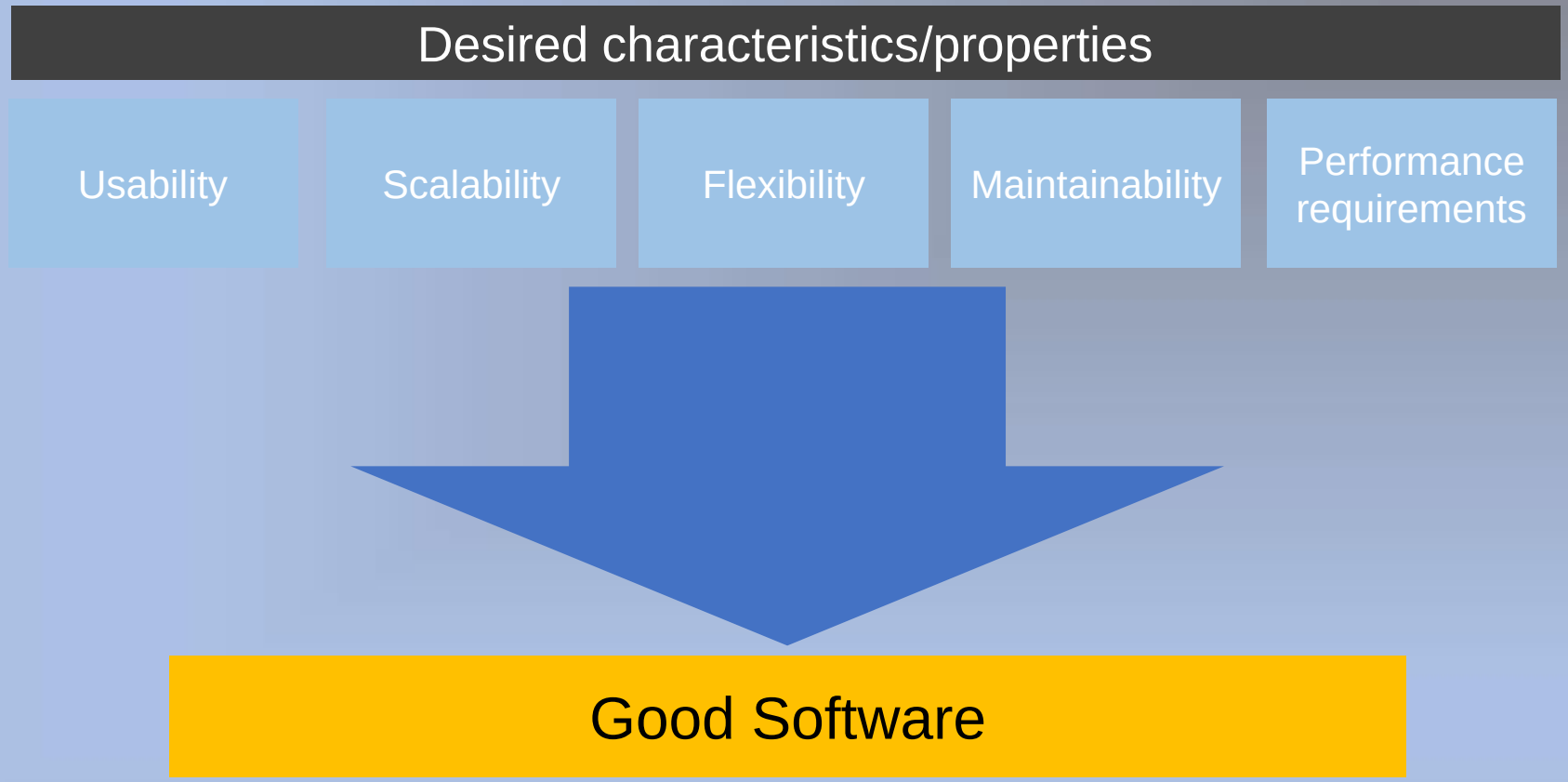
Design Principles

Sujata Batra

Overview: Good Software Features



Overview: Bad Class Design

- Badly designed classes may be able to perform tasks...but they may not be very efficient or robust.

Many of the problems with badly designed classes are exposed when existing applications need to be changed. They can be very difficult to amend because they have not been designed systematically.

Principles of Object-Oriented Design: Cohesion

Cohesion

Cohesion is a measure of how strongly related the responsibilities of a single module are.

With high cohesion, the reusability of the program increases.

Cohesion decreases if the methods of a class have little in common.

Sujata Batra

Principles of Object-Oriented Design: Coupling

Coupling

Coupling is the degree to which each program relies on other modules.

Coupling helps to achieve maintainability.

Coupling should be low (loose) if possible.

Principles of Object-Oriented Design: Low Coupling – Overview

Low coupling

- Low Coupling is useful because individual design elements should share as little information as possible with other design elements.
- Low Coupling can be achieved using interfaces

Principles of Object-Oriented Design: Low Coupling – Using Interfaces

Why do you need interfaces to achieve low coupling?

- Interfaces provide a means of setting a standard.
- Interfaces provide a level of abstraction that makes programs easier to understand.

Overview: Use Principles for Good Class Design

- Coding quality classes pays off!
- To create long-lasting, solid designs, follow good
- design principles.

- Well designed classes:

- Have one (or a small number of similar) well defined functions.

- Share only the information they need to with other classes.

Overview: Use Principles for Good Class Design

- Coding quality classes pays off!
- To create long-lasting, solid designs, follow good
- design principles.

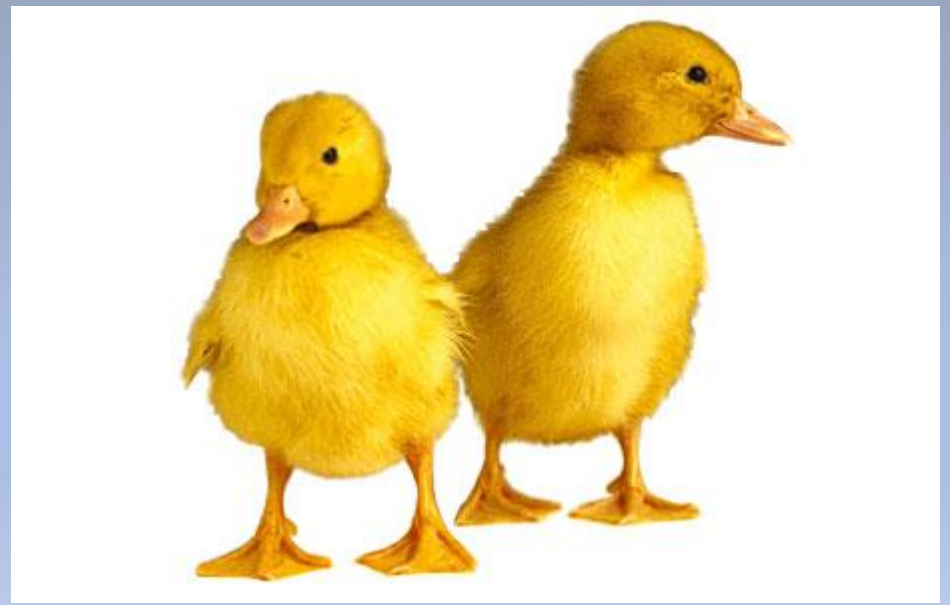
- Well designed classes:

- Have one (or a small number of similar) well defined functions.

- Share only the information they need to with other classes.

Good Design

- Two Aspects
  - Design should make the application work.
  - Design is reusable, flexible and maintainable.



Design Principle

- Is a basic tool or technique that can be applied to designing or writing code to make that code more maintainable , flexible and extensible.

Principle # 1: Single Responsibility Principle (SRP)

- A class should have a single reason to change.

- Problem Statement:

  An internet and GUI application is planning to use a modem class which should be able to perform the following –

  - Connect to the server.
  - Disconnect from the server.
  - Send the data to server.
  - Receive data from the server.
  - Draws the shape of the modem.

The first four behaviour is required by the internet application while the last is required by the GUI application.

Solution#1

```
/** The Modem class having different methods. */
class Modem {
    /* Dial-in to the server */
    public void dial(String connectionString) {

    }
    /* Disconnect from the server */
    public void hangUp() {

    }
    /* Send data to server */
    public void send(char c) {

    }
    /* Receive data from server */
    public char receive() {

        return ' ';
    }
    /* Draw the model for a GUI application */
    public void draw() {

    }
}
```

Sujata Batra

Solution#1 | Problems

- Responsibility means “Reason for a change” (in context of SRP).
- Modem class has the following responsibilities:
  - Connection Related
  - Data Transmission Related
  - Drawing related

- InternetGUI application will have the access to methods which it does not require. (deployment overhead)
- If only draw method is changed, redeployment required in both the applications.

