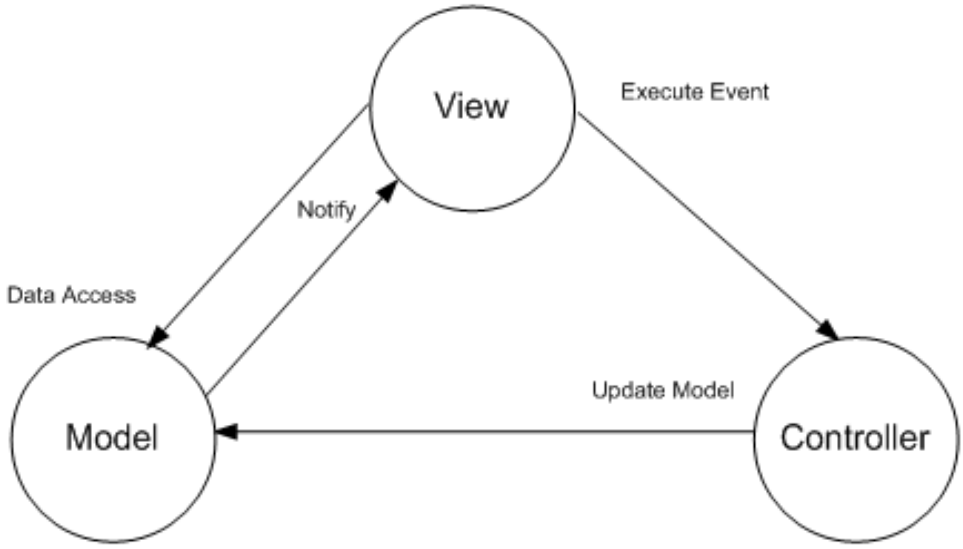
Approfondimento per l'esame
Ingegneria del Software (2016/2017)

Model View Controller: un'implementazione tramite la composizione dei design pattern Observer, Strategy e Composite

Tommaso Scarlatti
5784154

1. Introduzione

Il Model View Controller (MVC) è un pattern architetturale largamente diffuso nello sviluppo del software. Tradizionalmente è utilizzato per implementare delle GUI (Graphical User Interface) su applicazioni desktop, web e mobile.
L'MVC ha la capacità di dividere un'applicazione software in tre parti tra loro interconnesse, al fine di separare la rappresentazione interna dell'informazione dal modo in cui essa è presentata e dal modo in cui l'utente interagisce con essa.



1.1 Componenti

Come già accennato sopra, l'MVC è costituito da tre componenti:

- **Model:** è il componente principale del pattern. Esprime il comportamento dell'applicazione in un modo indipendente dalla sua rappresentazione, esponendone le funzionalità, incapsulando i dati e in particolare la logica di dominio.
- **View:** può essere una qualsiasi rappresentazione del modello, ed invia input al controller. Sono ammesse più rappresentazioni della della stessa informazione: ad esempio, un insieme di dati tabulari può essere ugualmente rappresentato sia tramite un istogramma che un grafico a torta.
- **Controller:** accetta input dall'utente e li converte in istruzioni per il model.

1.2 Interazioni

Oltre a scomporre l'applicazione in tre parti distinte, l'MVC definisce le interazioni che sussistono tra essi:

- Il *model* mantiene un'informazione. Questa è modificata secondo le istruzioni che arrivano dal controller ed è rappresentata tramite la view.
- La *view* si aggiorna ogni qual volta il model modifica il proprio stato.
- Il *controller* invia comandi al model per aggiornare il suo stato, mappandoli direttamente dalle azioni dell'utente. Può inoltre inviare direttamente istruzioni alla view associata, in modo da poter modificare il modo in cui il model è rappresentato.

