1

Design Pattern

Ogni pattern descrive un **problema** che ricorre più volte nel nostro ambiente, descrive poi il **nucleo della soluzione** del problema in modo da poter utilizzare tale soluzione un milione di volte senza mai farlo allo stesso modo

2

una soluzione generica e riusabile a problemi frequenti nel software design

3

Classificazione dei Design Pattern

Strutturali	Comportamentali	Creazionali
Adapter Bridge Facade Proxy	Command Observer Strategy	Abstract Factory Builder Pattern

4

Classificazione dei Design Pattern

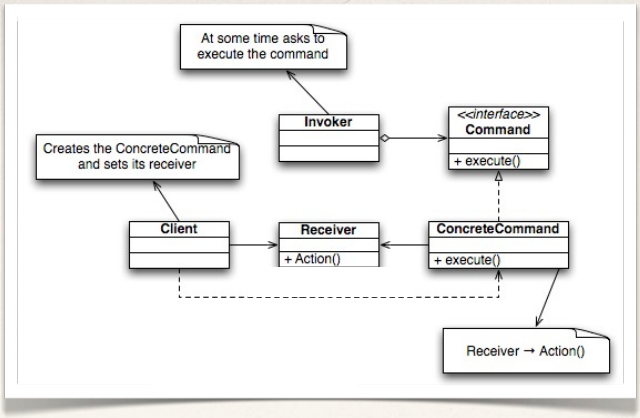
Strutturali	Comportamentali	Creazionali
Adapter Bridge Facade Proxy	**Command** **Observer** **Strategy**	Abstract Factory Builder Pattern

Bold se trattato in questa lezione

5

Comportamentali

6



Command

Permette di isolare la porzione di codice che effettua un'azione dal codice che ne richiede l'esecuzione; l'azione è incapsulata nell'oggetto Command.

7

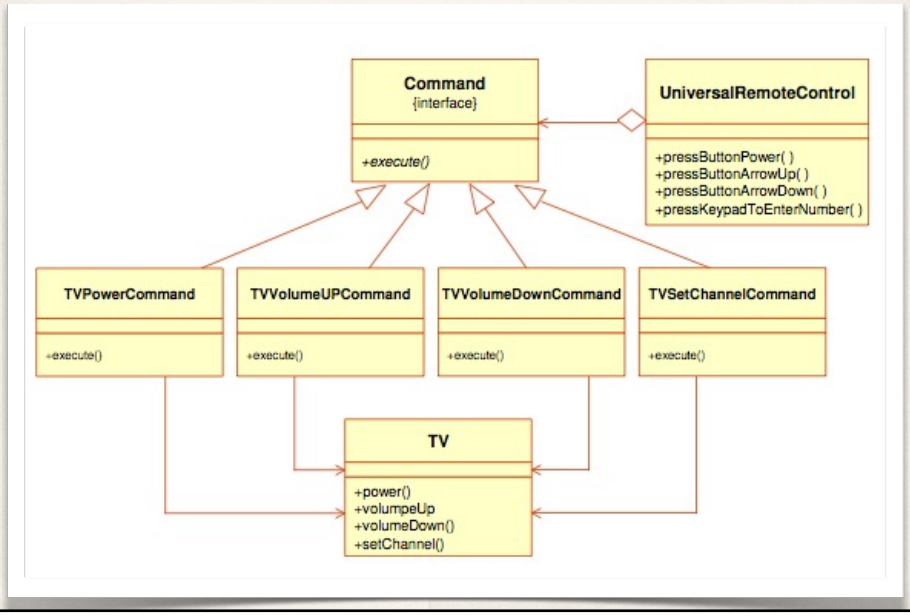
Command
Un telecomando universale

Un telecomando universale consente di gestire diversi tipi di elettrodomestici. Durante la fase di progettazione del telecomando si riesce a definire il numero di tasti e la etichetta di ogni singolo tasto, ma non le operazione concrete da eseguire (vale dire, il segnale particolare da spedire all'elettrodomestico).

Queste operazioni verranno aggiunte al momento di configurare il telecomando (ad esempio, l'utente potrebbe scaricare da internet le classi necessarie per i propri elettrodomestici).
Il problema consiste nella progettazione di una classe (telecomando), in grado di inoltrare richieste verso oggetti che saranno noti solo in fasi successive di sviluppo.

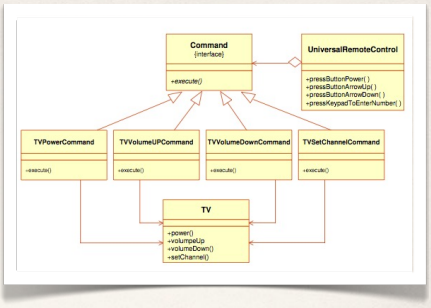
8

Command
Un telecomando universale



9

Command
Un telecomando universale



Command: classe Command.
Dichiara una interfaccia per l'esecuzione di una operazione.

