THINKOPEN

DESIGN PATTERNS STRUTTURALI

- Thinkopen Academy 2018-07-19

CHI SONO

Simone Celia, FullStack Java Developer, lavoro nell'ambito della consulenza informatica da 5 anni, in Thinkopen da 2

DOVE TROVARE QUESTE SLIDE

(e relativo codice eseguibile)

https://github.com/simon387/structural-patterns-presentation

RIFERIMENTI

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Design Patterns: Elements of Reusable Object Oriented Software“.
- https://www.docenti.unina.it/webdocenti-be/allegati/materiale-didattico/98381
- https://www.journaldev.com
- wikipedia
- stackoverflow

THINKOPEN

INIZIAMO!

UN PO' DI STORIA

- Introdotti dall'architetto Christopher Alexander. Architetto edile, non software!

> *"Ogni pattern descrive un problema che si presenta ferquentemente, e quindi descrive il nucleo della soluzione così che si possa impiegarla milioni di volte, senza produrre due volte la stessa realizzazione"*

UN PO' DI STORIA

- Il principio è ugualmente valido, anche se riferito ad oggetti, classi ed interfacce piuttosto che ad elementi architettonici come muri, archi, pilastri, ecc... .

UN PO' DI STORIA (CONTINUA)

- I design patterns hanno raggiunto molta popolarità nel mondo della **computer science** con il libro *Design Patterns: Elements of Reusable Object-Oriented Software* scritto dalla **Gang of Four**

PATTERN

individua un'**IDEA**, uno schema **GENERALE E RIUSABILE**

Non è paragonabile ad un componente riusabile perchè:

- non è un oggetto fisico
- deve essere contestualizzato all'interno del particolare problema applicativo

PATTERN (CONTINUA)

due istanze di uno stesso pattern (ad esempio applicate a problemi diversi) tipicamente sono diverse proprio per la contestualizzazione in domini differenti.

SCOPO DEI PATTERNS

- Catturare l'esperienza e la *saggezza* degli esperti
- Evitare di reinventare ogni volta le stesse cose

COSA FORNISCE UN DESIGN PATTERN AL PROGETTISTA SOFTWARE?

- Una soluzione codificata e consolidata per un problema ricorrente
- Un'astrazione di granualità e livello di astrazione più elevati di una classe

COSA FORNISCE UN DESIGN PATTERN AL PROGETTISTA SOFTWARE? (CONTINUA)

- Un supporto alla comunicazione delle caratteristiche del progetto
- Un modo per progettare software con caratteristiche predefinite
- Un supporto alla progettazione di sistemi complessi

DEFINIZIONE

Ogni pattern descrive un problema specifico che ricorre più volte e descrive il nucleo della soluzione a quel problema, in modo da poter utilizzare tale soluzione milioni di volte.

ABBASTANZA ASTRATTI

In modo da poter essere condivisi da progettisti con punti di vista diversi e che conoscono tecnologie e linguaggi diversi

NÉ COMPLESSI NÉ DOMAIN-SPECIFIC

Non sono rivolti alla specifica applicazione ma riusabili
in parti di applicazioni diverse

CARATTERISTICHE

Un Design Pattern

- nomina
- astrae
- identifica

gli aspetti chiave di una struttura comune di design che la rendono utile nel contesto del riuso in ambito object-oriented

CARATTERISTICHE (CONTINUA)

Un Design Pattern identifica:

- le classi (e le istante) partecipanti
- le associazioni ed i ruoli
- le modalità di collaborazione tra le classi coinvolte
- la distribuzione delle responsabilità nella soluzione del particolare problema di design considerato

TIPOLOGIE DI PATTERN

Esistono diverse tipologie di pattern, che si differenziano principalmente per la scala ed il livello di astrazione:

1. Architectural Pattern
2. **Design Pattern**
3. Idioms

ARCHITECTURAL PATTERN

- Utili per strutturare un sistema in sottosistemi

DESIGN PATTERN

- Operano essenzialmente a livello di un singolo sottosistema evidenziando le carattersistiche delle classi coinvolte e delle associazioni tra class

IDIOMS

- Utili per l'implementazione di specifici aspetti di design in un particolare linguaggio di programmazione
- *Common Practice*

COME SONO FATTI I DESIGN PATTERNS

- nome
- problema
- soluzione
- conseguenze

NOME

- il **nome** del pattern, è utile per descrivere la sua funzionalità in una o due parole.

PROBLEMA

- il **problema** nel quale il pattern è applicabile. Spiega il problema e il contesto, a volte descrive dei problemi specifici del design mentre a volte può descrivere strutture di classi e oggetti. Può anche includere una lista di condizioni che devono essere soddisfatte precedentemente perchè il pattern possa essere applicato

SOLUZIONE

- la **soluzione** che descrive in modo astratto come il pattern risolve il problema. Descrive gli elementi che compongono il design, le loro responsabilità e le collaborazioni

CONSEGUENZE

- le **conseguenze** portate dall'applicazione del pattern. Spesso sono tralasciate ma sono importanti per poter valutare i costi-benefici dell'utilizzo del pattern.

ESEMPIO DESCRIZIONE DP

- **Nome e classificazione** del pattern
- **Sinonimi**: altri nomi del pattern
- **Scopo**: cosa fa il pattern? a cosa serve?
- **Motivazione**: scenario che illustra un design problem
- **Applicabilità**: situazioni in cui si applica il pattern
- **Struttura**: rappresentazione delle classi in stile OMT
- **Partecipanti**: classi e oggetti inclusi nel pattern

ESEMPIO DESCRIZIONE DP (CONTINUA)

- **Collaborazioni**: come i partecipanti collaborano
- **Conseguenze**: come consegue i suoi obiettivi il pattern?
- **Implementazione**: che tecniche di codifica sono necessarie?
- **Codice di esempio**: scritto in un linguaggio ad oggetti
- **Usi noti**: esempi d'applicazione del pattern in sistemi reali
- **Pattern correlati**: con quali pattern si dovrebbe usare?

