Chapter Five

Design Patterns

Chapter Outlines

- Introduction to Design Patterns
- Creational Patterns
  - Builder, Factory Method, Singleton
- Structural Patterns
  - Adapter, Bridge, Decorator, Façade
- Behavioral Patterns
  - Iterator, Mediator, Template

Design Pattern

- A design pattern is a tested solution to a standard programming problem
- It simply captures the end product of the design process as relationships between classes and objects
- Someone has already faced the problem you are facing and has come up with a solution that implements all kinds of good design
- We only need to know the patterns and find the pattern that fits the problem

Model-View-Controller Pattern

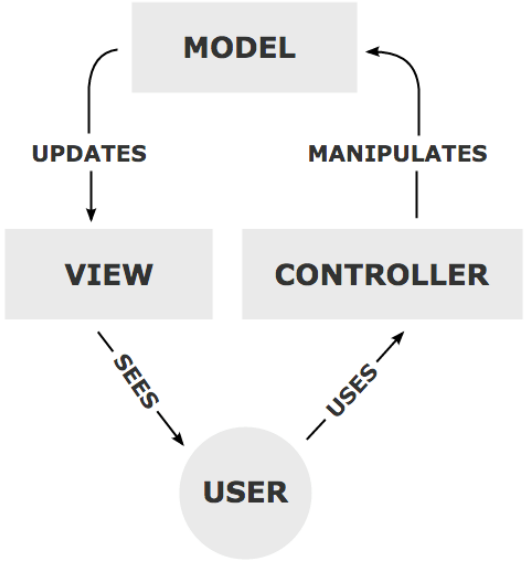
- Model-View-Controller (MVC) design pattern divides a given software application into three interconnected parts
- MVC decouples those all-in-one approaches to increase flexibility and reusability
- Adding new views and models won’t affect the others
- Controllers will link them (views and models) together for the end users

MVC Pattern

- Model
  - Stores data that is retrieved according to commands from the controller and displayed in the view
- View
  - Generates an output presentation to users on changes in the model
- Controller
  - Controller defines the way the user interface reacts to users' behaviors (requests, operations, etc.)
  - It can send commands to the model to update the model's state
  - It can send commands to its associated view to change the view's presentation of the model
  - It contains the business logic to control the behaviors of an application

The Usages of MVC

- MVC design pattern is extremely popular for designing web frameworks
  - Most of the modern software are built based on MVC design pattern
  - Many languages support MVC design pattern
- We have already adopted the MVC
  - JavaBean are the *Models*
  - JavaFX Nodes are the *Views*
  - JavaFX Application class (control flow) and other Java classes (business logics) are the *Controllers*
- MVC can contain other patterns



https://blog.csdn.net/drivermonkey/article/details/49755271

Types of Design Pattern

- There are 23 well-known design patterns (by *Gamma*, *Helm*, *Johnson*, and *Vlissides*)

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

Different Classification

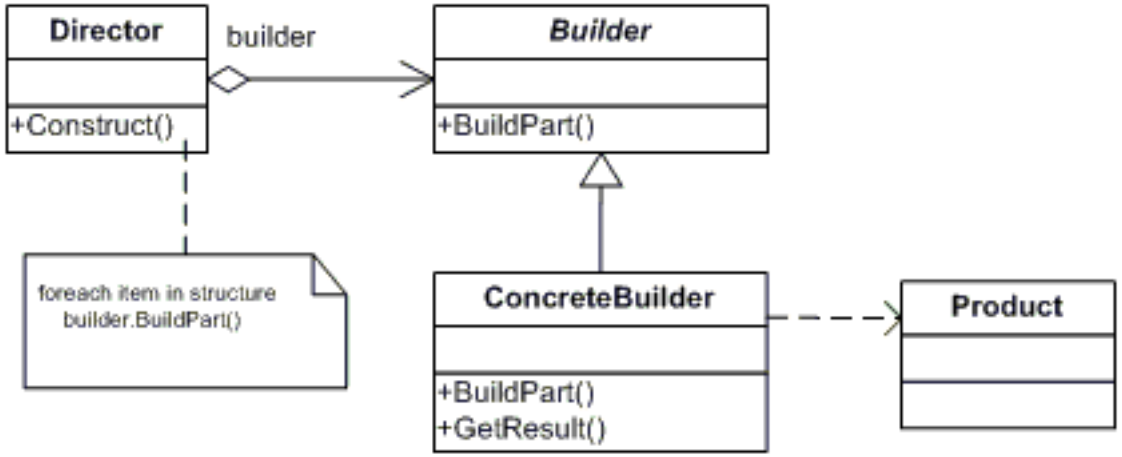
- Interface Patterns
  - Adapter, Bridge, Composite, Façade
- Responsibility Patterns
  - Chain of Responsibility, Flyweight, Mediator, Observer, Proxy, Singleton
- Construction Patterns
  - Abstract Factory, Builder, Factory Method, Memento, Prototype
- Operation Patterns
  - Command, Interpreter, State, Strategy, Template Method
- Extension Patterns
  - Decorator, Iterator, Visitor

Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

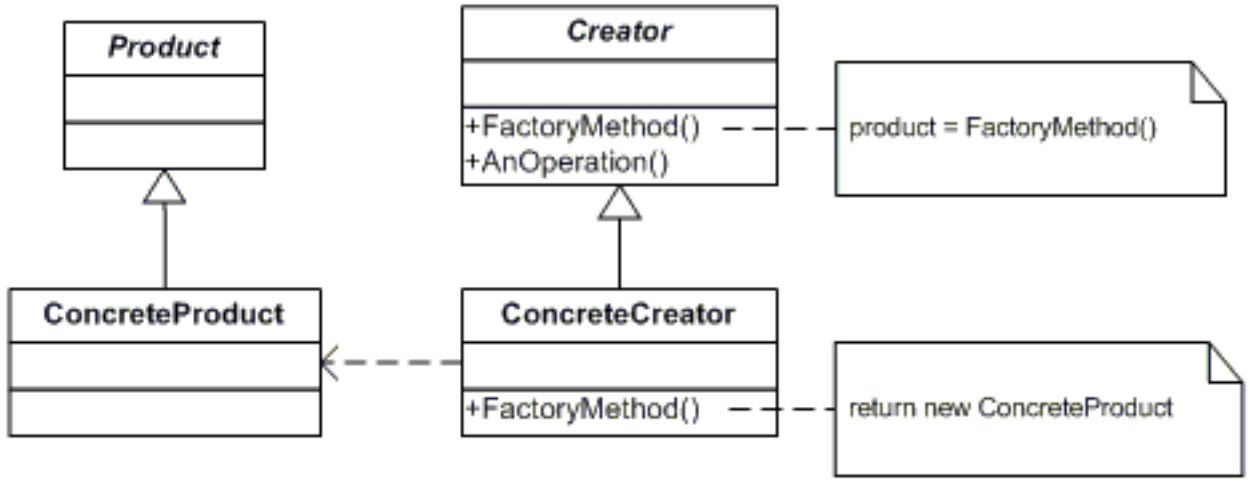
Creational Pattern: Builder

- Separates object construction from its representation
- Separate the construction of a complex object from its representation so that the same construction processes can create different representations



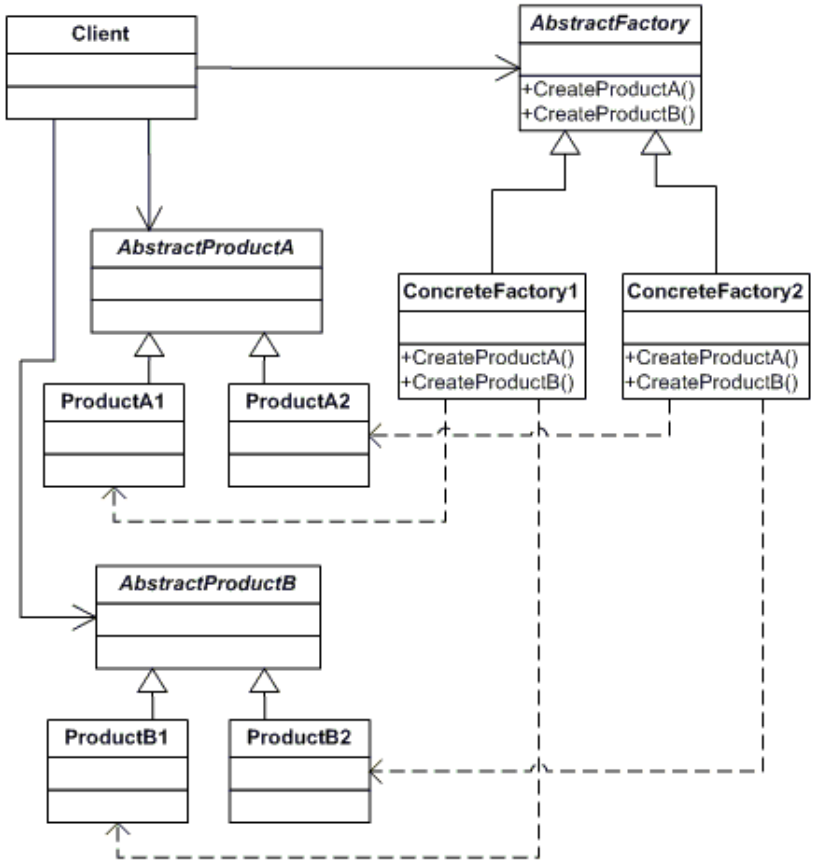
Creational Pattern: Factory Method

- Creates an instance of several derived classes
- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses



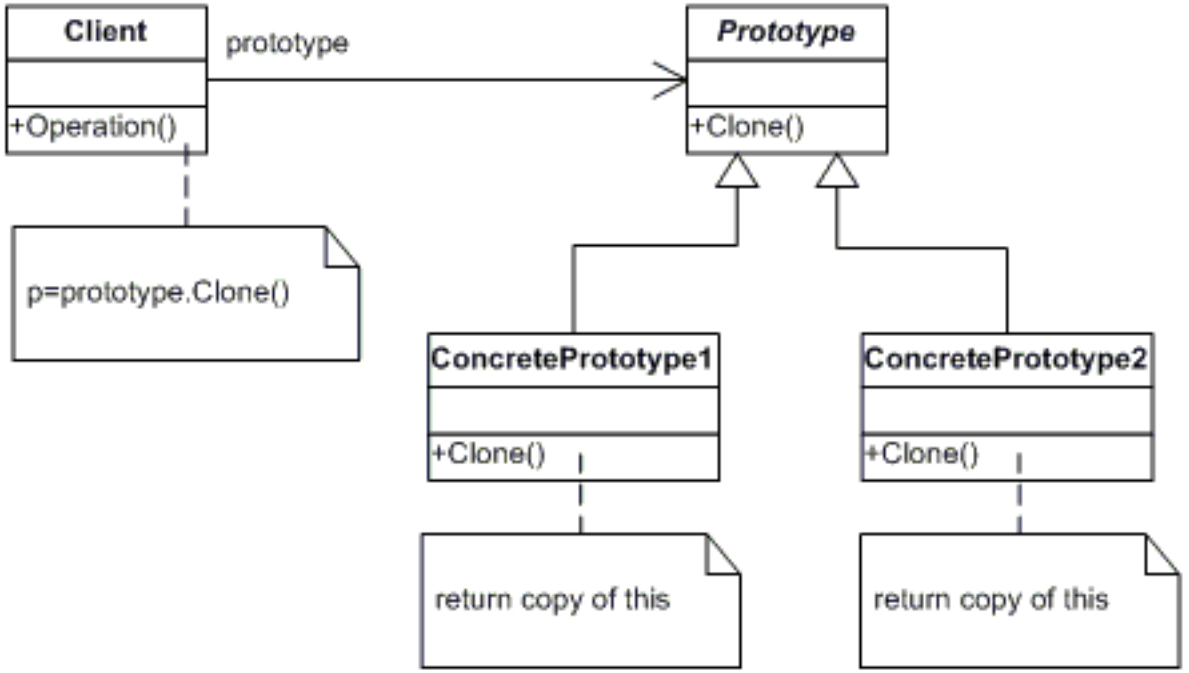
Creational Pattern: Abstract Factory

- Creates an instance of several families of classes
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes



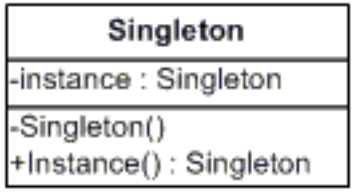
Creational Pattern: Prototype

- A fully initialized instance to be copied or cloned
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype



Creational Pattern: Singleton

- A class of which only a single instance can exist
- Ensure a class only has one instance, and provide a global point of access to it

