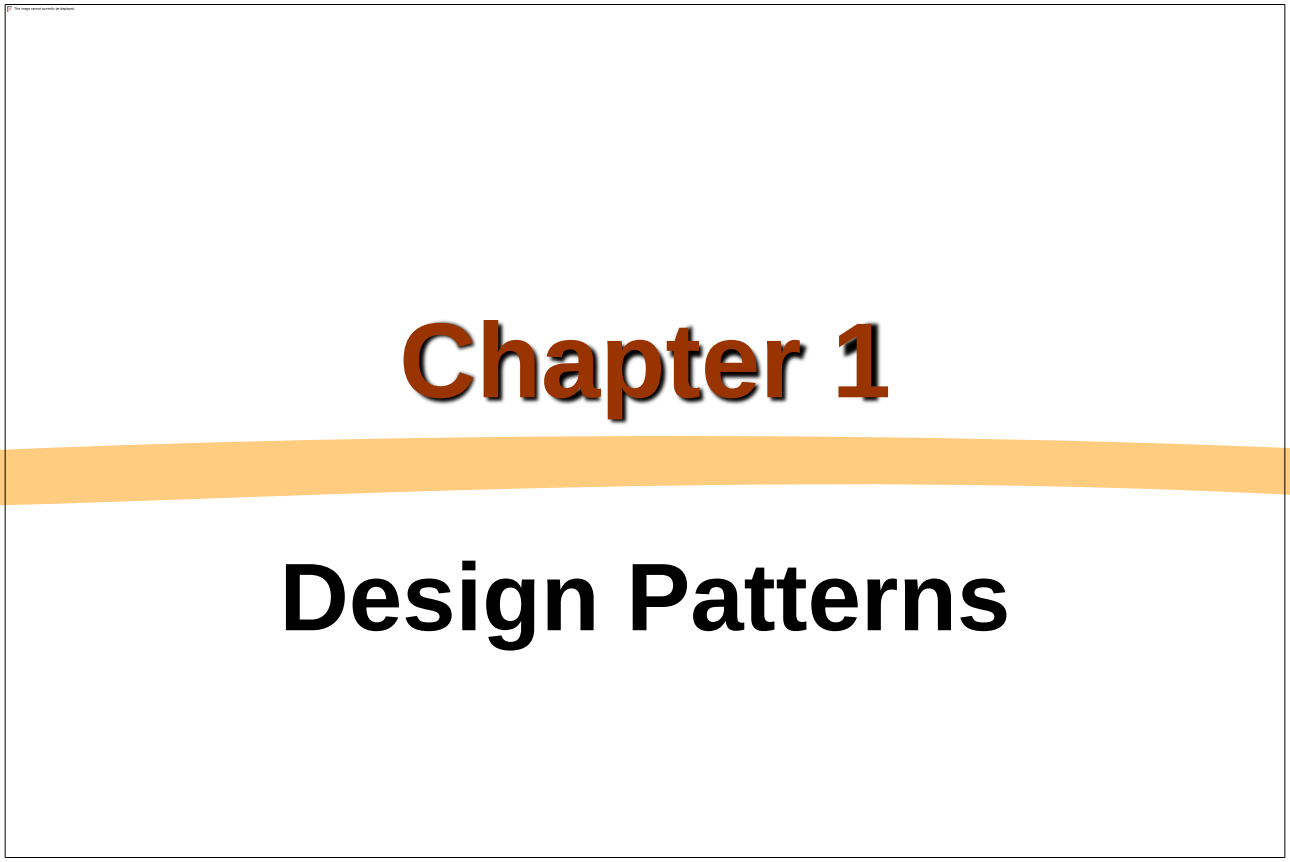


Chapter 1

Design Patterns

Design Patterns

- A **design pattern** is a general repeatable solution to a commonly occurring problem in software design.
- It is a description or template for how to solve a problem that can be used in many different situations.
- Standard solutions to commonly recurring problems.
- A *pattern* is the outline of a reusable solution to a general problem encountered in a particular context.
- Many of them have been systematically documented for all software developers to use.

Design Patterns

- A good pattern should
  - Be as general as possible
  - Contain a solution that has been proven to effectively solve the problem in the indicated context.
- *Studying patterns is an effective way to learn from the experience of others*

Pattern description

- **Context**

  The general situation in which the pattern applies.

- **Problem**

  A short sentence or two raising the main difficulty.

- **Forces**

  The issues or concerns to consider when solving the problem.

- **Solution**

  The recommended way to solve the problem in the given context.

Pattern description

- **Antipatterns** (Optional)

  Solutions that are inferior or do not work in this context.

- **Related patterns** (Optional)

  Patterns that are similar to this pattern.

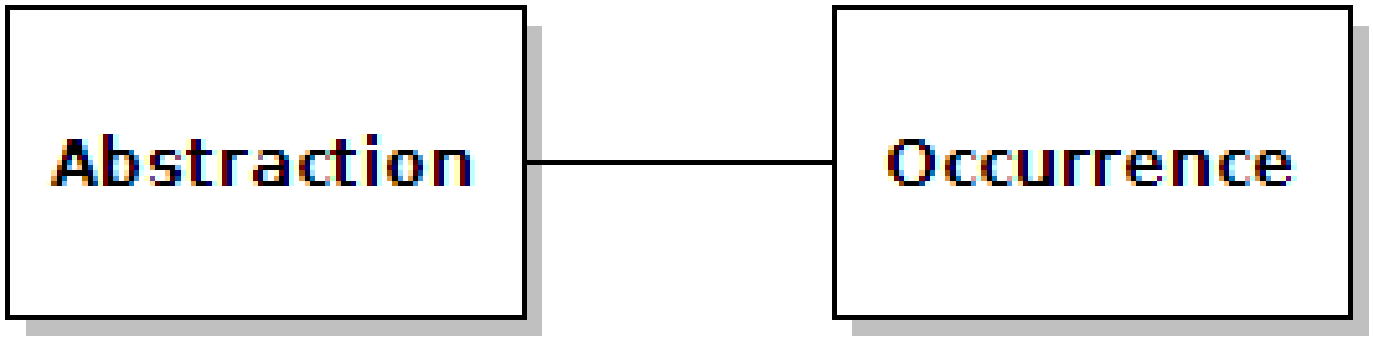
- **References**

  Who developed or inspired the pattern.

The Abstraction-Occurrence Pattern

- ***Context***:
  - Often in a domain model you find a set of related objects (*occurrences)*.
  - The members of such a set share common information
    - but also differ from each other in important ways.
- ***Problem***:
  - What is the best way to represent such sets of occurrences in a class diagram?
- ***Forces***:
  - You want to represent the members of each set of occurrences without duplicating the common information

The Abstraction-Occurrence Pattern

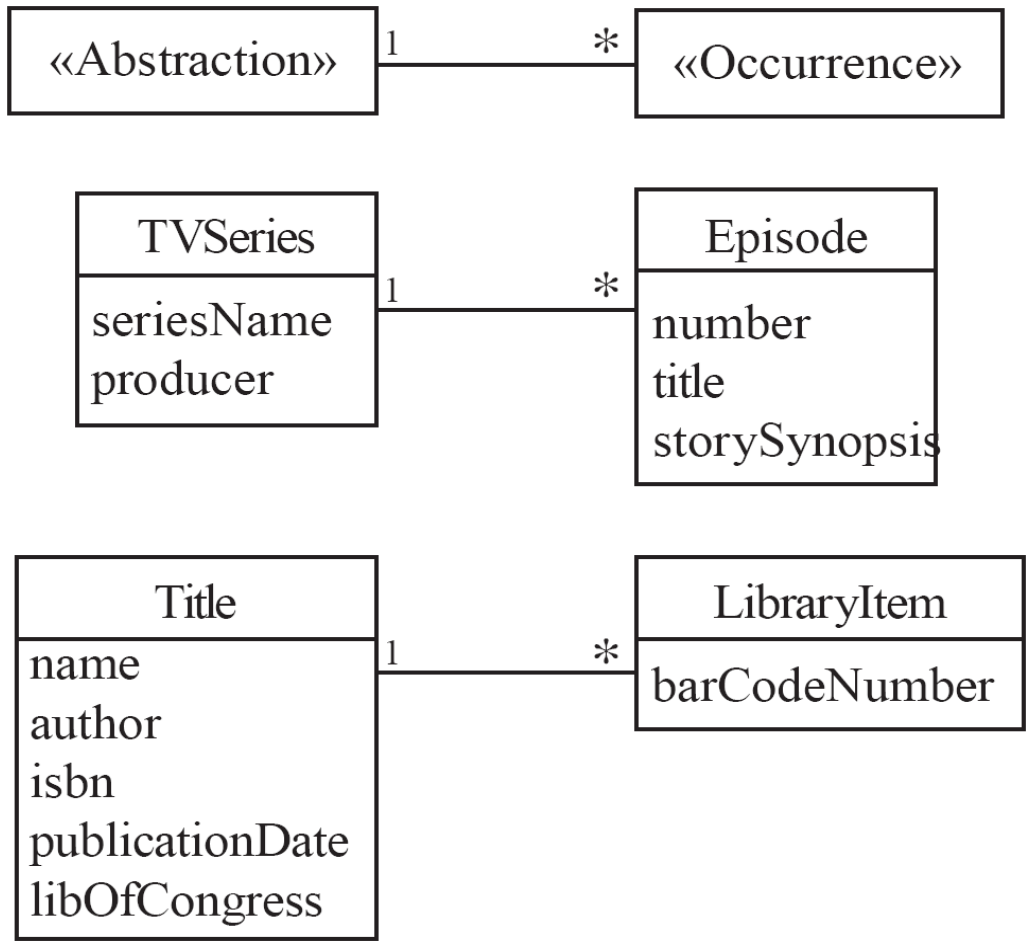


Abstraction - instances of this class represent abstractions: groupings of things that are abstractly the same.

Occurrence - instances of this class represent things that can be grouped into abstractions.

The Abstraction-Occurrence Pattern

- ***Solution:***



The Abstraction-Occurrence Pattern

```
// Abstraction class: Represents the general concept of a Book
class Book {
    private String title;
    private String author;
    private String ISBN;

    public Book(String title, String author, String ISBN) {
        this.title = title;
        this.author = author;
        this.ISBN = ISBN;
    }
```

The Abstraction-Occurrence Pattern

```
// Getters for abstraction properties
    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public String getISBN() {
        return ISBN;
    }
```

The Abstraction-Occurrence Pattern

```
 @Override
    public String toString() {
        return "Book: " + title + ", Author: " + author + ", ISBN: " +
ISBN;
    }
}

// Occurrence class: Represents individual copies of a book
class BookCopy {
    private int copyNumber; // Unique for each copy
    private String status;  // e.g., Available, Issued, Damaged
    private Book book;      // Link to the abstraction (Book)
```

The Abstraction-Occurrence Pattern

```
public BookCopy(int copyNumber, String status, Book book) {
        this.copyNumber = copyNumber;
        this.status = status;
        this.book = book;
    }

    // Getters and setters
    public int getCopyNumber() {
        return copyNumber;
    }

    public String getStatus() {
        return status;
    }
```

The Abstraction-Occurrence Pattern

```
public void setStatus(String status) {
        this.status = status;
    }

    public Book getBook() {
        return book;
    }

    @Override
    public String toString() {
        return "BookCopy: Copy #" + copyNumber + ", Status: " +
status + ", " + book.toString();
    }
}
```

The Abstraction-Occurrence Pattern

```
// Main class to test the pattern
public class LibrarySystem {
    public static void main(String[] args) {
        // Create the Abstraction (a book)
        Book book = new Book("Effective Java", "Joshua Bloch",
"1234567890");
        // Create multiple occurrences (copies of the book)
        BookCopy copy1 = new BookCopy(1, "Available", book);
        BookCopy copy2 = new BookCopy(2, "Issued", book);
        BookCopy copy3 = new BookCopy(3, "Damaged", book);
        // Display details of all book copies
        System.out.println(copy1);
        System.out.println(copy2);
        System.out.println(copy3);    }}
```

The Abstraction-Occurrence Pattern

Output:

BookCopy: Copy #1, Status: Available, Book: Effective Java, Author: Joshua Bloch, ISBN: 1234567890

BookCopy: Copy #2, Status: Issued, Book: Effective Java, Author: Joshua Bloch, ISBN: 1234567890

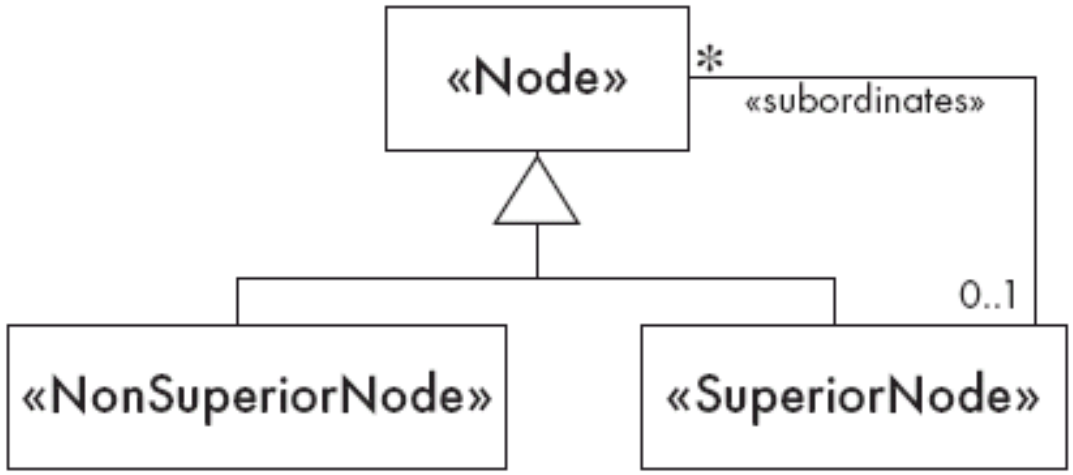
BookCopy: Copy #3, Status: Damaged, Book: Effective Java, Author: Joshua Bloch, ISBN: 1234567890

