Structural Design Pattern

Ms. Nehal Adhvaryu

Introduction

Concerned with how classes and objects can be composed, to form larger structures.

These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

simplifies the structure by identifying the relationships.

Introduction

Types

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Fly Weight
- Proxy

Adapter

Also known as Wrapper.

Match interface of different classes.

Converts the interface of a class into another interface that a client wants.

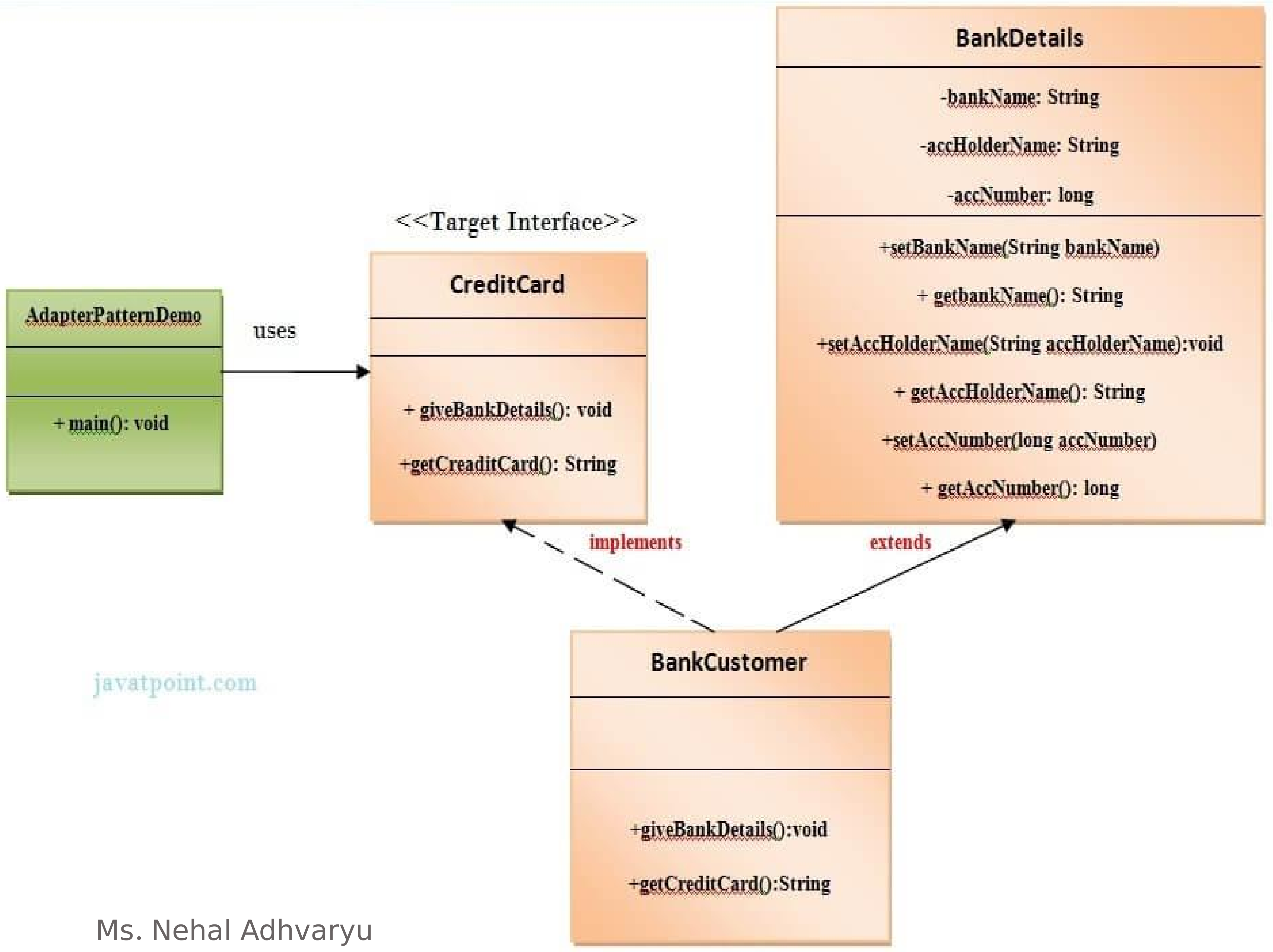
To provide the interface according to client requirement while using the services of a class with a different interface.

