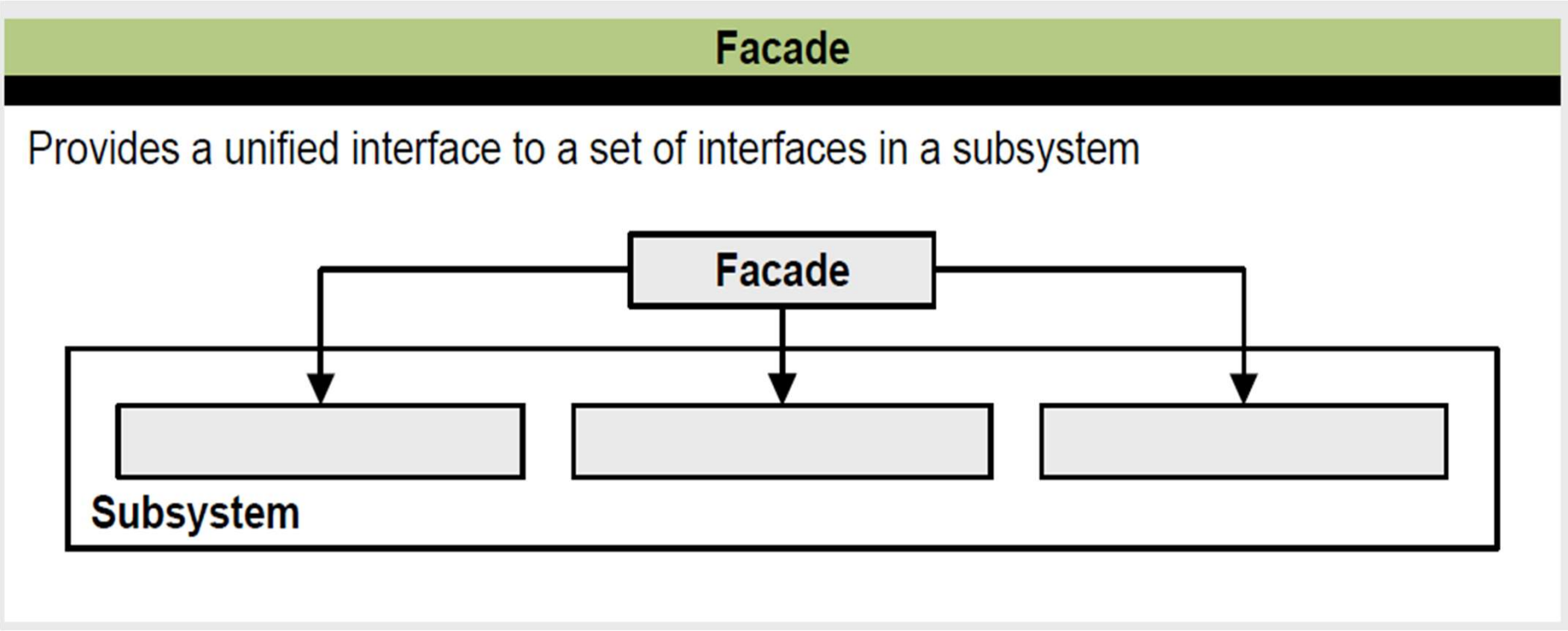
Real-time Example in Application

- Decorator Design Pattern in Java •
  java.io.BufferedInputStream(InputStream) •
  java.io.DataInputStream(InputStream) •
  java.io.BufferedOutputStream(OutputStream) •
  java.util.zip.ZipOutputStream(OutputStream) •
  java.util.Collections#checked[List|Map|Set|SortedSet|SortedMap]()

Façade Pattern

- Also known as –
- In French, Facade means **frontage**. So whatever we see is a facade.
- The primary motive behind **Facade Design Pattern** is a simple interface of a complex system. The system can be complex, but the interface has to be simple to operate the operations.
- The single class represents the entire system. Your system can contain an N number of subsystems, but the user would see it as a single system.
- The inner implementation should handle the entire complexity so the user should not be worried about that.

Class Diagram



Real-time Example

- A building that has ten floors with 2-3 rooms each. For a quality purpose, you'd need cement and bricks, etc. Here to build the structure you need to maintain many things which is a complex process.
  You will give this task to the building contractor who will manage these things in a better manner. You'll pay him 5,00,000 Rupees to that contractor, and you will explain him with your ideas. Then that contractor will handle these complicated things on behalf of us.

Real-time Example in Application

Following are the examples in Java API where Façade pattern is used:

- ✓ javax.faces.context.FacesContext, it internally uses among others the abstract/interface types LifeCycle, ViewHandler, NavigationHandler and many more without that the enduser has to worry about it (which are however overrideable by injection).
- ✓ javax.faces.context.ExternalContext, which internally uses ServletContext, HttpSession, HttpServletRequest, HttpServletResponse, etc.

Flyweight Pattern

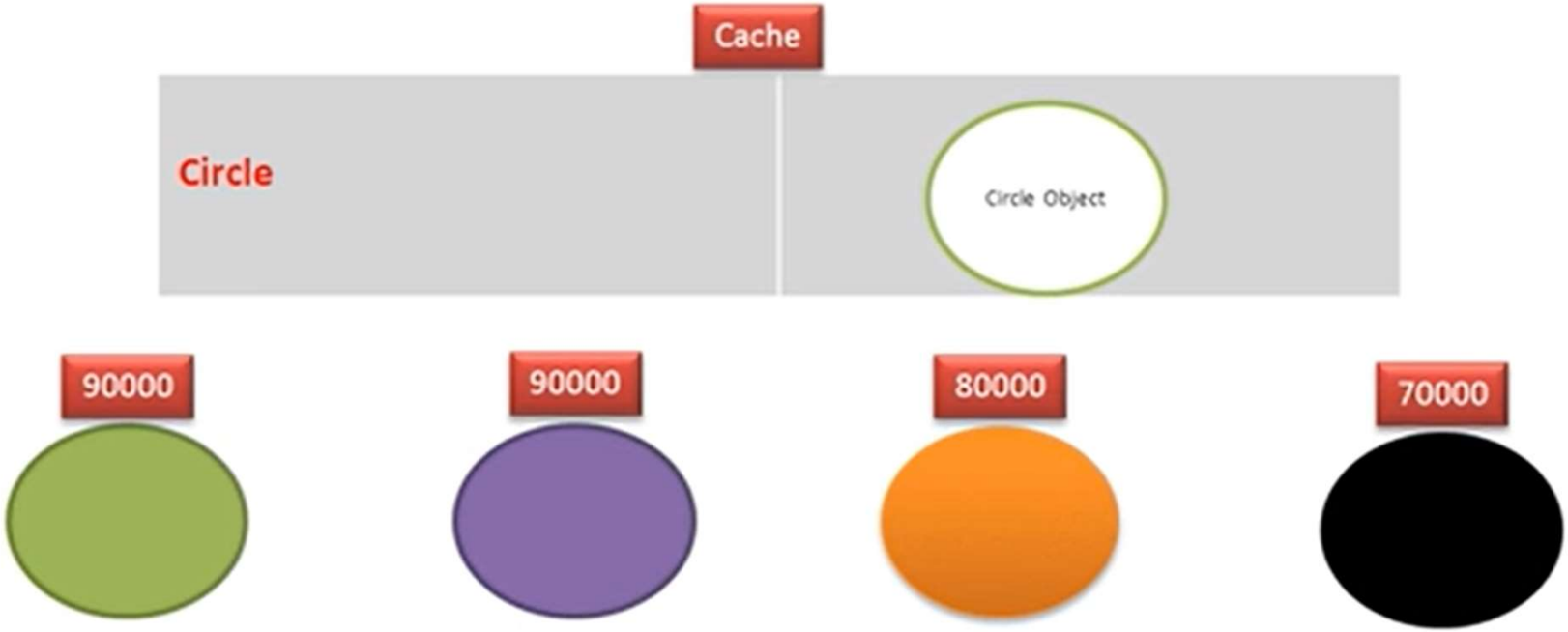
- Use sharing to support large numbers of fine-grained objects efficiently.
- Used when there is a need to create a large number of objects of similar nature. A large number of objects consumes a large amount of memory and this pattern provides a solution for reducing the load on the memory by sharing objects.
- This pattern is used to reduce the number of objects created to decrease the memory footprint and increase performance.
- Flyweight pattern try to reduce already existing similar kind of objects by storing them and creating new objects when no matching objects are found.
- This pattern has 2 states
  - Intrinsic – states that are CONSTANTS and are stored in the memory
  - Extrinsic- states that are NOT CONSTANTS and need to be calculated on the fly and therefore not stored in the memory.
- flyweights make one look like many

Flyweight Pattern

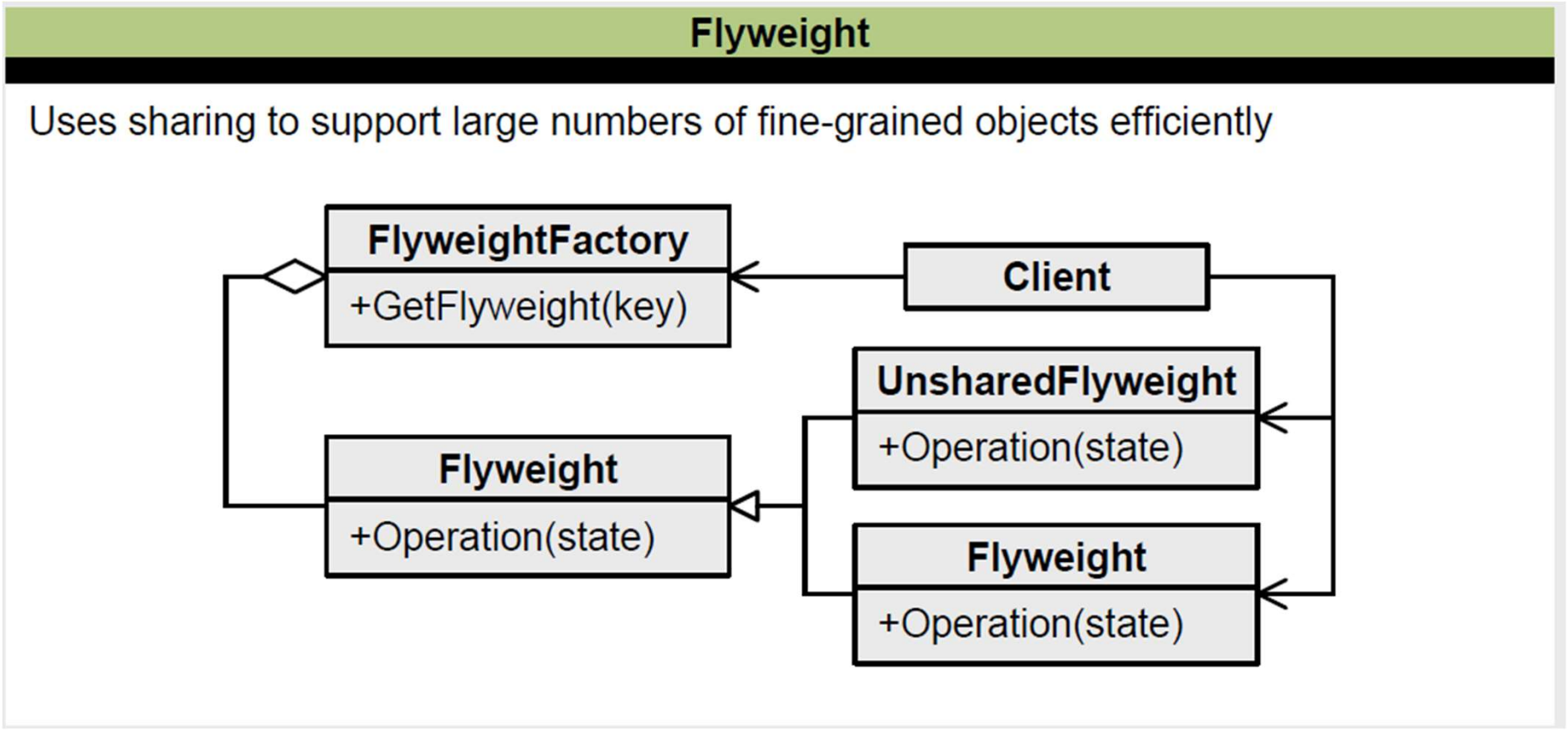
Advantages:

- You can reduce the memory consumption of heavy objects that can be controlled identically.
- You can **reduce the total number of unique objects** in the system.
- You can maintain centralized states of many “virtual” objects.

Example : Flyweight Pattern



Class Diagram



Real-time Example

- Suppose a company needs to print business cards for its employees. In this case, what is the starting point? The print department can create a common template where the company logo, address, and other details are already printed (intrinsic) and later the company adds the particular employee details (extrinsic) on the cards.

Real-time Example in Application

You can also investigate the definition of the intern() method of String in the same way to get the following information:

"A pool of strings, initially empty, is maintained privately by the class String. When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned.

Otherwise, this String object is added to the pool, and a reference to this String object is returned."

Proxy Pattern

- There are situations where you want to restrict direct communication between an intended object and the outside world. It is a substitute or placeholder
- Types:

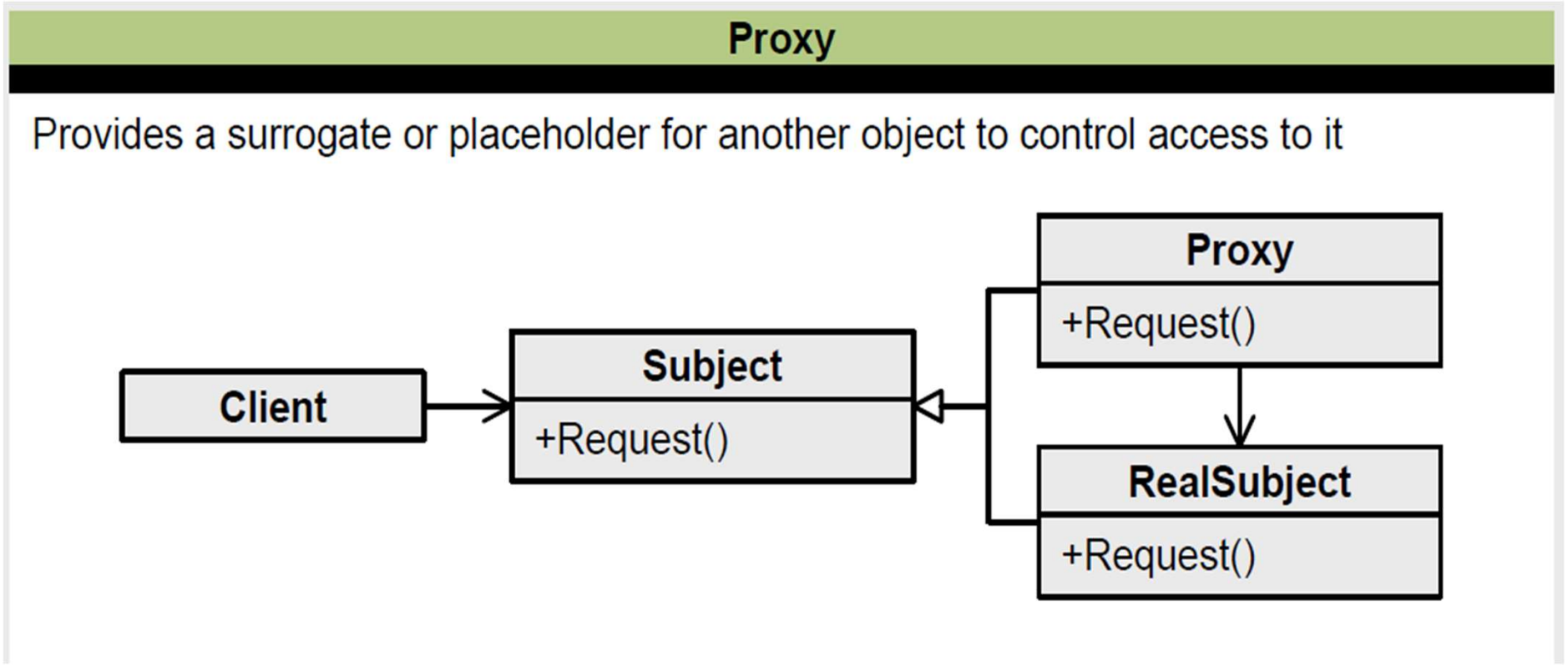
1. A remote proxy provides a local representative for an object in a different address space.

2. A virtual proxy creates expensive objects on demand.

3. A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.

4. A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed.

Class Diagram



Real-time Example

- Consider the case when a particular teacher in a school is absent due to some unavoidable circumstances. The school authority can send another teacher as a replacement to teach the classes.
- An ATM implementation can hold proxy objects for bank information that actually exists on a remote server

Real-time Example in Application

- A very common use of a proxy is when users do not want to disclose the true IP address of their machine; instead, they want to make it anonymous.
- In the java.lang.reflect package, the Proxy class and the InvocationHandler interface support a similar concept.
- The package java.rmi.* also provides methods through which an object in one Java virtual machine can invoke methods on an object that resides in a different Java virtual machine.

Behavioural Design Pattern

- Focus on how the classes and objects collaborate with each other to carry out their responsibilities
- It focuses on the communication between the objects and dependency between objects.
- Concerned with algorithms

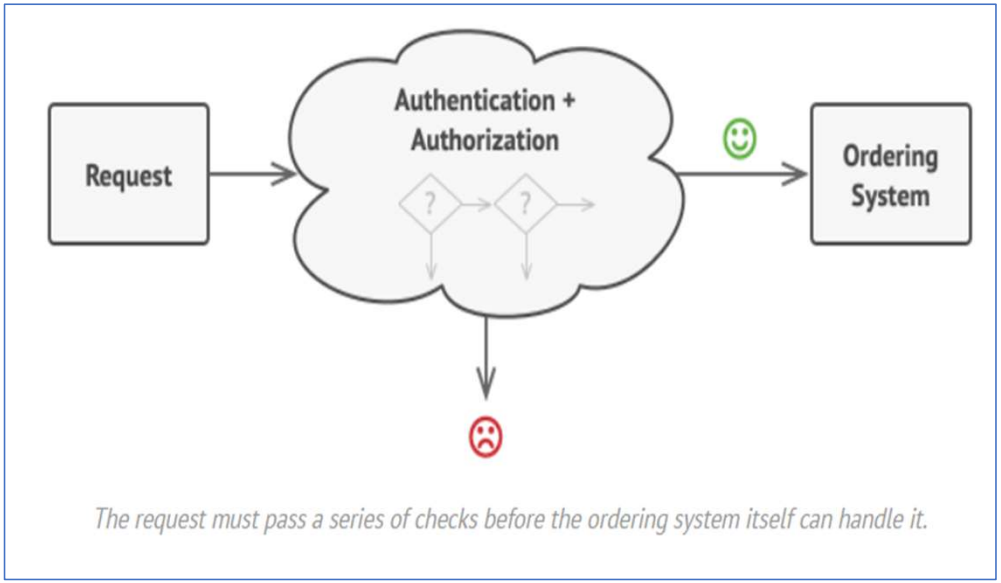
Behavior Design Patterns

- Chain of Responsibility. Delegates commands to a chain of processing objects.
- Command. Creates objects which encapsulate actions and parameters.
- Interpreter. Implements a specialized language.
- Iterator. Accesses the elements of an object sequentially without exposing its underlying representation.
- Mediator. Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- Memento. Provides the ability to restore an object to its previous state.
- Observer. Is a publish/subscribe pattern which allows a number of observer objects to see an event.
- State. Allows an object to alter its behavior when its internal state changes.
- Strategy. Allows one of a family of algorithms to be selected on-the-fly at run-time.
- Template Method. Defines the skeleton of an algorithm as an abstract class, allowing its sub-classes to provide concrete behavior.
- Visitor. Separates an algorithm from an object structure by moving the hierarchy of methods into one object.

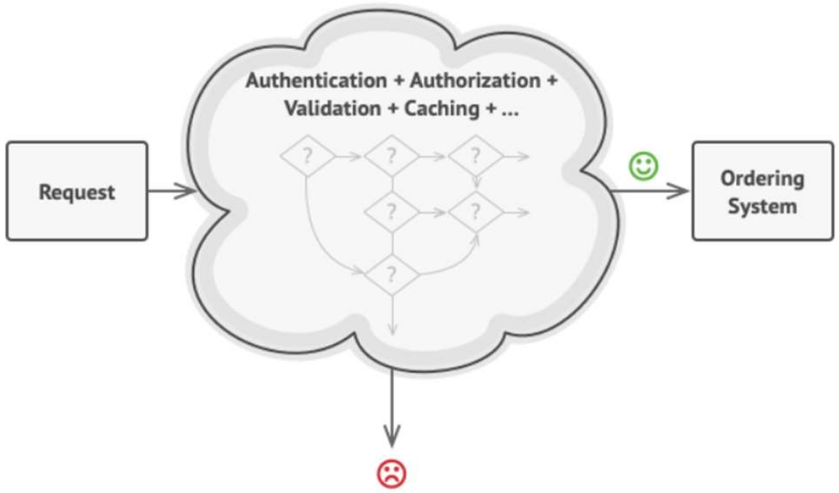
Chain of Responsibility

- It passes the request between multiple objects via handlers
- Avoid coupling the sender of a request to its receiver by giving more than one receiver object a chance to handle the request.
- Chain the receiving objects and pass the request along with the chain until an object handles it.
- It creates a chain of receiver objects for a request.
- Each receiver contains a reference to another receiver. If one receiver cannot handle the request then it will pass to the next receiver.
- (One receiver handles a request in the chain) or (One or more receivers in the chain handles a request).

