Ingegneria del Software 1

07 - Model View Controller

Martedì, Aprile 4, 2016

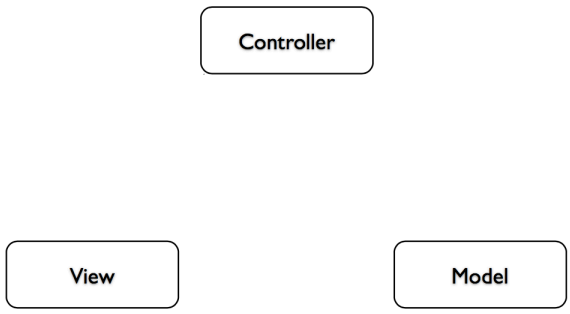
Claudio Menghi, Alessandro Rizzi

April 5, 2017

Contents

1 Model View Constroller

Il modello Model View Controller è un pattern architettuale originariamente proposto da Trygve Reenskaug nel 1979 durante una visita allo Xerox Palo Alto Research Center (PARC). Il model view controller viene inzialmente creato come una soluzione al problema di fornire agli utenti il controllo delle loro informazioni quando visualizzate su perspectives multipli, in particolare quando si devono trattare data-set di grandi dimensioni.



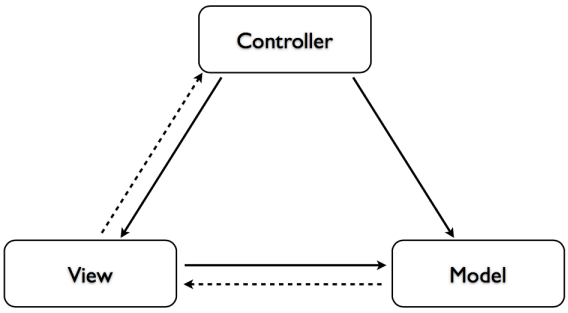
- *Model*: è una rappresentazione astratta del sistema che si desidera modellare.
  - Il modello include i dati assieme ai metodi (logica) necessari per processare questi dati.
  - Il modello non include nessuna informazione di *come* i dati sono visualizzati
- *View* (Interface): è solitamente attaccata ad un modello. In genere, ad un determinato modello possono essere attaccate una o più Views, ogni view è capace di mostrare una o più rappresentazione del modello (esempio una rappresentazione su schermo o cartacea). La view consente di:
  - mostrare dei dati a un utente
  - essere aggiornata quando il modello cambia
- *Controller* (Coordinator): riceve gli input dalla view e li trasforma in azioni che il modello può eseguire. Delle azioni potrebbero essere il click su un bottone nel caso di una GUI o delle richieste GET e POST in una applicazione web. Il controller in alcuni casi potrebbe selezionare nuove view, per esempio nel caso delle richieste HTTP.

Interazione View-Model. Possono esserci due tipi di interazioni tra view e model:

- *push model* la view si registra al modello e aspetta notifiche
- *pull model* la view è responsabile di eseguire delle query sul modello quando deve aggiornare la sua visualizzazione

1.1 Interazione "classica" tra i componenti dell'MVC

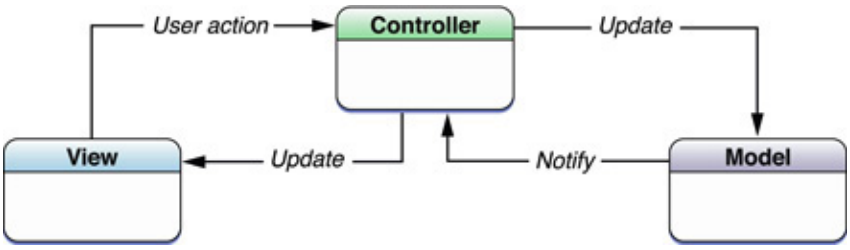
Una delle possibili interazioni del Model View Controller (quella proposta inizialmente nel 1979) è la seguente: La *view* osserva (pattern observer o listener) il modello. Ogni cambiamento del modello viene propagato come una notifica che la view riceve. Nota il modello non è direttamente consapevole di chi osserva, semplicemente manda un messaggio agli ascoltatori. Il *controller* è azionato dalla view (o da una sua logica interna). Quando



un azione viene eseguita sulla view, una notifica viene mandata al controller. Il controller osserva la view (mediante un osservatore o un listener). Il controller conosce il modello sottostante. Quando un azione viene eseguita sulla view (un comando chiamato sull'interfaccia del server), l'azione viene segnalata al controller, il controller accede al modello e lo aggiorna in relazione all'azione dell'utente. *In alcune architetture il controllore potrebbe anche essere responsabile di cambiare la view, per esempio nelle architetture enterprise di java.*