Adapter

Advantage

It allows two or more previously incompatible objects to interact.

It allows reusability of existing functionality.


BankDetails
-bankName: String
-accHolderName: String
-accNumber: long
+setBankName(String bankName)
+ getbankName(): String
+setAccHolderName(String accHolderName):void
+ getAccHolderName(): String
+setAccNumber(long accNumber)
+ getAccNumber(): long
<<Target Interface>>
CreditCard
+ giveBankDetails(): void
+getCreaditCard(): String
AdapterPatternDemo
+ main(): void
uses
implements
extends
BankCustomer
+giveBankDetails():void
+getCreditCard():String
javatpoint.com

Adapter

Target Interface: This is the desired interface class which will be used by the clients.

Adaptee class: This is the class which is used by the Adapter class to reuse the existing functionality and modify them for desired use.

Adapter class: This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class.

Client: This class will interact with the Adapter class.

Bridge

The Bridge Pattern is also known as **Handle or Body**.

Decouple the functional abstraction from the implementation so that the two can vary independently.

Advantage

- It enables the separation of implementation from the interface.
- It improves the extensibility.
- It allows the hiding of implementation details from the client.

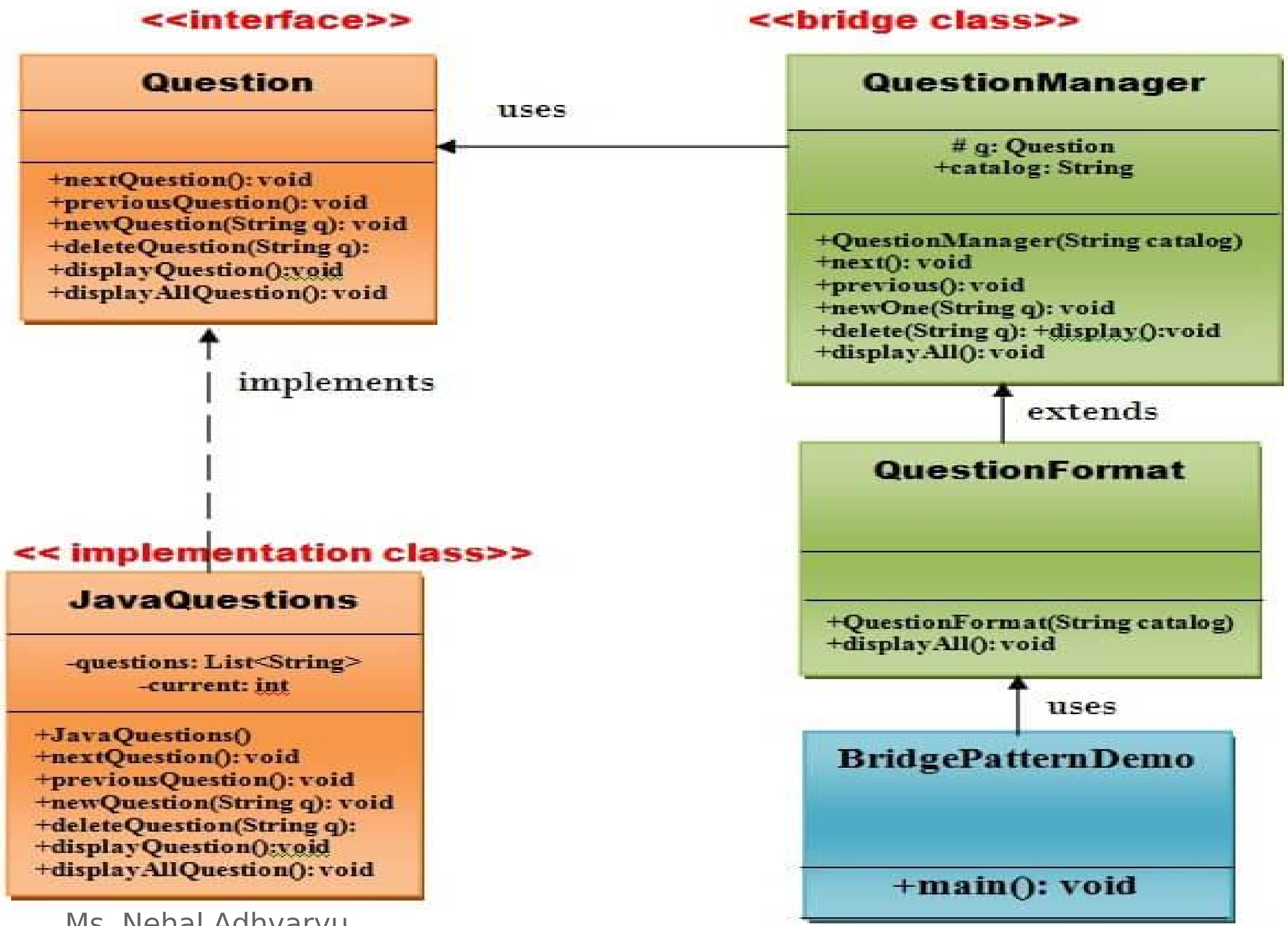
Bridge

Usage

When you don’t want a permanent binding between the functional abstraction and its implementation.

