Model View Controller

Ingegneria del Software B

Università degli Studi di Brescia
Claudio Gandelli - 65588

Pattern Architetturali

- Concetti caratterizzanti l'**architettura** di un software
  - Separazione di più alto livello in "parti"
  - Decisioni condivise sulle interazioni e la dislocazione delle principali componenti di un sistema

- I *pattern architetturali* operano ad un livello diverso (più ampio) rispetto ai *design pattern*, esprimono schemi di base per impostare l'organizzazione strutturale di un sistema software

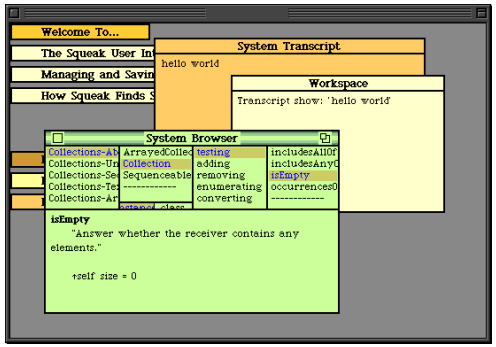
Layering

- Tecnica più comune ed efficace per *separare* un sistema software complesso
- Indipendenza tra i livelli dal basso verso l'alto
- Tre livelli principali
  - Presentation
  - Domain (Business Logic)
  - Data

Origini di MVC

- Nasce come framework sviluppato per Smalltalk negli anni ’70

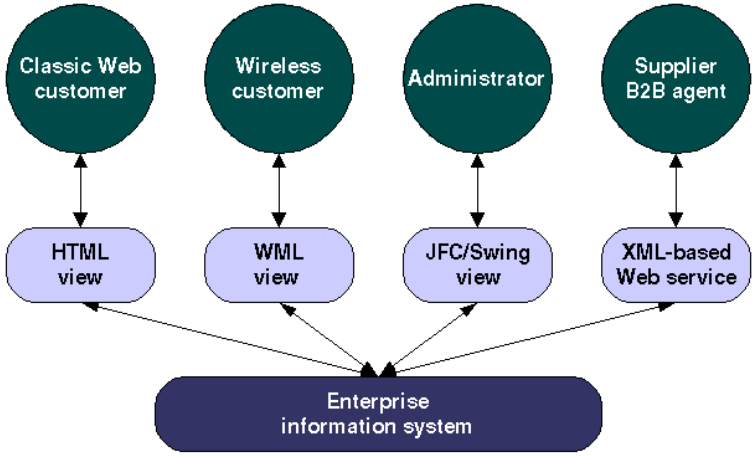
- Necessità di progettazione di interfacce multi-finestra dei sistemi più diffusi all’epoca, che andavano via via diffondendosi



MVC Oggi

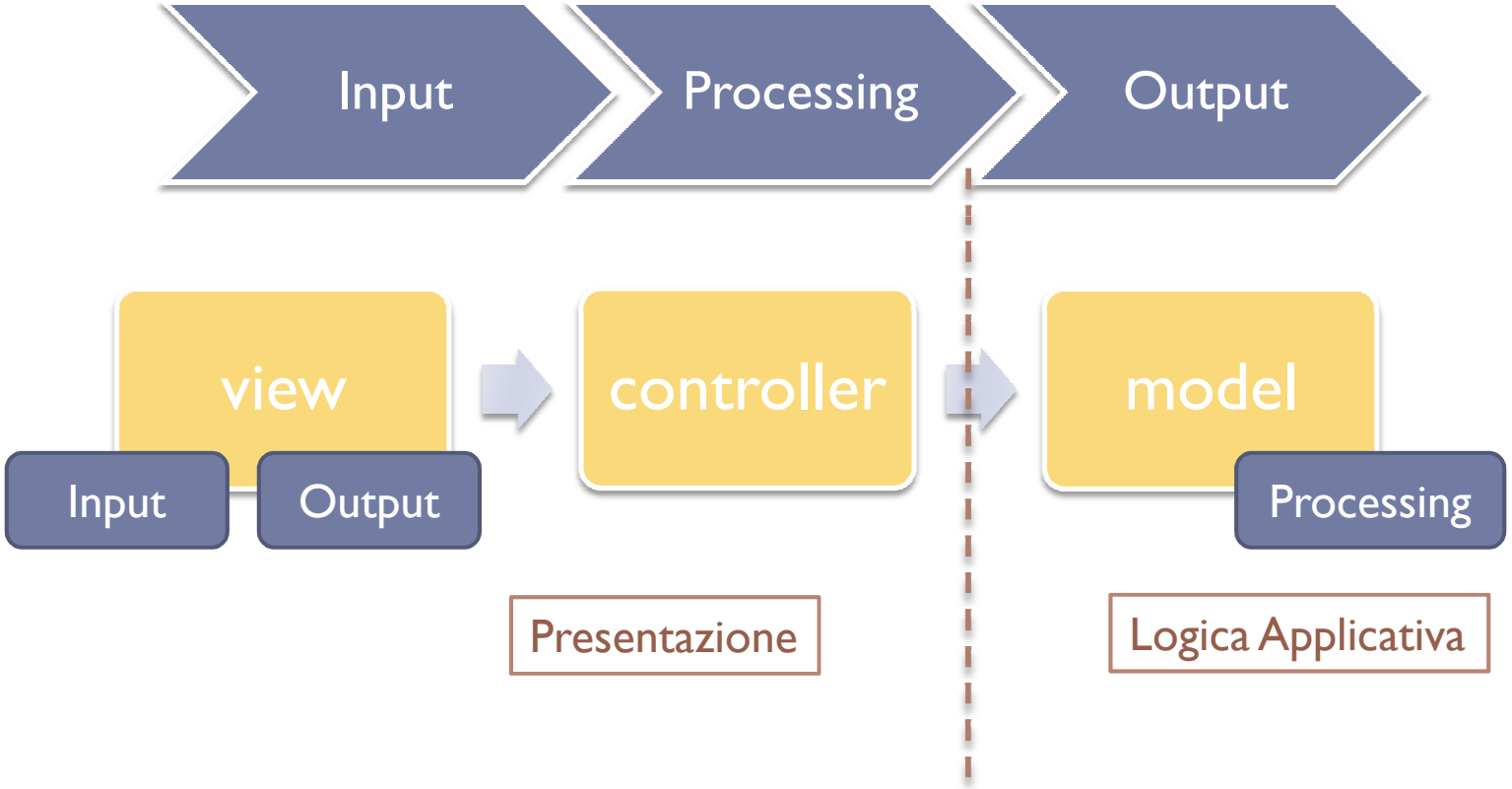
- Oggi più di allora MVC viene utilizzato in soluzioni software per qualsiasi linguaggio e tecnologia