Solution#2

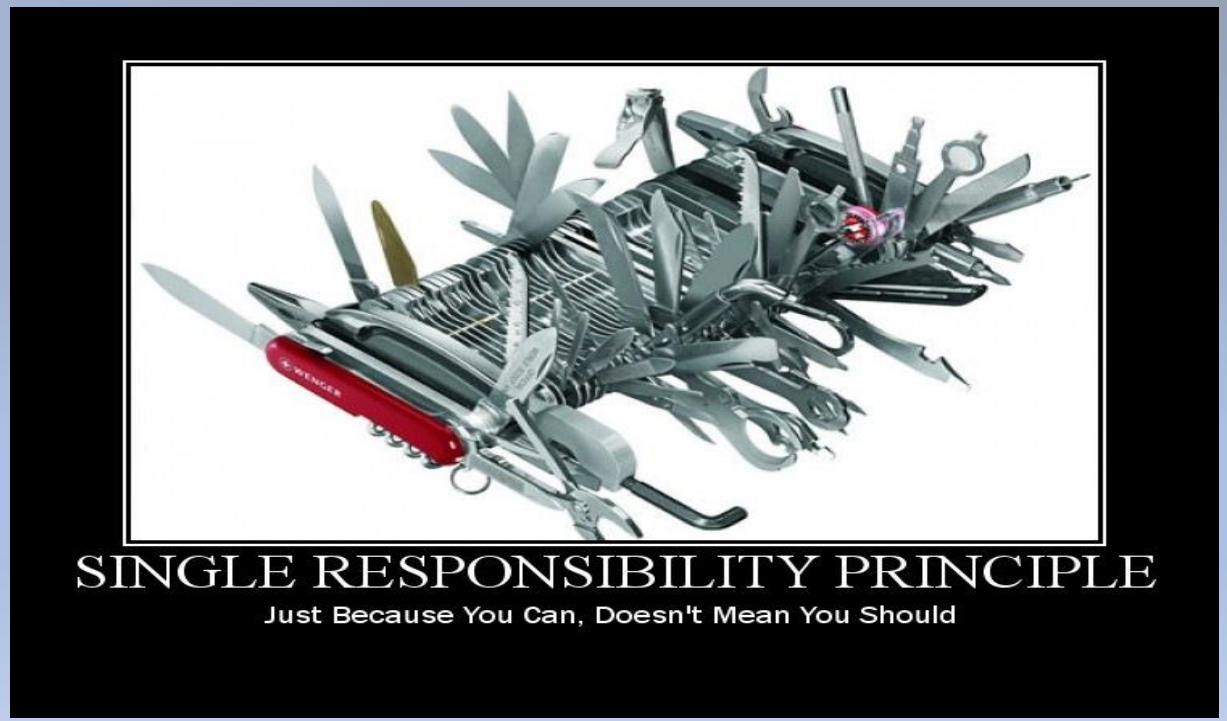
```java
/** The class having server connection methods. */
class Connection {
    /* Dial-in to the server */
    public void dial(String connectionString) {

    }

    /* Disconnect from the server */
    public void hangUp() {

    }
}
```

```java
/** The class having data transmission methods. */
class DataTransmission {
    /* Send data to server */
    public void send(char c) {

    }

    /* Receive data from server */
    public char receive() {

        return ' ';
    }
}
```

```java
/** The class having GUI drawing methods. */
class GUIDrawing {
    /* Draw the model for a GUI application */
    public void modemDraw() {

    }
}
```

Advantages

- Code complexity reduced.
- Improved Readability.
- Reduced Coupling.
- Code has better chances of cleanly evolving.



Sujata Batra

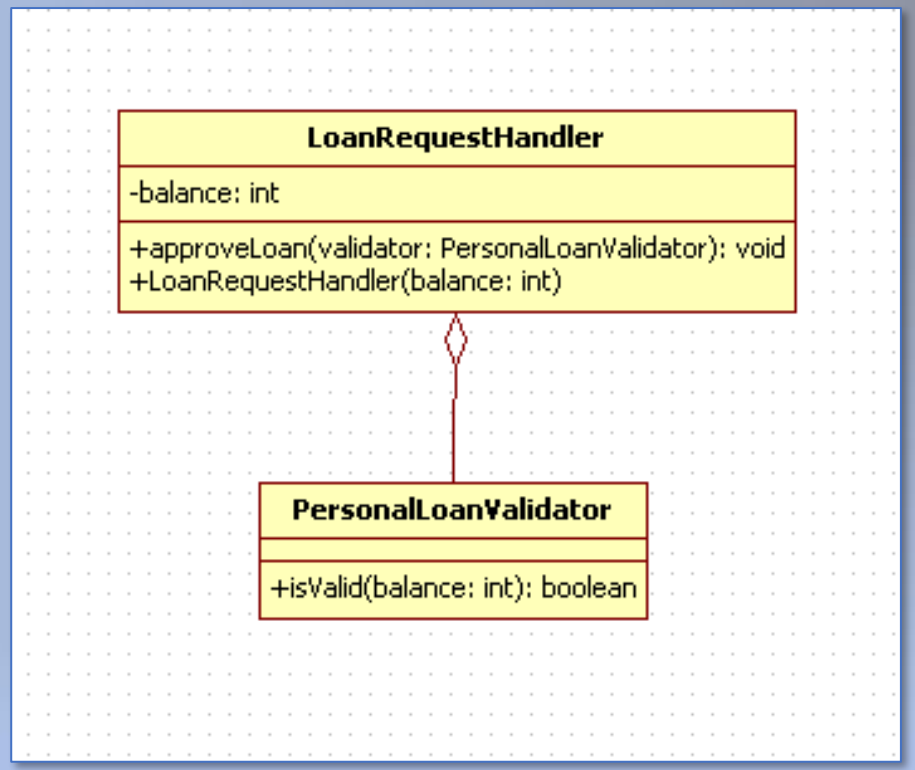
Principle #2: The Open-Closed Principle (OCP)

- A software module (classes) should be open for extension and closed for modification.

- Problem Statement:

  Design a *banking application* in which a particular loan is passed through a *software module* for approval. This software module approves the loan only if the balance in the applicant's bank account is above certain value.

UML Diagram



Sujata Batra

```java
class LoanRequestHandler {

    /** Initial bank balance */
    private int balance;

    /** Creates a new instance of
     *  LoanRequestHandler */
    public LoanRequestHandler(int balance) {
        this.balance = balance;
    }

        /** The method to approve the loan */
    public void approveLoan(PersonalLoanValidator
                            validator) {
        if (validator.isValid(balance)) {
            // sanction the loan
            System.out.println("Loan approved...");
        }
        else {
            System.out.println("Sorry...");
        }
    }
}
```

Solution#1

```java
class PersonalLoanValidator {

    /** Creates a new instance */
    public PersonalLoanValidator() {
    }

    /** Method having the logic for loan approval */
    public boolean isValid(int balance) {
        if (balance > 1000)
            return true;
        else
            return false;
    }
}
```

Sujata Batra

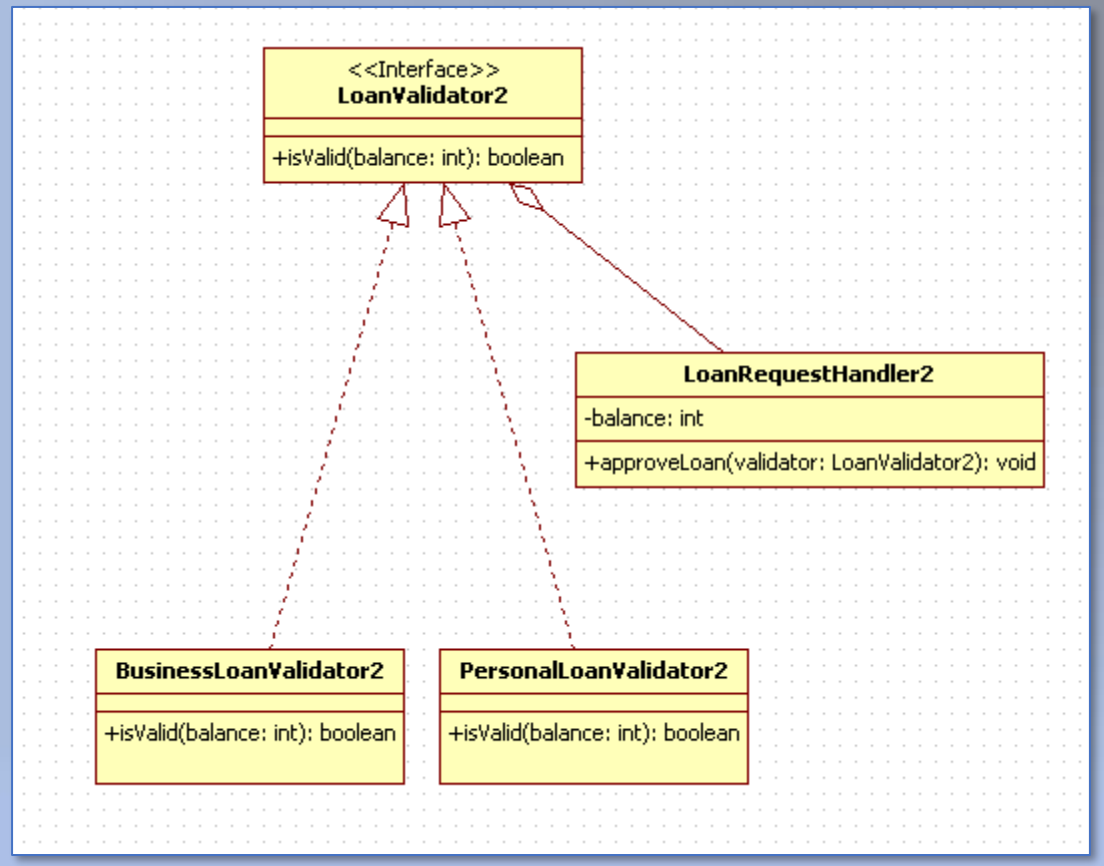
Problem in Solution#1

- High Coupling between *LoanRequestHandler* and *PersonalLoadValidator*.
- Every time bank decides to provide a different type of loan, modifications are required in both the classes.

- How to solve above design problems?

- Avoid the strong coupling between the two classes.
- Abstract the common loan validator and use this abstracted entity in the *LoanRequestHandler*.

