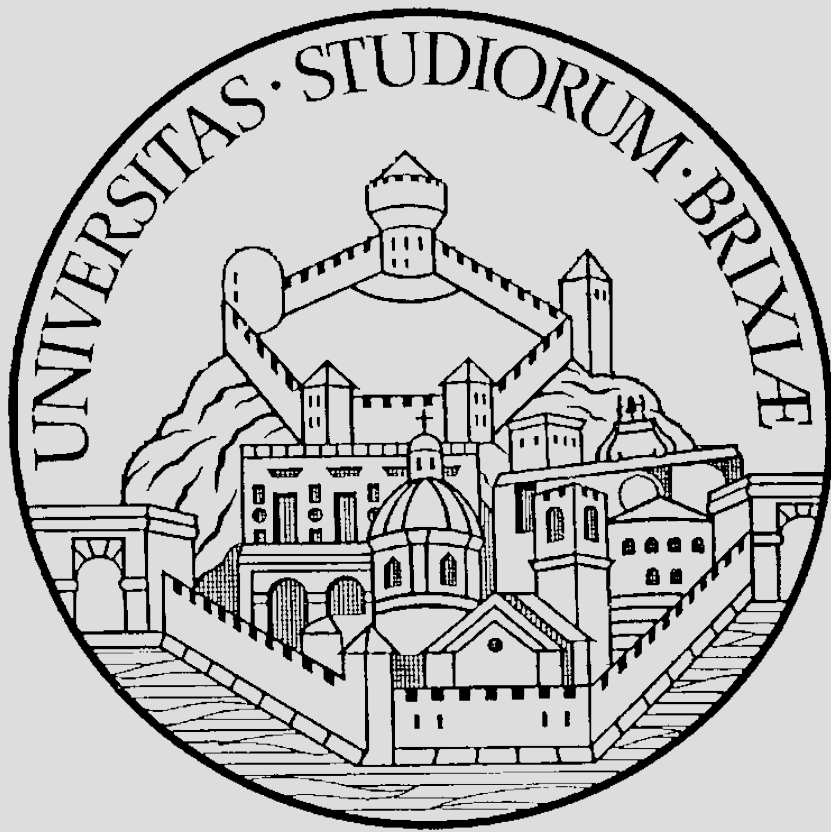


Università degli Studi di Brescia
Corso di Laurea Specialistica in Ingegneria Informatica
Corso di Ingegneria del Software B



Il Pattern MVC

Model View Controller

Marco Iora

Matr. 65574

Cenni storici

- Descritto per la prima volta nel 1979 da Trygve Reenskaug presso il gruppo di sviluppo Smalltalk al PARC (Xerox Palo Alto Research Center).
- Sviluppato da Jim Althoff (e altri) nel 1984 per divenire parte della libreria di classi di Smalltalk-80 R2.0, come fondamento portante per lo sviluppo di GUI.
- Dal 1984 ad oggi viene adottato in molti linguaggi di programmazione (Objective-C, Java, C# .NET ...) che pongono MVC alla base delle proprie librerie di interazione utente.
- Dagli anni 90 MVC viene adottato anche per lo sviluppo di applicazioni web (ASP.NET MVC, RubyOnRails, CakePHP, Symfony, SpringFramework, Django, Joomla ...).

Classificazione GoF

- MVC non è descritto dalla GoF.
- Tipicamente le implementazioni di MVC fanno uso di uno o più pattern descritti dalla GoF.
- Creazionale, strutturale o comportamentale?
 - Architetturale:

Il pattern opera ad un livello più ampio rispetto ai “design pattern” e descrive uno schema generale per costruire un software suddiviso in componenti con specifici e distinti compiti.
 - E' un pattern per strutturare un sistema software, non un singolo componente.

Obiettivo - Scopo

- Facilitare la programmazione in sistemi che prevedono rappresentazioni **multiple** e **sincronizzate** di uno stesso insieme di informazioni.
- Rendere **agile** il processo di modifica di sistemi le cui interfacce sono soggette a frequenti ritocchi (per adattamenti, trasposizioni o aggiunte).
- **Separare** il lavoro degli sviluppatori in base a competenze specializzate.

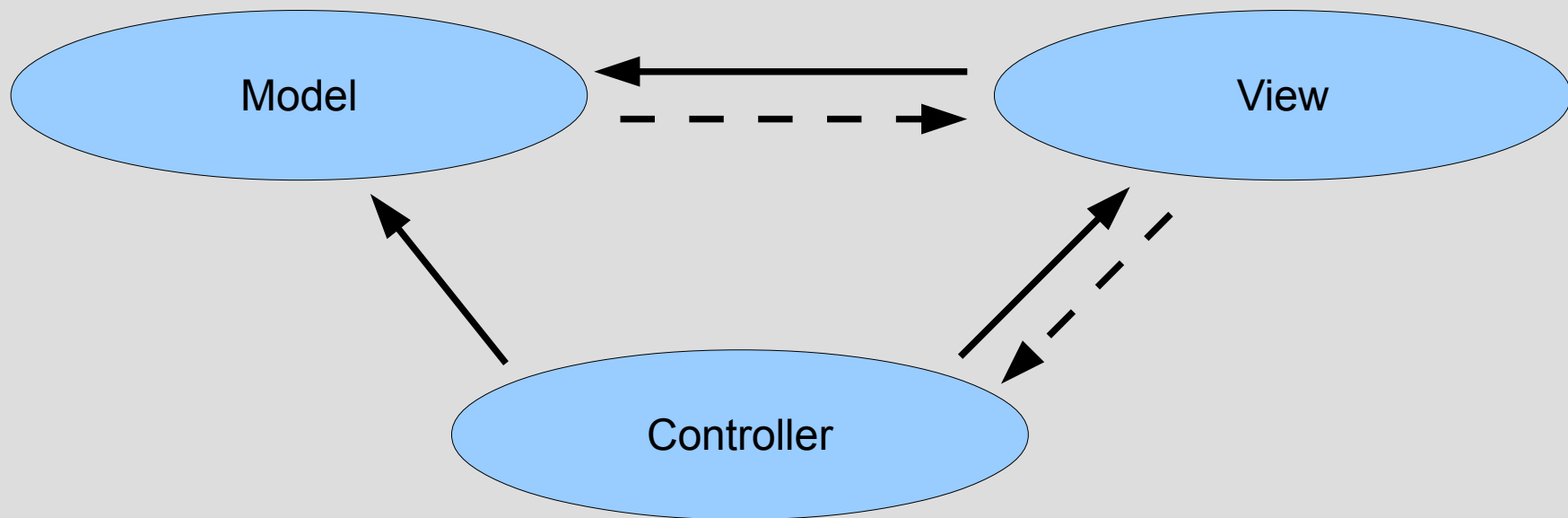
(Daniele Bertolotti)