ConcreteCommand:
classi TVPowerCommand, TVVolumeUpCommand, TVVolumeDownCommand e TVSetChannelCommand.

Ognuno definisce una associazione tra un Receiver e una azione. Implementa l'operazione execute, invocando la corrispondente operazione del Receiver.

Invoker: classe UniversalRemoteControl. Richiede tramite il Command l'esecuzione di una operazione.

Receiver: classe TV. Porta avanti le operazioni richieste. Qualunque classe può essere Receiver.

10

Class TV - Receiver (1)

```
public class TV {
  public static final int ON = 1;
  public static final int OFF = 0;

  private int power = OFF;
  private int volume = 0;
  private int channel = 2;

  public void power() {
    if(power == OFF) {
       power = ON;
       System.out.println("The TV is ON.”); }
    else {
       power = OFF;
       System.out.println("The TV is OFF.”); }
  }

  public void volumeUp() {
    if(power == ON) {
      if(volume < 10)
        volume++;
      System.out.println("Current volume level is " + volume); }
    else
      System.out.println("You must turn on the TV.");
  }
```

11

Class TV - Receiver (2)

```
  public void volumeDown() {
    if(power == ON) {
      if(volume > 0)
        volume--;
      System.out.println("Current volume level is " + volume); }
    else
      System.out.println("You must turn on the TV.");
  }

  public void setChannel(int ch) {
    if(power == ON) {
      if(ch >= 1 && ch < 140)
        channel = ch;
      System.out.println("Current channel is " + channel); }
    else
      System.out.println("You must turn on the TV.");
  }
}
```

12

Interface Command

Command specifica l'interfaccia con la quale comunica lo UniversalRemoteControl (**Invoker**) le sue richieste alla TV (**Receiver**), e che ogni singolo ConcreteCommand deve implementare. Si noti che questa interfaccia non fornisce riferimento alcuno verso qualche particolare **Receiver**

```
public interface Command {

   public abstract void execute();

}
```

13

Class UniversalRemoteControl – Invoker (1)

L'UniversalRemoteControl viene configurato nel costruttore con i diversi ConcreteCommand incaricati di inoltrare le richieste al Receiver. Si noti che le operazioni richiamano il metodo execute di ogni comando

```
public class UniversalRemoteControl {

  private Command buttonPower, buttonArrowUp, buttonArrowDown,
        keypadToEnterNumber;

  public UniversalRemoteControl(Command pw, Command au, Command ad,
        Command nu) {
    buttonPower = pw;
    buttonArrowUp = au;
    buttonArrowDown = ad;
    keypadToEnterNumber = nu;
  }
```

14

Class UniversalRemoteControl – Invoker (2)

```
   public void pressButtonPower(){
      buttonPower.execute();
   }

   public void pressButtonArrowUp(){
      buttonArrowUp.execute();
   }

   public void pressButtonArrowDown(){
      buttonArrowDown.execute();
   }

   public void pressKeypadToEnterNumber(){
      keypadToEnterNumber.execute();
   }
}
```

15

Concrete Commands (1)

```
public class TVPowerCommand implements Command {
  private TV theTV;

  public TVPowerCommand(TV someTV) {
    theTV = someTV;
  }

  public void execute(){
    theTV.power();
  }
}

public class TVVolumeUpCommand implements Command {
  private TV theTV;

  public TVVolumeUpCommand(TV someTV) {
    theTV = someTV;
  }

  public void execute(){
    theTV.volumeUP();
  }
}
```

16

Concrete Commands (2)

```
public class TVVolumeDownCommand implements Command {
  private TV theTV;

  public TVVolumeDownCommand (TV someTV) {
    theTV = someTV;
  }

  public void execute(){
    theTV.volumeDown();
  }
}
public class TVSetChannelCommand implements Command {
  private TV theTV;

  public TVSetChannelCommand (TV someTV) {
    theTV = someTV;
  }

  public void execute(){
    int channel = getChannel(); //Da impl.
    theTV.setChannel(channel);
  }
}
```

17

Client Example (1)

crea un oggetto della classe TV, gestito da un UniversalRemoteControl, configurato con dei comandi concreti

```
public class CommandExample {
  public static void main(String[] args) throws IOException {

     TV aCommonTV = new TV();

     Command tvpower = new TVPowerCommand(aCommonTV);
     Command tvVolUp = new TVVolumeUpCommand(aCommonTV);
     Command tvVolDn = new TVVolumeDownCommand(aCommonTV);
     Command tvSetCh = new TVSetChannelCommand(aCommonTV);

     UniversalRemoteControl remote = new
       UniversalRemoteControl(tvpower, tvVolUp,tvVolDn, tvSetCh);

     BufferedReader br = new BufferedReader(new
       InputStreamReader(System.in));

     ...
```