1.2 Modifiche all'MVC

Una modifica al pattern MVC utilizzata nei sistemi Apple consiste nel posizionare il controller tra la view e il modello. La differenza principale tra questo framework e la versione classica è che le notifica del cambiamento dello stato del modello sono comunicate alla view *attraverso* il controller. Il controller fa da "mediatore" tra i dati del modello e la view in entrambe le direzioni: le azioni sulla view devono passare per il controller per essere tramutate in azioni del modello. I cambiamenti del modello sono comunicati alla view dopo che sono passati per il controllore. In questo caso, la view chiama un metodo sul controllore, il controllore esegue l'azione sul modello, se l'azione ha provocato dei cambiamenti il modello notifica i suoi ascolatori: i controller, i quali a loro volta notificano la modifica alla view. Il controllore può essere suddiviso in altri sottocontrollori.



Per maggiori informazioni potete fare riferimento a `https://developer.apple.com/library/mac/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html`.

1.3 MVC client/Server

In un architettura client/server l'MVC può essere visto a diversi livelli di astrazione a seconda della granularità con cui si affronta il problema. Se si considera "l'applicazione finale" la view rappresenta l'interfaccia con cui il client (utenti) interagisce. In questo contesto esistono due approcci:

- *server-side* MVC: è il caso in cui model, view e controller sono eseguiti sul sever. È per esempio il caso delle pagine HTML (View) che vengono generate sul server
- *client-side* MVC: è il caso in cui model e controller sono posti sul client. È il caso per esempio delle applicazioni per tablet.

Se invece consideriamo come "granularità" il server, i nostri client (utenti) sono le persone che useranno le API, ovvero le interfacce della nostra applicazione (View) che esponiamo, il controllore è il coordinatore che gestisce l'interazione tra le view e il modello che contiene lo stato del gioco.

2 Esercizi

2.1 Esercizio 1: MVC

Gestire il seguente gioco: Dei carcerati sono in una prigione. I poliziotti della prigione decidono di salvare i carcerati se dimostrano una capacità collaborativa e una certa attitudine nel risolvere problemi. I prigionieri vengono lasciati nel cortile della prigione per un determinato tempo al fine di accordarsi relativamente a una strategia per salvarsi. Dal giorno seguente in poi un prigioniero alla volta verrà fatto entrare in una cella t contenente un interruttore. Un prigioniero può entrare più volte nella cella prima che un altro prigioniero entri e ad intervalli di tempo non prefissati. L'interruttore può essere in due stati ON e OFF. Ogni prigionierio può muovere l'interruttore da ON a OFF e viceversa o lasciarlo nello stato attuale. Tra un prigioniero e l'altro l'interruttore non viene toccato dai poliziotti. L'unica informazione che si conosce è che l'interruttore è inizialmente a OFF. Il gioco continua finchè uno dei prigionieri dice: "ogni prigioniero è stato nella cella prima di me *almeno* una volta". Se è corretto tutti i prigionieri sono liberati altrimenti vengono uccisi.