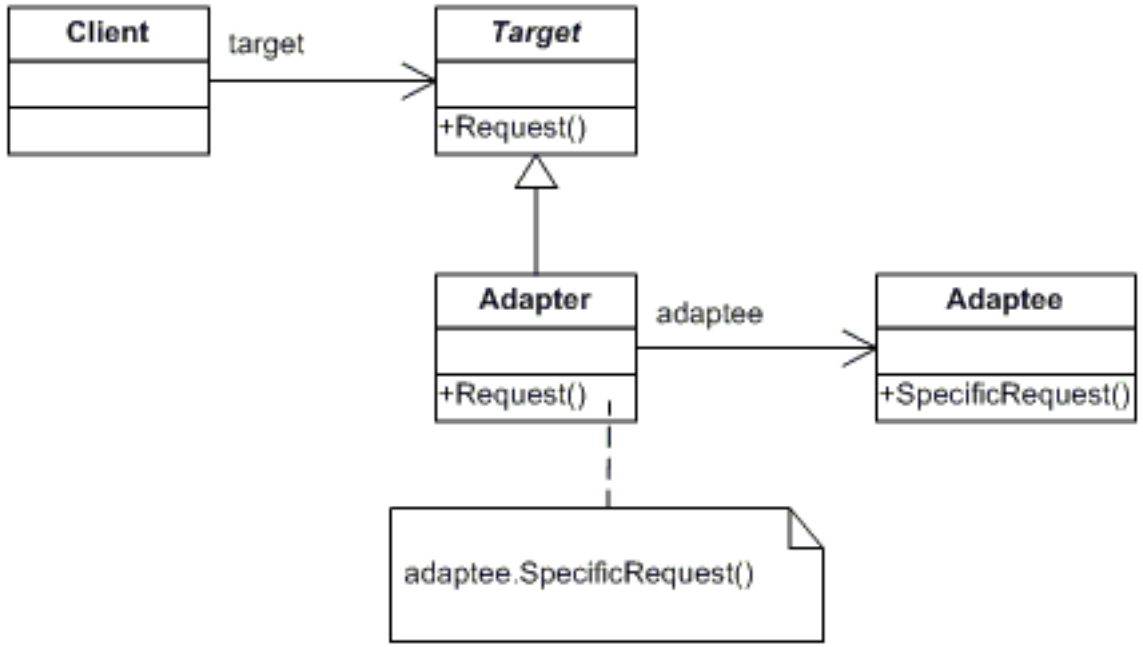


Structural Patterns

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

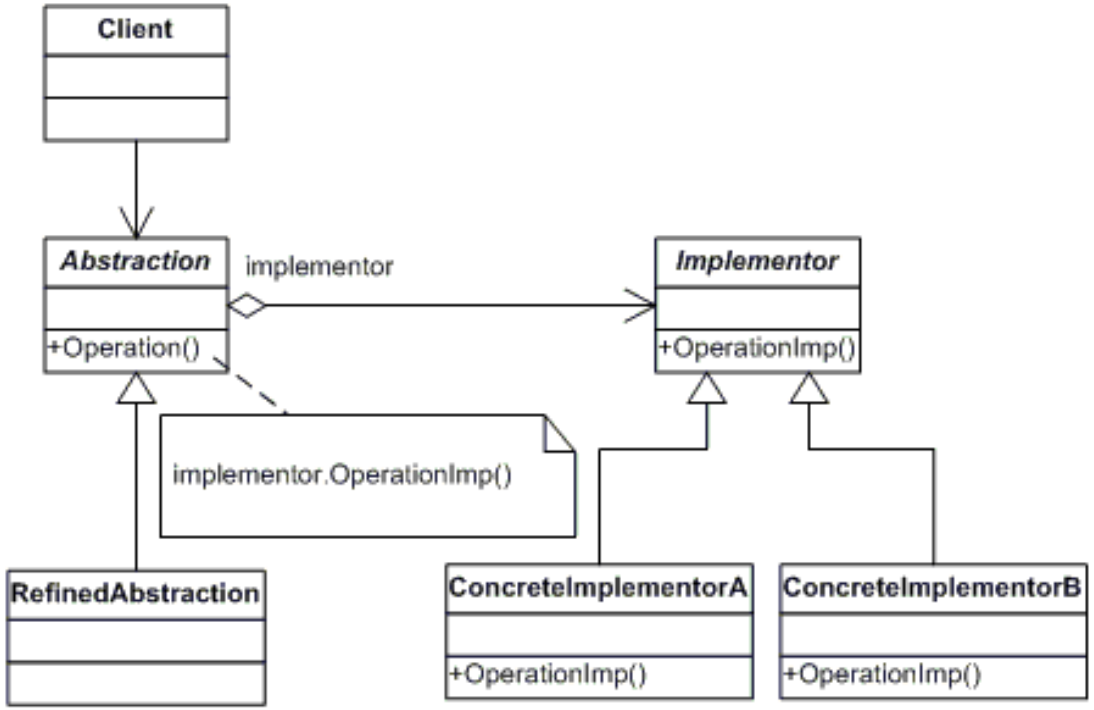
Structural Pattern: Adapter

- Match interfaces of different classes
- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces



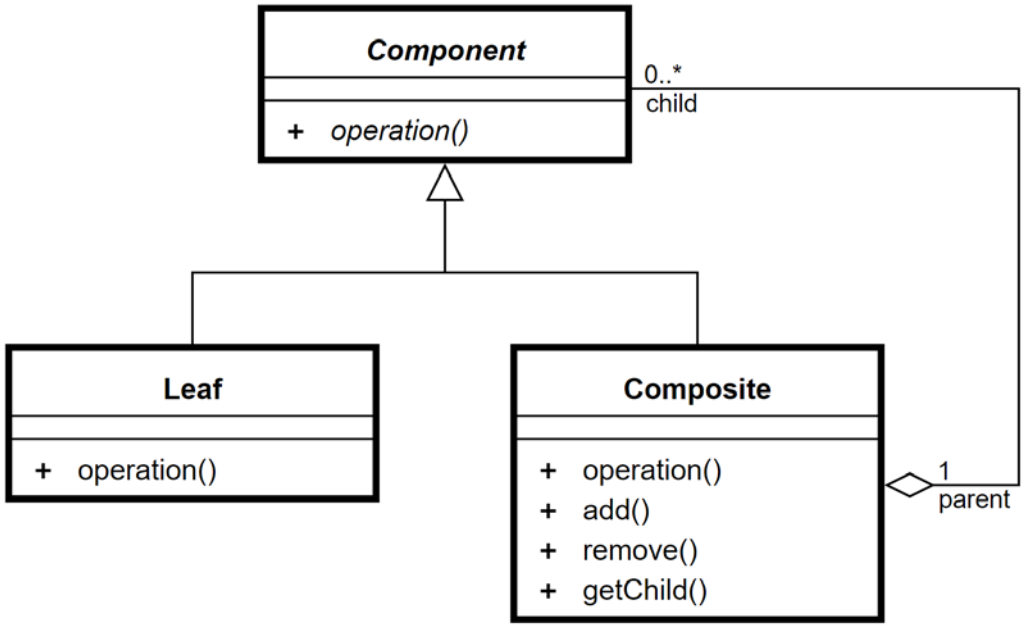
Structural Pattern: Bridge

- Separates an object's interface from its implementation
- Decouple an abstraction from its implementation so that the two can vary independently



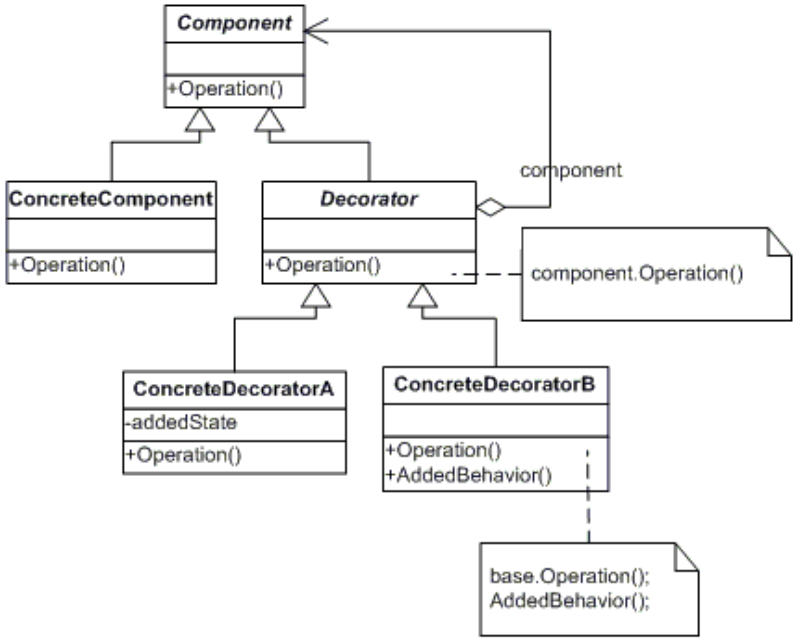
Structural Pattern: Composite

- A tree structure of simple and composite objects
- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly



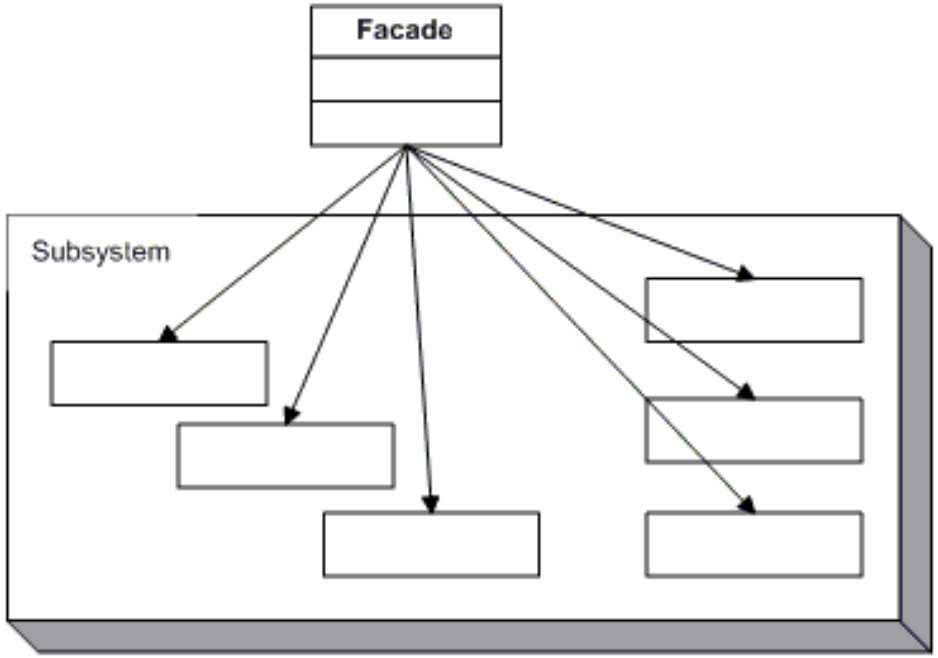
Structural Pattern: Decorator

- Add responsibilities to objects dynamically
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality



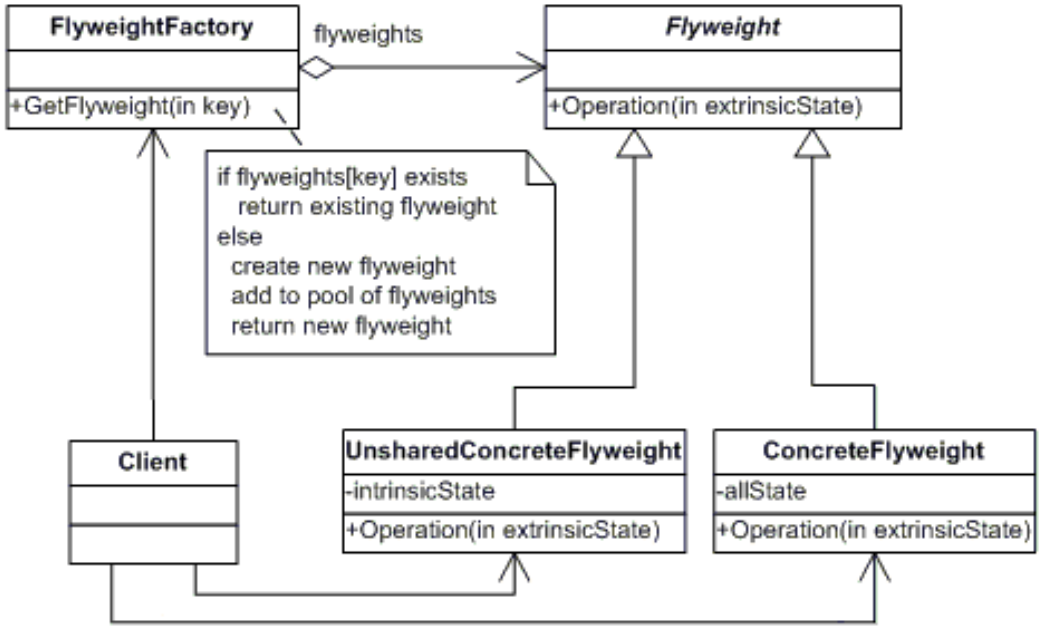
Structural Pattern: Façade

- A single class that represents an entire subsystem
- Provide a unified interface to a set of interfaces in a system. Façade defines a higher-level interface that makes the subsystem easier to use



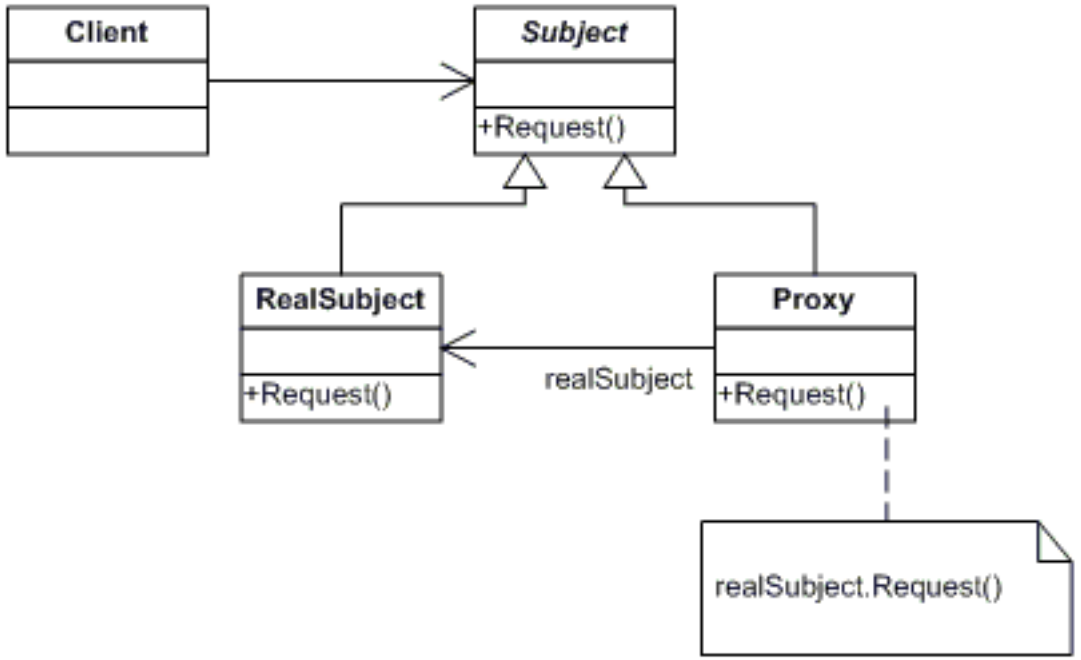
Structural Pattern: Flyweight

- A fine-grained instance used for efficient sharing
- Use sharing to support large numbers of fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously



Structural Pattern: Proxy

- An object representing another object
- Provide a surrogate or placeholder for another object to control access to it