Ruoli

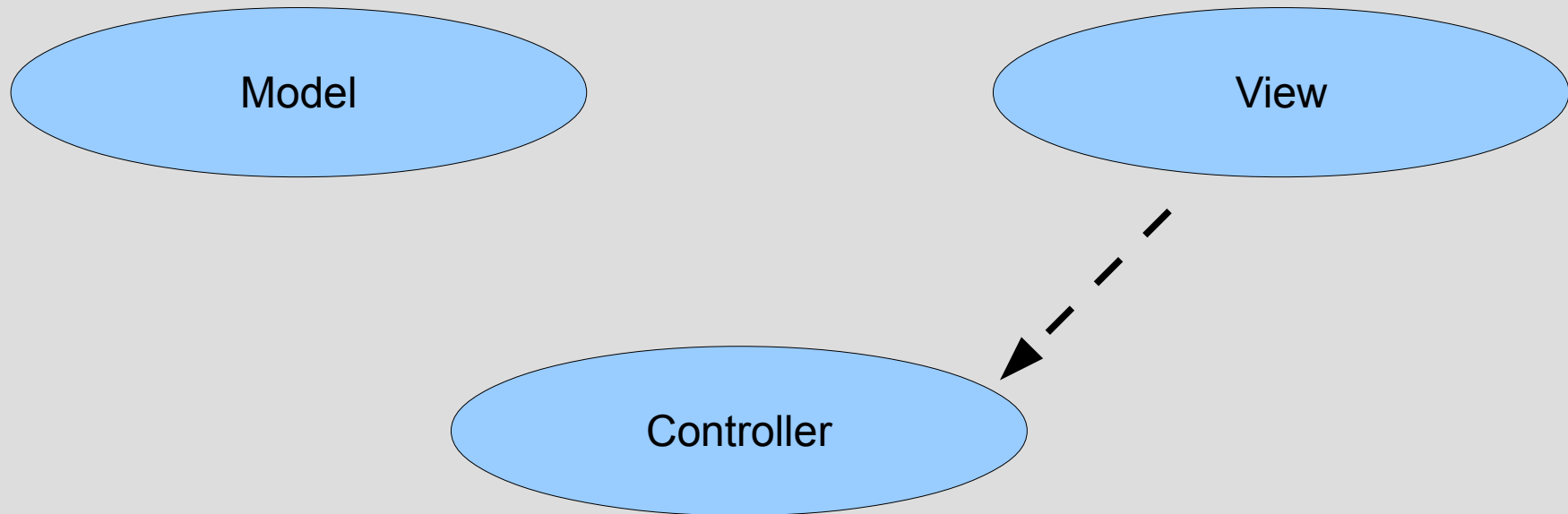
Il pattern è basato sulla separazione delle responsabilità tra i componenti che svolgono i tre ruoli:

- **Model:** gestisce il comportamento ed i dati del dominio dell'applicazione, risponde a richieste di informazioni sul suo stato (tipicamente dal layer View), e risponde a “comandi” di cambiamento di stato (tipicamente dal layer Controller).
- **View:** fornisce una rappresentazione del modello in una forma fruibile dall'utente. Possono esistere più componenti View per un singolo modello.
- **Controller:** riceve l'input dell'utente e istruisce il Model affinché svolga particolari azioni in base a tale input.

Struttura ed interazioni

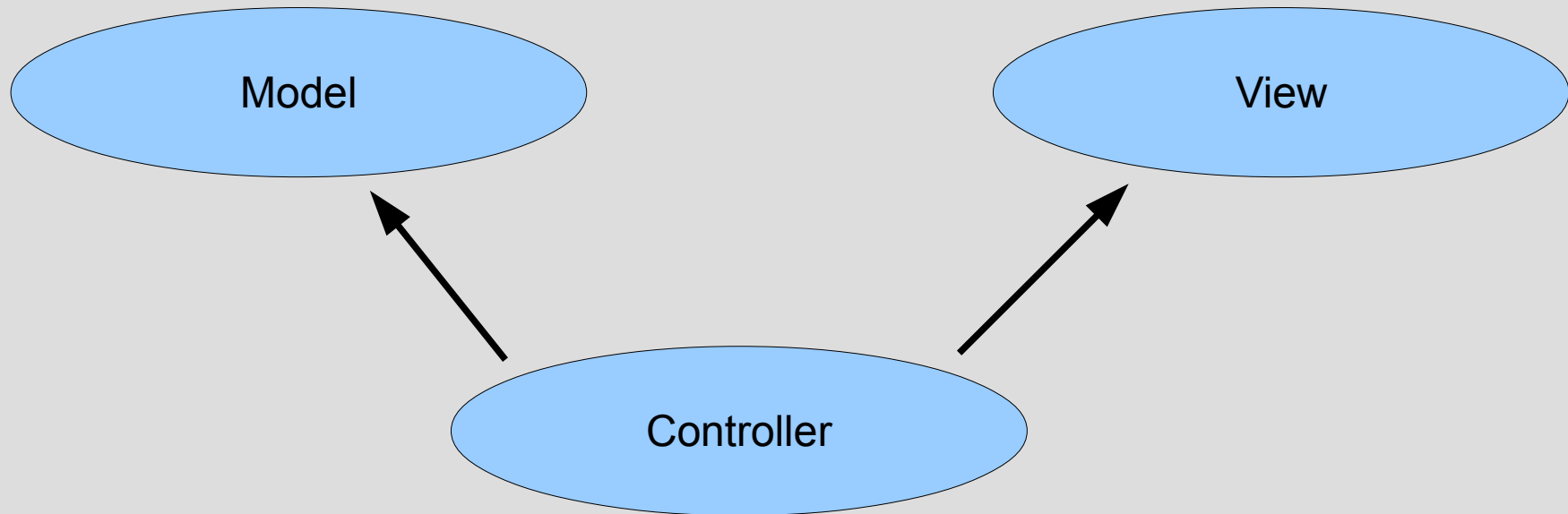


Struttura ed interazioni



Le azioni dell'utente vengono notificate (ad es. con un meccanismo publish/subscribe) dalla **view** al **controller**.

