Programmazione Java Avanzata

PATTERN

Anno 2011/2012

Pattern

- Una soluzione progettuale generale a un problema ricorrente
- Elementi per il riuso di software ad oggetti
- Vantaggiosi e ‘’obbligatori’’ da usare

Pattern

Un pattern è caratterizzato da:

- *Nome* - rappresentativa del pattern stesso
- *Problema* - la situazione (o condizione) alla quale si può applicare il pattern
- *Soluzione* – la configurazione degli elementi adatta a risolvere il problema
- *Conseguenze*, vantaggi (risultati) e svantaggi (vincoli) che derivano dall'applicazione del pattern

Pattern

Importanti sono le conseguenze:

Esse comprendono considerazioni di tempo e di spazio, possono descrivere implicazioni del pattern con alcuni linguaggi di programmazione e l'impatto con il resto del progetto

Pattern

Il pattern Singleton

Pattern - Singleton

Semplice da presentare e implementare (in Java), risulta molto utilizzato

Nome: Singleton

Problema: assicurare che per una determinata classe esista un'unica istanza attiva.

Utile nel caso in cui si abbia la necessità di centralizzare informazioni e comportamenti in un'unica entità condivisa da tutti i suoi utilizzatori. Es. Spooler di stampa

Pattern - Singleton

Soluzione: associare alla classe stessa la responsabilità di creare le proprie istanze.

Conseguenze:

- accesso controllato alla singola istanza;
- possibilità di usare un numero variabile di istanze;
- una maggiore flessibilità (es. posso modificare i dati gestiti dal singleton senza invalidare l'unicità dell'istanza)

Pattern - Singleton

Implementazione in Java:

```
public class Gestore {
    private static Gestore gestore = null;

    private Gestore() {}

    public static synchronized Gestore getGestore() {
        if (gestore == null)
            gestore = new Gestore();
        return gestore;
    }
}
```

Pattern - Singleton

Controllo dell'unicità:

```
public static void main(String[] args) {

        Gestore g1=Gestore.getGestore();
        System.out.println(g1.toString());

        Gestore g2=Gestore.getGestore();
        System.out.println(g2.toString());
}
```

Pattern

Il pattern MVC

(separazione delle responsabilità)

Pattern - MVC

Scheda (molto) sintetica

Nome: MVC (Model-Control-View)

Problema: manutenibilità del codice (vedi prossime slide)

Soluzione: suddivisione del codice in tre ‘’classi’’ distinte (vedi prossime slide)

Conseguenze: promuove il riuso del codice, consente lo sviluppo del codice in parallelo, facilita l’implementazione di nuove funzionalità, e molte altre cose positive …