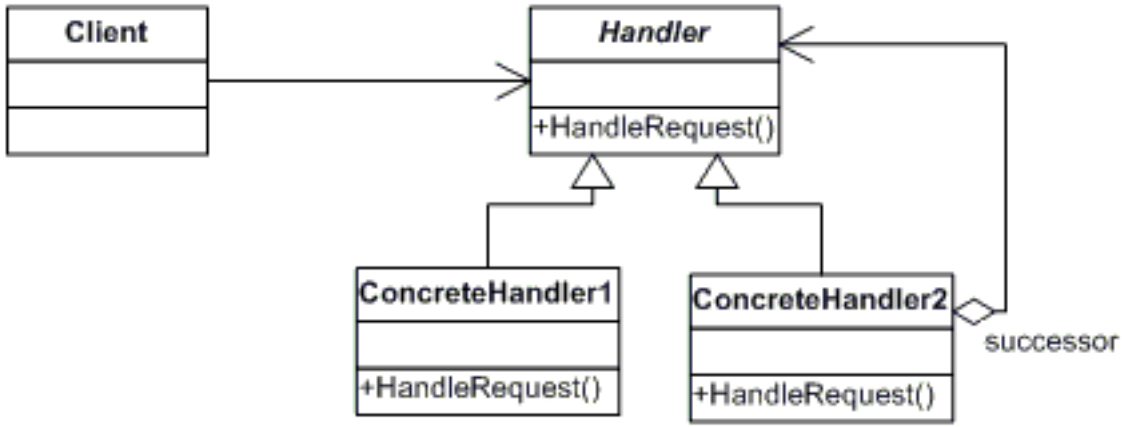


Behavioral Patterns

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

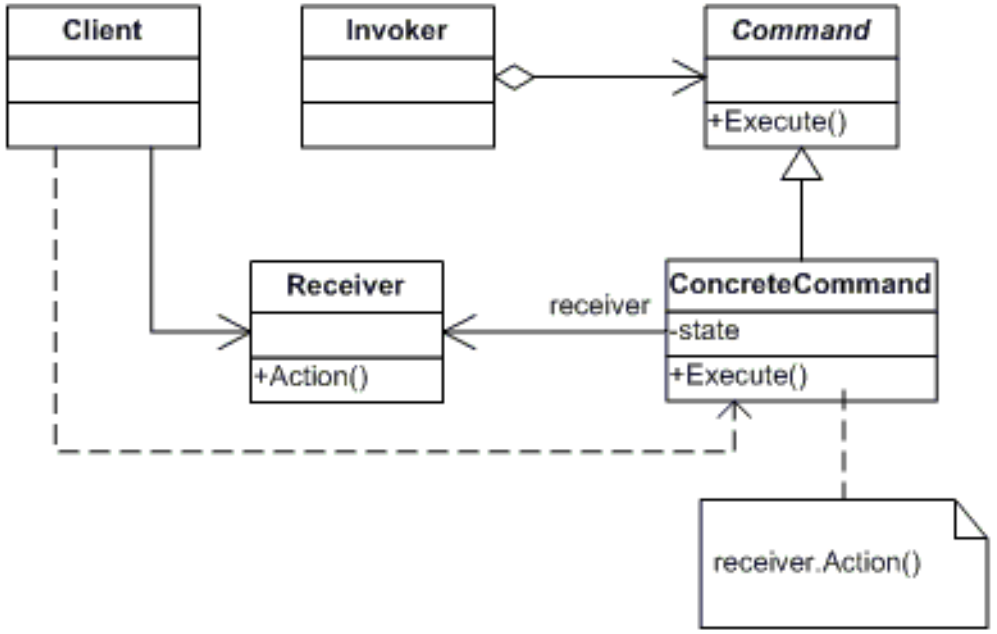
Behavioral Pattern:
Chain of Responsibility

- A way of passing a request between a chain of objects
- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it



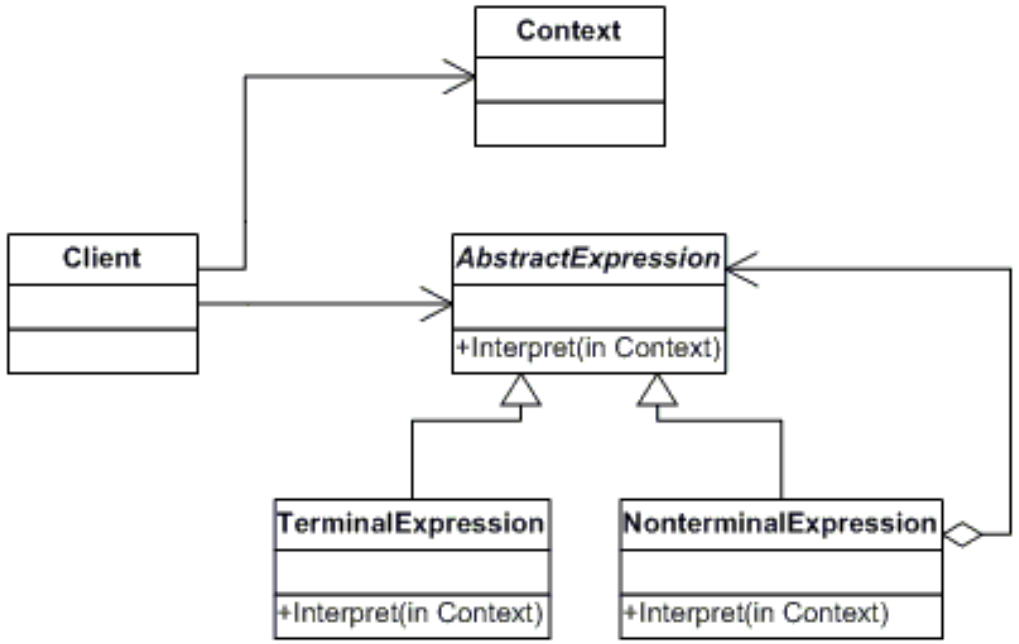
Behavioral Pattern: Command

- Encapsulate a command request as an object
- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations



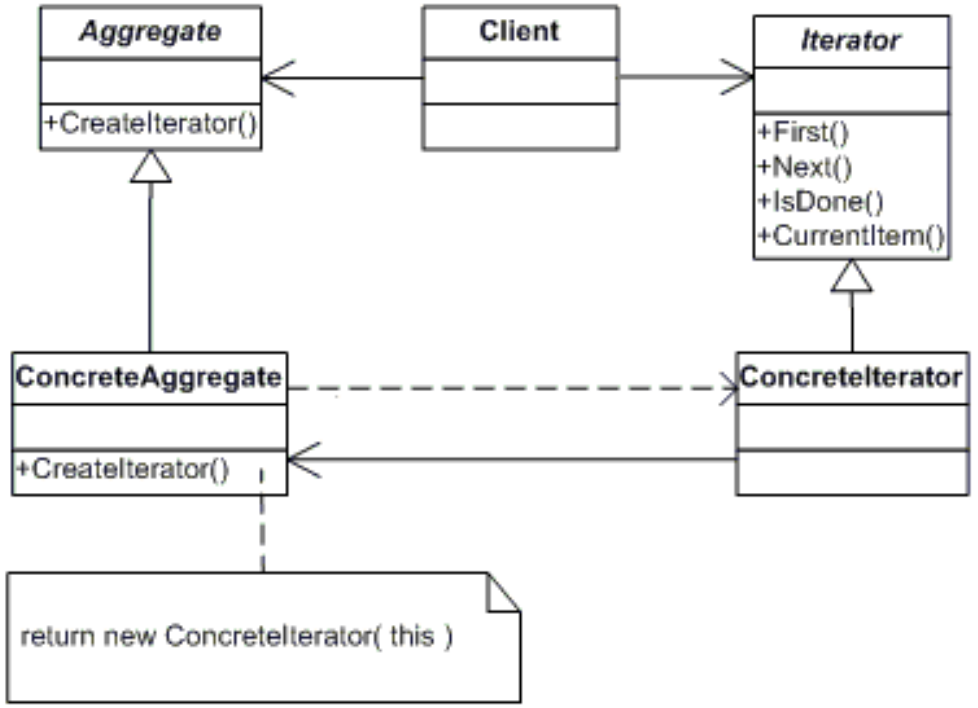
Behavioral Pattern: Interpreter

- A way to include language elements in a program
- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language



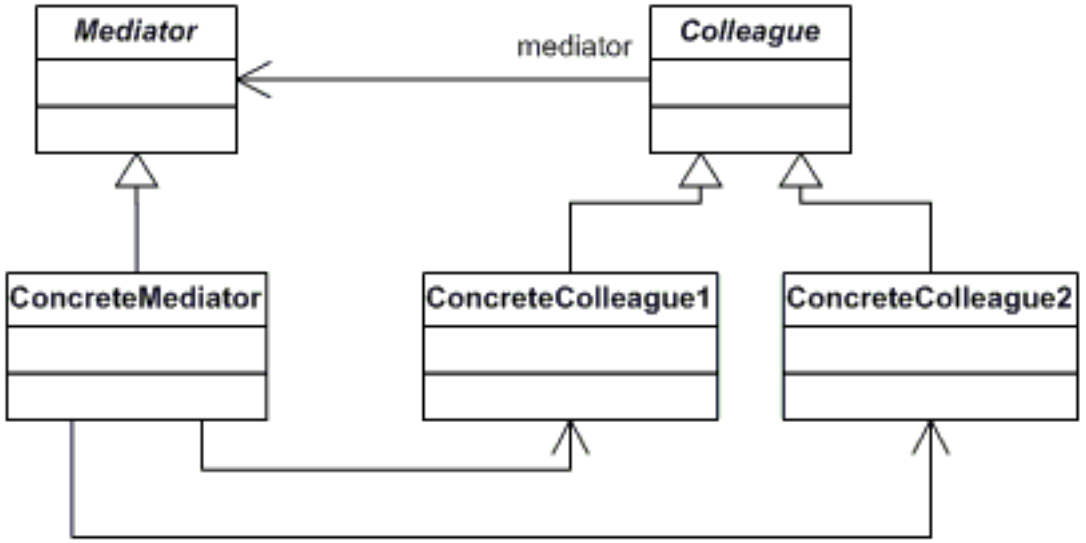
Behavioral Pattern: Iterator

- Sequentially access the elements of a collection
- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



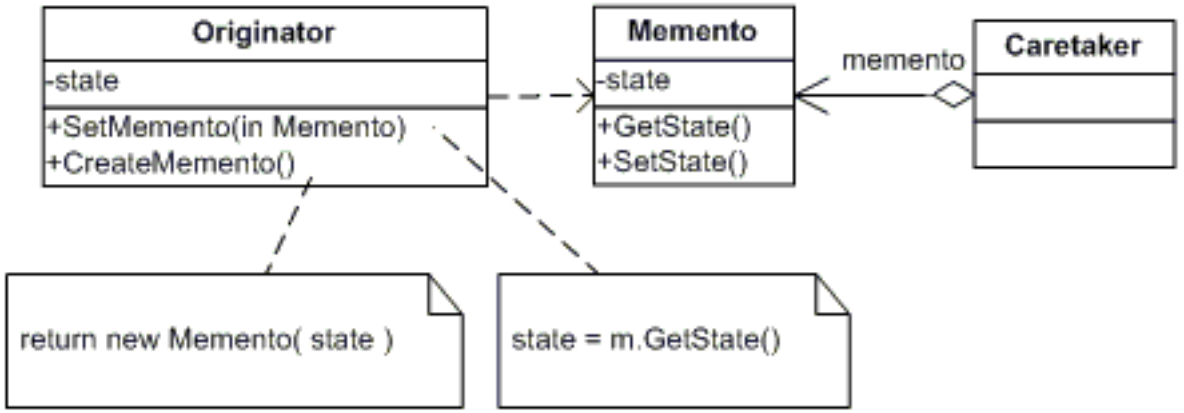
Behavioral Pattern: Mediator

- Defines simplified communication between classes
- Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently



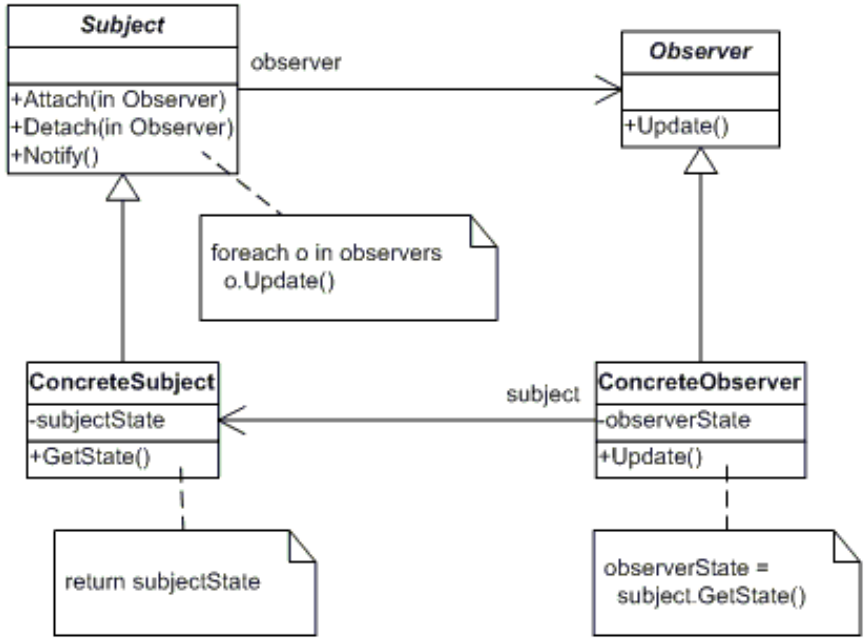
Behavioral Pattern: Memento

- Capture and restore an object's internal state
- Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later



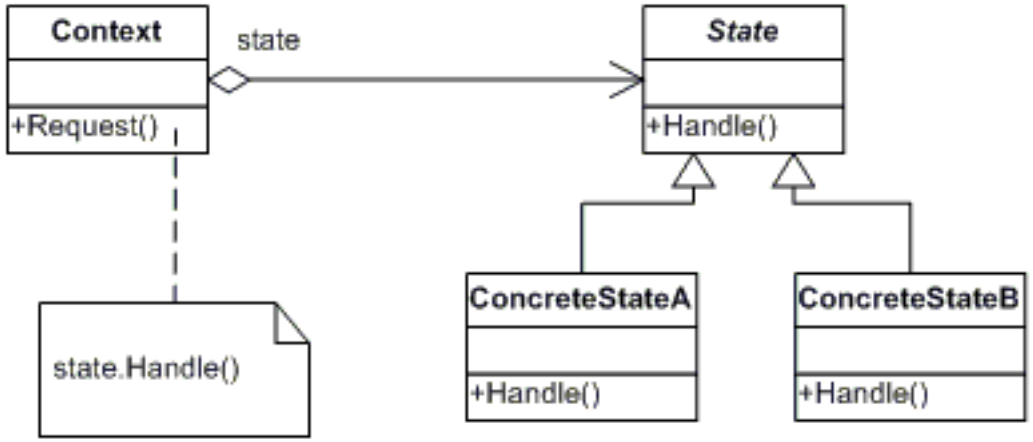
Behavioral Pattern: Observer

- A way of notifying change to a number of classes
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically



