Marco Faella

Elementi di programmazione di interfacce Grafiche. Il pattern OBSERVER.

Lezione n. 9

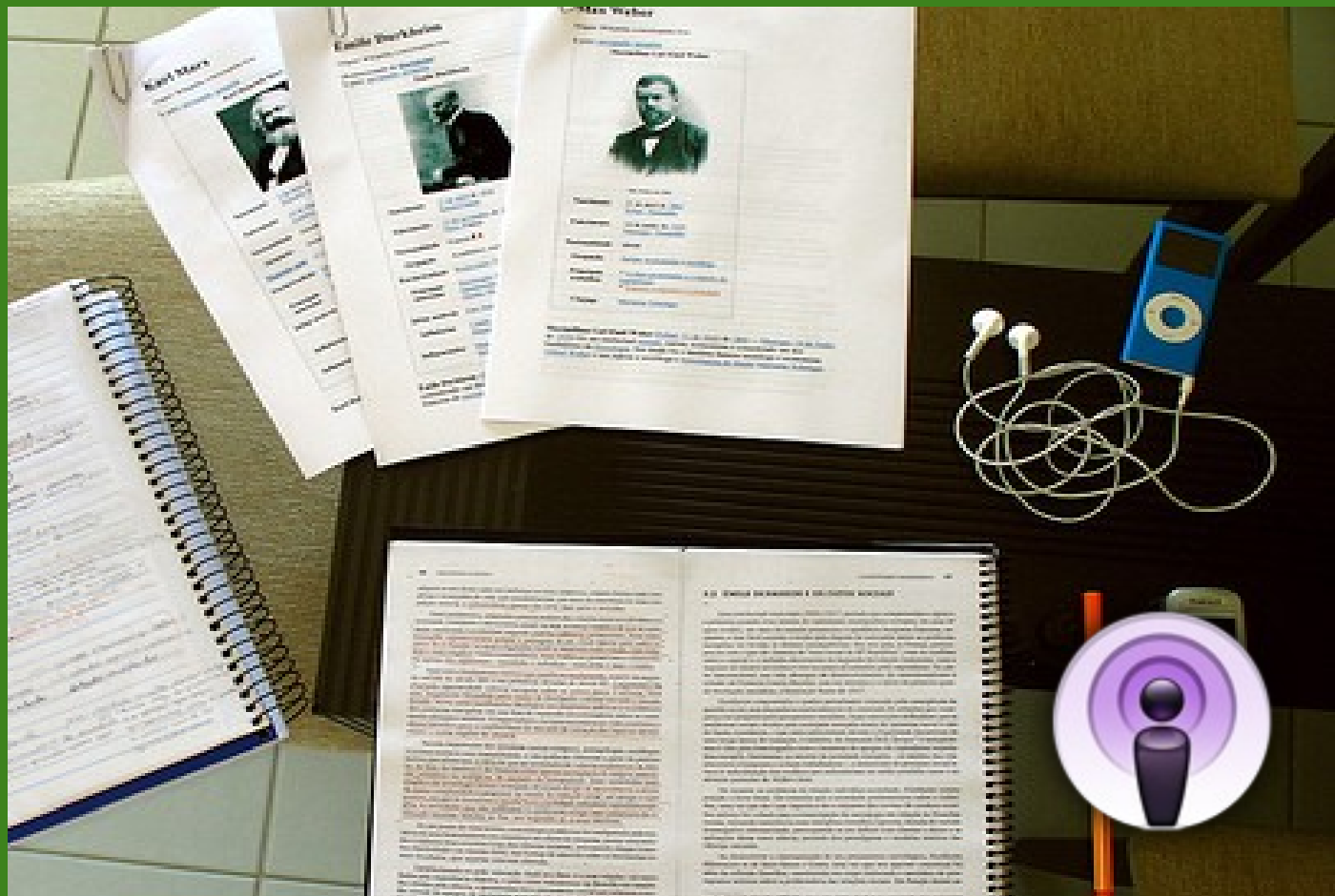
Parole chiave:
Java

Corso di Laurea:
Informatica

Insegnamento:
Linguaggi di Programmazione II

Email Docente:
faella.didattica@gmail.com

A.A. 2009-2010



- Unico tra i linguaggi di programmazione più diffusi, Java offre nella sua libreria standard un framework per la realizzazione di interfacce grafiche (GUI)
- Tale framework si basa sulle due librerie **Swing** e **AWT** (*Abstract Windowing Toolkit*)