Chain of Responsibility – Problem

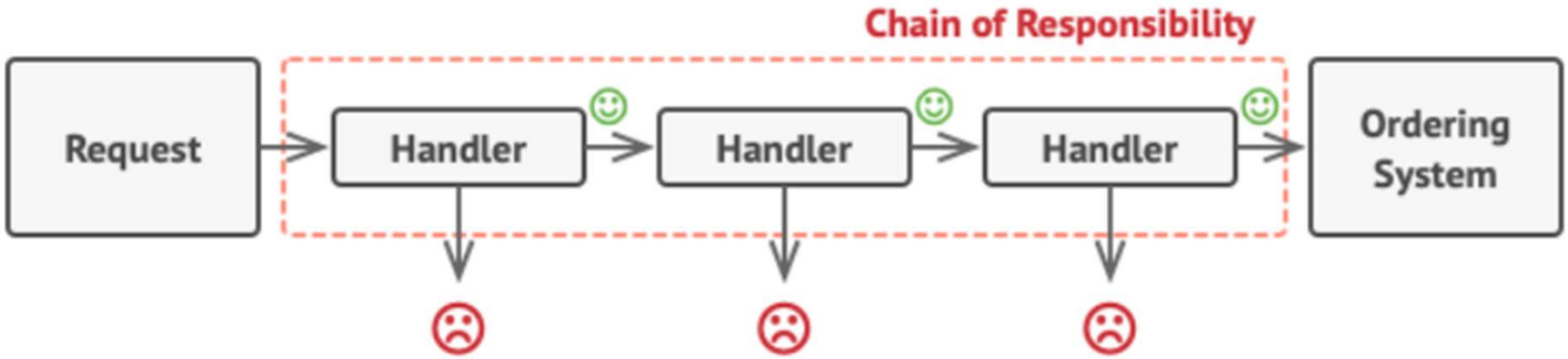


The request must pass a series of checks before the ordering system itself can handle it.



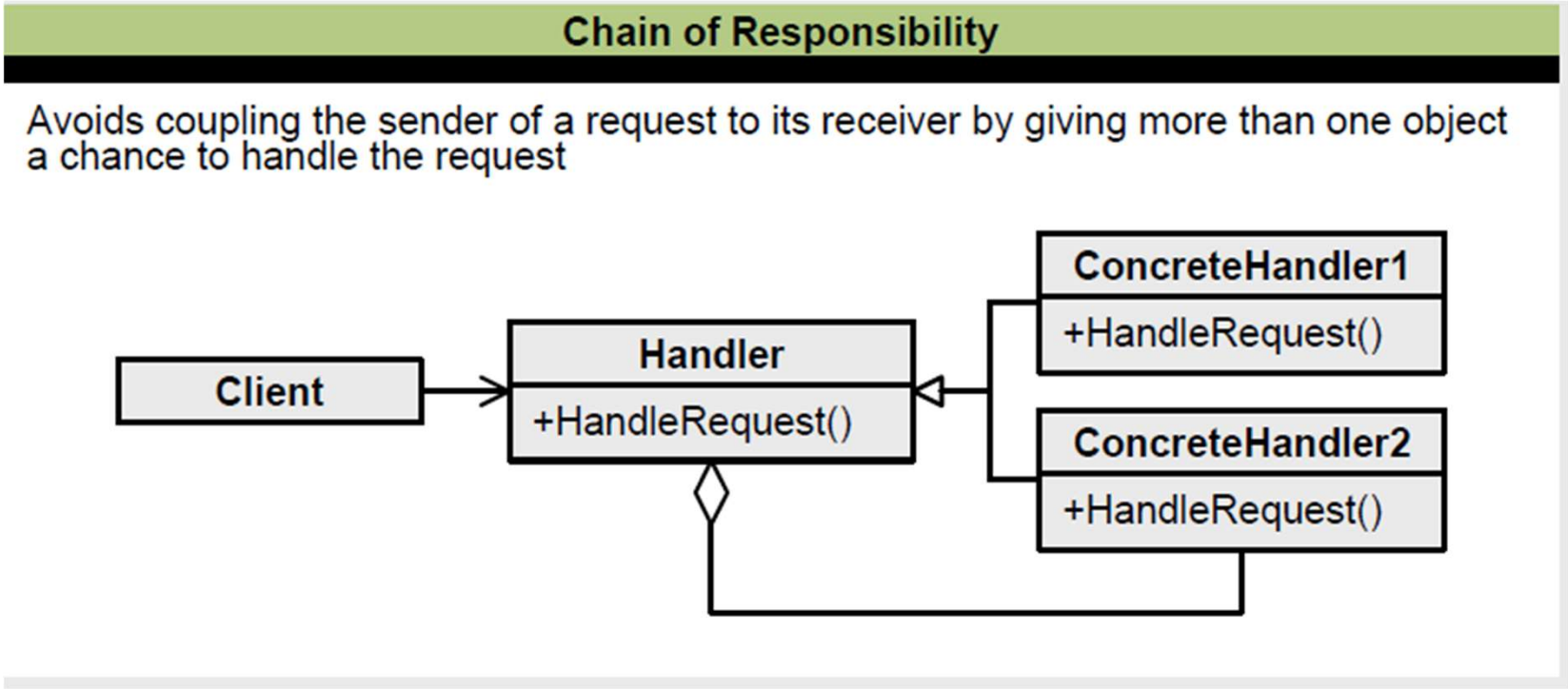
The bigger the code grew, the messier it became.

Chain of Responsibility –Solution

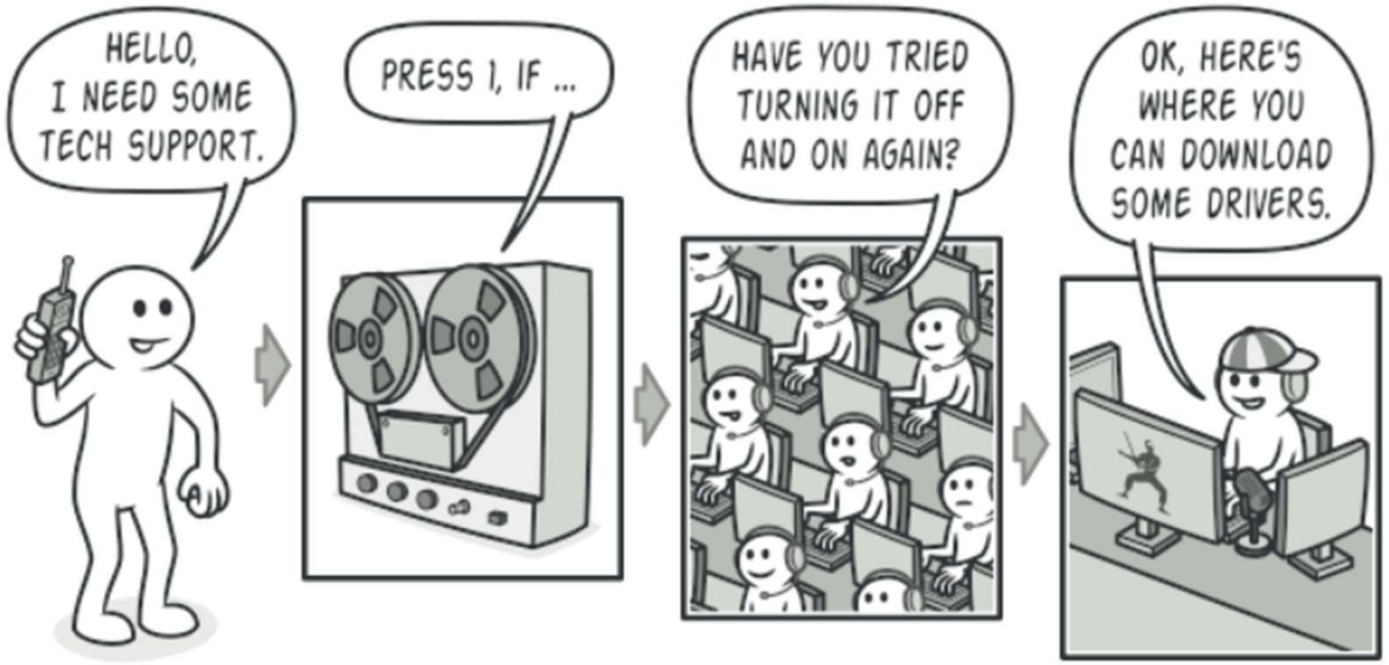


Handlers are lined up one by one, forming a chain.

Class Diagram



Real-time Example



A call to tech support can go through multiple operators.