CATEGORIE DI PATTERN

Esistono diverse categorie di pattern, spesso sono divisi per funzione (purpose) e dominio (scope) del pattern

FUNZIONE (PURPOSE), OVVERO COSA FA IL PATTERN

- **Creazionali**: forniscono meccanismi per la creazione di oggetti
- **Strutturali**: gestiscono la separazione tra interfaccia e implementazione e le modalità di composizione tra oggetti
- **Comportamentali**: consentono la modifica del comportamento degli oggetti minimizzando la necessità di cambiare il codice

DOMINIO (SCOPE), INDICA SE IL PATTERN SI APPLICA A CLASSI O OGGETTI

- **Class pattern**: si focalizzano su relazioni fra classi e sottoclassi. Tipicamente si riferiscono a situazioni statiche, ovvero riguadano il compile-time
- **Object pattern**: si focalizzano su oggetti (istanze) e le loro relazioni. Tipicamente si riferiscono a situazioni dinamiche (run-rime).

CATEGORIA CREATIONAL

la vedrete domani ¯\(ツ)/¯

CATEGORIA STRUCTURAL

- sono dedicati alla composizione di classi e oggetti per creare delle strutture più grandi
- è possibile creare delle classi che ereditano da più classi per consentire di utilizzare proprietà di più superclassi indipendenti
- ad esempio permettono di far funzionare insieme delle librerie indipendenti

ELENCO PATTERN STRUCTURAL GOF

DP	Descrizione
Adapter	Converte l'interfaccia di una classe in un'altra permettendo a due classi di lavorare assieme anche se hanno interfacce diverse
Bridge	Disaccoppia un'astrazione dalla sua implementazione in modo che possano variare in modi indipendente

DP	Descrizione
Composite	Compone oggetti in strutture ad albero per implementare delle composizioni ricorsive
Decorator	Aggiunge nuove responsabilità ad un oggetto in modo dinamico, è alternativa alle sottoclassi per estenderne le funzionalità
Facade	Provvede un'interfaccia unificata per le interfacce di un sottosistema in modo da rendere più facile il loro utilizzo

DP	Descrizione
Proxy	Provvede un surrogato di un oggetto per controllarne gli accessi
Flyweight	Usa la condivisione per supportare in modo efficiente un gran numero di oggetti con fine granualità

ADAPTER

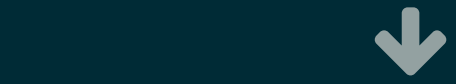
- è usato per far lavorare assieme interfacce non correlate.
- l'oggetto che unisce queste interfacce è chiamato **Adapter**

ADAPTER

Un esempio di Adapter nel mondo reale potrebbe essere quello dei carica batterie per cellulare. Una batteria potrebbe avere bisogno di 3 Volt per caricarsi mentre la normale presa di corrente ne ha 220. Quindi il carica batterie fa da *Adapter* tra la presa di corrente e la batteria.

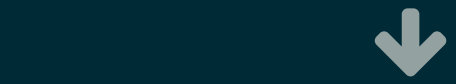
Volt.java

```
01  package adapter;
02
03  public class Volt {
04
05      private int volts;
06
07      public Volt(int volts) {
08          this.volts = volts;
09      }
10
11      public int getVolts() {
12          return volts;
13      }
14
15      public void setVolts(int volts) {
16          this.volts = volts;
```



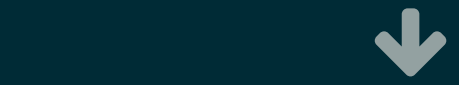
Socket.java

```
01  package adapter;
02
03  public class Socket {
04
05      public Volt getVolt() {
06          return new Volt(240);
07      }
08  }
```



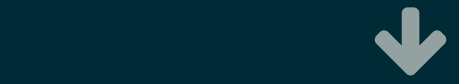
SocketAdapter.java

```
01  package adapter;
02
03  public interface SocketAdapter {
04
05      public Volt get240Volt();
06
07      public Volt get12Volt();
08
09      public Volt get3Volt();
10  }
```



SocketClassAdapterImpl.java

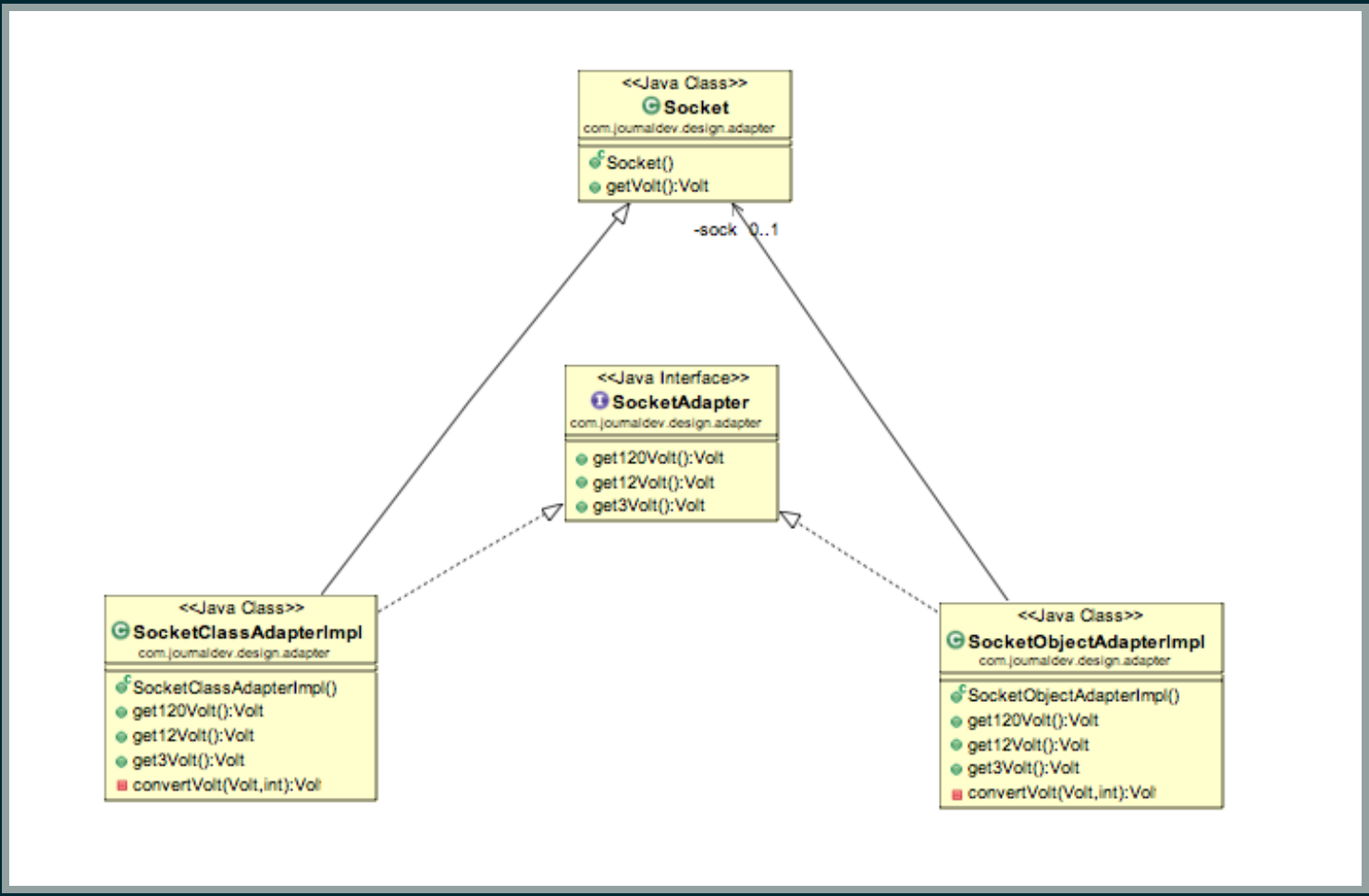
```
01  package adapter;
02
03  public class SocketClassAdapterImpl extends Socket implements
04
05      @Override
06      public Volt get240Volt() {
07          return getVolt();
08      }
09
10      @Override
11      public Volt get12Volt() {
12          Volt volt = getVolt();
13          return convertVolt(volt, 10);
14      }
15
16      @Override
```