- **AWT** è stata introdotta con la prima versione di Java (1.0)
  - AWT usa gli elementi grafici (*widget*) propri del sistema operativo ospite
  - Quindi, i programmi presentano una diversa veste grafica in ciascun sistema operativo
- **Swing**, introdotta con Java 1.2, offre dei propri elementi grafici, identici per tutti i sistemi operativi
  - Più esattamente, Swing si occupa di disegnare il contenuto delle finestre, mentre la cornice della finestra e la sua barra del titolo restano affidate alla GUI del sistema operativo
- Swing non sostituisce interamente AWT, ma si appoggia ad essa per alcune funzionalità

- Gran parte dei meccanismi presentati in questa lezione sono in realtà automatizzati dai moderni ambienti di sviluppo, come Netbeans ed Eclipse
- Tuttavia, resta comunque interessante conoscere il sistema sottostante, soprattutto dal punto di vista della progettazione
- In particolare, in queste librerie si possono vedere all'opera diversi **design pattern**

Creare una finestra

- Il seguente codice mostra come creare una finestra con Swing/AWT
- Eseguendo il programma, viene mostrata la finestra riportata nella figura a destra

```
import java.awt.*;
import javax.swing.*;

public class FrameTest
{
   public static void main(String[] args)
   {
      final int WIDTH=200, HEIGHT=200;

      // creo la finestra
      JFrame frame = new JFrame();
      // imposto le dimensioni iniziali
      frame.setSize(WIDTH, HEIGHT);
      // imposto l'operazione di chiusura
      frame.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE);
      // mostro la finestra
      frame.setVisible(true);
   }
}
```

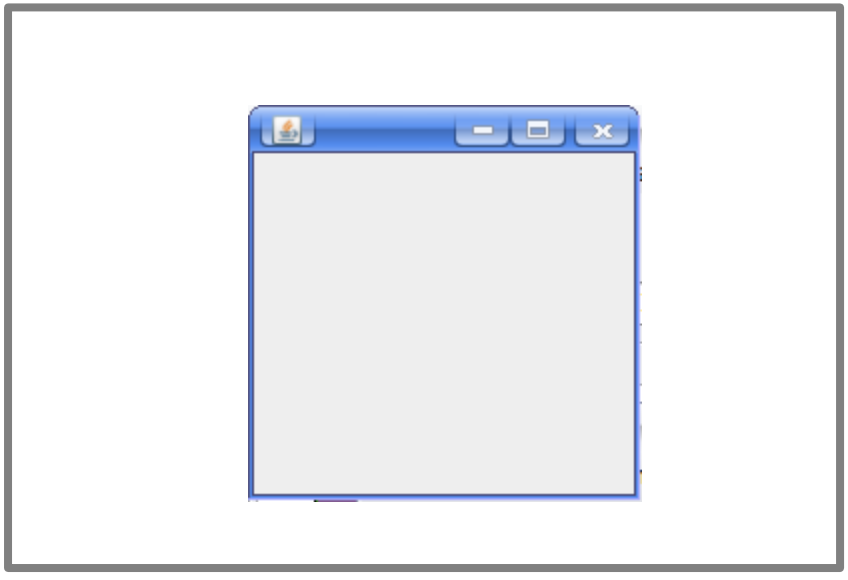


Figura 1: La finestra creata dal programma a sinistra.