The General Hierarchy Pattern

- ***Context***:
  - Objects in a hierarchy can have one or more objects above them (superiors),
    - and one or more objects below them (subordinates).
  - Some objects cannot have any subordinates
- ***Problem***:
  - How do you represent a hierarchy of objects, in which some objects cannot have subordinates?
- ***Forces***:
  - You want a flexible way of representing the hierarchy
    - that prevents certain objects from having subordinates
  - All the objects have many common properties and operations

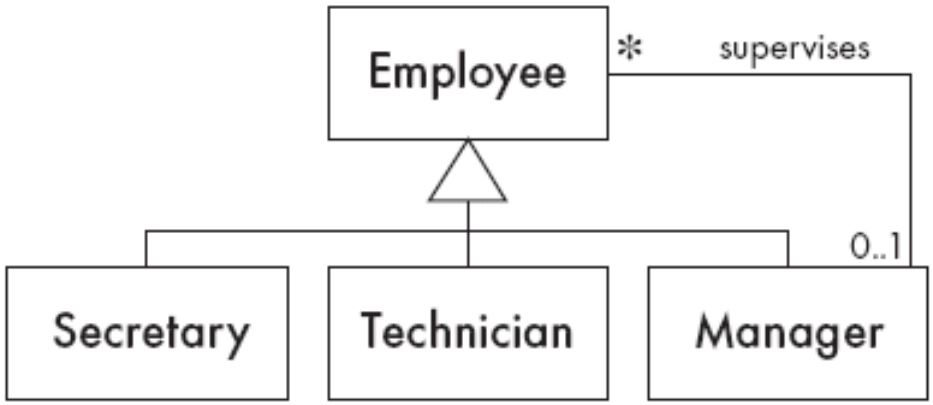
The General Hierarchy Pattern

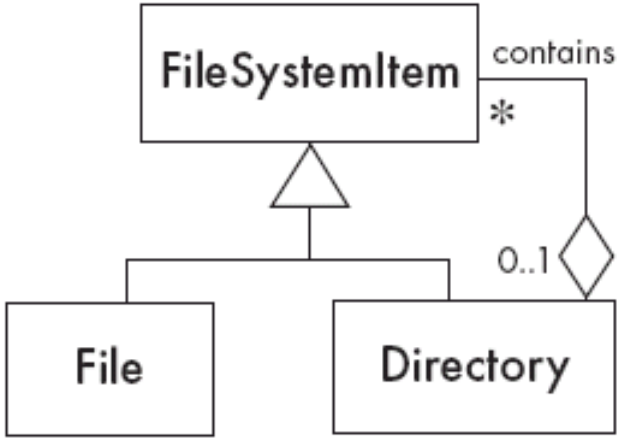
- ***Solution:***



The General Hierarchy Pattern

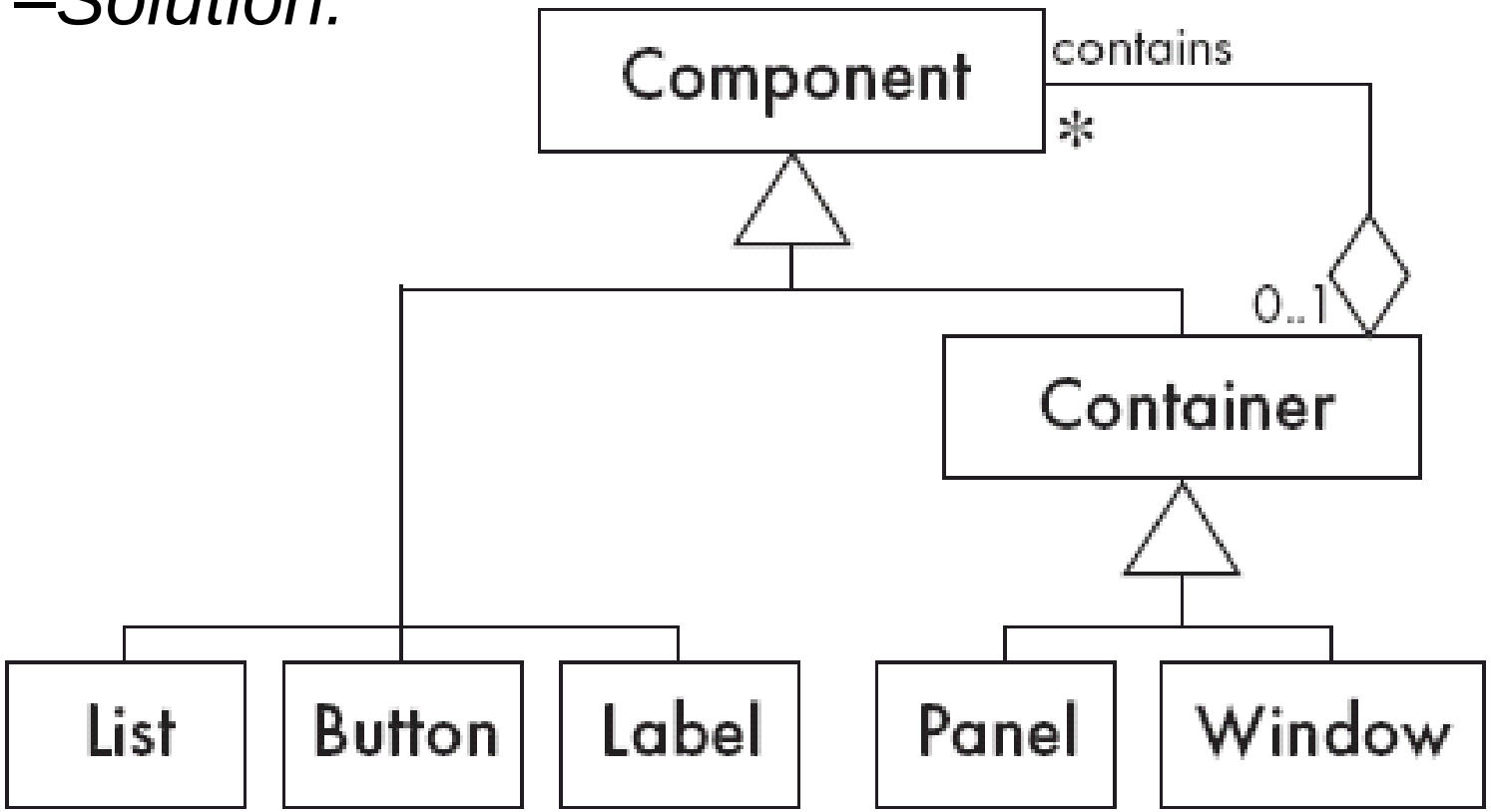
- ***Solution:***





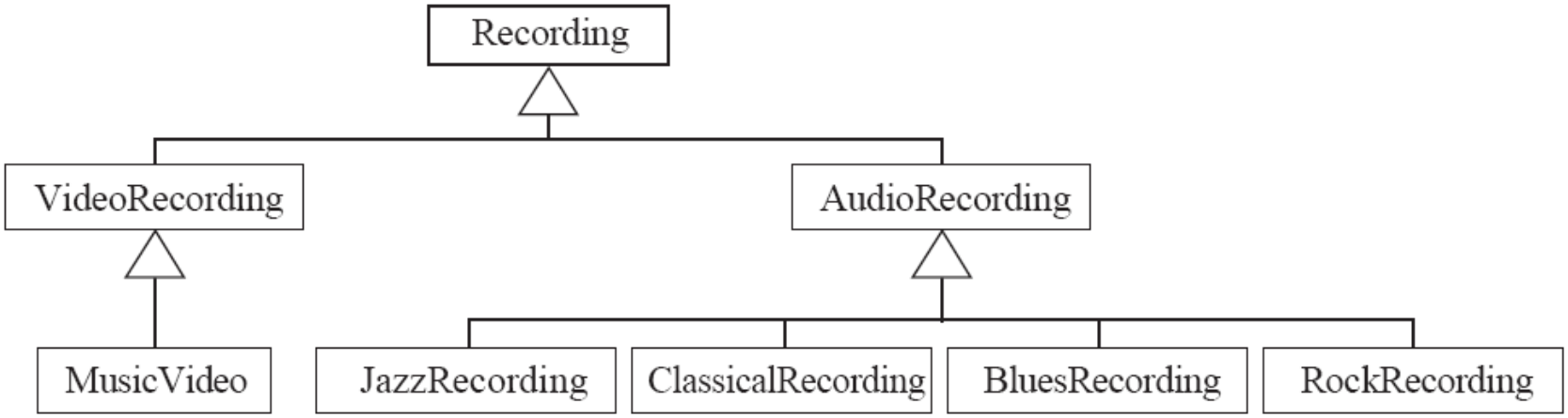
The General Hierarchy Pattern

–*Solution:*



The General Hierarchy Pattern

- Antipattern:



The General Hierarchy Pattern

```
import java.util.ArrayList;
import java.util.List;
// General class representing an Employee in a hierarchy
class Employee {
    private String name;
    private String position;
    private List<Employee> subordinates; // List of child nodes
     (employees)

    // Constructor to initialize an employee
    public Employee(String name, String position) {
        this.name = name;
        this.position = position;
        this.subordinates = new ArrayList<>();
    }
```

The General Hierarchy Pattern

```
// Add a subordinate to this employee
    public void addSubordinate(Employee e) {
        subordinates.add(e);
    }

    // Display the hierarchy starting from this employee
    public void displayHierarchy(String indent) {
        System.out.println(indent + position + ": " + name);
        for (Employee e : subordinates) {
            e.displayHierarchy(indent + "   ");
        }
    }
}
```

The General Hierarchy Pattern

```
// Main class to test the General Hierarchy Pattern
public class GeneralHierarchyExample {
    public static void main(String[] args) {
        // Creating the top-level entity (CEO)
        Employee ceo = new Employee("John Doe", "CEO");

        // Creating Managers reporting to CEO
        Employee headSales = new Employee("Alice Smith", "Head of
    Sales");
        Employee headMarketing = new Employee("Bob Johnson",
    "Head of Marketing");
```

The General Hierarchy Pattern

```
// Adding Managers under CEO
    ceo.addSubordinate(headSales);
    ceo.addSubordinate(headMarketing);


    // Creating employees under Sales and Marketing departments
    Employee salesExecutive1 = new Employee("Charlie Brown",
"Sales Executive");
    Employee salesExecutive2 = new Employee("Diana Prince",
"Sales Executive");
    Employee marketingExecutive1 = new Employee("Ethan
Hunt", "Marketing Executive");
```

The General Hierarchy Pattern

```
// Adding subordinates under Sales Manager
        headSales.addSubordinate(salesExecutive1);
        headSales.addSubordinate(salesExecutive2);

        // Adding subordinate under Marketing Manager
        headMarketing.addSubordinate(marketingExecutive1);

        // Displaying the complete hierarchy
        System.out.println("Company Hierarchy:");
        ceo.displayHierarchy("");
    }
}
```

The General Hierarchy Pattern

Output:

Company Hierarchy:

CEO: John Doe

  Head of Sales: Alice Smith

    Sales Executive: Charlie Brown

    Sales Executive: Diana Prince

  Head of Marketing: Bob Johnson

    Marketing Executive: Ethan Hunt

The Player-Role Pattern

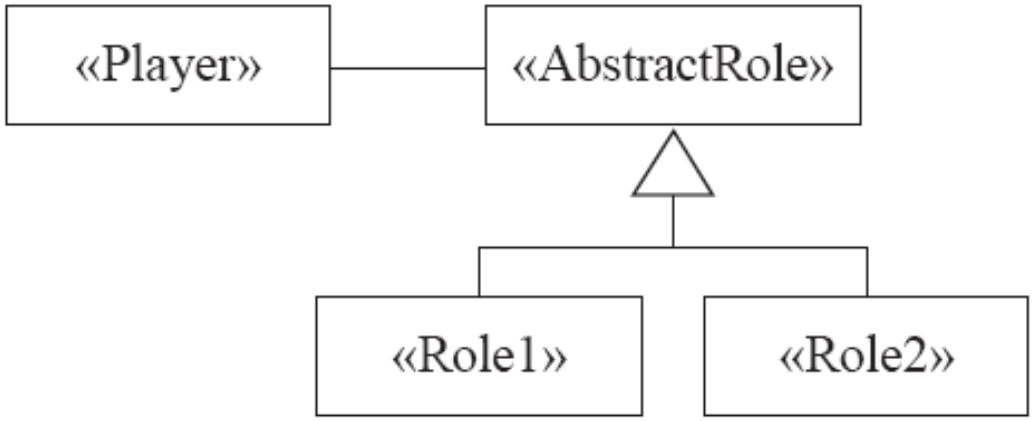
- ***Context***:
  - A *role* is a particular set of properties associated with an object in a particular context.
  - An object may *play* different roles in different contexts.
- ***Problem***:
  - How do you best model players and roles so that a player can change roles or possess multiple roles?