When both the functional abstraction and its implementation need to extended using sub-classes.

It is mostly used in those places where changes are made in the implementation does not affect the clients.



Ms. Nehal Adhvaryu

Composite

Allows us to treat different object in a similar fashion.

Every body use composite pattern directly or indirectly in a project because composite pattern is nothing but the implementation of interface.

Advantage

It defines class hierarchies that contain primitive and complex objects.

It makes easier to you to add new kinds of components.

It provides flexibility of structure with manageable class or interface.

Composite

Usage:

When you want to represent a full or partial hierarchy of objects.

When the responsibilities are needed to be added dynamically to the individual objects without affecting other objects. Where the responsibility of object may vary from time to time.

Composite

Components / Elements

Component :

The interface that all the components implement. This can also be a class.

Implements default behavior for the interface common to all classes as appropriate.

Leaf

Defines behavior for primitive objects in the composition.

A leaf has no children.

The unique objects in their own ways that implement or extend the Component.

Composite

Components / Elements

Composite

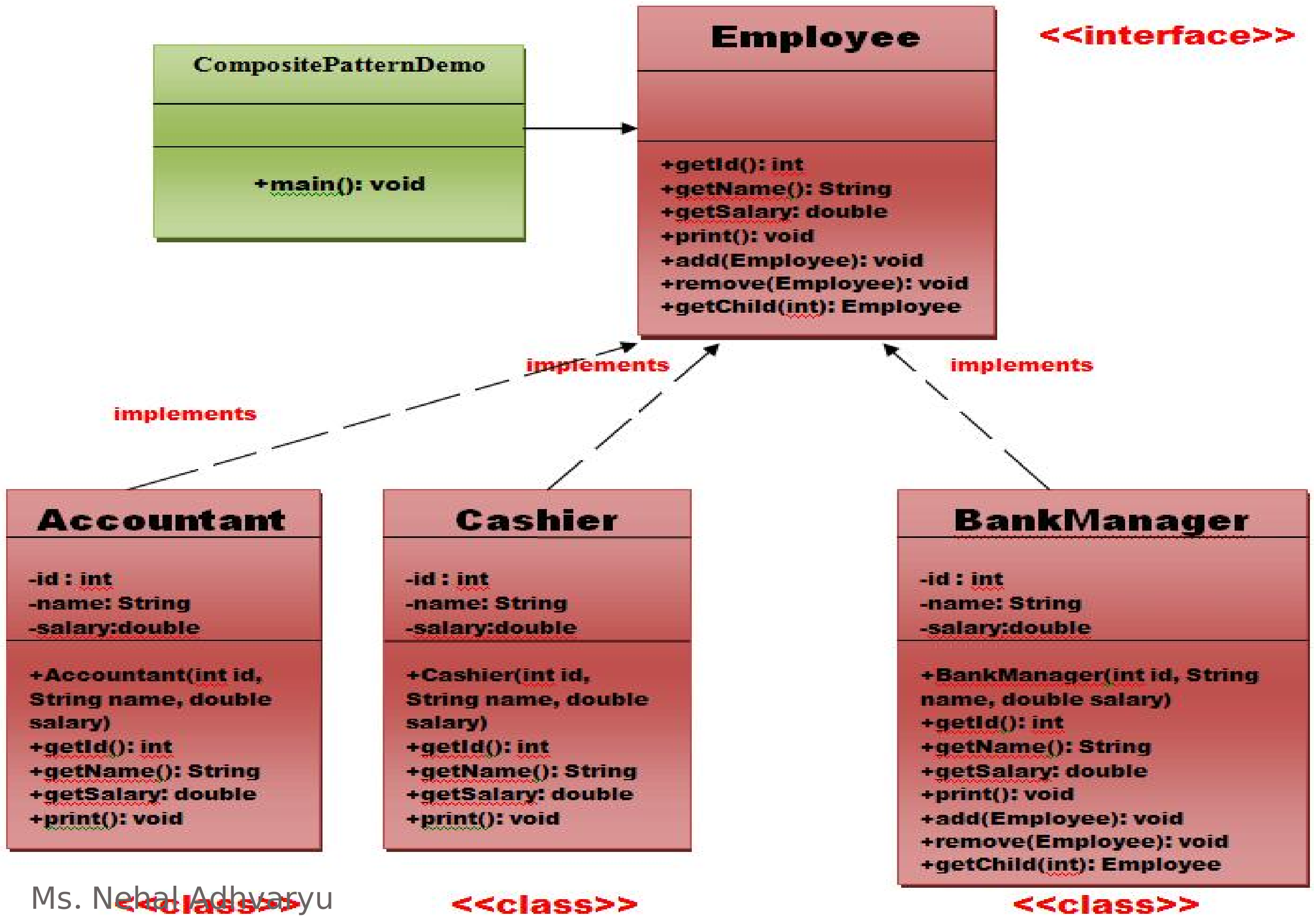
The single object made up of many components.