- Punti di forza
  - Opportunità di accesso ai dati tramite più interfacce
  - Più dispositivi e tecnologie adottate
  - Separazione delle responsabilità tra presentazione e logica di controllo



Scopo

- MVC indica un modello per separare un'applicazione in tre elementi distinti: **model**, **view** e **controller**



Vantaggi

- Presentazione e Modello del dominio sono concetti diversi, con obiettivi diversi e professionisti diversi
  - GUI
    - Usabilità, Design, ...
  - Modello
    - Regole, Logica di funzionamento, Dati, ...
- Diverse necessità di presentazione di un Modello a seconda del contesto
  - La separazione permette di lavorare a nuove interfacce sopra lo stesso modello logico
- Il testing delle parti non visuali di un'applicazione è generalmente più semplice

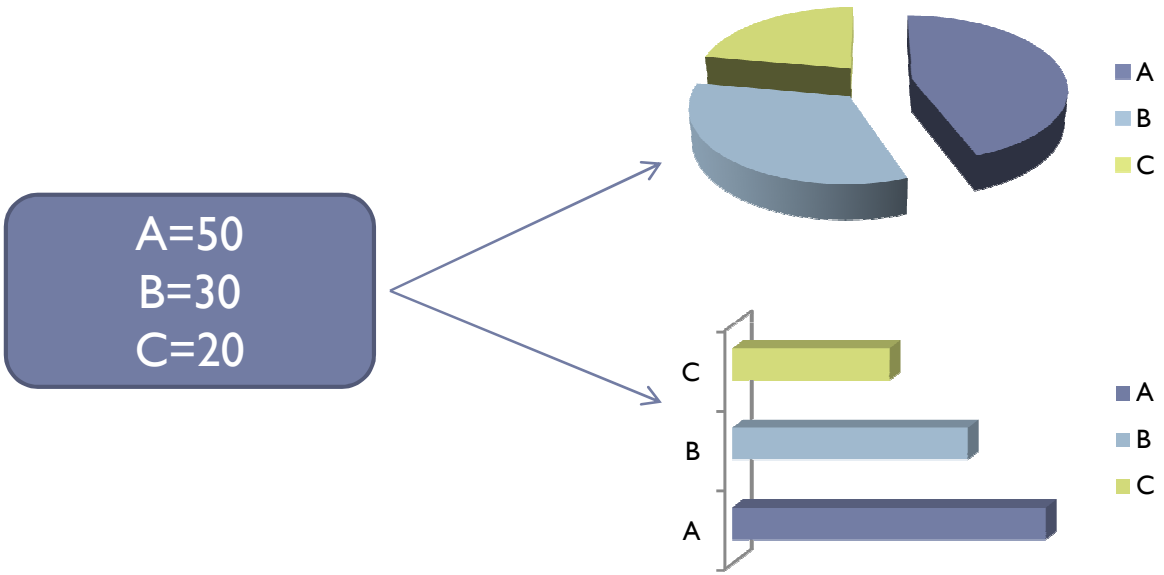
Osservazioni

- Un punto chiave della separazione offerta da MVC è la direzione delle dipendenze

- Chi lavora sul model, ossia sulla logica applicativa, può restare all'oscuro di come i dati verranno presentati all'utente
- Chi lavora alle interfacce può facilmente crearne di nuove senza imporre variazioni al *core* dell'applicazione

Relazione view/model

- Viste multiple sincronizzate sullo stesso oggetto del domino



- Il meccanismo publish-subscribe definito dal pattern Observer permette a tutte le viste di rimanere sincronizzate, senza creare una dipendenza rivolta dal modello alla presentazione