The Player-Role Pattern

- ***Forces***:
  - It is desirable to improve encapsulation by capturing the information associated with each separate role in a class.
  - You want to avoid multiple inheritance.
  - You cannot allow an instance to change class

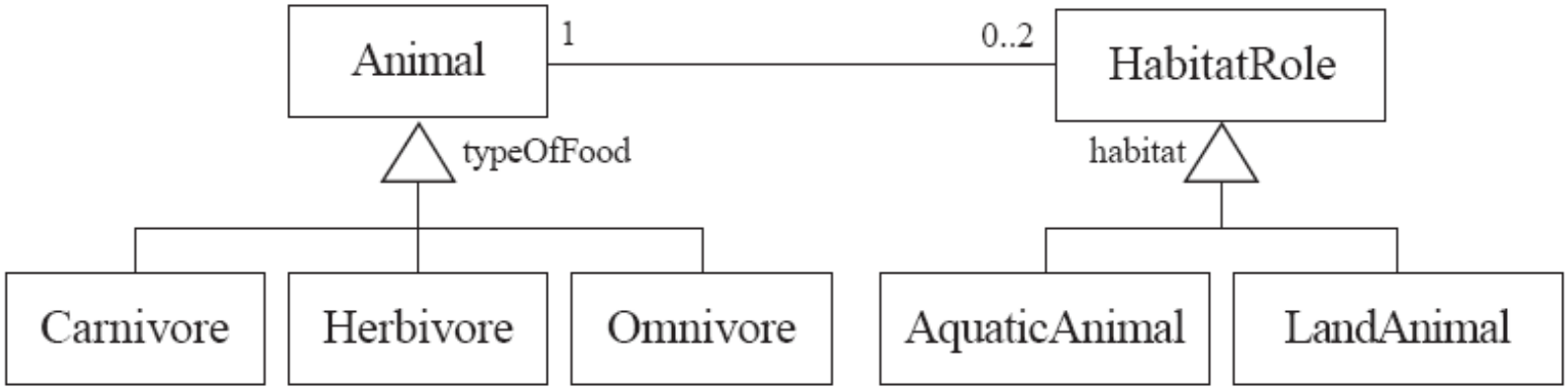
The Player-Role Pattern

- ***Solution***:



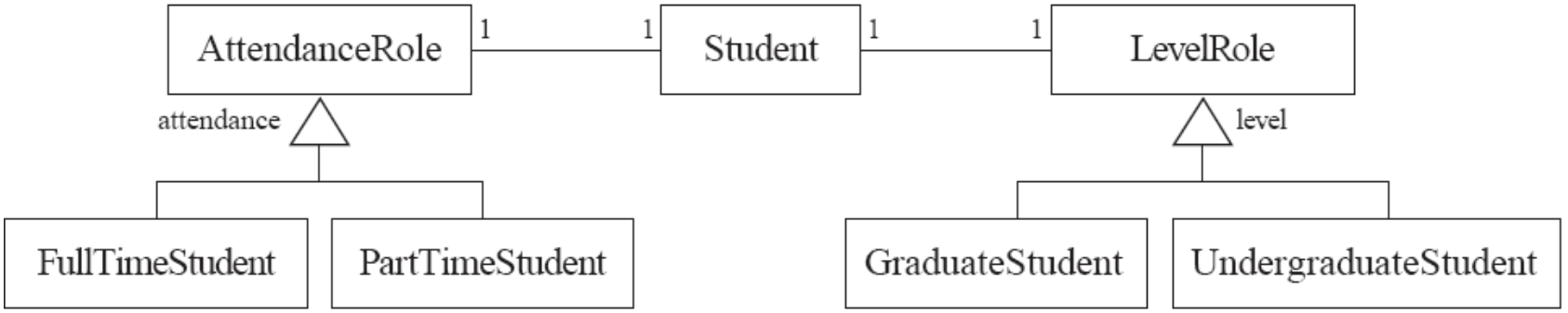
The Player-Role Pattern

- Example 1:



The Player-Role Pattern

- Example 2:



A student can have:

- No attendance role.
- FT or PT, but not both.
- No Level role.
- GS or UG, but not both.

The Player-Role Pattern

- Antipatterns:
  - Merge all the properties and behaviours into a single «Player» class and not have «Role» classes at all.
  - Create roles as subclasses of the «Player» class.

The Player-Role Pattern

- **Player**:
  - Represents the main entity (e.g., a person, object, or component).
  - The player can assume one or more roles.
- **Role**:
  - Represents a behavior or responsibility that a player can assume.
  - Multiple roles can be dynamically attached to or detached from the player.
- **Flexibility**:
  - New roles can be added without modifying the player.

The Player-Role Pattern

The following **Java** implementation demonstrates the **Player-Role Pattern**. A player can act as a **Warrior**, **Healer**, or **Mage** in a game.

```
import java.util.ArrayList;
import java.util.List;
// Interface representing a Role
interface Role {
    void performRole();
}
// Concrete Role: Warrior
class Warrior implements Role {
    @Override
    public void performRole() {
        System.out.println("Fighting as a Warrior with strength and swords.");    }}
```

The Player-Role Pattern

```
// Concrete Role: Healer
class Healer implements Role {
    @Override
    public void performRole() {
        System.out.println("Healing teammates with magical potions.");
    }
}
// Concrete Role: Mage
class Mage implements Role {
    @Override
    public void performRole() {
        System.out.println("Casting powerful spells as a Mage.");
    }
}
```

The Player-Role Pattern

```
// Player class that can assume multiple roles
class Player {
    private String name;
    private List<Role> roles;
    public Player(String name) {
        this.name = name;
        this.roles = new ArrayList<>();
    }
    // Add a role to the player
    public void addRole(Role role) {
        roles.add(role);
        System.out.println(name + " has taken on a new role: " +
role.getClass().getSimpleName());    }
//getClass() and getSimpleName() are coming from java.lang.Object
```

The Player-Role Pattern

```
// Remove a role from the player
    public void removeRole(Role role) {
        roles.remove(role);
        System.out.println(name + " has dropped the role: " +
role.getClass().getSimpleName());
    }
    // Perform all roles assigned to the player
    public void performRoles() {
        System.out.println("\n" + name + " is performing their roles:");
        if (roles.isEmpty()) {
            System.out.println("No roles assigned.");
        }
        for (Role role : roles) {
            role.performRole();        }    }  }
```

The Player-Role Pattern

```
// Main class to demonstrate the Player-Role Pattern
public class PlayerRolePatternExample {
    public static void main(String[] args) {
        // Create a player
        Player player = new Player("Arthur");

        // Create roles
        Role warrior = new Warrior();
        Role healer = new Healer();
        Role mage = new Mage();

        // Assign roles to the player
        player.addRole(warrior);
        player.addRole(healer);
```

The Player-Role Pattern

```
// Perform the roles
        player.performRoles();

        // Dynamically add another role
        player.addRole(mage);
        player.performRoles();

        // Remove a role
        player.removeRole(warrior);
        player.performRoles();
    }
}
```

The Player-Role Pattern

Output:

Arthur has taken on a new role: Warrior

Arthur has taken on a new role: Healer

Arthur is performing their roles:

Fighting as a Warrior with strength and swords.

Healing teammates with magical potions.

Arthur has taken on a new role: Mage

Arthur is performing their roles:

Fighting as a Warrior with strength and swords.

Healing teammates with magical potions.

Casting powerful spells as a Mage.

Arthur has dropped the role: Warrior

Arthur is performing their roles:

Healing teammates with magical potions.

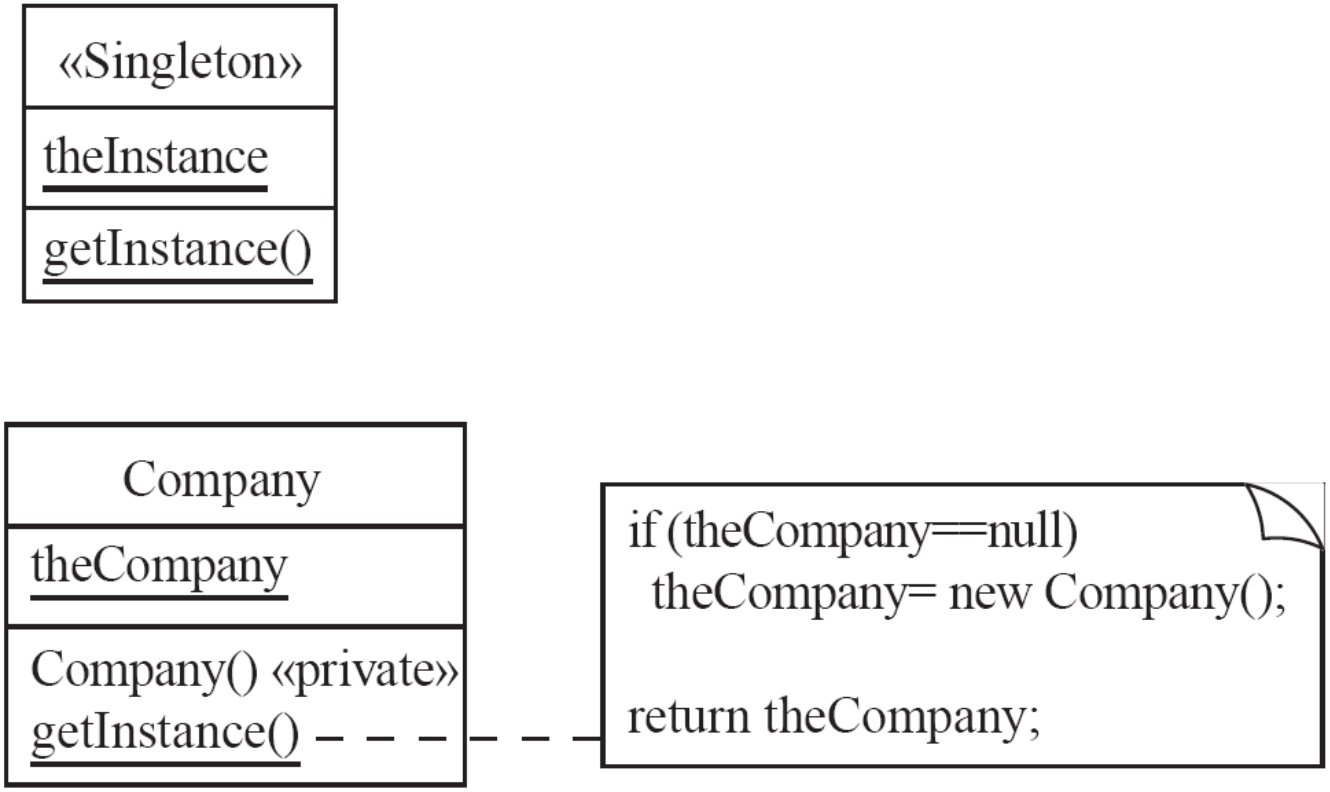
Casting powerful spells as a Mage.

The Singleton Pattern

- ***Context***:
  - It is very common to find classes for which only one instance should exist (*singleton)*
- ***Problem***:
  - How do you ensure that it is never possible to create more than one instance of a singleton class?
- ***Forces***:
  - The use of a public constructor cannot guarantee that no more than one instance will be created.
  - The singleton instance must also be accessible to all classes that require it.

The Singleton Pattern

- ***Solution:***



The Singleton Pattern

```
class Singleton {
    // Step 1: Private static instance of the same class
    private static Singleton singleInstance;

    // Step 2: Private constructor to prevent external instantiation
    private Singleton() {
        System.out.println("Singleton Instance Created");
    }
```

The Singleton Pattern

```
// Step 3: Public static method to provide the single instance
    public static Singleton getInstance() {
        if (singleInstance == null) { // Lazy initialization
            singleInstance = new Singleton();
        }
        return singleInstance;
    }

    // Sample method for demonstration
    public void showMessage() {
        System.out.println("Hello from Singleton Pattern!");
    }
}
```

The Singleton Pattern

```
public class SingletonDemo {
    public static void main(String[] args) {
        // Get the single instance of Singleton class
        Singleton instance1 = Singleton.getInstance();
        instance1.showMessage();
        // Trying to get another instance
        Singleton instance2 = Singleton.getInstance();
        // Check if both references point to the same object
        if (instance1 == instance2) {
            System.out.println("Both instances are the same!");
        } else {
            System.out.println("Instances are different!");
        }    }  }
```

The Singleton Pattern

Output:

Singleton Instance Created

Hello from Singleton Pattern!

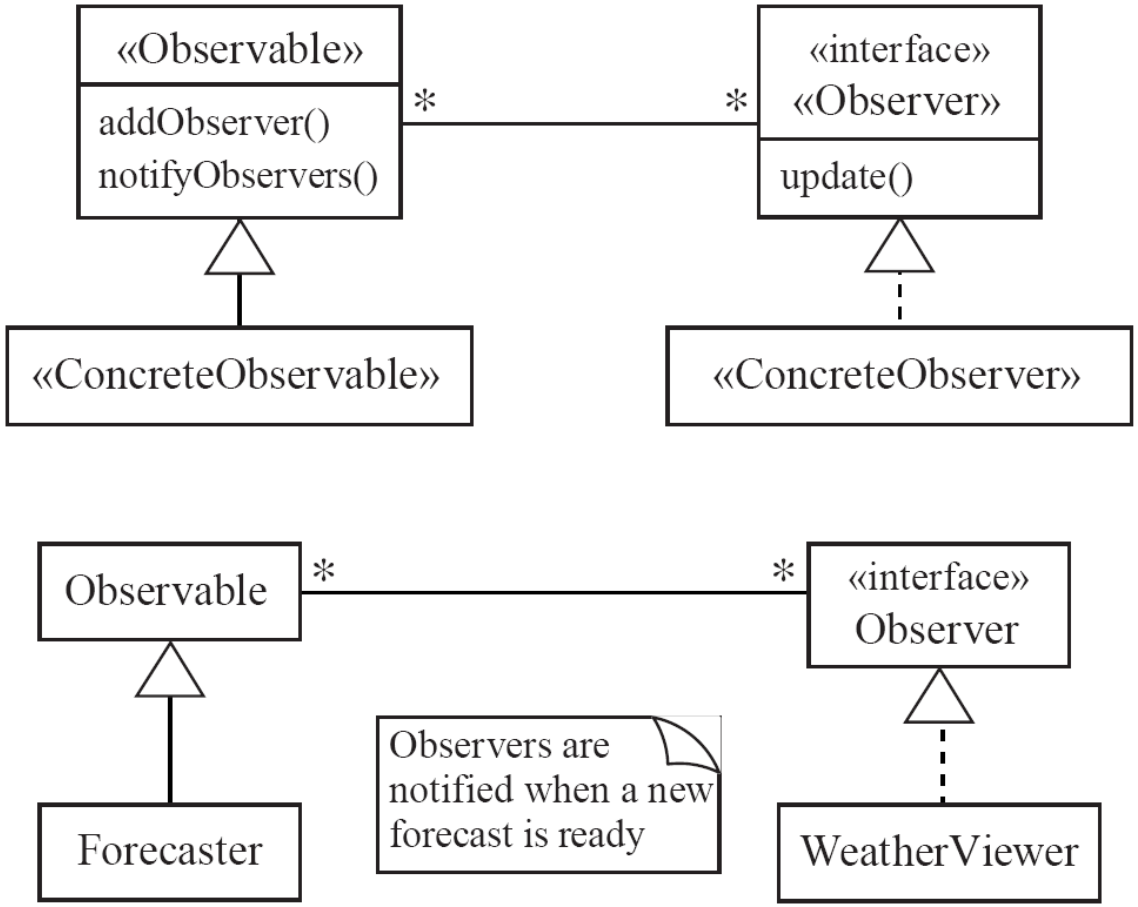
Both instances are the same!