AdapterPatternTest.java

```
01  package adapter;
02
03  public class AdapterPatternTest {
04
05      public static void main(String[] args) {
06
07          SocketAdapter sockAdapter = new SocketClassAdapterImpl
08
09          Volt v3 = getVolt(sockAdapter, 3);
10          Volt v12 = getVolt(sockAdapter, 12);
11          Volt v240 = getVolt(sockAdapter, 120);
12          System.out.println("v3 volts using Class Adapter=" + v
13          System.out.println("v12 volts using Class Adapter=" +
14          System.out.println("v240 volts using Class Adapter=" +
15      }
16
```

ADAPTER DIAGRAMMA UML



ADAPTER: ESEMPI NELLA JDK

- java.util.Arrays#asList()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)

BRIDGE

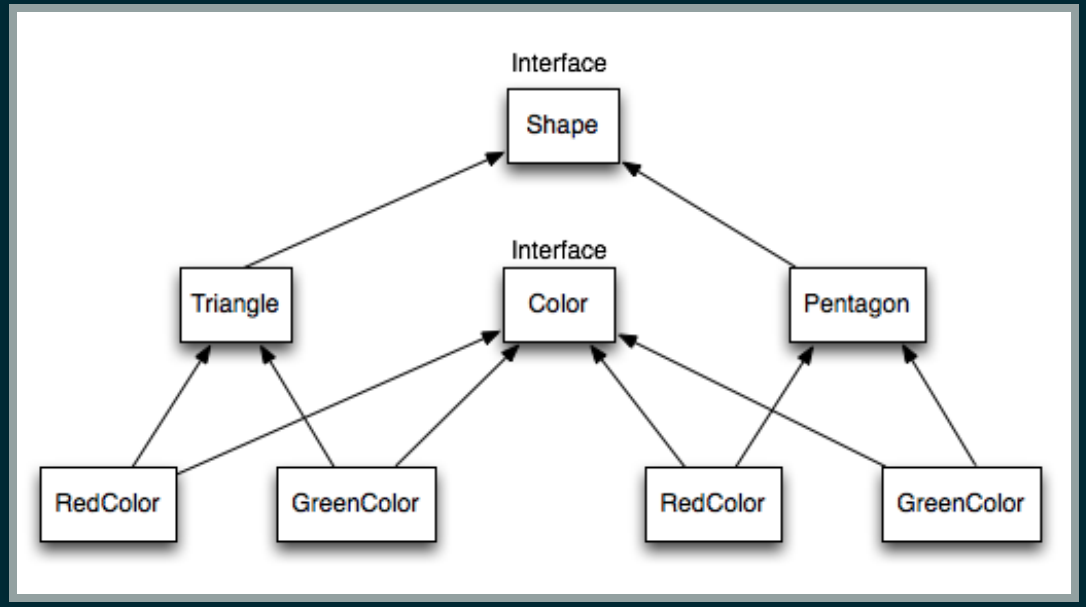
Quando abbiamo ereditarietà sia nelle interfacce che nelle implementazioni, il DP bridge è usato per disaccoppiare le interfacce dall'implementazione e per nascondere i dettagli dell'implementazione al client.