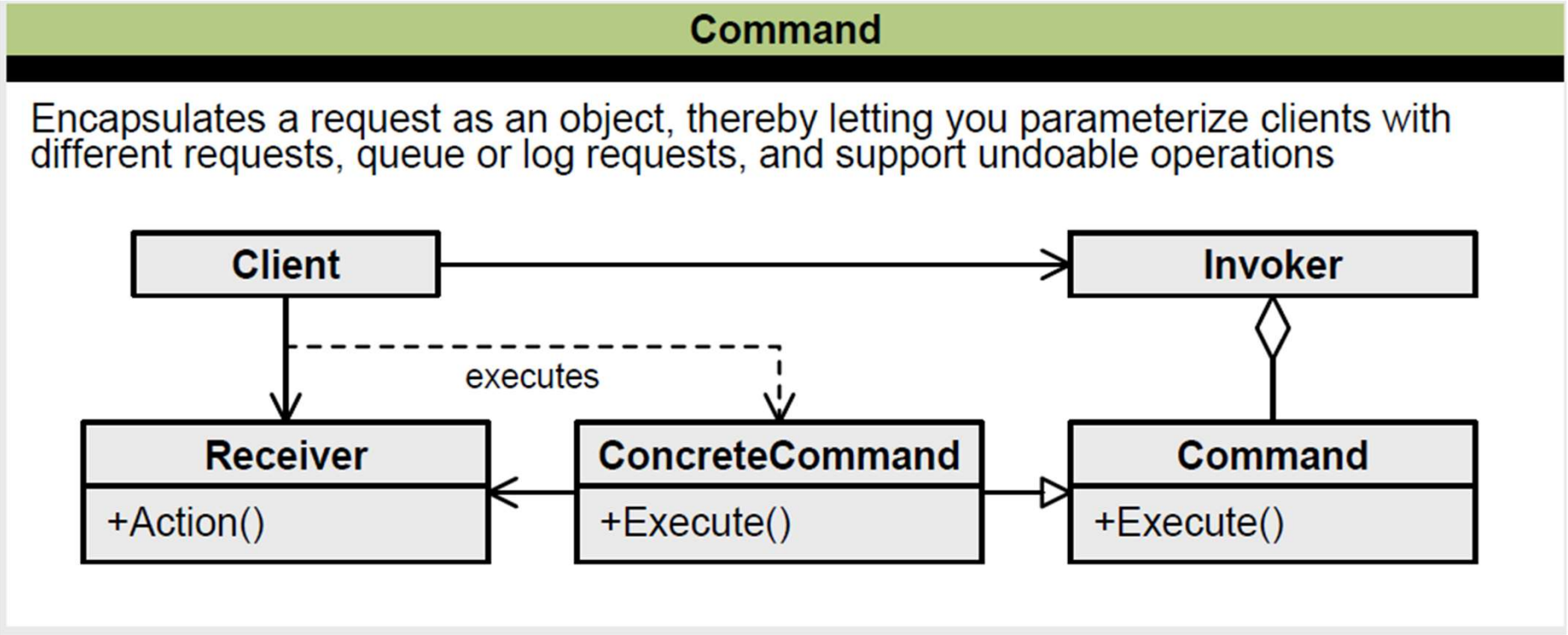
Real-time Example in Application

- Let’s take an example if you are using Amazon website and you are trying to log into Amazon website using username and password. The client sends your credentials to the server. So, here client isn’t aware of Server details.Chain of servers would handle that request. Let’s consider a first server which handles the permission-related job. The next server handles the billing process and the next n number of servers handle some other jobs.Here, the single request has been handled by the chain of servers

Command Pattern

- It encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Using this pattern, you encapsulate a method invocation process
- This pattern is often used in a multithreaded environment.
- Advantages:
  - The Command pattern decouples the sender and receiver objects
  - An invoker can handle different types of objects that perform different types of works
  - New commands can be added without affecting the existing system
  - Most importantly, you can support the undo (and redo) operations.
  - It should be noted that once you create a command object, it does not mean that the computation starts immediately. You can schedule it for later or place it in a job queue and execute it later.

Class Diagram



Real-time Example

- When you draw a picture, you may need to redraw (undo) some parts of it to make it better.

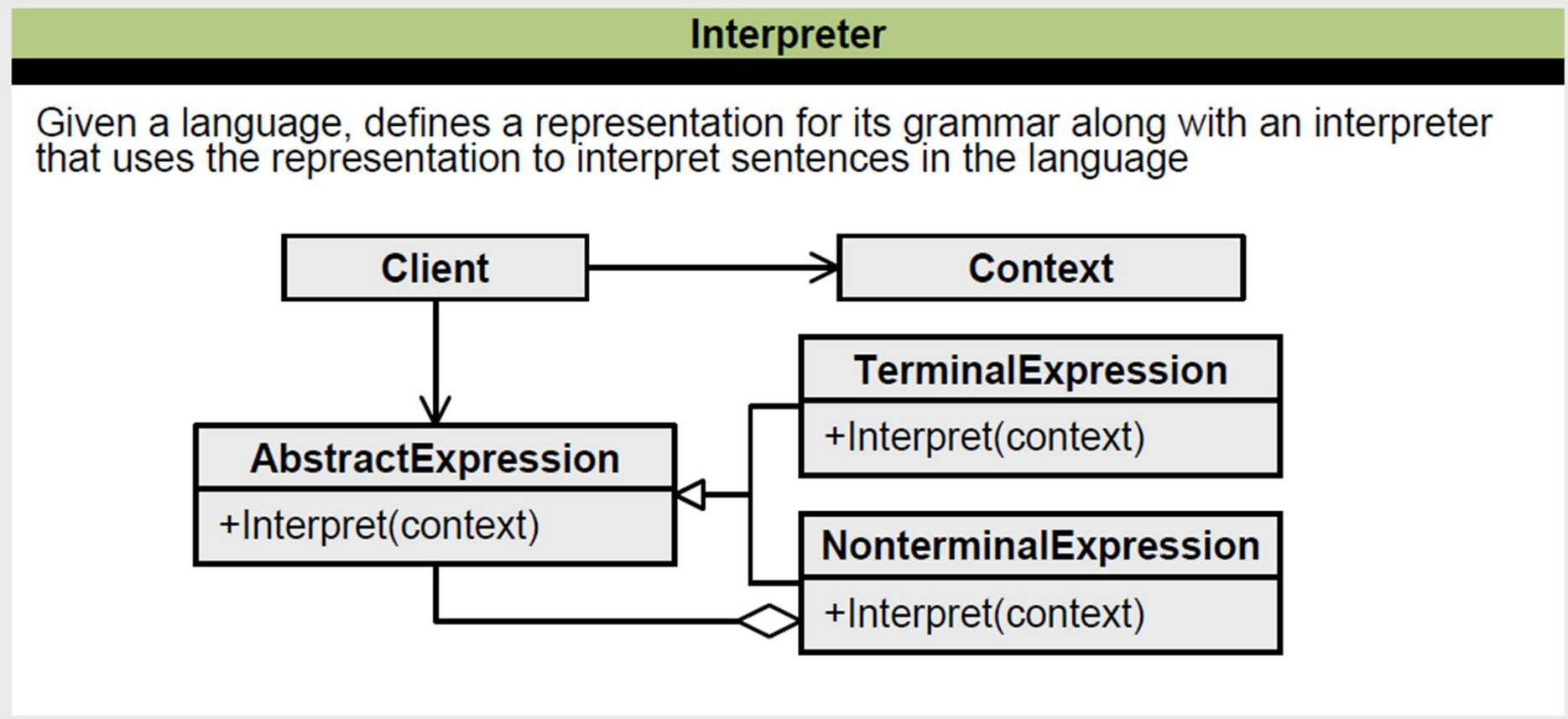
Real-time Example in Application

- When you use the java.lang.Runnable interface and use the run() method to invoke the target receiver's method, you use Command design pattern.

Interpreter Design Pattern

- It provides a way to evaluate language grammar or expression.
- It is useful for developing domain-specific languages or notations.
- It defines the ability to set a language's grammar.
- Java compiler that interprets Java source code into byte-code.
- Defines a grammatical representation for a language and an interpreter to interpret the grammar.
- It is widely used in compilers that are implemented with Object Oriented languages.

Class Diagram



Real-time Example

- Let's consider Google translator; it will translate from any language to any language. Here, Google translator is acting as an interpreter which translates from one language (English) to another language (Tamil) on behalf of us. It follows the language rules which means Grammar for translating.

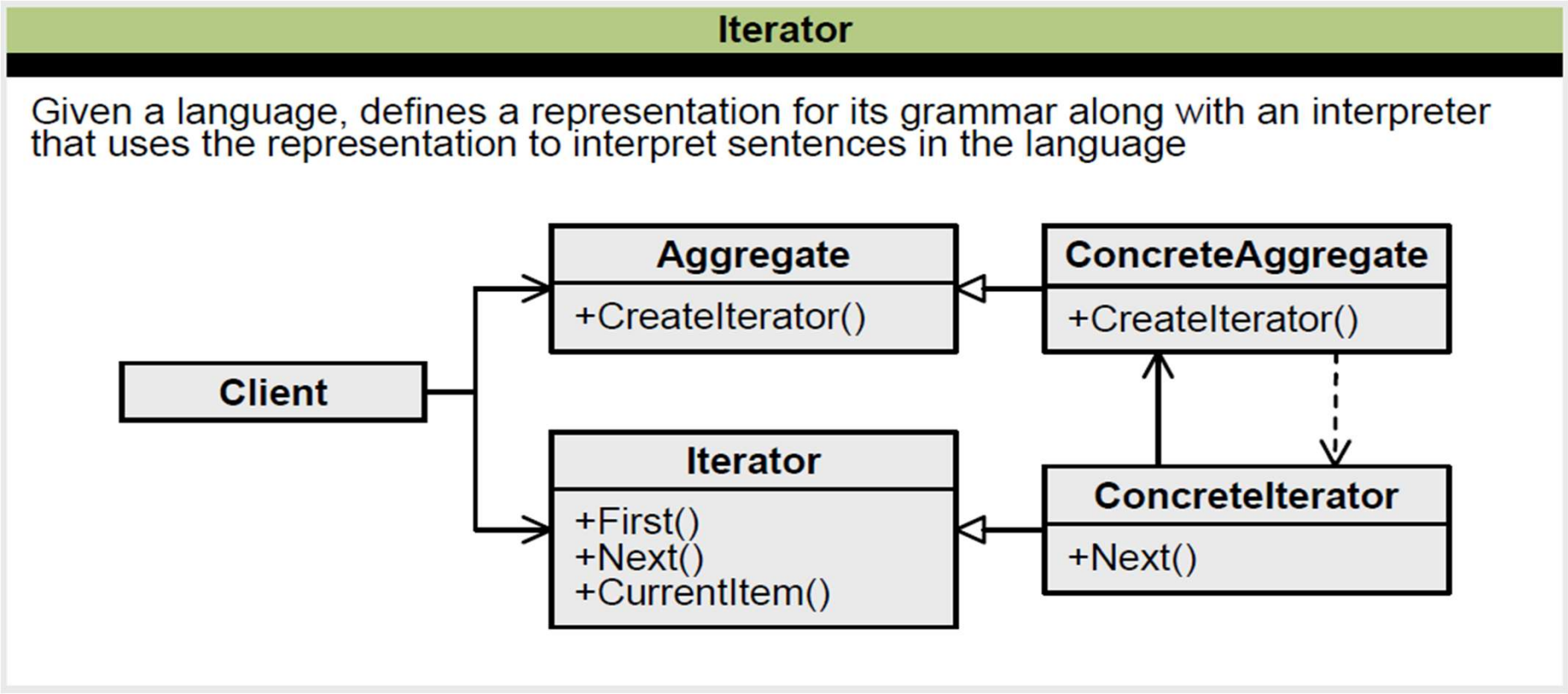
Real-time Example in Application

- If you notice the java.util.Format class, you will learn that this abstract class is used for formatting locale-sensitive information such as dates, messages, and numbers. So, any subclass of it can be considered to use this pattern.

Iterator Pattern

- It accesses the elements of a collection object sequentially without any need to know its underlying representation.
- The collection can be a List, Set, Array and it can be anything.
- In OOPs, Iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements.
- Abstract the details of traversing the collections. Example: Different types of collections in an application such as an array, linked list, etc. We can traverse and iterate the elements without knowing the actual implementation.