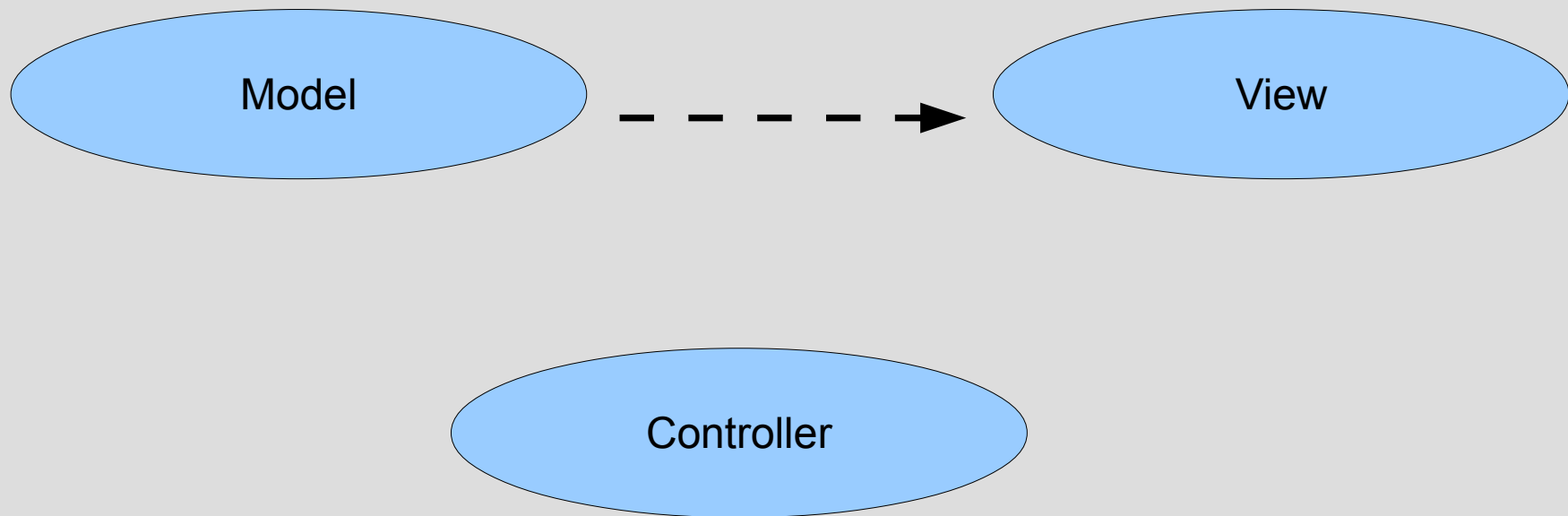
Struttura ed interazioni



Il **controller** modifica il **model** in base all'input dell'utente. Eventualmente modifica in modo diretto anche la/le **view**.

Il **controller** svolge il ruolo di traduttore e dispatcher tra le azioni dell'utente e gli oggetti (**model**) che rappresentano il dominio dell'applicazione.

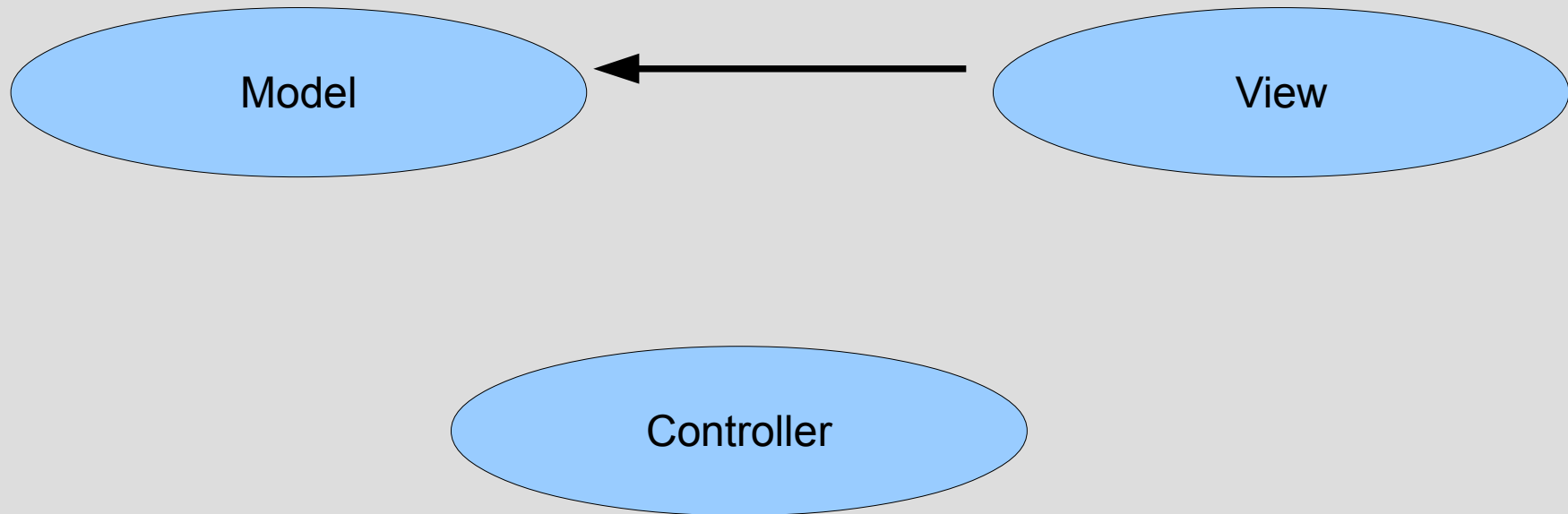
Struttura ed interazioni



Ad ogni modifica del proprio stato il **model** invia notifica a tutte le **view** interessate (publish-subscribe).

A seconda delle implementazioni il **model** può notificare solo alcune modifiche, in base alle proprietà che le **view** hanno dichiarato essere di proprio interesse.

Struttura ed interazioni

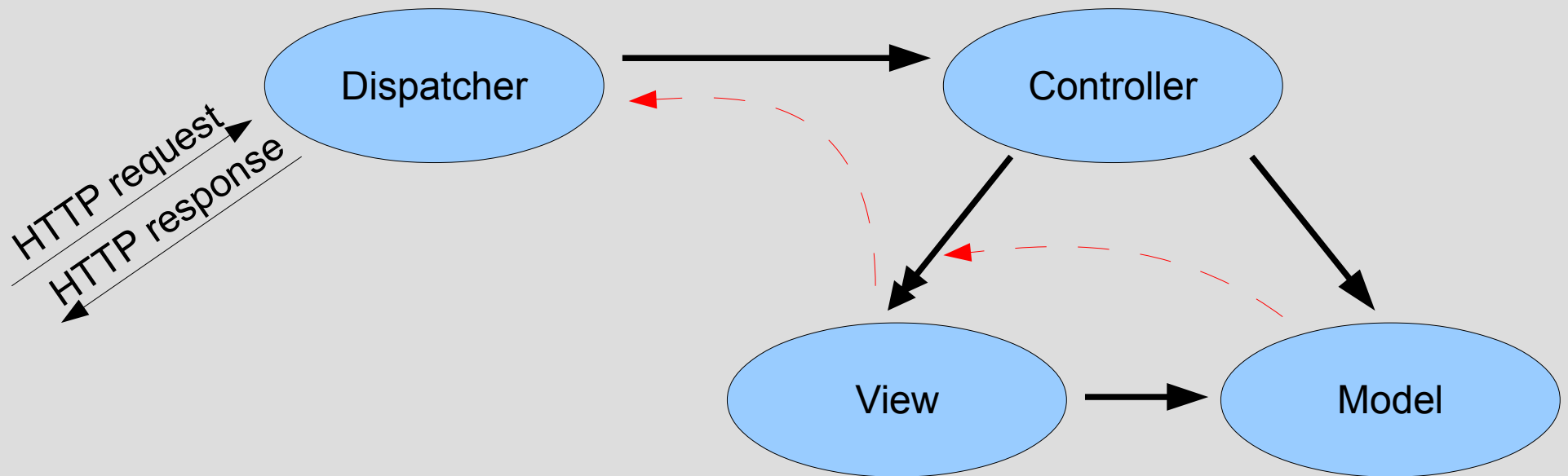


La **view** aggiorna la propria visualizzazione ottenendo i dati “freschi” dal **model** cui è associata.

E nel web?