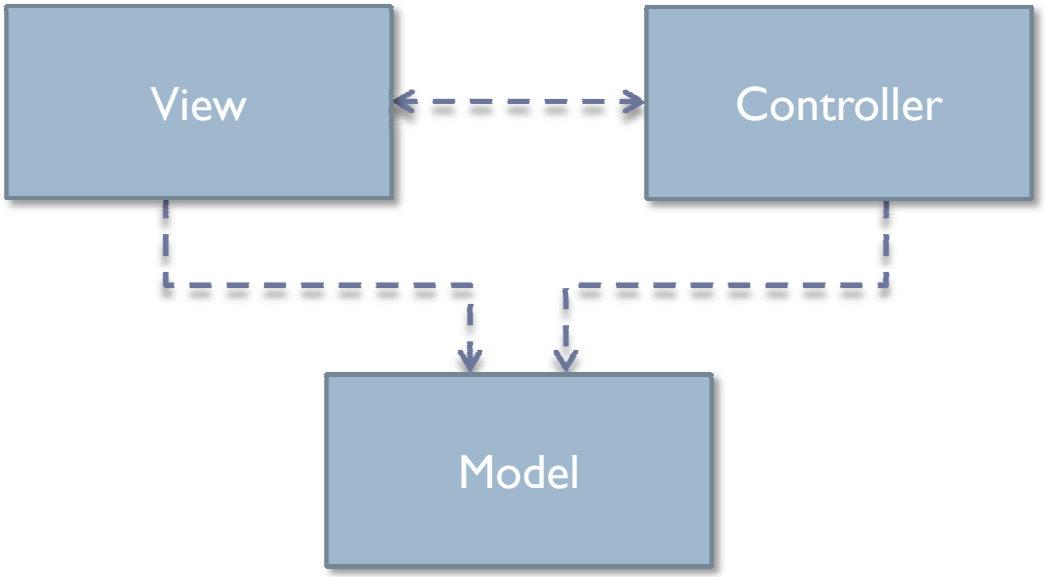
Relazione view/controller

- Le comunicazioni tra la *view* ed il *controller* associato sono dirette, dato che i due elementi sono progettati per funzionare assieme (MVC tradizionale)
- La separazione tra view e controller permette di realizzare interfacce con comportamenti diversi mantenendo la stessa *view*
  - Pattern *Strategy* sul controller
- Talvolta questa separazione è stata trascurata, soprattutto in sistemi con un solo controller per vista (*Document-View, MFC in Visual C++*)

Struttura

- L'input dell'utente, la logica applicativa e il feedback visivo sono separati e gestiti dai tre elementi di MVC

- Realizzazione di MVC per Smalltalk:
  - Riferimenti reciproci tra View e Controller
  - Model Passivo
  - Model Attivo