The Observer Pattern

- ***Context***:
  - When an association is created between two classes, the code for the classes becomes inseparable.
  - If you want to reuse one class, then you also have to reuse the other.
- ***Problem***:
  - How do you reduce the interconnection between classes, especially between classes that belong to different modules or subsystems?
- ***Forces***:
  - You want to maximize the flexibility of the system to the greatest extent possible .

The Observer Pattern

- ***Solution:***



The Observer Pattern

- Antipatterns:
  - Connect an observer directly to an observable so that they both have references to each other.
  - Make the observers *subclasses* of the observable.

The Observer Pattern

This implementation simulates a weather forecasting system where:

- **Forecaster** acts as the ConcreteObservable (Subject).
- **WeatherViewer** acts as the ConcreteObserver (Observer).
- The Observer interface defines the update() method for observers to get updates when the state of the subject changes.

The Observer Pattern

```
import java.util.ArrayList;
import java.util.List;

// Observer Interface
interface Observer {
    void update(String forecast);
}

// Observable Class
abstract class Observable {
    private List<Observer> observers = new ArrayList<>();

    public void addObserver(Observer observer) {
        observers.add(observer);    }
```

The Observer Pattern

```
public void notifyObservers(String forecast) {
        for (Observer observer : observers) {
            observer.update(forecast);
        }
    }
}

// ConcreteObservable (Forecaster)
class Forecaster extends Observable {
    private String currentForecast;
```

The Observer Pattern

```
public void setForecast(String forecast) {
        this.currentForecast = forecast;
        System.out.println("Forecaster: New weather forecast: " + forecast);
        notifyObservers(forecast); // Notify all observers of the change
    }
}

// ConcreteObserver (WeatherViewer)
class WeatherViewer implements Observer {
    private String name;

    public WeatherViewer(String name) {
        this.name = name;    }
```

The Observer Pattern

```
@Override
    public void update(String forecast) {
        System.out.println(name + " received the weather update: " + forecast);
    }
}

// Main Class to Demonstrate the Observer Pattern
public class ObserverPatternExample {
    public static void main(String[] args) {
        // Create the Observable (Forecaster)
        Forecaster forecaster = new Forecaster();

        // Create Observers (WeatherViewers)
```

The Observer Pattern

```
WeatherViewer viewer1 = new WeatherViewer("Weather Channel");
        WeatherViewer viewer2 = new WeatherViewer("Mobile App");
        WeatherViewer viewer3 = new WeatherViewer("Website");

        // Add Observers to the Forecaster
        forecaster.addObserver(viewer1);
        forecaster.addObserver(viewer2);
        forecaster.addObserver(viewer3);

        // Update the forecast and notify observers
        forecaster.setForecast("Sunny with a chance of showers");
        System.out.println();
        forecaster.setForecast("Heavy rain expected tomorrow");
    }
}
```

The Observer Pattern

Output:

Forecaster: New weather forecast: Sunny with a chance of showers

Weather Channel received the weather update: Sunny with a chance of showers

Mobile App received the weather update: Sunny with a chance of showers

Website received the weather update: Sunny with a chance of showers

Forecaster: New weather forecast: Heavy rain expected tomorrow

Weather Channel received the weather update: Heavy rain expected tomorrow

Mobile App received the weather update: Heavy rain expected tomorrow

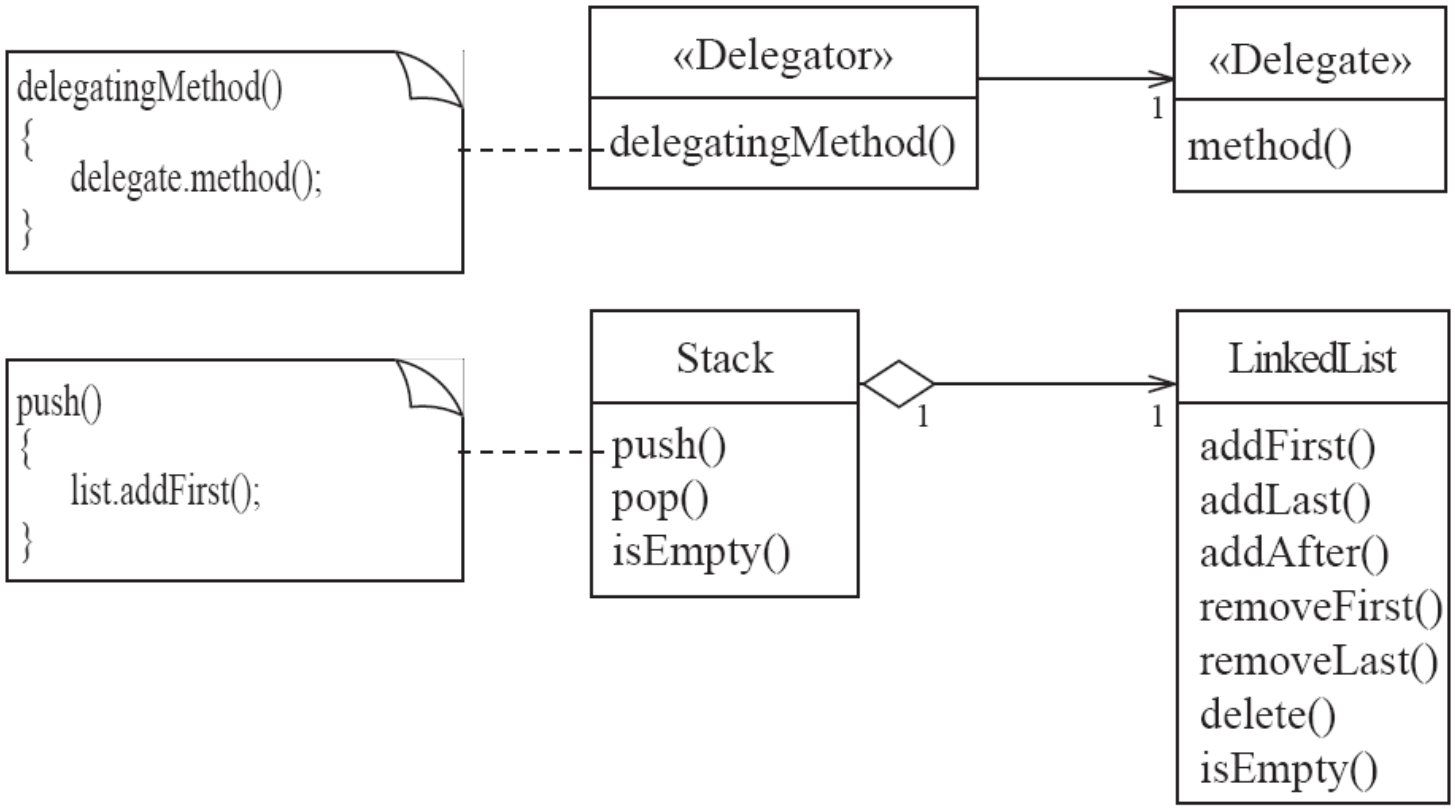
Website received the weather update: Heavy rain expected tomorrow

The Delegation Pattern

- ***Context***:
  - You are designing a method in a class
  - You realize that another class has a method which provides the required service
  - Inheritance is not appropriate
    - E.g. because the isa rule does not apply
- ***Problem***:
  - How can you most effectively make use of a method that already exists in the other class?
- ***Forces***:
  - You want to minimize development cost by reusing methods

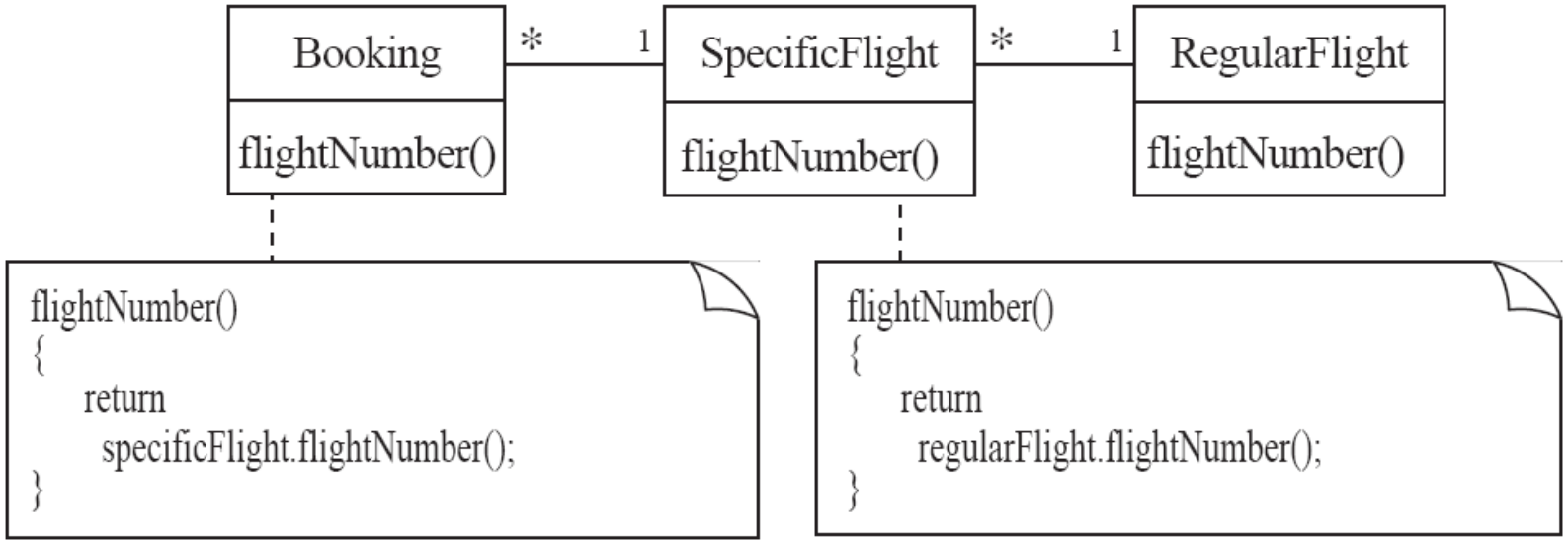
The Delegation Pattern

- ***Solution***



The Delegation Pattern

Example:



The Delegation Pattern

- Antipatterns
  - Overuse generalization and *inherit* the method that is to be reused
  - Instead of creating a *single* method in the «Delegator» that does nothing other than call a method in the «Delegate »
    - consider having many different methods in the «Delegator» call the delegate's method
  - Access non-neighboring classes

return specificFlight.regularFlight.flightNumber();

return getRegularFlight().flightNumber();

The Delegation Pattern

```
// Step 1: Define the Printer Interface
interface Printer {
    void print(String document);
}

// Step 2: Concrete Implementation of Canon Printer
class CanonPrinter implements Printer {
    @Override
    public void print(String document) {
        System.out.println("Canon Printer: Printing document - " + document);
    }
}
```

The Delegation Pattern

```
// Step 3: Concrete Implementation of Epson Printer
class EpsonPrinter implements Printer {
    @Override
    public void print(String document) {
        System.out.println("Epson Printer: Printing document - " + document);
    }
}
```

The Delegation Pattern

```
// Step 4: Printer Manager (Delegator)
class PrinterManager implements Printer {
    private Printer delegatePrinter; // Holds a reference to the delegate printer
    public PrinterManager(Printer printer) {
        this.delegatePrinter = printer;    }
    // Set or change the printer dynamically
    public void setPrinter(Printer printer) {
        this.delegatePrinter = printer;    }
    @Override
    public void print(String document) {
        System.out.println("PrinterManager: Delegating print task...");
        delegatePrinter.print(document); // Delegates the task
    }
}
```

The Delegation Pattern

```
// Step 5: Main Class to Demonstrate Delegation Pattern
public class DelegationPatternExample {
    public static void main(String[] args) {
        // Create specific printers
        Printer canonPrinter = new CanonPrinter();
        Printer epsonPrinter = new EpsonPrinter();
        // Create a PrinterManager that delegates to CanonPrinter
        PrinterManager manager = new PrinterManager(canonPrinter);
        // Print a document using the CanonPrinter
        manager.print("Report.pdf");
        System.out.println();
        // Change the delegate printer dynamically to EpsonPrinter
        manager.setPrinter(epsonPrinter);
        // Print the same document using the EpsonPrinter
        manager.print("Report.pdf");    }}
```

The Delegation Pattern

Output:

PrinterManager: Delegating print task...

Canon Printer: Printing document - Report.pdf

PrinterManager: Delegating print task...