BRIDGE SECONDO LA GOF

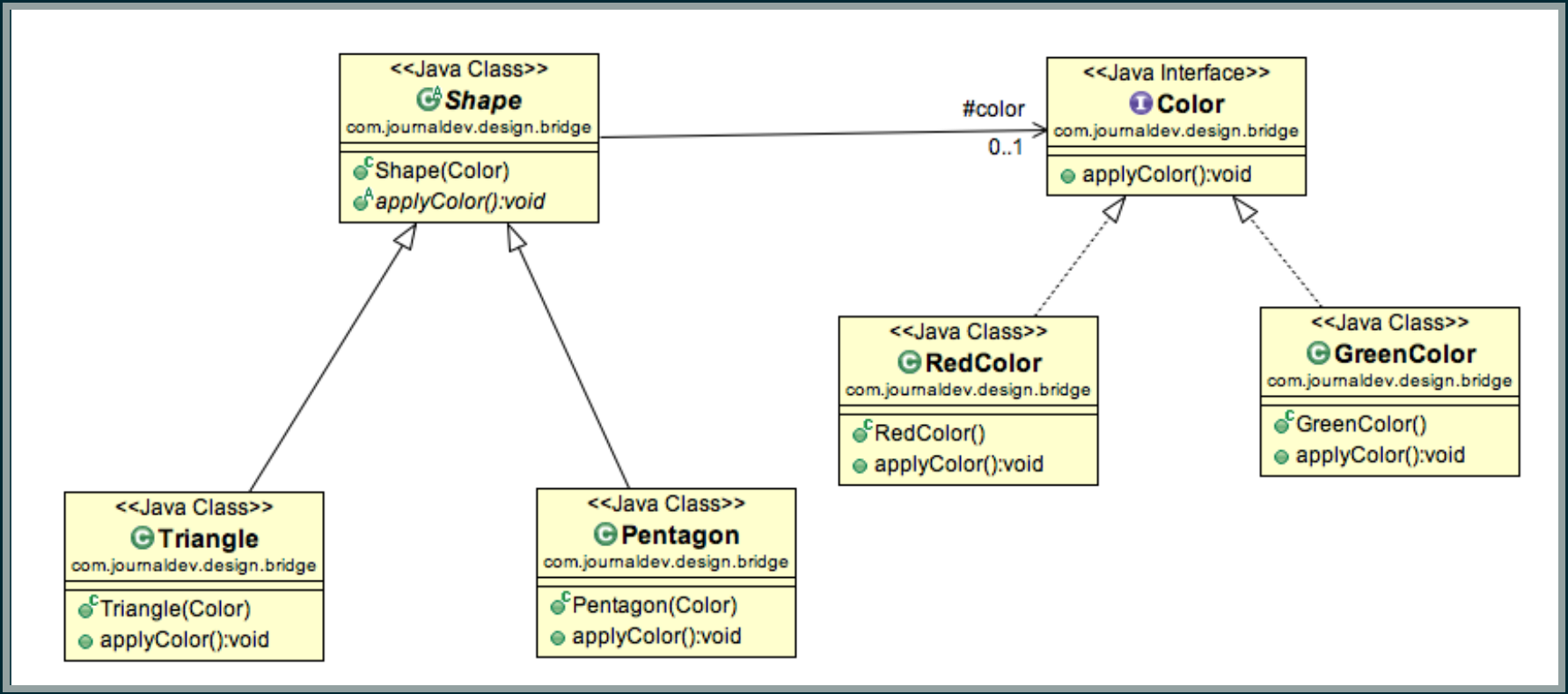
"Decouple an abstraction from its implementation so that the two can vary independently"

BRIDGE

Capiamolo meglio con un esempio. Mettiamo di avere questa situazione:



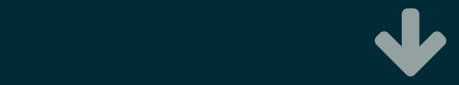
Applicando il Bridge disaccoppiamo le interfacce dall'implementazione tramite l'uso della composizione.



BRIDGE: JAVA CODE

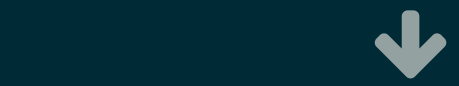
Color.java

```
01  package bridge;
02
03  public interface Color {
04
05      public void applyColor();
06  }
```



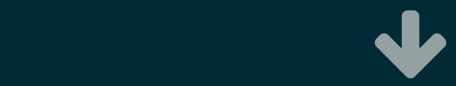
GreenColor.java

```
01  package bridge;
02
03  public class GreenColor implements Color {
04      @Override
05      public void applyColor() {
06          System.out.println("green.");
07      }
08  }
```



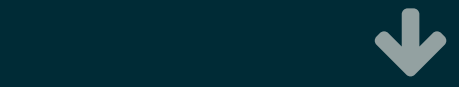
RedColor.java

```
01  package bridge;
02
03  public class RedColor implements Color {
04      @Override
05      public void applyColor() {
06          System.out.println("red.");
07      }
08  }
```



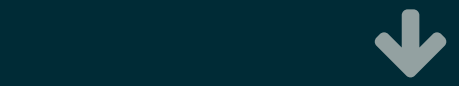
Shape.java

```
01  package bridge;
02
03  public abstract class Shape {
04
05      //composition - implementor
06      protected Color color;
07
08      //constructor with implementor as input argument
09      public Shape(Color color) {
10          this.color = color;
11      }
12
13      abstract public void applyColor();
14  }
```



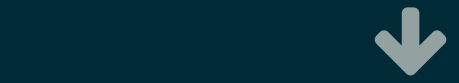
Triangle.java

```
01  package bridge;
02
03  public class Triangle extends Shape {
04
05      public Triangle(Color color) {
06          super(color);
07      }
08
09      @Override
10      public void applyColor() {
11          System.out.print("Triangle filled with color ");
12          color.applyColor();
13      }
14
15  }
```



Pentagon.java

```
01  package bridge;
02
03  public class Pentagon extends Shape {
04      public Pentagon(Color color) {
05          super(color);
06      }
07
08      @Override
09      public void applyColor() {
10          System.out.print("Pentagon filled with color ");
11          color.applyColor();
12      }
13  }
```



BridgePatternTest.java

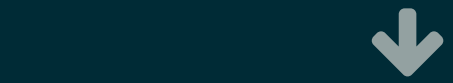
```
01  package bridge;
02
03  public class BridgePatternTest {
04
05      public static void main(String[] args) {
06          Shape triangle = new Triangle(new RedColor());
07          triangle.applyColor();
08
09          Shape pentagon = new Pentagon(new GreenColor());
10          pentagon.applyColor();
11      }
12  }
```

COMPOSITE

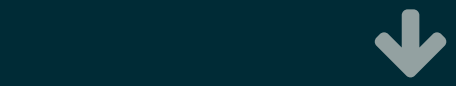
Quando dobbiamo creare una struttura in modo tale che gli oggetti di tale struttura debbano essere trattati allo stesso modo, possiamo applicare il DP composite.

COMPOSITE - ESEMPIO PRATICO

Ad esempio un diagramma è un oggetto che consiste di altri oggetti come cerchi, linee, triangoli ecc... .



Se volessimo riempire tutto il disegno del diagramma diciamo di rosso, lo stesso colore sarà applicato a tutti gli oggetti dell'insieme.



Questo disegno è composto da differenti componenti, i quali hanno certe operazioni in comune.

IL COMPOSITE DP È COMPOSTO DAI SEGUENTI OGGETTI:

- Base Component
- Leaf
- Composite

BASE COMPONENT

è l'interfaccia (o classe astratta) per tutti gli oggetti della composizione. Il client lo usa per lavorare con gli oggetti della composizione.

LEAF

Definisce il comportamento degli elementi della composizione. E' il building block della composizione ed implementa il Base Component. Non deve avere references agli altri componenti.

