PREGRADO

UNIDAD 2: SOFTWARE FEATURE LEVEL DESIGN & PATTERNS

GOF STRUCTURAL DESIGN PATTERNS

SI720 | Diseño y Patrones de Software

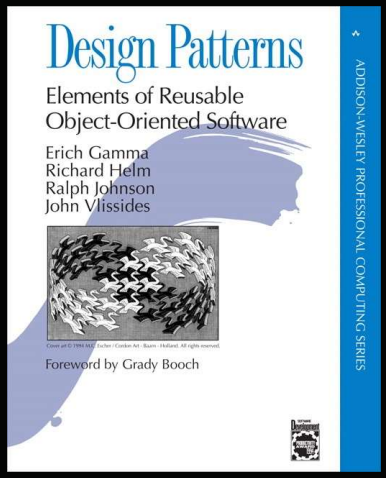
Al finalizar la unidad, el estudiante elabora y comunica artefactos de diseño de software aplicando principios básicos y patrones de diseño para un dominio y contexto determinados

AGENDA

INTRO
ADAPTER
BRIDGE
DECORATOR
FAÇADE
PROXY
OTHERS

GoF design patterns

Creational	Structural	Behavioral
Builder	Adapter	Chain of responsibility
Factory Method	Bridge	Command
Prototype	Composite	Interpreter
Singleton	Decorator	Iterator
	Façade	Mediator
	Flyweight	Memento
	Proxy	Observer
		States
		Strategy
		Template Method
		Visitor



Structural Design Patterns

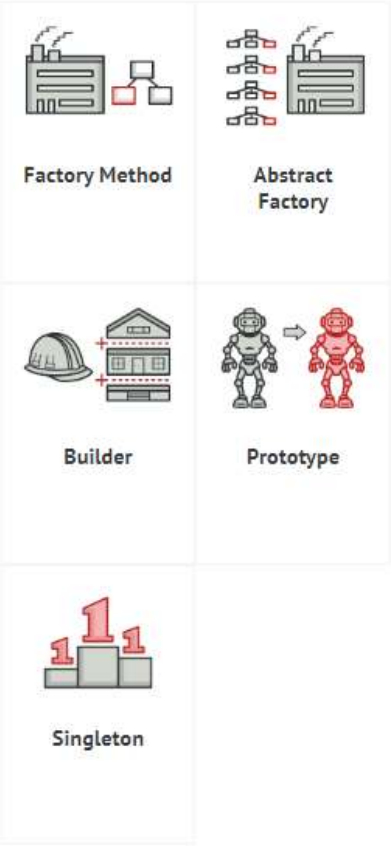
- Se centran en cómo las clases y los objetos se componen para formar estructuras mayores.
- Simplifican la estructura en base la identificación de relaciones.
- Se enfocan en cómo las clases descienden unas de otras y cómo éstas están compuestas por otras clases.

Structural Design Patterns

Pattern	Description
Adapter	Converts the interface of a class into another interface that a client wants. This Pattern provides the interface according to client requirement while using the services of a class with a different interface. It is member of Wrapper Patterns.
Bridge	Decouples the functional abstraction from the implementation so that the two can vary independently. It is also known as Handle or Body.
Composite	Allows clients to operate in generic manner on objects that may or may not represent a hierarchy of objects. It provides flexibility of structure with manageable class or interface.
Decorator	Attachs a flexible additional responsibilities to an object dynamically. This Pattern uses composition instead of inheritance to extend the functionality of an object at runtime. It is member of Wrapper Patterns.
Façade	Provides a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client. It describes a higher-level interface that makes the sub-system easier to use. Practically, every Abstract Factory is a type of Façade.
Flyweight	Reuses already existing similar kind of objects by storing them and create new object when no matching object is found. It reduces the number of objects, the amount of memory, and storage devices required if the objects are persisted.
Proxy	Represents another object and provides the control for accessing the original object. We can perform many operations like hiding the information of original object, on demand loading etc.

Patrones de Creación

Mecanismos de creación de objetos, incrementan flexibilidad y reutilización de código

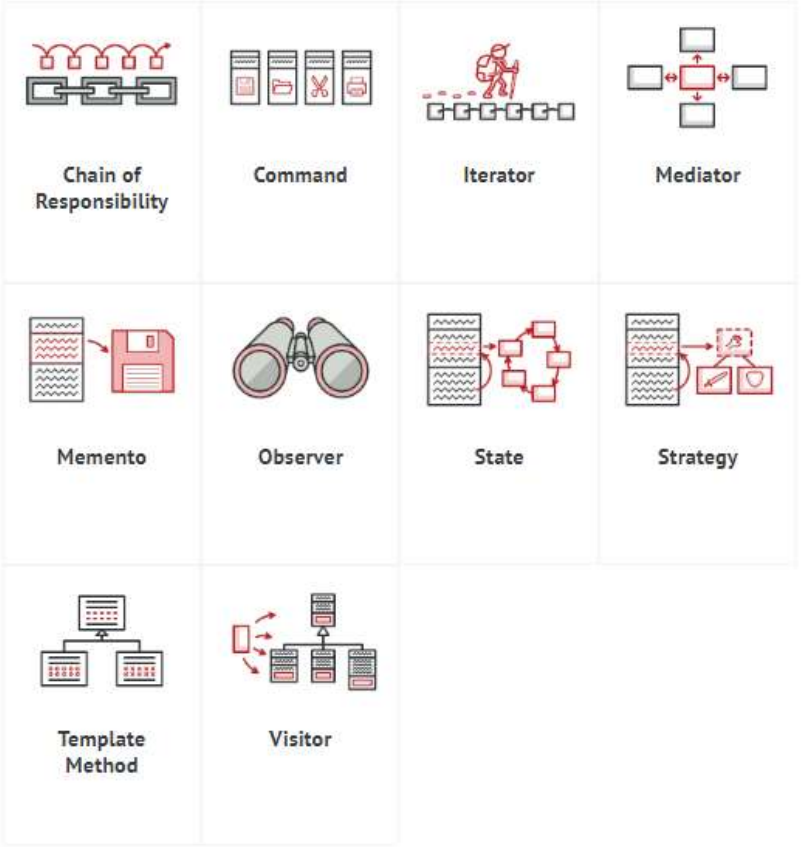


Patrones de Estructura

Facilitan la organización de objetos y clases en estructura mas grandes, manteniendo la estructura flexible y eficiente

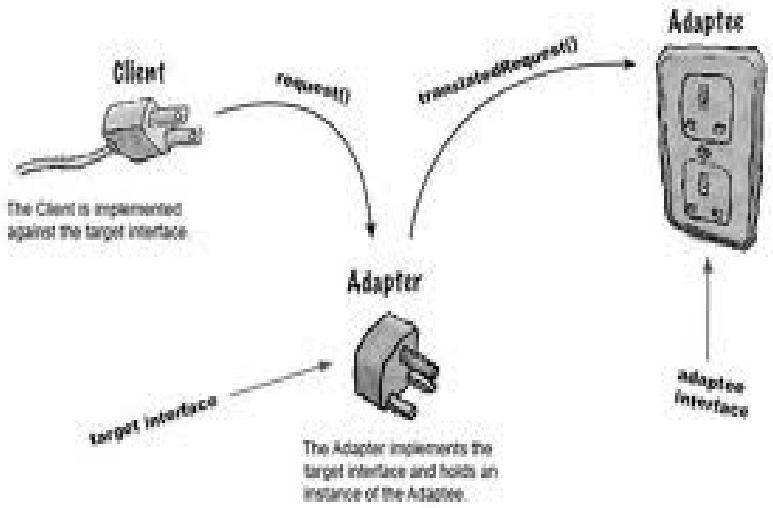
Patrones de Comportamiento

Facilitan el manejo de algoritmos y asignación de responsabilidades entre objetos