UML Diagram (Solution)



```java
interface LoanValidator2 {
    public boolean isValid(int bankBalance);
}
```

```java
class BusinessLoanValidator2 implements LoanValidator2 {
    public boolean isValid(int balance) {
        // logic to approve business loan
        return true;
    }
}

class PersonalLoanValidator2 implements LoanValidator2 {
    public boolean isValid(int balance) {
        // logic to approve business loan
        return true;
    }
}
```

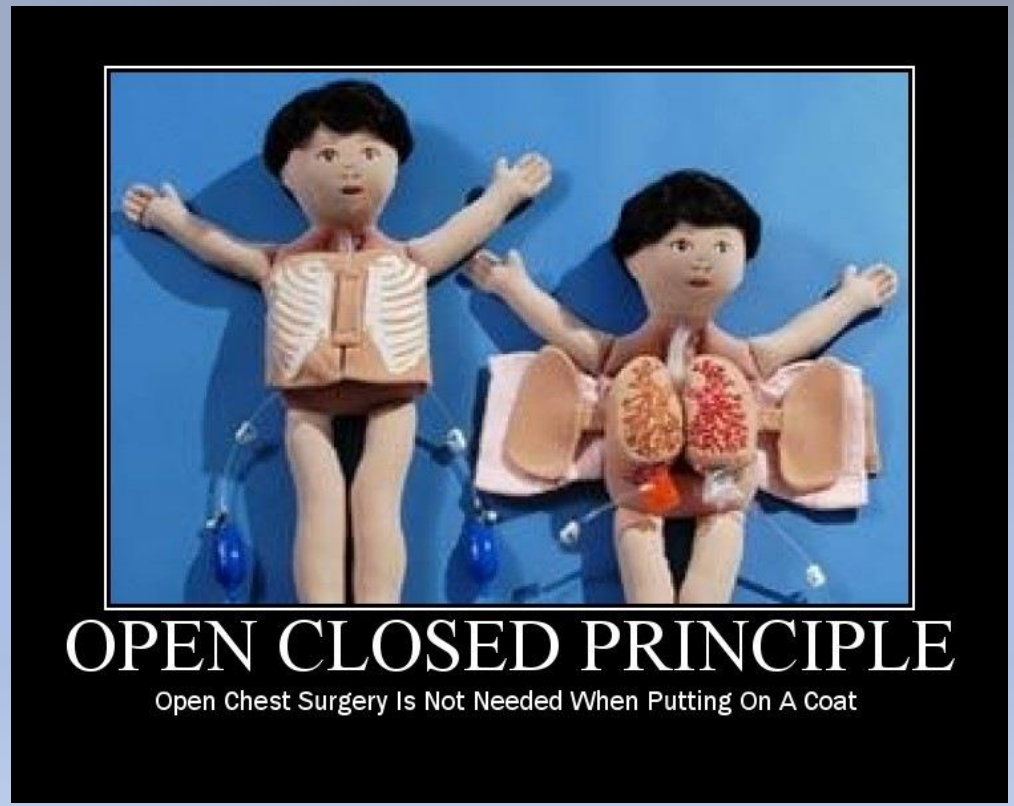
```java
class LoanRequestHandler2 {
    /** Bank Balance */
    private int balance;

    // Add the constructor to initialize balance...

    /** The method to approve the loan */
    public void approveLoan(LoanValidator2 validator) {
        if (validator.isValid(balance)) {
            // sanction the loan
            System.out.println("Loan approved...");
        }
        else {
            System.out.println("Sorry...");
        }
    }
}
```

Solution#2

Sujata Batra

Benefits and Usage

- Low Coupling between *LoanRequestHandler* and *PersonalLoadValidator*.
- No modification required in *LoanRequestHandler* to handle different types of loan in future.



Principle # 3: Liskov's Substitution Principle (LSP)

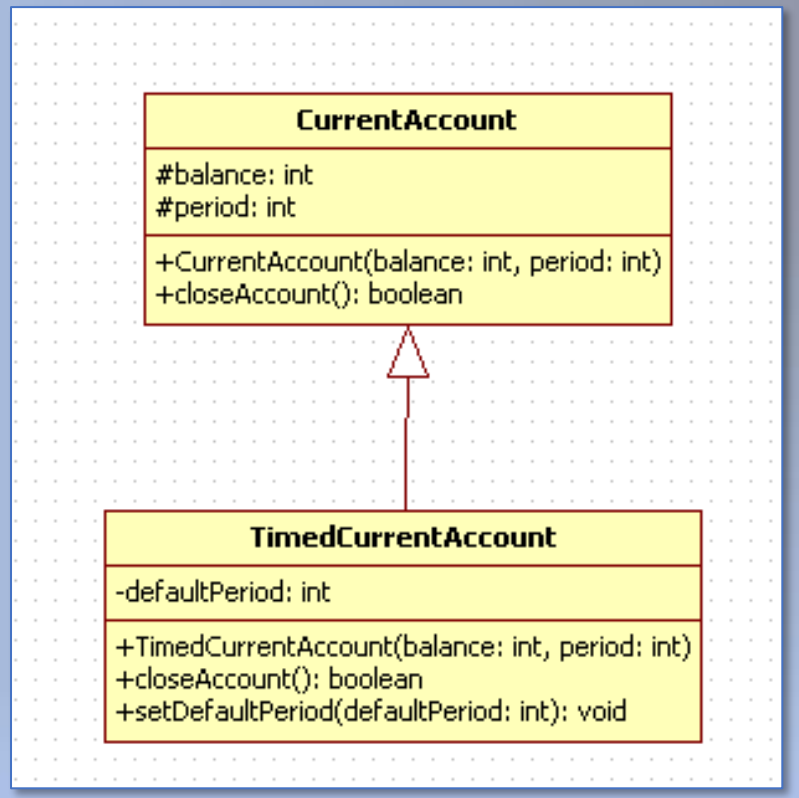
- Subtypes must be completely substitutable for their base types.

- Problem Statement:

  A bank manages two types of accounts – Current Account and Timed Current Accounts. The current account can be closed only if the balance on that account is positive. To close timed current account, the balance needs to be positive and it needs to be open for at least 6 months. The Timed Current Accounts give additional 1% interest. Design the classes.

- Some Facts:
  - All methods except closing an Account and calculating interest rate are same.

UML Diagram



Sujata Batra

```
class CurrentAccount {

    /** Holds value of property balance. */
    protected int balance;
    /** Holds value of property period. */
    protected int period;

    /** Creates a new instance of Account */
    public CurrentAccount(int balance, int period) {
        this.balance = balance;
        this.period = period;
    }

    /** closes the account */
    public void closeAccount() {
    }
}
```

Solution#1

```
public void closeAnAccount() {
    CurrentAccount ac = <Implementation>
    System.out.println("Account close result: " + ac.closeAccount());
}
```

```
        if (balance > 0 && period > defaultPeriod)
            return true;
        else
            return false;
    }

    /**
     * Setter for property defaultPeriod.
     */
    public void setDefaultPeriod(int defaultPeriod) {
        this.defaultPeriod = defaultPeriod;
    }
}
```

Behaviour dependant upon the object type.

Sujata Batra

Problem in Solution#1

- Unexpected Result.
- User to know the internal implementation of the classes.

- How to solve above design problems?
  - Subclasses should completely replace the base class.

Design by Contract Principle

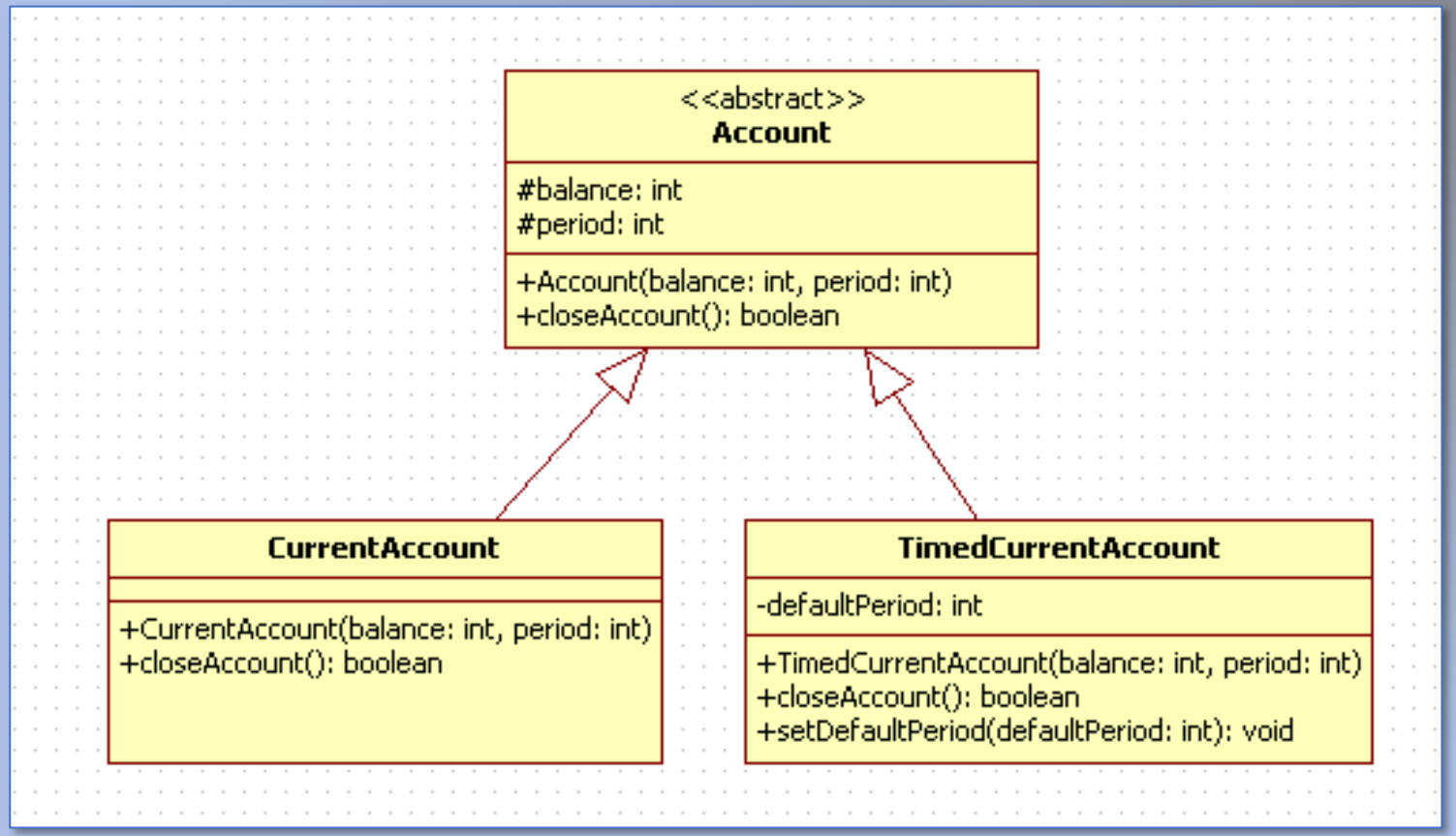
Design by Contract

- Each method can have pre-conditions and post-conditions.