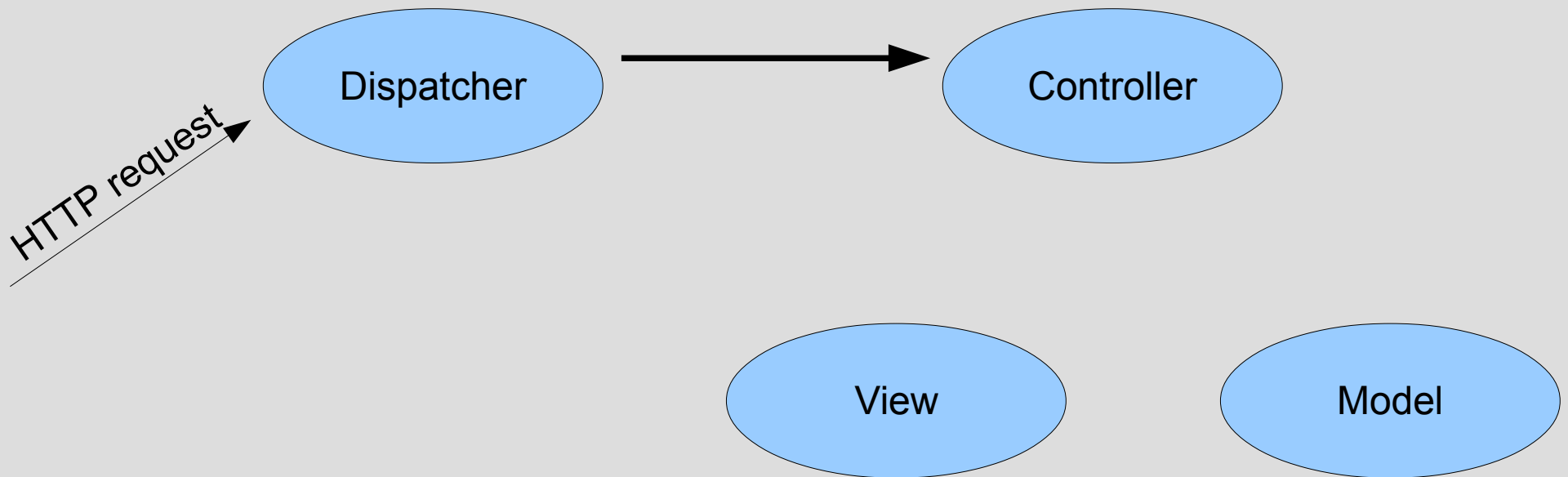
- La difficoltà sta nell'instaurare un meccanismo publish/subscribe affidabile e snello tra **view** e **model** così come tra **controller** e **view**.
- Recenti implementazioni, grazie al continuo perfezionamento delle tecnologie permettono di realizzare un sistema molto simile a quello finora descritto. (AJAX, Comet, Long polling, WebSockets)
- Esistono molti framework web basati sull'architettura MVC, cosa hanno in comune con quanto visto finora? ...

MVC nel web



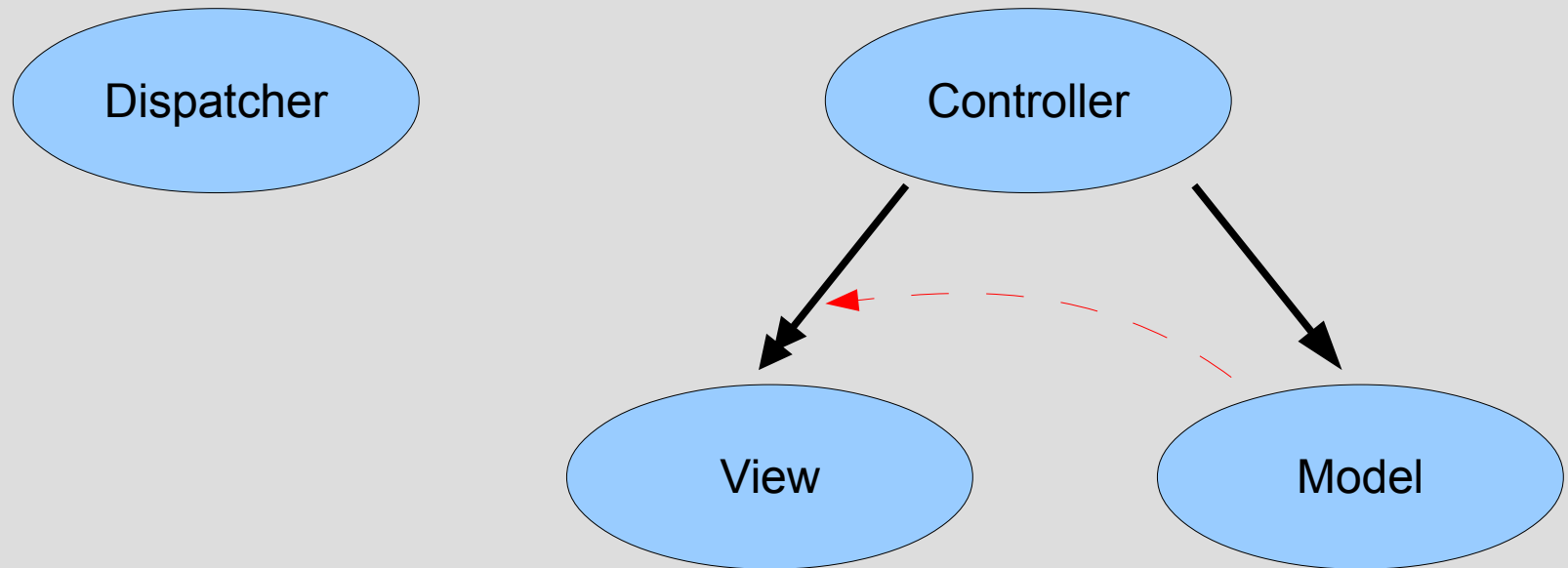
- Non viene utilizzato alcun meccanismo publish/subscribe, in particolare tra view e model.
- Questa “forma” di MVC permette comunque il disaccoppiamento tra le componenti (model, view, controller), che, in particolare nel web, sono solitamente sviluppate da personale specializzato.
- L'architettura di massima pressoché uguale in tutti i sistemi per web basati su MVC, e qui semplificata, diminuisce i tempi di apprendimento necessari a passare da un framework ad un altro.