AGENDA

INTRO
ADAPTER
BRIDGE
DECORATOR
FAÇADE
PROXY
OTHERS



Structural Design Patterns
Adapter

Convierte la interfaz de una clase en otra que es la que esperan los clientes

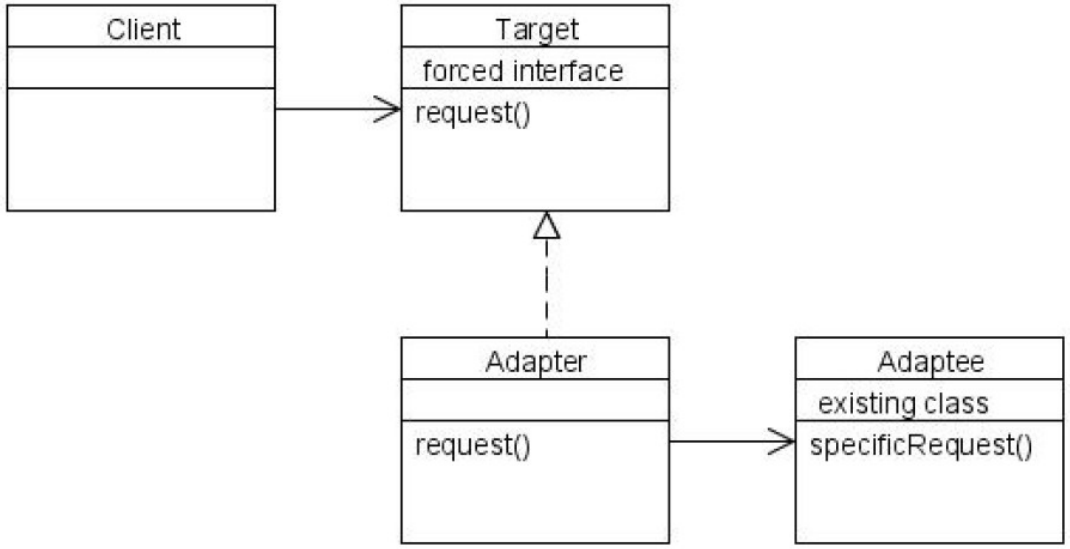
Adapter

Aplicabilidad

- Cuando se requiere **utilizar una clase existente y su interface no calza** con la que necesitas
- Cuando requieres crear una clase reutilizable que coopere con otras no relacionadas, no necesariamente van a tener interfaces compatibles
- Cuando se requiere incrementar la transparencia de las clases
- Cuando se requiere hacer un **kit de conexión**

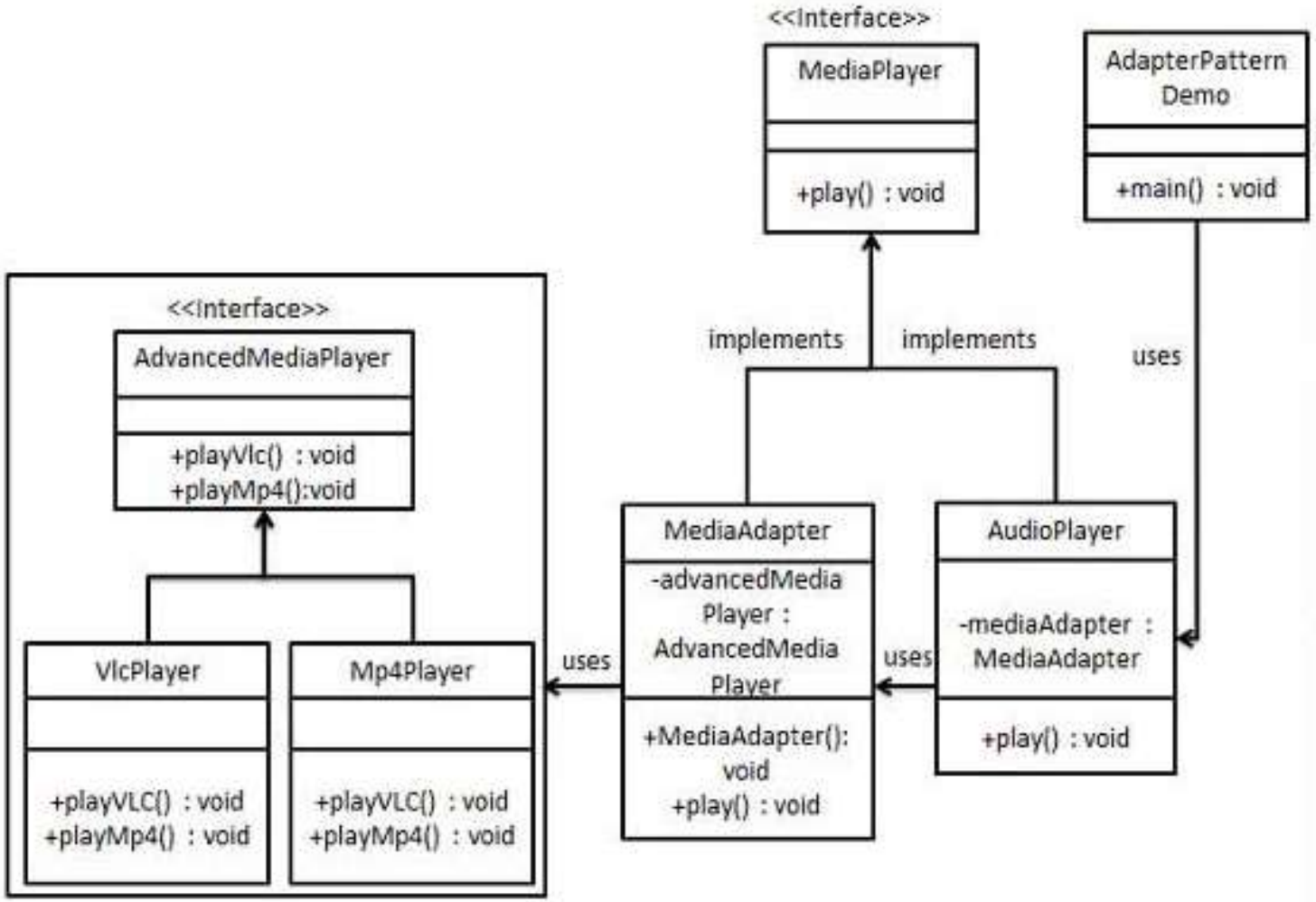
Adapter

- Cliente espera una interface específica (llamada interface destino)
- Una interface disponible no calza con la interface destino
- Adapter hace el puente entre la interface destino y la interface disponible
- La interface disponible es llamada Adaptee



Adapter

Puente entre 2 interfaces incompatibles



Ejemplo

AdvancedMediaPlayer.java

```
public interface AdvancedMediaPlayer {
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
```

VlcPlayer.java

```
public class VlcPlayer implements AdvancedMediaPlayer{
   @Override
   public void playVlc(String fileName) {
      System.out.println("Playing vlc file. Name: "+ fileName);
   }

   @Override
   public void playMp4(String fileName) {
      //do nothing
   }
}
```

Mp4Player.java

```
public class Mp4Player implements AdvancedMediaPlayer{

   @Override
   public void playVlc(String fileName) {
      //do nothing
   }

   @Override
   public void playMp4(String fileName) {
      System.out.println("Playing mp4 file. Name: "+ fileName);
   }
}
```

Ejemplo

MediaPlayer.java

```
public interface MediaPlayer {
   public void play(String audioType, String fileName);
}
```

MediaAdapter.java

```
public class MediaAdapter implements MediaPlayer {

   AdvancedMediaPlayer advancedMusicPlayer;

   public MediaAdapter(String audioType){

      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();

      }else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }

   @Override
   public void play(String audioType, String fileName) {

      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }
      else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
}
```