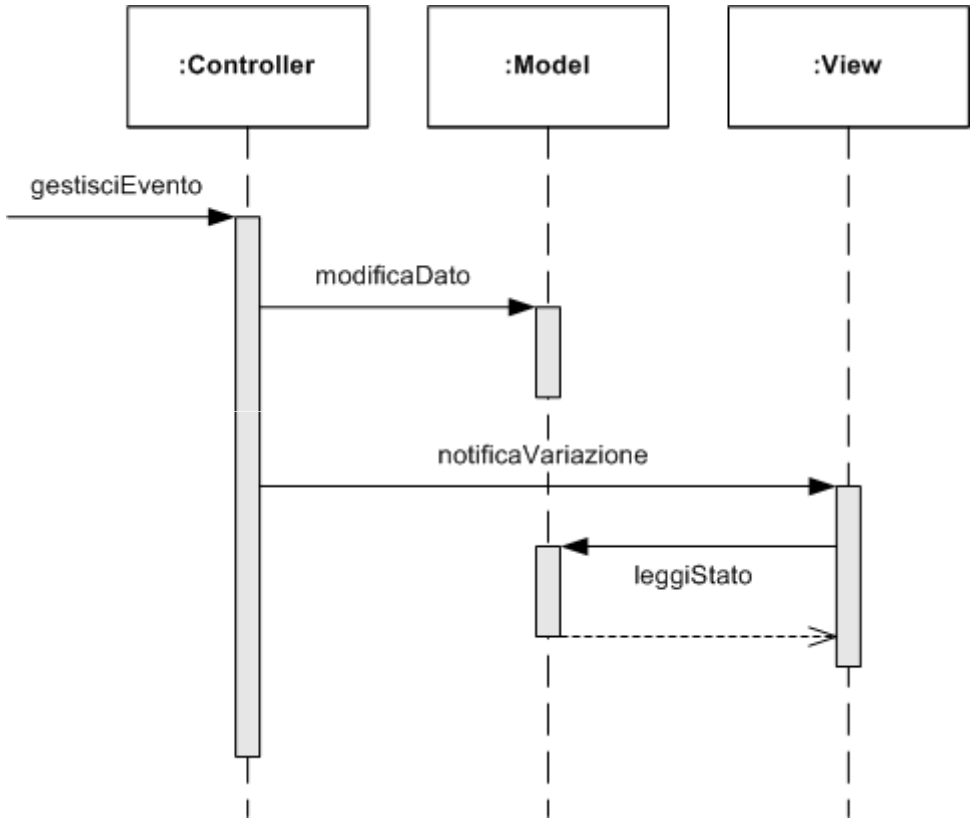


Model Passivo

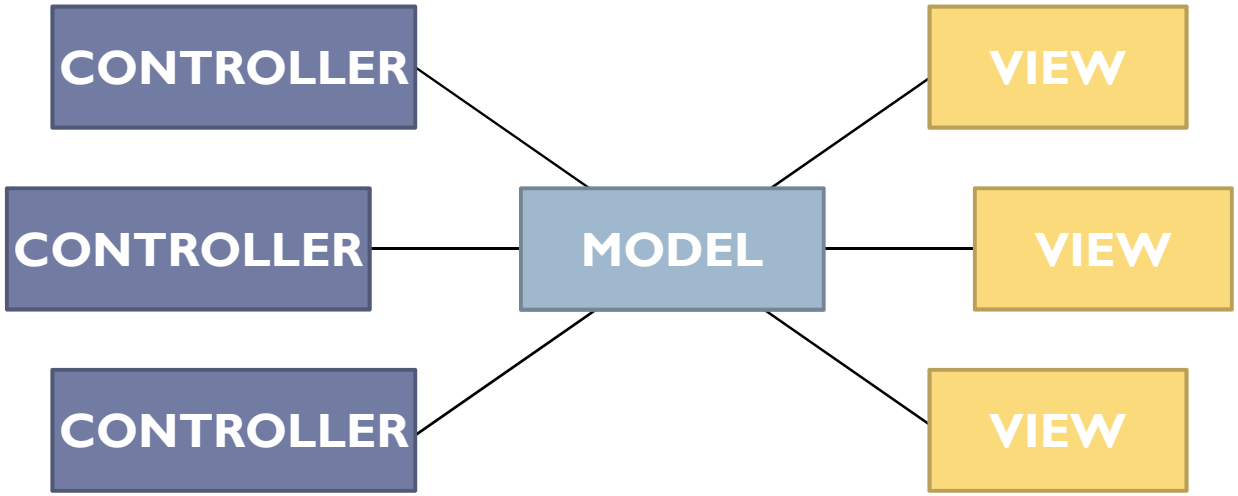
- Un solo *controller*, e solo quello, ha la possibilità di interagire con il *model* e cambiarne lo stato

- Il *model* non ha la responsabilità di informare quando viene modificato
- Il *controller*, che interpreta la richiesta dell'utente, si accolla questa responsabilità
  - Notifica alla view che c'è stata una modifica
  - Specifica alla view quale è stata la modifica
- Il *model* risulta un contenitore passivo del dato, agisce solo su richiesta dell'unico controller