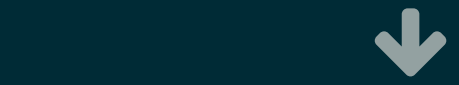
COMPOSITE

Definisce il comportamento dei componenti con i figli(leaf), memorizza i componenti figli, implementa le operazione correlate ai figli definite dall'interfaccia Component

COMPOSITE - ESEMPIO JAVA

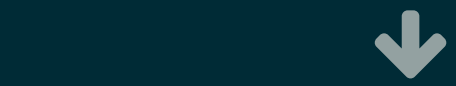
Shape.java

```
01  package composite;
02
03  public interface Shape {
04
05      public void draw(String fillColor);
06  }
```



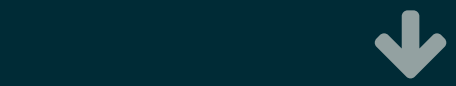
Triangle.java

```
01  package composite;
02
03  public class Triangle implements Shape {
04      @Override
05      public void draw(String fillColor) {
06          System.out.println("Drawing Triangle with color " + fi
07      }
08  }
```



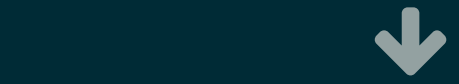
Circle.java

```
01  package composite;
02
03  public class Circle implements Shape {
04      @Override
05      public void draw(String fillColor) {
06          System.out.println("Drawing Circle with color" + fillC
07      }
08  }
```



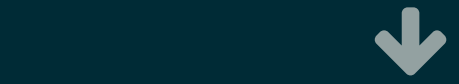
Drawing.java

```
01  package composite;
02
03  import java.util.ArrayList;
04  import java.util.List;
05
06  public class Drawing implements Shape {
07
08      //collection of Shapes
09      private List<Shape> shapes = new ArrayList<>();
10
11      @Override
12      public void draw(String fillColor) {
13          for (Shape shape : shapes) {
14              shape.draw(fillColor);
15          }
16      }
```

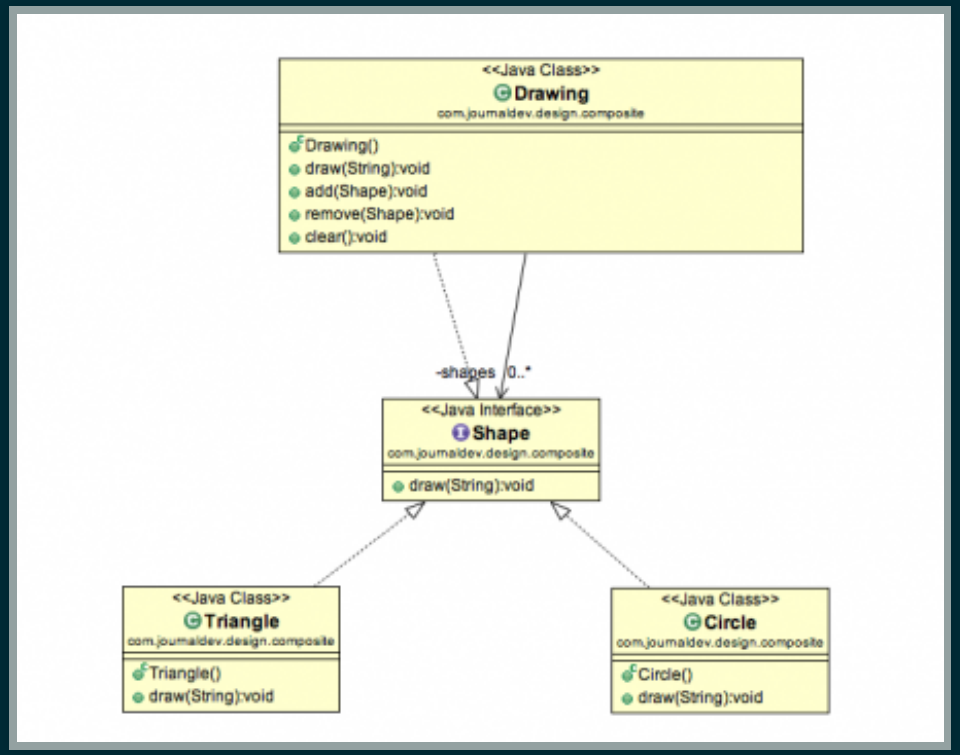


TestCompositePattern.java

```
01  package composite;
02
03  public class TestCompositePattern {
04
05      public static void main(String[] args) {
06          Shape tri = new Triangle();
07          Shape tri1 = new Triangle();
08          Shape cir = new Circle();
09
10          Drawing drawing = new Drawing();
11          drawing.add(tri1);
12          drawing.add(tri1);
13          drawing.add(cir);
14
15          drawing.draw("Red");
16
```

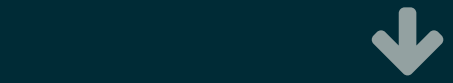


COMPOSITE UML



COMPOSITE - CONCLUSIONI

- dovrebbe essere utilizzato solo quando il gruppo di oggetti si deve comportare come se fosse una cosa sola
- può essere usato per comporre strutture ad albero

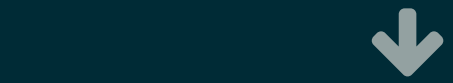


COMPOSITE DENTRO LA JVM

java.awt.Container#add(Component) è un ottimo esempio di composite, è usato moltissimo in Swing

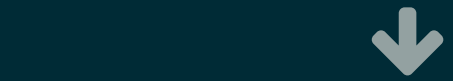
DECORATOR

è usato per modificare le funzionalità di un oggetto a runtime. Allo stesso tempo le altre istanze della stessa classe non saranno affette da questa modifica



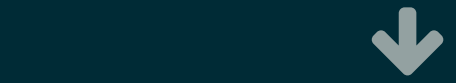
DECORATOR

Di solito usiamo l'ereditarietà o la composizione per estendere i comportamenti di un oggetto, ma questo viene fatto al momento della compilazione e va a coinvolgere tutti gli oggetti di una classe.



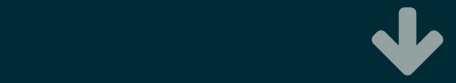
DECORATOR

Non possiamo aggiungere nessuna nuova funzionalità o rimuovere un comportamento a runtime - questo è il momento in cui il design pattern *Decorator* ci viene in aiuto!