MVC nel web



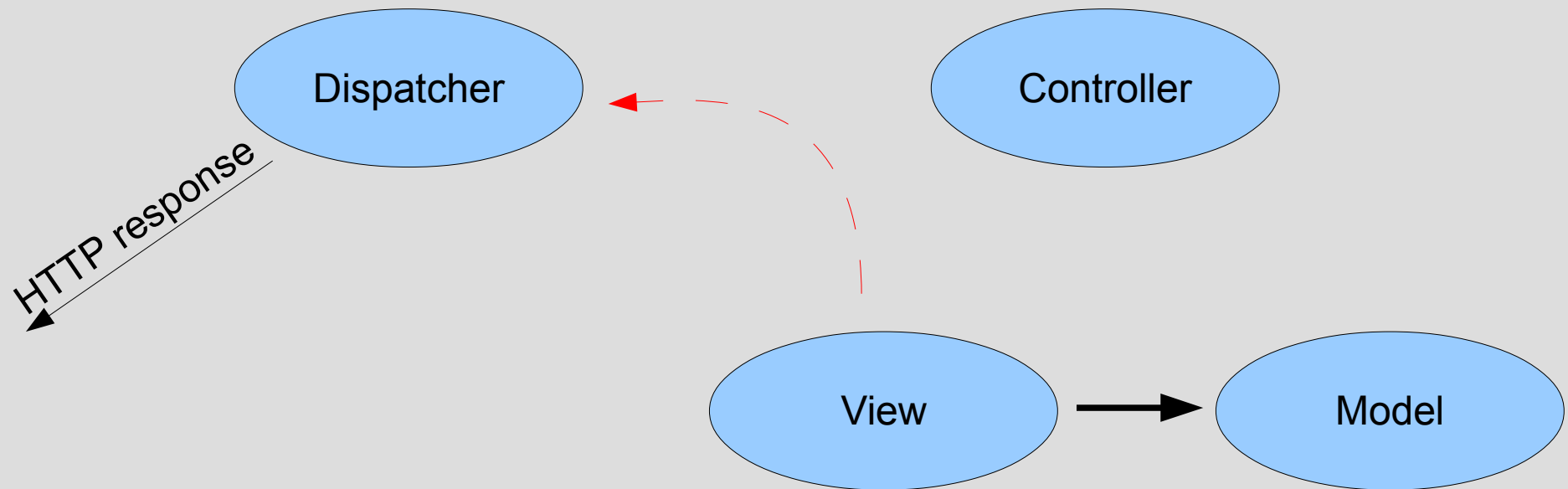
- Il **Dispatcher**, o Front Controller, riceve tutte le richieste HTTP provenienti dai client e, in base a regole di “routing”, sceglie dal pool di **controller** quello corretto per la gestione dell'input utente.
- Al **controller** viene passata la richiesta HTTP, insieme a numerose altre informazioni necessarie per la gestione della richiesta.

MVC nel web



- Il **controller** ottiene (o inizializza) il **model** sul quale svolge le operazioni necessarie per la corretta gestione della richiesta utente.
- Quindi costruisce una nuova **view**, tipicamente basata su template.
- Durante la creazione viene passato alla **view** il relativo **model**.
- Compito del **controller** è quindi interpretare la richiesta utente e agire di conseguenza sul **model**. Quindi fare una prima scelta (**view**) sul modo in cui verranno presentati all'utente i risultati dell'interazione.

MVC nel web

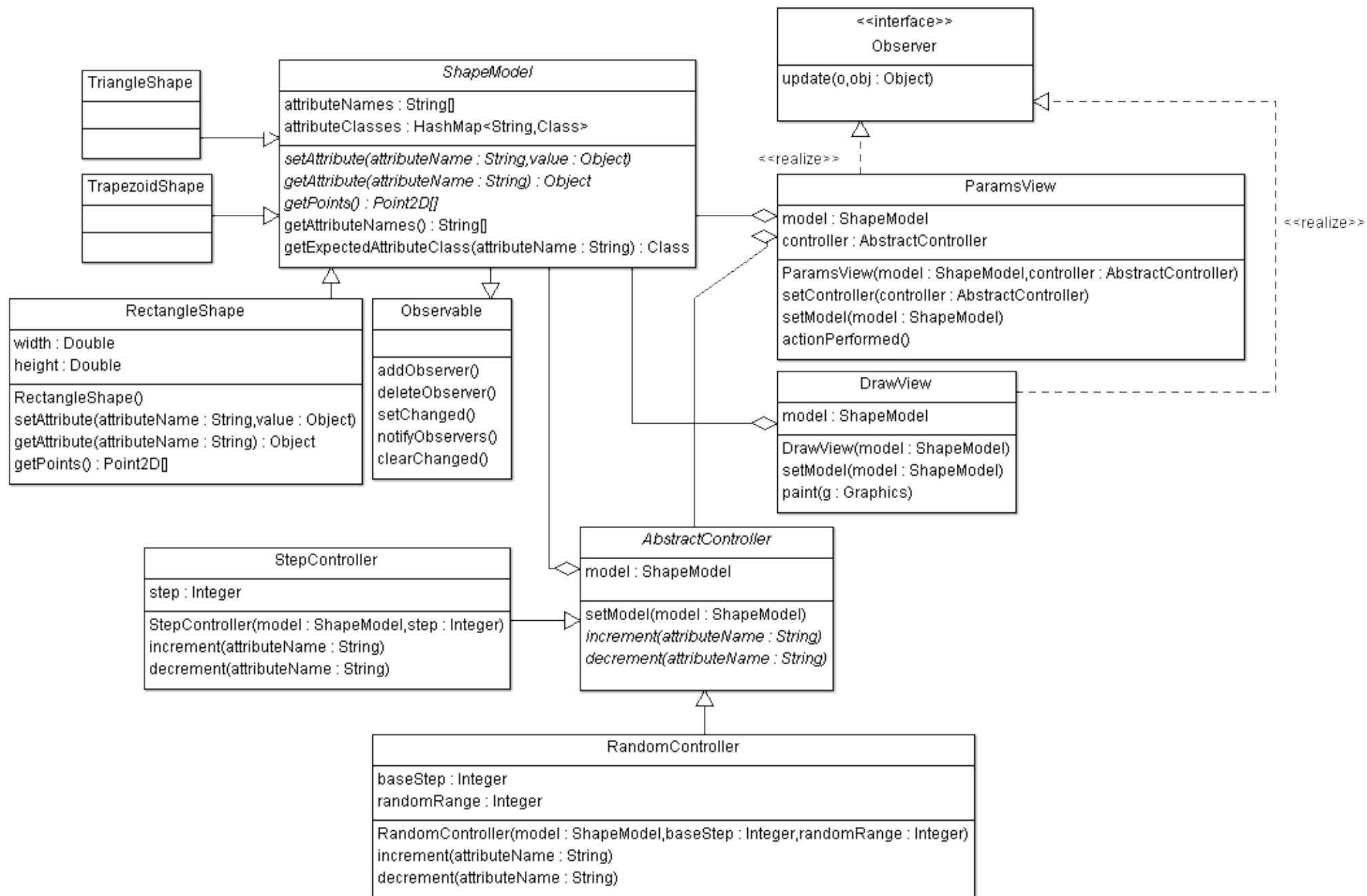


- La **view** ottiene dal **model** le informazioni che utilizza per popolare il proprio template.
- Il **controller** restituisce la **view** (a questo punto pronta per essere interpretata da un browser) al **dispatcher**.
- Il **dispatcher** a sua volta costruisce una risposta HTTP e vi ingloba la **view** appena preparata.
- Infine il browser interpreta la risposta HTTP e la mostra in forma grafica all'utente.