Pattern - MVC

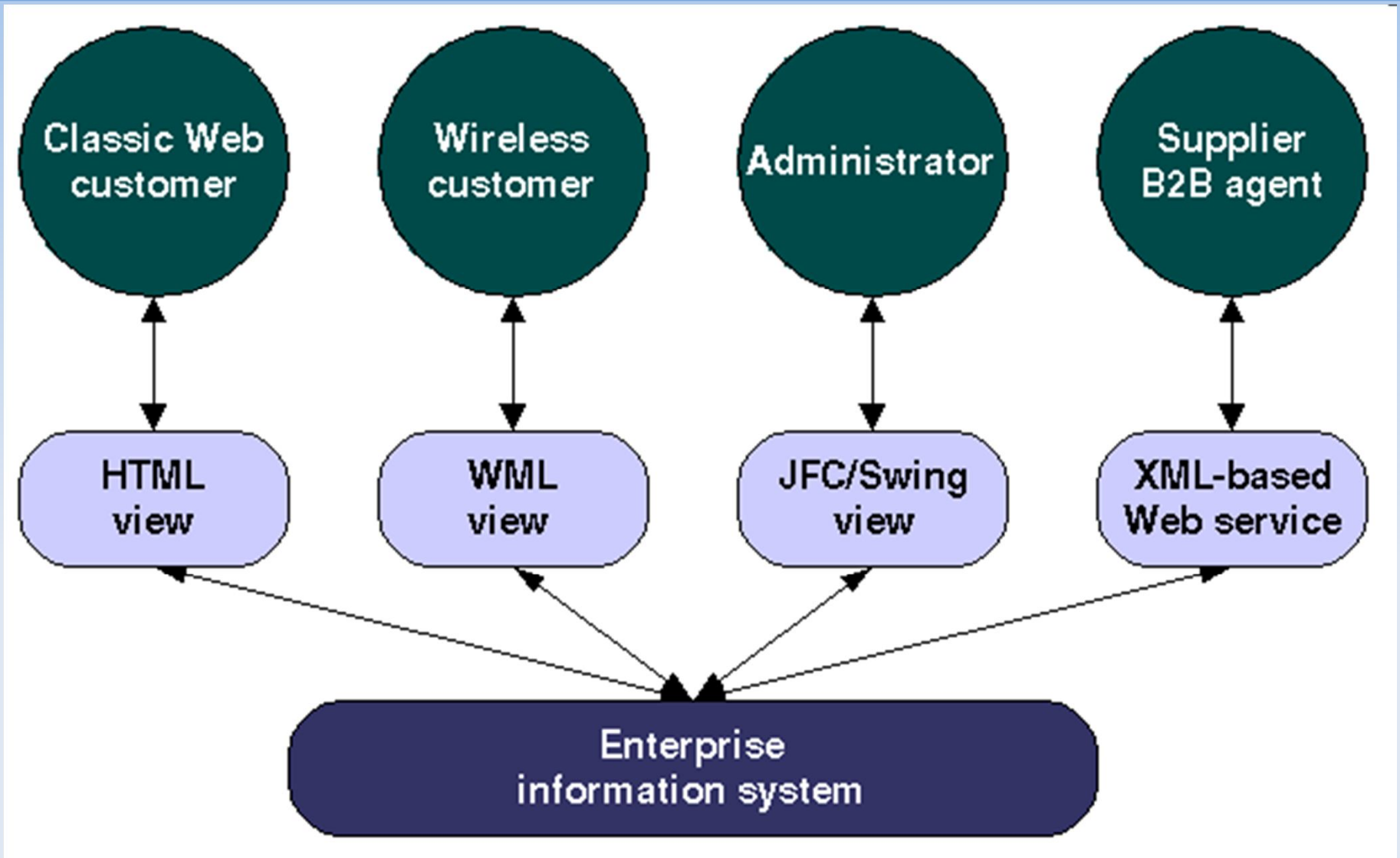
Problema a cui si applica: esempio

Consideriamo un sito e le pagine per gestire l'autenticazione di un utente.

Una pagina risponde alla richiesta di login, elabora i dati, memorizza delle informazioni, e mostra il risultato della procedura (autorizzato o no).

Una modifica ad una delle funzionalità sopra coinvolge tutte le altre funzionalità.

Esempio


Classic Web customer
Wireless customer
Administrator
Supplier B2B agent
HTML view
WML view
JFC/Swing view
XML-based Web service
Enterprise information system

Esempio: Vincoli

- Gli stessi dati aziendali devono essere presentati in più modi: HTML, WML, JFC/Swing, XML.
- Gli stessi dati aziendali devono essere aggiornati a seguito di differenti tipi di interazione: uso di link su una pagina HTML o WML, uso di pulsanti su una GUI JFC/Swing, uso di messaggi SOAP scritti in XML.
- Supportare molteplici tipi di presentazioni (view) ed interazioni non deve avere impatto sui componenti

Problemi/1

- Il mix fra il codice che riguarda la logica dell'applicazione (login, inserimento informazioni, ecc.) e quello che riguarda la presentazione dei risultati (HTML, XML, …) causa:
  - Duplicazione di codice (relativo alla logica) in ogni nuova interfaccia (presentazione) che viene aggiunta all'applicazione.
  - Difficoltà nel debug del codice e nella manutenzione.