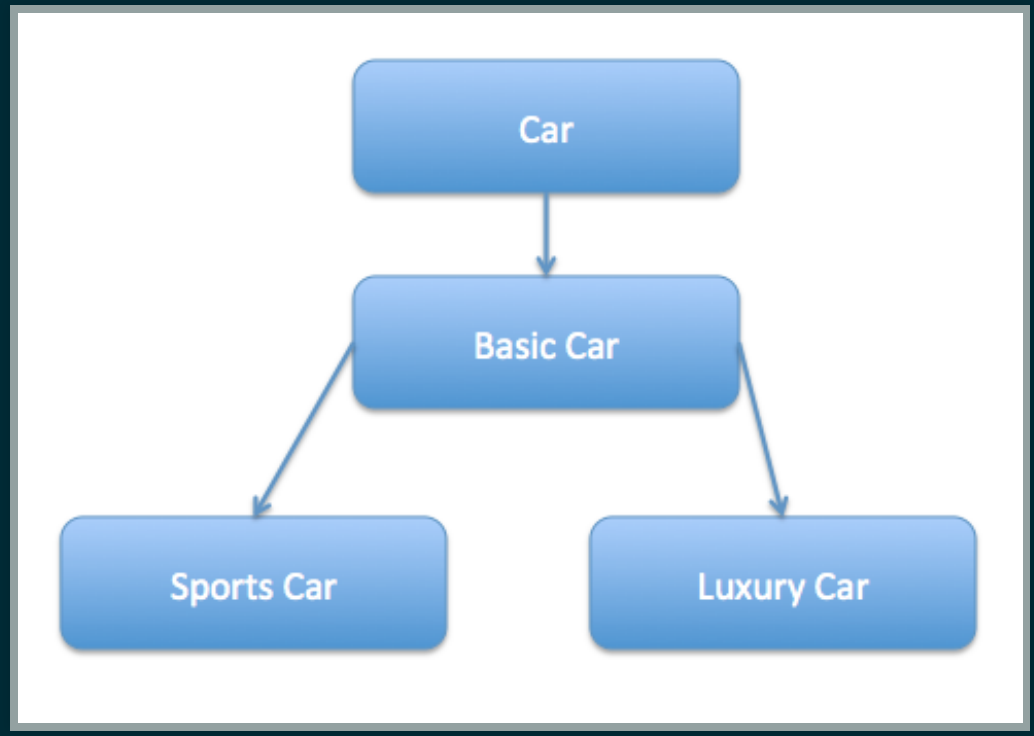


DECORATOR

Supponiamo di voler implementare diversi tipi di automobili; possiamo creare un'interfaccia *Car* per definire il metodo *assemble()* e poi possiamo avere una *Basic car* che potrà essere estesa in auto sportiva e auto di lusso.

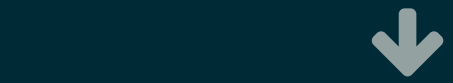


DECORATOR



DECORATOR

Ma se vogliamo avere a runtime un'auto che è sia sportiva che di lusso, l'implementazione diventa complessa, specie se vogliamo specificare altre cose. Se avessimo 10 tipi diversi di macchine, la logica d'implementazione e composizione diventerebbe impossibile da gestire.



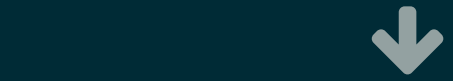
THINKOPEN

DECORATOR

DECORATOR

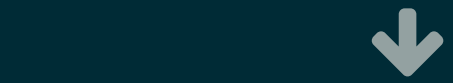
Struttura del decorator:

1. component interface
2. component implementation
3. decorator
4. concrete decorator



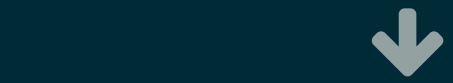
COMPONENT INTERFACE

è l'interfaccia o la *classe atratta* che definisce i metodi che saranno implementati. Nel nostro caso *Car* sarà il nostro component interface



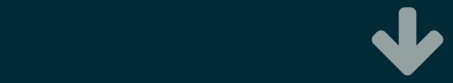
COMPONENT IMPEMENTATION

è l'implementazione base dell'interfaccia definita prima: *BasicCar*



DECORATOR

la classe *Decorator* implementa la component interface ed ha una relazione *HAS-A* sempre con la component interface. La variabile del component sarà accessibile ai figli.

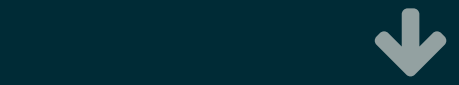


CONCRETE DECORATORS

Estendono il decorator e ne modificano il comportamento.

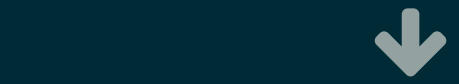
Car.java

```
01  package decorator;
02
03  public interface Car {
04
05      public void assemble();
06  }
```



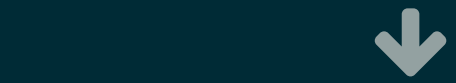
BasicCar.java

```
01  package decorator;
02
03  public class BasicCar implements Car {
04      @Override
05      public void assemble() {
06          System.out.println("Basic Car.");
07      }
08  }
```



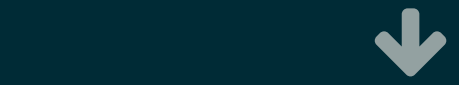
CarDecorator.java

```
01  package decorator;
02
03  public class CarDecorator implements Car {
04
05      protected Car car;
06
07      public CarDecorator(Car car) {
08          this.car = car;
09      }
10
11      @Override
12      public void assemble() {
13          this.car.assemble();
14      }
15  }
```



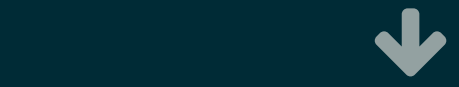
SportsCar.java

```
01  package decorator;
02
03  public class SportsCar extends CarDecorator {
04
05      public SportsCar(Car car) {
06          super(car);
07      }
08
09      @Override
10      public void assemble() {
11          super.assemble();
12          System.out.println(" Adding features of Sports Car.");
13      }
14  }
```



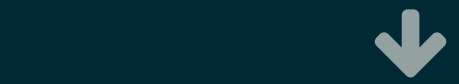
LuxuryCar.java

```
01  package decorator;
02
03  public class LuxuryCar extends CarDecorator {
04
05      public LuxuryCar(Car car) {
06          super(car);
07      }
08
09      @Override
10      public void assemble() {
11          super.assemble();
12          System.out.println(" Adding features of Luxury Car.");
13      }
14  }
```