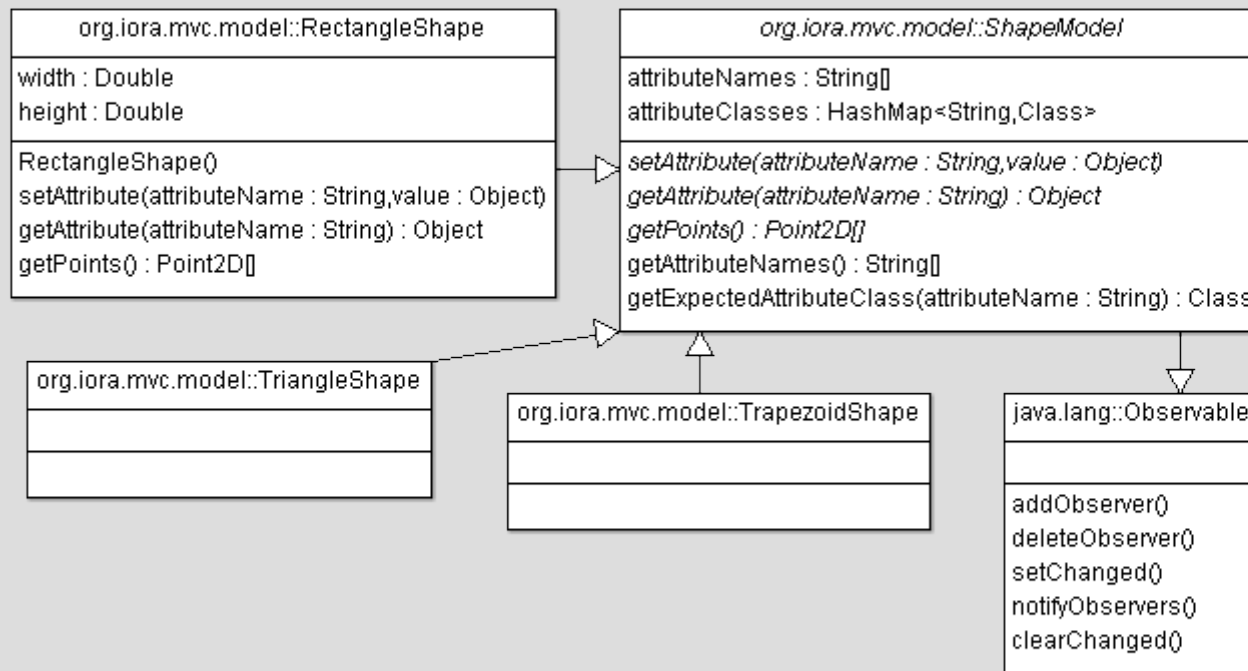
Un esempio concreto

Una applicazione Java che permette di visualizzare forme grafiche (rettangolo, triangolo, trapezio) in modo sincronizzato alla modifica delle relative caratteristiche di base (lati, angoli).

- **View:** una classe per l'interazione utente ed una per il disegno della forma.
- **Controller:** due controller d'esempio (interscambiabili in run-time) per la modifica dei parametri del model.
- **Model:** tre classi d'esempio (interscambiabili in run-time) che “esportano” i propri parametri a view e controller così da poter essere configurate e visualizzate da questi.



Le classi del model



- La classe astratta ShapeModel estende Observable.
- Contiene l'elenco degli attributi supportati dal **model** e il relativo tipo.
- Le classi concrete che estendono ShapeModel popolano l'array *attributeNames* e *attributeClasses* oltre a fornire le implementazioni di `setAttribute`, `getAttribute` e `getPoints`
- Quando viene richiamato `setAttribute` viene utilizzato il meccanismo di publish per notificare agli oggetti in ascolto che è avvenuta una modifica al model.

Costruire un model

- RectangleShape è un model per forme rettangolari, parametrizzato dagli attributi base e altezza (width e height).
- Il costruttore inizializza l'array degli attributi supportati dallo specifico **model** e, per ognuno, dichiara il tipo di dato accettato.
- Infine inizializza con dei valori di default i propri attributi.

```
public static final String ATTRIBUTE_HEIGHT
= "Height";
public static final String ATTRIBUTE_WIDTH =
"Width";

private Double width;
private Double height;

public RectangleShape() {

this.attributeNames = new String[]
{ATTRIBUTE_WIDTH, ATTRIBUTE_HEIGHT};
this.attributeClasses = new HashMap<String,
Class>();

this.attributeClasses.put(ATTRIBUTE_WIDTH,
Double.class);

this.attributeClasses.put(ATTRIBUTE_HEIGHT,
Double.class);

width = new Double(80);
height = new Double(50);

}
```

Agire sul model

- Il metodo *setAttribute* verifica se il nome attributo è supportato e se il tipo del valore da assegnare è congruo con quello atteso.
- Se il valore è valido e diverso da quello dello stato corrente del **model**, il nuovo valore viene assegnato.
- Se è avvenuta una modifica allo stato del **model** viene lanciata una notifica attraverso il meccanismo fornito dalla superclasse Observable.

```
public void setAttribute(String
attributeName, Object value) throws
IllegalArgumentException {

    if (!hasAttribute(attributeName) || !
isValidObject(attributeName, value)) {
        throw new IllegalArgumentException();
    } else {
        if
(attributeName.equals(ATTRIBUTE_WIDTH)) {
            if (this.width.compareTo((Double)
value) != 0) {
                this.width = (Double) value;
                setChanged();
                notifyObservers();
            }
        } else if
(attributeName.equals(ATTRIBUTE_HEIGHT)) {
            if (this.height.compareTo((Double)
value) != 0) {
                this.height = (Double) value;
                setChanged();
                notifyObservers();
            }
        }
    }
}
```

Ottenere informazioni dal model

- Il metodo *getAttribute* verifica se il nome attributo è supportato.
- Se l'attributo è supportato ritorna il valore di tale attributo nello stato corrente del **model**.

```
public Object getAttribute(String
attributeName) throws
IllegalArgumentException {

    if (!hasAttribute(attributeName)) {
        throw new IllegalArgumentException();
    } else {
        if
(attributeName.equals(ATTRIBUTE_WIDTH)) {
            return width;
        } else if
(attributeName.equals(ATTRIBUTE_HEIGHT)) {
            return height;
        }
    }

    throw new RuntimeException();
}
```

Ottenere informazioni dal model (2)

- Il nostro **model** è costituito da una parte direttamente osservabile e controllabile (tramite metodi *setAttribute* e *getAttribute*) e di una che è solo osservabile.
- *getPoints* fornisce l'elenco dei vertici della figura da disegnare. I vertici sono calcolati in base ad una formula che non è altro che la “business logic” della nostra applicazione.
- Ogni classe concreta del **model** è specializzata su una forma geometrica. Pertanto le classi del **model**, oltre a differire per l'elenco degli attributi (ed eventualmente i relativi tipi), si distinguono per la formula implementata nel metodo *getPoints*.

```
public Point2D[] getPoints() {  
  
    Point2D.Double[] result = new  
    Point2D.Double[4];  
  
    result[0] = new Point2D.Double(0, 0);  
    result[1] = new Point2D.Double(width, 0);  
    result[2] = new Point2D.Double(width,  
    height);  
    result[3] = new Point2D.Double(0, height);  
  
    return result;  
}
```

Un trapezio?