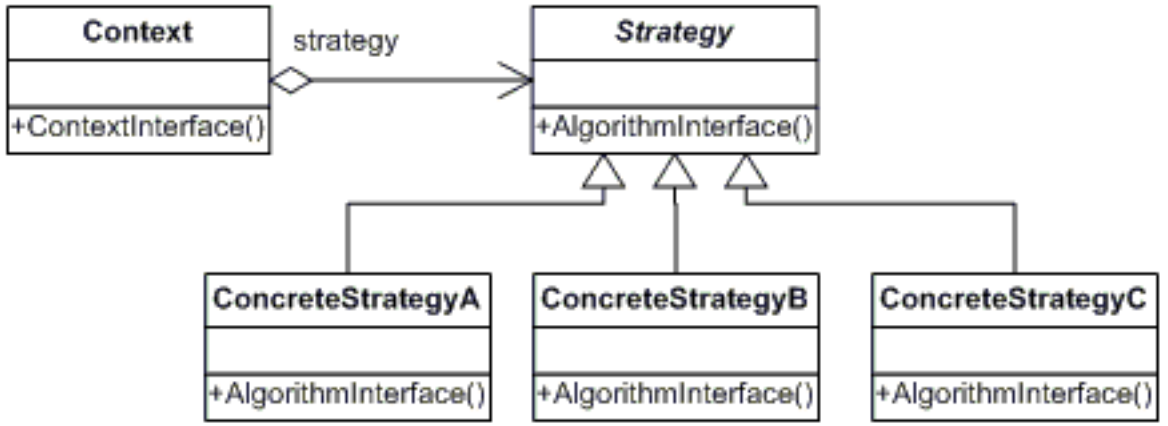
Behavioral Pattern: State

- Alter an object's behavior when its state changes
- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



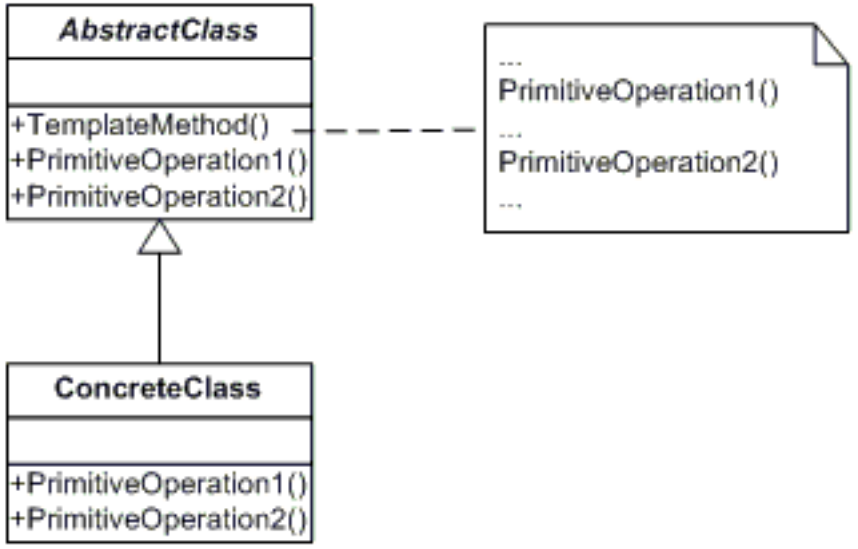
Behavioral Pattern: Strategy

- Encapsulates an algorithm inside a class
- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it



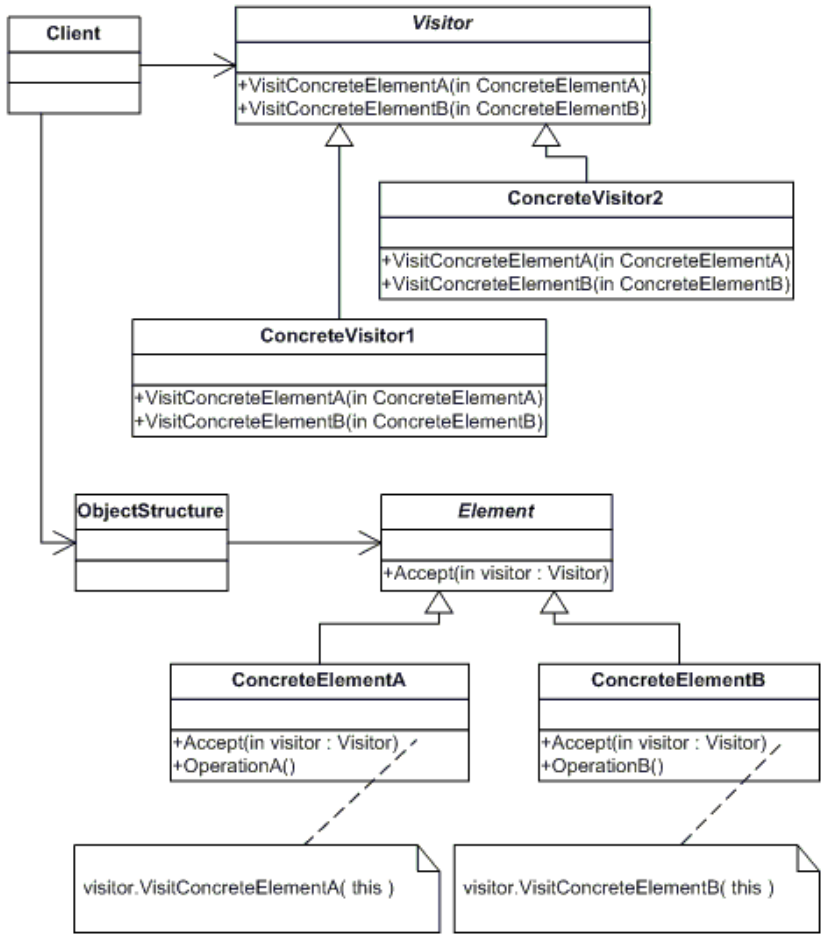
Behavioral Pattern: Template

- Defer the exact steps of an algorithm to a subclass
- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure



Behavioral Pattern: Visitor

- Defines a new operation to a class without change
- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates



Builder Pattern

- This pattern provides one of the best ways to create a complex object in a simple way
- The JavaBean pattern may be in an inconsistent state partway through its construction because it splits across multiple call
- This pattern enforces the consistency by manually freezing the object when its construction is complete
- Instead of making the desired object directly, the client calls a constructor (or static factory) with all of the required parameters and gets a builder object

Person JavaBean

```
01.  public class PersonBean {
02.    private String firstName, lastName, gender;
03.    private double height;
04.    private int age;
05.
06.    public String getFirstName() {return firstName;}
07.    public void setFirstName(String firstName) {this.firstName = firstName;}
08.    public String getLastName() {return lastName;}
09.    public void setLastName(String lastName) {this.lastName = lastName;}
10.    public String getGender() {return gender;}
11.    public void setGender(String gender) {this.gender = gender;}
12.    public double getHeight() {return height;}
13.    public void setHeight(double height) {this.height = height;}
14.    public int getAge() {return age;}
15.    public void setAge(int age) {this.age = age;}
16.  }
```

JavaBean Mutators

- Setting the values of a JavaBean could be lengthy

```
01.  …
10.      PersonBean p1 = new PersonBean();
11.      p1.setFirstName("Paul");
12.      p1.setLastName("Chan");
13.      p1.setGender("Male");
14.      p1.setHeight(180);
15.      p1.setAge(20);
16.      PersonBean p2 = new PersonBean();
17.      p2.setFirstName("Joe");
18.      p2.setLastName("Yeung");
19.      p2.setGender("Male");
20.      p2.setHeight(175);
21.      p2.setAge(21);
22.  …
```

Builder Constructor

- Isolate the construction in an inner class (Builder)

```
01.  public class PersonBean {
02.    private final String firstName, lastName, gender;
03.    private final double height;
04.    private final int age;
05.
06.    public static class Builder {
07.      // Required properties
08.      private final String firstName, lastName;
09.      // Optional properties
10.      private String gender;
11.      private double height;
12.      private int age;
13.
14.      // Pass the required properties to the Builder
15.      public Builder(String firstName, String lastName) {
16.        this.firstName = firstName;
```

Builder Constructor (cont.)

```
17.         this.lastName = lastName;
18.       }
19.
20.       public Builder setGender(String gender) {
21.         this.gender = gender;
22.         return this;
23.       }
24.
25.       public Builder setHeight(double height) {
26.         this.height = height;
27.         return this;
28.       }
29.
30.       public Builder setAge(int age) {
31.         this.age = age;
32.         return this;
33.       }
```

Builder Constructor (cont.)

```
34.
35.       public PersonBean build() {
36.         return new PersonBean(this);
37.       }
38.     }
39.
40.     private PersonBean(Builder builder) {
41.       this.firstName = builder.firstName;
42.       this.lastName = builder.lastName;
43.       this.gender = builder.gender;
44.       this.height = builder.height;
45.       this.age = builder.age;
46.     }
47.
48.     public String getFirstName() {
49.       return firstName;
50.     }
```

Builder Constructor (cont.)

```
51.
52.     public String getLastName() {
53.       return lastName;
54.     }
55.
56.     public String getGender() {
57.       return gender;
58.     }
59.
60.     public double getHeight() {
61.       return height;
62.     }
63.
64.     public int getAge() {
65.       return age;
66.     }
67.   }
```

Builder Constructor (cont.)

- This style of programming JavaBean is to ensure all properties are set only on **creation**
- Once it is created, they cannot be changed anymore
- Some properties must be initialized on creation
  - Unique name to classify the objects
  - Primary key column of a table
- Other properties are set optionally on creation

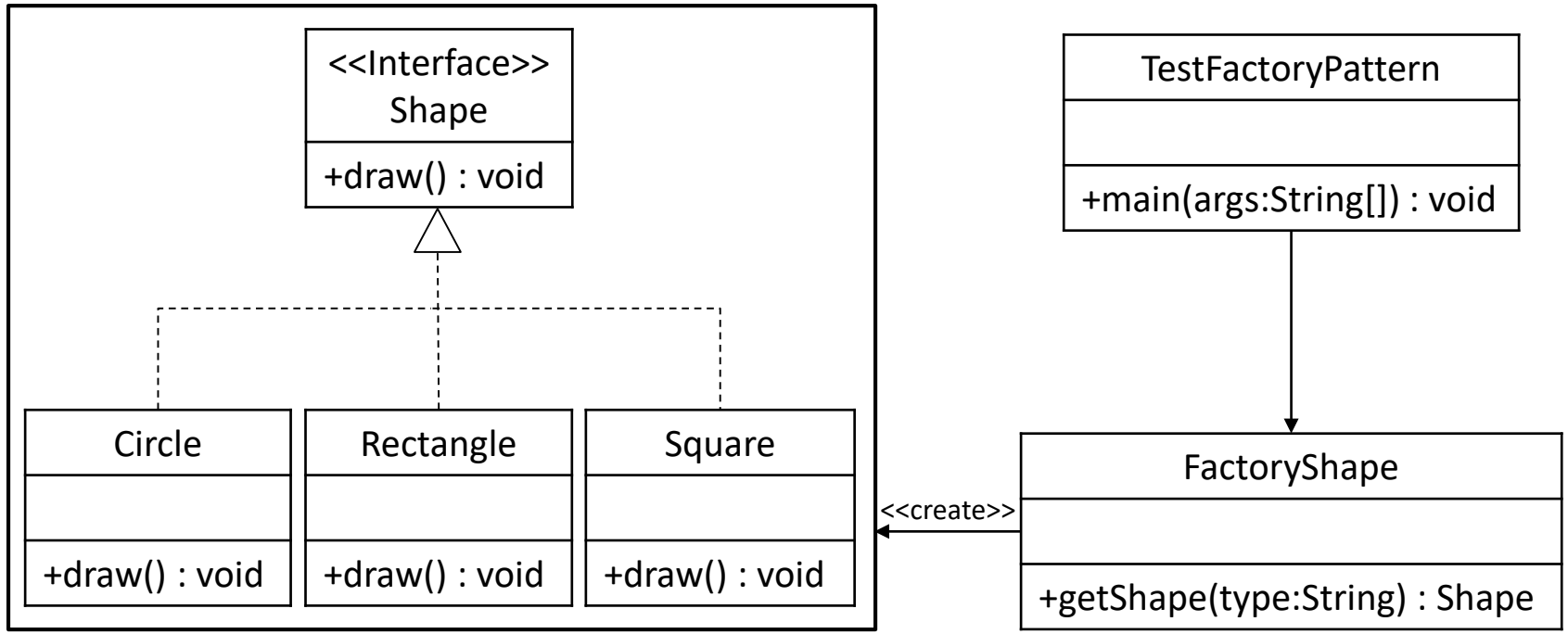
```
01.  …
10.  PersonBean p1 = new PersonBean.Builder("Paul", "Chan").setGender("Male")
11.                .setHeight(180).setAge(20).build();
12.  PersonBean p2 = new PersonBean.Builder("Joe", "Yeung").setGender("Male")
13.                .build();
```

Factory Method Pattern

- Factory method / Factory pattern is one of the mostly used design patterns in Java
- It is an operation that both creates an object and isolates a client from knowing which class to instantiate
- We can create objects without exposing the creation logic to clients and refer to newly created objects using a common interface
- Clients call a factory method to create a new object without using the constructor method

Factory Example

- Let the factory (FactoryShape) determines which class to instantiate when creating an object, and isolates the clients from knowing which class to be instantiated



Factory Shape

```
01.  public interface Shape {
02.    public void draw();
03.  }
04.
05.
06.  class Circle implements Shape {
07.    public void draw() {System.out.println("Circle Shape");}
08.  }
09.
10.  class Rectangle implements Shape {
11.    public void draw() {System.out.println("Rectangle Shape");}
12.  }
13.
14.  class Square implements Shape {
15.    public void draw() {System.out.println("Square Shape");}
16.  }
17.
```

Factory Shape (cont.)

```
01.  public class FactoryShape  throws InstantiationException, IllegalAccessException, ClassNotFoundException  {
02.    public Shape getShape(String type) { // determine which class to create
03.      return (Shape) Class.forName(type).newInstance();
04.    }
05.  }
```

```
01.  public class TestFactoryPattern {
02.    public static void main(String[] args) throws InstantiationException, IllegalAccessException,
     ClassNotFoundException {
03.      FactoryShape factory = new FactoryShape();
04.      Shape s1 = factory.getShape("Rectangle");
05.      s1.draw();
06.      Shape s2 = factory.getShape("Circle");
07.      s2.draw();
08.      Shape s3 = factory.getShape("Square");
09.      s3.draw();
10.    }
11.  }
```

Factory Campus Venue

- Create the campus venue and its subclasses

```
01.  public abstract class CampusVenue {
02.    public abstract int getCapacity();
03.  }
04.  class Classroom extends CampusVenue {
05.    public int getCapacity() { return 25; }
06.    public String toString() { return "Classroom"; }
07.  }
08.  class LectureTheater extends CampusVenue {
09.    public int getCapacity() {return 100;}
10.    public String toString() { return "LectureTheater"; }
11.  }
12.  class Auditorium extends CampusVenue {
13.    public int getCapacity() {return 200;}
14.    public String toString() { return "Auditorium"; }
15.  }
```

Factory Campus Venue (cont.)