DecoratorPatternTest.java

```
01  package decorator;
02
03  public class DecoratorPatternTest {
04
05      public static void main(String[] args) {
06
07          Car sportCar = new SportsCar(new BasicCar());
08          sportCar.assemble();
09          System.out.println("\n******");
10
11          Car sportLuxuryCar = new SportsCar(new LuxuryCar(new B
12          sportLuxuryCar.assemble();
13
14      }
15  }
```

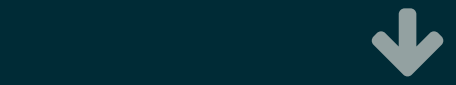


DECORATOR - PUNTI IMPORANTI

- facile da mantenere ed estendere
- ha come svantaggio l'uso di molti oggetti simili (decorators)
- è usato molto nelle classi **JAVA IO**, come la **FileReader** e la **BufferedReader**

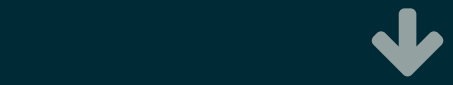
FACADE

Fornisce un'interfaccia unificata ad un set di interfacce di un sottosistema. Rende il sottosistema più facile da utilizzare.



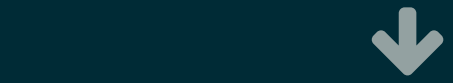
FACADE

Supponiamo di avere un'applicazione con una serie di interfacce per usare database di tipo Mysql/Oracle e che genera differenti tipi di report, in HTML, PDF, ecc...



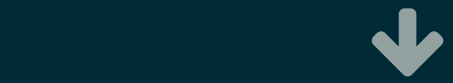
FACADE

quindi avremo differenti set di interfacce che lavorano con differenti tipi di database. Adesso un'applicazione potrebbe utilizzare queste interfacce per accedere ai database e generare i report.



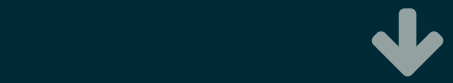
FACADE

Ma quando la complessità aumenta, o i nomi delle interfacce iniziamo a diventare poco chiari, l'applicazione inizierà ad essere difficile da mantenere



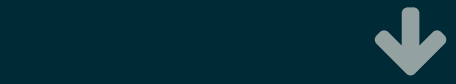
FACADE

Quindi per aiutare l'applicazione possiamo applicare il pattern Facade per scrivere un'interfaccia *wrapper*



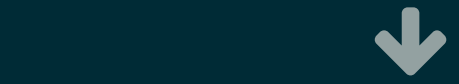
MySqlHelper.java

```
01  package facade;
02
03  import java.sql.Connection;
04
05  public class MySqlHelper {
06
07      public static Connection getMySqlDBConnection() {
08          //get MySql DB connection using connection parameters
09          return null;
10      }
11
12      public void generateMySqlPDFReport(String tableName, Conne
13          //get data from table and generate pdf report
14      }
15
16      public void generateMySqlHTMLReport(String tablename, Conn
```



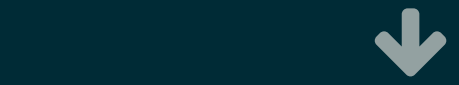
OracleHelper.java

```
01  package facade;
02
03  import java.sql.Connection;
04
05  public class OracleHelper {
06
07      public static Connection getOracleDBConnection() {
08          //get Oracle DB connection using connection parameters
09          return null;
10      }
11
12      public void generateOraclePDFReport(String tablename, Conn
13          //get data from table and generate pdf report
14      }
15
16      public void generateOracleHTMLReport(String tablename, Cor
```



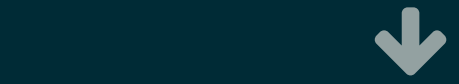
HelperFacade.java

```
01  package facade;
02
03  import java.sql.Connection;
04
05  public class HelperFacade {
06
07      public static void generateReport(DBTypes dbType, ReportTy
08
09          Connection connection = null;
10          switch (dbType) {
11              case MYSQL:
12                  connection = MySqlHelper.getMySqlDBConnection(
13                  MySqlHelper mySqlHelper = new MySqlHelper();
14                  switch (reportType) {
15                      case HTML:
16                          mySqlHelper.generateMySqlHTMLReport(t
```



FacadePatternTest.java

```
01  package facade;
02
03  import java.sql.Connection;
04
05  public class FacadePatternTest {
06
07      public static void main(String[] args) {
08
09          String tableName = "Employee";
10
11          //generating Mysql HTML report and Oracle PDF report w
12          Connection connection = MySqlHelper.getMySqlDBConnecti
13          MySqlHelper mySqlHelper = new MySqlHelper();
14          mySqlHelper.generateMySqlHTMLReport(tableName, connect
15
16          Connection connection1 = OracleHelper.getOracleDBConn
```



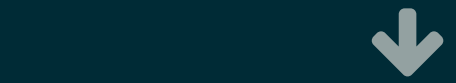
THINKOPEN

FACADE - PUNTI IMPORTANTI

PROXY

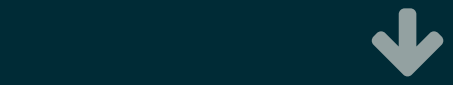
definione della **GoF**:

"Provide a surrogate or placeholder for another object to control access on it"



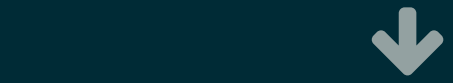
PROXY

la definizione è molto semplice, il Proxy è utilizzato quando vogliamo fornire un accesso controllato ad una funzionalità



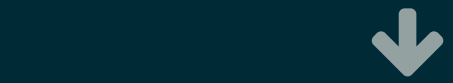
PROXY

diciamo che abbiamo una classe che può eseguire alcuni comandi in un sistema. Se la usiamo noi non ci sono problemi, ma se dovessimo delegarne l'uso ad un altro client ci potrebbero essere seri problemi di sicurezza



PROXY

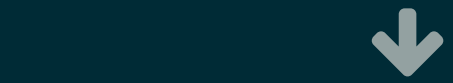
ed ecco che una classe che fa da Proxy ci può aiutare ad avere un accesso controllato al programma.



