

Introduzione

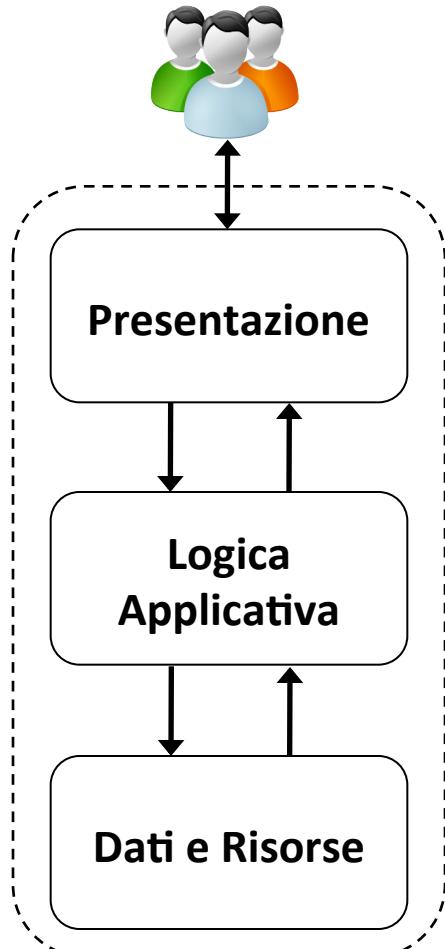
Obiettivo: definire uno schema generale di riferimento (*pattern architettonico*) per la progettazione e strutturazione di applicazioni di tipo interattivo

Pattern Architettonico

- definisce il più alto livello di *astrazione* di un sistema software
- descrive la *struttura* di un sistema software in termini di
 - *sottosistemi* e relative responsabilità
 - linee guida per gestire le *relazioni* e l' *interazione* tra sottosistemi
- la scelta del pattern architettonico è una scelta fondamentale e influenza direttamente le fasi di progettazione e realizzazione

Struttura di una Applicazione Software

La struttura di una applicazione software, e più in generale di un sistema informativo, è caratterizzata da **tre livelli**



- **Presentazione:** insieme dei componenti che gestiscono l'interazione con l'utente
- **Logica Applicativa:** insieme dei componenti che realizzano la logica applicativa, implementano le funzionalità richieste e gestiscono il flusso dei dati
- **Dati e Risorse:** insieme dei componenti che gestiscono i dati che rappresentano le informazioni utilizzate dall'applicazione secondo il modello concettuale del dominio

Model – View – Controller (MVC)