Epson Printer: Printing document - Report.pdf

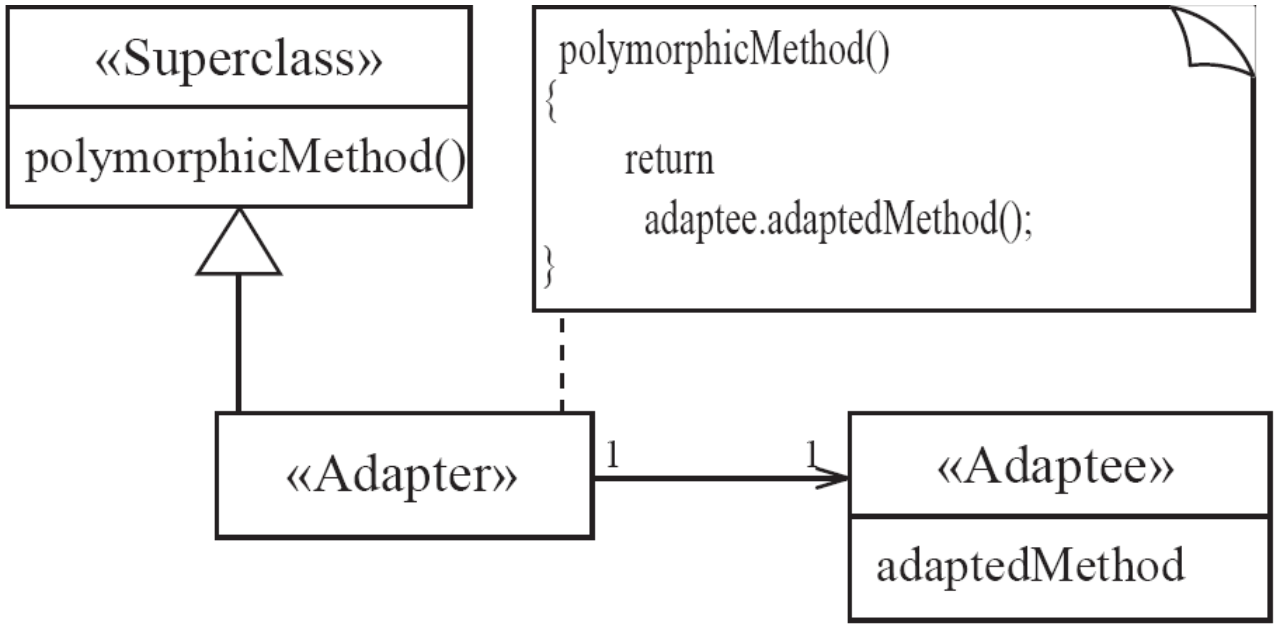
The Adapter Pattern

- ***Context***:
  - You are building an inheritance hierarchy and want to incorporate it into an existing class.
  - The reused class is also often already part of its own inheritance hierarchy.
- ***Problem***:
  - How to obtain the power of polymorphism when reusing a class whose methods
    - have the same function
    - but *not* the same signature

    as the other methods in the hierarchy?
- ***Forces***:
  - You do not have access to multiple inheritance or you do not want to use it.

The Adapter Pattern

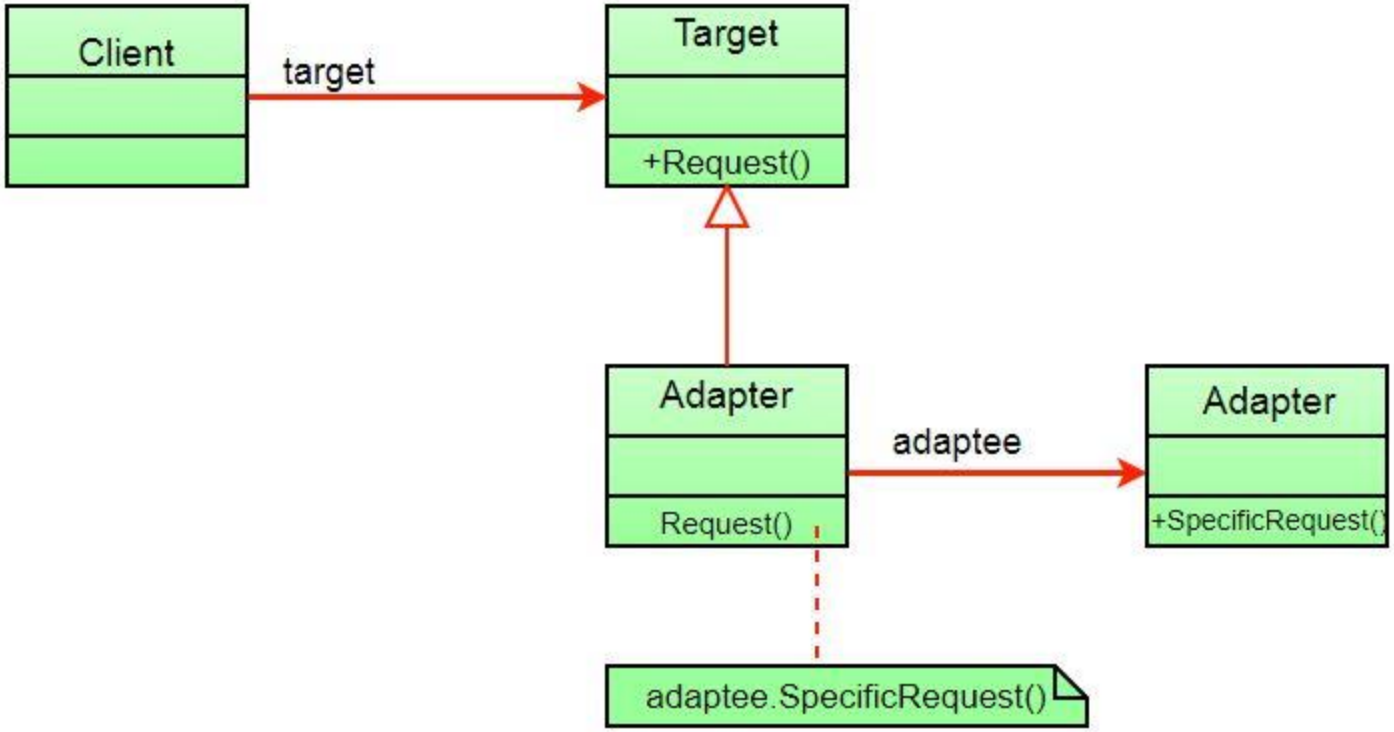
- ***Solution:***



The Adapter Pattern

- This pattern is easy to understand as the real world is full of adapters. For example consider a USB to Ethernet adapter. We need this when we have an Ethernet interface on one end and USB on the other. Since they are incompatible with each other. we use an adapter that converts one to other. This example is pretty analogous to Object Oriented Adapters. In design, adapters are used when we have a class (Client) expecting some type of object and we have an object (Adaptee) offering the same features but exposing a different interface.
- To use an adapter:
  - The client makes a request to the adapter by calling a method on it using the target interface.
  - The adapter translates that request on the adaptee using the adaptee interface.
  - Client receive the results of the call and is unaware of adapter's presence.

The Adapter Pattern



The Adapter Pattern

- Suppose you have a Bird class with fly() , and makeSound() methods. And also a ToyDuck class with squeak() method. Let's assume that you are short on ToyDuck objects and you would like to use Bird objects in their place. Birds have some similar functionality but implement a different interface, so we can't use them directly. So we will use adapter pattern. Here our client would be ToyDuck and adaptee would be Bird.

The Adapter Pattern

```
// Java implementation of Adapter pattern


interface Bird
{
    // birds implement Bird interface that allows
    // them to fly and make sounds adaptee interface
    public void fly();
    public void makeSound();
}
```

The Adapter Pattern

```
class Sparrow implements Bird
{
    // a concrete implementation of bird
    public void fly()
   {
        System.out.println("Flying");
    }
    public void makeSound()
    {
        System.out.println("Chirp Chirp");
    }
}
```

The Adapter Pattern

```
interface ToyDuck
{
    // target interface
    // toyducks dont fly they just make
    // squeaking sound
    public void squeak();
}
class PlasticToyDuck implements ToyDuck
{
    public void squeak()
    {
        System.out.println("Squeak");
    }
}
```

The Adapter Pattern

```
class BirdAdapter implements ToyDuck
{
    // You need to implement the interface your
    // client expects to use.
    Bird bird;
    public BirdAdapter(Bird bird)
    {
        // we need reference to the object we
        // are adapting
        this.bird = bird;
    }
```

The Adapter Pattern

```
public void squeak()
    {
        // translate the methods appropriately
        bird.makeSound();
    }
}
class Main
{
    public static void main(String args[])
    {
        Sparrow sparrow = new Sparrow();
        ToyDuck toyDuck = new PlasticToyDuck();
```

The Adapter Pattern

```
// Wrap a bird in a birdAdapter so that it
        // behaves like toy duck
        ToyDuck birdAdapter = new BirdAdapter(sparrow);
        System.out.println("Sparrow...");
        sparrow.fly();
        sparrow.makeSound();
        System.out.println("ToyDuck...");
        toyDuck.squeak();
        // toy duck behaving like a bird
        System.out.println("BirdAdapter...");
        birdAdapter.squeak();
    }
}
```

The Adapter Pattern

- Output:

Sparrow...

Flying

Chirp Chirp

ToyDuck...

Squeak

BirdAdapter...

Chirp Chirp

The Façade Pattern

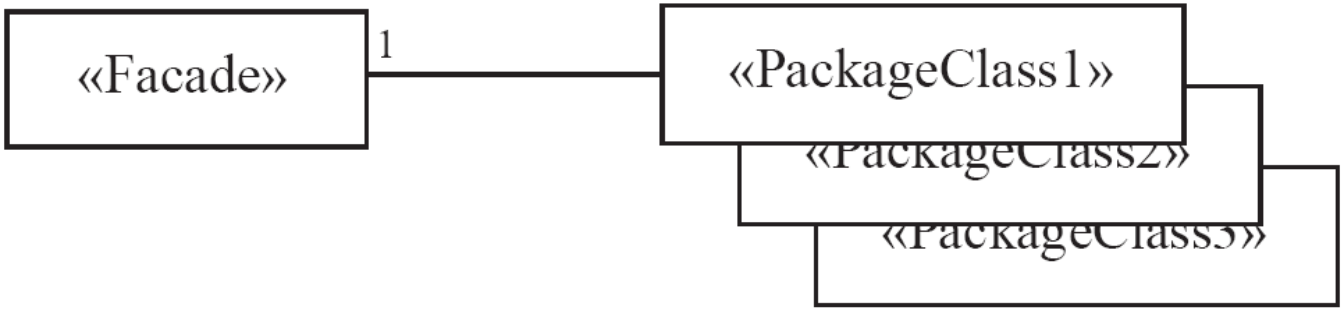
- ***Context***:
  - Often, an application contains several complex packages.
  - A programmer working with such packages has to manipulate many different classes
- ***Problem***:
  - How do you simplify the view that programmers have of a complex package?
- ***Forces***:
  - It is hard for a programmer to understand and use an entire subsystem
  - If several different application classes call methods of the complex package, then any modifications made to the package will necessitate a complete review of all these classes.

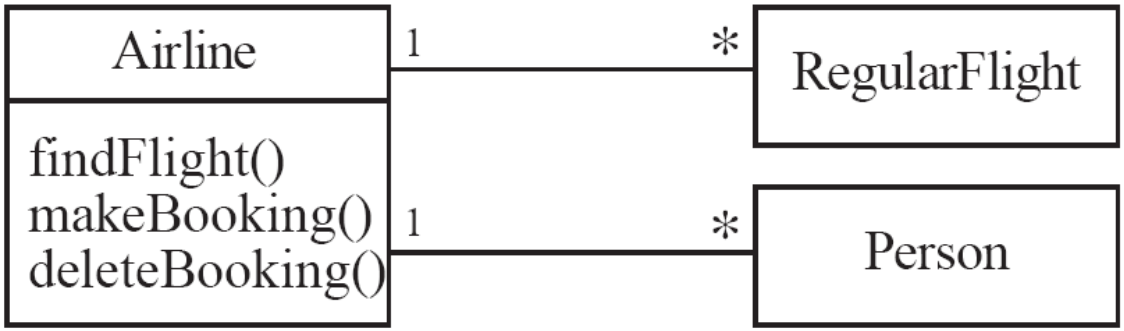
The Façade Pattern

- Facade 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.
- The **facade pattern** (also spelled *façade*) is a software-design pattern commonly used in object-oriented programming. Analogous to a facade in architecture, a facade is an object that serves as a front-facing interface masking more complex underlying or structural code.

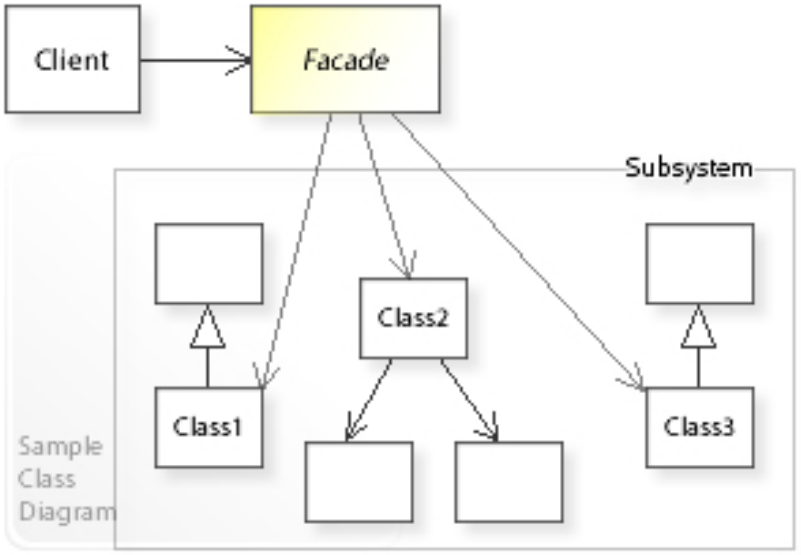
The Façade Pattern

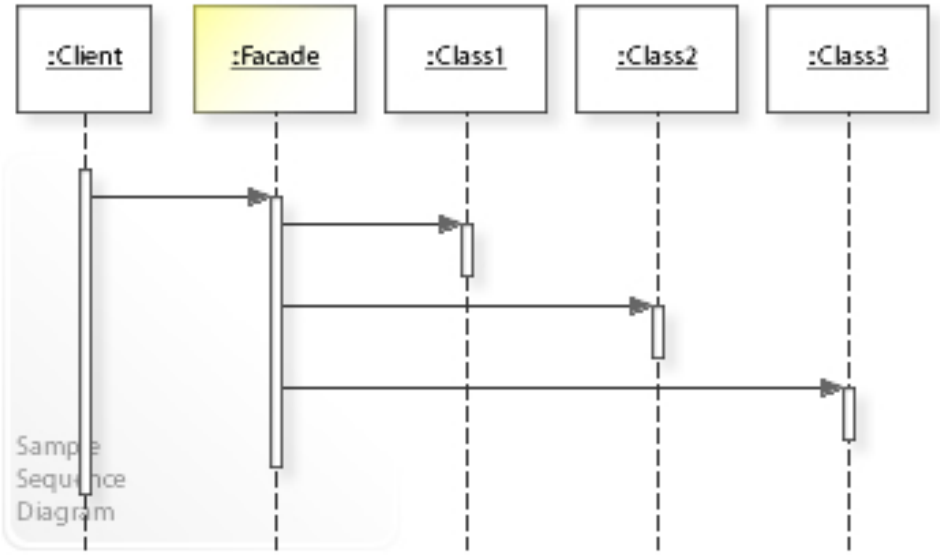
- ***Solution:***





The Façade Pattern





The Immutable Pattern

- ***Context***:
  - An immutable object is an object that has a state that never changes after creation
- ***Problem***:
  - How do you create a class whose instances are immutable?
- ***Forces***:
  - There must be no loopholes that would allow ‘illegal’ modification of an immutable object
- ***Solution***:
  - Ensure that the constructor of the immutable class is the *only* place where the values of instance variables are set or modified.
  - Instance methods which access properties must not have side effects.
  - If a method that would otherwise modify an instance variable is required, then it has to return a *new* instance of the class.