Model Passivo

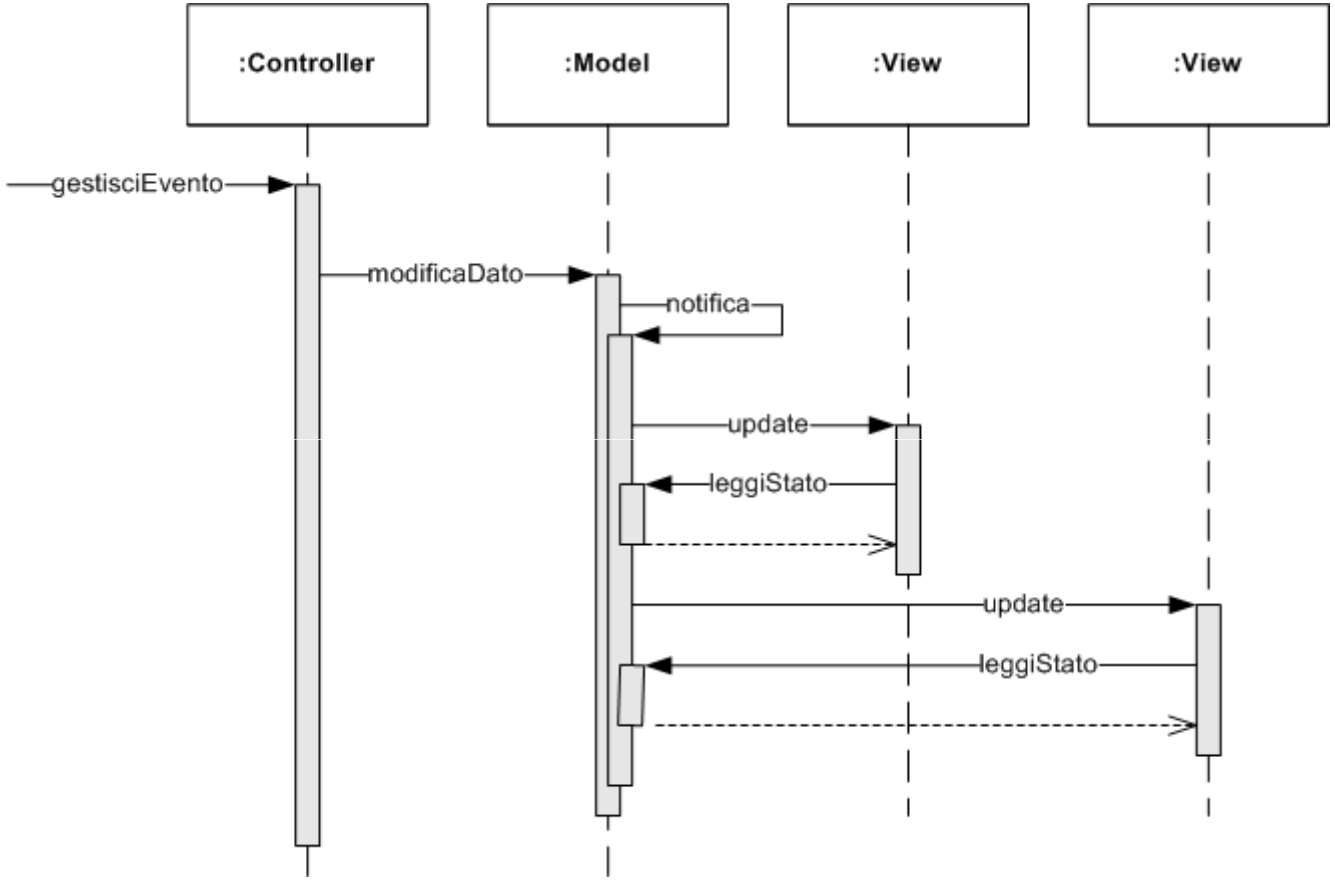


Model Attivo



- Non c'è un unico elemento che può scatenare modifiche dello stato del model
- Alle viste associate in quel momento al model deve giungere una notifica della variazione

Model Attivo



Partecipanti - Model

- Rappresenta le informazioni del dominio all'interno della logica applicativa (*domain logic*). La logica applicativa aggiunge significato ai dati grezzi, solitamente persistenti e trattati con un meccanismo di storage

- Indipendente dagli altri elementi
  - Offre servizi al livello superiore
  - Notifica le variazioni di stato ai propri *subscriber*

- In MVC non viene trattato il problema dell'accesso ai dati in quanto si suppone sia sottostante o incapsulato nel *model*

Partecipanti - View

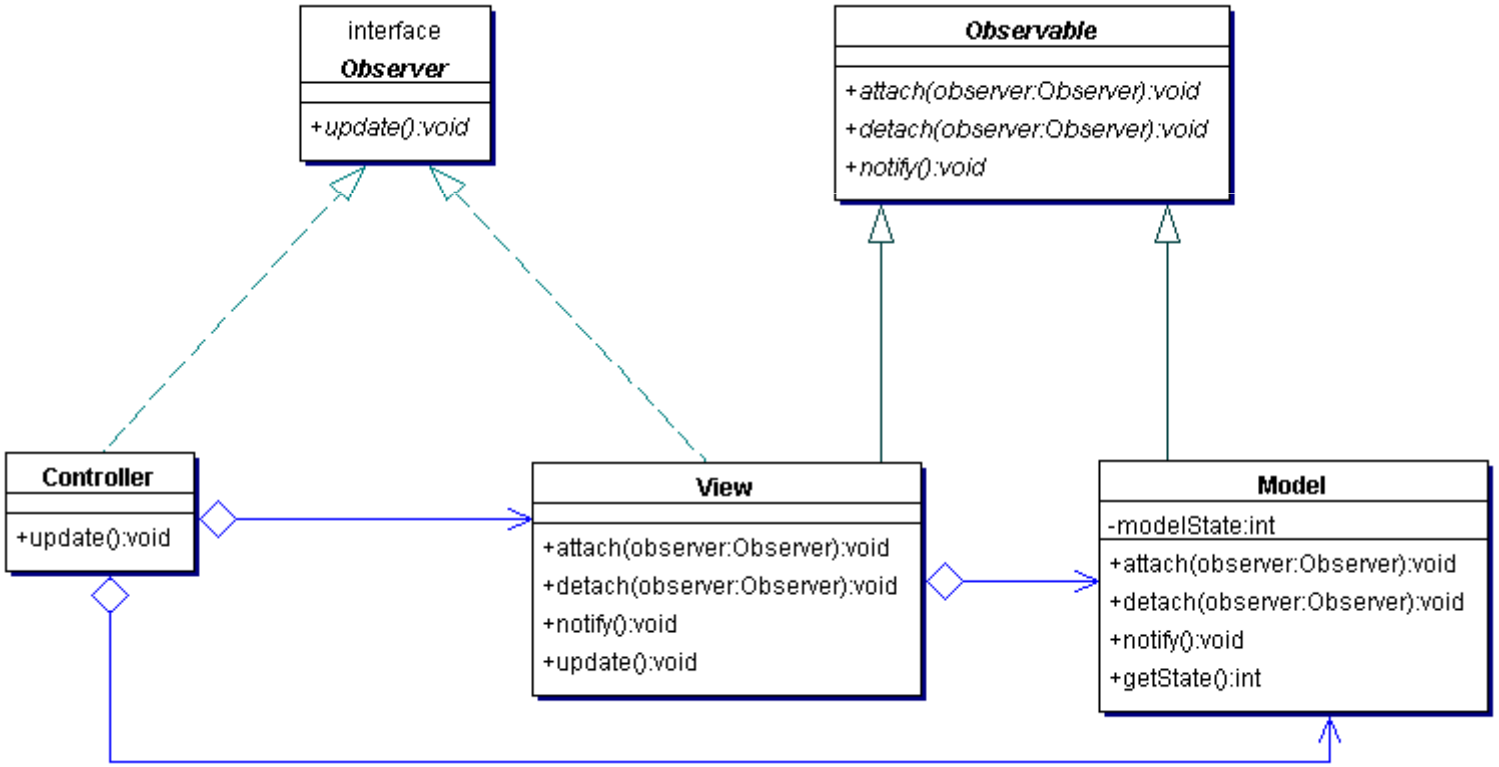
- Rappresentazione del modello adatta all'interazione con l'utente, tipicamente un elemento della GUI
  - Contiene un riferimento al *model*
  - Contiene un riferimento al controller (in MVC tradizionale)
  - Si "abbona" al *model*
- Garantisce che il suo stato rifletta quello del *model* cui è associata. Qualora lo stato del *model* cambiasse, una notifica ne permetterebbe l'aggiornamento
- Possono coesistere viste multiple di un solo *model* per scopi diversi
  - Observer permette il broadcasting delle modifiche al model