Implements child related operations in the component interface.

Client

Manipulates objects in the composition through the component interface.

Composite


CompositePatternDemo
+main(): void
Employee
+getId(): int
+getName(): String
+getSalary: double
+print(): void
+add(Employee): void
+remove(Employee): void
+getChild(int): Employee
<<interface>>
implements
implements
implements
Accountant
-id : int
-name: String
-salary:double
+Accountant(int id, String name, double salary)
+getId(): int
+getName(): String
+getSalary: double
+print(): void
Cashier
-id : int
-name: String
-salary:double
+Cashier(int id, String name, double salary)
+getId(): int
+getName(): String
+getSalary: double
+print(): void
BankManager
-id : int
-name: String
-salary:double
+BankManager(int id, String name, double salary)
+getId(): int
+getName(): String
+getSalary: double
+print(): void
+add(Employee): void
+remove(Employee): void
+getChild(int): Employee
<<class>>
<<class>>
<<class>>

Decorator

Says that just ‘attach a flexible additional responsibilities to an object dynamically‘.

In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

The Decorator Pattern is also known as Wrapper.

Decorator

Advantage

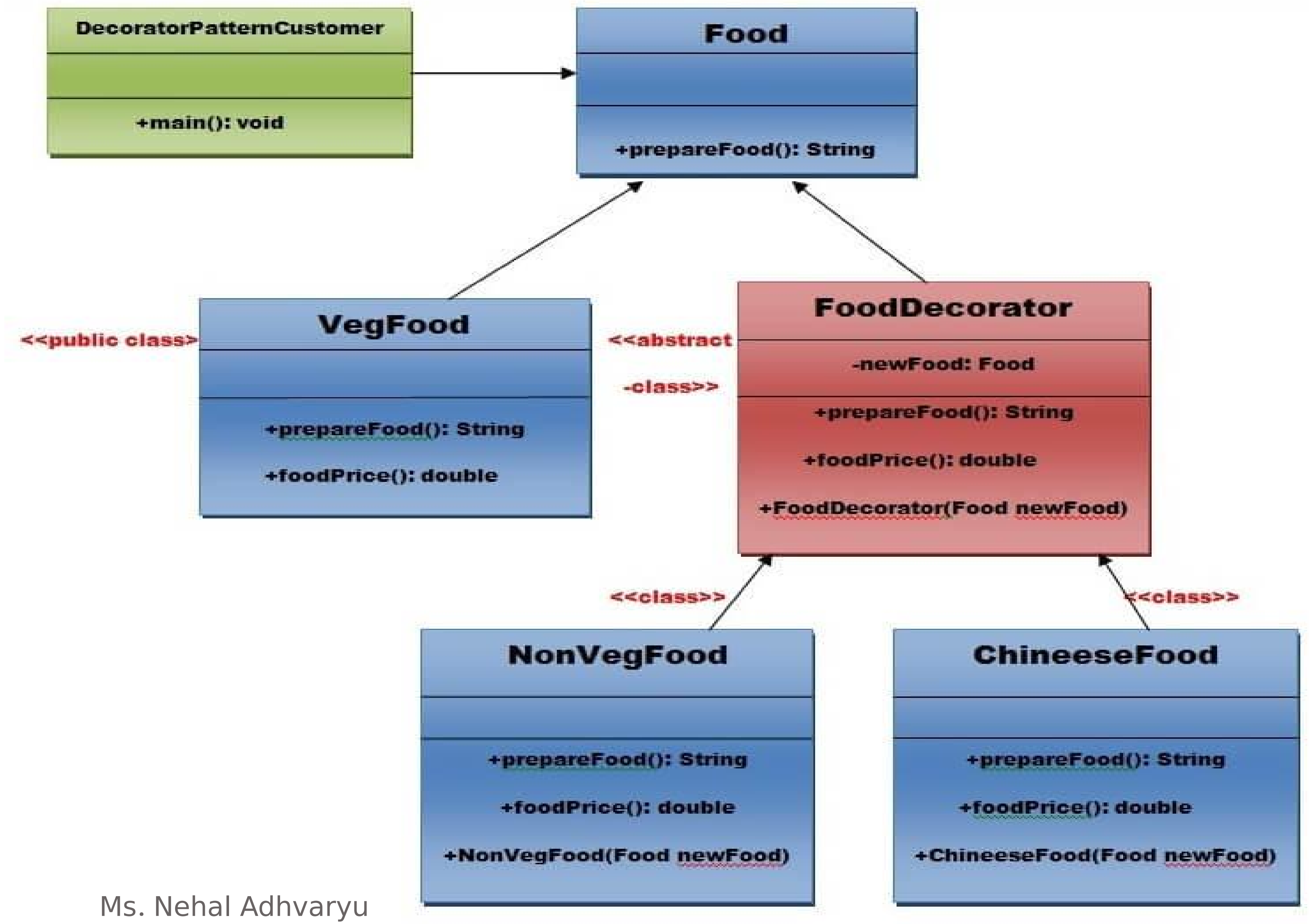
- It provides greater flexibility than static inheritance.
- It enhances the extensibility of the object, because changes are made by coding new classes.
- It simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object.

Usage

When you want to transparently and dynamically add responsibilities to objects without affecting other objects.

When you want to add responsibilities to an object that you may want to change in future.

Extending functionality by sub-classing is no longer practical.


DecoratorPatternCustomer
+main(): void
Food
+prepareFood(): String
VegFood
<<public class>
+prepareFood(): String
+foodPrice(): double
FoodDecorator
<<abstract
-class>>
-newFood: Food
+prepareFood(): String
+foodPrice(): double
+FoodDecorator(Food newFood)
<<class>>
NonVegFood
+prepareFood(): String
+foodPrice(): double
+NonVegFood(Food newFood)
<<class>>
ChineeseFood
+prepareFood(): String
+foodPrice(): double
+ChineeseFood(Food newFood)

Facade

A single class that represent an entire subsystem.

Its purpose is to hide internal complexity of an system and provide simple interface.