Creare una finestra

- All'inizio, importiamo tutte le classi dei pacchetti java.awt e javax.swing, anche se in questo semplice esempio utilizziamo solo la classe javax.swing.JFrame

- Poi, creiamo un oggetto di tipo JFrame e impostiamo alcune sue proprietà con opportuni metodi
  - il metodo setSize imposta le dimensioni iniziali della finestra
  - il metodo setDefaultCloseOperation serve ad impostare il comportamento del programma quando la finestra in questione viene chiusa
    - il parametro JFrame.EXIT_ON_CLOSE (costante di classe) indica che il programma deve terminare quando la finestra viene chiusa
- L'ultima chiamata a metodo serve a rendere la finestra visibile

- A questo punto, il main termina
- Non termina l'intera applicazione, perché, essendo stata resa una finestra visibile, è partito automaticamente un nuovo thread che ne gestisce gli eventi
- L'applicazione terminerà solo quando l'utente cliccherà sull'apposita icona nella cornice della finestra

Aggiungere widget

- Aggiungiamo alla finestra creata precedentemente degli elementi grafici, come pulsanti e campi di testo, chiamati *widget*
- Di seguito sono riportati i passaggi principali, commentati nella prossima slide

```
// creo la finestra
JFrame frame = new JFrame();
// creo due pulsanti e un campo di testo
JButton helloButton = new JButton("Say Hello");
JButton goodbyeButton = new JButton("Say Goodbye");
final int FIELD_WIDTH = 20;
JTextField textField = new JTextField(FIELD_WIDTH);
textField.setText("Click a button!");

// ottengo un riferimento al "pannello del contenuto" (contentPane) della finestra
Container contentPane = frame.getContentPane();
// imposto il layout del pannello
contentPane.setLayout(new GridLayout(2,2));
// aggiungo al pannello i 3 elementi che ho creato
contentPane.add(helloButton);
contentPane.add(goodbyeButton);
contentPane.add(textField);
```

Aggiungere widget

- I widget principali (pulsanti, caselle di testo, barre di scorrimento, checkbox, etc.) corrispondono ad altrettante classi (JButton, JTextField, etc.), tutte sottoclassi di Component
- Per aggiungere un widget ad una finestra, si ottiene prima il Container che rappresenta lo spazio inizialmente vuoto contenuto nella finestra, chiamando il metodo getContentPane di JFrame
- La disposizione grafica dei widget all'interno di un Container si controlla impostando un *layout*
- Ciascun layout è identificato da una apposita classe, secondo le disposizioni del pattern STRATEGY, che sarà illustrato nella lezione successiva
- In questo caso, utilizziamo il layout GridLayout, che dispone gli elementi in una griglia
- I parametri (2,2) passati al costruttore rappresentano il numero di righe e di colonne della griglia
- Infine, si invoca il metodo add di Container, che aggiunge un elemento al contenitore
- Il risultato è quello nella figura a destra