1.3 Vantaggi e svantaggi

- **Supporta view multiple:** la struttura dell'MVC permette alla view di essere completamente separata dal model. In questo modo la user interface può mostrare più views dello stesso model ad un determinato istante.
- **Adatto ai cambiamenti:** la GUI tende molto spesso a cambiare più frequentemente del modello, e dato che quest'ultimo non dipende dalle varie views, aggiungerne una nuova all'applicazione non comporta nessun cambiamento.
  Lo scope del cambiamento è quindi "confinato" all'interno del contesto della view.
- **Complessità:** il pattern aumenta leggermente la complessità della soluzione, introducendo nuovi livelli di dipendenza tra i componenti. Inoltre incrementa la natura "event-driven" della user interface, rendendo le pratiche di debugging più complicate.
- **Costo di update frequenti:** sebbene il model non sia vincolato alle varie views, il programmatore deve sempre stare attento alla connessione tra i due componenti. Qualora il model venga modificato, è necessario inviare una richiesta di update alle views. Se il model viene modificato in maniera troppo frequente, le views potrebbero essere inondate di richieste e questo potrebbe portare ad un malfunzionamento dell'applicazione a causa del carico eccessivo di lavoro necessario.

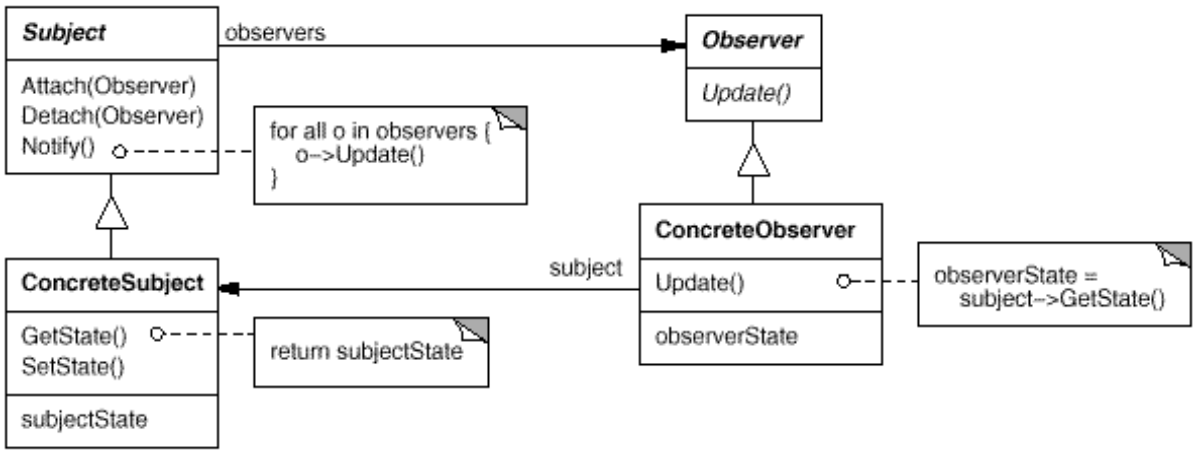
2. Design pattern coinvolti

Si può pensare al Model View Controller come ad una composizione di design patterns. Guardando alle tre componenti:

- *Model* usa il pattern **Observer** per mantenere le views e i controllers allineati con il proprio stato.
- *View* e *Controller* utilizzano il pattern **Strategy**: possiamo pensare infatti alla view come il contesto in cui i vari controller, che incapsulano un determinato comportamento, possono agire.
- *View* inoltre utilizza internamente il pattern **Composite** per gestire in modo uniforme i vari elementi che compongono la user interface.

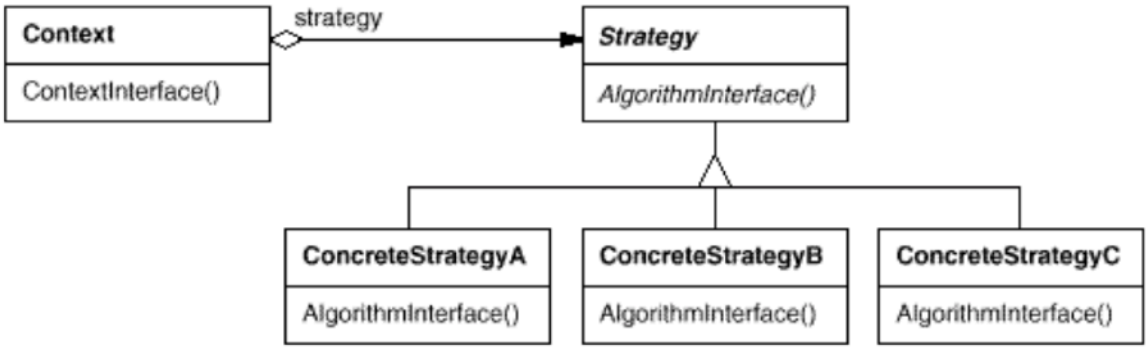
Di seguito si riportano i diagrammi UML dei tre pattern coinvolti, presi direttamente dal libro "Design Patterns" della celebre "Gang of Four", e una breve descrizione delle loro principali funzionalità.