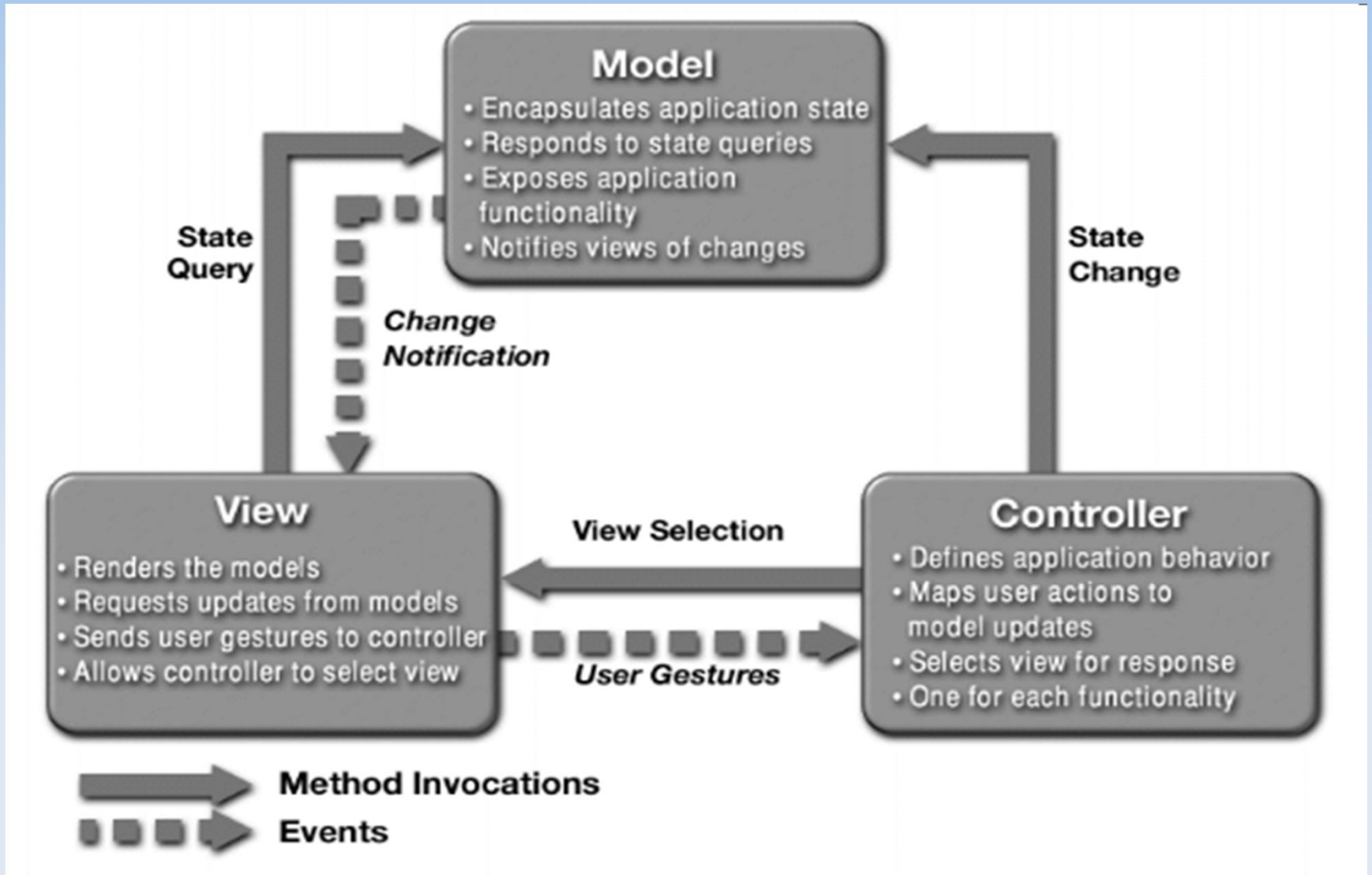
Problemi/2

- Difficoltà nel mantenere costante la logica al variare della presentazione (presentazione → logica).
- Se in futuro cambia la logica, si devono operare tante modifiche per quante volte essa è ripetuta nei moduli di presentazione (logica → presentazione).
- L'implementazione di una pagina richiede competenze estese (Java, HTML, XML, SOAP, …)