Class Diagram



Real-time Example

- Suppose your company has decided to promote some employees based on their performances. All the managers get together and set a common criterion for promotion. Then they iterate over the records of the employees one by one to mark the potential candidates for promotion.

Real-time Example in Application

You may have already used Java's built-in Iterator interface, java. util.Iterator. When you use interfaces like java.util.Iterator or java.util.Enumeration, you basically use this pattern. The concepts of iterators and enumerators have existed for a long

time. In simple words, enumerators can produce the next element based on a criterion whereas using iterators, you cycle a sequence from a starting point to the endpoint. In Java, you can treat an Iterator as an improved version of an Enumeration, because the JDK documentation says the following: Iterator takes the place of Enumeration in the Java Collections Framework. Iterators differ from

enumerations in two ways:

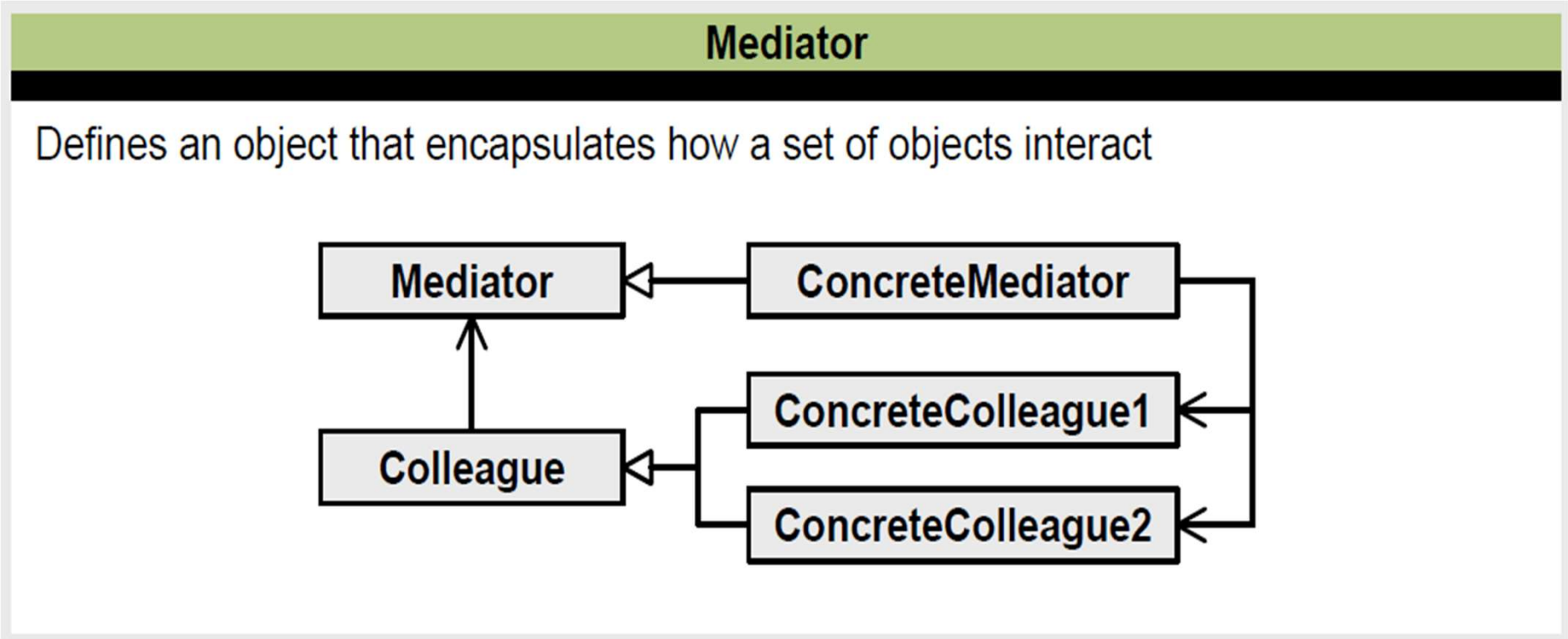
- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics. Method names have been improved.
- The java.util.Scanner class that implements Iterator<String>

also follows this pattern. It allows you to iterate over an input stream.

Mediator Pattern

- It reduces the communication complexity between multiple objects.
- The pattern provides a mediator object which handles all the communication difficulties between the objects.
- The mediator is the communication center for the objects.
- When an object needs to communicate with other objects, it doesn't call the other objects directly; instead, it calls the mediator object which handles those hurdles.
- Reduce the coupling between the objects.
- A mediator is an intermediary through whom different objects talk to each other. It takes responsibility for controlling and coordinating the interactions of a specific group of objects that cannot refer to each other explicitly. As a result, you can reduce the number of direct interconnections and promote loose coupling among them in your application.

Class Diagram



Real-time Example

- Let's consider Air Traffic Control (ATC) which is used to control the air traffic. The primary purpose of ATC is to prevent collisions between planes during landings and takeoffs. Let's suppose three planes are trying land on a particular pathway. If there is no ATC, those will collide each other. ATC communicates with all the planes that are trying to land on that pathway. ATC decides which plane needs to land first and then the next. By this way, it is overcoming the big accident.

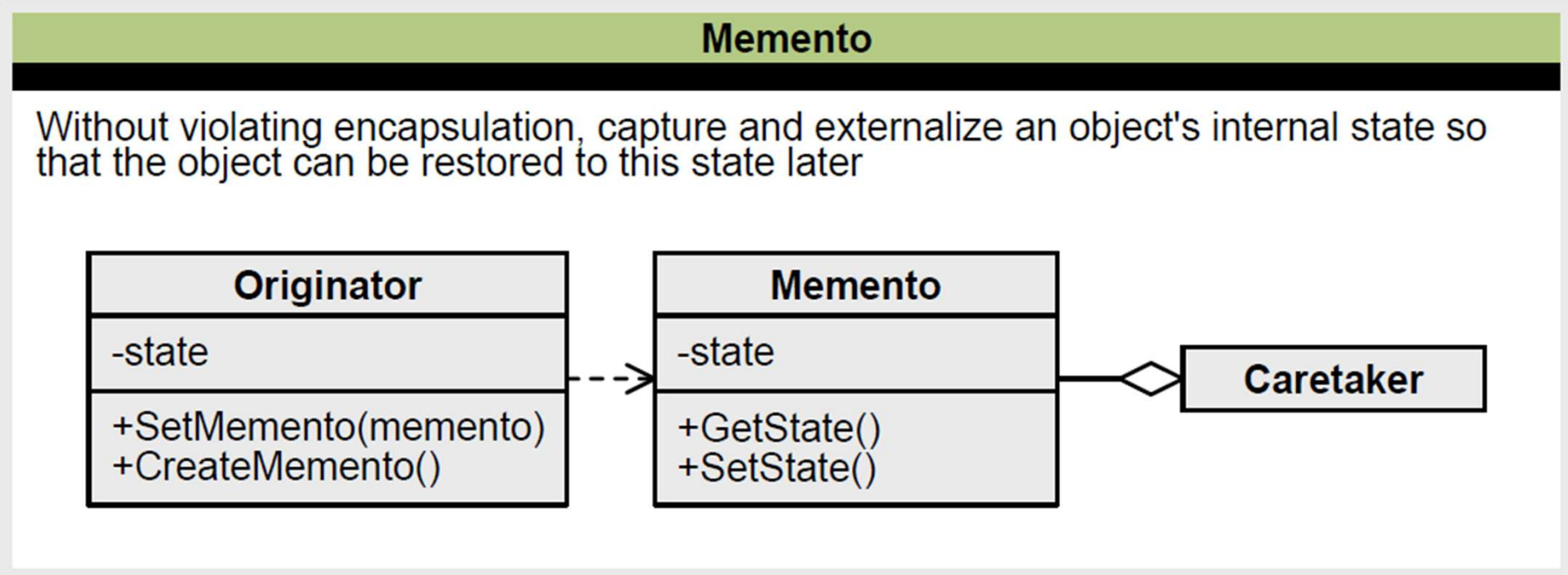
Real-time Example in Application

- When a program consists of many classes and the logic is distributed among them, the code becomes harder to read and maintain. In these scenarios, if you want to bring changes into the system's behavior, it can be difficult unless you use the Mediator pattern. Here are a few more examples:
- The execute() method inside the java.util.concurrent.Executor interface follows this pattern.
- The javax.swing.ButtonGroup class is another example that supports this pattern. This class has a method named setSelected() which ensures that a user can provide a new selection.
- The different overloaded versions of various schedule() methods of
- the java.util.Timer class also can be considered as following this pattern.

Memento Pattern

- It lets you save and restore the previous state of an object without revealing the details of its implementation.
- It restores an object to its previous state.
- Example: “undo” or “rollback”.
- Basic object that is stored in different states.
- Caretaker: Holds an ArrayList that contains all previous versions of the Memento. It can store and retrieve stored Mementos.
- The caretaker is not allowed to make any changes to a memento. It simply passes the memento to an originator. From its standpoint, it sees a narrow interface to the memento. On the contrary, the originator sees a wide interface because it has full access to the memento. The originator uses this facility to restore its previous state.

Class Diagram



Real-time Example

- Let's consider a bank example. If your bank debits some amount mistakenly, you would call your bank and conveys your query. The bank checks your transaction and if the operation was done by mistake they bank would return you the debited amount. Here, the bank is rollbacking your transaction by changing the state to the previous state.

Real-time Example in Application

- Consider a Java Swing example. You may see the use of the JTextField class, which extends the abstract class JTextComponent and provides undo support mechanism. The text component does not itself provide the history buffer by default but does provide the UndoableEdit records that can be used in conjunction with a history buffer to provide the undo/redo support. The support is provided by the Document model, which allows one to attach UndoableEditListener implementations."
- In this case an implementation of javax.swing.undo.UndoableEdit acts like a memento and an implementation of javax.swing.text.Document acts like an originator and javax.swing.undo.UndoManager acts as a caretaker.