- Factory class to generate campus venue

```
01.  public class CampusVenueFactory {
02.    public CampusVenue reserveVenue(int people) {
03.      if (people <= 25) {
04.        return new Classroom();
05.      } else if (people <= 100) {
06.        return new LectureTheater();
07.      } else if (people <= 200) {
08.        return new Auditorium();
09.      } else {
10.        System.out.println("Out of range!");
11.        return null;
12.      }
13.    }
14.  }
15.
```

Factory Campus Venue (cont.)

- Booking a campus venue with different number of people

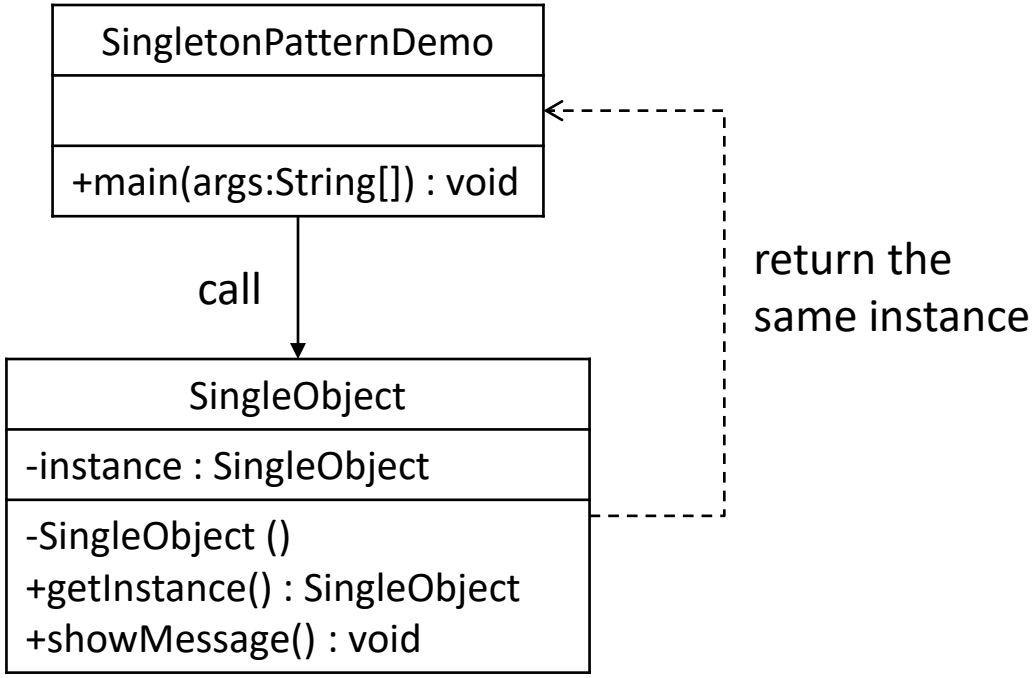
```
01.  public class BookingCampusVenue {
02.    public static void main(String[] args) {
03.      CampusVenueFactory factory = new CampusVenueFactory();
04.      CampusVenue venue = factory.reserveVenue(60);
05.      System.out.println("You have booked a " + venue.toString());
06.      venue = factory.reserveVenue(20);
07.      System.out.println("You have booked a " + venue.toString());
08.      venue = factory.reserveVenue(180);
09.      System.out.println("You have booked a " + venue.toString());
10.    }
11.  }
```

Singleton Pattern

- The intent of the Singleton pattern is to ensure that a class has only one instance and to provide a global point of access to it
- This pattern involves a single class which is responsible to create an object while making sure that only single object gets created
- This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class

Singleton Example

- Define the constructor as *private*
- Example: Database Connection Pool, Message Queue, etc.



Singleton Demonstration

```
01.  package ipm.esap.comp221;
02.
03.  public class SingleObject {
04.    private static SingleObject instance = new SingleObject();
05.
06.    private SingleObject() {
07.      System.out.println("Create a Single Object.");
08.    }
09.
10.    public static SingleObject getInstance() {
11.      return instance;
12.    }
13.
14.    public void showMessage() {
15.      System.out.println("Show some messages.");
16.    }
17.  }
```

Adapter Pattern

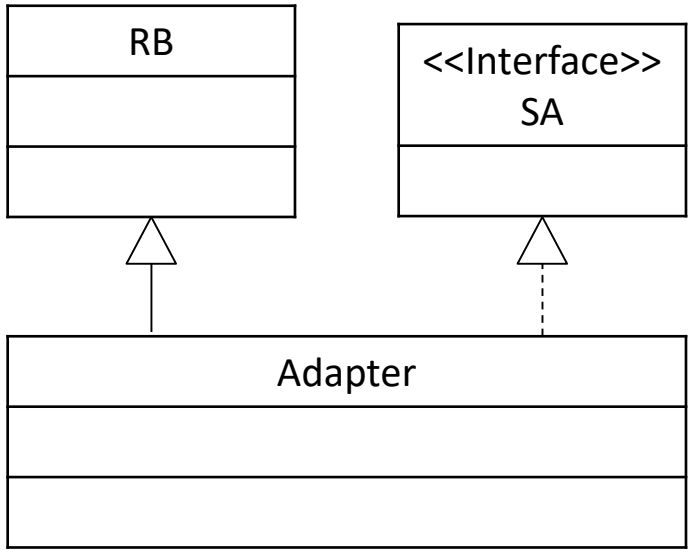
- Adapter pattern works as a bridge between two incompatible interfaces
  - This design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces
  - This design pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces
- A real life example could be a case of card reader which acts as an adapter between memory card and a laptop
  - Plug in the memory card into a card reader and plugin the card reader into a laptop, so that the memory card can be read via the laptop

Adapter Example

- RA(receive only A) <- SA(send only A)

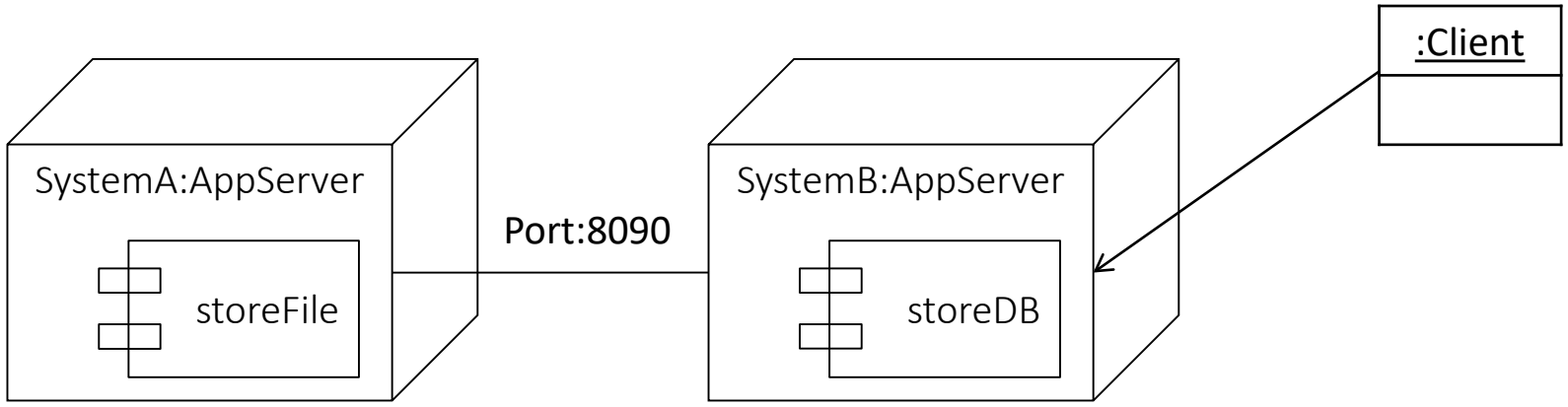
- If there is a new receiver (RB), then create a new adapter for not changing the SA and RA
  - RB <- adapter <- SA

- Adapter class extends RB and implements SA



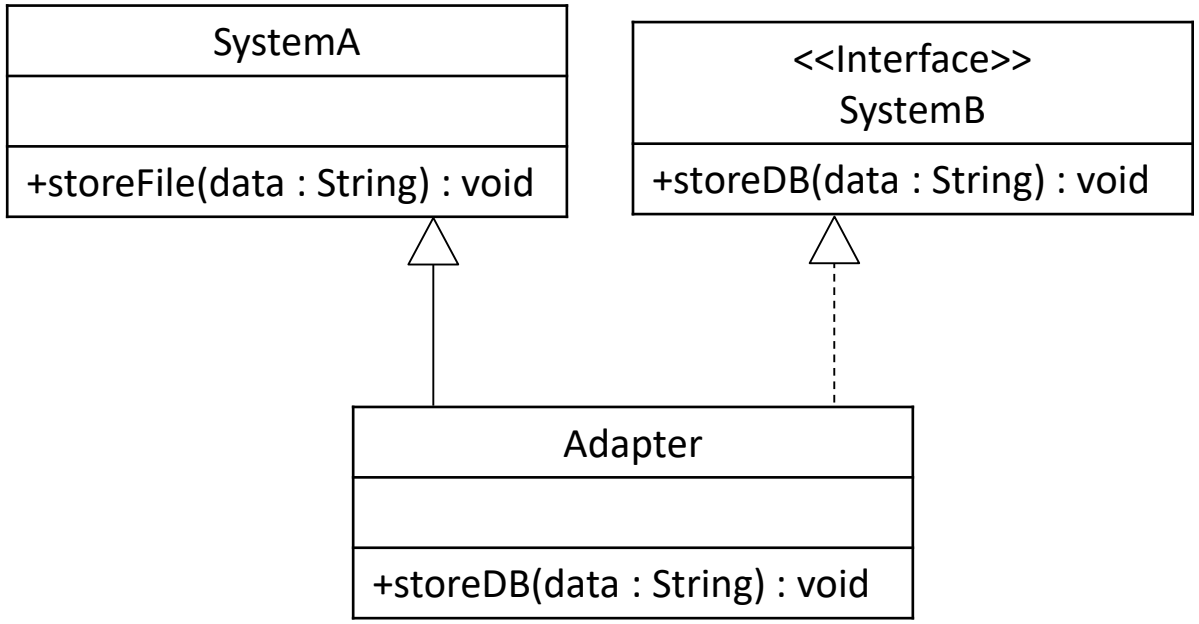
Adapter Example

- System A (a backend system) uses *file systems* to store data while System B (an online system) uses *database* to store data
- Since there are many online systems, the company wants to keep all the online transactional data centrally in the System A



Adapter Example

- We can create an adapter for them
  - System A is the existing system, and System B is the required interface
  - So, the adapter extends System A and implements System B



Adapter Demonstration

- We can write the adapter with the following pattern
- It implements the System B method (*storeDB*)to adapt System A method (*storeFile*)

```
01.  public class Adapter extends SystemA implements SystemB {
02.    public Adapter() {
03.      System.out.println("Data migrating…");
04.    }
05.
06.    // use the System B interface
07.    public void storeDB(String data) {
08.      // adapt the data to system A method (Convert to file format)
09.      super.storeFile(data);
10.    }
11.  }
```

Abstraction

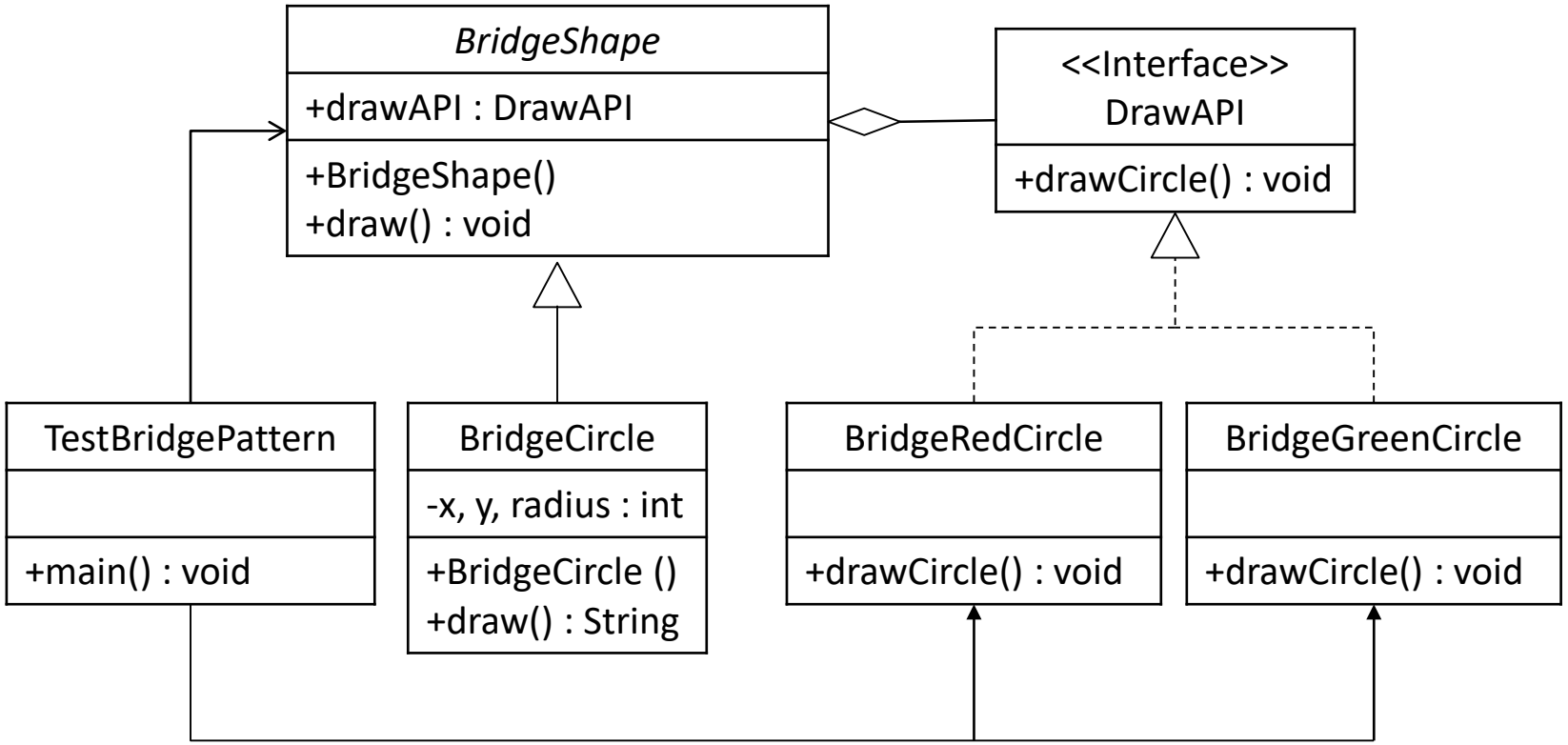
- The ordinary way to implement an abstraction is to create a class hierarchy
- It may end up putting many operations at the top, and some of them may not be used by the subclasses
- The better idea is to create a ***BRIDGE*** by moving the set of abstract operations to an interface