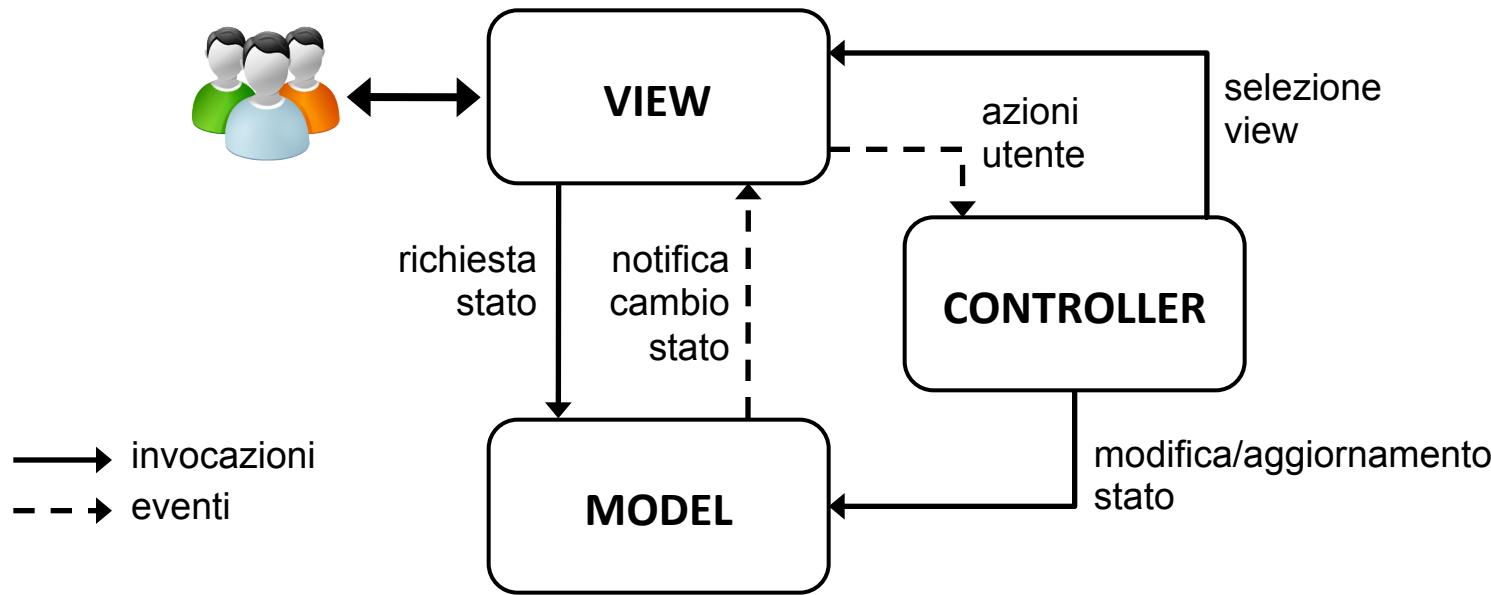
- **Pattern architettonale** per la progettazione e strutturazione modulare di applicazioni software *interattive*
- Originariamente introdotto da Trygve Reenskaug (sviluppatore Smalltalk presso lo Xerox Palo Alto Research Center) nel 1979
- Consente di separare e disaccoppiare il modello dei dati (**model**) e la logica applicativa (**controller**) dalle modalità di visualizzazione e interazione con l'utente (**view**)
 - l'applicazione deve separare i componenti software che implementano il **modello delle informazioni**, dai componenti che implementano la **logica di presentazione** e la **logica di controllo** che gestiscono tali informazioni

MVC: Struttura e Responsabilità (1/2)

Il pattern MVC identifica tre componenti di base

1. **Model:** rappresenta il modello dei dati di interesse per l' applicazione
 - incapsula lo stato dell' applicazione
 - gestisce l' accesso ai dati
 - fornisce le funzionalità per l' aggiornamento dello stato e l' accesso ai dati
 - notifica al *view* i cambiamenti di stato
2. **View:** fornisce una rappresentazione grafica ed interattiva del *model*
 - definisce le modalità di presentazione dei dati e dello stato dell' applicazione
 - consente l' interazione con l' utente
 - riceve notifiche dal *model* e aggiorna la visualizzazione
3. **Controller:** definisce la logica di controllo e le funzionalità applicative
 - gestisce gli eventi ed i comandi generati dall' utente
 - opera sul *model* (modifiche, aggiornamenti, inserimenti) in base agli eventi ed ai comandi ricevuti
 - può selezionare/aggiornare il *view* in base al risultato del processamento o alle scelte dell' utente

MVC: Struttura e Responsabilità (2/2)



- **Model**
 - notifica cambiamenti di stato/dei dati al **view**
- **View**
 - ha riferimento al **model** e può interrogarlo per ottenere lo stato corrente
 - notifica al **controller** gli eventi generati dall' interazione con l' utente
- **Controller**
 - ha riferimento al **model** e al **view**

MVC: Interazioni Fondamentali (1/2)

Nella fase di inizializzazione dell' applicazione

1. viene creato il **model**
2. viene creato il **view** fornendo un riferimento al **model**
3. viene creato il **controller** fornendo riferimenti al **model** e al **view**
4. il **view** si registra come *listener* (o *observer*) del **model**
 - per ricevere notifiche di aggiornamento dal **model** (*observable*)
5. il **controller** si registra come *listener* (o *observer*) del **view**
 - per ricevere dal **view** (*observable*) gli eventi generati dall' utente

MVC: Interazioni Fondamentali (2/2)

Quando un utente interagisce con l'applicazione

1. il **view** riconosce l'azione dell'utente (es. pressione di un bottone) e notifica il **controller** registrato come *listener*
2. il **controller** interagisce con il **model** per realizzare la funzionalità richiesta ed aggiornare/modificare lo stato o i dati
3. il **model** notifica al **view** registrato come *listener* le modifiche e gli aggiornamenti
4. il **view** aggiorna la visualizzazione sulla base del nuovo stato
 - il nuovo stato e le info aggiornate per modificare la visualizzazione possono essere ottenuti dal view con
 - **approccio push:** il model notifica al view sia il cambiamento di stato che le informazioni aggiornate
 - **approccio pull:** il view riceve dal model la notifica del cambiamento di stato e poi accede al model per ottenere le informazioni aggiornate