2.1 Observer



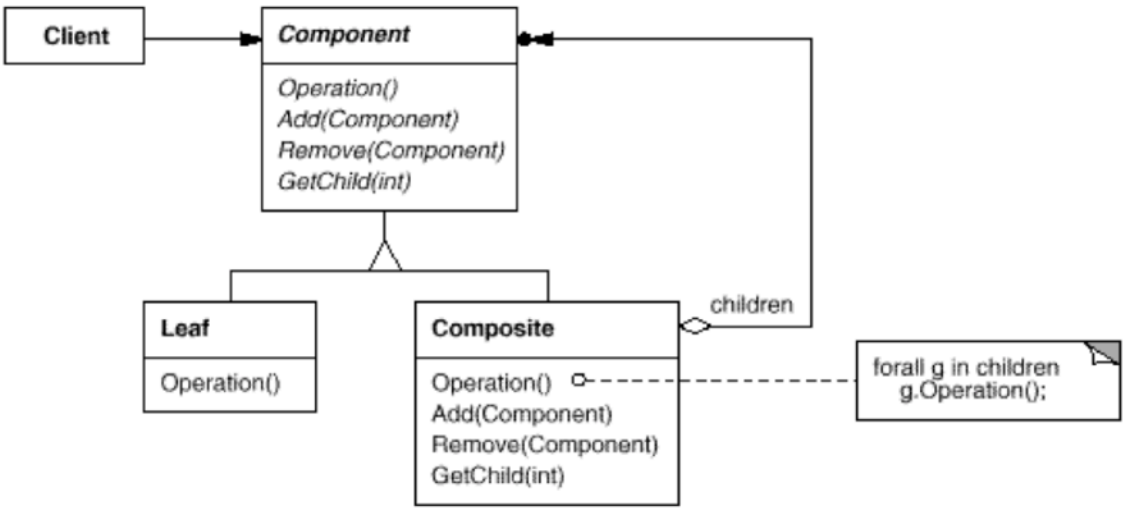
Il pattern **Observer** permette di definire una dipendenza uno a molti fra oggetti, in modo tale che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti da questo siano notificati e aggiornati automaticamente.

2.2 Strategy



Il pattern **Strategy** permette di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. Strategy permette inoltre agli algoritmi di variare indipendentemente dai client che ne fanno uso.