18

Client Example (2)

```
   ...

   int nOption = 0;

   while(nOption != 9) {
      System.out.println("Select a button to press or 9 to exit");
      System.out.println("1.- POWER");
      System.out.println("2.- ARROW UP");
      System.out.println("3.- ARROW DOWN");
      System.out.println("4.- NUMERIC KEYPAD");

      nOption = Integer.parseInt(br.readLine());

      switch(nOption) {
        case 1: remote.pressButtonPower(); break;
        case 2: remote.pressButtonArrowUp(); break;
        case 3: remote.pressButtonArrowDown(); break;
        case 4: remote.pressKeypadToEnterNumber(); break;
      }
   }
  }
}
```

19

Command
Un telecomando universale

L'interfaccia Command non ha riferimento alcuno al Receiver da gestire.

Questo conduce a una implementazione dell'Invoker completamente svincolata da qualche particolare tipologia di Receiver.

In questo modo l'Invoker potrebbe gestire non solo TV, ma qualunque altra tipologia di oggetto.

20

Command
Supportare l'undo

Il Command Design pattern consente di implementare facilmente il comando Undo che annulla un'azione precedente.

21

Command
Supportare l'undo - Gestione di un conto corrente

Classe Astratta Command

```
public abstract class Command {

    public Command(Account pAccount, double pAmount) {
        this.account = pAccount;
        this.amount = pAmount;
    }

    public abstract void execute();

    public abstract void undo();

    protected Account account;
    protected double amount;
}
```

22

Supportare l'undo - Gestione di un conto corrente

Concrete Commands: Deposit

```
public class Deposit extends Command {

  public Deposit(Account pAccount, double pAmount) {
    super(pAccount, pAmount);
  }

  @Override
  public void execute() {
    this.account.setBalance(this.account.getBalance() + this.amount);
  }

  @Override
  public void undo() {
    this.account.setBalance(this.account.getBalance() - this.amount);
  }

  public String toString() {
    return this.account.getOwner() + " deposita " + this.amount;
  }
}
```

23

Supportare l'undo - Gestione di un conto corrente

Concrete Commands: Withdraw

```
public class Withdraw extends Command {

  public Withdraw(Account pAccount, double pAmount) {
      super(pAccount, pAmount);
  }

  @Override
  public void execute() {
    this.account.setBalance(this.account.getBalance() - this.amount);
  }

  @Override
  public void undo() {
    this.account.setBalance(this.account.getBalance() + this.amount);
  }

  public String toString() {
    return this.account.getOwner() + " preleva " + this.amount;
  }
}
```

24

Receiver
Supportare l'undo - Gestione di un conto corrente

```
public class Account {
    public Account(String pOwner, double pDeposit) {
        this.owner = pOwner;
        this.balance = pDeposit; }

    public double getBalance() {
        return balance; }
    public void setBalance(double balance) {
        this.balance = balance; }

    public String getOwner() {
        return owner; }
    public void setOwner(String owner) {
        this.owner = owner; }

    public String toString() {
        return this.owner + ": " + this.balance; }

    private double balance;
    private String owner;
}
```

25

Invoker
Supportare l'undo - Gestione di un conto corrente

```
import java.util.Stack;

public class Invoker {

    public Invoker() {
        this.stack = new Stack<Command>();
    }

    public void execute(Command pCommand) {
        this.stack.push(pCommand);
        pCommand.execute();
    }
    public void undo() {
        if(!this.stack.isEmpty()) {
            Command command = this.stack.pop();
            command.undo();
        }
    }
    private Stack<Command> stack;
}
```

26

Main (1)
Supportare l'undo - Gestione di un conto corrente

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Situazione iniziale:");
        System.out.println(rich.toString());
        System.out.println(poor.toString());
        System.out.println();
        deposit(rich, 1000);
        withdraw(rich, 20000);
        deposit(poor, 100);
        withdraw(poor, 50);
        undo();
        undo();
        undo();
    }
    private static void undo() {
        invoker.undo();
        System.out.println("UNDO");
        System.out.println(rich.toString());
        System.out.println(poor.toString());
        System.out.println();
    }
. . .
```

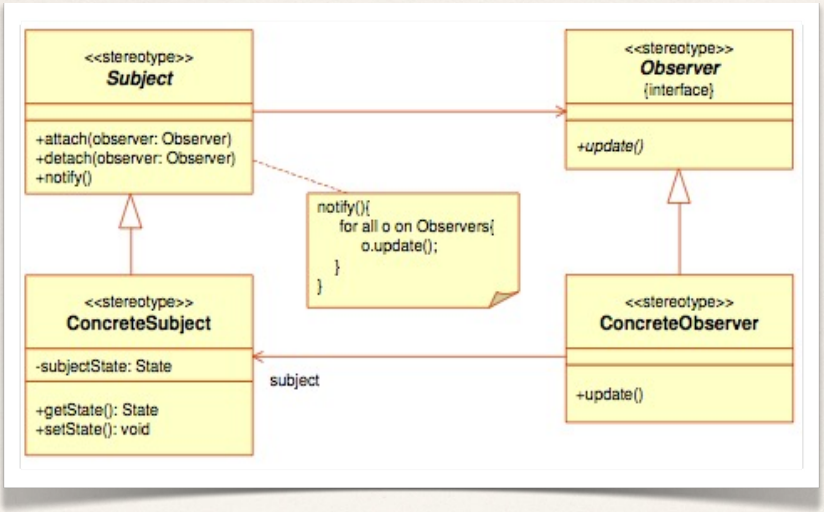
27

Main (2)
Supportare l'undo - Gestione di un conto corrente