How to create Immutable class in Java?

- Following are the requirements:
  - The class must be declared as final (So that child classes can't be created)
  - Data members in the class must be declared as final (So that we can't change the value of it after object creation)
  - A parameterized constructor
  - Getter method for all the variables in it
  - No setters (To not have the option to change the value of the instance variable)

Example to create Immutable class

```
// An immutable class
public final class Student
{
    final String name;
    final int regNo;

    public Student(String name, int regNo)
    {
        this.name = name;
        this.regNo = regNo;
    }
    public String getName()
    {
        return name;
    }
    public int getRegNo()
    {
        return regNo;
    }
}
```

Example to create Immutable class

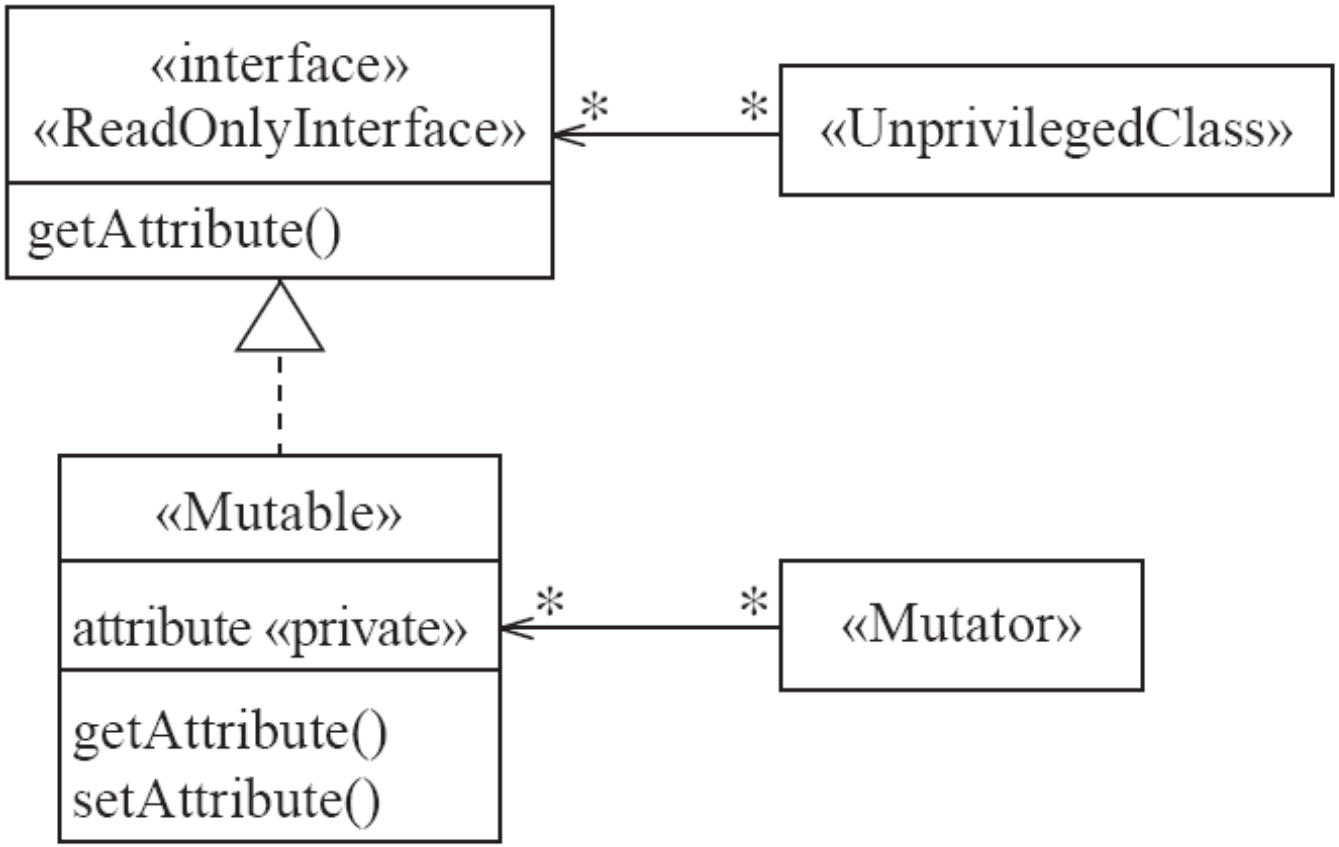
```
// Driver class
class Test
{
    public static void main(String args[])
    {
        Student s = new Student("ABC", 101);
        System.out.println(s.getName());
        System.out.println(s.getRegNo());

        // Uncommenting below line causes error
        // s.regNo = 102;
    }
}
```

The Read-only Interface Pattern

- ***Context***:
  - You sometimes want certain privileged classes to be able to modify attributes of objects that are otherwise immutable
- ***Problem***:
  - How do you create a situation where some classes see a class as read-only whereas others are able to make modifications?
- ***Forces***:
  - Restricting access by using the **public**, **protected** and **private** keywords is not adequately selective.
  - Making access **public** makes it public for both reading and writing

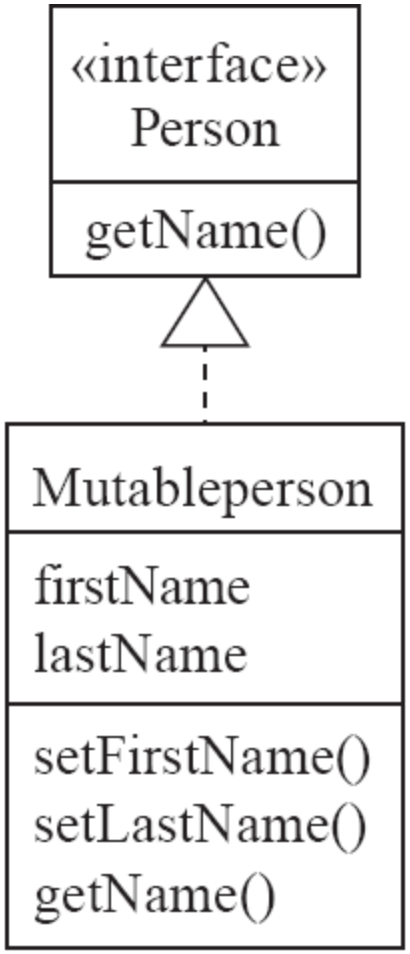
The Read-only Interface Pattern

- ***Solution:***



The Read-only Interface Pattern

- Example:



The Read-only Interface Pattern

- Antipatterns:
  - Make the read-only class a *subclass* of the «Mutable» class
  - Override all methods that modify properties
    - such that they throw an exception

The Read-only Interface Pattern

```
// ReadOnlyInterface.java
interface ReadOnlyInterface {
    int getAttribute();
}

// Mutable.java
class Mutable implements ReadOnlyInterface {
    private int attribute; // Private attribute

    public Mutable(int attribute) {
        this.attribute = attribute;
    }
```

The Read-only Interface Pattern

```
 // Getter method for the attribute
    @Override
    public int getAttribute() {
        return attribute;
    }

    // Setter method for the attribute
    public void setAttribute(int attribute) {
        this.attribute = attribute;
    }
}
```

The Read-only Interface Pattern

```
// UnprivilegedClass.java
class UnprivilegedClass {
    private ReadOnlyInterface readOnly;

    public UnprivilegedClass(ReadOnlyInterface readOnly) {
        this.readOnly = readOnly;
    }

    public void displayAttribute() {
        System.out.println("UnprivilegedClass accessing attribute: " +
readOnly.getAttribute());
    }
}
```

The Read-only Interface Pattern

```
// Mutator.java
class Mutator {
    private Mutable mutable;

    public Mutator(Mutable mutable) {
        this.mutable = mutable;
    }

    public void modifyAttribute(int newValue) {
        System.out.println("Mutator changing attribute to: " +
newValue);
        mutable.setAttribute(newValue);
    }
}
```

The Read-only Interface Pattern

```
// Main.java
public class Main {
    public static void main(String[] args) {
        // Creating a Mutable instance
        Mutable mutable = new Mutable(42);

        // UnprivilegedClass only has read access
        UnprivilegedClass unprivileged = new
UnprivilegedClass(mutable);
        unprivileged.displayAttribute();

        // Mutator has write access
        Mutator mutator = new Mutator(mutable);
        mutator.modifyAttribute(84);
```

The Read-only Interface Pattern

```
 // Verify changes
        unprivileged.displayAttribute();
    }
}
```

The Read-only Interface Pattern

Output:

UnprivilegedClass accessing attribute: 42

Mutator changing attribute to: 84

UnprivilegedClass accessing attribute: 84

The Proxy Pattern

- ***Context***:
  - Often, it is time-consuming and complicated to create instances of a class (*heavyweight* classes).
  - There is a time delay and a complex mechanism involved in creating the object in memory
- ***Problem***:
  - How to reduce the need to create instances of a heavyweight class?
- ***Forces***:
  - We want all the objects in a domain model to be available for programs to use when they execute a system's various responsibilities.
  - It is also important for many objects to persist from to run of the same program

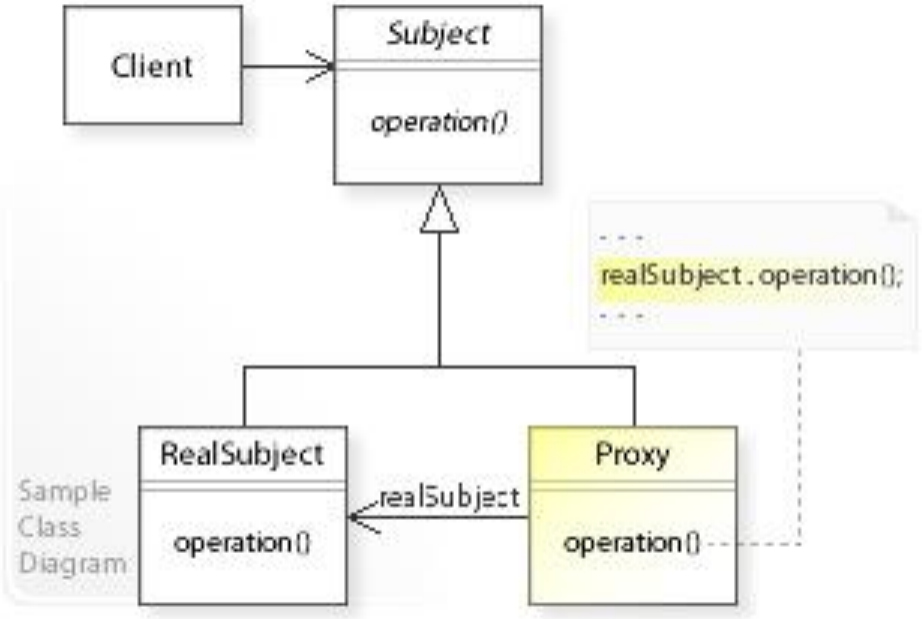
The Proxy Pattern

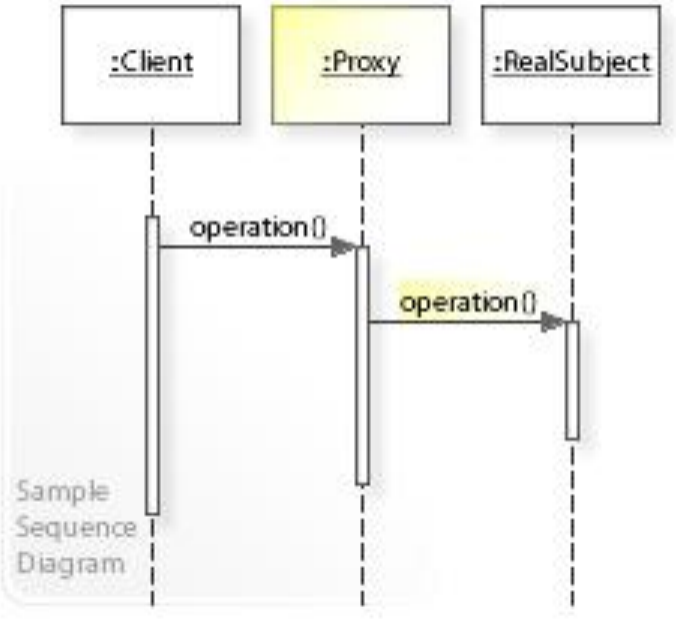
- A *proxy*, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate. In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide additional logic. In the proxy, extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked. For the client, usage of a proxy object is similar to using the real object, because both implement the same interface.

What problems can the Proxy design pattern solve?

- The access to an object should be controlled.
- Additional functionality should be provided when accessing an object.