AdapterPatternDemo.java

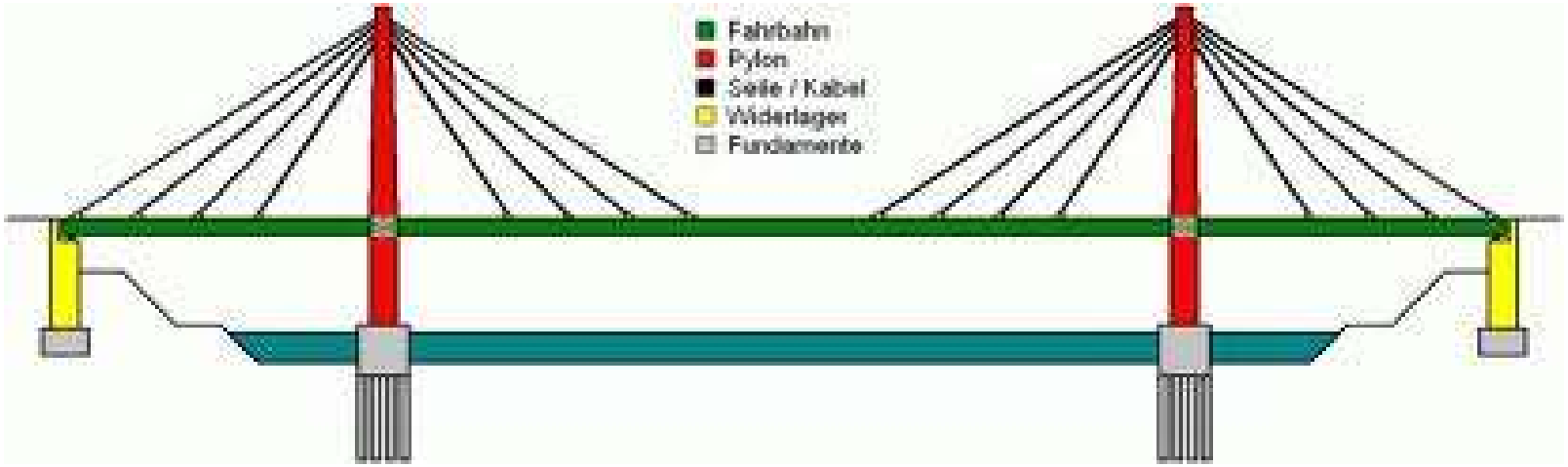
```
public class AdapterPatternDemo {
   public static void main(String[] args) {
      AudioPlayer audioPlayer = new AudioPlayer();

      audioPlayer.play("mp3", "beyond the horizon.mp3");
      audioPlayer.play("mp4", "alone.mp4");
      audioPlayer.play("vlc", "far far away.vlc");
      audioPlayer.play("avi", "mind me.avi");
   }
}
```

AudioPlayer.java

```
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;

   @Override
   public void play(String audioType, String fileName) {

      //inbuilt support to play mp3 music files
      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: " + fileName);
      }

      //mediaAdapter is providing support to play other file formats
      else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      }

      else{
         System.out.println("Invalid media. " + audioType + " format not supported");
      }
   }
}
```

AGENDA

INTRO
ADAPTER
BRIDGE
DECORATOR
FAÇADE
PROXY
OTHERS



Structural Design Patterns
Bridge

Desacoplar una abstracción de su implementación para que ambas varíen de forma independiente

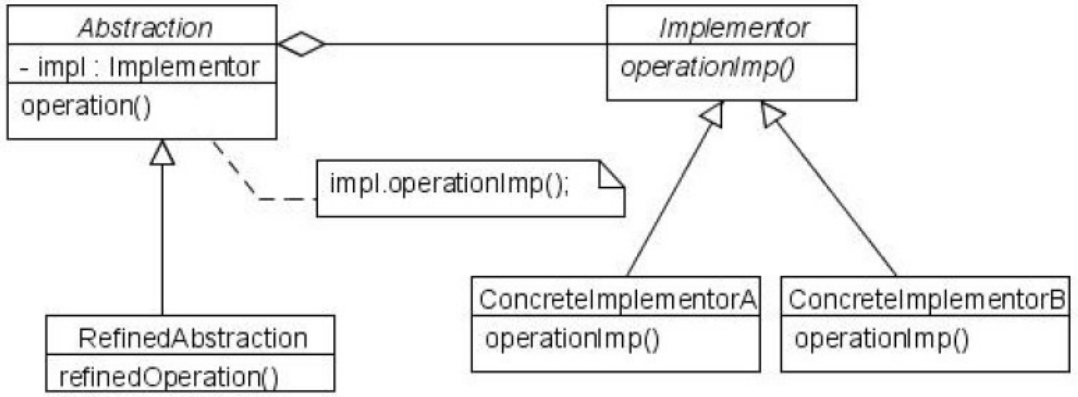
Bridge

Aplicabilidad

- Cuando se requiere separar la estructura abstracta y su implementación concreta.
- Cuando se requiere compartir una implementación entre múltiples objetos.
- Cuando se requiere reutilizar recursos existentes en una forma “fácil de extender”.
- Cuando se requiere ocultar los detalles de la implementación a los clientes. El cambio en la implementación no impacta a los clientes.

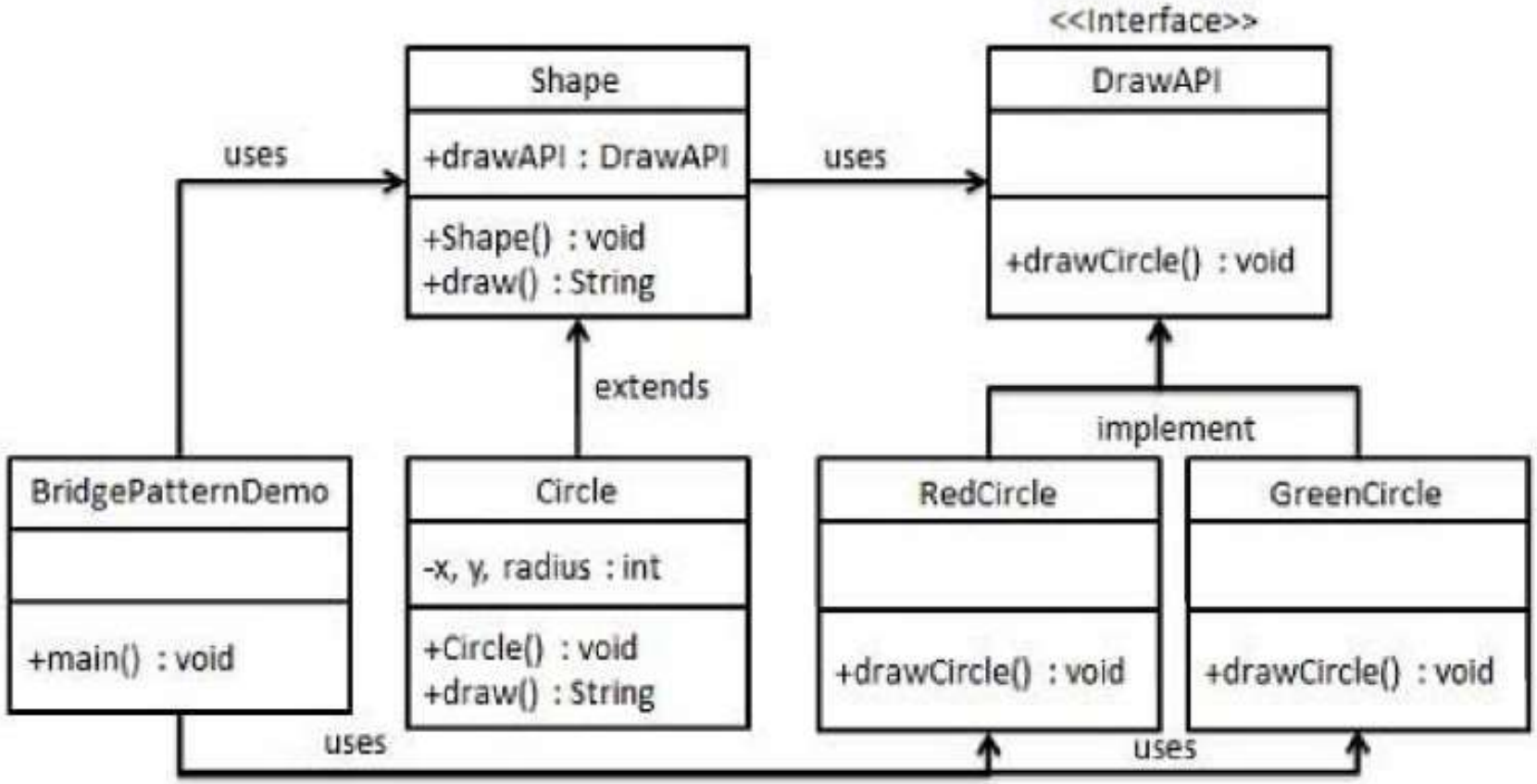
Bridge

- **Abstraction** define la interface abstracta y mantiene la referencia a **Implementor.**
- **Refined Abstraction** extiende la interface definida por **Abstraction**.
- **Implementor** define la interfaz para la implementación de las clases
- **ConcreteImplementor** implementa la interfaz **Implementor**