MVC: Considerazioni

- Il pattern Model-View-Controller definisce una **architettura concettuale** di riferimento
 - *indipendente* dal linguaggio di programmazione
 - utile per impostare la *struttura generale* dell' applicazione in fase di progettazione
 - non definisce in maniera univoca schemi realizzativi ed implementazione
 - le modalità implementative possono dipendere dal linguaggio di programmazione e dal contesto applicativo
- In applicazioni interattive complesse l' architettura software è spesso costituita da un **insieme di componenti** con relazioni di tipo MVC
 - i componenti **model** encapsulano dati e funzionalità
 - possono esserci più componenti **view** per uno stesso **model**
 - ad ogni **view** può essere associato un componente **controller**
- Numerosi *framework* per diversi linguaggi di programmazione sono riconducibili al pattern MVC
 - Java Swing
 - Apple Cocoa
 - PureMVC (per C++, Flex, JavaScript, C#, Perl, PHP, Python, Ruby...)
 -

MVC e Qualità del Software (1/2)

Una progettazione architetturale che

- si basa sulla separazione dei ruoli dei componenti software
 - rende strutturalmente indipendenti moduli con funzionalità differenti
- favorisce qualità esterne ed interne del software

Qualità esterne

- ✓ **estendibilità**
 - semplicità di progetto e decentralizzazione dell' architettura
 - software facilmente estendibile agendo su moduli specifici
- ✓ **riusabilità**
 - possibilità di estrarre e riutilizzare componenti
- ✓ **interoperabilità**
 - interazione tra moduli con ruoli differenti
 - possibilità di creare gerarchie tra componenti

MVC e Qualità del Software (2/2)