Bridge Pattern

- Bridge design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them
- This pattern involves an interface which acts as a bridge which makes the functionality of the concrete classes independent from the super class
- Both types of classes (abstract, interface) can be altered structurally without affecting each other

Bridge Example

- Two trees to grow without affecting each others
- The input parameters are omitted for simplicity



Bridge Demonstration

- Create an abstract class *BridgeShape* using the *DrawAPI* interface

```
01.  public abstract class BridgeShape {
02.    public DrawAPI drawAPI;
03.
04.    public BridgeShape(DrawAPI drawAPI) {
05.      this.drawAPI = drawAPI;
06.    }
07.
08.    public abstract void draw();
09.  }
```

```
01.  public interface DrawAPI {
02.    public void drawCircle(int radius, int x, int y);
03.  }
```

Bridge Demonstration (cont.)

```
01.  public class BridgeCircle extends BridgeShape {
02.    private int x, y, radius;
03.
04.    public BridgeCircle(int x, int y, int radius, DrawAPI drawAPI) {
05.      super(drawAPI);
06.      this.x = x;
07.      this.y = y;
08.      this.radius = radius;
09.    }
10.
11.    public void draw() {
12.      drawAPI.drawCircle(radius, x, y);
13.    }
14.  }
15.
```

Bridge Demonstration (cont.)

```
01.  public class BridgeRedCircle implements DrawAPI {
02.    @Override
03.    public void drawCircle(int radius, int x, int y) {
04.      System.out.println("Drawing Circle[ color: red, radius: " + radius +
05.                    ", x: " + x + ", " + y + " ]");
06.    }
07.  }
```

```
01.  public class BridgeGreenCircle implements DrawAPI {
02.    @Override
03.    public void drawCircle(int radius, int x, int y) {
04.      System.out.println("Drawing Circle[ color: green, radius: " + radius +
05.                    ", x: " + x + ", " + y + " ]");
06.    }
07.  }
```

Bridge Demonstration (cont.)

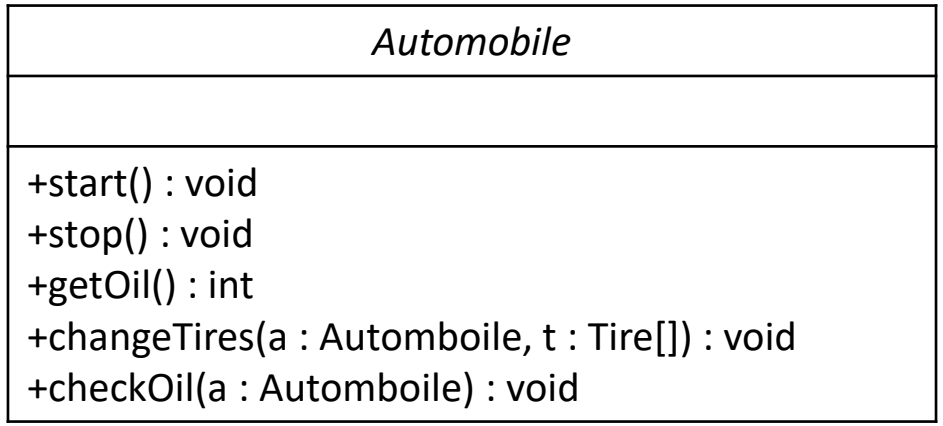
- Use the *BridgeShape* and *DrawAPI* classes to draw different colored circles
- Output

```
Drawing Circle[ color: red, radius: 10, x: 100, 100 ]
Drawing Circle[ color: green, radius: 10, x: 100, 100 ]
```

```
01.  public class TestBridgePattern {
02.    public static void main(String[] args) {
03.      BridgeShape redCircle = new BridgeCircle(100, 100, 10, new BridgeRedCircle());
04.      BridgeShape greenCircle = new BridgeCircle(100, 100, 10, new BridgeGreenCircle());
05.      redCircle.draw();
06.      greenCircle.draw();
07.    }
08.  }
```

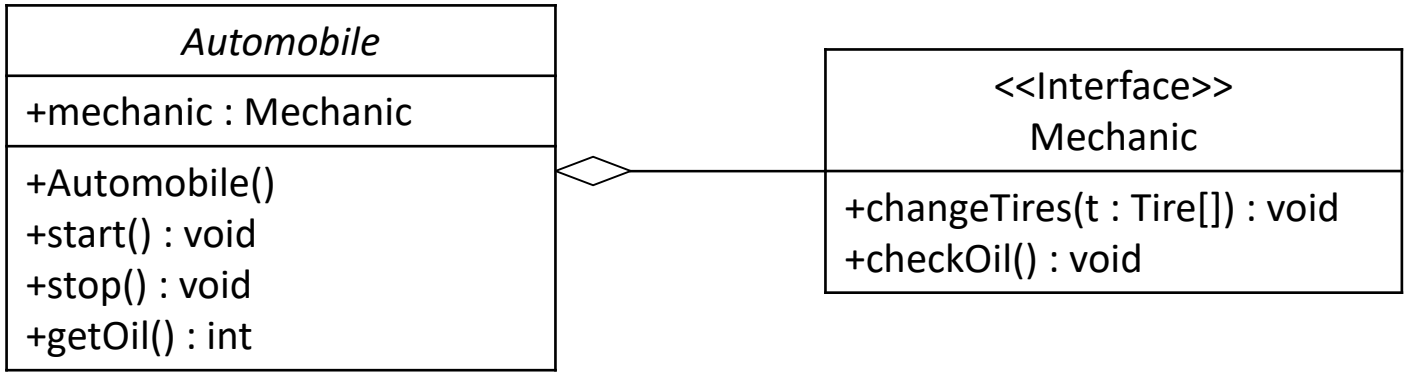
Bridge Example

- An Automobile class has many operations
- We can create a super class to contain all the operations



Bridge Example

- In order to let the super class continue to grow, we can create a bridge by moving the operations to an interface
- Now, we can add more operations under the ***Mechanic*** interface without affecting the ***Automobile*** abstract class, vise versa

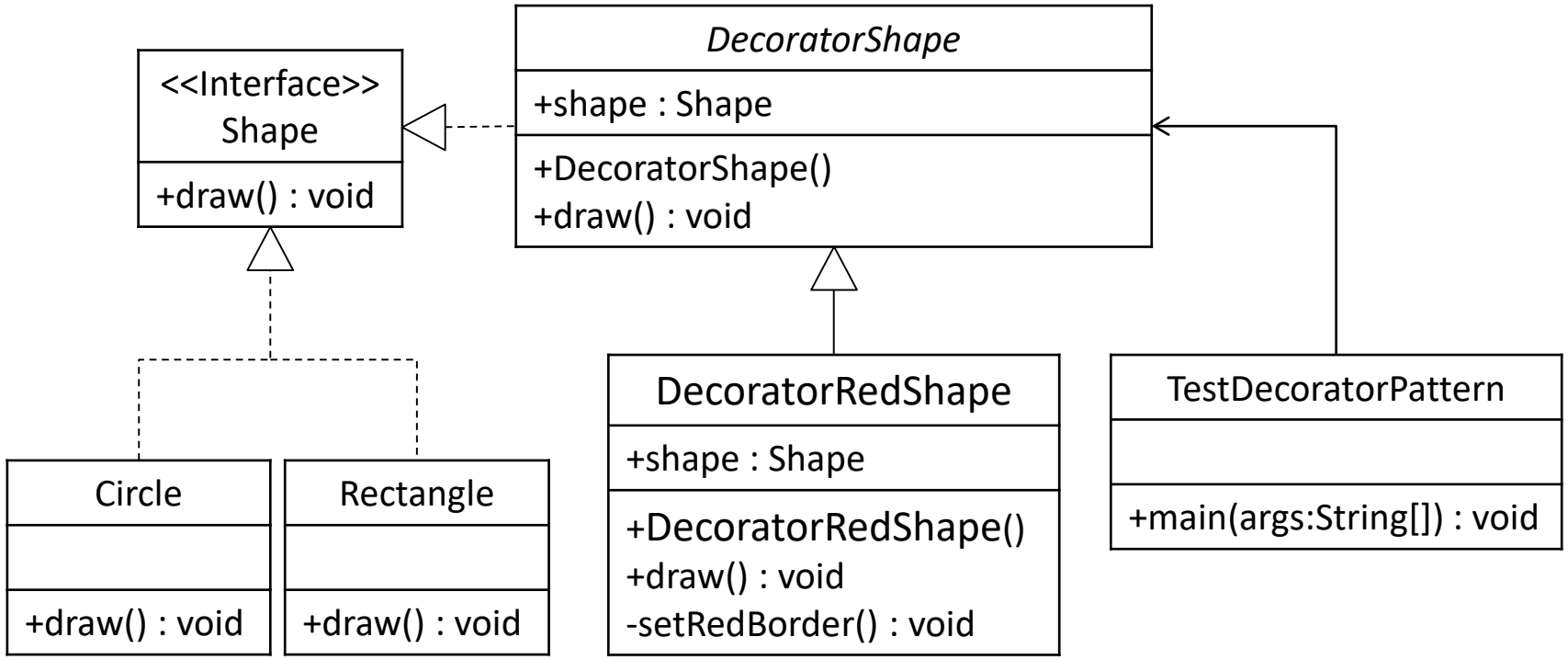


Decorator Pattern

- Decorator pattern allows a user to add new functionality to an existing object without altering its structure
- This pattern creates a decorator class which wraps the original class and provides additional functionality to keep the class methods unchanged

Decorator Example

- Decorator pattern allows a user to add new functionality to an existing object without altering its structure



Decorator Demonstration

```
01.  public interface Shape {
02.    public void draw();
03.  }
04.
05.
06.  class Circle implements Shape {
07.    public void draw() {System.out.println("Circle Shape");}
08.  }
09.
10.  class Rectangle implements Shape {
11.    public void draw() {System.out.println("Rectangle Shape");}
12.  }
13.
14.  class Square implements Shape {
15.    public void draw() {System.out.println("Square Shape");}
16.  }
17.
```

Decorator Demonstration (cont.)

- Create abstract decorator class implementing the Shape interface

```
01.  public abstract class DecoratorShape implements Shape {
02.    public Shape decoratedShape;
03.
04.    public DecoratorShape(Shape decoratedShape) {
05.      this.decoratedShape = decoratedShape;
06.    }
07.
08.    public void draw(){
09.      decoratedShape.draw();
10.    }
11.  }
```

Decorator Demonstration (cont.)

```
01.  public class DecoratorRedShape extends DecoratorShape {
02.    public DecoratorRedShape(Shape decoratedShape) {
03.      super(decoratedShape);
04.    }
05.
06.    @Override
07.    public void draw() {
08.      decoratedShape.draw();
09.      setRedBorder(decoratedShape);
10.    }
11.
12.    // Adding additional methods to the DecoratorShape
13.    private void setRedBorder(Shape decoratedShape) {
14.      System.out.println("Border Color: Red");
15.    }
16.  }
17.
```

Decorator Demonstration (cont.)