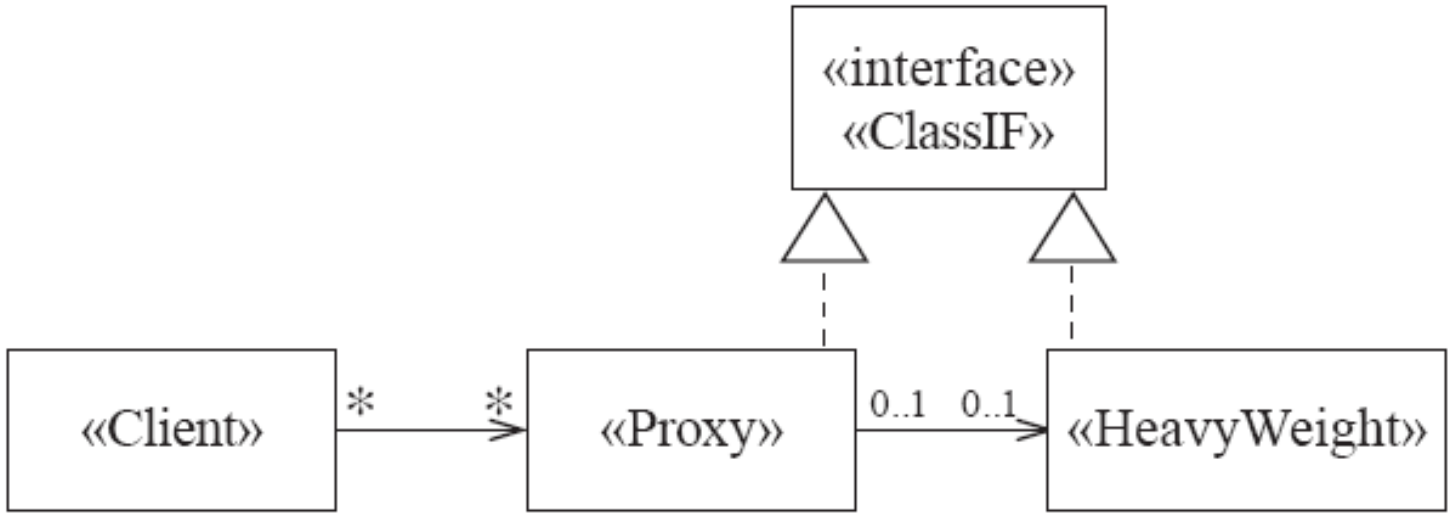
The Proxy Pattern





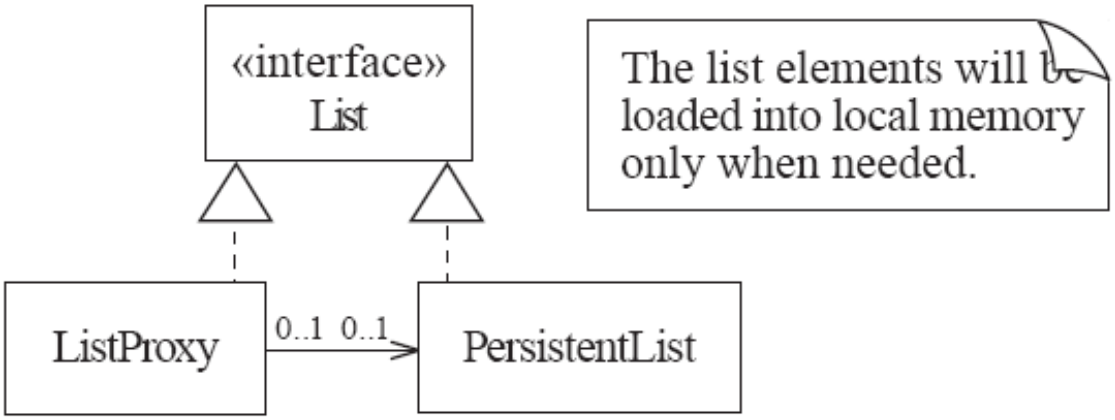
The Proxy Pattern

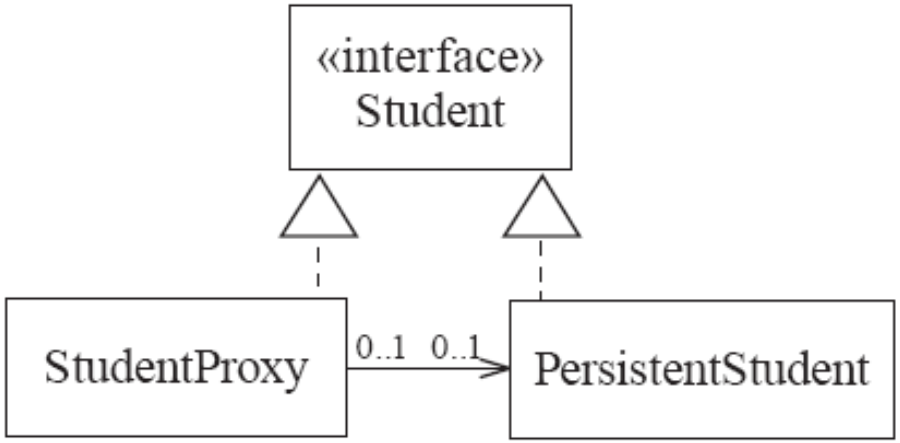
- ***Solution:***



The Proxy Pattern

- Examples:





The Proxy Pattern

```
// Student.java (Interface)
public interface Student {
    String getDetails();
    void saveDetails(String details);
}

// PersistentStudent.java (Real Subject)
class PersistentStudent implements Student {
    private String details;

    public PersistentStudent(String initialDetails) {
        this.details = initialDetails;
    }
```

The Proxy Pattern

```
    @Override
    public String getDetails() {
        System.out.println("Fetching details from PersistentStudent.");
        return details;
    }

    @Override
    public void saveDetails(String details) {
        System.out.println("Saving details to PersistentStudent.");
        this.details = details;
    }
}
```

The Proxy Pattern

```
// StudentProxy.java (Proxy)
class StudentProxy implements Student {
    private PersistentStudent persistentStudent;

    public StudentProxy(String initialDetails) {
        // Lazy initialization of the real object
        this.persistentStudent = new PersistentStudent(initialDetails);
    }
    @Override
    public String getDetails() {
        System.out.println("Proxy delegating 'getDetails' to PersistentStudent.");
        return persistentStudent.getDetails();
    }
```

The Proxy Pattern

```
@Override
    public void saveDetails(String details) {
        System.out.println("Proxy delegating 'saveDetails' to PersistentStudent.");
        persistentStudent.saveDetails(details);
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        // Create a proxy instance
        Student studentProxy = new StudentProxy("John Doe - Initial Details");
```

The Proxy Pattern

```
 // Access methods through the proxy
        System.out.println("Student Details: " + studentProxy.getDetails());

        // Save new details through the proxy
        studentProxy.saveDetails("John Doe - Updated Details");
        System.out.println("Student Details: " + studentProxy.getDetails());
    }
}
```

The Proxy Pattern

Output:

Proxy delegating 'getDetails' to PersistentStudent.

Fetching details from PersistentStudent.

Student Details: John Doe - Initial Details

Proxy delegating 'saveDetails' to PersistentStudent.

Saving details to PersistentStudent.

Proxy delegating 'getDetails' to PersistentStudent.

Fetching details from PersistentStudent.

Student Details: John Doe - Updated Details

The Factory Pattern

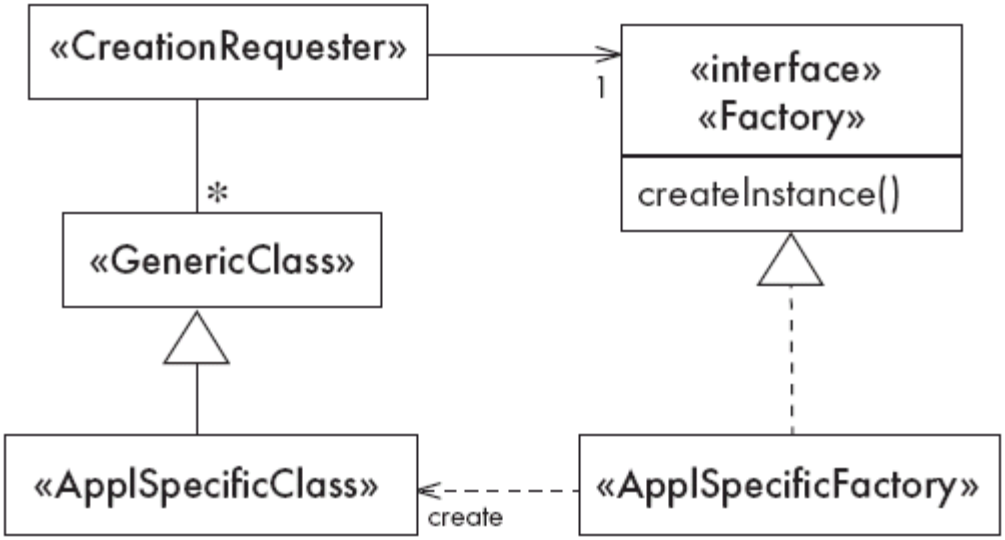
- ***Context***:
  - A reusable framework needs to create objects; however the class of the created objects depends on the application.
- ***Problem***:
  - How do you enable a programmer to add new application-specific class into a system built on such a framework?
- ***Forces***:
  - We want to have the framework create and work with application-specific classes that the framework does not yet know about.
- ***Solution***:
  - The framework delegates the creation of application-specific classes to a specialized class, the Factory.
  - The Factory is a generic interface defined in the framework.
  - The factory interface declares a method whose purpose is to create some subclass of a generic class.

The Factory Pattern

- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

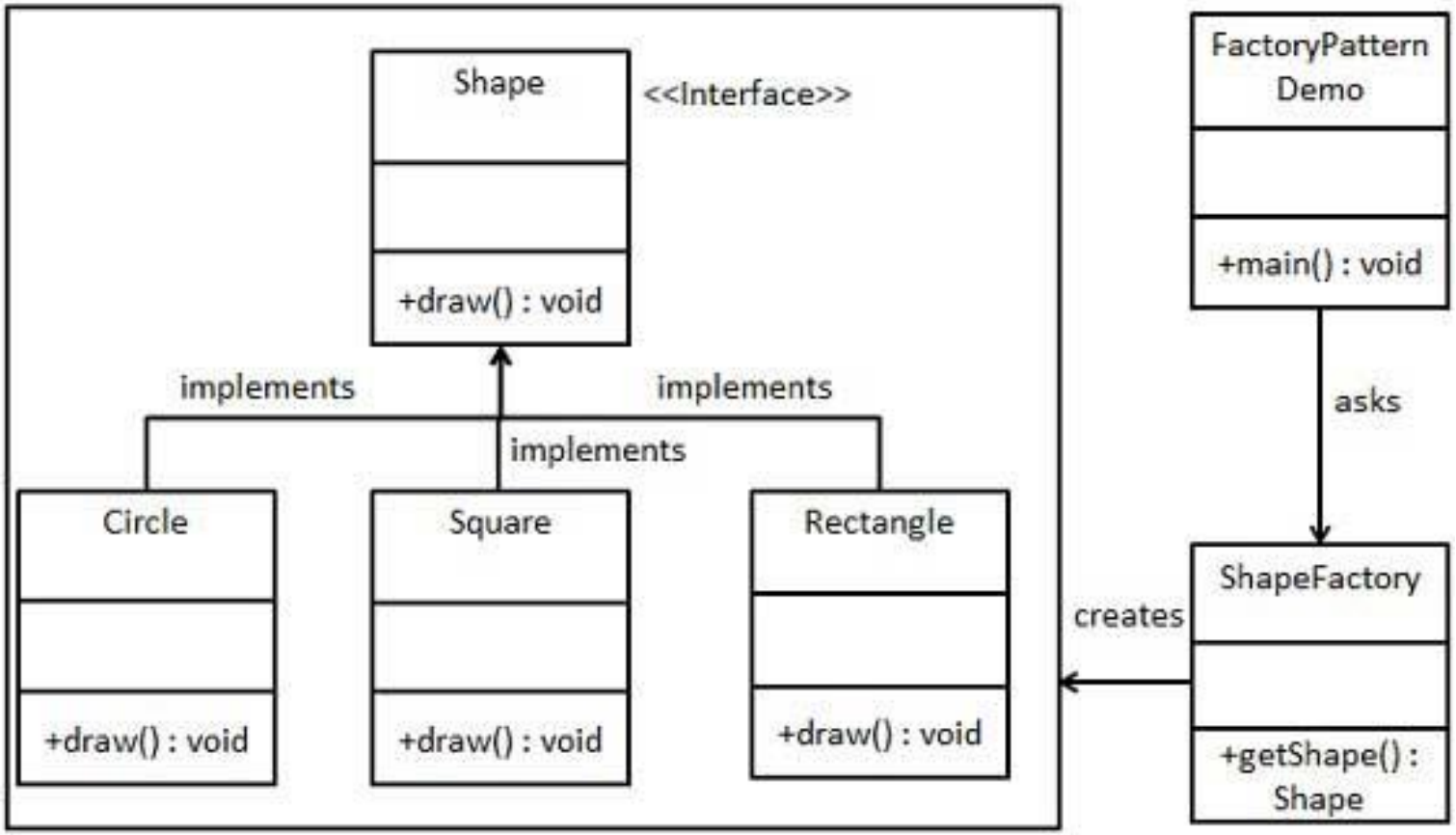
The Factory Pattern

- Solution



The Factory Pattern

- Example



The Factory Pattern

```
// Step 1: Create a common interface
interface Shape {
    void draw();
}

// Step 2: Implement the interface in concrete classes
class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}
```

The Factory Pattern

```
class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle");
    }
}

class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Square");
    }
}
```

The Factory Pattern

```
// Step 3: Create a Factory to generate objects of the concrete classes
class ShapeFactory {
   // Use a method to get objects of different types
   public Shape getShape(String shapeType) {
      if (shapeType == null) {
         return null;
      }
      switch (shapeType.toLowerCase()) {
         case "circle":
            return new Circle();
         case "rectangle":
            return new Rectangle();
```

The Factory Pattern

```
case "square":
                return new Square();
            default:
                return null;
        }
    }
}

// Step 4: Test the Factory Pattern
public class FactoryPatternExample {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();
```