Partecipanti - Controller

- Processa e risponde agli eventi, tipicamente azioni dell'utente, e può agire modificando lo stato del model
  - Contiene un riferimento al *model*
  - Contiene un riferimento alla view
  - Ottiene gli input dalla *view* associata e li mappa in richieste verso il *model*
  - Permette di modificare il comportamento del sistema in relazione alle azioni dell'utente

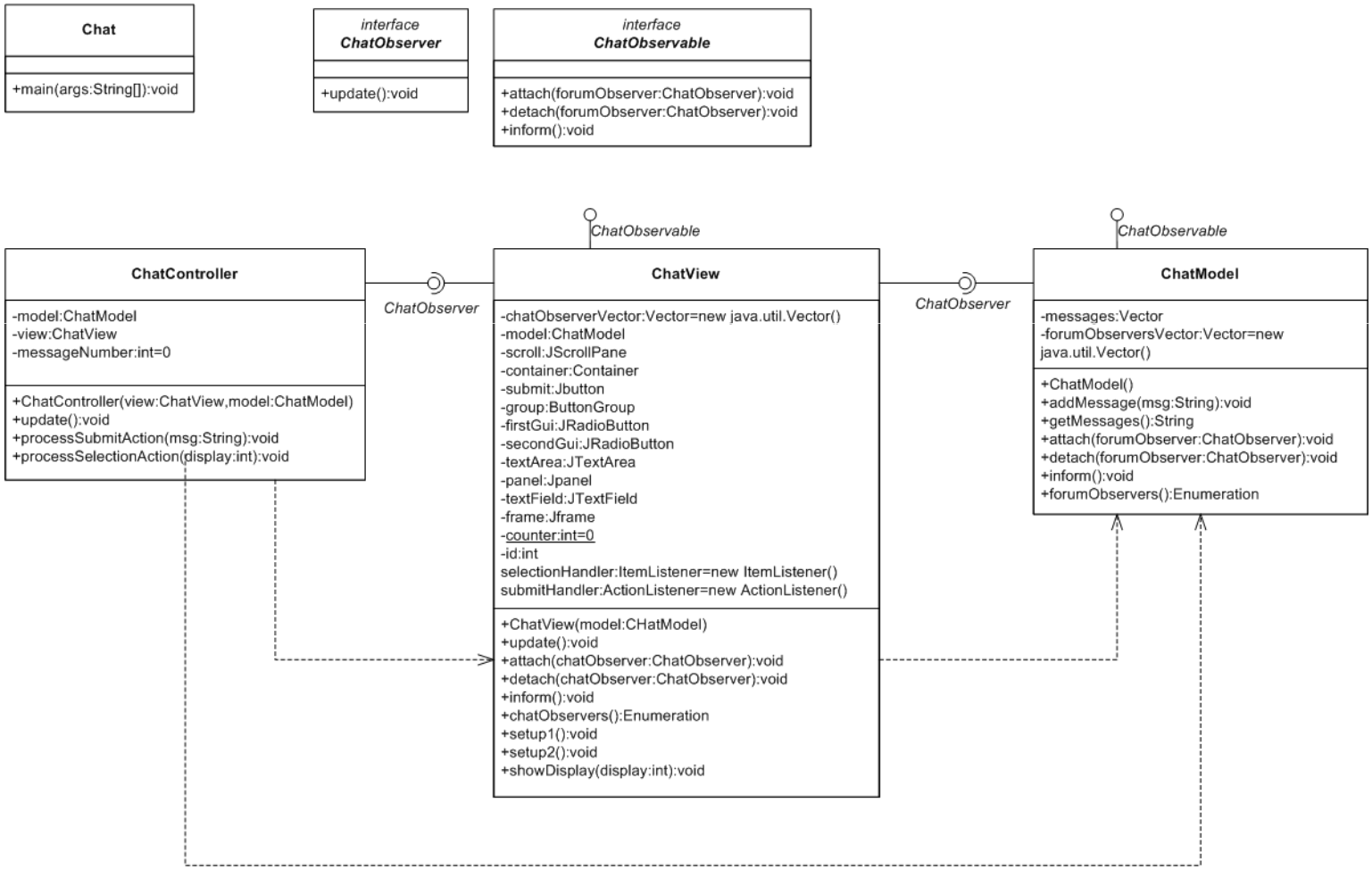
MVC non Tradizionale

- Sia la relazione tra *view* e *model*, che quella tra *view* e *controller* sono regolate dal pattern *Observer*. *View* è osservata dal *controller* per intercettare le azioni dell'utente ed è contemporaneamente abbonata al *model*



