Marco Faella

Il paradigma Model-View-Controller. Il pattern Strategy.

Lezione n. 10

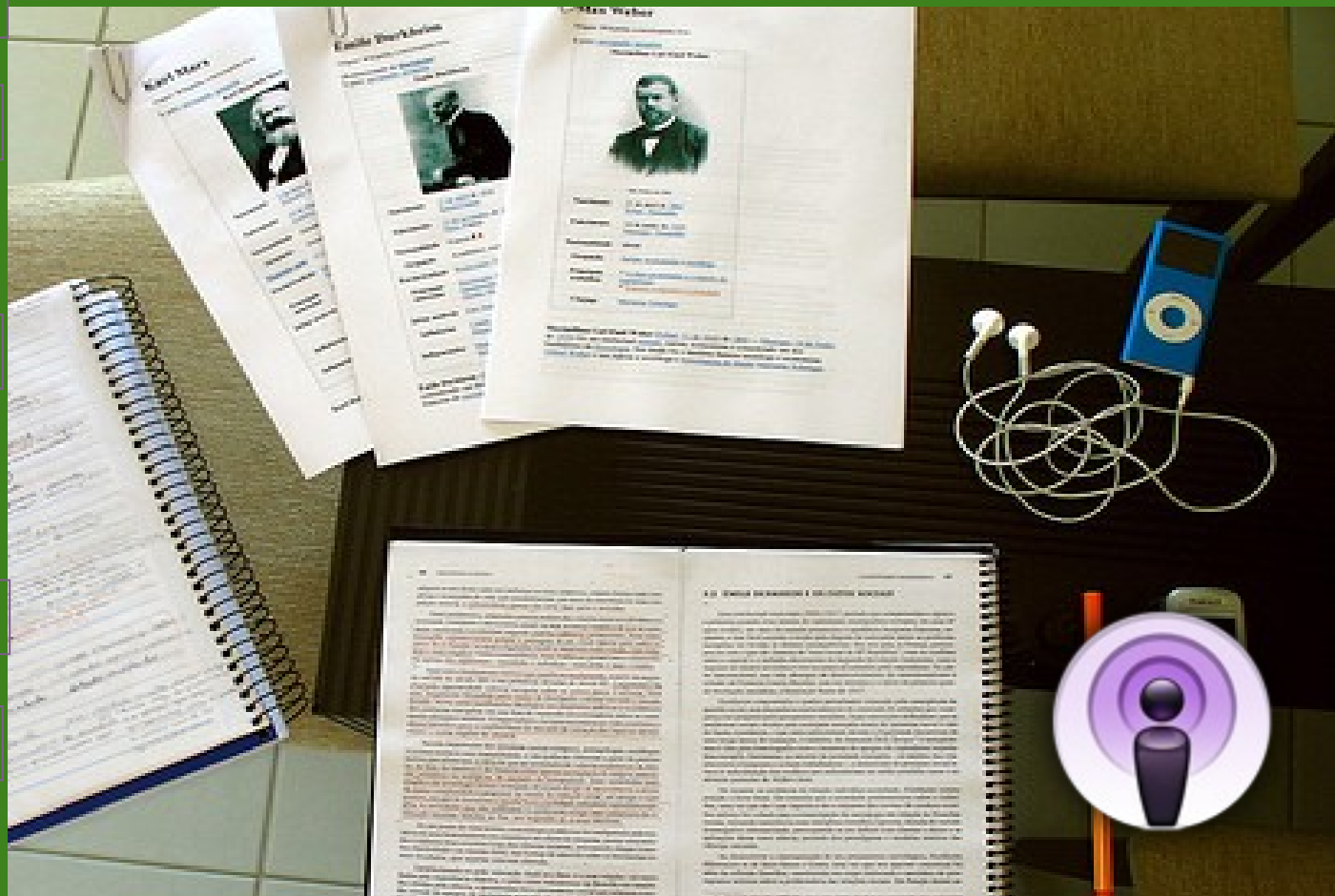
Parole chiave:
Java

Corso di Laurea:
Informatica

Insegnamento:
Linguaggi di Programmazione II

Email Docente:
faella.didattica@gmail.com

A.A. 2009-2010



Il paradigma Model-View-Controller

- Il paradigma MVC consiste nel distinguere, all'interno di un sistema informatico, tre tipologie di componenti:
  - quelle che rappresentano i dati in oggetto (Model),
  - quelle che si occupano di presentare i dati all'utente (View)
  - e quelle che si occupano dell'interazione con l'utente (Controller)