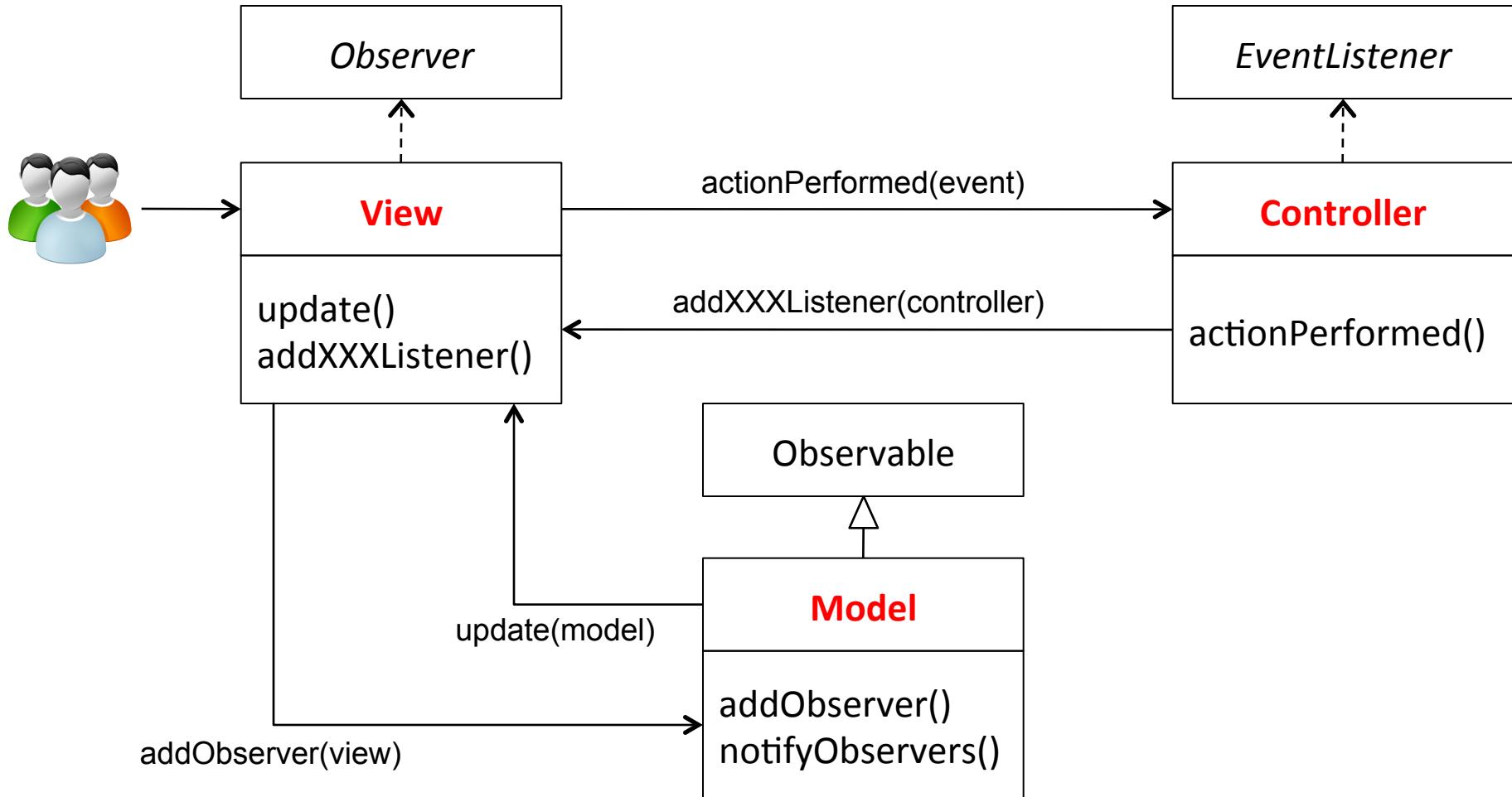
Qualità interne

- ✓ strutturazione
 - struttura del software riflette le caratteristiche del dominio applicativo (dati + controllo + interazione e visualizzazione)
- ✓ modularità
 - organizzazione del software in componenti con funzionalità definite
- ✓ comprensibilità
 - ruoli e funzioni dei componenti sono facilmente identificabili
- ✓ manutenibilità
 - possibilità di intervenire su componenti specifici con effetti nulli o limitati su altri componenti

MVC ed Eventi in Java (1/2)

- Nel paradigma MVC l' interazione tra componenti è basata su meccanismi di propagazione e gestione di **eventi**
 - componenti **view** notificano le azioni dell' utente ai componenti **controller**
 - componenti **model** notificano cambiamenti di stato ai componenti **view**
- In Java si ha che
 - l' interazione tra **view** e **controller** avviene in base al meccanismo di propagazione e gestione **eventi Swing/AWT**
 - i componenti **controller** sono **EventListener** (es. **ActionListener**, **MouseListener**...) associati ai componenti grafici **view** (es. **JButton**)
 - l' interazione tra **model** e **view** avviene secondo il pattern **Observer-
Observable**
 - i componenti **model** estendono la classe **Observable**
 - i componenti **view** implementano l' interfaccia **Observer** e si registrano presso i componenti **model** (**model.addObserver(view)**)
 - i componenti **model** notificano i cambiamenti ai componenti **view** registrati come **observers** (**notifyObservers()**)
 - i componenti **view** ricevono le notifiche (**update(model)**) e aggiornano la visualizzazione