- Pre-conditions
  - Criteria to be met before the method offers a behaviour
- Post-condition
  - State or behaviour offered by the method after it meets preconditions

A sub-type can only have
weaker pre-conditions
and
stronger post-conditions
than its base class.

UML Diagram(Solution)



```java
abstract class Account {
    /** Holds value of property balance. */
    protected int balance;
    /** Holds value of property period. */
    protected int period;

    /** Creates a new instance of Account */
    public Account(int balance, int period) {
        this.balance = balance;
        this.period = period;
    }

    /**
     * Make the closeAccount method abstract so
     * that the user does not assume its implementation.
     */
    public abstract boolean closeAccount();
}
```

Solution#2

```java
class CurrentAccount2 extends Account {

    /** closes the account */
    public boolean closeAccount() {
        if (balance > 0)
            return true;
        else
            return false;
    }
}
```

```java
class TimedCurrentAccount2 extends Account {

    /** Holds value of property defaultPeriod. */
    private int defaultPeriod;

    // Constructor and setter of instance variable

    public boolean closeAccount() {
        if (balance > 0 && period > defaultPeriod)
            return true;
        else
            return false;
    }
}
```

Benefits and Usage

- The user won't assume the implementation of closeAccount method.
- Always Remember

"Derived classes must enforce weaker pre-conditions and stronger post-conditions while overriding any base class method"

Sujata Batra

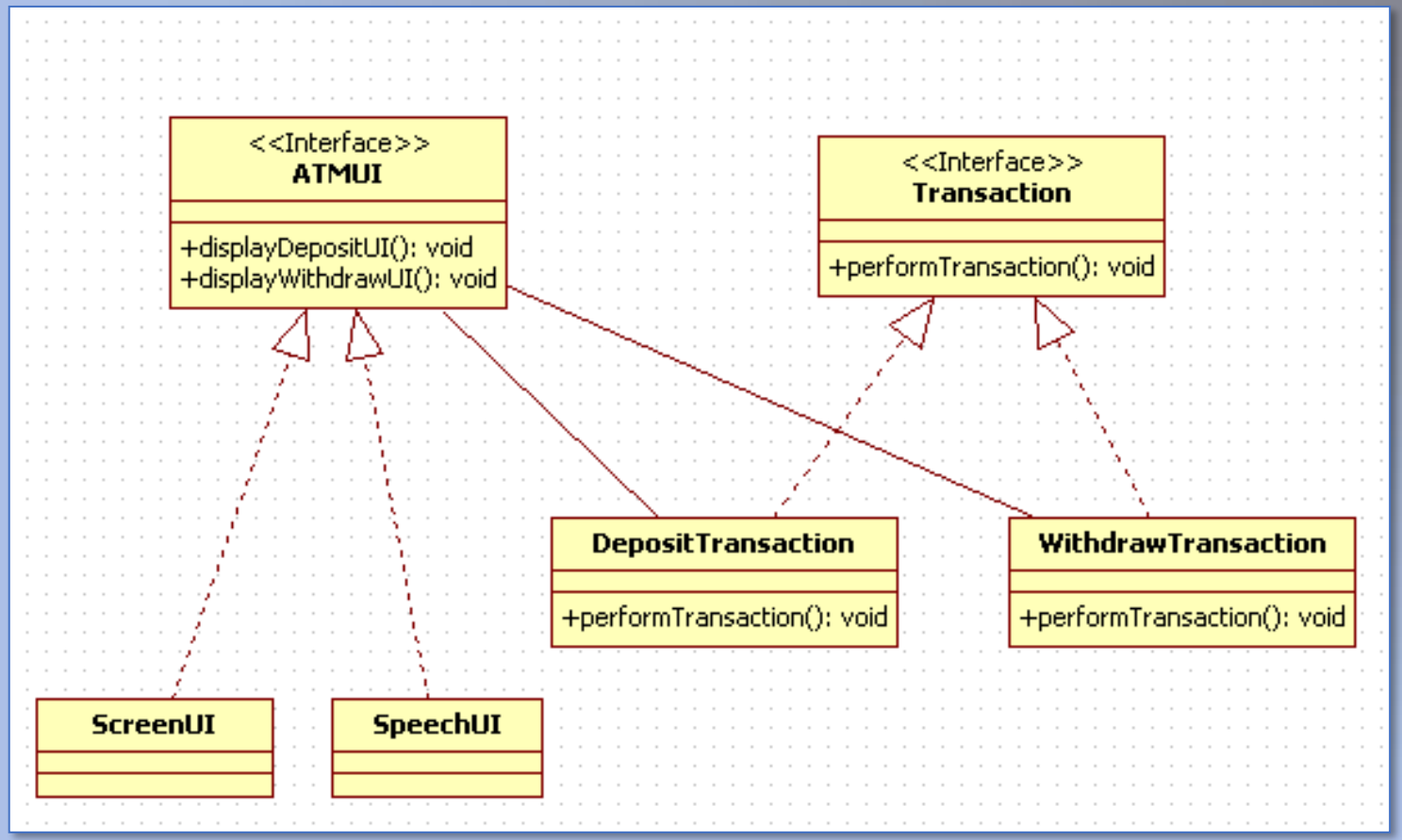
Principle # 4: Interface Segregation Principle (ISP)

- Many specific interfaces are better than a single general interface.

- Problem Statement:

  An ATM can have the following types of output – a screen and a speech. It supports the following two types of transactions – deposit and withdrawal. A company supports two types of ATMs – only screen (for ppl who can see) and screen with speech (for handicapped ppl). Design the classes.

UML Diagram



Sujata Batra

Solution#1

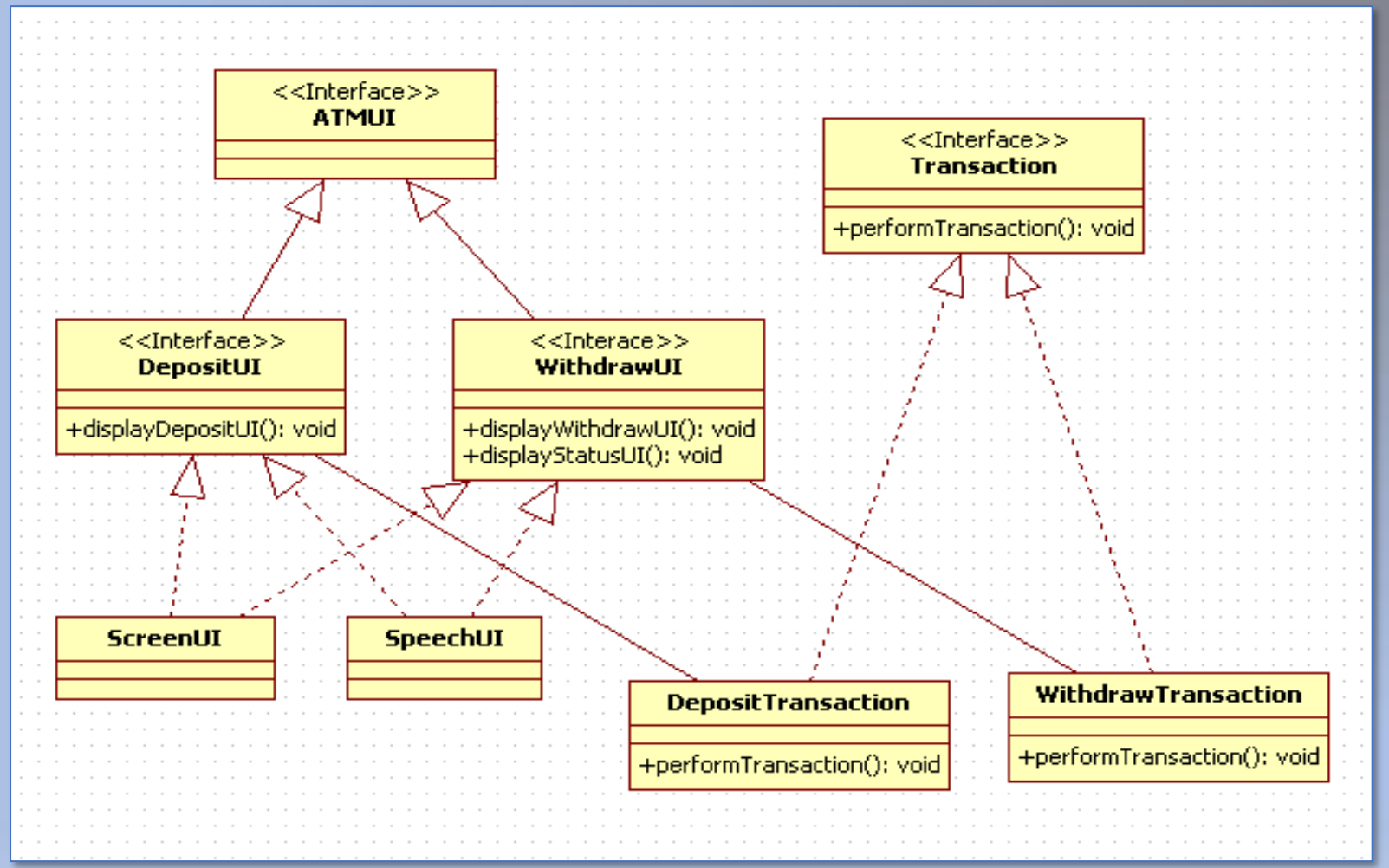
```
2
3  /** ATM UI interface having different methods that need
4   *  to be called by the transaction classes.
5   */
6  interface ATMUI {
7      public void displayDepositUI();
8      public void displayWithdrawUI();
9  }
10
11 class ScreenUI implements
12  // Implement all the meth
13 }
14
15 class SpeechUI implements
16  // Implement all the met
17 }
```

```
19 // Transaction Interface
20 interface  Transaction {
21   public abstract void performTransaction();
22 }
23
24 // Deposit transaction class implementing Transaction
25 class DepositTransaction implements Transaction {
26     public void performTransaction() {
27         // Perform the deposit transaction
28         ATMUI atmUI = <Implementation>
29         atmUI.displayDepositUI();
30     }
31 }
32
33 // Withdraw transaction class implementing Transaction
34 class WithdrawTransaction implements Transaction {
35     public void performTransaction() {
36         // Perform the withdraw transaction
37         ATMUI atmUI = <Implementation>
38         atmUI.displayWithdrawUI();
39     }
40 }
```