- Si tratta di un *pattern architetturale*, cioè di uno schema di progettazione di alto livello
- Tale paradigma trova la sua applicazione principale nei programmi con interfaccia grafica (GUI) e nei sistemi con interfaccia web
- Il paradigma detta la distinzione di responsabilità tra le tre categorie e il flusso di informazioni tra di esse

- E' stato proposto ufficialmente in un articolo del 1988 da Glenn Krasner e Stephen Pope (*"A cookbook for using the model-view controller user interface paradigm in Smalltalk-80"*)

- Il modello rappresenta i dati in sé, indipendentemente dal modo in cui questi saranno visualizzati
- Il modello non deve dipendere né dalle viste né dai controller con cui interagirà
- Tuttavia, il modello deve offrire le seguenti funzionalità:
  - per le **viste**, deve offrire metodi che espongano i dati in sola **lettura**
  - per i **controller**, deve offrire metodi che permettano di **modificare** i dati in risposta a delle richieste dell'utente
  - infine, in caso di cambiamento dei dati, deve **avvisare le viste**, affinché queste possano rispecchiare il cambiamento
  - quest'ultima funzionalità viene tipicamente realizzata seguendo il pattern OBSERVER, oggetto della lezione precedente

Esempio: un editor HTML

- Consideriamo l'esempio di un editor di pagine HTML
- Il **modello** è rappresentato dai caratteri che compongono la pagina
  - classe **HtmlPage**
- Possiamo immaginare diverse **viste** dei dati
  - una vista carattere per carattere
    - classe **HtmlRawView**
  - una vista *WYSIWYG* (what you see is what you get)
    - classe **HtmlRenderedView**
- **Controller**: l'observer che riceve gli eventi dall'utente e produce modifiche sul modello
  - classe **InsertListener**
- La figura a destra mostra il flusso di messaggi tipico del paradigma MVC
- Notiamo che il controller, ricevendo un evento dalla GUI, come l'inserimento di un carattere da parte dell'utente, effettua una modifica del Model