Bridge

Desacoplar una abstracción de su implementación para que ambas varíen de forma independiente



Bridge

DrawAPI.java

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

RedCircle.java

```
public class RedCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: red, radius: " + radius + ", x: " + x
   }
}
```

GreenCircle.java

```
public class GreenCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: green, radius: " + radius + ", x: " + x
   }
}
```

Bridge

Shape.java

```
public abstract class Shape {
   protected DrawAPI drawAPI;

   protected Shape(DrawAPI drawAPI){
      this.drawAPI = drawAPI;
   }
   public abstract void draw();
}
```

Circle.java

```
public class Circle extends Shape {
   private int x, y, radius;

   public Circle(int x, int y, int radius, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
      this.radius = radius;
   }

   public void draw() {
      drawAPI.drawCircle(radius,x,y);
   }
}
```

BridgePatternDemo.java

```
public class BridgePatternDemo {
   public static void main(String[] args) {
      Shape redCircle = new Circle(100,100, 10, new RedCircle());
      Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

      redCircle.draw();
      greenCircle.draw();
   }
}
```

AGENDA

INTRO
ADAPTER
BRIDGE
DECORATOR
FAÇADE
PROXY
OTHERS

Structural Design Patterns
Decorator

Permite **adjuntar responsabilidades adicionales** y modificar el funcionamiento de una instancia **dinámicamente**.

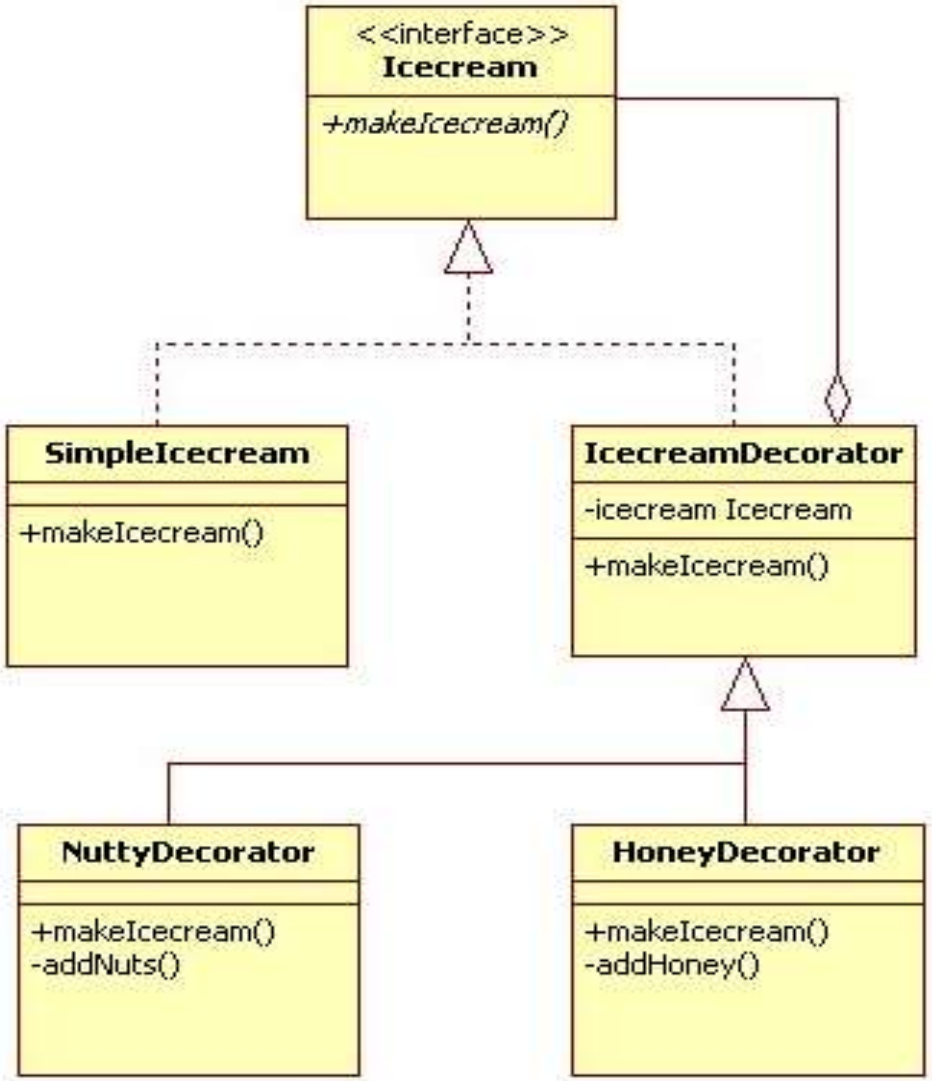
Alternativa flexible a la herencia para extender la funcionalidad

Decorator

Aplicabilidad

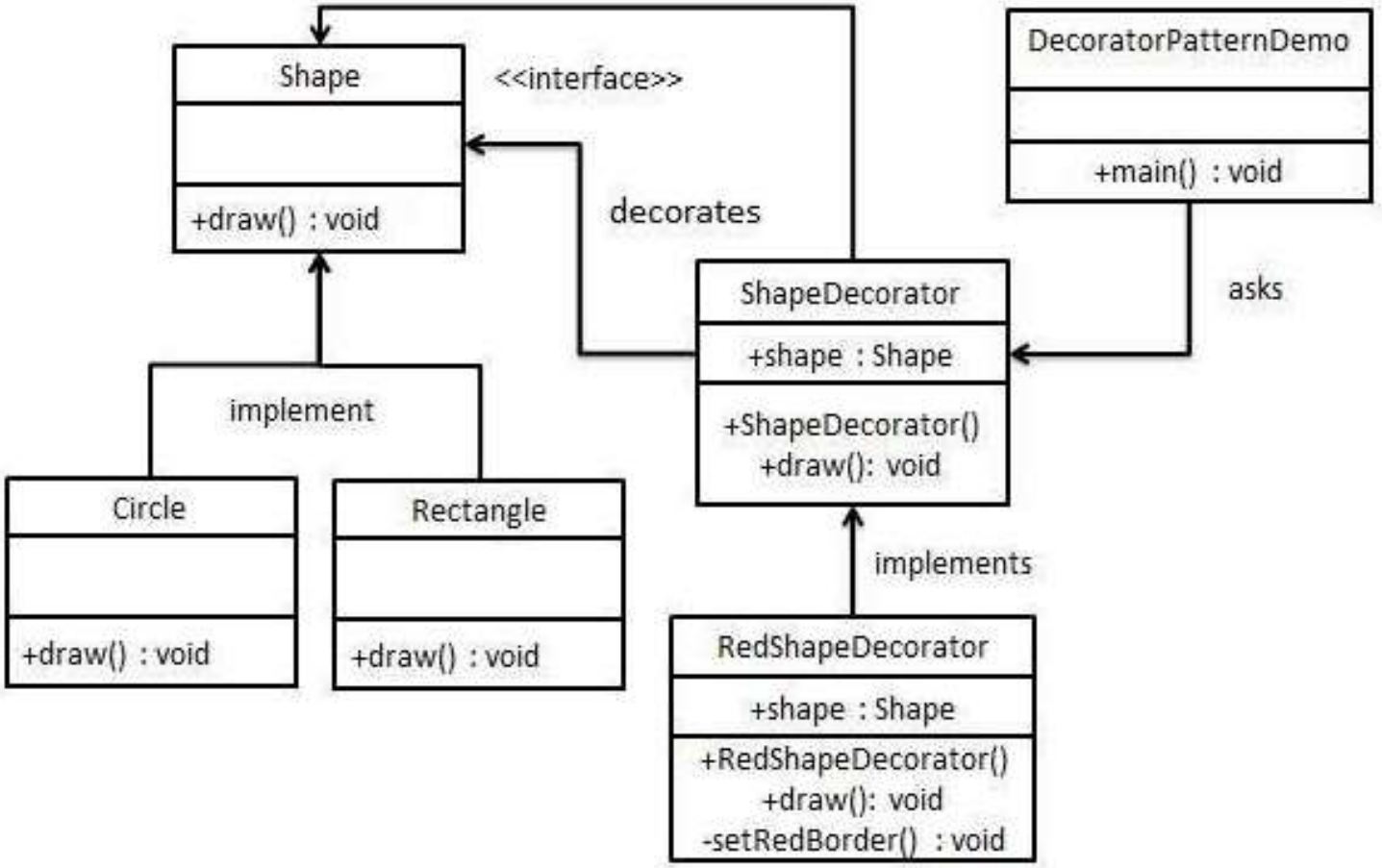
- Cuando se requiere añadir responsabilidades a objetos individuales de forma dinámica y transparente sin afectar el objeto original u otros objetos
- Cuando se requiere añadir responsabilidades a objetos que se podría querer cambiar en un futuro
- Cuando la extensión por herencia no es práctica