2.3 Composite

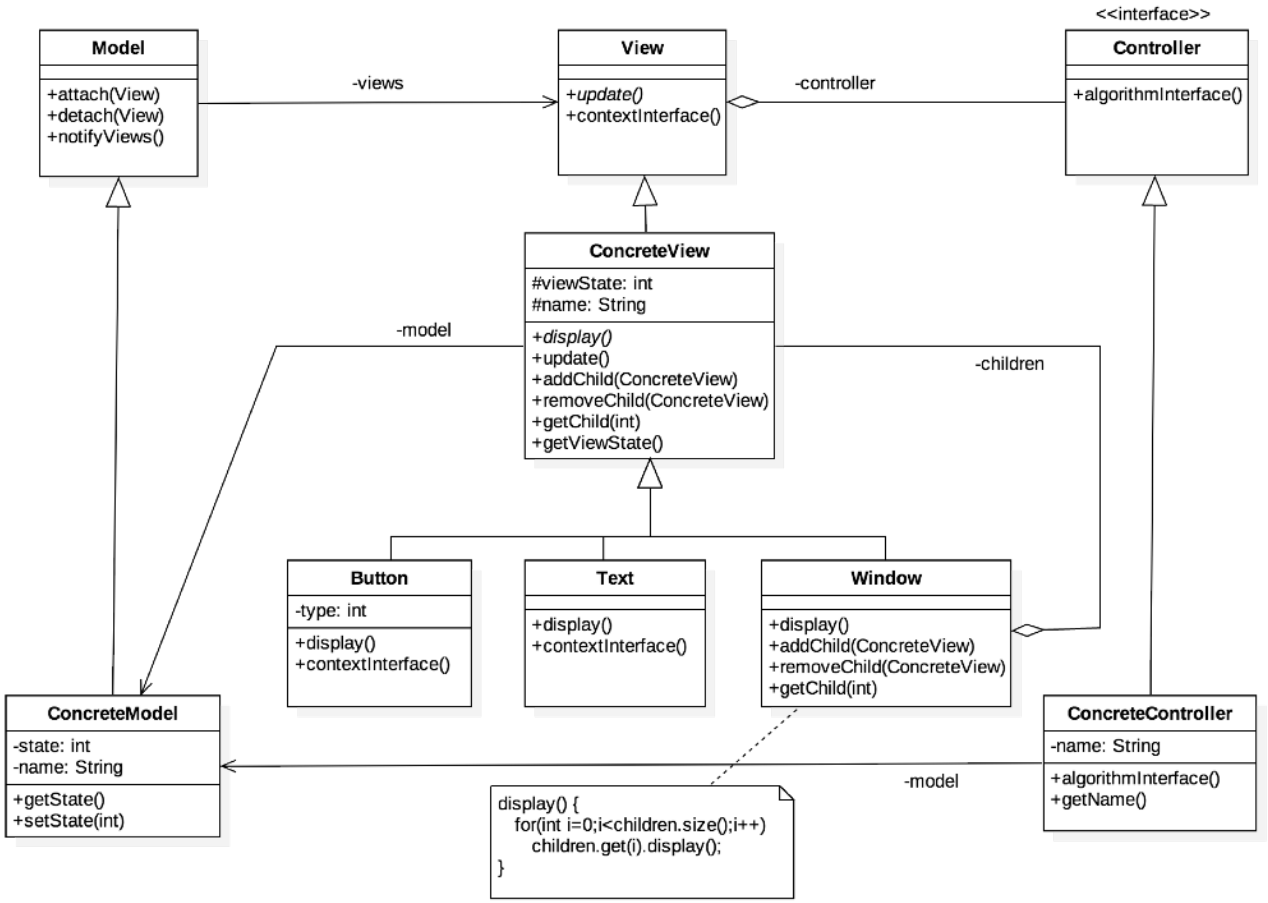


Il pattern **Composite** permette di comporre oggetti in strutture ad albero per rappresentare gerarchie parte-tutto e consentire al client di trattare oggetti singoli e composizioni in modo uniforme.

3. Implementazione

Di seguito si riporta il diagramma UML della composizione dei tre pattern, specificando per ogni classe il ruolo che essa assume all'interno di uno o più pattern. Nella sezione 3.3 ci si focalizza sui dettagli e le scelte di questa particolare implementazione, ricorrendo all'uso di un mockup.

3.1 Diagramma UML



3.2 Partecipanti

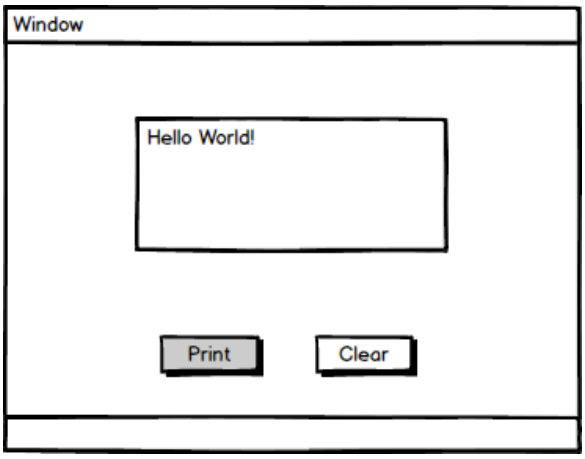
- **Model:** rappresenta il *Subject* nel pattern Observer. E' una classe astratta ed espone i metodi fondamentali per registrare, eliminare e notificare le views collegate.
- **ConcreteModel:** rappresenta il *ConcreteSubject* nel pattern Observer. Contiene lo stato del modello a cui le views sono interessate. Per semplicità, in questa implementazione, è stato definito come un numero intero.