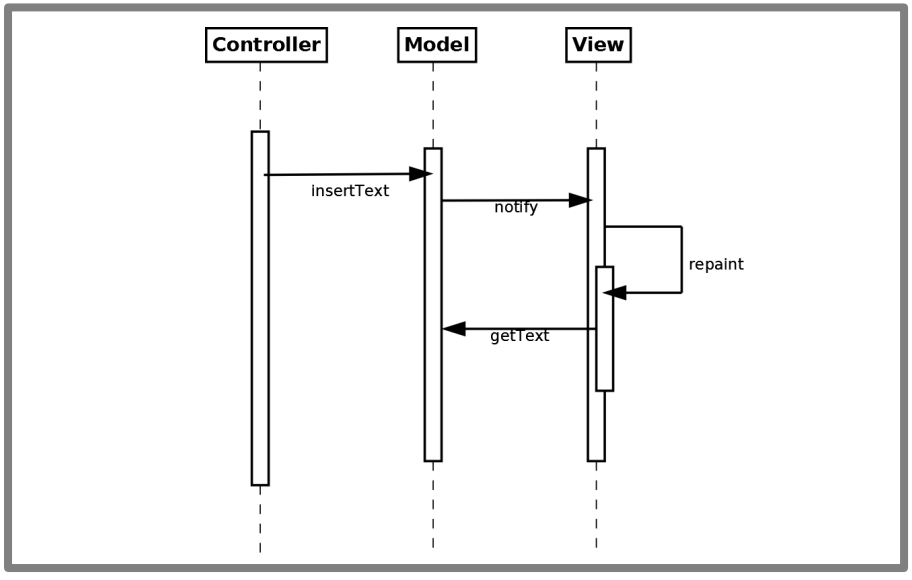
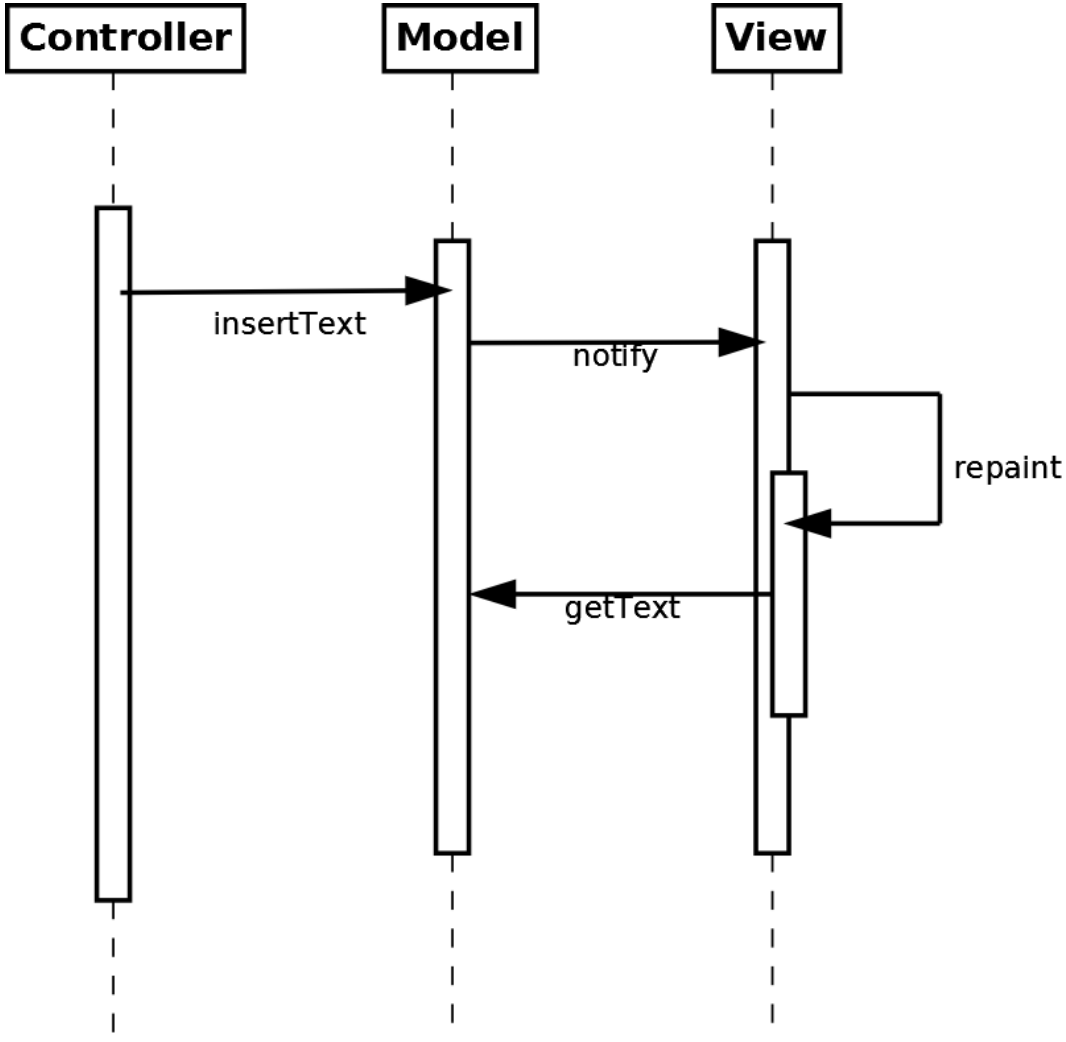


Figura 1: Diagramma di sequenza UML dell'interazione tra controller, modello e vista.