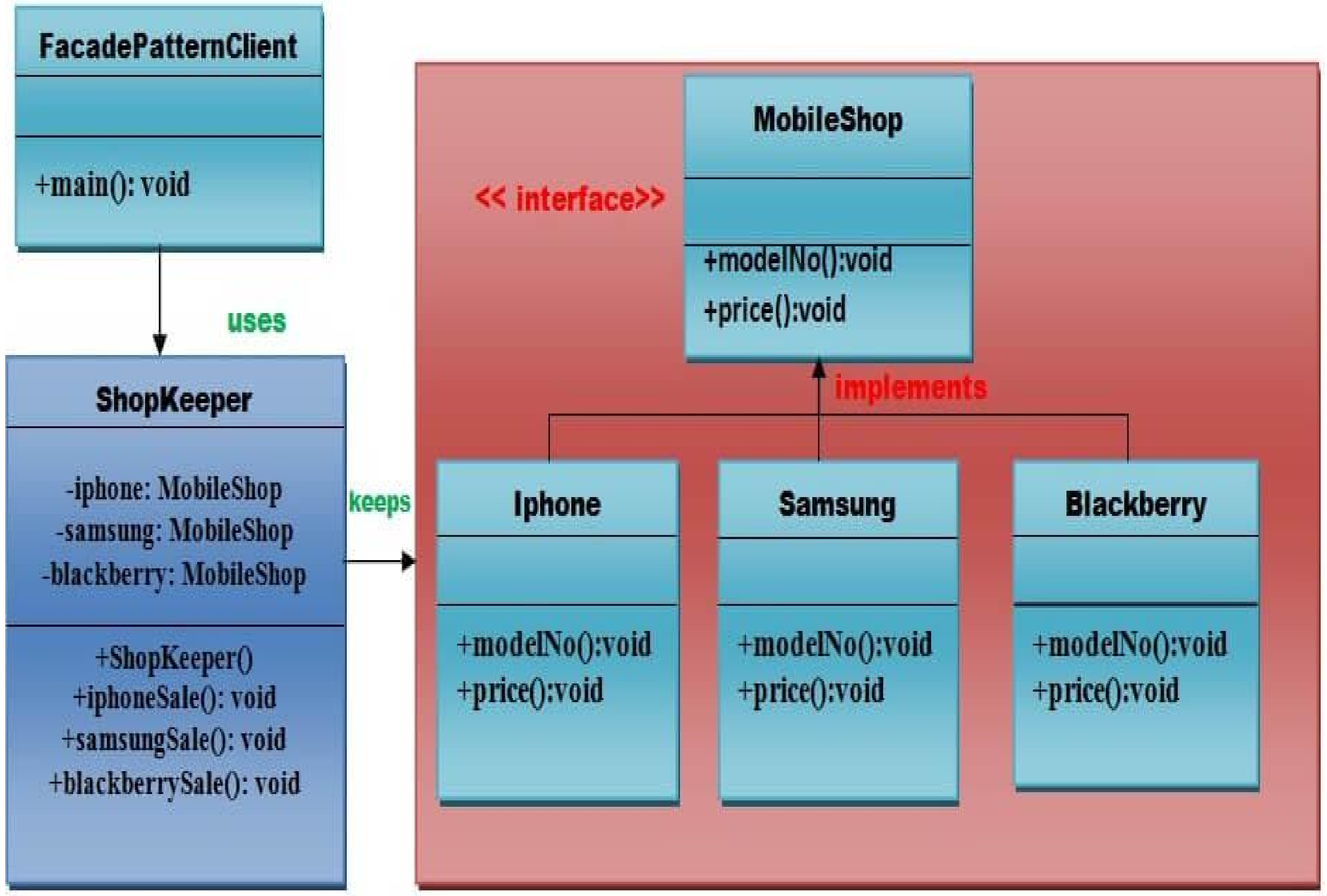
Advantage

- Reduce network calls.
- Reduce coupling
- Help in establishing transaction boundary.

Usage

When you want to provide simple interface to a complex sub-system.

When several dependencies exist between clients and the implementation classes of an abstraction.


FacadePatternClient
+main(): void
uses
ShopKeeper
-iphone: MobileShop
-samsung: MobileShop
-blackberry: MobileShop
+ShopKeeper()
+iphoneSale(): void
+samsungSale(): void
+blackberrySale(): void
keeps
<< interface>>
MobileShop
+modelNo():void
+price():void
implements
Iphone
+modelNo():void
+price():void
Samsung
+modelNo():void
+price():void
Blackberry
+modelNo():void
+price():void

Flyweight

To reuse already existing similar kind of objects by storing them and create new object when no matching object is found.

Advantage:

It reduces the number of objects.

It reduces the amount of memory and storage devices required if the objects are persisted

Flyweight

Usage

- When an application uses number of objects
- When the storage cost is high because of the quantity of objects.
- When the application does not depend on object identity.

Proxy

Proxy is the object that is being called by the client to access the real object behind the scene.

It represent functionality of another class.

Provides the control for accessing the original object.

Also known as Surrogate or Placeholder.

Advantage:

It provides the protection to the original object from the outside world.

Proxy

Types of Proxy:

Remote

Provides a local representation of the object which is actually stored in the different address location.