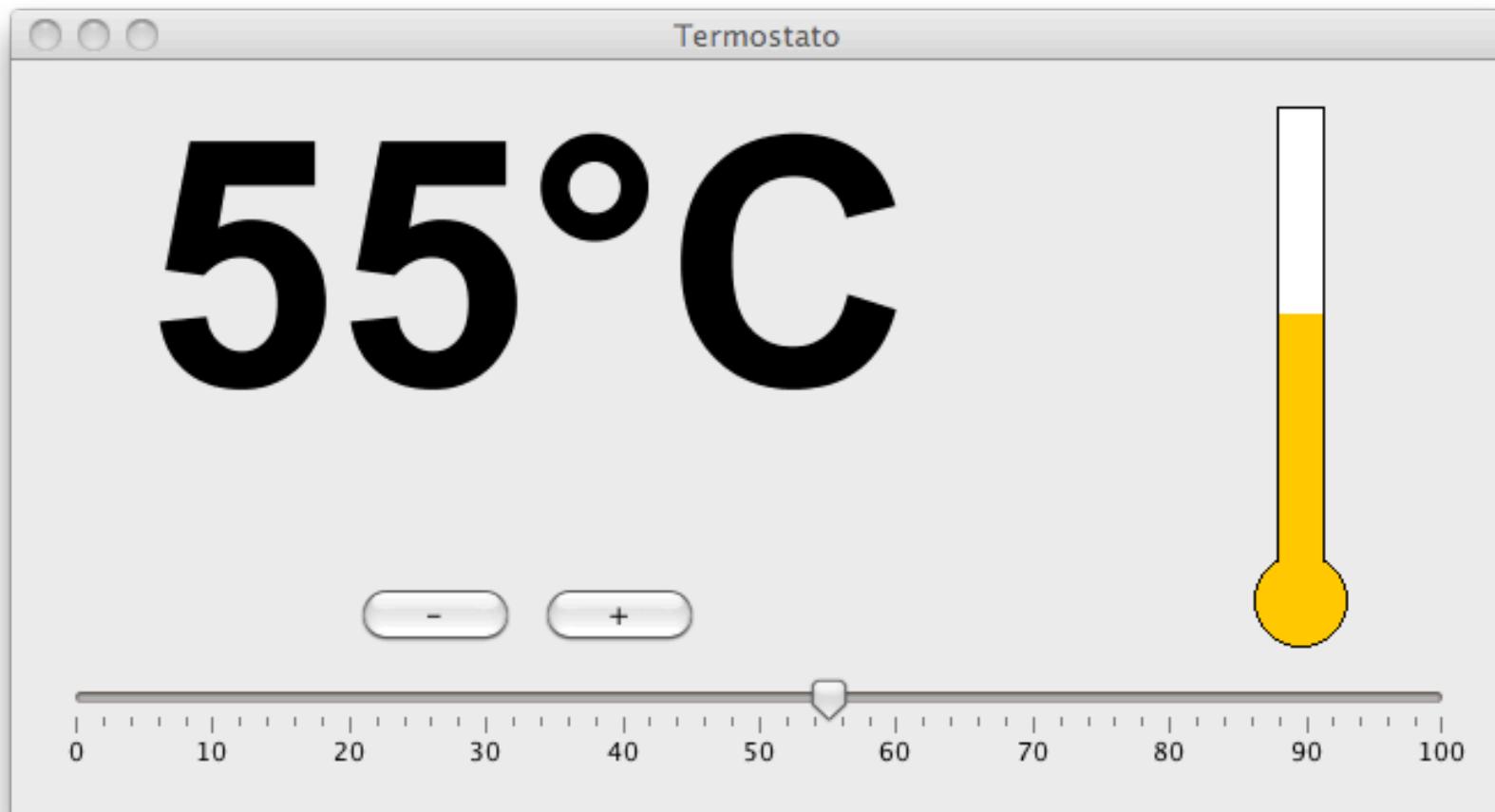
MVC ed Eventi in Java (2/2)



Esempio: Termostato (1/2)

- Si vuole realizzare un' applicazione interattiva che consente di impostare la temperatura tramite un termostato
 - **model**: rappresenta lo stato del termostato con il valore di temperatura impostato
 - **view**: consente all' utente di selezionare il valore di temperatura e visualizza con modalità differenti il valore che viene impostato
 - **controller**: riceve i comandi dall' utente per impostare la temperatura e aggiorna il model di conseguenza

Esempio: Termostato (2/2)