Esempio: un editor HTML



- Quindi, il controller non dialoga direttamente con le viste

- Nato per le interfacce grafiche, il paradigma MVC è stato poi applicato in altri domini, come le applicazioni basate sul web
- Molti framework per la realizzazione di applicazioni web si ispirano a MVC, ma lo hanno reinterpretato ed adattato alle loro esigenze

- Per approfondire l'argomento, si vedano anche altri pattern architetturali simili, come Model-View-Presenter (MVP)

Il pattern STRATEGY

- Il pattern **STRATEGY** si occupa di quelle situazioni in cui una classe accetta una procedura come argomento
- In un linguaggio orientato agli oggetti, questa procedura deve presentarsi sotto forma di oggetto di un'apposita classe

- **Contesto:**
  - Una classe (*Context*) può sfruttare diverse varianti di un algoritmo
  - I clienti della classe vogliono fornire versioni particolari dell'algoritmo

- **Soluzione:**
  - Definire un'interfaccia (*Strategy*) che rappresenti un'astrazione dell'algoritmo
  - Per fornire una variante dell'algoritmo, un cliente costruisce un oggetto di una classe (*ConcreteStrategy*) che implementa l'interfaccia Strategy e lo passa alla classe Context
  - Ogni volta che deve eseguire l'algoritmo, la classe Context invoca il corrispondente metodo dell'oggetto che concretizza la strategia

Diagramma del pattern STRATEGY

- La figura a destra rappresenta il diagramma delle classi UML proprio del pattern STRATEGY

- Come si vede, in questo caso il diagramma aggiunge ben poco alla descrizione della soluzione
- Il metodo doWork, il cui nome, come sempre, è puramente convenzionale, rappresenta il punto di accesso all'algoritmo
- Anche se non è mostrato dal diagramma, la classe Context deve avere un metodo per accettare un oggetto di tipo Strategy

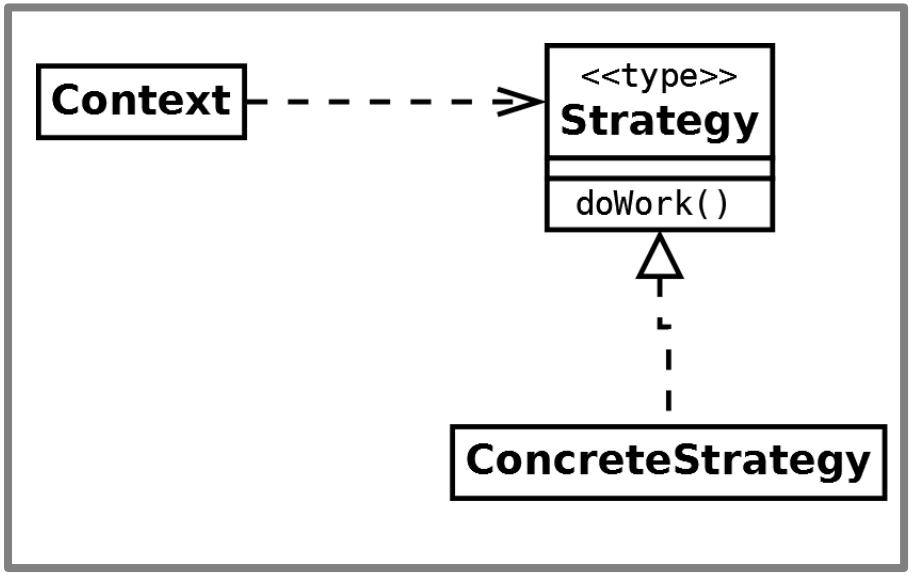


Figura 2: Diagramma UML del pattern STRATEGY.

Applicazione del pattern STRATEGY

- Abbiamo già incontrato un esempio di applicazione del pattern STRATEGY: nella **scelta del layout** di un componente grafico Swing/AWT
- Ricordiamo che per impostare la disposizione degli elementi grafici all'interno di un contenitore, si utilizza una chiamata del tipo:

```
contentPane.setLayout(new GridLayout(2,2));
```

- L'oggetto anonimo di tipo GridLayout rappresenta proprio la disposizione che si intende dare a questo pannello: in questo caso, una griglia di due righe e due colonne

- Oltre a GridLayout, la libreria standard fornisce altre classi di layout, tra cui:
  - **BoxLayout** dispone il contenuto su un'unica riga, o colonna
  - **FlowLayout** dispone il contenuto su più righe, modificando dinamicamente la disposizione quando il contenitore in questione viene ridimensionato

Applicazione del pattern STRATEGY

- Tutte le classi di layout implementano l'interfaccia *LayoutManager*, i cui metodi principali sono i seguenti:

  `Dimension` **`minimumLayoutSize`**`(Container parent)`
  restituisce le dimensioni minime richieste da questo layout

  `Dimension` **`preferredLayoutSize`**`(Container parent)`
  restituisce le dimensioni ideali secondo questo layout

  `void` **`layoutContainer`**`(Container parent)`
  dispone gli elementi secondo questo layout

- A questo punto, è facile vedere che siamo in presenza di un'applicazione del pattern Strategy, come evidenziato dalla slide successiva

Applicazione del pattern STRATEGY

- La figura a destra mostra il rapporto tra le classi coinvolte nella **scelta di un layout** per un contenitore Swing/AWT
- La classe *Container* ha un metodo per impostare un layout, il cui argomento è di tipo *LayoutManager*
  - la classe Container corrisponde alla classe Context del pattern
- L'interfaccia *LayoutManager* offre diversi metodi, legati alla realizzazione di un particolare layout
  - tale interfaccia corrisponde all'interfaccia *Strategy* del pattern
  - i suoi metodi sono tutti esempi del metodo "doWork" del pattern
- Infine, *GridLayout* è un esempio di classe concreta che implementa l'interfaccia

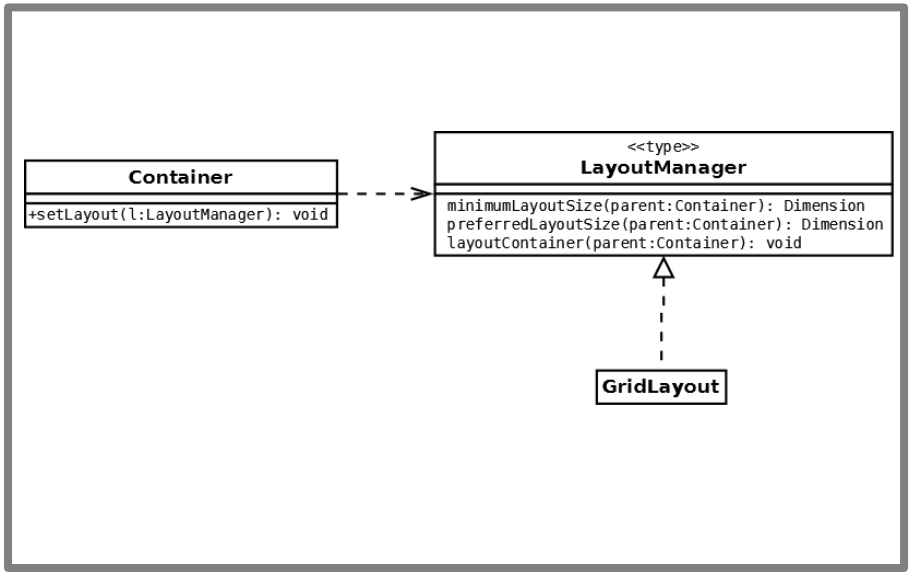


Figura 3: Diagramma UML relativo alla scelta di un layout per un contenitore Swing/AWT.