```
    private static void deposit(Account pAccount, double pAmount) {
        Command cmd = new Deposit(pAccount, pAmount);
        invoker.execute(cmd);
        System.out.println("DO: " + cmd.toString());
        System.out.println(pAccount.toString());
        System.out.println();
    }

    private static void withdraw(Account pAccount, double pAmount) {
        Command cmd = new Withdraw(pAccount, pAmount);
        invoker.execute(cmd);
        System.out.println("DO: "+ cmd.toString());
        System.out.println(pAccount.toString());
        System.out.println();
    }

    private static Account rich = new Account("Rich Man", 100000);
    private static Account poor = new Account("Poor Man", 5);
    private static Invoker invoker = new Invoker();
}
```

28

Observer



Consente di associare molti oggetti ad uno, in modo che se quest'ultimo cambia il suo stato, tutti gli altri sono notificati e aggiornati automaticamente

29

Observer
Esempio

Al cambio di stato di un oggetto (Subject) un altro oggetto incaricato del suo monitoraggio (Observer), deve essere notificato.
Il problema è trovare un modo nel quale gli eventi dell'oggetto di riferimento (Subject), siano comunicati a tutti gli altri interessati (Observers).

30

Observer

Esempio

Subject: Classe Subject. Ha conoscenza dei propri Observer. Fornisce operazioni per l'aggiunta e cancellazione di oggetti Observer. Fornisce operazioni per la notifica agli Observer.

```
public class Subject {

     private List observers = new ArrayList();

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

     public void removeObserver(Observer o) {
          observers.remove(o); }

     public void notifyObservers() {
          Observer o;
          Iterator i = observers.iterator();
          while(i.hasNext()) {
               o = (Observer) i.next();
               o.update(); } }
}
```

31

Observer

Esempio

Observer: Interfaccia Observer. Specifica una interfaccia per la notifica di eventi agli oggetti interessati in un Subject.

```
public interface Observer {

     public void update();
}
```

32

Observer
Esempio

ConcreteSubject: Classe ConcreteSubject. Possiede uno stato di interesse dei ConcreteSubject. Invoca le operazioni di notifica ereditate dal Subject, quando devono essere informati i ConcreteObserver.

```
public class ConcreteSubject extends Subject {

     private String state = "";

     public String getState() {
          return state;
     }

     public void setState(String state) {
          this.state = state;
          notifyObservers();
     }
}
```

33

ConcreteObserver (1)
Esempio

ConcreteObserver: Classe ConcreteObserver. Implementa l'operazione di aggiornamento dell'Observer

```
public class ConcreteObserver implements Observer {
     private String state = "";
     private String name = null;
     private ConcreteSubject s = null;

     public ConcreteObserver(String pName) {
          this.name = pName;
     }

     public String getName() {
          return this.name;
     }

. . .
```

34

ConcreteObserver (2)

Esempio

```
. . .
     public void attach(ConcreteSubject s) {
          if (s != null) {
               this.s = s;
               this.s.addObserver(this);}
     }

     public void detach() {
          if(s != null) {
               s.removeObserver(this);
               s = null;}
     }

     public void update() {
          if (s != null) {
               state = s.getState();
               System.out.println(name + ": update received from
                         Subject, state changed to:" + state);}
     }
}
```

35

Observer

Esempio

Client: Classe ObserverTest.

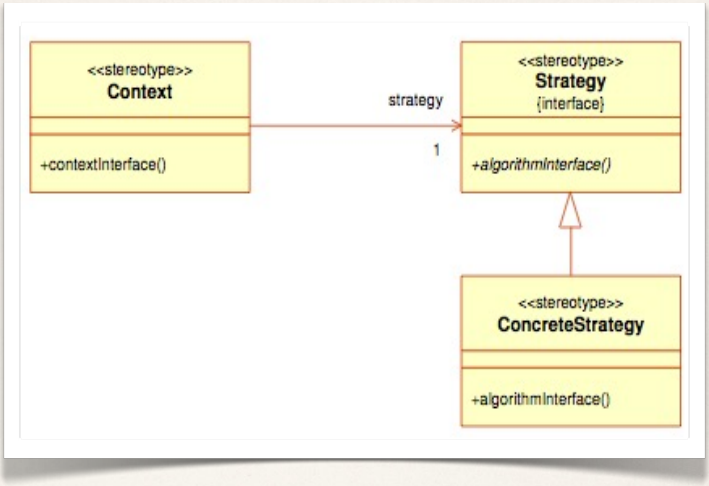
```
public class ObserverTest {

   public static void main(String[] args) {

     Observer observer1 = new ConcreteObserver("Observer 1");
     Observer observer2 = new ConcreteObserver("Observer 2");