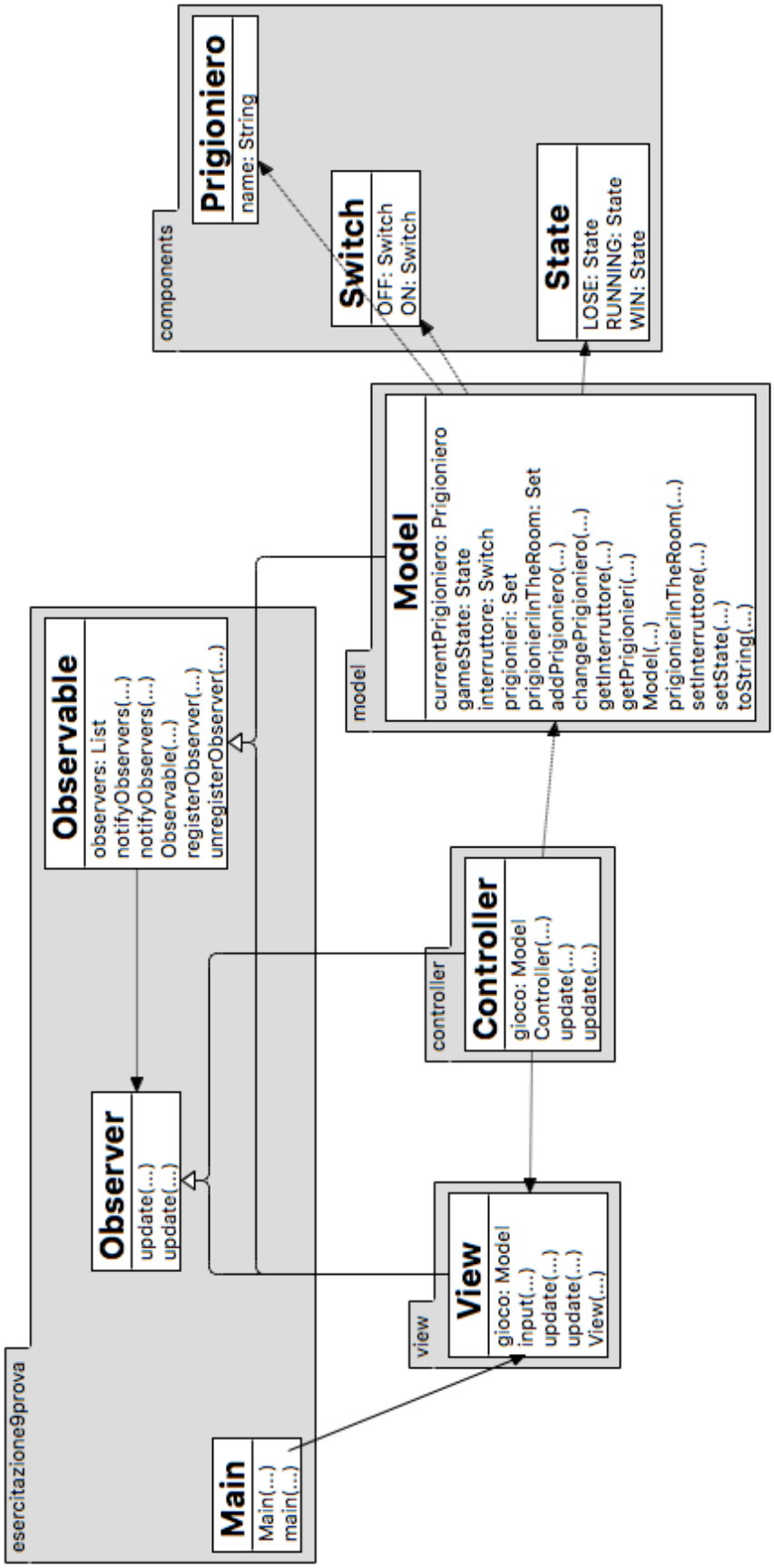


Figure 1: Class diagram relativo alla soluzione dell'esercizio 1

2.1.1 Observable/observer

```
public abstract class Observable {

	private List<Observer> observers;

	public Observable(){
		observers=new ArrayList<Observer>();
	}

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

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

	public void notifyObservers(){
		for(Observer o: this.observers){
			o.update();
		}
	}
	public <C> void notifyObservers(C c){
		for(Observer o: this.observers){
			o.update(c);
		}

	}
}
```

```
public interface Observer {

	public void update();
	public <C> void update(C change);
}
```

2.1.2 Model

```
public enum State {
	RUNNING, WIN, LOSE
}
```

```
public enum Switch {
	ON, OFF
}
```

```
public class Prigioniero {
```

```
	private final String name;

	public Prigioniero(String name){
		this.name=name;
	}

	@Override
	public String toString() {
		return "Prigioniero [name=" + name + "]";
	}

	public String getName() {
		return name;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((name == null) ? 0 : name.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Prigioniero other = (Prigioniero) obj;
		if (name == null) {
			if (other.name != null)
				return false;
		} else if (!name.equals(other.name))
			return false;
		return true;
	}
}
```

```
public class Model extends Observable{

	private Set<Prigioniero> prigionieri;
	private Set<Prigioniero> prigionieriInTheRoom;
	private Prigioniero currentPrigioniero;
	private Switch interruttore;
	private State gameState;

	public Model() {
		prigionieri=new HashSet<Prigioniero>();
		prigionieriInTheRoom=new HashSet<Prigioniero>();
		this.addPrigioniero(new Prigioniero("Luca"));
```

```
		this.addPrigioniero(new Prigioniero("Pietro"));
		this.addPrigioniero(new Prigioniero("Paolo"));
		this.interruttore=Switch.OFF;
		gameState=State.RUNNING;
		this.changePrigioniero();
	}

	public void setState(State state){
		this.gameState=state;
		System.out.println("I am the model I am notifying my observers with a new
		    game state");

		this.notifyObservers(gameState);
	}

	public void addPrigioniero(Prigioniero prigioniero){
		currentPrigioniero=prigioniero;
		this.prigionieri.add(prigioniero);
	}

	public Switch getInterruttore() {
		return interruttore;
	}

	public Set<Prigioniero> getPrigionieri(){
		return Collections.unmodifiableSet(this.prigionieri);
	}
	public Set<Prigioniero> prigionieriInTheRoom(){
		return Collections.unmodifiableSet(this.prigionieriInTheRoom);
	}


	public void setInterruttore(Switch value){
		this.interruttore=value;

	}

	public void changePrigioniero(){
		List<Prigioniero> prigionieriList=new ArrayList<Prigioniero>(prigionieri);
		Collections.shuffle(prigionieriList);
		this.currentPrigioniero=prigionieriList.get(0);
		this.prigionieriInTheRoom.add(currentPrigioniero);
		System.out.println("I am the model I am notifying my observers");

		this.notifyObservers();
	}

	@Override
	public String toString() {
		return "Gioco [currentPrigioniero=" + currentPrigioniero
				+ ", interruttore=" + interruttore + "]";
	}
}
```