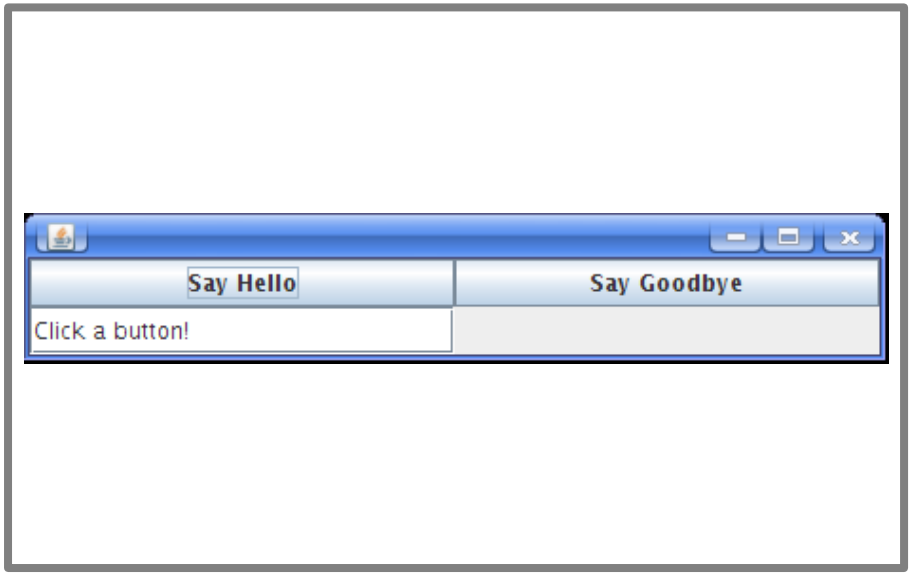


Figura 2: La finestra creata dal programma della slide precedente.

- Allo stato attuale, la pressione dei pulsanti non provoca nessuna reazione nell'applicazione
- Supponiamo di voler modificare il contenuto del campo di testo in corrispondenza della pressione di un pulsante
- A questo scopo, dobbiamo costruire un oggetto deputato a reagire all'evento “pressione del pulsante”
- Poi, registriamo questo oggetto come interessato a ricevere determinati eventi
- Questo tipo di oggetto viene chiamato “listener” (ascoltatore) e deve implementare l'interfaccia ActionListener:

```
public interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

- Il metodo actionPerformed viene richiamato automaticamente quando si genera un evento di interesse
- Per segnalare che un oggetto listener intende ricevere gli eventi generati da un determinato oggetto grafico, si chiama il metodo addActionListener di quest'ultimo, passando il primo come argomento
- La prossima slide illustra i passaggi principali

Il metodo main viene modificato come segue:

```
// creo la finestra, etc.
...
// creo il campo di testo
final JTextField textField = new JTextField(FIELD_WIDTH);
textField.setText("Click a button!");

// classe locale che gestirà gli eventi
class Ascoltatore implements ActionListener {
   public Ascoltatore(String s) { msg = s; }
   private String msg;
   public void actionPerformed(ActionEvent event) {
      textField.setText(msg);
   }
}

// creo il primo pulsante
JButton helloButton = new JButton("Say Hello");
// associo un oggetto osservatore al pulsante
helloButton.addActionListener(new Ascoltatore("Hello, World"));