Decorator


<<interface>>
Icecream
+makeIcecream()
SimpleIcecream
+makeIcecream()
IcecreamDecorator
-icecream Icecream
+makeIcecream()
NuttyDecorator
+makeIcecream()
-addNuts()
HoneyDecorator
+makeIcecream()
-addHoney()

Decorador

Agregar funcionalidad a un objeto existente



Decorador

Shape.java

```
public interface Shape {
   void draw();
}
```

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Circle");
   }
}
```

ShapeDecorator.java

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

RedShapeDecorator.java

```
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

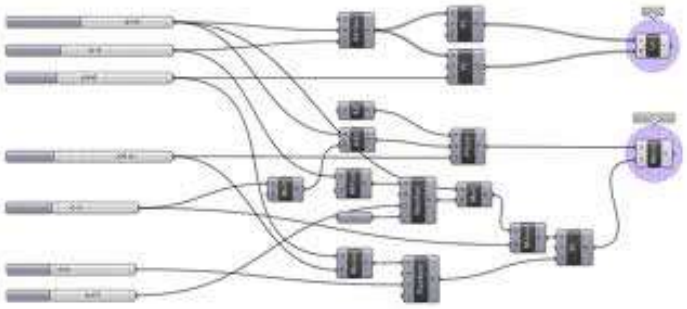
Decorador

DecoratorPatternDemo.java

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

      Shape circle = new Circle();

      Shape redCircle = new RedShapeDecorator(new Circle());

      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      System.out.println("Circle with normal border");
      circle.draw();

      System.out.println("\nCircle of red border");
      redCircle.draw();