- La classe *TrapezoidShape* è del tutto analoga a *RectangleShape*.
- Definisce quattro parametri: bottomSide, topSide, height e angle12. Ovvero il lato di base, il lato superiore, l'altezza e l'angolo (in gradi sessagesimali) tra il lato di base ed il lato di sinistra.
- Il metodo *getPoints* implementa la formula per determinare i vertici del trapezio tramite funzioni trigonometriche.

```
private Double bottomSide;
private Double topSide;
private Double height;
private Double angle12;

public Point2D[] getPoints() {

    double angle12Rad =
        angle12.doubleValue()*Math.PI/180.0;

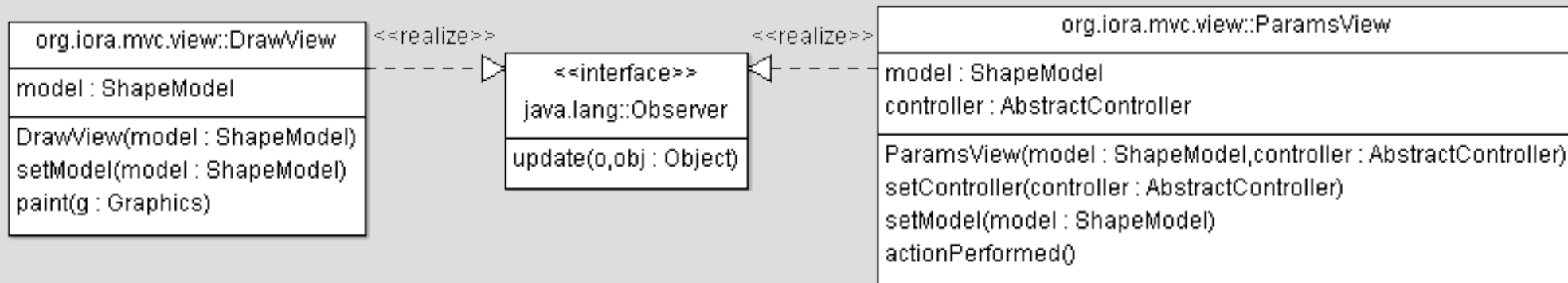
    Point2D.Double[] result = new
        Point2D.Double[4];

    result[0] = new Point2D.Double(0, 0);
    result[1] = new
        Point2D.Double(height/Math.tan(angle12Rad), h
            eight);
    result[2] = new
        Point2D.Double(topSide+height/Math.tan(angle
            12Rad), height);
    result[3] = new Point2D.Double(bottomSide,
        0);

    return result;

}
```

Le view



- Due **view** distinte si appoggiano sullo stesso **model**.
- *ParamsView* permette la visualizzazione e configurazione dei parametri.
- *DrawView* disegna i lati del poligono.
- Entrambe sono *Observer* e sottoscrivono gli eventi di notifica del **model**.
- Entrambe supportano il cambio di **model** in run-time.
- *DrawView* non ha alcun riferimento ad un controller in quanto è una **view** “passiva”, non permette infatti alcuna interazione con l'utente.

DrawView: setModel

- *DrawView* implementa un meccanismo semplice per configurare il **model** cui è associato.
- Se è già stato associato ad un **model** in precedenza, rimuove la propria sottoscrizione agli aggiornamenti.
- Quindi salva come riferimento il nuovo **model** e si abbona agli aggiornamenti di quest'ultimo.
- Ridisegna l'interfaccia grafica affinché corrisponda fin da subito alla forma descritta dal **model**.

```
public final void setModel(ShapeModel model)
{
    if (this.model != null) {
        this.model.deleteObserver(this);
    }
    this.model = model;
    this.model.addObserver(this);
    repaint();
}
```

DrawView: paint e update

- *DrawView* disegna linee che congiungono i vertici indicati dal **model**.
- Ottiene i vertici invocando il metodo *getPoints*, quindi traccia una linea che congiunge le coppie di punti così ottenuti.
- Il metodo *update* non fa altro che richiedere al sottosistema grafico di richiamare il metodo *paint* non appena possibile, così da aggiornare il disegno coerentemente con il **model**.

```
public void paint(Graphics g) {
    super.paint(g);
    Graphics2D g2 = (Graphics2D) g;
    Point2D[] points = getModel().getPoints();

    g2.transform(AffineTransform.getTranslateInstance(WIDTH/2, HEIGHT/2));
    g2.transform(AffineTransform.getScaleInstance(1.0, -1.0));
    g2.setStroke(new BasicStroke(2));

    for (int i = 0; i < points.length; i++) {
        Point2D from = points[i];
        Point2D to = points[(i+1)%points.length];
        g2.drawLine((int) from.getX(), (int) from.getY(), (int) to.getX(), (int) to.getY());
    }

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

ParamsView: setModel

- *ParamsView* visualizza una riga di componenti per ogni parametro esposto dal **model**.
- Si riconfigura per visualizzare i nuovi parametri quando viene invocato il metodo *setModel*.
- Per ogni parametro è presente: un'etichetta per visualizzarne il nome, un campo di testo per visualizzarne il valore, un pulsante per incrementarne il valore ed uno per decrementarlo.
- Effettua inoltre, le medesime operazioni viste in *DrawView*: *setModel* per quanto riguarda l'iscrizione alle notifiche e l'aggiornamento del riferimento al **model**.

```
String[] attributeNames =
model.getAttributeNames();
for (int i=0;i < attributeNames.length; i++)
{
    String attributeName = attributeNames[i];
    Class attributeClass =
model.getExpectedAttributeClass(attributeName);
    if (attributeClass == Double.class) {
        JLabel lbl = new
JLabel(attributeName);
        add(lbl, new GridBagConstraints(...));
        JTextField txt = new
JTextField(model.getAttribute(attributeName)
.toString());
        txt.setName(attributeName);
        txt.setColumns(8);
        txt.setEditable(false);
        add(txt, new GridBagConstraints(...));
        JButton btnUp = new JButton(new
ImageIcon(getClass().getResource("/plus.png"
)));
        btnUp.setActionCommand("UP");
        btnUp.setName(attributeName);
        btnUp.addActionListener(this);
        add(btnUp, new GridBagConstraints(..);
        ...
    }
}
```

ParamsView: update

- *ParamsView* reagisce alle notifiche del model modificando il valore visualizzato nei propri campi di testo.
- Il metodo *update* cicla sui componenti grafici posseduti dalla **view** alla ricerca dei campi di testo. Per ognuno ottiene il nome del parametro cui è associato, quindi chiede al **model** il valore corrente di tale parametro ed infine lo imposta come nuovo valore visualizzato.

```
public void update(Observable o, Object arg)
{
    Component[] components = getComponents();