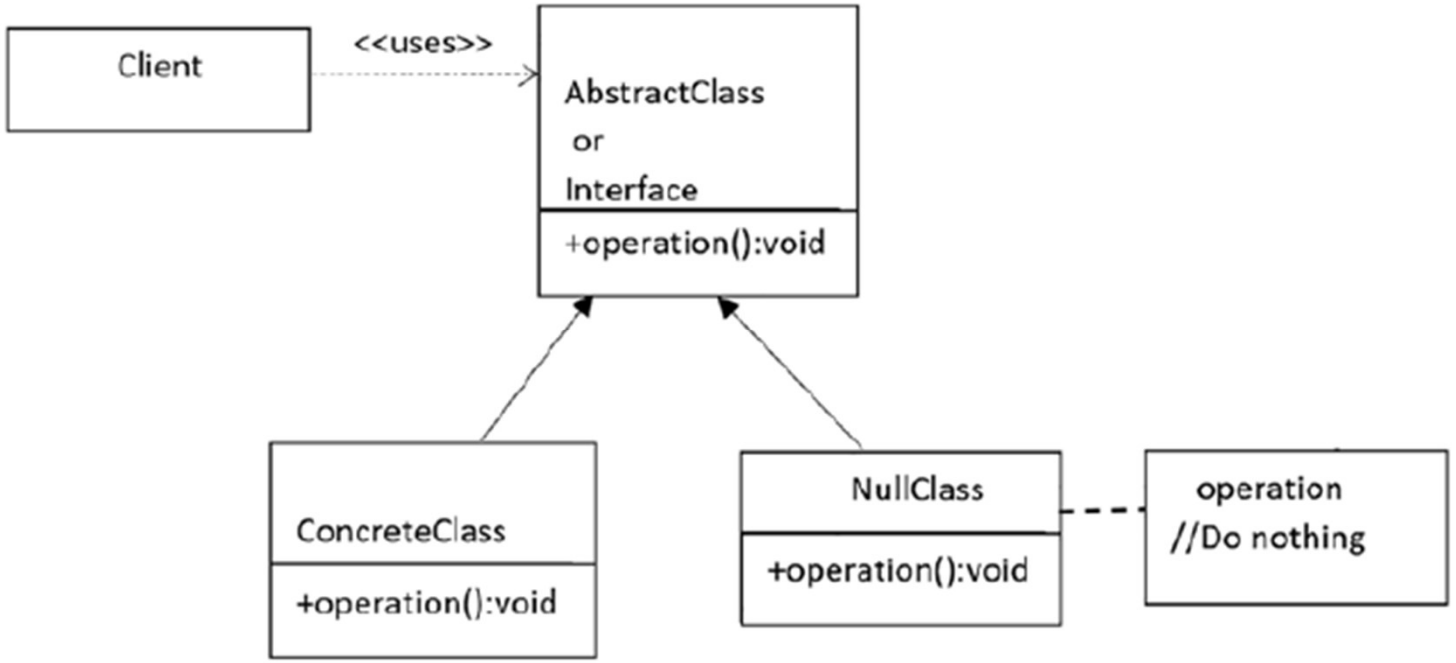
Null Object Design Pattern

- It replaces the “null” check.
- Traditional method: null check for every value which has some limitations in code flow. If we follow the “Null object design pattern” then we can get over that hurdle.
- The default behavior for objects which has “null” or “nothing.
- The null object class is the default implementation.
- It would be handy if we want some special flow for something which has nothing or a null value.
- The pattern can implement a “do-nothing” relationship, or it can provide a default behavior when an application encounters a null object instead of a real object. Using this pattern, our core aim is to make a better solution by avoiding a “null objects check” or a “null collaborations check” through if blocks.

Null Object Design Pattern

- When to use this pattern?
  - ✓ You do not want to encounter a NullPointerException (for example, if by mistake you try to invoke a method from a null object).
  - ✓You like to ignore lots of null checks in your code.
  - ✓You want to make your code cleaner and easily maintainable.

Class Diagram



Real-time Example

- A washing machine can wash properly if there is a smooth water supply without any internal leakage. But suppose, on one occasion, you forget to supply the water before you start washing the clothes, but you pressed the button that initiates washing the clothes. The washing machine should not damage itself in such a situation so it may beep an alarm to get your attention and indicate that there is no water supply at the moment.

Real-time Example in Application

- In Java, you may have seen the use of various adapter classes in the java.awt.event package. These classes can be considered as closer examples to the Null Object pattern. For example, consider the MouseMotionAdapter class. It is an abstract class but it contains methods with empty bodies like mouseDragged(MouseEvent e){} and mouseMoved(MouseEvent e){}. However, since the adapter class is tagged with the abstract keyword, you cannot directly create objects of the class.
- Basically, in an enterprise application, you can avoid a big number of **null checks and if/else blocks** using this design pattern. The following implementation will give you a quick overview of this pattern.

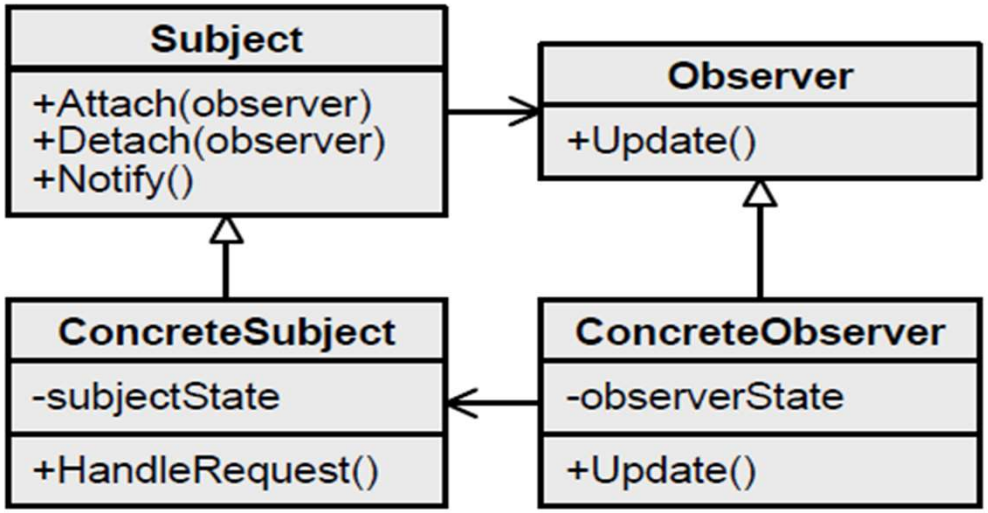
Observer Pattern

- Also known as -- Listener Design Pattern.
- The main aim of Observer Design Patterns is one to many dependencies between objects. One object changes state, and all its dependents are notified and updated automatically.
- In this pattern, there are **two types of objects**. One is an observer and the other is subject.
- What is an **observer**? In simple words, it is an object that needs to be informed when interesting stuff happens in another object.
- The object about whom an observer is interested is called the **subject**.
- Object (Subject) maintains all its dependents called observers.
- Subject (Publisher) doesn't need to know anything about the observers (subscribers or listeners).
- An object (subject) can send notifications to multiple observers (a set of objects) at the same time. Observers can decide how to respond/react to these notifications.
- Very loose coupling
- Limitation: The subject (Publisher) may send updates that may don't make sense to the observers (listeners).

Class Diagram

Observer

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



Real-time Example

- Let's consider an online shopping website. If you are willing to buy an electrical gadget which is in huge demand, you have to wait till the product is available. Since you are very eager to buy this product so you will register it on that site. If that product is available, the site will notify you either by mail or message. Here, you are the observer. Similarly, many observers are eager to buy that product. Here, observers won't have information of other observers, but the online shopping site (Subject) knows information about all the observers. The subject has information like Phone number, email id of all the observers. Whenever the product is available, the subject sends the notification to all the observers.

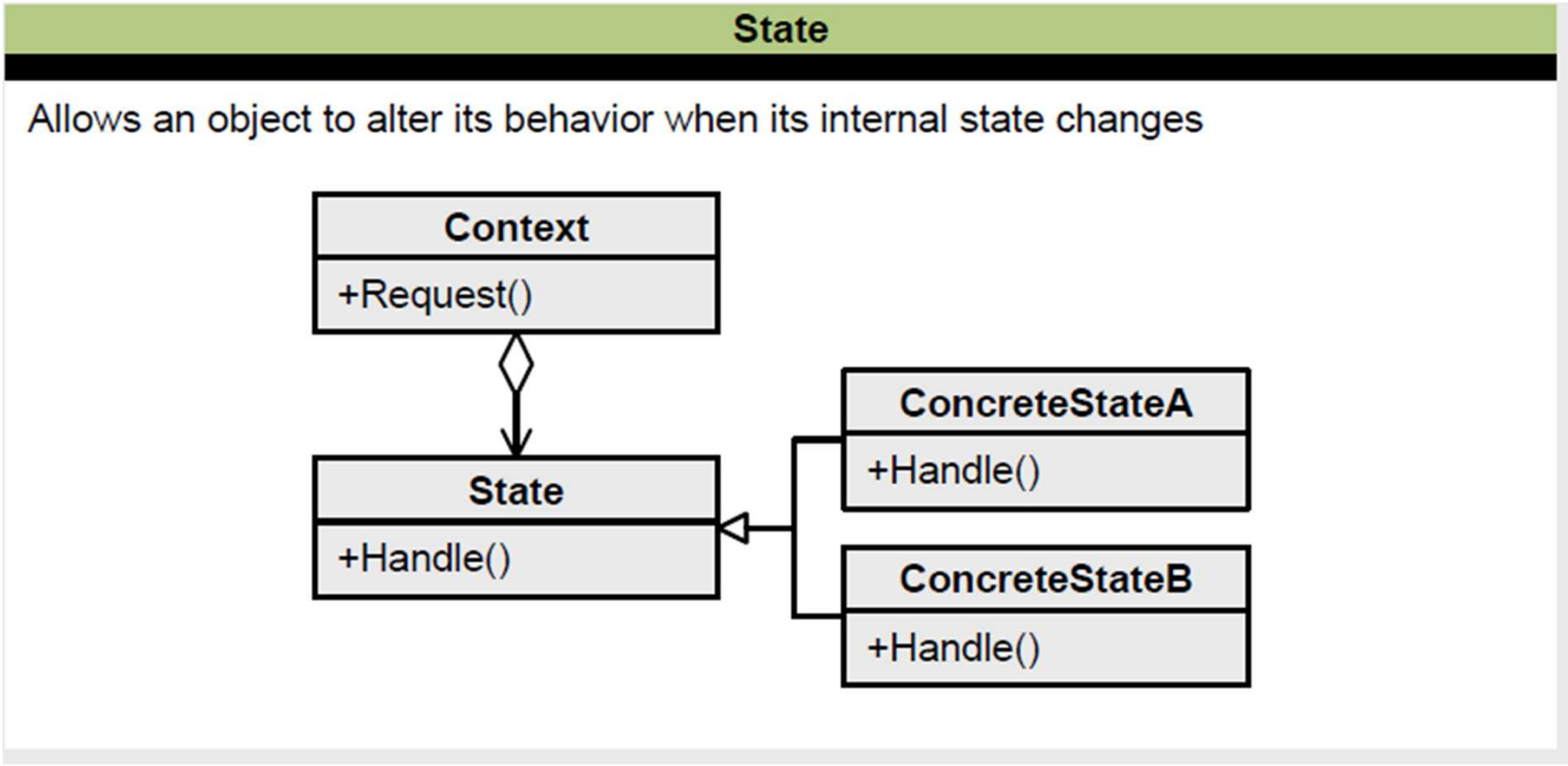
Real-time Example in Application

- In Java programming, you see the use of event listeners. These listeners are nothing but observers only. Before Java 9, there were readymade constructs such as the Observable class and the Observer interface. Observable can have multiple observers who need to implement the Observer interface. Java 9 onwards they are deprecated
- For reliable and ordered messaging among threads, consider using one of the concurrent data structures in the java.util.concurrent package. For reactive streams style programming, see the java.util.concurrent.Flow API.