      System.out.println("\nRectangle of red border");
      redRectangle.draw();
   }
}
```

AGENDA

INTRO
ADAPTER
BRIDGE
DECORATOR
FAÇADE
PROXY
OTHERS



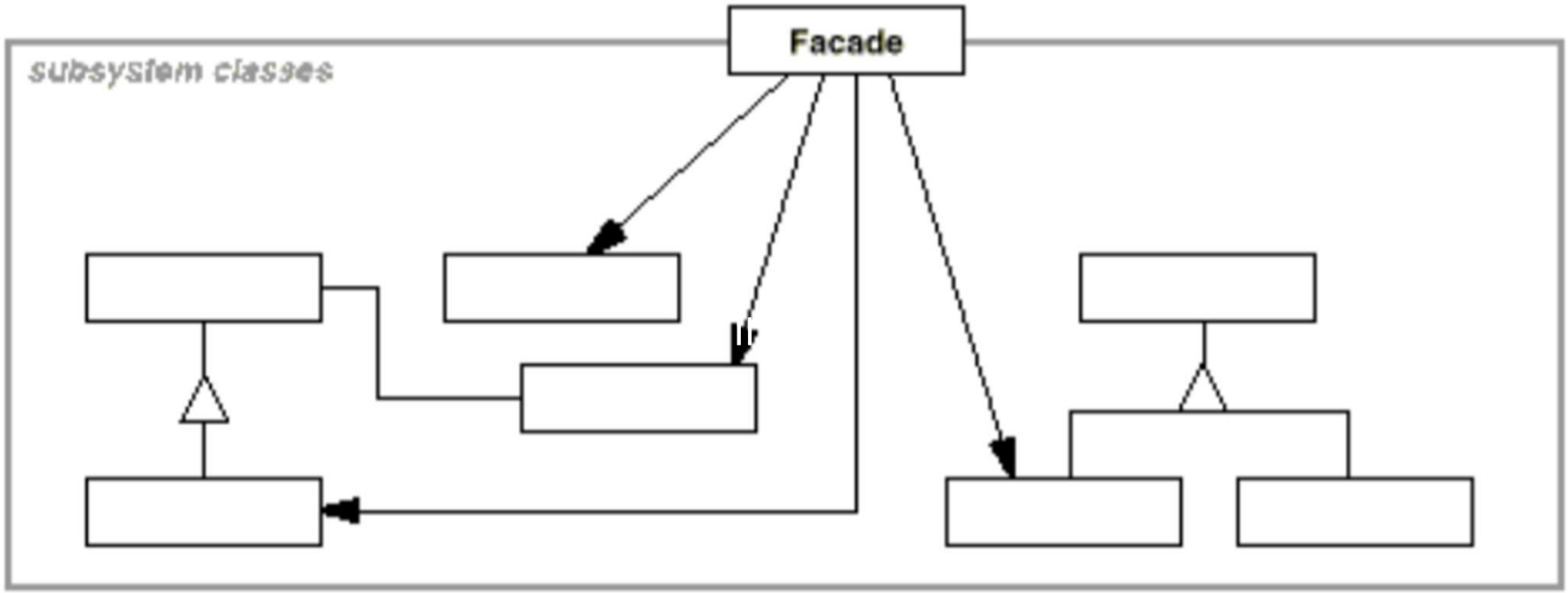
Structural Design Patterns
Façade

Provee una interfaz única para un conjunto de interfaces dentro de un subsistema.

Define una interfaz de nivel superior que hace que el uso del subsistema sea mas fácil.

Façade

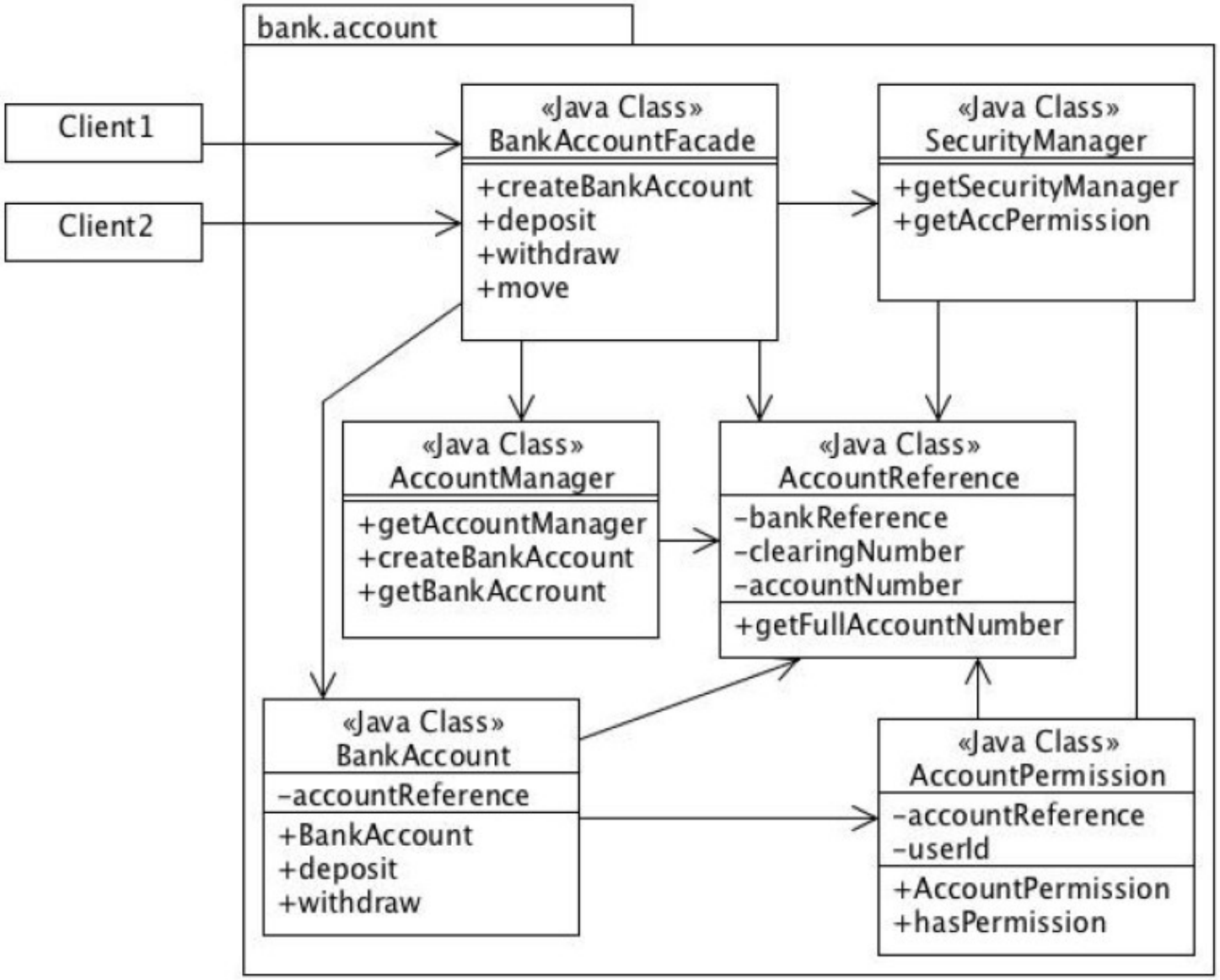
- **Aplicabilidad**
  - Proporcionar **una interfaz simple para un subsistema** complejo
  - Cuando hay muchas dependencias entre los clientes y las clases del subsistema
  - Queremos dividir en capa nuestros subsistemas



Façade

Structure

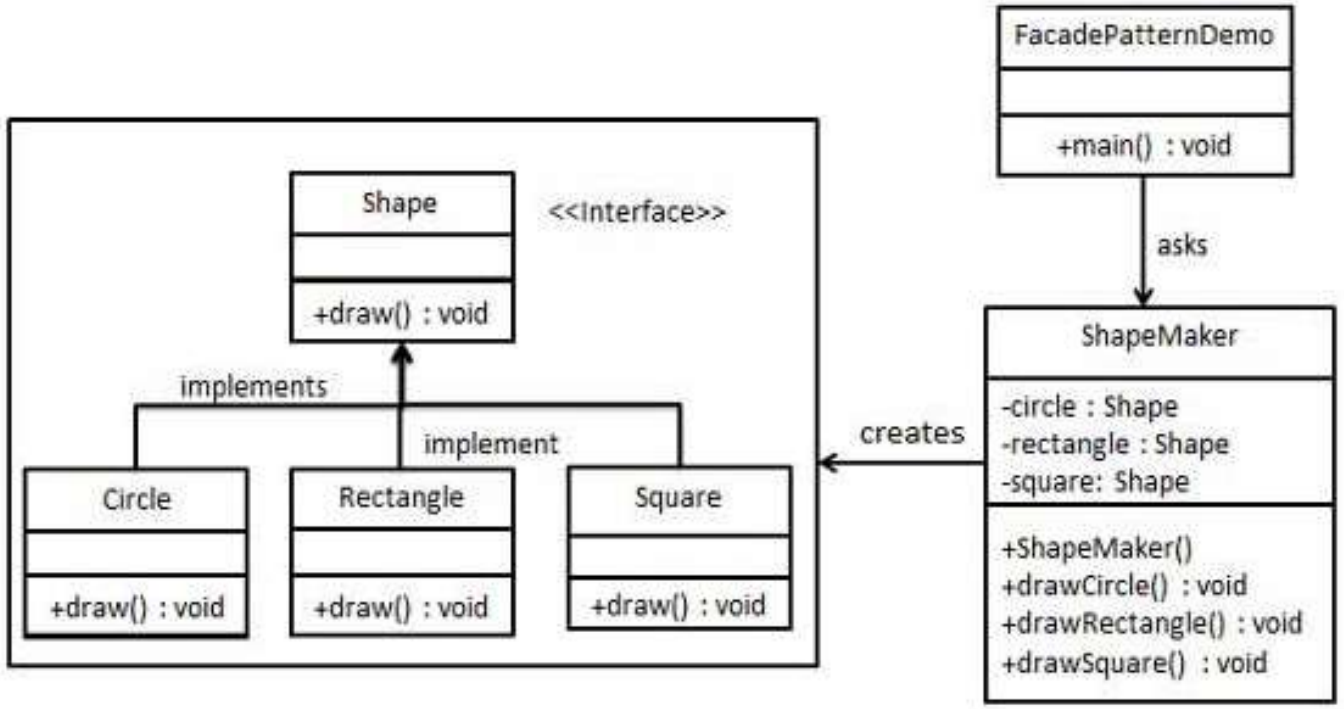
Façade


bank.account
Client1
Client2
«Java Class»
BankAccountFacade
+createBankAccount
+deposit
+withdraw
+move
«Java Class»
SecurityManager
+getSecurityManager
+getAccPermission
«Java Class»
AccountManager
+getAccountManager
+createBankAccount
+getBankAccrount
«Java Class»
AccountReference
-bankReference
-clearingNumber
-accountNumber
+getFullAccountNumber
«Java Class»
BankAccount
-accountReference
+BankAccount
+deposit
+withdraw
«Java Class»
AccountPermission
-accountReference
-userId
+AccountPermission
+hasPermission

Façade

Facade: oculta complejidad de otro sistema e implementa una interfaz.

Provee una interfaz única para un conjunto de interfaces dentro de un subsistema. Define una interfaz de nivel superior que hace que el uso del subsistema sea mas fácil.



Façade

Shape.java

```
public interface Shape {
   void draw();
}
```

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Rectangle::draw()");
   }
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Square::draw()");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Circle::draw()");
   }
}
```