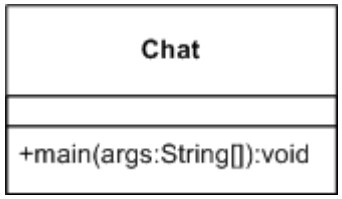
Esempio

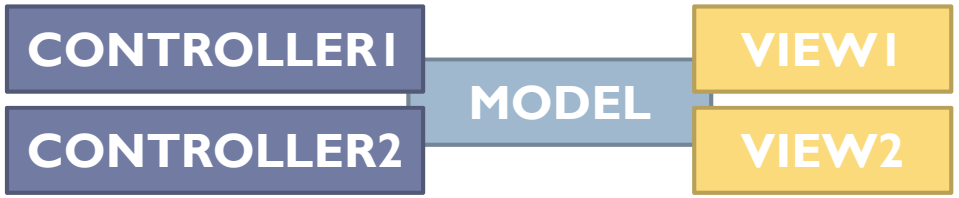
- Una semplice chat locale in Java



La classe Chat

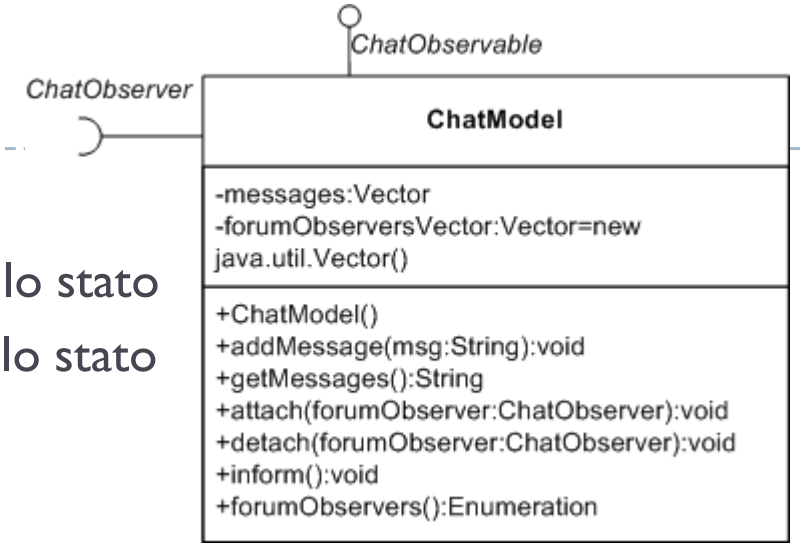
- Definisce il main()
- Esegue lo startup dell'applicazione






```
public class Chat {
    public static void main(String[] args) {
        ChatModel model = new ChatModel();
        ChatView view1 = new ChatView(model);
        ChatView view2 = new ChatView(model);
        ChatController controller1 = new ChatController(view1, model);
        ChatController controller2 = new ChatController(view2, model);
    }
}
```

La classe ChatModel

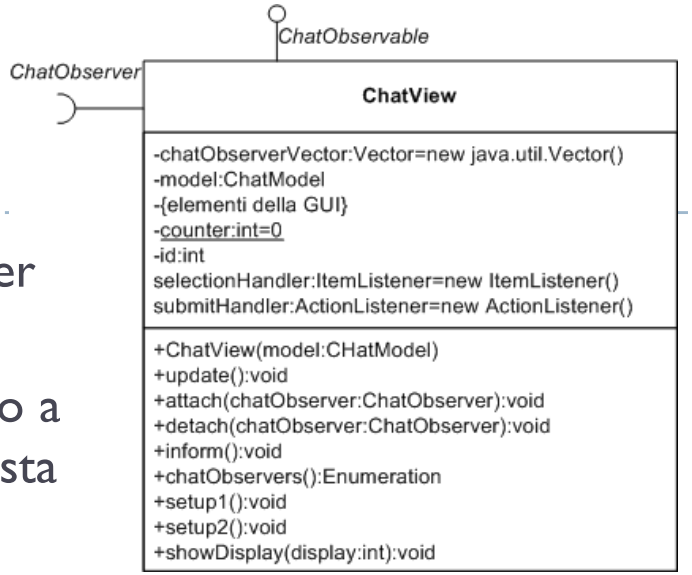


- Implementa ChatObservable
- Contiene un metodo per accedere allo stato
- Contiene un metodo per modificare lo stato

```
public void addMessage(String msg) {
        messages.add(msg);
        inform();
    }

public String getMessages() {
        int length = messages.size();
        String allMessages = "";
        for (int i = 0; i < length; ++i) {
            allMessages += (String)messages.elementAt(i)+"\n";}
        return allMessages;
    }
```