2.1.3 Azioni

```
public abstract class Action {

	private final Model gioco;

	public Action(Model gioco){
		this.gioco=gioco;
	}

	protected Model getGioco(){
		return this.gioco;
	}

	public abstract void esegui();
}
```

```
public class Scommetti extends Action{

	public Scommetti(Model gioco) {
		super(gioco);
	}

	@Override
	public void esegui() {
		if(this.getGioco().prigionieriInTheRoom().containsAll(this.getGioco().getPrigionieri())){
			this.getGioco().setState(State.WIN);
		}
		else{
			this.getGioco().setState(State.LOSE);
		}

	}
}
```

```
public class TurnOff extends Action{

	public TurnOff(Model gioco) {
		super(gioco);
	}

	@Override
	public void esegui() {
		this.getGioco().setInterruttore(Switch.OFF);
		this.getGioco().changePrigioniero();
	}
}
```

```
public class TurnOn extends Action{

	public TurnOn(Model gioco) {
```

```
            super(gioco);
      }

      @Override
      public void esegui() {
            this.getGioco().setInterruttore(Switch.ON);
            this.getGioco().changePrigioniero();
      }
}
```

2.1.4 Controller

```
public class Controller implements Observer {

      private final Model gioco;

      public Controller(Model gioco, View view) {
            this.gioco = gioco;
            view.registerObserver(this);
      }

      public <C> void update(C change) {
            System.out.println("I am the controller I have been notified by the view
                with an action C");
            Action azione;
            if (change.equals(Switch.ON.toString())) {
                  azione = new TurnOn(this.gioco);
            } else {
                  if (change.equals(Switch.OFF.toString())) {
                        azione = new TurnOff(gioco);
                  } else {

                        azione = new Scommetti(gioco);
                  }
            }
            azione.esegui();
      }

      public void update() {
            System.out.println("I am the controller I have been notified by the
                view");

      }
}
```

2.1.5 View

```
public class View extends Observable implements Observer {

   //ATTENZIONE!
```

```
   // IN GENERALE NON E' UNA BUONA PRATICA FORNIRE ALLA VIEW UN
   //REFERENCE AL MODELLO
   // INFATTI, CON QUESTA SCELTA SAREBBE POSSIBILE MODIFICARE
   // IL MODELLO SENZA PASSARE PER IL CONTROLLER
      private final Model gioco;

      public View(Model gioco) {
            gioco.registerObserver(this);
            System.out.println(gioco);
            this.gioco=gioco;
      }

      public void input(String input) {
            System.out.println("I am the view I am notifying my observers");
            this.notifyObservers(input);
      }

      public void update() {
            System.out.println("I am the view I have been notified by the model ");
            System.out.println(gioco);
      }

      public <C> void update(C change) {
            System.out.println("I am the view I have been notified by the model with
                an update C");

            if (change.equals(State.WIN)) {
                  System.out.println("Avete vinto");
            } else {
                  System.out.println("Siete morti");
            }
            System.exit(0);

      }

}
```

2.1.6 Main

```
public class Main {

      public static void main(String[] args) throws IOException {
            Scanner in = new Scanner(System.in);
            Model gioco=new Model();
            View view =new View(gioco);
            Controller controller=new Controller(gioco, view);

            while(true){
                  System.out.println("Dimmi il comando ON/OFF ");
                  String comando = in.nextLine();
                  view.input(comando);
            }
```

```
        }
}
```

2.2 Esercizi per casa

Provare a implementare la versione “modificata” dell’MVC presentata in 1.2.