- **View:** rappresenta simultaneamente l'*Observer* nel pattern Observer e il *Context* nel pattern Strategy. E' una classe astratta, contiene un riferimento ad un oggetto *Controller* ed è configurato con un oggetto *ConcreteController*. Su quest'ultimo chiama il metodo *algorithmInterface()* tramite l'utilizzo del metodo *contextInterface(),* che ha l'intento di rappresentare l'interazione dell'utente (*Client*) con la view.
- **ConcreteView:** rappresenta il *Component* nel pattern Composite, ed ereditando da View è a sua volta un *Observer.* E' una classe astratta, dichiara l'interfaccia per tutti gli oggetti che fanno parte della composizione e fornisce un'implementazione "fake" dei metodi child-related. Ha un riferimento ad un *ConcreteModel* e fornisce un'implementazione per il metodo *update().* Definisce inoltre un metodo astratto *display()* che verrà implementato dalle sottoclassi.
- **Button:** rappresenta una *Leaf* del pattern Composite e fornisce un'implementazione concreta per il metodo *display()*.
- **Text:** come *Button,* anch'essa rappresenta una *Leaf* del pattern composite e fornisce una particolare implementazione del metodo *display()* per mostrare a video una stringa di testo generata sulla base dello stato del *ConcreteModel* a cui punta.
- **Window:** rappresenta il *Composite* del pattern Composite. Esegue l'override dei metodi child-related della classe *ConcreteView* e memorizza in una lista i componenti figli. Implementa il metodo *display()* effettuandone il forwarding su tutti i figli.
- **Controller:** rappresenta lo *Strategy* nel pattern Strategy. Dichiara un'interfaccia comune a tutti i *ConcreteController* supportati.
- **ConcreteController:** rappresenta il *ConcreteStrategy* nel pattern Strategy ed implementa l'algoritmo caratterizzante *algorithmInterface()* definito nell'interfaccia.

3.3 In dettaglio

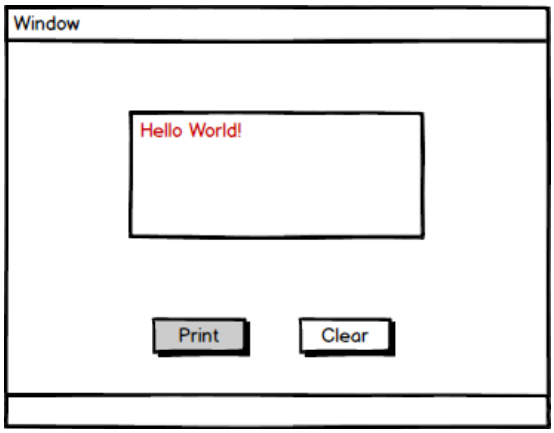
Per evidenziare le funzionalità del Model View Controller è stato scelto uno scenario semplicistico in cui è presente un unico modello concreto, il cui stato può assumere soltanto tre valori interi: 0, 1 o 2.
L'interfaccia grafica è formata da tre componenti fondamentali: una finestra (*Window*), due bottoni (*Button*) e infine un'area di testo modificabile (*Text*).
L'utente può interagire con l'applicazione solo mediante i due bottoni "Print" e "Clear". Ad ognuno dei due bottoni è stato associato inizialmente un *ConcreteController* differente, in particolare al bottone "Print" è stato associato *ConcreteControllerOne*, mentre al bottone "Clear" è stato associato *ConcreteControllerZero*. E' inoltre disponibile un terzo controller: *ConcreteControllerRed*. I controller differiscono tra loro soltanto nell'implementazione di *algorithmInterface()*, ponendo lo stato del model rispettivamente a 1, 0 e 2.