State Pattern

- It lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.
- The State pattern suggests that you create new classes for all possible states of an object and extract all state-specific behaviors into these classes.
- To transition the context into another state, replace the active state object with another object that represents that new state. This is possible only if all state classes follow the same interface and the context itself works with these objects through that interface.

Class Diagram



Real-time Example

- The buttons and switches in your smartphone behave differently depending on the current state of the device:

✓When the phone is unlocked, pressing buttons lead to executing various functions.

✓When the phone is locked, pressing any button leads to the unlock screen.

✓When the phone's charge is low, pressing any button shows the charging screen.

- The functionalities of a traffic signal or television can be considered examples of the State pattern.
- You can change the channel if the TV is already in the switched-on mode. It will not respond to the channel change request if it is in the switched-off mode.

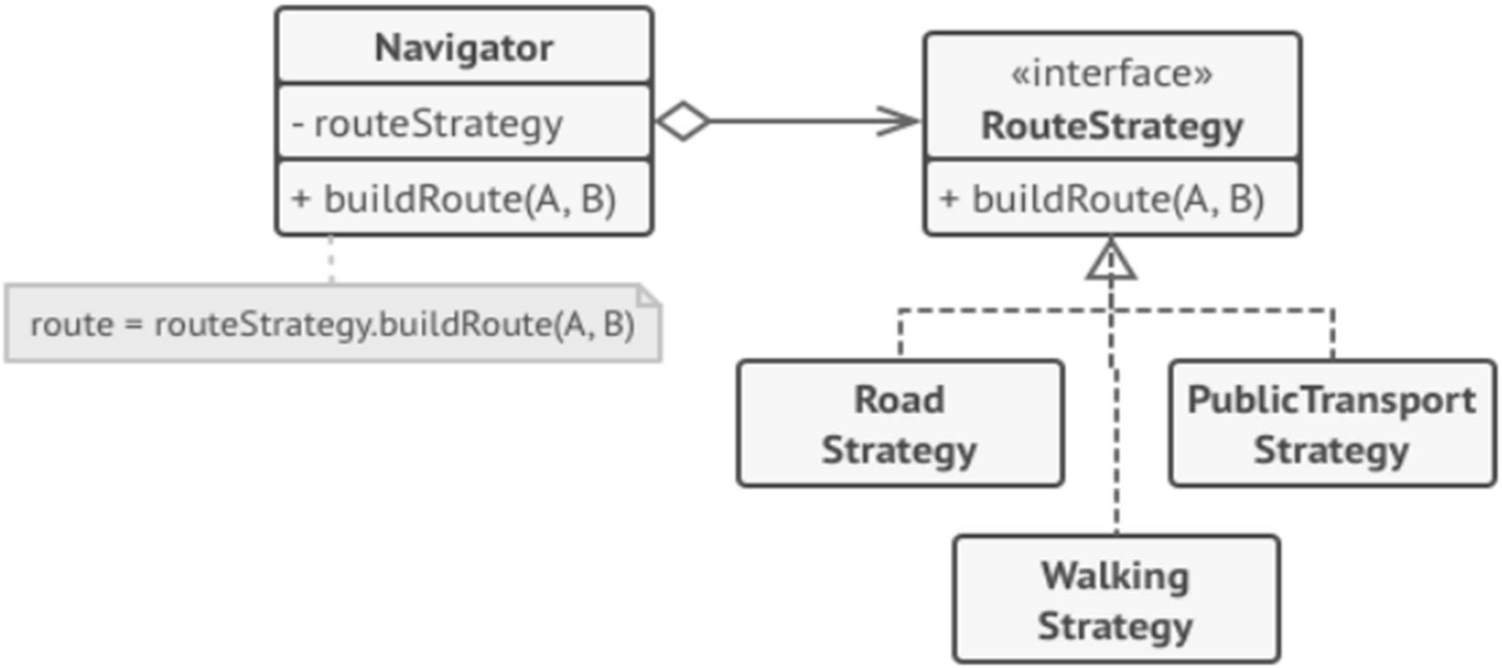
Real-time Example in Application

- Consider the scenario of a network connection, say a TCP connection. An object can be in various states. For example, a connection might already be established, a connection might be closed, or the object has already started listening through the connection. When this connection receives a request from other objects, it responds according to its present state.

Strategy Pattern

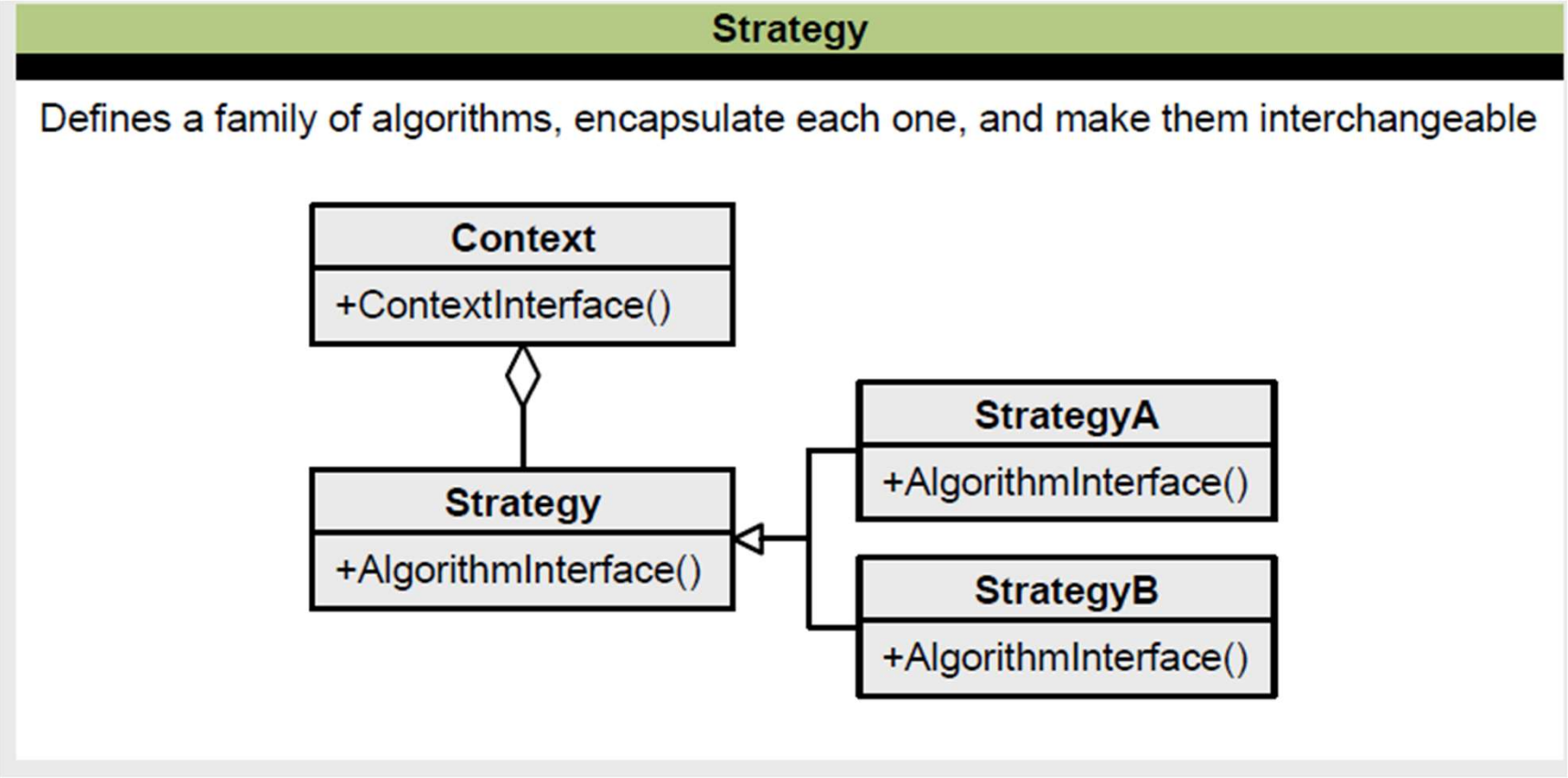
- It lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
- The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called *strategies*.
- **Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime**
- **Use the pattern to isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic.**
- **Use the pattern when your class has a massive conditional statement that switches between different variants of the same algorithm.**

Strategy Pattern



Route planning strategies.

Class Diagram



Real-time Example

- Consider a common scenario in a soccer match. Suppose team A has a 1–0 lead over team B toward the end of the game. In a situation like this, instead of attacking, team A becomes defensive in order to maintain the lead. At the same time, team B goes for an all-out attack to score the equalizer.

Real-time Example in Application

- You can consider the interface java. util.Comparator in this context. You can implement this interface and provide different algorithms to do various comparisons using the compare() method. This comparison result can be further used in various sorting techniques. The Comparator interface plays the role of the strategy interface in this context.