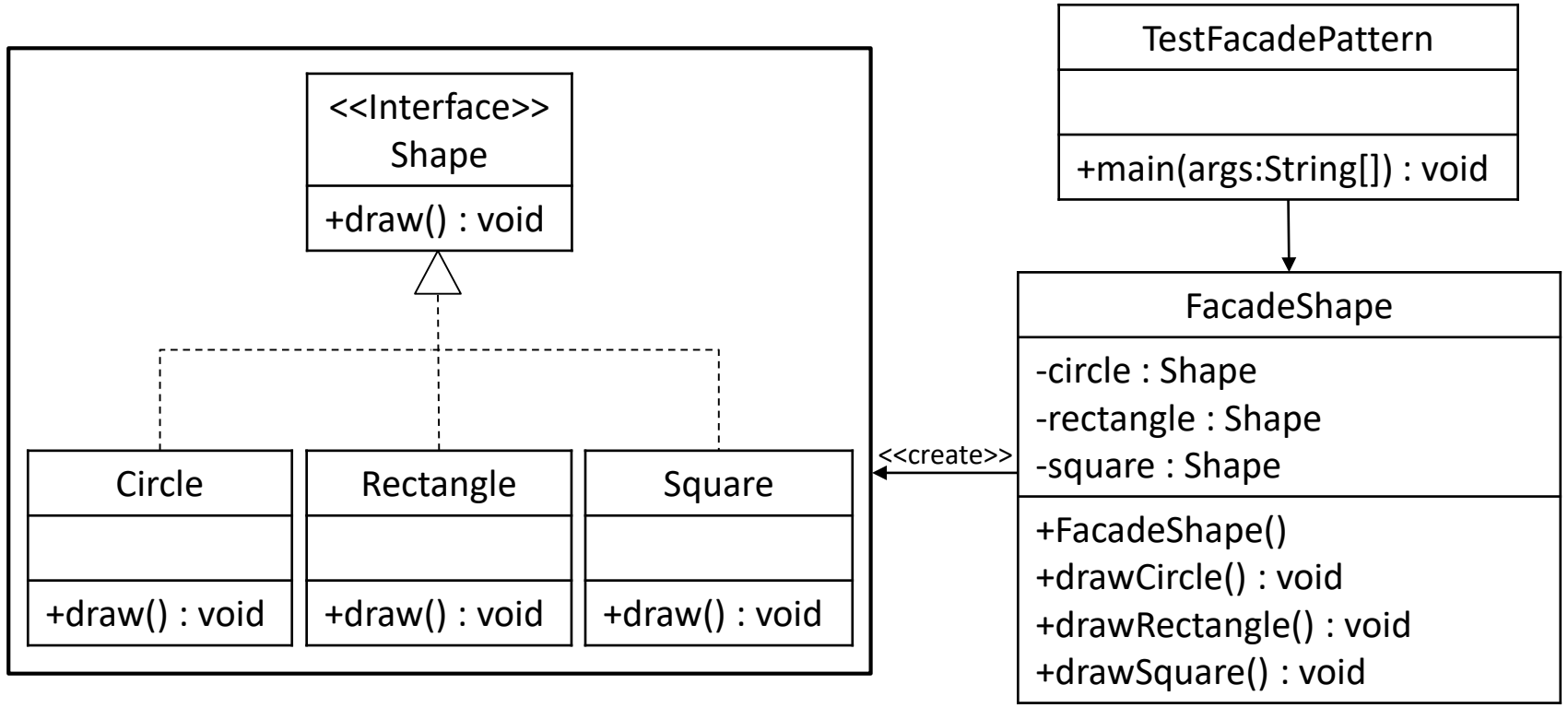
```
01.  public class TestDecoratorPattern {
02.    public static void main(String[] args) {
03.      Shape circle = new Circle();
04.      Shape redCircle = new DecoratorRedShape(new Circle());
05.      Shape redRectangle = new DecoratorRedShape(new Rectangle());
06.
07.      System.out.println("Circle with normal border");
08.      circle.draw();
09.
10.      System.out.println("\nCircle of red border");
11.      redCircle.draw();
12.
13.      System.out.println("\nRectangle of red border");
14.      redRectangle.draw();
15.    }
16.  }
17.
```

Façade Pattern

- Façade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system
- This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities
- This pattern involves a single class which provides ***simplified*** methods required by client and delegates calls to methods of existing system classes

Façade Example

- A main method to connect all objects



Façade Demonstration

```
01.  public interface Shape {
02.    public void draw();
03.  }
04.
05.
06.  class Circle implements Shape {
07.    public void draw() {System.out.println("Circle Shape");}
08.  }
09.
10.  class Rectangle implements Shape {
11.    public void draw() {System.out.println("Rectangle Shape");}
12.  }
13.
14.  class Square implements Shape {
15.    public void draw() {System.out.println("Square Shape");}
16.  }
17.
```

Façade Demonstration (cont.)

```
01.  public class FacadeShape {
02.    private Shape circle, rectangle, square;
03.
04.    public FacadeShape() {
05.      circle = new Circle();
06.      rectangle = new Rectangle();
07.      square = new Square();
08.    }
09.
10.    public void drawCircle() {
11.      circle.draw();
12.    }
13.
14.    public void drawRectangle() {
15.      rectangle.draw();
16.    }
17.
```

Façade Demonstration (cont.)

```
18.     public void drawSquare() {
19.       square.draw();
20.     }
21.   }
```

```
01.   public class TestFacadePattern {
02.     public static void main(String[] args) {
03.       FacadeShape facadeShape = new FacadeShape();
04.       facadeShape.drawCircle();
05.       facadeShape.drawRectangle();
06.       facadeShape.drawSquare();
07.     }
08.   }
```

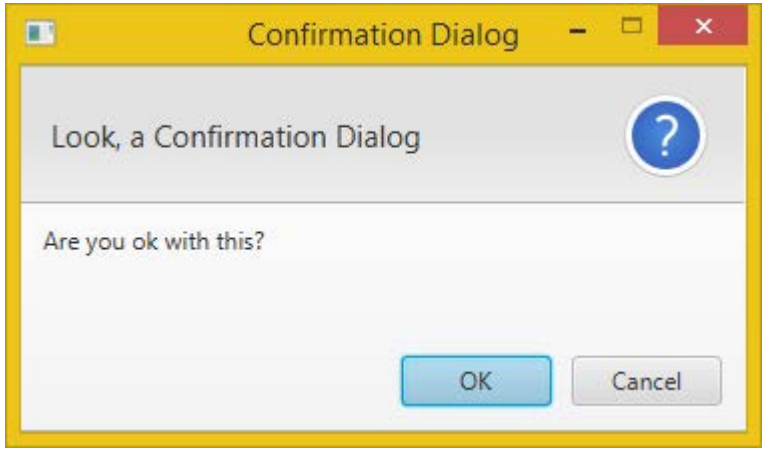
JavaFX Alert class

- A façade is a configurable, reusable class with a higher-level interface that makes the subsystem easier to use
- The JavaFX *Alert* class is one of the examples of using the façade pattern
- Use the *Alert* class with simple setting to create the complex dialog boxes
  - information, warning, error, confirmation, etc.

JavaFX Alert class

- JavaFX *Alert* class confirmation box

```
11.  Alert alert = new Alert(AlertType.CONFIRMATION);
12.  alert.setTitle("Confirmation Dialog");
13.  alert.setHeaderText("Look, a Confirmation Dialog");
14.  alert.setContentText("Are you ok with this?");
15.  alert.showAndWait();
```

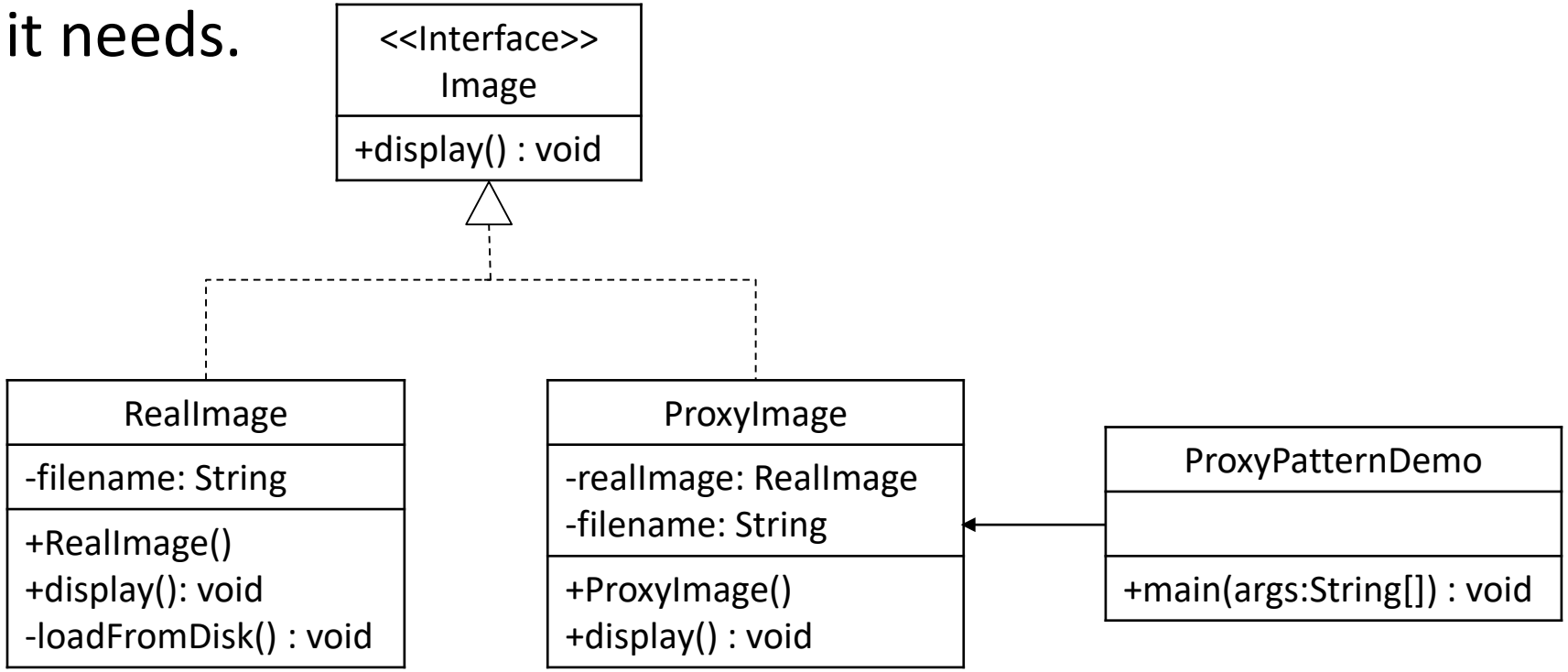


Proxy Pattern

- In proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern.
- In proxy pattern, we create object having original object to interface its functionality to outer world.

Proxy Example

- ProxyImage is a proxy class to reduce memory footprint of RealImage object loading.
- ProxyPatternDemo will use ProxyImage to get an Image object to load and display as it needs.



Proxy Demonstration

```
01.  public interface Image {
02.    public void display();
03.  }
04.
05.
06.  public class RealImage implements Image {
07.     private String fileName;
08.     public RealImage(String fileName) {
09.          this.fileName = fileName;
10.         loadFromDisk(fileName);
11.     }
12.    @Override
13.     public void display() {
14.          System.out.println("Displaying " + fileName);
15.     }
16.     private void loadFromDisk(String fileName) {
17.          System.out.println("Loading " + fileName);
18.     }
19.  }
```

Proxy Demonstration

```
20.  public class ProxyImage implements Image {
21.    private RealImage realImage;
22.    private String fileName;
23.    public ProxyImage(String fileName) {
24.      this.fileName = fileName;
25.    }
26.    @Override
27.    public void display() {
28.        if(realImage == null) {
29.          realImage = new RealImage(fileName);
30.        }
31.        realImage.display();
32.    }
33.  }
34.
```

Proxy Demonstration

```
35.  public class ProxyPatternDemo {
36.    public static void main(String[] args) {
37.      Image image = new ProxyImage("test_10mb.jpg");
38.
39.       //image will be loaded from disk
40.       image.display();
41.       System.out.println("");
42.
43.       //image will not be loaded from disk
44.       image.display();
45.    }
46.  }
```

Loop / Iteration

- We have several approaches to loop. They usually due with arrays only
  - for loop with integer indexes
  - while loop with integer indexes
  - extended for loop (foreach)
- The alternative way to do iteration is recursion

```
01.  // recursion
02.  public long factorial(long n) {
03.    if (n <= 1) {
04.      return 1;
05.    } else {
06.      return n * factorial(n - 1);
07.    }
08.  }
```

Common loop patterns

```
01.  int index = 0;
02.  List<String> fruitList = new ArrayList<String>();
03.  fruitList.add("Apple"); fruitList.add("Banana"); fruitList.add("Cherry");
04.
05.  while (index <= 10) { // while loop
06.    System.out.println("Counting: " + index++);
07.  }
08.
09.  for (int i = 10; i > 0; --i) { // for loop
10.    System.out.println("Counting down: " + i);
11.  }
12.
13.  for (String fruit : fruitList) { // extended for loop
14.    System.out.println(fruit);
15.  }
16.
17.  fruitList.forEach(fruit -> System.out.println(fruit)); // lambda Express
```

Iterator Pattern

- Iterator pattern is to provide a way to access the elements of a collection sequentially
- We will use the *Iterator* class for demonstrating the Iterator Pattern
- The Iterator class can work with a collection
  - Iterator class: hasNext(), next(), remove()
- Control how to loop the data from a collection
  - order
  - content

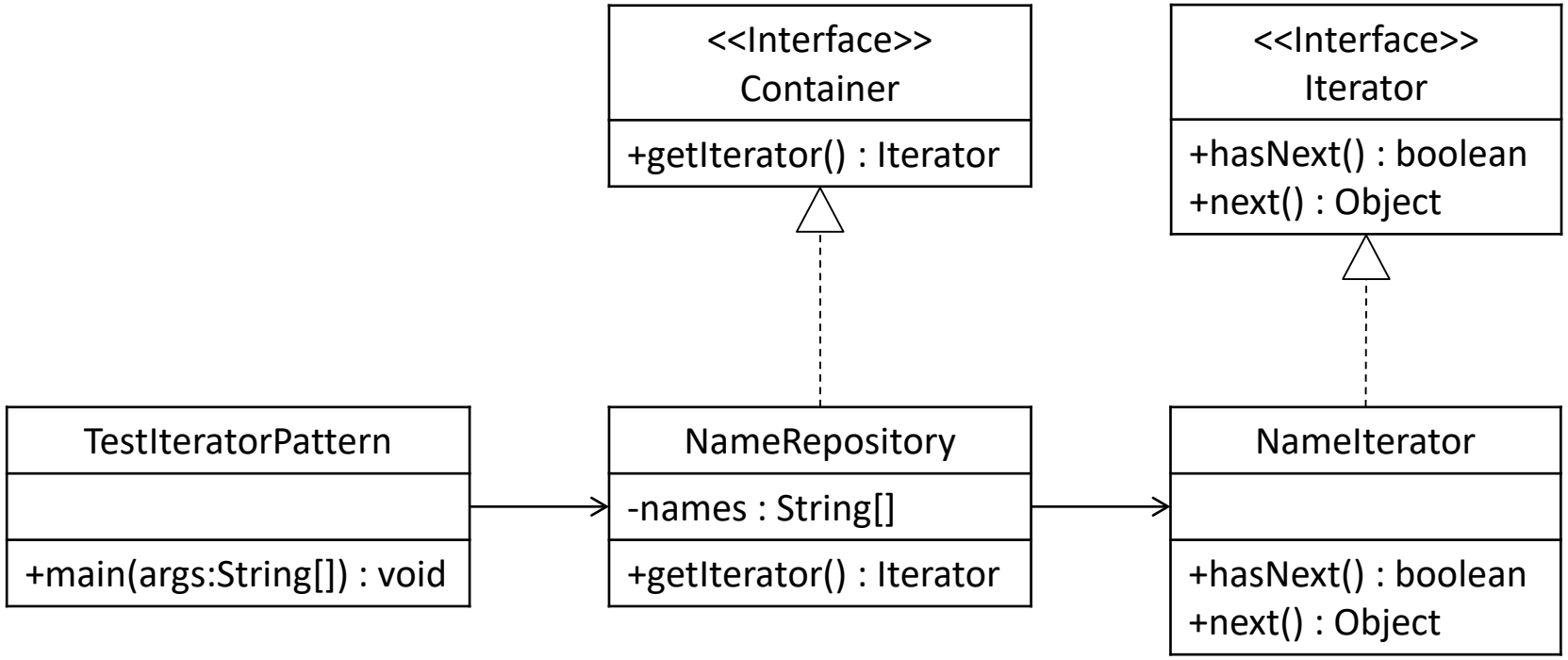
Iterator Demonstration

- Iterator Example (no constructor)

```
01.  List students = new ArrayList();
02.  students.add("Paul Chan");
03.  students.add("Joe Yeung");
04.  students.add("Pinky Lam");
05.  Iterator iter = students.iterator();
06.  while (iter.hasNext()) {
07.    System.out.println(iter.next());
08.  }
```