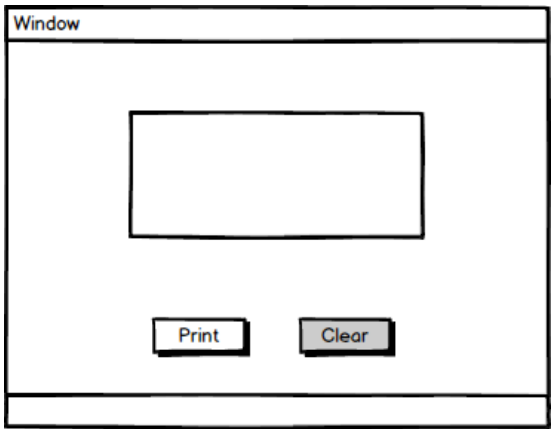
Di seguito si riporta uno schizzo dell'interfaccia grafica dell'applicazione, realizzato con il software "Balsamiq Mockups 3".



Questo particolare mockup si riferisce al caso in cui lo stato del model è pari ad 1. L'oggetto Text tramite il metodo *display()* stampa una stringa "Hello World!", allo stesso tempo il bottone "Print" è nello stato "OFF", ossia non è possibile cliccarlo.
Se lo stato del model è pari a 0, invece, non viene visualizzato nessun messaggio, ed è lo stato del bottone "Clear" ad essere "OFF" questa volta.
Nel caso in cui lo stato del model sia pari a 2, si visualizza la stessa situazione del primo caso, fatta eccezione per il colore rosso della scritta "Hello World!".

Per completezza riportiamo qui sotto anche gli altri due casi:





4. Codice

Di seguito si elencano tutte le classi utilizzate nell'implementazione. Per ogni classe viene indicato il nome e il ruolo che implementa in uno o più pattern.

Model (Subject)

```
package mvc;

import java.util.ArrayList;

public abstract class Model {

public ArrayList<View> views;
	public Model() {
		views = new ArrayList<View>();
	}

	public void notifyViews() {
		for(int i=0;i<views.size();i++) {
			views.get(i).update();
		}
	}

	public void attach(View v) {
		views.add(v);
	}

	public void detach(View v) {
		views.remove(v);
	}
}
```

ConcreteModel (ConcreteSubject)

```
package mvc;

public class ConcreteModel extends Model {
	private int state;
	private String name;

	public ConcreteModel(int initialState, String name) {
		super();
		state = initialState;
		this.name = name;
```

```
	}

	public String getName() {
		return name;
	}

	public int getState() {
		return state;
	}

	public void setState(int state) {
		this.state = state;
		notifyViews();
	}
}
```

View (Observer)

```
package mvc;

public abstract class View {
	protected Controller controller;

	public abstract void update();

	protected View() {
	}

	protected View(Controller controller) {
		this.controller = controller;
	}

	public void setController(Controller c) {
		controller = c;
	}

	public void contextInterface() {
		controller.algorithmInterface();
	}
}
```

ConcreteView (ConcreteObserver, Component)

```
package mvc;

import java.lang.Exception;
```

```
public abstract class ConcreteView extends View {
	protected String name;
	protected ConcreteModel model;
	protected int viewState;

	public ConcreteView(ConcreteModel model) {
		this.model = model;
		model.attach(this);
		viewState = model.getState();
	}

	public abstract void display();

	public int getViewState() {
		return viewState;
	}

	public void update() {
		viewState = model.getState();
	}

	public void addChild(ConcreteView cv) throws Exception {
		throw new Exception("addChild method used in a leaf!");
	}

	public void removeChild(ConcreteView cv) throws Exception {
		throw new Exception("removeChild method used in a
		leaf!");
	}

	public ConcreteView getChild(int i) throws Exception {
		throw new Exception("getChild method used in a leaf!");
	}
}
```

Window (Composite)

```
package mvc;

import java.util.ArrayList;

public class Window extends ConcreteView {
	private ArrayList<ConcreteView> children;

	public Window(ConcreteModel cm, String name) {
```

```
            super(cm);
            this.name = name;
            children = new ArrayList<ConcreteView>();
      }

      public String getName() {
            return name;
      }

      @Override
      public void addChild(ConcreteView cv) {
            children.add(cv);
      }

      @Override
      public void removeChild(ConcreteView cv) {
            children.remove(cv);
      }

      @Override
      public ConcreteView getChild(int i) {
            return children.get(i);
      }

      @Override
      public void display() {
            System.out.println("Visualize " + name);
            for(int i = 0; i < children.size(); i++)
                  children.get(i).display();
      }

}
```