The Factory Pattern

```
// Get a Circle object and call its draw method
        Shape shape1 = shapeFactory.getShape("circle");
        if (shape1 != null) shape1.draw();

        // Get a Rectangle object and call its draw method
        Shape shape2 = shapeFactory.getShape("rectangle");
        if (shape2 != null) shape2.draw();

        // Get a Square object and call its draw method
        Shape shape3 = shapeFactory.getShape("square");
        if (shape3 != null) shape3.draw();
```

The Factory Pattern

```
// Try an invalid shape type
        Shape shape4 = shapeFactory.getShape("triangle");
        if (shape4 == null) {
            System.out.println("Invalid shape type");
        }
    }
}
```

The Factory Pattern

Output:

Drawing a Circle

Drawing a Rectangle

Drawing a Square

Invalid shape type

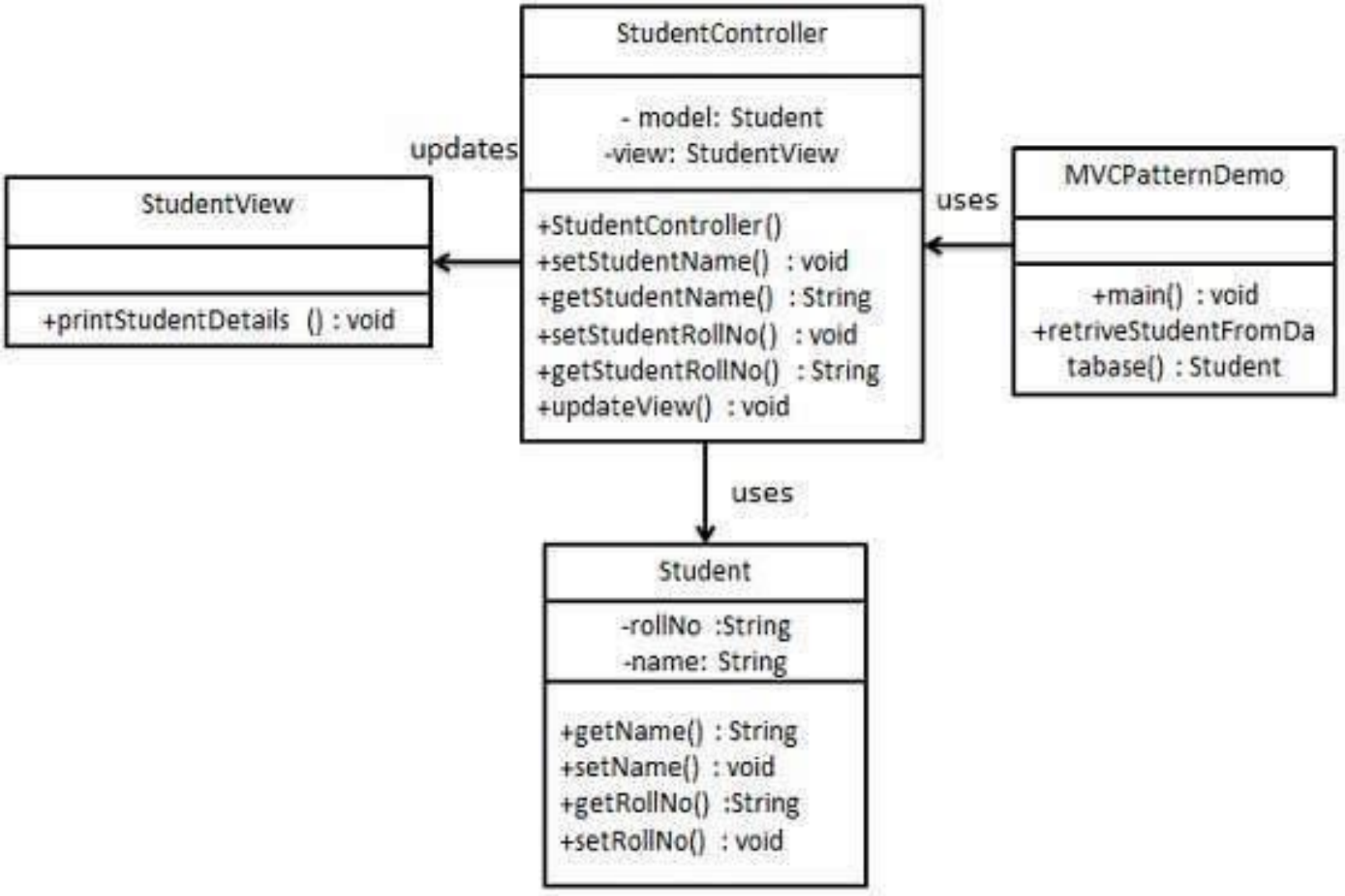
Model-View-Controller Pattern

- MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.
  - **Model** - Model represents an object. It can also have logic to update controller if its data changes.
  - **View** - View represents the visualization of the data that model contains.
  - **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

Model-View-Controller Pattern

- Implementation
  - We are going to create a *Student* object acting as a model.*StudentView* will be a view class which can print student details on console and *StudentController* is the controller class responsible to store data in *Student* object and update view *StudentView* accordingly.
  - *MVCPatternDemo*, our demo class, will use *StudentController* to demonstrate use of MVC pattern.

Model-View-Controller Pattern



Model-View-Controller Pattern

- Step 1
  - Create Model.
  - *Student.java*

```
public class Student
{
        private String rollNo;
        private String name;
        public String getRollNo()
        {
                return rollNo;
        }
```

Model-View-Controller Pattern

```
        public void setRollNo(String rollNo)
        {
                this.rollNo = rollNo;
        }
        public String getName()
        {
                return name;
        }
        public void setName(String name)
        {
                this.name = name;
        }
}
```

Model-View-Controller Pattern

- Step 2
  - Create View.
  - *StudentView.java*

```
public class StudentView
{
        public void printStudentDetails(String studentName, String
        studentRollNo)
        {
                System.out.println("Student: ");
                System.out.println("Name: " + studentName);
                System.out.println("Roll No: " + studentRollNo);
        }
}
```

Model-View-Controller Pattern

- Step 3
  - Create Controller.
  - *StudentController.java*

```
public class StudentController
{
        private Student model;
        private StudentView view;
        public StudentController(Student model, StudentView view)
        {
                this.model = model;
                this.view = view;
        }
```

Model-View-Controller Pattern

```
public void setStudentName(String name)
{
        model.setName(name);
}
public String getStudentName()
{
        return model.getName();
}
public void setStudentRollNo(String rollNo)
{
        model.setRollNo(rollNo);
}
```

Model-View-Controller Pattern

```
        public String getStudentRollNo()
        {
                return model.getRollNo();
        }
        public void updateView()
        {
                view.printStudentDetails(model.getName(),
                 model.getRollNo());
        }
}
```

Model-View-Controller Pattern

- Step 4
  - Use the *StudentController* methods to demonstrate MVC design pattern usage.
  - *MVCPatternDemo.java*

```
public class MVCPatternDemo
{
        public static void main(String[] args)
        {
                //fetch student record based on his roll no from the
                //database
                Student model = retriveStudentFromDatabase();
                //Create a view : to write student details on console
                StudentView view = new StudentView();
```

Model-View-Controller Pattern

```
        StudentController controller = new StudentController(model, view);
        controller.updateView(); //update model data
        controller.setStudentName("John");
        controller.updateView();
    }
        private static Student retriveStudentFromDatabase()
        {
          Student student = new Student();
          student.setName("Robert");
          student.setRollNo("10");
          return student;
        }
}
```

Model-View-Controller Pattern

- Step 5
  - Verify the output.

Student:

Name: Robert

Roll No: 10

Student:

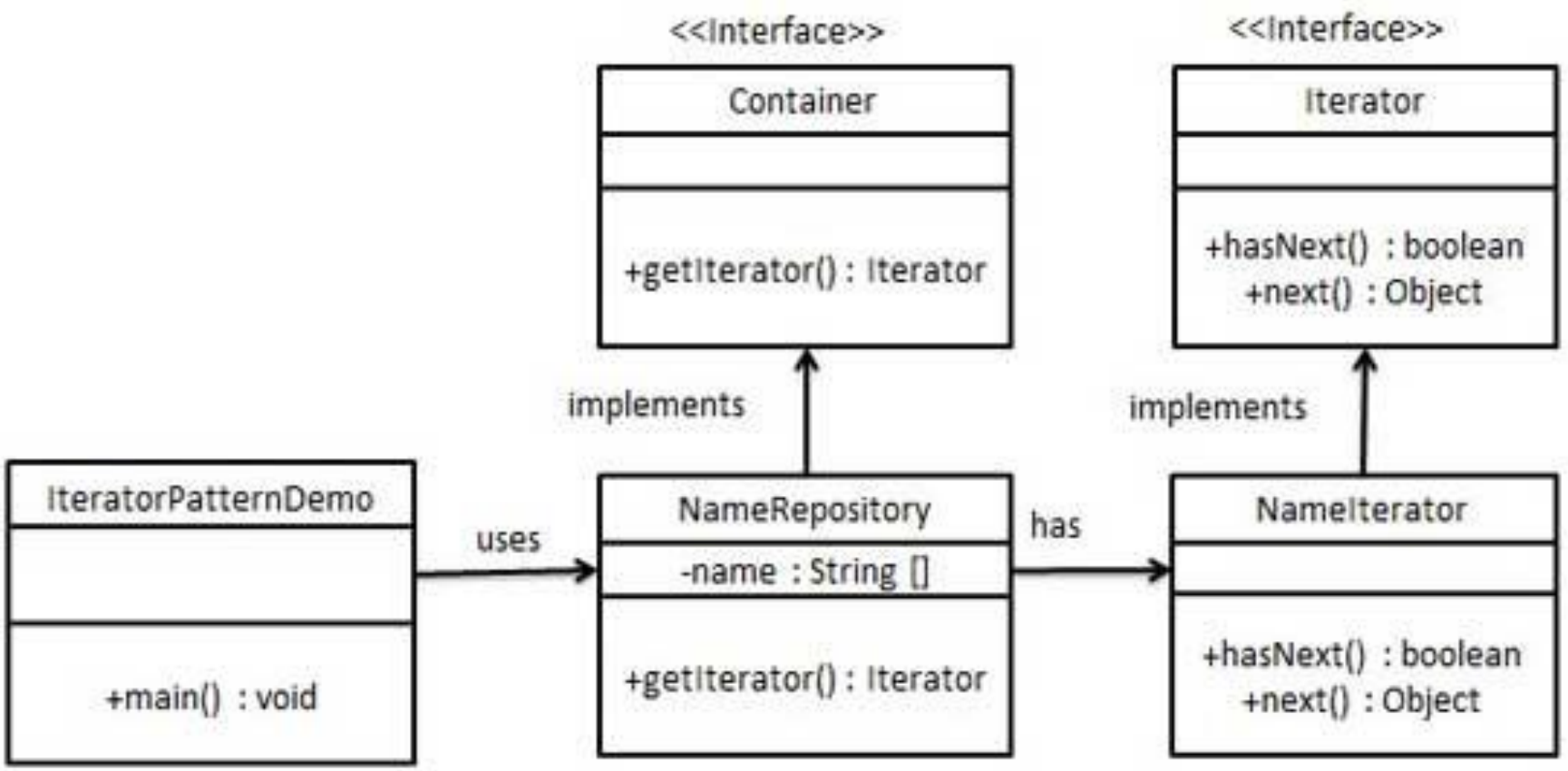
Name: John

Roll No: 10

Iterator pattern

- Iterator pattern is very commonly used design pattern in Java and .Net programming environment. This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.
- Iterator pattern falls under behavioral pattern category.
- Implementation
  - We're going to create a *Iterator* interface which narrates navigation method and a *Container* interface which returns the iterator. Concrete classes implementing the *Container* interface will be responsible to implement *Iterator* interface and use it
  - *IteratorPatternDemo*, our demo class will use *NamesRepository*, a concrete class implementation to print a *Names* stored as a collection in *NamesRepository*.

Iterator pattern



Iterator pattern

- Step 1
  - Create interfaces.
  - *Iterator.java*

```
public interface Iterator
{
        public boolean hasNext();
        public Object next();
}
```

  - *Container.java*

```
public interface Container
{
        public Iterator getIterator();
}
```

Iterator pattern

- Step 2
  - Create concrete class implementing the *Container* interface. This class has inner class *NameIterator* implementing the *Iterator* interface.
  - *NameRepository.java*

```
public class NameRepository implements Container
{
        public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};
        //Override
        public Iterator getIterator()
        {
                return new NameIterator();
        }
```

Iterator pattern

```
private class NameIterator implements Iterator
{
        int index;
        //Override
        public boolean hasNext()
        {
                if(index < names.length)
                {
                        return true;
                }
                return false;
        }
```

Iterator pattern

```
        //Override
        public Object next()
        {
                if(this.hasNext())
                {
                        return names[index++];
                }
                return null;
        }
    }
}
```

Iterator pattern

- Step 3
  - Use the *NameRepository* to get iterator and print names.
  - *IteratorPatternDemo.java*

```
public class IteratorPatternDemo
{
        public static void main(String[] args)
        {
                NameRepository namesRepository = new
NameRepository();
                for(Iterator iter = namesRepository.getIterator();
iter.hasNext();)
                {
                        String name = (String)iter.next();
                        System.out.println("Name : " + name);
}}}
```

Iterator pattern

- Step 4
  - Verify the output.

Name : Robert

Name : John

Name : Julie

Name : Lora

Difficulties and Risks When Creating Design Patterns

- **Patterns are not a panacea:**
  - Whenever you see an indication that a pattern should be applied, you might be tempted to blindly apply the pattern.
  - This can lead to unwise design decisions .
- *Resolution:*
  - *Always understand in depth the forces that need to be balanced, and when other patterns better balance the forces.*
  - *Make sure you justify each design decision carefully.*

Difficulties and Risks When Creating Class Diagrams

- **Developing patterns is hard**
  - Writing a good pattern takes considerable work.
  - A poor pattern can be hard to apply correctly
- *Resolution:*
  - *Do not write patterns for others to use until you have considerable experience both in software design and in the use of patterns.*
  - *Take an in-depth course on patterns.*
  - *Iteratively refine your patterns, and have them peer reviewed at each iteration.*

Thank you

Questions?