Iterator Example

- Implements the Iterator interface to define your own way to retrieve data



Iterator Demonstration

```
01.  public class TestIteratorPattern {
02.    public static void main(String[] args) {
03.      NameRepository namesRepository = new NameRepository();
04.      for (Iterator iter = namesRepository.getIterator(); iter.hasNext();) {
05.        String name = (String) iter.next();
06.        System.out.println("Name : " + name);
07.      }
08.    }
09.  }
```

```
01.  public interface Iterator {
02.    public boolean hasNext();
03.    public Object next();
04.  }
```

```
01.  public interface Container {
02.    public Iterator getIterator();
03.  }
```

Iterator Demonstration

```
01.  public class NameRepository implements Container {
02.    private String names[] = {"Paul Chan", "Joe Yeung", "Pinky Lam"};
03.
04.    @Override
05.    public Iterator getIterator() {
06.      return new NameIterator();
07.    }
08.
09.    private class NameIterator implements Iterator {
10.      int index;
11.
12.      @Override
13.      public boolean hasNext() {
14.        if (index < names.length) {
15.          return true;
16.        }
17.        return false;
```

Iterator Demonstration

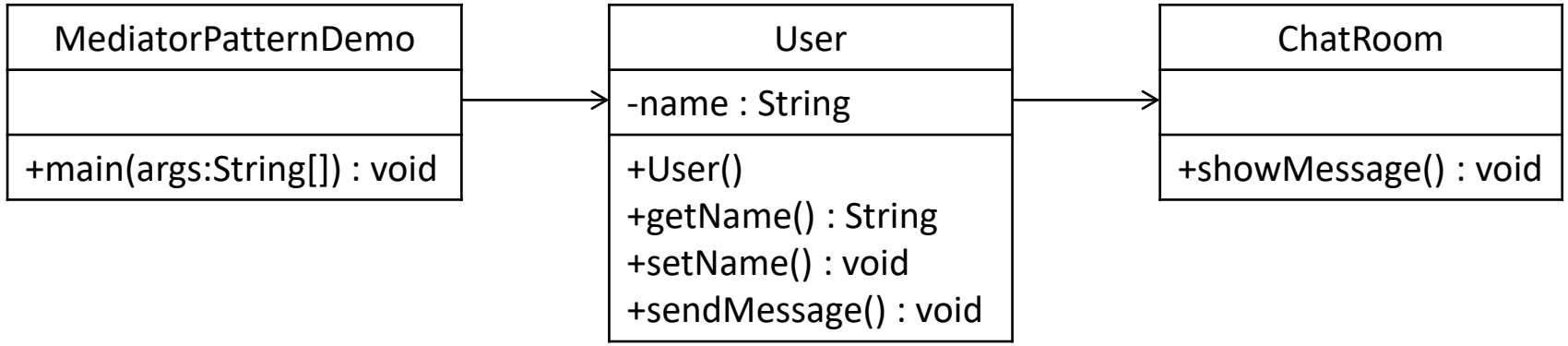
```
18.         }
19.
20.         @Override
21.         public Object next() {
22.           if (this.hasNext()) {
23.             return names[index++];
24.           }
25.           return null;
26.         }
27.       }
28.     }
29.
30.
```

Mediator Pattern

- Mediator pattern is used to reduce communication complexity between multiple objects or classes
- This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling
- Promote the many-to-many relationships between interacting peers to "full object status“
- A single mediator class to replace multiple adapter classes

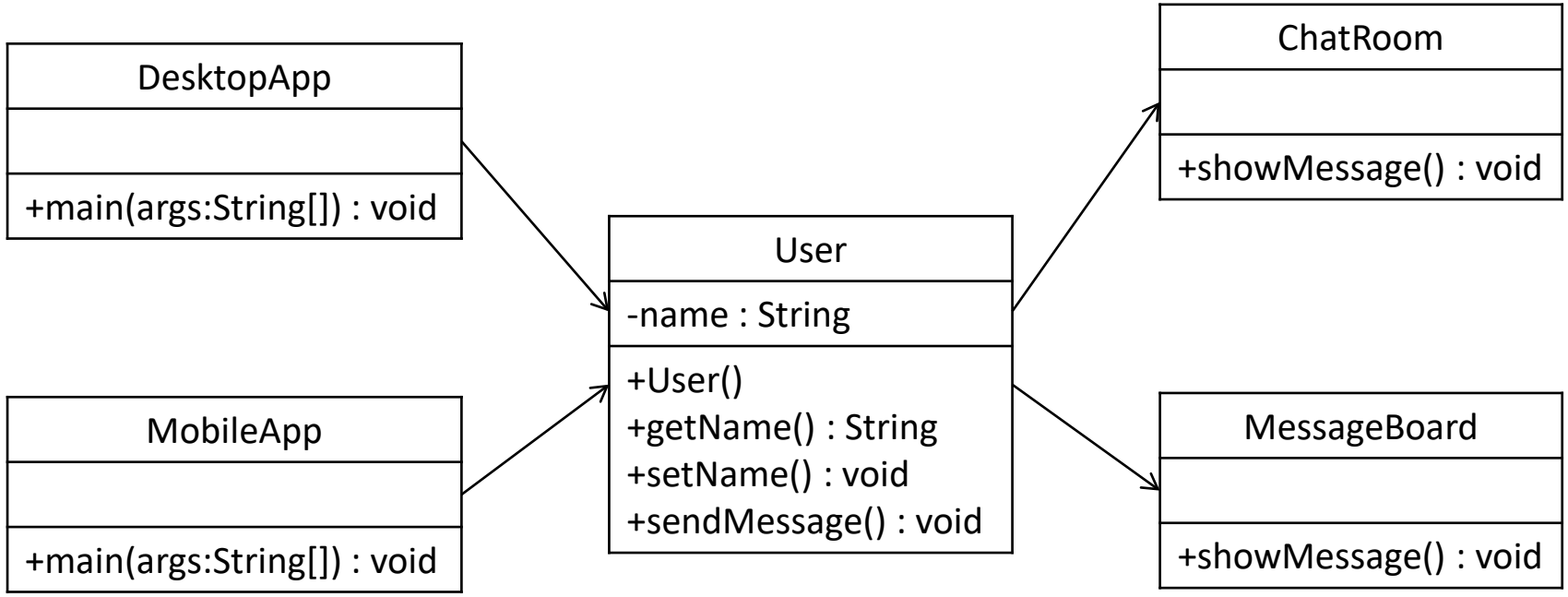
Mediator Example

- The classes are linked



Mediator Example (cont.)

- The classes are linked as star



Mediator Demonstration

```
01.  public class ChatRoom {
02.    public static void showMessage(User user, String message) {
03.      System.out.println(new Date().toString());
04.      System.out.println(" [" + user.getName() + "] : " + message);
05.    }
06.
```

```
01.  public class User {
02.    private String name;
03.    public User(String name) {this.name = name;}
04.    public String getName() {return name;}
05.    public void setName(String name) {this.name = name;}
06.    // Multiple users can send message to chatroom
07.    public void sendMessage(String message) {
08.      ChatRoom.showMessage(this, message);
09.    }
10.  }
```

Mediator Demonstration

- Use the *User* object to show communications between them

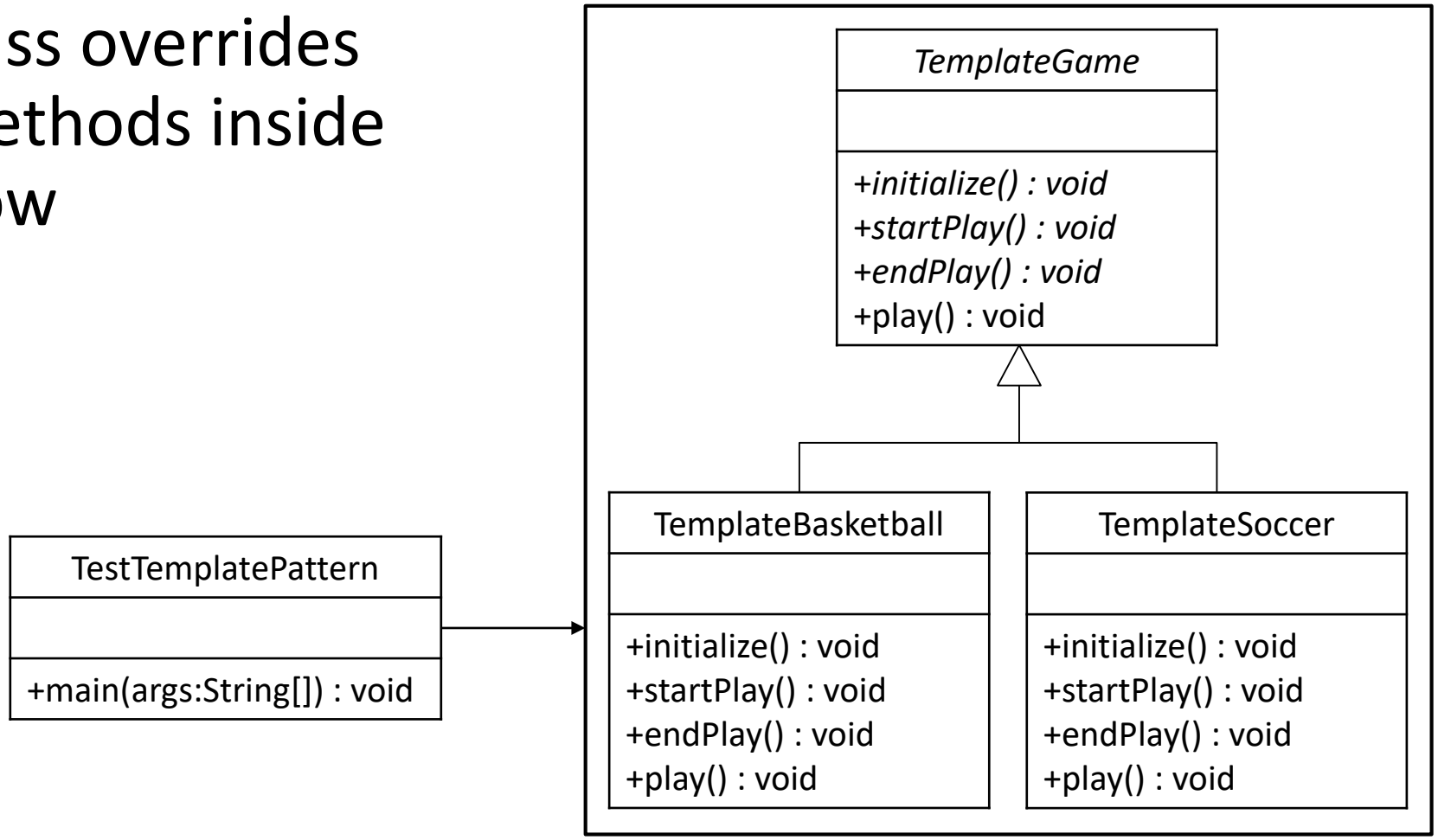
```
01.  public class TestMediatorPattern {
02.    public static void main(String[] args) {
03.      User paul = new User("Paul Chan");
04.      User joe = new User("Joe Yeung");
05.      paul.sendMessage("Hi! Joe!");
06.      joe.sendMessage("Hello! Paul!");
07.    }
08.  }
```

Template Pattern

- In Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods
- Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class
- Super class has a flow to contain a list of methods
- Each subclass follow the super class flow but override the methods inside the flow

Template Example

- Subclass overrides the methods inside the flow



Template Demonstration

```
01.  public abstract class TemplateGame {
02.    abstract void initialize();
03.    abstract void startPlay();
04.    abstract void endPlay();
05.
06.    // template method (work flow)
07.    public final void play() {
08.      // initialize the game
09.      initialize();
10.      // start game
11.      startPlay();
12.      // end game
13.      endPlay();
14.    }
15.  }
16.
17.
```

Template Demonstration

```
01.  public class TemplateBasketball extends TemplateGame {
02.    @Override
03.    void initialize() {
04.      System.out.println("Basketball Game Initialized! Start playing.");
05.    }
06.
07.    @Override
08.    void startPlay() {
09.      System.out.println("Basketball Game Started. Enjoy the game!");
10.    }
11.
12.    @Override
13.    void endPlay() {
14.      System.out.println("Basketball Game Finished!");
15.    }
16.  }
17.
```

Template Demonstration

```
01.  public class TemplateSoccer extends TemplateGame {
02.    @Override
03.    void initialize() {
04.      System.out.println("Soccer Game Initialized! Start playing.");
05.    }
06.
07.    @Override
08.    void startPlay() {
09.      System.out.println("Soccer Game Started. Enjoy the game!");
10.    }
11.
12.    @Override
13.    void endPlay() {
14.      System.out.println("Soccer Game Finished!");
15.    }
16.  }
17.
```

Template Demonstration

- Use the Game's template method play() to demonstrate a defined way of playing game

```
01.  public class TestTemplatePattern {
02.    public static void main(String[] args) {
03.      TemplateGame game = new TemplateBasketball();
04.      game.play();
05.      game = new TemplateSoccer();
06.      game.play();
07.    }
08.  }
```

Summary

- There are different types of design patterns such as creational, structural, and behavioral patterns
- They can helps us to solve problem quickly based on the patterns
- Following the design pattern can make our codes more extendable, reusable, and maintainable