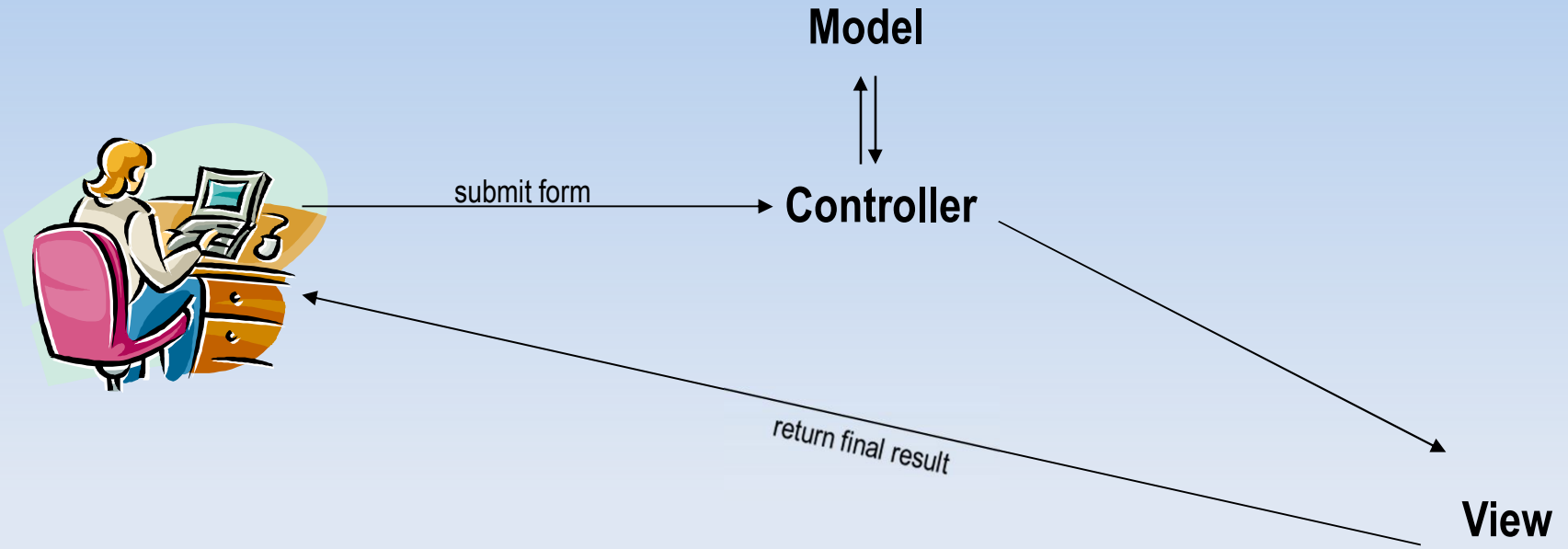
Soluzione: Model-View-Controller

- **Model.** Si occupa solo dei dati utili all'applicazione e fornisce i metodi per accedere ad essi.
- **View**. Si occupa solo della gestione dell'interfaccia utente. Non si preoccupa della logica applicativa o dell'elaborazione/validazione dell'input. Garantisce che l'interfaccia rifletta lo stato corrente del modello.
- **Controller**. Si occupa soltanto di validare/elaborare l'input dell'utente (fornito dalla parte View) e tradurlo in aggiornamenti (messaggi) che vengono passati al modello. Non si occupa di come sia ricevuto l'input (interfaccia) o di che cosa faccia il modello con questi aggiornamenti.