PROXY

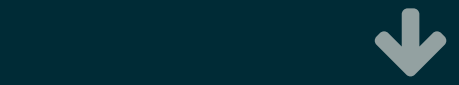
in questo esempio di Proxy abbiamo:

- un'interfaccia per eseguire il comando
- un'implementazione a questa interfaccia
- una classe di proxy che limita i comandi ad utenti non Admin



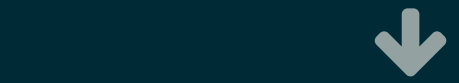
CommandExecutor.java

```
01  package proxy;
02
03  public interface CommandExecutor {
04
05      public void runCommand(String cmd) throws Exception;
06  }
```



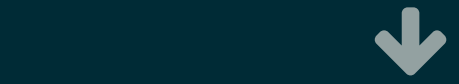
CommandExecutorImpl.java

```
01  package proxy;
02
03  public class CommandExecutorImpl implements CommandExecutor {
04
05      @Override
06      public void runCommand(String cmd) throws Exception {
07          //some heavy implementation
08          Runtime.getRuntime().exec(cmd);
09          System.out.println("'" + cmd + "' command executed");
10      }
11  }
```



CommandExecutorProxy.java

```
01  package proxy;
02
03  public class CommandExecutorProxy implements CommandExecutor {
04
05      private boolean isAdmin;
06      private CommandExecutor executor;
07
08      public CommandExecutorProxy(String user, String pwd) {
09          if ("scelia".equals(user) && "password".equals(pwd)) {
10              isAdmin = true;
11          }
12          executor = new CommandExecutorImpl();
13      }
14
15      @Override
16      public void runCommand(String cmd) throws Exception {
```

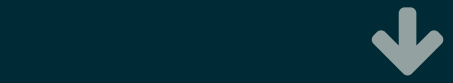


PROXY

Il package Java RMI usa molto il pattern proxy

FLYWEIGHT

Viene utilizzato quando ci sono da creare tanti oggetti di una classe. Siccome ogni oggetto consuma memoria, che è cruciale ad esempio per dispositivi mobili o embedded, il design patter Flyweight ci aiuta a ridurre il carico sulla memoria condividendo gli oggetti



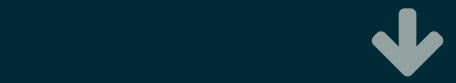
FLYWEIGHT

prima di utilizzare il DP Flyweight ci sono da considerare i seguenti fattori:

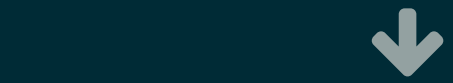
- il numero di oggetti creati dall'applicazione potrebbe essere enorme
- la creazione di oggetti in memoria potrebbe essere molto time consuming
- le proprietà degli oggetti possono essere separate in una parte variabile e da una che può essere riutilizzata, in modo da condividere quest'ultima fra differenti istanze

FLYWEIGHT

inoltre ci serve creare un **Flyweight factory** che ritorni gli shared objects.

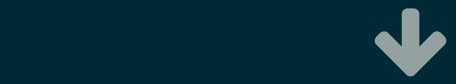


Ad esempio dobbiamo creare un disegno con linee ed ovali; quindi avremo un interfaccia *Shape* e le implementazioni *Line* e *Oval*. La classe Oval avrà proprietà intrinseche per determinare se riempire l'Ovale con un dato colore o meno, mentre Line non avrà alcuna proprietà intrinseca.



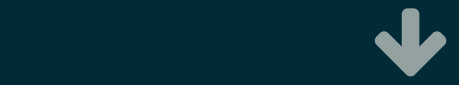
Shape.java

```
01  package flyweight;
02
03  import java.awt.*;
04
05  public interface Shape {
06
07      public void draw(Graphics g, int x, int y, int width, int
08  }
```



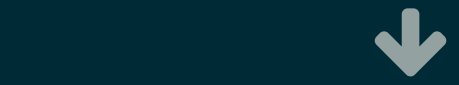
Line.java

```
01  package flyweight;
02
03  import java.awt.*;
04
05  public class Line implements Shape {
06
07      public Line() {
08          System.out.println("Creating Line object");
09          //adding time delay
10          try {
11              Thread.sleep(2000);
12          } catch (InterruptedException e) {
13              e.printStackTrace();
14          }
15      }
16
```



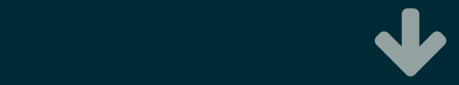
Oval.java

```
01  package flyweight;
02
03  import java.awt.*;
04
05  public class Oval implements Shape {
06
07      //intrinsic property
08      private boolean fill;
09
10      public Oval(boolean fill) {
11          this.fill = fill;
12          System.out.println("Creating Oval object with fill=" +
13
14          //adding time delay
15          try {
16              Thread.sleep(2000);
```



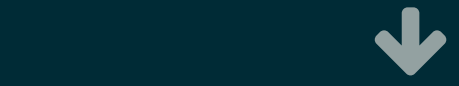
ShapeFactory.java

```
01  package flyweight;
02
03  import java.util.HashMap;
04
05  public class ShapeFactory {
06
07      private static final HashMap<ShapeType, Shape> shapes = ne
08
09      public static Shape getShape(ShapeType type) {
10          Shape shapeImpl = shapes.get(type);
11
12          if (shapeImpl == null) {
13              if (type.equals(ShapeType.OVAL_FILL)) {
14                  shapeImpl = new Oval(true);
15              } else if (type.equals(ShapeType.OVAL_NOFILL)) {
16                  shapeImpl = new Oval(false);
```



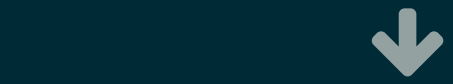
DrawingClient.java

```
01  package flyweight;
02
03  import javax.swing.*;
04  import java.awt.*;
05  import java.awt.event.ActionEvent;
06  import java.awt.event.ActionListener;
07
08  import flyweight.ShapeFactory.ShapeType;
09
10  public class DrawingClient extends JFrame {
11
12      private static final long serialVersionUID = -135020043728
13      private final int WIDTH;
14      private final int HEIGHT;
15
16      private static final ShapeType shapes[] = {ShapeType.LINE
```



FLYWEIGHT - ESEMPI NELLA JDK

Tutti i metodi `valueOf()` delle **wrapper classes** usano oggetti cached usando Flyweight DP. L'esempio più noto è nella classe String con l'implementazione **String Pool**.



FLYWEIGHT - PUNTI IMPORTANTI

- nel nostro esempio non obblighiamo il client ad utilizzare il Flyweight, ma volendo si potrebbe fare, sono scelte di design
- introduce complessità nei progetti, bisogna valutarne il trade-off a seconda dei casi

THE END

GRAZIE DELLA PAZIENZA!