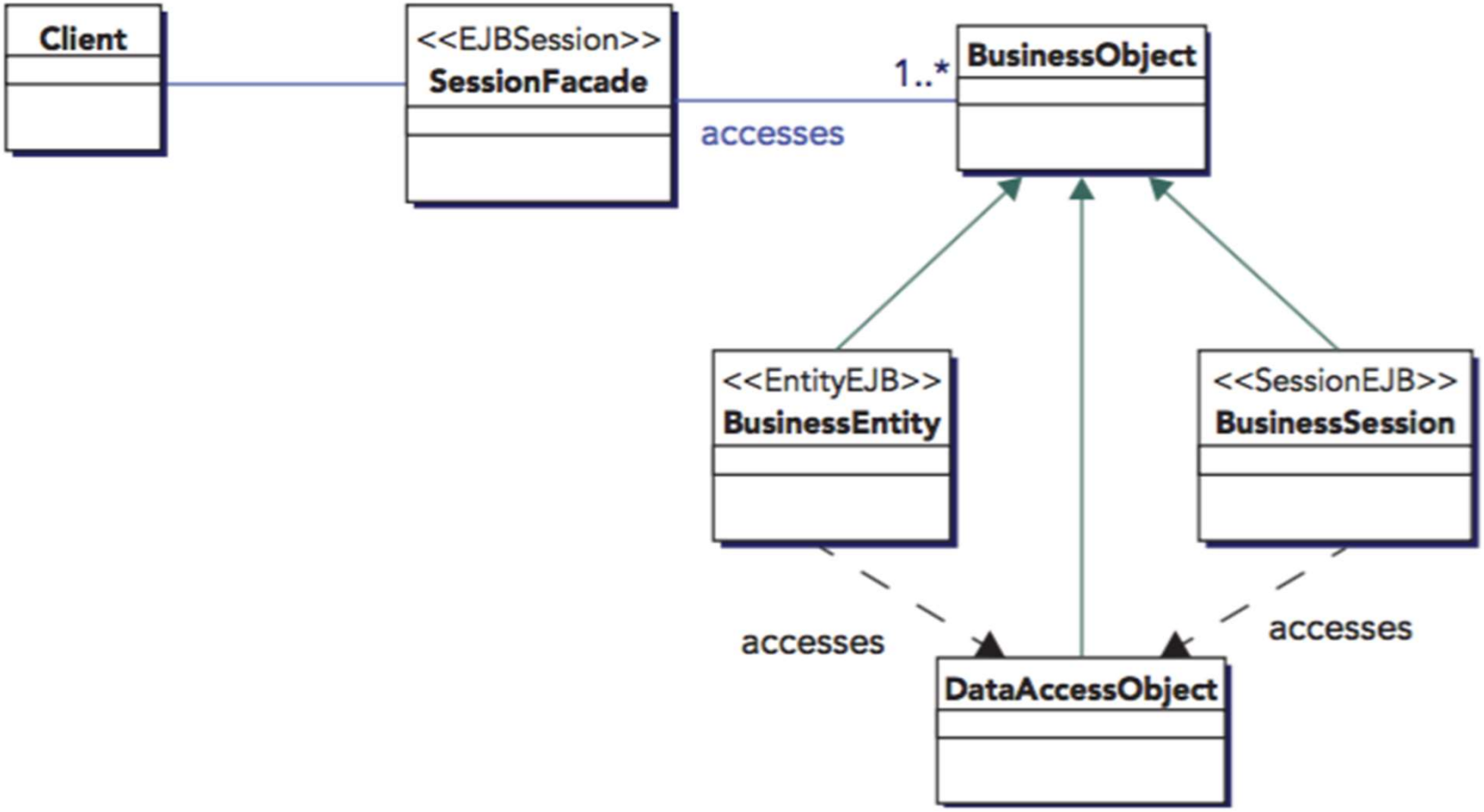
Façade

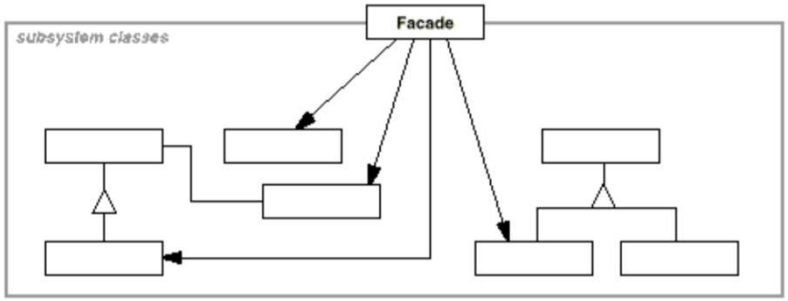
FacadePatternDemo.java

```
public class FacadePatternDemo {
   public static void main(String[] args) {
      ShapeMaker shapeMaker = new ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();
   }
}
```

ShapeMaker.java

```
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }

   public void drawCircle(){
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
}
```





Façade

Sample

```
package com.designpatterns.facade;

public class WashingMachine {
  public void heavilySoiled() {
    setWaterTemperature(100);
    setWashCycleDuration(90);
    setSpinCycleDuration(10);
    addDetergent();
    addBleach();
    addFabricSoftener();
    heatWater();
    startWash();
  }
  public void lightlySoiled() {
    setWaterTemperature(40);
    setWashCycleDuration(20);
    setSpinCycleDuration(10);
    addDetergent();
    heatWater();
    startWash();
  }
}
```

Façade

Sample

```
// to use the façade
new WashingMachine().lightlySoiled();
```

```
package com.designpatterns.facade;
import javax.ejb.Stateless;
@Stateless
public class CustomerService {
  public long getCustomer(int sessionID) {
    // get logged in customer id
    return 100005L;
  }
    public boolean checkId(long x) {
    // check if customer id is valid
    return true;
  }
}
```

```
package com.designpatterns.facade;
import javax.ejb.Stateless;
@Stateless
public class LoanService {
  public boolean checkCreditRating(long id, double amount) {
    // check if customer is eligible for the amount
    return true;
  }
}
```

```
package com.desisgnpatterns.facade;
import javax.ejb.Stateless;
@Stateless
public class AccountService {
  public boolean getLoan(double amount) {
    // check if bank vault has enough return true;
  }
  public boolean setCustomerBalance(long id, double amount) {
    // set new customer balance
    return true;
  }
}
```

Façade

Sample


```
package com.designpatterns.facade;
import javax.ejb.Stateless;
import javax.inject.Inject;

@Stateless
public class BankServiceFacade {
    @Inject
    CustomerService customerService;
    @Inject
    LoanService loanService;
    @Inject
    AccountService accountService;
    public boolean getLoan(int sessionId, double amount) {
        boolean result = false;
        long id = customerService.getCustomer(sessionId);
        if(customerService.checkId(id)){
            if(loanService.checkCreditRating(id, amount)){
                if(accountService.getLoan(amount)){
                    result = accountService.setCustomerBalance(id, amount);
                }
            }
        }
        return result;
    }
}
```

Façade

Sample

```ruby
# app/facades/dashboard.rb
class Dashboard
  def initialize(user)
    @user = user
  end
  def new_status
    @new_status ||= Status.new
  end
  def statuses
    Status.for(user)
  end
  def notifications
    @notifications ||= user.notifications
  end
  private
  attr_reader :user
end
```

Façade

Sample

```
# app/controllers/dashboards_controller.rb
class DashboardsController < ApplicationController
  before_filter :authorize

  def show
    @dashboard = Dashboard.new(current_user)
  end
end
```

```
# app/views/dashboards/show.html.erb
<%= render 'profile' %>
<%= render 'groups', groups: @dashboard.group %>
<%= render 'statuses/form', status: @dashboard.new_status %>
<%= render 'statuses', statuses: @dashboard.statuses %>
```

Façade

Sample

AGENDA

INTRO
ADAPTER
BRIDGE
DECORATOR
FAÇADE
PROXY
OTHERS

Structural Design Patterns
Proxy

Proporciona un representante o sustituto de otro objeto para controlar el acceso a éste.