Solution#1 | Problems

- Contract change in one UI will affect the other UI implementations.
- Fat interface – Grow rapidly.

UML Diagram (Solution)



Solution#2

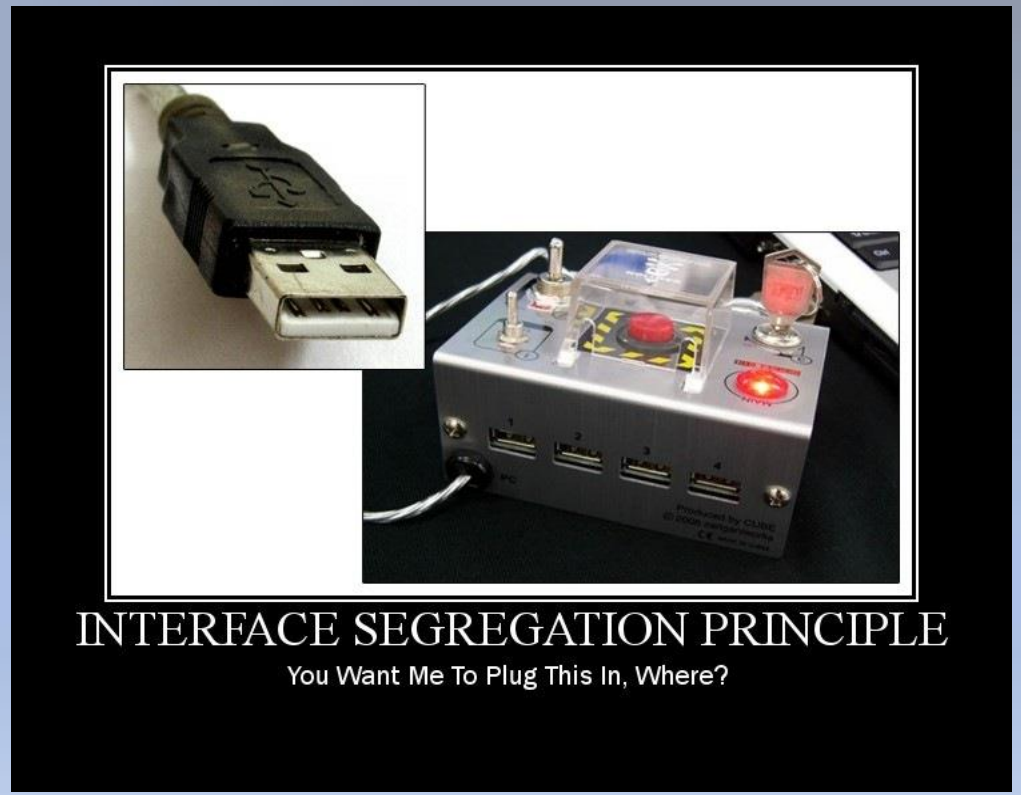
```
/** The base interface to all the interfaces.*/
interface ATMUI {
}

/** The
interfa
    pub
}

/** The
interfa
    pub
    pub
}
```

```
// Transaction Interface
interface  Transaction {
  public abstract void performTransaction();
}

// Deposit transaction class implementing Transaction
class DepositTransaction implements Transaction {
    public void performTransaction() {
        // Perform the deposit transaction
        DepositUI depositUI = <Implementation>
        depositUI.displayDepositUI();
    }
}

// Withdraw transaction class implementing Transaction
class WithdrawTransaction implements Transaction {
    public void performTransaction() {
        // Perform the withdraw transaction
        WithdrawUI withdrawUI = <Implementation>
        withdrawUI.displayWithdrawUI();
    }
}
```

Sujata Batra

Advantages

- Small and relevant contracts.
- Easy to understand and maintain.
- Contract change in one UI will not affect the other UI implementations.



Principles # 5: Dependency Inversion Principle (1 of 2)

- In the Dependency Inversion Principle (DIP) abstractions should not depend on details. Details should depend on abstractions.
- The key to implement DIP is to define appropriate interfaces or abstract classes, and provide implementation in derived (concrete) classes. Users refer to the interface or abstract class and not on the concrete implementation for any functionality.
- Your mailbox defines the size of packages you receive, the packages don’t define the size of your mailbox.

Principles of Object-Oriented Design: Dependency Inversion Principle (2 of 2)

Advantages of DIP:

- Higher level classes are created that do not depend on the lower level classes to function.
- Modules become loosely coupled and flexible for future extensions.

- If we consequently apply the Open/Closed Principle and the Liskov Substitution Principle to our code, it will also follow the Dependency Inversion Principle.

Design Patterns

Sujata Batra

Introduction

- General repeatable solution to a commonly occurring problem in software **design**.
- A **design pattern** isn't a finished **design** that can be transformed directly into code.
- It is a description or template for how to solve a problem that can be used in many different situations.

Benefits of Design Pattern

- They give the developer a selection of tried and tested solutions to work with
- They are language neutral and so can be applied to any language that supports object-orientation
- They aid communication by the very fact that they are well documented and can be researched if that is not the case.
- They have a proven track record as they are already widely used and thus reduce the technical risk to the project
- They are highly flexible and can be used in practically any type of application or domain

What is Gang of Four(GoF)

- In 1994, four authors :
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides

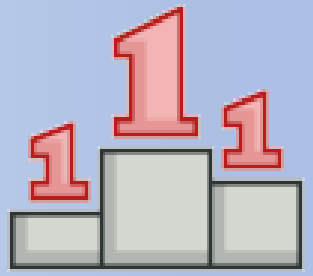
  published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.
  - These authors are collectively known as **Gang of Four (GOF)**.

Types of Design Pattern

- Creational
  - Deals with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.
- Structural
  - Ease the design by identifying a simple way to realize relationships between entities.
- Behavioural
  - Identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

Types of Creational Design Pattern

- Singleton
- Factory
- Abstract Factory
- Prototype
- Builder



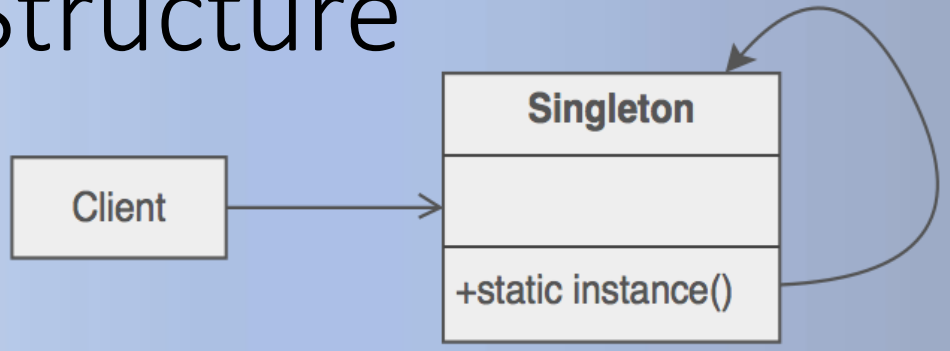
Singleton pattern

Sujata Batra

Singleton Design Pattern

- Intent
  - Ensure a class has only one instance, and provide a global point of access to it.
  - Encapsulated "just-in-time initialization" or "initialization on first use".
- **Problem**
  - Application needs one, and only one, instance of an object.

Structure



Make the class of the single instance responsible for access and "initialization on first use". The single instance is a private static attribute. The accessor function is a public static method.