// faccio lo stesso per il secondo pulsante
JButton goodbyeButton = new JButton("Say Goodbye");
goodbyeButton.addActionListener(new Ascoltatore("Goodbye, World!"));
```

- Con le modifiche della slide precedente, l'applicazione risponde alla pressione dei due pulsanti, inserendo le stringhe “Hello, World” e “Goodbye, World” nel campo di testo
- Utilizziamo la **classe locale Ascoltatore** per realizzare gli oggetti ascoltatori necessari
- Essendo locale, tale classe locale non è visibile al di fuori del metodo main
- La classe Ascoltatore può accedere alla variabile locale textField, in quanto questa è stata dichiarata *final*
- Esercizio: modificare il codice in modo da utilizzare una o più classi anonime al posto della classe Ascoltatore

- Il metodo addActionListener registra un ascoltatore interessato specificamente alla pressione di quel pulsante
- Il meccanismo degli eventi è molto più complesso, e prevede la possibilità di rilevare gli eventi più disparati, come i singoli movimenti del mouse o pressioni di tasti della tastiera
- E' possibile associare più listener alla stessa sorgente di eventi, chiamando più volte il metodo appropriato (in questo caso addActionListener)
    - quando si verificherà l'evento in questione, saranno chiamati tutti i listener registrati

Il meccanismo di eventi ed ascoltatori viene codificato dal pattern OBSERVER, dove la metafora dell'ascolto viene sostituita da quella dell'osservazione

- **Contesto:**
  1) Un oggetto (*soggetto*) genera eventi
  2) Uno o più oggetti (*osservatori*) vogliono essere informati del verificarsi di tali eventi

- **Soluzione:**
  1) Definire un'interfaccia, chiamata *Observer e dotata di* un metodo *notify*, che sarà implementata dagli osservatori
  2) Il soggetto ha un metodo (chiamato *attach*) per registrare un osservatore
  3) Il soggetto gestisce l'elenco dei suoi osservatori registrati
  4) Quando si verifica un evento, il soggetto informa tutti gli osservatori registrati, invocando il loro metodo notify

Diagramma del pattern OBSERVER

- La figura a destra rappresenta il diagramma delle classi tipico del pattern OBSERVER
- Sono presenti la classe *Subject* (soggetto osservato) e l'interfaccia *Observer* (osservatore)
- Si noti la relazione di **aggregazione uno-a-molti** tra Subject e Observer
  - tale relazione indica che ogni oggetto di tipo Subject conserva un insieme di riferimenti ad oggetti di tipo Observer
  - ovvero, ogni Subject tiene traccia degli osservatori che si sono registrati chiamando attach

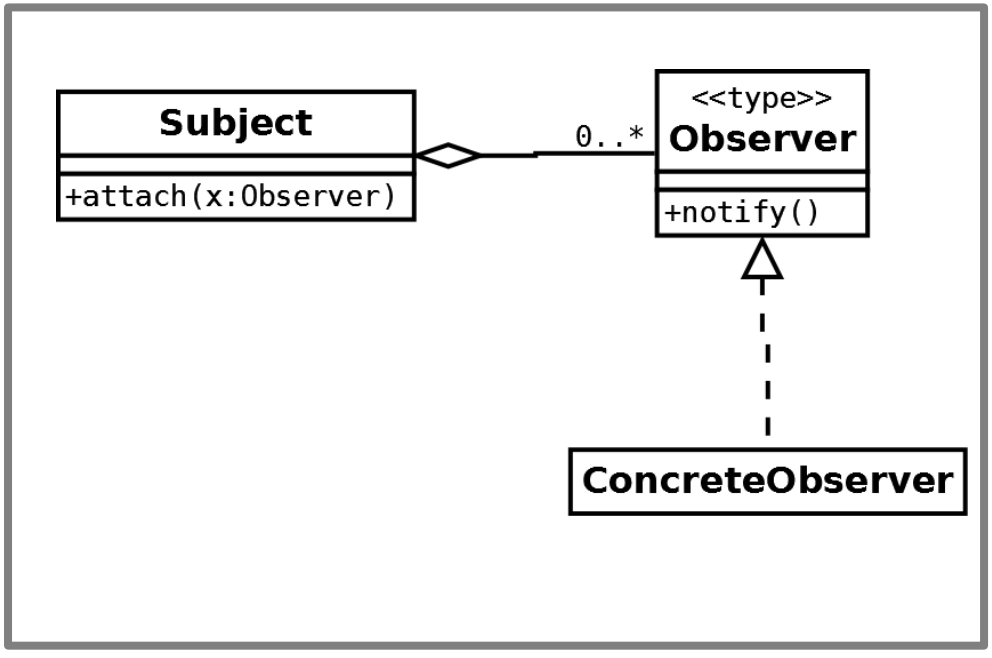
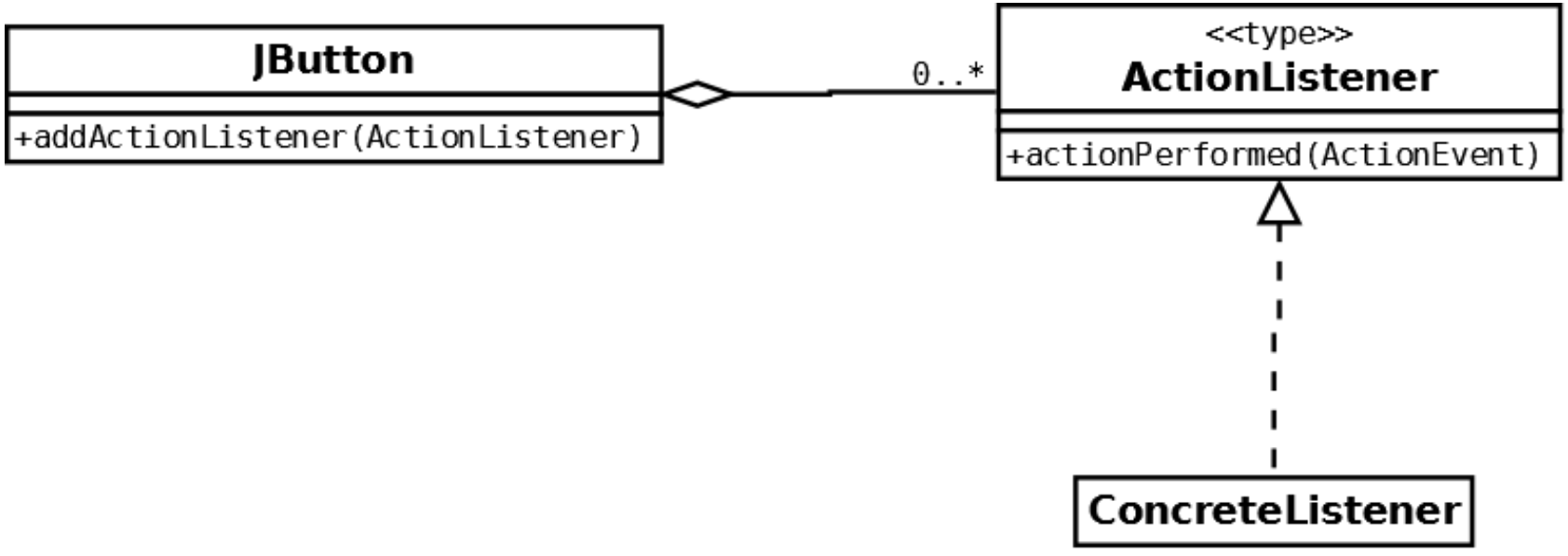


Figura 3: Diagramma UML del pattern OBSERVER.

Applicazione di OBSERVER



- La figura rappresenta il diagramma delle classi che lega la classe JButton all'interfaccia ActionListener
- C'è una corrispondenza diretta con lo schema proprio del pattern OBSERVER, riportato sulla slide precedente
- In particolare, JButton ricopre il ruolo di Subject e ActionListener quello di Observer

Supporto al pattern OBSERVER nella libreria standard

- Oltre che essere utilizzato dalle librerie grafiche, il pattern OBSERVER viene anche supportato dalla libreria standard tramite la classe **Observable** e l'interfaccia **Observer**, del package java.util

- L'interfaccia Observer rappresenta un generico osservatore
  - il suo contenuto è il seguente:

```
public interface Observer {
     void update(Observable o, Object arg);
}
```

- Il metodo update corrisponde a quello che nel pattern si chiama *notify*
  - cioè, è il metodo che l'oggetto osservato invocherà al verificarsi degli eventi di interesse
  - i due argomenti di update sono l'oggetto osservabile che sta chiamando e un argomento libero, che l'applicazione può utilizzare come vuole

Supporto al pattern OBSERVER nella libreria standard

- La classe Observable rappresenta un generico oggetto osservabile
  - i suoi metodi principali sono i seguenti:

```
public class Observable {
   public void addObserver(Observer o) { ... }
   public void notifyObservers(Object arg) { ... }
}
```

- Il metodo addObserver aggiunge un osservatore a questo oggetto
  - corrisponde al metodo attach del pattern
- Il metodo notifyObservers invoca il metodo update di tutti gli osservatori registrati, passando this come primo argomento e arg come secondo

- L'uso tipico di questa classe consiste nell'estenderla, in modo da aggiungere ad una nostra classe la capacità di essere osservata
- Altre classi nostre implementeranno Observer e fungeranno da osservatori
- In tal modo, non dovremo preoccuparci di implementare il meccanismo di registrazione e notifica degli osservatori
- Naturalmente, la classe che estende Observable perde la possibilità di utilizzare l'ereditarietà per altri scopi, visto che in Java è possibile estendere una sola classe