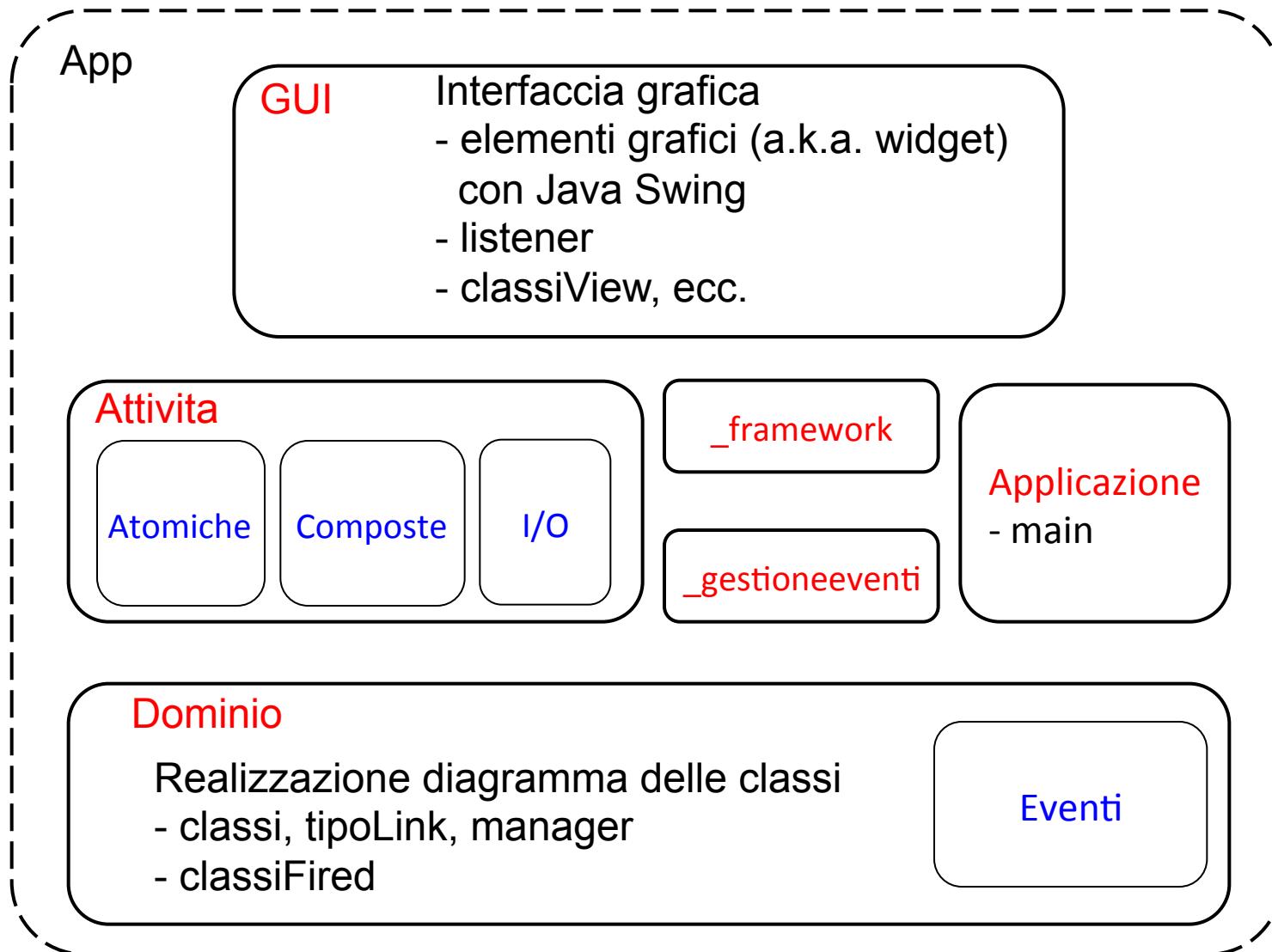


- 1 model
- 3 diverse view
- 2 controller
- Codice disponibile sul sito del corso

Applicazioni e Livelli

- Gli argomenti e la metodologie di progettazione e realizzazione presentati in questo corso possono essere ricondotti ad una **struttura a tre livelli**
 1. il **diagramma delle classi** (e corrispondente realizzazione) rappresenta il **dominio applicativo** di interesse
 - lo stato dell' applicazione è definito dalle istanze delle classi, dai link presenti tra di esse e dallo stato degli oggetti reattivi (secondo il **diagramma stati e transizioni** associato)
 2. il **diagramma delle attività** (e corrispondente realizzazione) descrive il comportamento dell' applicazione e definisce la **logica di controllo**
 - l' esecuzione delle attività opera sullo stato e produce modifiche (creazione nuove istanze e link, rimozione link, transizioni di stato, etc.)
 - l' esecuzione di attività può determinare l' interazione con l' utente (attività I/O grafiche) e la visualizzazione di schermate specifiche
 3. un' **interfaccia grafica** (realizzata con Java Swing) consente la **visualizzazione** dello stato dell' applicazione e l' **interazione** con l' utente
 - consente all' utente di interagire con l' applicazione per attivare e guidare la logica applicativa e di controllo
 - deve riflettere le modifiche e gli aggiornamenti che avvengono sullo stato (nuovi oggetti, eliminazioni oggetti, transizioni di stato, etc.)