     ConcreteSubject s = new ConcreteSubject();

     observer1.attach(s);
     observer2.attach(s);

     s.setState("New State 1");
     s.setState("New State 2");
   }
}
```

Output

```
Observer 1: update received from Subject, state changed to : New State 1
Observer 2: update received from Subject, state changed to : New State 1
Observer 1: update received from Subject, state changed to : New State 2
Observer 2: update received from Subject, state changed to : New State 2
```

36

Strategy



Il pattern Strategy è utile in quelle situazioni dove è necessario modificare dinamicamente gli algoritmi utilizzati da un'applicazione.

37

Strategy
Stampa di un Array

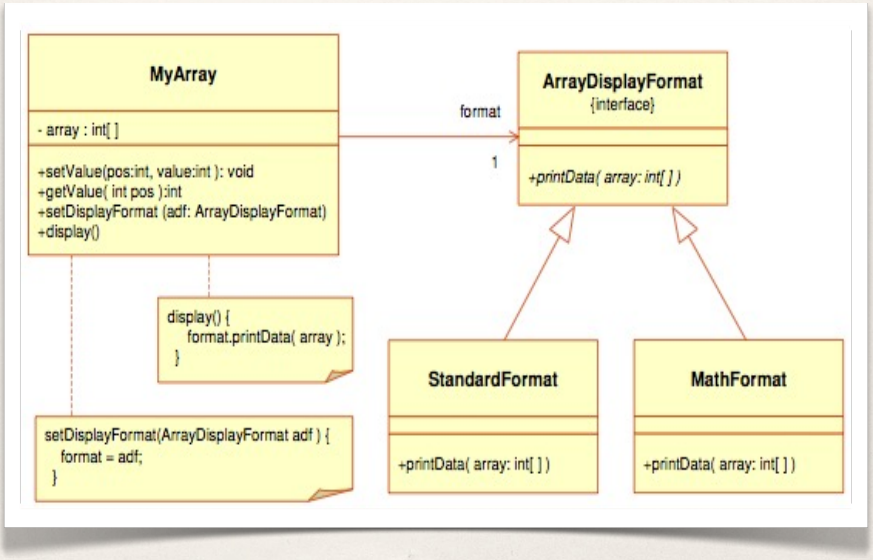
La progettazione di una applicazione che offre delle funzionalità matematiche considera la gestione di una apposita classe (MyArray) per la rappresentazione di vettori di numeri.

Tra i metodi di questa classe si è definito uno che esegue la stampa. Questo metodo potrebbe stampare il vettore nel seguente modo (chiamato, ad es. MathFormat):{ 12, -7, 3, ... }oppure di questo altro modo (chiamato, ad. es. SandardFormat):Arr[0]=12 Arr[1]=-7 Arr[2]=3 ...

Il problema è trovare un modo di isolare l'algoritmo che formatta e stampa il contenuto dell'array, per farlo variare in modo indipendente dal resto dell'implementazione della classe.

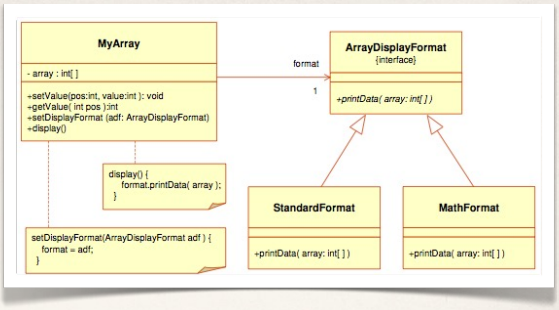
38

Strategy
Stampa di un Array



39

Strategy
Stampa di un Array



Strategy: interfaccia ArrayDisplayFormat. Dichiara una interfaccia comune per tutti gli algoritmi supportati. Il Context utilizza questa interfaccia per invocare gli algoritmi definiti in ogni ConcreteStrategy.

Concrete Strategy: classi StandardFormat e MathFormat. Implementano gli algoritmi che usano la interfaccia Strategy.

Context: classe MyArray. Viene configurato con un oggetto ConcreteStrategy e mantiene un riferimento verso esso.

40

Class MyArray - Context

Mantiene al suo interno un array di numeri, gestiti tramite i metodi setValue e getValue. La particolare modalità di stampa rimane a carico di oggetti che implementano l'interfaccia ArrayDisplayFormat. Il particolare oggetto che incapsula la procedura di stampa scelta, viene settato tramite il metodo setDisplayFormat.