Proxy

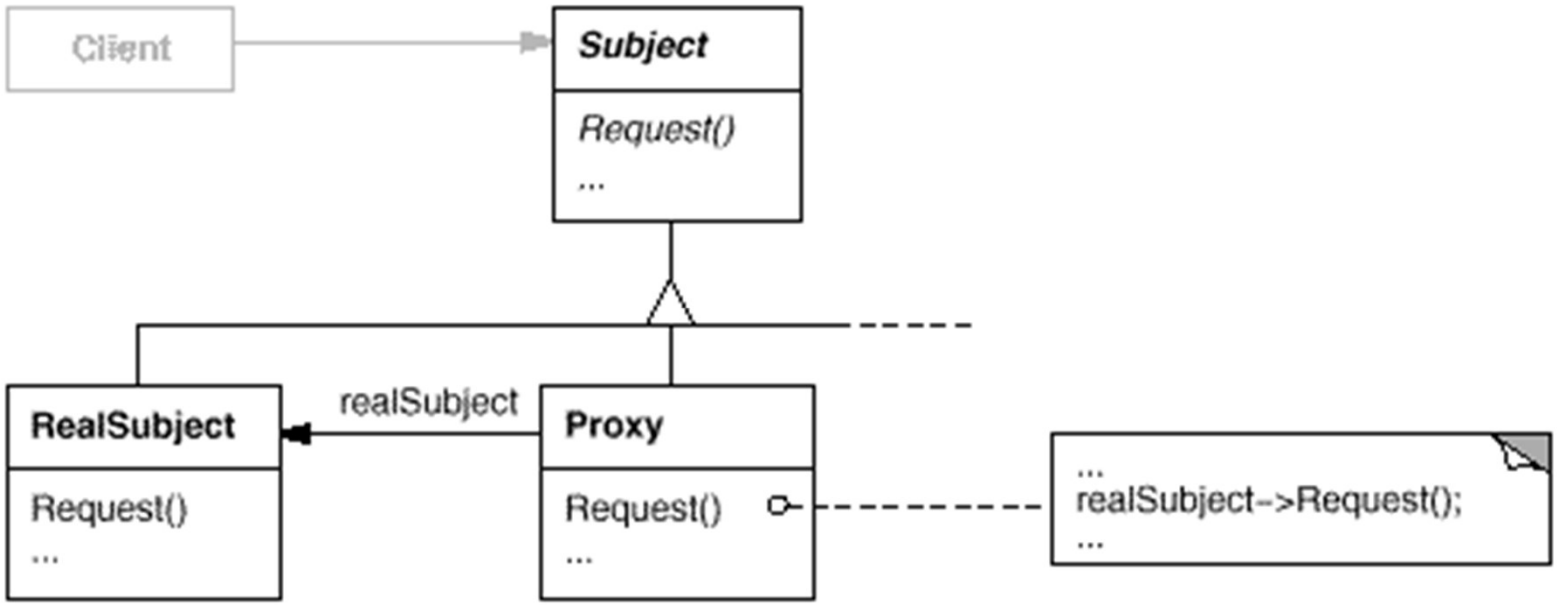
Aplicabilidad

- Cuando la creación de un **objeto es relativamente costoso** puede ser buena idea **remplazarlo con un proxy** que asegure que la instancia del objeto costoso se mantenga al mínimo

- Su implementación permite realizar el login y el chequeo de autorizaciones antes de acceder al objeto requerido

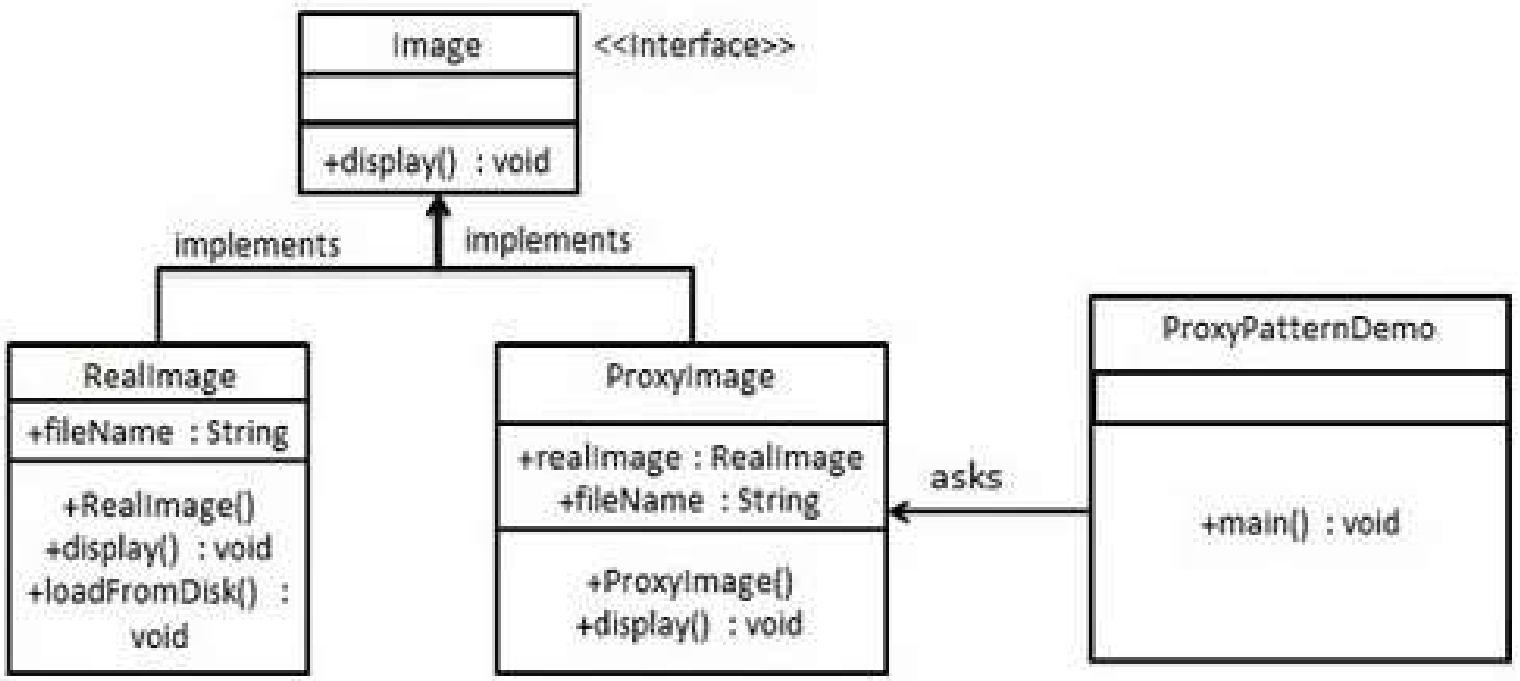
- Puede proveer una implementación local para un objeto remoto

Proxy


Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...

Proxy

Proporciona un representante o sustituto de otro objeto para controlar el acceso a éste.



Proxy

Image.java

```
public interface Image {
   void display();
}
```

RealImage.java

```
public class RealImage implements Image {

   private String fileName;

   public RealImage(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   @Override
   public void display() {
      System.out.println("Displaying " + fileName);
   }

   private void loadFromDisk(String fileName){
      System.out.println("Loading " + fileName);
   }
}
```

Proxy

```
ProxyImage.java
public class ProxyImage implements Image{

   private RealImage realImage;
   private String fileName;

   public ProxyImage(String fileName){
      this.fileName = fileName;
   }

   @Override
   public void display() {
      if(realImage == null){
         realImage = new RealImage(fileName);
      }
      realImage.display();
   }
}
```

```
ProxyPatternDemo.java
public class ProxyPatternDemo {

   public static void main(String[] args) {
      Image image = new ProxyImage("test_10mb.jpg");

      //image will be loaded from disk
      image.display();
      System.out.println("");

      //image will not be loaded from disk
      image.display();
   }
}
```

AGENDA

INTRO
ADAPTER
BRIDGE
DECORATOR
FAÇADE
PROXY
OTHERS

Structural Design Patterns

Otros

Composite

Ayuda a crear estructuras de árbol de objetos sin necesidad de obligar a los clientes a diferenciar entre las ramas y las hojas con respecto a su uso.

Permite a los clientes tratar objetos individuales y composiciones de objetos uniformemente.

Flyweight

Proporciona un mecanismo mediante el cual se puede evitar la creación de un gran número de objetos “costosos” para reutilizar en su lugar, instancias existentes para representar a los nuevos.

RESUMEN

Recordemos

Los **Design Patterns** describen cómo resolver problemas recurrentes de diseño de software orientado a objetos flexible y reutilizable.

Los tipo de patrones son: **Creational**, **Structural** y **Behavioral**.

Los **Structural Design Patterns** facilitan la organización de objetos y clases en estructura mas grandes, manteniendo la estructura flexible y eficiente.

REFERENCIAS

Para profundizar

- Design Patterns- Libro de Erich Gamma, John Vlissides, Ralph Johnson y Richard Helm.
- http://www.blackwasp.co.uk/gofpatterns.aspx
- http://www.w3sdesign.com/

PREGRADO

Ingeniería de Software

Escuela de Ingeniería de Sistemas y Computación | Facultad de Ingeniería

UPC
Universidad Peruana
de Ciencias Aplicadas

Prolongación Primavera 2390,
Monterrico, Santiago de Surco
Lima 33 - Perú
T 511 313 3333
https://www.upc.edu.pe

exígete, innova