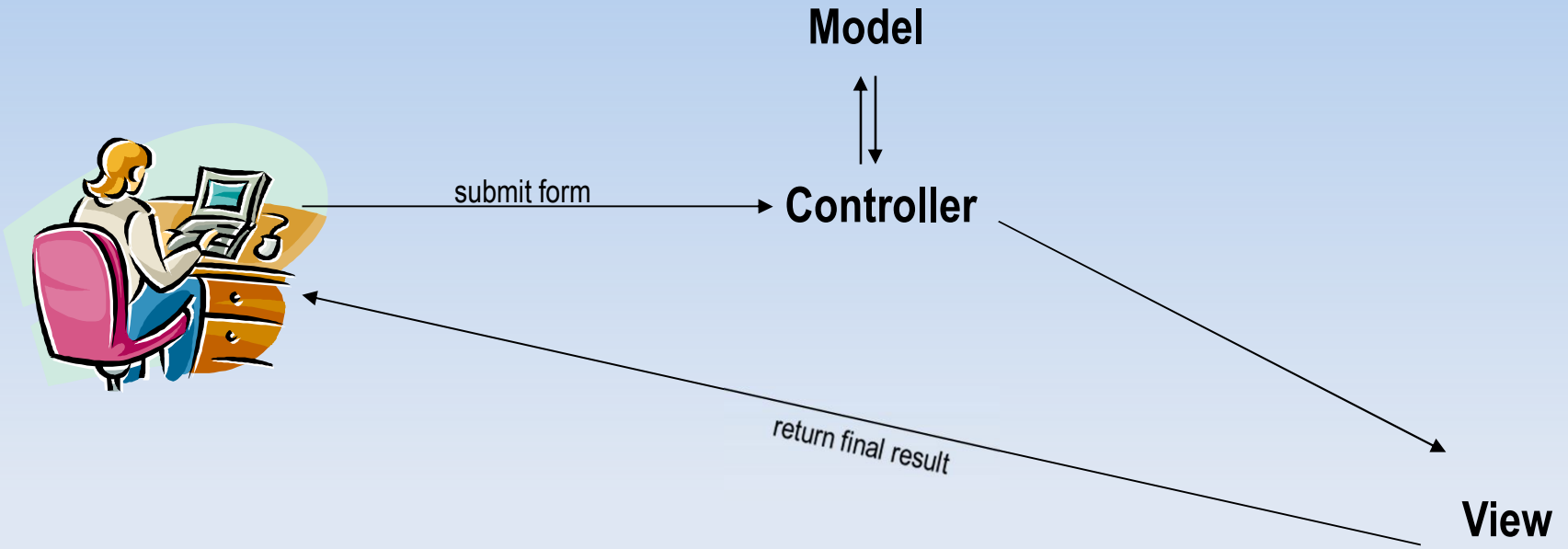
Soluzione: Model-View-Controller



Flusso di controllo MVC



Flusso di controllo MVC



Pattern MVC

Responsabilità del Model:

- Il model memorizza i dati e fornisce del metodi specifici per l'applicazione per definire i valori di questi.
- I metodi di gestione dei dati non sono generici. Sono personalizzati per l'applicazione e devono essere noti al controller e alla view.
- Il model deve anche fornire alle view un modo con cui queste possono registrarsi e annullare la loro registrazione. In tal modo, se determina che il suo stato è cambiato, deve inviare una notifica a tutte le viste registrate.

Pattern MVC

I dati potrebbero essere ovunque:

File locale, File remoto, DBMS Oracle, DBMS MySQL, …

Se sono su un DBMS, possiamo utilizzare Hibernate per implementare il Model.

Hibernate facilita la memorizzazione (persistenza) e gestione di oggetti Java associati a relazioni di un DBMS.

La suddivisione dell'applicazione in package dovrebbe rispecchiare l'implementazione del pattern MVC

Pattern MVC

I dati potrebbero essere ovunque:

File locale, File remoto, DBMS Oracle, DBMS MySQL, …

Se sono su un DBMS, possiamo utilizzare Hibernate per implementare il Model.

Hibernate facilita la memorizzazione (persistenza) e gestione di oggetti Java associati a relazioni di un DBMS.

Pattern

Il pattern DAO

(Data Access Object)

Pattern - DAO

Nome: DAO

Problema: disaccoppiare l'accesso ai dati rispetto alla sua memorizzazione sottostante.

Es. i metodi `create(…), read(…), update(…), delete(…)`

usati per la persistenza dei dati (CRUD) andrebbero tutti modificati al cambiare della sorgente dei dati (passo da Oracle a MySQL)

Pattern - DAO

Soluzione: sposto la logica di accesso ai dati all'oggetto DAO che indica quali funzionalità sono supportate nascondendo la sorgente

Conseguenze:

- portabilità delle applicazioni da una sorgente di dati ad un'altra
- posso iniziare l'implementazione (sfruttando il pattern MVC) senza dovermi occupare di quale sia la sorgente dei dati

Pattern - DAO

Implementazione in Java:

vedi esercitazione

Pattern

Il pattern Factory

(Factory Method)

Pattern - Factory

Nome: Factory

Problema: creare oggetti senza specificare esplicitamente la classe a cui gli oggetti appartengono.

Soluzione: definire metodi ad hoc (factory) per la creazione degli oggetti.