Implementation

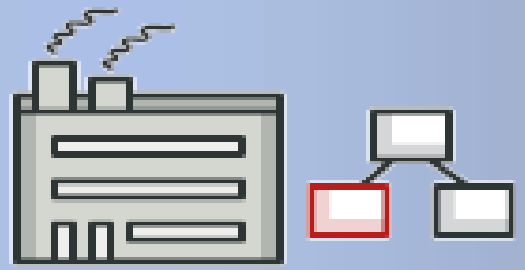
1.Define a private static attribute in the "single instance" class.
2.Define a public static accessor function in the class.
3.Do "lazy initialization" (creation on first use) in the accessor function.
4.Define all constructors to be protected or private.
5.Clients may only use the accessor function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

Lazy Initialization Implementation

```
public final class SingletonDemo {
    private static SingletonDemo instance = null;
    private SingletonDemo() { }

    public static synchronized SingletonDemo getInstance() {
        if (instance == null) {
            instance = new SingletonDemo();
        }

        return instance;
    }
}
```

Eager initialization

```
public final class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```



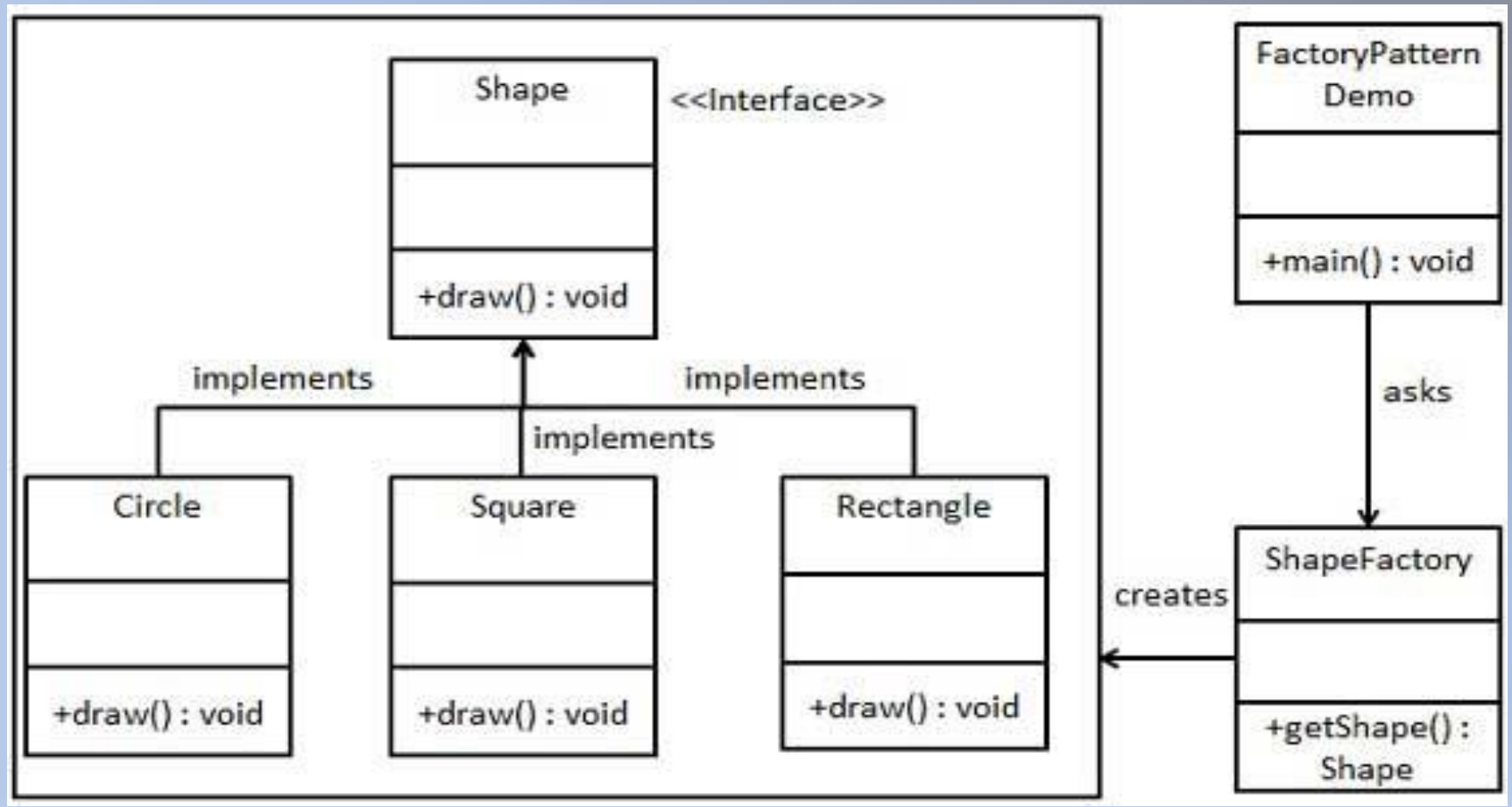
Factory Method pattern

Sujata Batra

Factory Design Pattern

- Uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.
- Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Problem
  - A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

Structure



Implementation

- If you have an inheritance hierarchy that exercises polymorphism, consider adding a polymorphic creation capability by defining a static factory method in the base class.
- Design the arguments to the factory method. What qualities or characteristics are necessary and sufficient to identify the correct derived class to instantiate?
- Consider making all constructors private or protected.

Behavioural Design Pattern

- Command
- Observer
- Iterator
- Template
- Chain of Responsibility
- Strategy
- Null Object

Sujata Batra

Iterator pattern

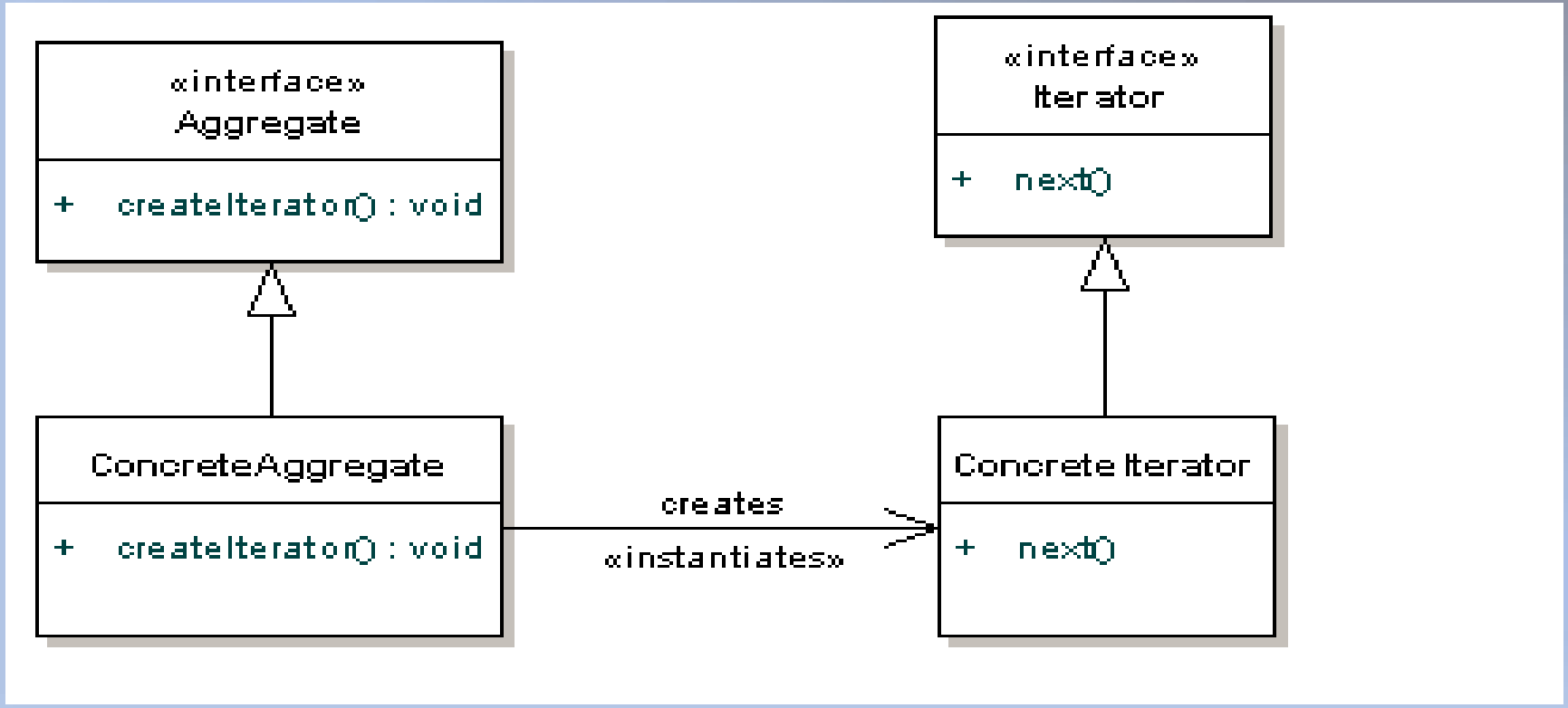
Sujata Batra

Iterator

- **Intent**
- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- The C++ and Java standard library abstraction that makes it possible to decouple collection classes and algorithms.
- Promote to "full object status" the traversal of a collection.
- Polymorphic traversal

- **Problem**
  - Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.

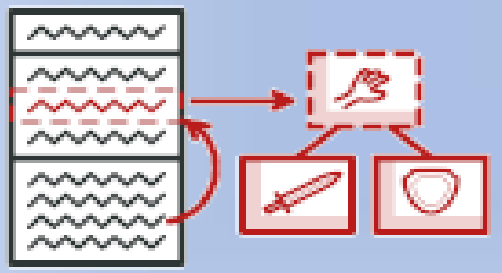
Structure



Implementation

- Add a create_iterator() method to the "collection" class, and grant the "iterator" class privileged access.
- Design an "iterator" class that can encapsulate traversal of the "collection" class.
- Clients ask the collection object to create an iterator object.
- Clients use the first(), is_done(), next(), and current_item() protocol to access the elements of the collection class.

Strategy pattern



Sujata Batra

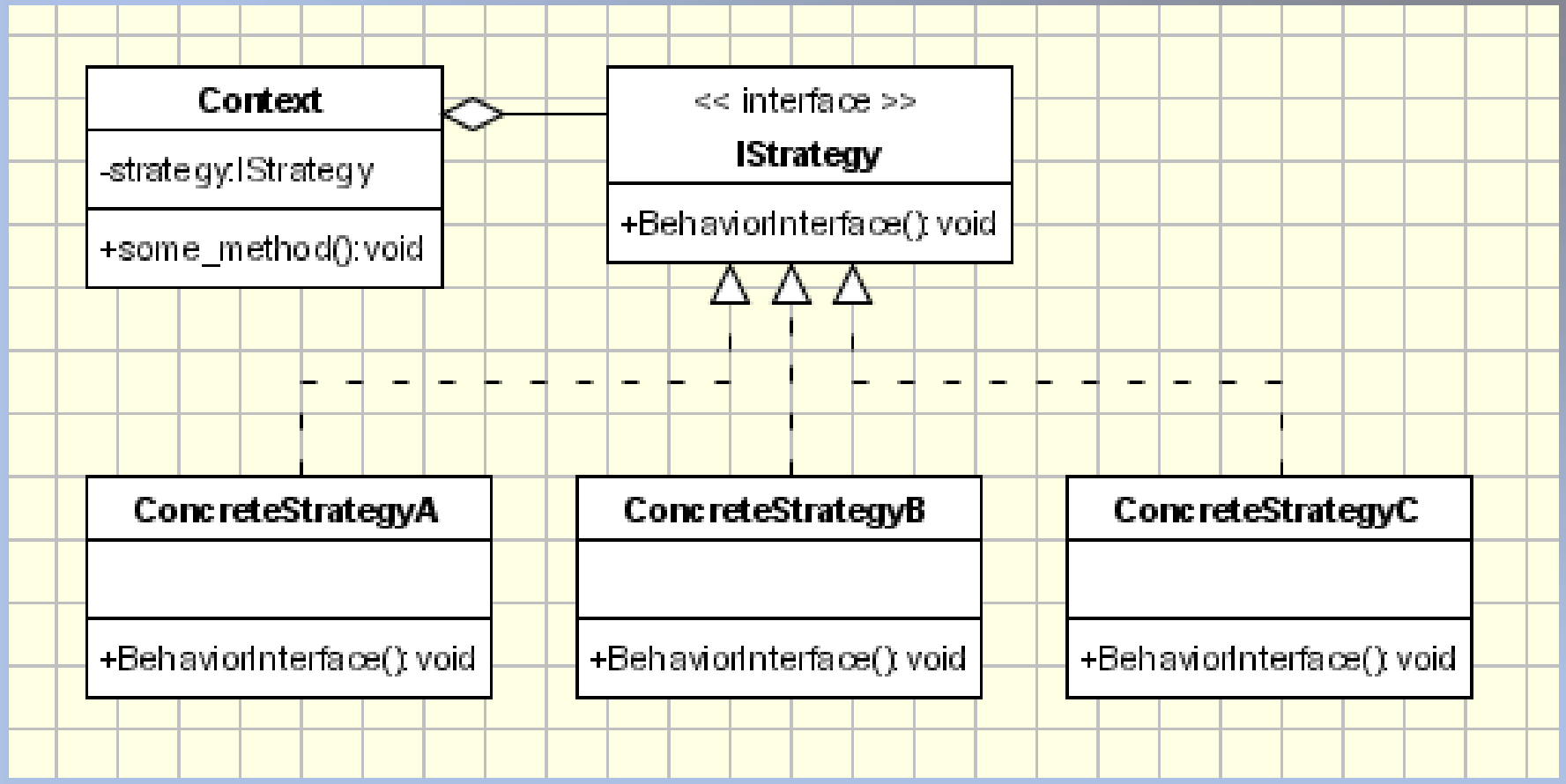
Strategy pattern

- **Intent**
  - *Define a family of algorithms, encapsulate each one, and make them interchangeable.*
  - *Strategy lets the algorithm vary independently from the clients that use it.*
  - *Capture the abstraction in an interface, bury implementation details in derived classes*

- Also known as **Policy Pattern.**
- **Example :**

- *Collections.sort()* method that takes Comparator parameter.
- Based on Comparators the Objects are getting sorted in different ways,

Strategy pattern



Strategy pattern Implementation

- Strategy
  - defines an interface common to all supported algorithms.
- Context
  - Uses to call the algorithms defined by concrete strategy
  - Contains the reference to a Strategy Object
- Concrete Strategy
  - each concrete strategy implements an algorithm.

If necessary, addition objects can be defined to pass data from context object to strategy.

Strategy and Concrete Strategy

- **Strategy**