Virtual

Consider a situation where there is multiple database call to extract huge size image. Since this is an expensive operation so here we can use the proxy pattern

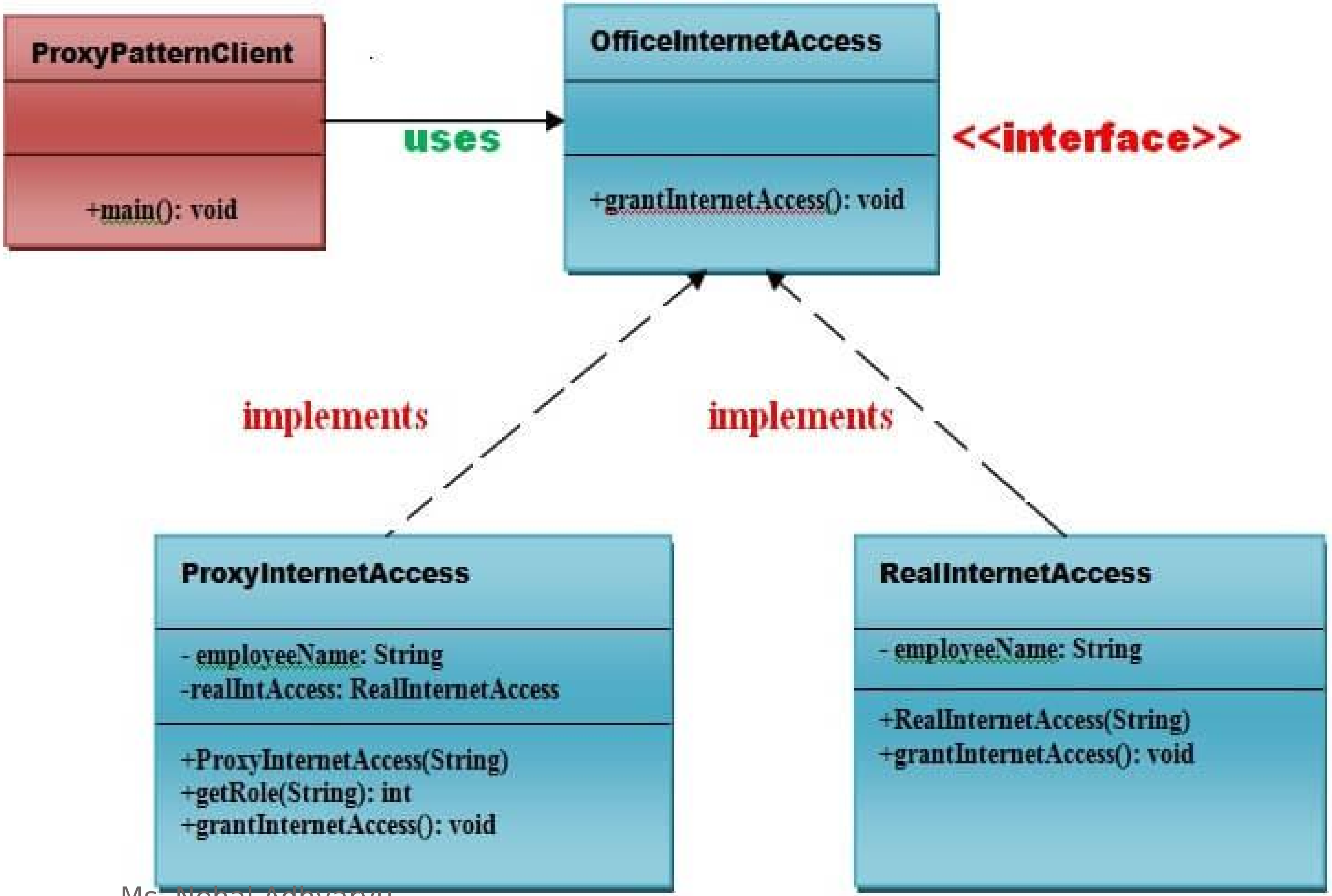
Proxy

Protection

It acts as an authorization layer to verify that whether the actual user has access the appropriate content or not. For example, a proxy server which provides restriction on internet access in office.

Smart

A smart proxy provides additional layer of security by interposing specific actions when the object is accessed. For example, to check whether the real object is locked or not before accessing it so that no other objects can change it.


ProxyPatternClient
+main(): void
uses
OfficeInternetAccess
+grantInternetAccess(): void
<<interface>>
implements
implements
ProxyInternetAccess
- employeeName: String
-realIntAccess: RealInternetAccess
+ProxyInternetAccess(String)
+getRole(String): int
+grantInternetAccess(): void
RealInternetAccess
- employeeName: String
+RealInternetAccess(String)
+grantInternetAccess(): void