```
public class MyArray {
    private int[] array;
    ArrayDisplayFormat format;

    public MyArray(int size) {array = new int[ size ];}
    public void setValue(int pos, int value) {array[pos] = value;}
    public int getValue(int pos) {return array[pos];}
    public int getLength( ) {return array.length;}
    public void setDisplayFormat(ArrayDisplayFormat adf) {format = adf;}
    public void display() {format.printData(array);}
}
```

41

Interfaccia ArrayDisplayFormat - Strategy

da implementare in ogni classe fornitrice dell'operazione di stampa.

```
public interface ArrayDisplayFormat {
    public void printData( int[] arr );
}
```

42

Concrete Strategies

```
public class StandardFormat implements ArrayDisplayFormat {
        public void printData(int[] arr) {
                System.out.print( "{ " );
                for(int i=0; i < arr.length-1; i++ )
                        System.out.print(arr[i] + ", ");
                System.out.println(arr[arr.length-1] + " }");
                        }
                }

public class MathFormat implements ArrayDisplayFormat {
        public void printData(int[] arr) {
                for(int i=0; i < arr.length ; i++ )
        System.out.println("Arr[ " + i + " ] = " + arr[i] + " ");
        }
}
```

43

Client

```
public class StrategyExample {
  public static void main (String[] arg) {
    MyArray m = new MyArray(10);
    m.setValue(1 , 6);
    m.setValue(0 , 8);
    m.setValue(4 , 1);
    m.setValue(9 , 7);
    System.out.println("This is the array in 'standard' format");
    m.setDisplayFormat(new StandardFormat());
    m.display();
    System.out.println("This is the array in 'math' format:");
    m.setDisplayFormat(new MathFormat());
    m.display();
  }
}
```

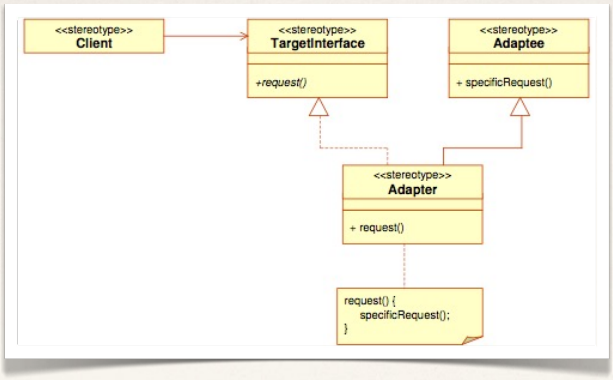
Output