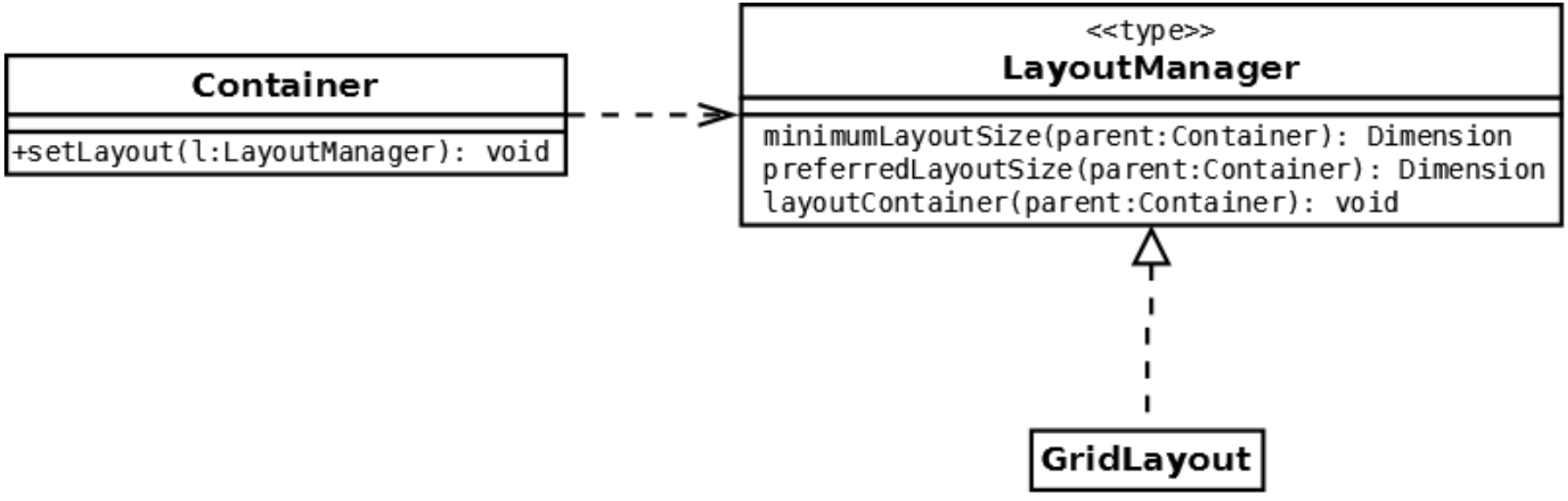


Figura 3: Diagramma UML relativo alla scelta di un layout per un contenitore Swing/AWT.

Altra applicazione del pattern STRATEGY

- Esaminiamo un'altra applicazione di STRATEGY che ritroviamo nella libreria standard
- La classe java.util.Collections contiene molti metodi di utilità, tra cui uno che permette di **ordinare una lista** di oggetti

```
public static void sort(List l, Comparator c)
```

- Questo metodo ordina la lista l utilizzando l'oggetto di tipo Comparator per effettuare i confronti tra gli oggetti presenti nella lista

- Quindi, l'interfaccia Comparator ricopre il ruolo di Strategy e sort è l'algoritmo che la utilizza

Mini-progetto: studio di funzioni

Proponiamo un esercizio piuttosto articolato, che permette di approfondire e mettere in pratica le nozioni apprese relativamente alle interfacce grafiche

Scopo: realizzare un'applicazione per lo **studio di funzioni parametriche** di una variabile reale

Ad esempio, parabole:

$a\,x^2 + b\,x + c$

con variabile x e parametri a, b, c

L'aspetto dovrebbe essere simile alla figura a destra

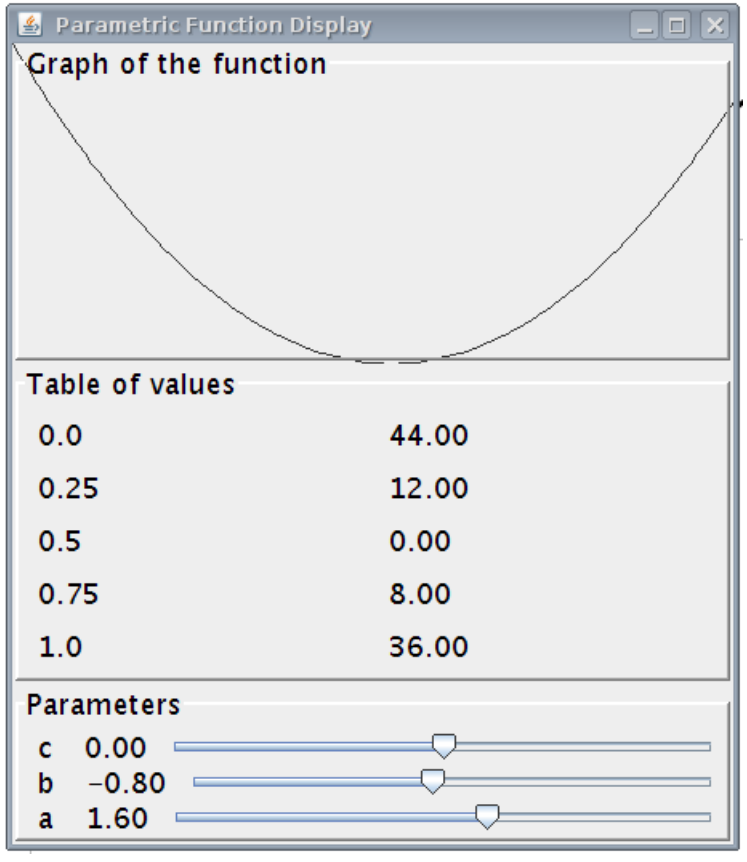


Figura 4: L'aspetto suggerito per l'applicazione da realizzare.