```
public interface PaymentStrategy {
    public void pay(double amount);
}
```

- **Concrete Strategy**

```
public class CardPayment implements PaymentStrategy {

public void pay(double amount) {
  System.out.println("Received Payment through Card");
}
}
```

Context

```
public class Patient {
        private  PaymentStrategy strategy;
         private double  amount;

public Patient(double amount) {
    super();
     this.amount = amount;
}
public PaymentStrategy getStrategy() {
return strategy;
}
public void setStrategy(PaymentStrategy strategy) {
this.strategy = strategy;
}
public void makePayment(){
   strategy.pay(4500);
}
}
```

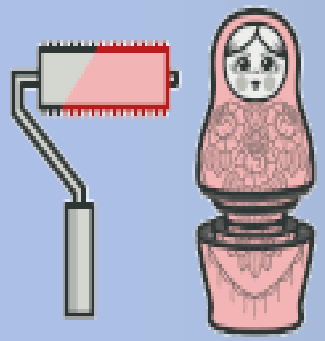
Client

```
public static void main(String[] args) {

Patient p1 =new Patient(4500);

Patient p2 =new Patient(3500);


p1.setStrategy(new CashlessPayment());

    p1. makePayment();

  p2.setStrategy(new CardPayment());

    p2. makePayment();


}
```

Structural Design Pattern

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

Sujata Batra



Decorator pattern

Sujata Batra

Decorator Pattern

- **Intent**
  - Pattern that attach new behaviors to objects by placing them inside wrapper objects that contain these behaviors.

- **Problem**
  - Need to add and remove responsibilities to an object dynamically
  - Inheritance is one of the solution to extend class behaviors.
  - Inheritance is static, can not add new subclasses to a program when it is already compiled and executed.

Decorator Pattern

- Decorator pattern is used a lot in Java IO classes, such as FileReader, bufferedReader etc.
- Elements of Decorator Pattern

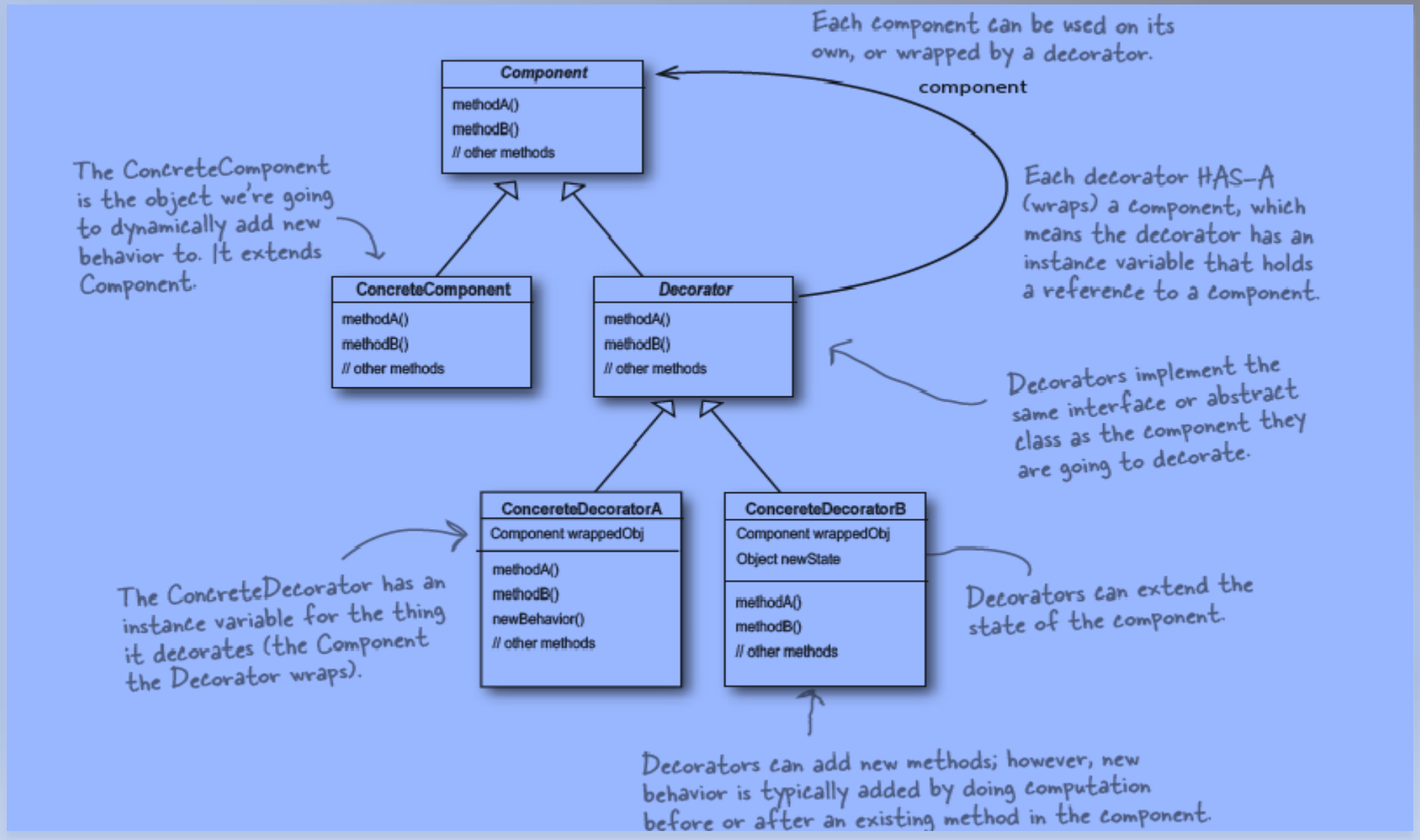
- **Component** - Interface for objects that can have responsibilities added to them dynamically.

- ConcreteComponent - Defines an object to which additional responsibilities can be added.

- Decorator - Maintains a reference to a Component object and defines an interface that conforms to Component's interface.

- **Concrete Decorators** - Concrete Decorators extend the functionality of the component by adding state or adding behavior.

Decorator



Sujata Batra

Decorator Pattern

- **The Component- They can be used on its own or wrapped wth Decorator**