```
This is the array in 'standard' format :
{ 8, 6, 0, 0, 1, 0, 0, 0, 0, 7 }

This is the array in 'math' format:
Arr[ 0 ] = 8
Arr[ 1 ] = 6
Arr[ 2 ] = 0
Arr[ 3 ] = 0
Arr[ 4 ] = 1
Arr[ 5 ] = 0
Arr[ 6 ] = 0
Arr[ 7 ] = 0
Arr[ 8 ] = 0
Arr[ 9 ] = 7
```

44

Strutturali

45



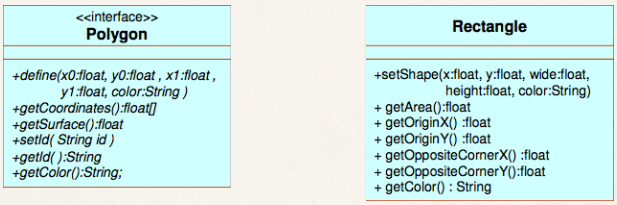
Class Adapter

Converte l'interfaccia di una classe in un'altra interfaccia attesa dai client. In questo modo, si consente la collaborazione tra classi che in altro modo non potrebbero interagire a causa delle loro diverse interfacce.

46

Class Adapter
Esempio

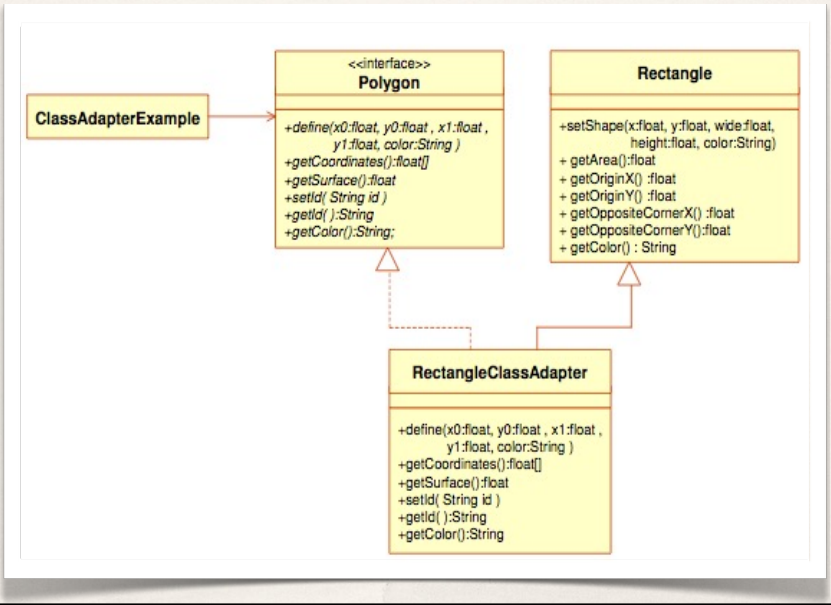
Si vuole sviluppare un'applicazione per lavorare con oggetti geometrici. Questi oggetti saranno gestiti dall'applicazione tramite un'interfaccia particolare (Polygon), che offre un insieme di metodi che gli oggetti grafici devono implementare. Si ha a disposizione una classe Legacy (Rectangle) che si potrebbe riutilizzare, che però ha un'interfaccia diversa.



Il problema consiste nella definizione di un modo di riutilizzare la classe esistente tramite una nuova interfaccia, ma senza modificare l'implementazione originale.

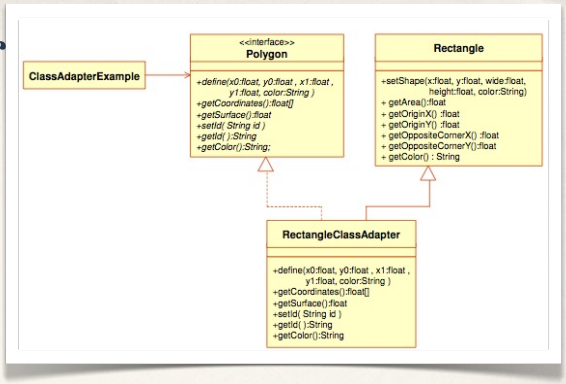
47

Class Adapter
Esempio



48

Class Adapter
Esempio



TargetInterface: interfaccia Polygon. Specifica l'interfaccia che il Client utilizza.

Client: classe ClassAdapterExample. Comunica con l'oggetto interessato tramite la TargetInterface.

Adaptee: classe Rectangle. Implementa una interfaccia che deve essere adattata.

Adapter: classe RectangleClassAdapter. Adatta l'interfaccia dell'Adaptee alla TargetInterface.

49

Classe Rectangle - Adaptee

```
public class Rectangle {
    private float x0, y0;
    private float width, height;
    private String color;

    public void setShape(float x, float y, float l, float a, String c) {
        x0 = x;
        y0 = y;
        width = l;
        height = a;
        color = c;}

    public float getArea() { return width * height;}

    public float getOriginX() { return x0;}

    public float getOriginY() { return y0;}

    public float getOppositeCornerX() { return x0 + width;}

    public float getOppositeCornerY() { return y0 + height;}

    public String getColor() { return color;}
}
```

50

Interfaccia Polygon - Target Interface

```
public interface Polygon {

  public void define(float x0, float y0, float x1, float y1, String color);

  public float[] getCoordinates() ;

  public float getSurface();

  public void setId(String id);

  public String getId();

  public String getColor();
}
```

51

Classe RectangleClassAdapter - Adapter

```
public class RectangleClassAdapter extends Rectangle implements Polygon{

  private String name = "NO NAME";

  public void define(float x0, float y0, float x1, float y1, String color)
{
      float l = x1 - x0;
      float a = y1 - y0;
      setShape(x0, y0, l, a, color);}

  public float getSurface() {return getArea();}

  public float[] getCoordinates() {
      float aux[] = new float[4];
      aux[0] = getOriginX();
      aux[1] = getOriginY();
      aux[2] = getOppositeCornerX();
      aux[3] = getOppositeCornerY();
      return aux;}

  public void setId(String id) {name = id;}

  public String getId() {return name;}
}
```

52

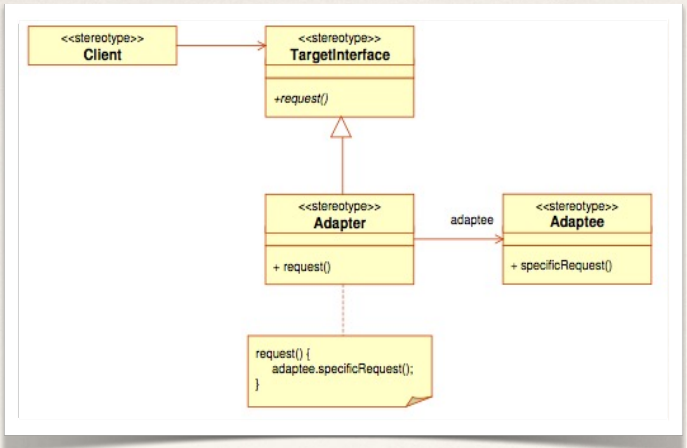
Classe ClassAdapterExample - Client