Conseguenze: rende completamente interscambiabili le implementazioni che soddisfano un'interfaccia

Pattern - Factory

Implementazione in Java:

vedi esercitazione

```
public class UtenteDAOFactory {
  private static UtenteDAO dao = null;
  public static UtenteDAO getDAO() {
    if ( dao == null ) {
      dao = new UtenteDAOSimpleImpl();
    }
    return dao;
  }
}
```

- *Modifico istanziazione del DAO*

```
UtenteDAO udf= UtenteDAOFactory.getDAO();
```

- *Modifico visibilità del costruttore dei DAOImpl*

Pattern

Il pattern Decorator

Pattern - Decorator

Nome: Decorator

Problema: aggiungere funzionalità (Decorazioni) ad una classe (Component) a run-time

Soluzione: - Definire un'interfaccia (astrazione) del Component, che le classi concrete implementano

Definire una classe (Decorator) che rappresenta l a decorazione, implementa l'interfaccia Component e contiene l’oggetto che decora

Pattern - Decorator

Gelato (Componente), Decorazioni (panna, 3_gusti, max, …)

```
public interface Gelato {
    public double getPrezzo();}


public class GelatoBase implements Gelato {
    public double getPrezzo() {return 1;}
}

abstract public class GelatoDecorator implements Gelato {
    private Gelato decoratedGelato;
    public GelatoDecorator(Gelato decoratedGelato) {
        this. decoratedGelato = decoratedGelato; }
    public double getPrezzo() {
        return decoratedGelato.getPrezzo();
    }
 }
```

Pattern - Decorator

```
public class ConPanna extends GelatoDecorator {
    public conPanna(Gelato decoratedGelato) {
        super(decoratedGelato);
    }
     public double getPrezzo() {
        return super.getPrezzo() + 0.2;
    }
}

public class Max extends GelatoDecorator {
    public Max(Gelato decoratedGelato) {
        super(decoratedGelato);
    }
     public double getPrezzo() {
        return super.getPrezzo() + 0.7;
    }
}

…
```

Pattern - Decorator

```
Gelato g = new GelatoBase();
System.out.println("Prezzo: " + g.getPrezzo());

g = new conPanna(g);  //Gelato con panna
System.out.println("Prezzo: " + g.getPrezzo());

g = new Max(g);  //Gelato Max con Panna
System.out.println("Prezzo: " + g.getPrezzo());

…
```

Pattern

Il pattern

Inversion Of Control

Pattern - IoC

Nome: Inversion Of Control

Problema: la dipendenza tra le classi

Il principio della separazione delle responsabilità (MVC) porta all'esplosione del numero delle classi (ognuna specializzata su una particolare responsabilità). Le classi arrivano ad accumulare dipendenza da decine di altre classi.

Le istanze per essere inizializzate richiedono di crearne altre, e queste ancora altre, e così via

Pattern - IoC

Conseguenze delle dipendenze (RFI):

1. E’ complicato modificare una classe perchè tale cambiamento coinvolge altre classi (Rigidità)
2. Effettuando una modifica ad una classe, un’altra parte del sistema smette di funzionare (Fragilità)
3. E’ complicato riutilizzare moduli di un’applicazione perchè prima devono essere “sganciati” dal resto del codice (Immobilità)

Pattern - IoC

Esempio:

```
public Class OggettoA { …}

public Class OggettoB {
      private OggettoA a;
      OggettoB (){
          this.a=new OggettoA(…);
      }
      …
}
```

OggettoB dipende da OggettoA
Il costruttore genera un forte accoppiamento tra le classi

Pattern - IoC

Principi alla base della soluzione:

1. I moduli di alto livello non devono dipendere da quelli di basso livello: entrambi dovrebbero dipendere solo da astrazioni (interfacce)

2. Le astrazioni non dovrebbero dipendere dai moduli di basso livello, questi ultimi dovrebbero dipendere dalle astrazioni.

Pattern - IoC

Dependency Injection come soluzione:

Es. *Construction Injection* - la dipendenza viene iniettata tramite l'argomento del costruttore

```
public Class OggettoB {

      private OggettoA a;
      OggettoB (OggettoA a){
         this.a=a;
      }
      …
}
```

Es. *Setter Injection* - la dipendenza viene iniettata attraverso un metodo "set"