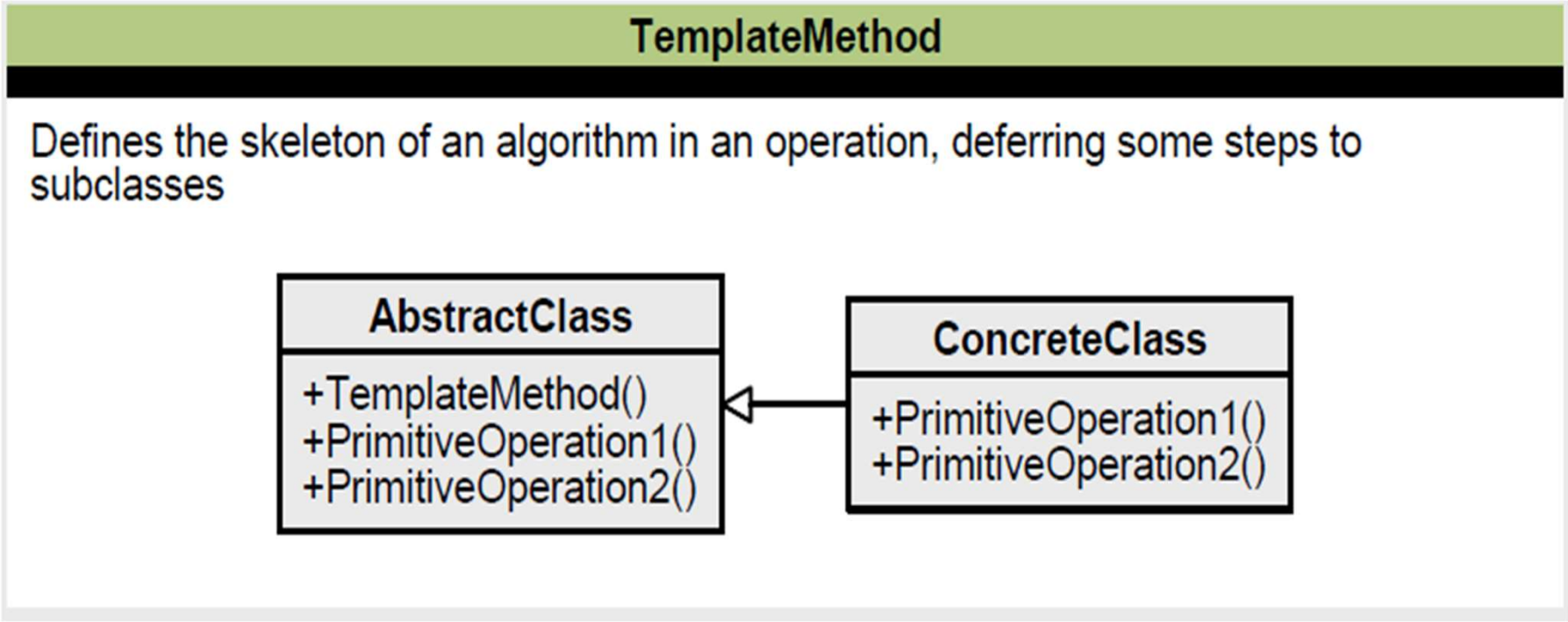
Template Method Design Pattern

- It defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
- It defines a sequence of steps of an algorithm.
- Subclasses can override the steps but it won’t allow to alter the sequence of steps.
- House building: basement, pillars, side walls, interior – concrete and wooden house
- Software development: analysis, design, develop and test.

Template Method Design Pattern

- Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure.
- Use the pattern when you have several classes that contain almost identical algorithms with some minor differences. As a result, you might need to modify all classes when the algorithm changes.

Class Diagram



Real-time Example

- When you order a pizza, the chef of the restaurant can use a basic mechanism to prepare the pizza, but they may allow you to select the final materials. For example, a customer can opt for different toppings such as bacon, onions, extra cheese, or mushrooms. So, just before the delivery of the pizza, the chef can include these choices.
- Let's suppose if you want to construct a house, and you need to follow several steps. You need to set up the basement; you need to construct the pillars, and you need to construct side walls and other steps. The house can be constructed using cement or wood, but you need to follow the above steps. So here the steps are common to construct any house. It can be a Concrete house, Wooden house, etc

Real-time Example in Application

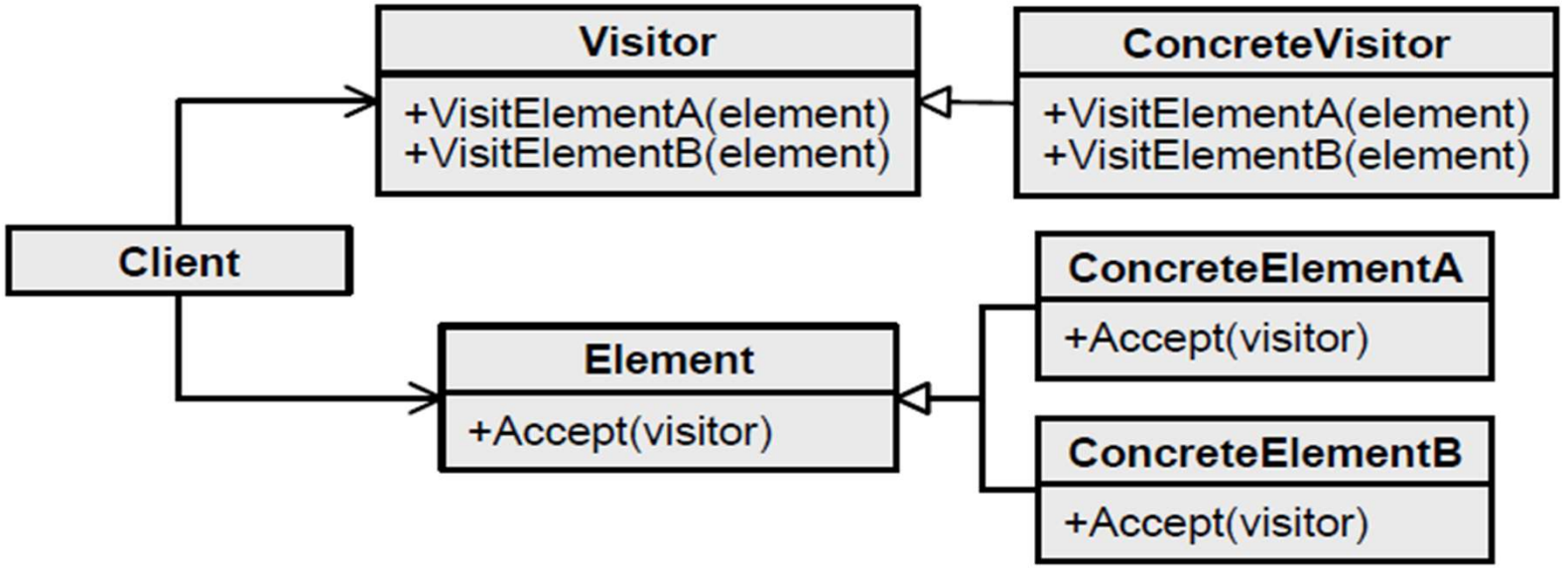
- The removeAll() method of java.util.AbstractSet.
- The size() is present in the parent class AbstractCollection

Visitor Design Pattern

- It represents an operation to be performed on the elements of an object structure. The Visitor pattern lets you define a new operation without changing the classes of the elements on which it operates.
- It lets you separate algorithms from the objects on which they operate.
- This pattern can be used in many different scenarios. Using this pattern, you often add new operations to an object without modifying its class definition. To achieve this, you separate an algorithm from the object structure. This may sound surprising to you, but it is one of the key usages of this pattern. This pattern also promotes the Open/Closed Principle.

Class Diagram -Visitor

Represents an operation to be performed on the elements of an object structure



Real-time Example

- Think of a taxi-booking scenario. When the taxi arrives at your door and you enter the vehicle, the taxi takes control of the transportation. It can take you to your destination through a new route that you are not familiar with, and in the worst case, it can alter the destination (which is a case generated due to improper use of the Visitor pattern).

Real-time Example in Application

- This pattern is useful when public APIs need to support *plug-in* operations. Clients can then perform their intended operations on a class (with the visiting class) without modifying the source. Here is another example for you:
- When you use the interface javax.lang.model.element. ElementVisitor<R, P> (where R is the return type of the visitor's method and P is the type of the additional parameter to the visitor's method), you follow this pattern

Rules of thumb

1. Behavioral patterns are concerned with the assignment of responsibilities between objects, or, encapsulating behavior in an object and delegating requests to it.
2. **Chain of responsibility**, **Command**, **Mediator**, and **Observer**, address how you can decouple senders and receivers, but with different trade-offs. **Chain of responsibility** passes a sender request along a chain of potential receivers. **Command** normally specifies a sender-receiver connection with a subclass. **Mediator** has senders and receivers reference each other indirectly. **Observer** defines a very decoupled interface that allows for multiple receivers to be configured at run-time.
3. **Chain of responsibility** can use **Command** to represent requests as objects.
4. **Chain of responsibility** is often applied in conjunction with **Composite**. There, a component's parent can act as its successor.
5. **Command** and **Memento** act as magic tokens to be passed around and invoked at a later time. In **Command**, the token represents a request; in **Memento**, it represents the internal state of an object at a particular time. Polymorphism is important to **Command**, but not to **Memento** because its interface is so narrow that a memento can only be passed as a value.

Rules of thumb

6. Command can use **Memento** to maintain the state required for an undo operation.

7. Macro**Command**s can be implemented with **Composite**.

8. A **Command** that must be copied before being placed on a history list acts as a **Prototype**.

9. Interpreter can use **State** to define parsing contexts.

10. The abstract syntax tree of **Interpreter** is a **Composite** (therefore **Iterator** and **Visitor** are also applicable).

11. Terminal symbols within **Interpreter**'s abstract syntax tree can be shared with **Flyweight**.

12. Iterator can traverse a **Composite**. **Visitor** can apply an operation over a **Composite**.

Rules of thumb

13. Polymorphic **Iterator**s rely on **Factory Method**s to instantiate the appropriate **Iterator** subclass.

14. **Mediator** and **Observer** are competing patterns. The difference between them is that **Observer** distributes communication by introducing "observer" and "subject" objects, whereas a **Mediator** object encapsulates the communication between other objects. We've found it easier to make reusable **Observer**s and Subjects than to make reusable **Mediator**s.

15. On the other hand, **Mediator** can leverage **Observer** for dynamically registering colleagues and communicating with them.

16. **Mediator** is similar to **Facade** in that it abstracts functionality of existing classes. **Mediator** abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects (i.e. it defines a multidirectional protocol). In contrast, **Facade** defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes (i.e. it defines a unidirectional protocol where it makes requests of the subsystem classes but not vice versa).

Rules of thumb

17. **Memento** is often used in conjunction with **Iterator**.
An **Iterator** can use a **Memento** to capture the state of an iteration.
The **Iterator** stores the **Memento** internally.

18. **State** is like **Strategy** except in its intent.

19. **Flyweight** explains when and how **State** objects can be shared.

20. **State** objects are often **Singleton**s.

21. **Strategy** lets you change the guts of an object. **Decorator** lets you change the skin.

22. **Strategy** is to algorithm. as **Builder** is to creation.

23. **Strategy** has 2 different implementations, the first is similar to **State**. The difference is in binding times (**Strategy** is a bind-once pattern, whereas **State** is more dynamic).

Rules of thumb

24. **Strategy** objects often make good **Flyweight**s.
25. **Strategy** is like **Template method** except in its granularity.
26. **Template method** uses inheritance to vary part of an algorithm. **Strategy** uses delegation to vary the entire algorithm.
27. The **Visitor** pattern is like a more powerful **Command** pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters.


The way to get started is to
quit talking and begin doing.
Walt Disney

Thank you

Deepikkaa.S