La classe ChatView



- Definisce due metodi *setup1()* e *setup2()* per realizzare due schermate
- Defisce due classi *listener* interne che vanno a delegare ai ChatController abbonati alla vista

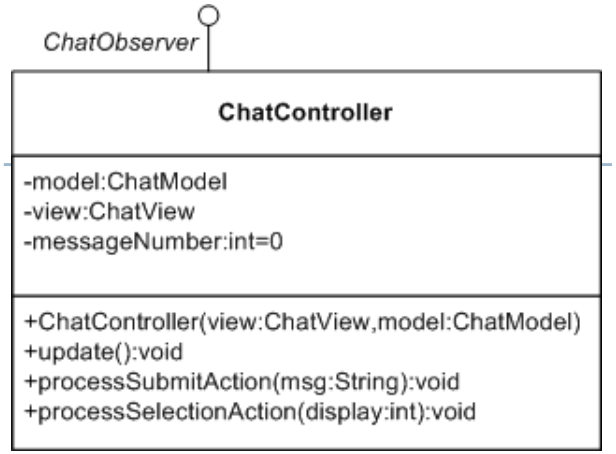
```
public void update() {
        textArea.setText(model.getMessages());}

ItemListener selectionHandler = new ItemListener() {
        public void itemStateChanged(ItemEvent e) {
            ChatController con;
            for (int i = 0; i < chatObserversVector.size(); i++) {
                con = (ChatController)chatObserversVector.get(i);
                con.processSelectionAction(e.getStateChange());}}};

ActionListener submitHandler = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            inform();
            ChatController con;
            for (int i = 0; i < chatObserversVector.size(); i++) {
                con = (ChatController)chatObserversVector.get(i);
                con.processSubmitAction("Client"+id+": "+textField.getText());}}};
```

La classe ChatController

- È abbonata alla vista
- In questo esempio la vera logica di controllo è implementata mediante i metodi *processSubmitAction()* e *processSelectionAction()*




```
public void update() {
        messageNumber++;
        System.out.println(messageNumber);
    }

public void processSubmitAction(String msg) {
        model.addMessage(msg);
    }

public void processSelectionAction(int display) {
        view.showDisplay(display);
    }
```

Demo

Altre implementazioni

- Scenario Rich Client: Swing
  - View e Controller collassano in un unico oggetto chiamato "*delegate*"
  - Il *delegate* interpreta l'interazione dell'utente e aggiorna il model
    - Gestisce la rappresentazione grafica
    - Restano comunque possibili viste multiple sullo stesso model

- Scenario Web:
  - Model: lato server (*entity bean*, *servlet, PHP, ASP, ...)*
  - Controller: lato server, mappa le richieste utente in azioni verso Model
    - Generalmente è il singolo punto d'accesso per le richieste HTTP
  - View*:* lato client, non può avvalersi delle notifiche asincrone di Observer, gli *update()* avvengono a seguito delle richieste dell'utente (HTML, ...)

    - Interazione dell'utente sull'interfaccia
    - Generazione di una richiesta HTTP
    - Il *controller* riceve la richiesta e individua un'azione
    - Il *controller* altera lo stato del *model*
    - Il *controller* sceglie una *view*
    - La *view* aspetta un'ulteriore interazione dell'utente

Conclusioni

- Vantaggi offerti da MVC
  - Sviluppo separato di *model*, *view* e *controller*
  - Testing delle classi del dominio svincolato dall'interfaccia grafica
  - Viste multiple sugli stessi oggetti del dominio

- Contro
  - Progettazione complessa
  - Il numero di classi aumenta drasticamente

Design Pattern correlati

- **Observer** garantisce il disaccoppiamento della presentazione dal modello
- In MVC le viste possono essere nidificate. Tramite **Composite** la classe CompositeView, sottoclasse di View, può sostituire View dove vi sia la necessità di viste all'interno di altre viste
- MVC permette anche di cambiare la risposta a seguito di un'azione sull'interfaccia senza cambiare la presentazione stessa tramite il Controller. La relazione View-Controller è un esempio di pattern **Strategy** e il controller rappresenta il comportamento da variare
- **Factory Method**, per determinare il controller di default da associare ad una vista
- **Decorator**, per aggiungere funzionalità alle viste

Riferimenti

- Patterns of Enterprise Application Architecture - M. Fowler
  Addison Wesley 2005
- Design Patterns - Gamma, Helm, Johnson, Vlissides
  Addison Wesley
- Java BluePrints - Model-View-Controller http://java.sun.com/blueprints/patterns/MVC-detailed.html
- Applications Programming in Smalltalk80: How to use Model-View-Controller - Steve Burbeck
  http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html
- Model-view-controller
  http://en.wikipedia.org/wiki/Model-view-controller
- Model-view-controller - Microsoft patterns & practices
  http://msdn2.microsoft.com/en-us/library/ms978748.aspx
- Model View Controller Pattern (MVC) - Claudio De Sio
  http://www.claudiodesio.com/ooa&d/mvc.htm