La Struttura delle nostre Applicazioni



La Struttura dei Package

```
app
|
+--- dominio
|   | -- NomeClasse.java
|   | -- NomeClasseFired.java
|   | -- TipoLinkNomeLink.java
|   | -- ManagerNomeLink.java
|   | -- EccezioneCardMinMax.java
|   | -- EccezionePrecondizioni.java
|   | --
|   \-- eventi
|       | -- NomeEvento.java
|       \-- ...
+--- attivita
|   | -- AttivitaIO.java
|   +--- atomiche
|       | -- NomeAttivita.java
|       \--...
|   +--- complesse
|       | -- AttivitaPrincipale.java
|       | -- NomeAttivitaComplessa.java
|       \-- ...
|   |
|   +--- _framework
|       | -- Executor
|       \-- Task
+--- _gestioneeventi
|   | -- Environment
|   \-- ...
+--- applicazione
|   | -- Main.java
|   |
\-- gui
    | -- ClasseGraficaSwing.java
    | -- XXXListener.java
    | -- NomeClasseView.java
    | -- ErrorNotifier.java
    \-- ...
```

Note

- Nel package **applicazione** è presente la classe **Main.java**
 - definisce ed implementa il metodo **main**
 - si occupa di inizializzare e avviare l' applicazione
- Nel package **gui** è presente la classe **ErrorNotifier.java**
 - definisce ed implementa il metodo

```
public static void notifyError(String message)
```
 - consente di mostrare all' utente un messaggio di errore al verificarsi di una eccezione
 - se in una porzione qualsiasi del codice si verifica una eccezione
 - l' eccezione viene catturata e gestita localmente
 - se è necessario visualizzare un messaggio di errore viene invocato il metodo **ErrorNotifier.notifyError** specificando il messaggio da visualizzare
 - la specifica modalità di visualizzazione del messaggio di errore può dipendere dalla applicazione (es. output su console, finestra, etc.) ed è definita nell' implementazione del metodo

MVC per Classi di Dominio e Widget (1/2)

- Durante la fase di progettazione vengono identificati i concetti e le entità che costituiscono il dominio applicativo
 - si definisce il **diagramma delle classi**
 - si identificano gli oggetti reattivi ai quali associare un **diagramma degli stati e transizioni**
- La logica applicativa e di controllo viene descritta tramite il **diagramma delle attività**
- In un' applicazione interattiva, un' opportuna **interfaccia grafica** consente all' utente di visualizzare lo stato ed i dati
- Durante l' esecuzione delle attività è spesso necessario **aggiornare e modificare** le informazioni visualizzate a causa di
 - creazione oggetti, link tra oggetti, etc.
 - transizioni di stato di oggetti reattivi

MVC per Classi di Dominio e Widget (2/2)

- Affinché l' interfaccia grafica rifletta i cambiamenti degli oggetti del dominio è necessario
 - identificare gli oggetti del dominio (**model**) cui corrisponde un elemento di visualizzazione grafica (**view**)
 - definire uno o più componenti grafici (**widget**) utilizzati per la rappresentazione e visualizzazione degli oggetti di dominio
 - stabilire una relazione **Model-View** (cioè **observable-observer**) tra gli oggetti del dominio e i corrispondenti componenti grafici
- Da un punto di vista realizzativo, per ogni classe **NomeClasse** del dominio cui è associata una visualizzazione
 - definire nel package **gui** una classe **NomeClasseView** che rappresenta il componente grafico (**widget**) associato all' oggetto di dominio **NomeClasse** (es. un pannello, una finestra, etc.)
 - registrare **NomeClasseView** come *observer* del corrispondente oggetto **NomeClasse** (secondo il pattern **observer-observable**)
 - identificare i cambiamenti di stato (es. inserimento nuovo link, aggiornamento variabile istanza, transizione stato) cui deve corrispondere un aggiornamento del componente **NomeClasseView** e notificare l' avvenuto cambiamento di stato (tramite **notifyObservers()**) affinché venga aggiornata la visualizzazione
- I dettagli realizzativi dipendono dalla logica applicativa e di controllo della specifica applicazione (si vedano esercitazioni/esercizi di esame)