```
public class ClassAdapterExample {

  public static void main(String[] arg) {

      Polygon block = new RectangleClassAdapter();

      block.setId("Demo");

      block.define(3, 4 , 10, 20, "RED");

      System.out.println("The area of "+ block.getId() + " is "+
block.getSurface() + ", and its color is " + block.getColor());}
}
```

Output

```
The area of Demo is 112.0, and its color is RED
```

53



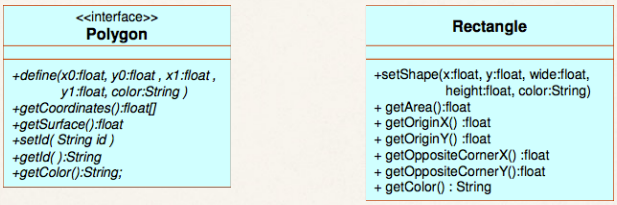
Object Adapter

Converte l'interfaccia di una classe in un'altra interfaccia attesa dai client. In questo modo, si consente la collaborazione tra classi che in altro modo non potrebbero interagire a causa delle loro diverse interfacce.

54

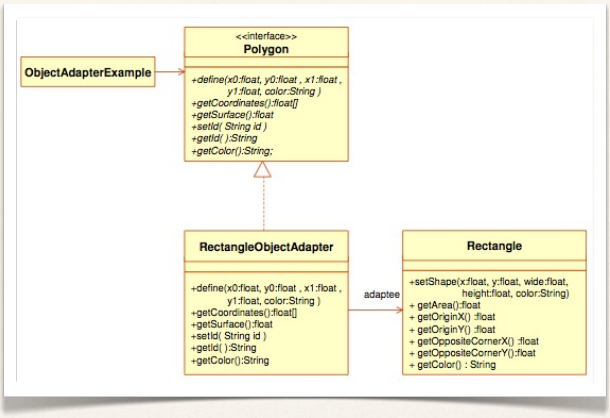
Object Adapter
Esempio

Implementiamo lo stesso esempio visto con il Class Adapter per vedere le differenze.



55

Object Adapter
Esempio

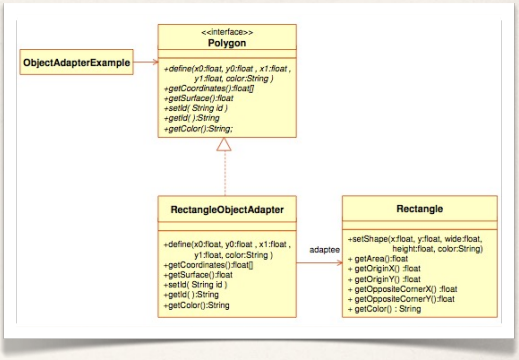


56

Object Adapter
Esempio

TargetInterface: interfaccia Polygon. Specifica l'interfaccia che il Client utilizza. (Stessa che nel Class Adapter)

Client: classe ObjectAdapterExample. Comunica con l'oggetto interessato tramite la TargetInterface.



Adaptee: classe Rectangle. Implementa una interfaccia che deve essere adattata. (Stessa che nel Class Adapter)

Adapter: classe RectangleObjectAdapter. Adatta l'interfaccia dell'Adaptee alla TargetInterface.

57

Classe Rectangle - Adaptee

Interfaccia Polygon - Target Interface

Come visto per il Class Adapter

58

Classe RectangleObjectAdapter - Adapter

```
public class RectangleObjectAdapter implements Polygon{
     Rectangle adaptee;
     private String name = "NO NAME";

     public RectangleObjectAdapter() { adaptee = new Rectangle();}

     public void define(float x0, float y0, float x1, float y1, String col) {
        float l = x1 - x0;
        float a = y1 - y0;
        adaptee.setShape(x0, y0, l, a, col);}

     public float getSurface() { return adaptee.getArea();}

     public float[] getCoordinates() {
        float aux[] = new float[4];
        aux[0] = adaptee.getOriginX();
        aux[1] = adaptee.getOriginY();
        aux[2] = adaptee.getOppositeCornerX();
        aux[3] = adaptee.getOppositeCornerY();
        return aux;}

     public void setId(String id) { name = id;}

     public String getId() {return name;}

     public String getColor() {return adaptee.getColor();}
}
```

59

Classe ObjectAdapterExample - Client

```
public class ObjectAdapterExample {

     public static void main( String[] arg ) {
          Polygon block = new RectangleObjectAdapter();
          block.setId("Demo”);
          block.define(3 , 4 , 10, 20, "RED”);
          System.out.println("The area of "+ block.getId() + " is ” +
block.getSurface() + ", and its color is " + block.getColor());
     }
}
```

Output

```
The area of Demo is 112.0, and its color is RED
```

60