Mini-progetto: studio di funzioni

Il primo pannello rappresenta il grafico della funzione in esame sul piano cartesiano

Il secondo pannello presenta il valore della funzione per alcuni valori della sua incognita

Il terzo pannello permette di modificare i parametri della funzione, entro un certo intervallo (ad es., da -10 a 10)

Cambiando i parametri, vengono immediatamente aggiornati i primi due pannelli

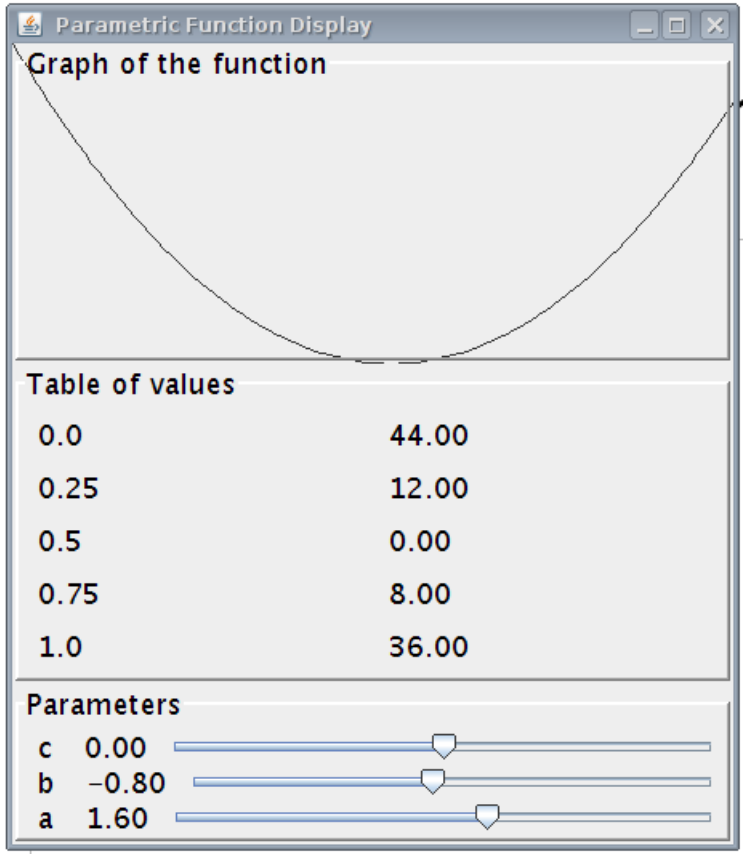


Figura 4: L'aspetto suggerito per l'applicazione da realizzare.

Mini-progetto: studio di funzioni

- Seguire il più possibile il framework MVC ed il pattern Observer
- In questo caso, il framework MVC si coniuga così:
  - Model: la funzione
  - View: il suo grafico, o una tabella di valori
  - Controller: le classi che rispondono allo spostamento degli slider e modificano i parametri della funzione

- Progettare l'applicazione in modo che possa funzionare con diverse tipologie di funzioni parametriche

- Basare l'applicazione su un'interfaccia che rappresenta una generica funzione parametrica:

```
public interface PFunction {
    // Returns the number of parameters
    int getNParams();
    // Sets the value of the i-th parameter to val
    void setParam(int i, double val);
    // Returns the current value of the i-th parameter
    double getParam(int i);
    // Returns the name of the i-th parameter
    String getParamName(int i);
    // Returns the value of this function on the argument x
    double eval(double x);
}
```

- Le classi che gestiscono le viste e i controller manipolano generici oggetti di tipo PFunction
- L'applicazione istanzia oggetti (un oggetto?) di tipo Parabola, che è una classe che implementa PFunction