```
public abstract class BankAccount {

  public abstract double balanceToMaintain();
  public abstract double deposit(double amount);


}
```

Decorator Pattern

- **The Concrete Component is the object ,to which new behavior is going to added to dynamically, It extends component**
  - //Concrete Component

```
public class SavingsAccount extends BankAccount {
@Override
public double deposit(double amount) {
return amount;
}

@Override
public double balanceToMaintain() {
return 1000;
}

}
```

The Decorator

- Each decorator Has-A wrapped component, the decorator has an instance variable that hold a reference to the component
- Decorators implement the same interface or abstract class as the component they are going to decorator

```
public abstract class FeatureDectorator extends BankAccount {


public  abstract double fundTransfer();

public abstract double balanceToMaintain();
}




}
```

Decorators

```
public class InternationalBanking extends FeatureDectorator {

private BankAccount account;
public InternationalBanking(BankAccount account) {
    this.account=account;

}

@Override
public double fundTransfer() {
// TODO Auto-generated method stub
return 0;
}
```

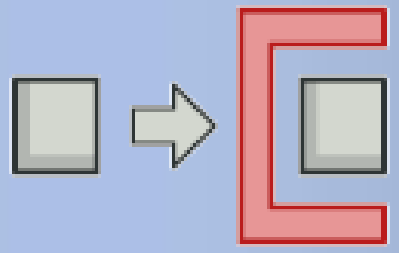
Decorator

```
public class OnlineBanking extends FeatureDectorator {

private BankAccount account;
public OnlineBanking(BankAccount account) {
   this.account = account;
}
public double fundTransfer() {
return 8000;
}
public double balanceToMaintain() {
return account.balanceToMaintain()+5000;
}
public double deposit(double amount) {
return 0;
}
public String chequeBookRequest(){
return "Allowed";
}
}
```

Sujata Batra

Test Decorators

```
public static void main(String[] args) {
    BankAccount account1 = new SavingsAccount();

    System.out.println("Minimum Balance "+account1.balanceToMaintain());

    OnlineBanking account1x = new OnlineBanking(account1);

    System.out.println("Minimum Balance with One Feature"+account1x.balanceToMaintain());

    System.out.println("Cheque Book Request"+account1x.chequeBookRequest());
    InternationalBanking account2x = new InternationalBanking(new OnlineBanking(account1));
        System.out.println("Minimum Balance with Two Feature"+account2x.balanceToMaintain());
        System.out.println("Airport Lounge Access" +account2x.airportLoungeAccess());  }}
```



Proxy pattern

Sujata Batra

Proxy Pattern

- ***proxy pattern is used when we want to provide controlled access of a functionality.***
- ***used to represent a complex with a simpler one.***

- *If creation of object is expensive*, its creation can be postponed till the very need arises and till then, a *simple object can represent it.*

- This simple object is called the “Proxy” for the complex object

- The Proxy holds reference to the RealObject ,Clients interact with the realObject through the proxy .

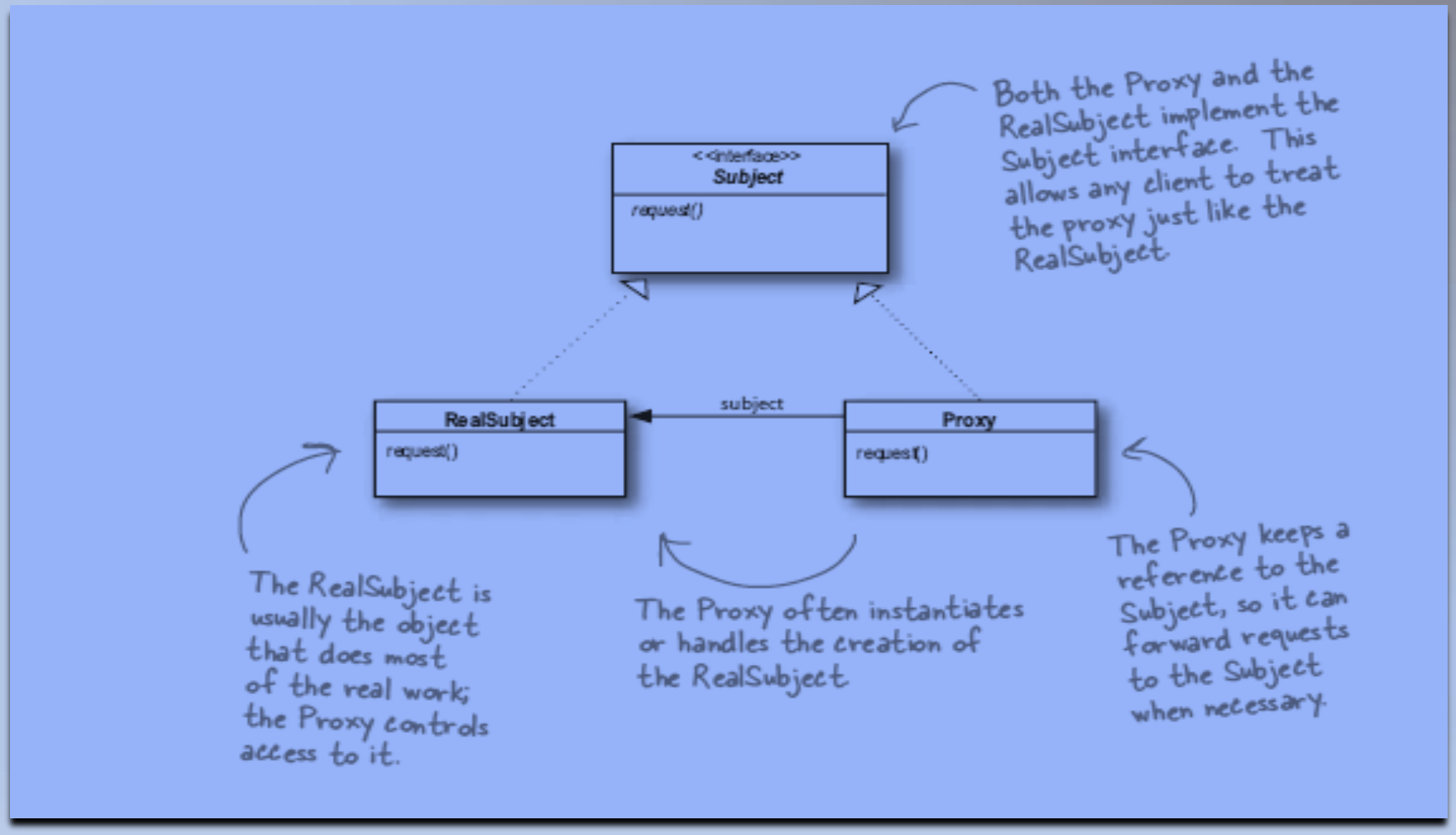
- Because both the proxy and RealObject implement the same interface .

- The Proxy can be substituted where Subject can be requested

Proxy Pattern

- The participants classes in the proxy pattern are:
- **Subject**
  - Interface implemented by the RealSubject and representing its services.
  - The interface must be implemented by the proxy as well so that the proxy can be used in any location where the RealSubject can be used.
- **Proxy**
  - Maintains a reference that allows the Proxy to access the RealSubject.
  - Implements the same interface implemented by the RealSubject so that the Proxy can be substituted for the RealSubject.
  - Controls access to the RealSubject and may be responsible for its creation and deletion.
  - Other responsibilities depend on the kind of proxy.
- **RealSubject** - the real object that the proxy represents.

Proxy



Proxy Pattern

```
public interface Subject {

public void request();

}


public class RealSubject implements Subject {

  public void request() {

        System.out.println("Hello From Real Subject");


  }
```

The Proxy Class

```
public class ProxySubject implements Subject {

  Subject sub;

  public void request() {

    sub = new RealSubject();
    sub.request();

  }

}
```

Test the Proxy

```
public class TestProxy {

  public void callProxy(Subject sub)
  {
    sub.request();
  }
    public static void main(String[] args) {

          ProxySubject pxySub = new ProxySubject();

          TestProxy tpxy = new TestProxy();

          tpxy.callProxy(pxySub);



  }
```