Button (Leaf)

```
package mvc;

public class Button extends ConcreteView {
      private int type; // Useful to distinguish between button
                        types

      public Button(ConcreteModel cm, String name, int type) {
            super(cm);
            this.name = name;
            this.type = type;
      }
```

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

	@Override
	public void contextInterface() {
		if(viewState == 0)
			System.out.println("Action not allowed! " + name +
			" is OFF!");
		else
			controller.algorithmInterface();
	}

	@Override
	public void display() {
		String state; // Maps viewState to a String type (1,2 ==
		ON | 0 == OFF)

		if (type == 1) {
			if (model.getState() == 0)
				viewState = 1;
			else
				viewState = 0;
		}
		if(viewState > 0)
			state = "ON";
		else
			state = "OFF";

		System.out.println("Visualize " + name + ". Is in state "
		+ state);
	}
}
```

Text (Leaf)

```
package mvc;

public class Text extends ConcreteView {

	public Text(ConcreteModel cm, String name) {
		super(cm);
		this.name = name;
	}
```

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

	@Override
	public void contextInterface() {
		System.out.println("Do nothing...");
	}

	@Override
	public void display() {
		if(viewState == 0)
			System.out.println("*No text displayed!* (Model
			state = 0)");
		else if(viewState == 1)
			System.out.println("Hello World! (Model state =
			1)");
		else
			System.out.println("RED Hello World! (Model state =
			2)");
	}
}
```

Controller (Strategy)

```
package mvc;

public interface Controller {
	public void algorithmInterface();
}
```

ConcreteController (ConcreteStrategy)

```
package mvc;

public class ConcreteControllerOne implements Controller {
	private String name;
	private ConcreteModel model;

	public ConcreteControllerOne(ConcreteModel cm, String name) {
		model = cm;
		this.name = name;
	}

	public String getName() {
```

```
            return name;
      }

      @Override
      public void algorithmInterface() {
            System.out.println(name + " activated!");
            model.setState(1);
      }
}
```

Client

```
package mvc;

public class Client {

      public static void main(String[] args) throws Exception {

            /* Objects creation */
            Model m = new ConcreteModel(0, "Model");
            ConcreteView pb = new Button(m, "Print Button", 1);
            ConcreteView cb = new Button(m, "Clear Button", 0);
            ConcreteView t = new Text(m, "Text");
            ConcreteView w = new Window(m, "Window");
            Controller cc1 = new ConcreteControllerOne(m,
            "ControllerOne");
            Controller cc0 = new ConcreteControllerZero(m,
            "ControllerZero");
            Controller ccr = new ConcreteControllerRed(m,
            "ControllerRed");

            /* Add ConcreteControllers to ConcreteViews */
            pb.setController(cc1);
            cb.setController(cc0);
            t.setController(cc1);
            w.setController(cc1);

            /* Add children to the composite Window */
            w.addChild(pb);
            w.addChild(cb);
            w.addChild(t);

            /* Display initial view */
            w.display();
            System.out.println("");
```

```
            /* Press "Print Button" */
            pb.contextInterface();
            w.display();
            System.out.println("");

            /* Press "Clear Button" */
            cb.contextInterface();
            w.display();
            System.out.println("");

            /* Change ConcreteController and press "Print Button" */
            pb.setController(ccr);
            pb.contextInterface();
            w.display();
            System.out.println("");

            /* Try to push an OFF button */
            pb.contextInterface();

      }

}
```

L'output che si ottiene dalla classe Client è il seguente:

```
Visualize Window
Visualize Print Button. Is in state ON
Visualize Clear Button. Is in state OFF
*No text displayed!* (Model state = 0)

ControllerOne activated!
Visualize Window
Visualize Print Button. Is in state OFF
Visualize Clear Button. Is in state ON
Hello World! (Model state = 1)

ControllerZero activated!
Visualize Window
Visualize Print Button. Is in state ON
Visualize Clear Button. Is in state OFF
*No text displayed!* (Model state = 0)

ControllerRed activated!
Visualize Window
Visualize Print Button. Is in state OFF
Visualize Clear Button. Is in state ON
RED Hello World! (Model state = 2)

Action not allowed! Print Button is OFF!
```