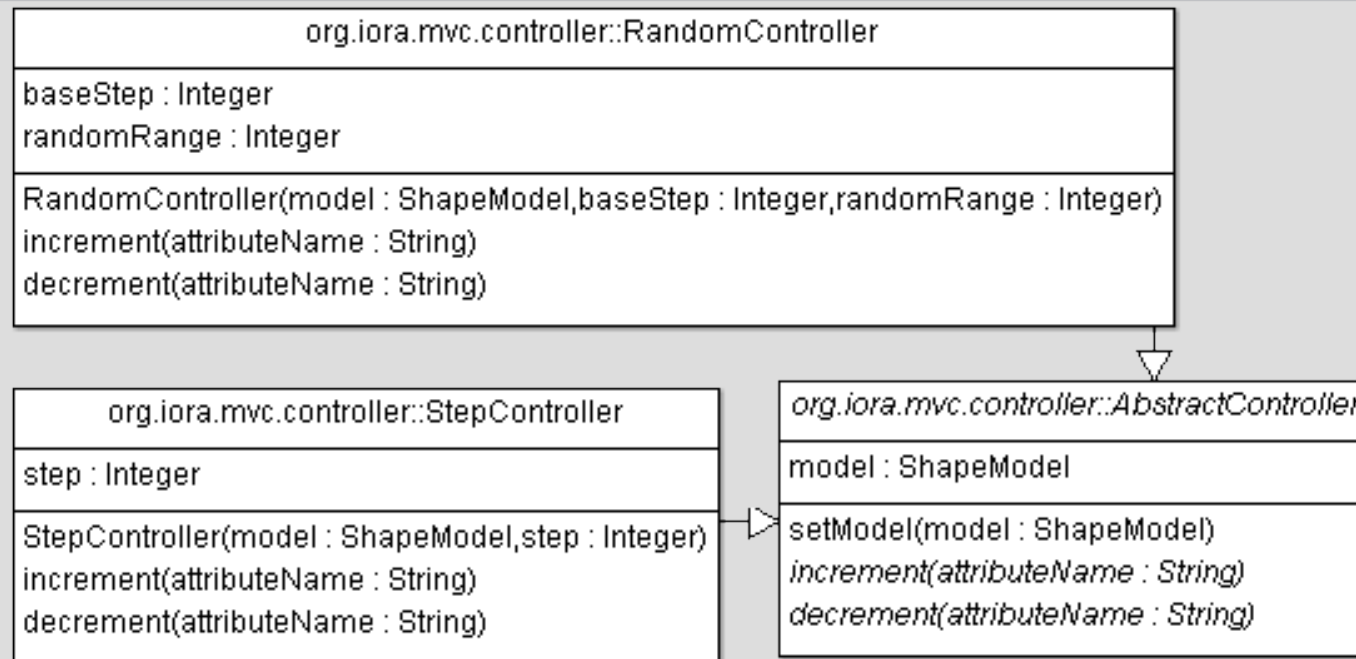
    for (int i = 0; i < components.length; i++)
    {
        if (components[i] instanceof JTextField)
        {
            ((JTextField)
components[i]).setText(model.getAttribute(components[i].getName()).toString());
        }
    }
}
```

ParamsView: azioni

- *ParamsView* richiama i metodi *increment* o *decrement* del **controller** quando viene premuto un pulsante dall'utente.
- Ai metodi *increment* e *decrement* viene passato il nome dell'attributo da modificare.
- E' compito del **controller** agire di conseguenza sul **model**.

```
public void actionPerformed(ActionEvent e) {  
  
    if (e.getActionCommand().equals("UP")) {  
        controller.increment(((Component)  
e.getSource()).getName());  
        return;  
    }  
  
    if (e.getActionCommand().equals("DOWN")) {  
        controller.decrement(((Component)  
e.getSource()).getName());  
        return;  
    }  
}
```

I controller



- Due **controller** distinti permettono di svolgere azioni in modo differente sul **model**.
- *StepController* incrementa e decrementa i parametri di un valore fisso, configurato in inizializzazione.
- *RandomController* invece aggiunge o toglie valori di volta in volta diversi.
- Entrambi estendono la classe astratta *AbstractController* che dichiara l'interfaccia e implementa il metodo *setModel* per la modifica in run-time del **model** associato.

AbstractController

- *AbstractController* contiene solo un riferimento al **model** corrente.
- Non ha riferimenti alla/e **view** in quanto non ha necessità di agire direttamente su di essa/e.
- Ogni implementazione concreta di *AbstractController* può la propria specifica interpretazione (e quindi implementazione) delle azioni *increment* e *decrement*.

```
public abstract class AbstractController {
    private ShapeModel model;

    public ShapeModel getModel() {
        return model;
    }

    public void setModel(ShapeModel model) {
        this.model = model;
    }

    public AbstractController(ShapeModel
model) {
        this.model = model;
    }

    public abstract void increment(String
attributeName);
    public abstract void decrement(String
attributeName);
}
```

StepController

- *StepController* implementa i metodi *increment* e *decrement*.
- Il costruttore configura il parametro *step* del **controller** che definisce l'intervallo da sommare o sottrarre al valore corrente del **model** quando vengono invocati il metodo *increment* o *decrement*.

```
public class StepController extends
AbstractController {

    private int step;

    public StepController(ShapeModel model, int
step) {
        super(model);
        this.step = step;
    }

    @Override
    public void increment(String attributeName)
    {
        if
        (Double.class.isAssignableFrom(getModel().ge
tExpectedAttributeClass(attributeName))) {
            Double value = (Double)
getModel().getAttribute(attributeName);

            value = value.doubleValue()+this.step;

            getModel().setAttribute(attributeName,
value);
        }
    }
    ...
}
```


Il collante

- La classe che implementa il metodo Main inizializza un **model** di base (*RectangleShape*), un **controller** di base (*StepController*) e le due **view**. Crea due finestre (*JFrame*); in una pone un'istanza di *ParamView* nell'altra un'istanza di *DrawView*.
- Nella finestra contenente l'istanza di *ParamView* aggiunge anche un semplice menu che permette di modificare in run-time il **model** associato alle **view**, ed analogamente il **controller**.
- Ora una dimostrazione ...

Conclusioni

- MVC forza un approccio allo sviluppo in cui si separano *model*, *view* e *controller*.
- Grazie a questo persone distinte possono collaborare su uno stesso progetto senza darsi forti gomitate.
- MVC esalta il riuso. Il problema per cui è stato inventato questo approccio era la complessità di gestione derivante da classi di interfaccia grafica in stile “faccio tutto io”.
- MVC esalta la delega. Il programma realizzato per la dimostrazione poteva essere strutturato in modo più semplice facendo uso dell'ereditarietà (es. View che estende Model).
Non sarebbe stato altrettanto semplice mantenere due View sincronizzate sullo stesso model, ne modificare in run-time il comportamento dei pulsanti increment e decrement, ne permettere la visualizzazione di forme diverse caratterizzate da insiemi di parametri e formule di calcolo dei vertici distinti.
- MVC non è la soluzione a tutti i problemi: è complesso e per svolgere semplici operazioni il numero di classi introdotto può essere effettivamente eccessivo.

MVC e i design pattern

MVC è un pattern architetturale; le applicazioni basate su MVC spesso fanno uso di vari pattern della GoF.

- **Observer**, come abbiamo visto, viene utilizzato per la comunicazione (disaccoppiata) tra **model** e **view**.
- **Composite** può essere utilizzato per costruire interfacce grafiche complesse formate da tanti piccoli “sistemi” MVC.
- **Strategy** può essere utilizzato quando si vuole modificare il comportamento di risposta alle azioni dell'utente in run-time, selezionando un controller differente.
- **Factory Method** può essere utilizzato per specificare un controller standard per una view.
- **Decorator** può essere utilizzato per aggiungere funzionalità/comportamenti ad una view già esistente.

Riferimenti

- Patterns of Enterprise Application Architecture, M. Fowler, Addison Wesley 2005
- Design Patterns, Gamma Helm Johnson Vlissides, Addison Wesley
- Model-View-Controller, <http://en.wikipedia.org/wiki/Model-view-controller>
- GUI Architectures, <http://martinfowler.com/eaDev/uiArchs.html>
- Model-View-Controller, <http://msdn.microsoft.com/en-us/library/ff649643.aspx>