ODD

Object

Design

Document

FormAct

| Riferimento | FRM\_ODD\_v0.9 |
| --- | --- |
| Versione | 0.9 |
| Data | 02/02/2022 |
| Destinatario | Prof.ssa F. Ferrucci |
| Presentato da | Team I-Basher |
| Approvato da |  |

RevisionHistory

| **Data** | **Versione** | **Descrizione** | **Autori** |
| --- | --- | --- | --- |
| 20/12/2021 | 0.1 | Prima stesura | Tutti |
| 23/12/2021 | 0.2 | Creazione package diagram | Tutti |
| 24/12/2021 | 0.3 | Stesura class interfaces per i package autenticazione, registrazione e gestione pf | Tutti |
| 03/01/2022 | 0.4 | Stesura class interfaces per i package pf, pf personalizzato | Tutti |
| 04/01/2022 | 0.5 | Completamento capitolo 1 | Tutti |
| 15/01/2022 | 0.6 | Aggiornamento class interfaces e scrittura glossario | Tutti |
| 17/01/2022 | 0.7 | Revisione | Tutti |
| 25/02/2022 | 0.8 | Piccoli accorgimenti | Tutti |
| 07/08/2022 | 0.9 | Aggiornamento descrizione design pattern | Tutti |

Sommario

1. Introduzione
   1. Object design goals
   2. Linee guida per la documentazione dell’interfaccia
   3. Definizioni, acronimi e abbreviazioni
2. Packages
3. Class interfaces
4. Design patterns

1 Introduzione

FormAct si propone di fornire un insieme di insegnamenti nel campo delle discipline tecnologiche informatiche.

Gli insegnamenti proposti prendono il nome di percorsi formativi.

Il sistema offre vari percorsi formativi prestabiliti dal sistema.

Ogni percorso formativo è insegnato da un formatore ad uno studente, quindi le lezioni sono individuali e sono svolte a distanza.

Un formatore interessato ad insegnare potrà insegnare un percorso formativo di un’area affine al suo ambito disciplinare.

Per fare questo, il formatore creerà un suo percorso formativo attenendosi agli ambiti imposti dalla piattaforma.

La piattaforma può essere gestita da più amministratori.

Gli amministratori si occupano principalmente di:

- visualizzare gli utenti;

- visualizzare i dati anagrafici di tutti gli utenti;

- visualizzare i percorsi formativi in cui si è iscritto uno studente;

- visualizzare tutti i percorsi formativi;

- visualizzare le valutazioni di un formatore;

- disiscrivere un utente.

Tutti gli utenti possono modificare il loro profilo.

Un formatore, oltre a creare un percorso formativo, può decidere di:

- visualizzare tutti i suoi percorsi formativi;

- visualizzare la scheda dei percorsi formativi;

- eliminare i suoi percorsi formativi;

Uno studente può:

- cercare un percorso formativo;

- ottenere un piano formativo personalizzato adatto alle sue esigenze con l’ausilio di un modulo di intelligenza artificiale;

- iscriversi ad un percorso formativo;

- valutare un formatore;

- disiscriversi da un percorso formativo.

**1.1 Object design goals**

Per l’implementazione e lo sviluppo del sistema si è scelto di utilizzare la programmazione orientata agli oggetti nello specifico l’utilizzo del linguaggio Java, quindi gli obiettivi che deve raggiungere il sistema sono le stesse caratteristiche della programmazione ad oggetti e sono elencati qui di seguito.

**Riusabilità:**

Il sistema deve basarsi sulla riusabilità del codice attraverso l’utilizzo di ereditarietà e design patterns.

**Incapsulamento:**

Il sistema deve garantire la segretezza sui dettagli implementativi delle classi grazie all’utilizzo delle interfacce, rendendo possibile l’utilizzo di funzionalità offerte da diversi componenti o layer sotto forma di black box, cioè scatola nera, con la quale l'interazione avviene solo e solamente tramite i metodi definiti dall’ 'interfaccia.

**Robustezza:**

Il sistema deve risultare robusto, reagendo correttamente a situazioni impreviste attraverso il controllo degli errori e la gestione delle eccezioni.

**1.2 Linee guida per la documentazione dell’interfaccia**

Le linee guida includono una lista di regole che gli sviluppatori dovrebbero rispettare durante la progettazione delle interfacce. Per la loro costruzione si è fatto riferimento alla convenzione java nota come Sun Java Coding Conventions.

Di seguito una lista di link alle convenzioni usate per definire le linee guida:

* Java Sun: <https://checkstyle.sourceforge.io/sun_style.html>
* HTML: https://www.w3schools.com/html/html5\_syntax.asp

**1.3 Definizioni, acronimi e abbreviazioni**

Vengono riportati di seguito alcune definizioni presenti nel documento:

* **Package:** gruppo di classi, interfacce o file correlati;
* **Design pattern:** template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e flessibilità;
* **Interfaccia**: insieme di signature delle operazioni offerte dalla classe;
* **View:** nel pattern MVC rappresenta ciò che viene visualizzato a schermo da un utente e che gli permette di interagire con le funzionalità offerte dalla piattaforma;
* **Model:** nel pattern MVC rappresenta la componente che si occupa di catturare il comportamento dell'applicazione in termini di dominio del problema, è indipendente dall'interfaccia utente, ovvero dalla componente View. Il model gestisce direttamente i dati.
* **Control:** nel pattern MVC è il responsabile di accettare l'input dell’utente e di convertirlo in comandi per il model e/o la view. Inoltre è la componente principale che si occupa di eseguire la logica applicativa, e quindi definisce il comportamento dell’applicazione.
* **DAO:** Data Access Object, è un design pattern che ci occupa della gestione della persistenza. Fondamentalmente è una [classe](https://it.wikipedia.org/wiki/Classe_(informatica)) con relativi [metodi](https://it.wikipedia.org/wiki/Metodo_(informatica)) che rappresenta un'[entità tabellare](https://it.wikipedia.org/wiki/Modello_relazionale) di un [database management system](https://it.wikipedia.org/wiki/Database_management_system) basato sul [modello relazionale](https://it.wikipedia.org/wiki/Modello_relazionale). Il vantaggio relativo all'uso del DAO è dunque il mantenimento di una rigida separazione tra le componenti di un'applicazione, come possono essere il model e la view, dalla persistenza.

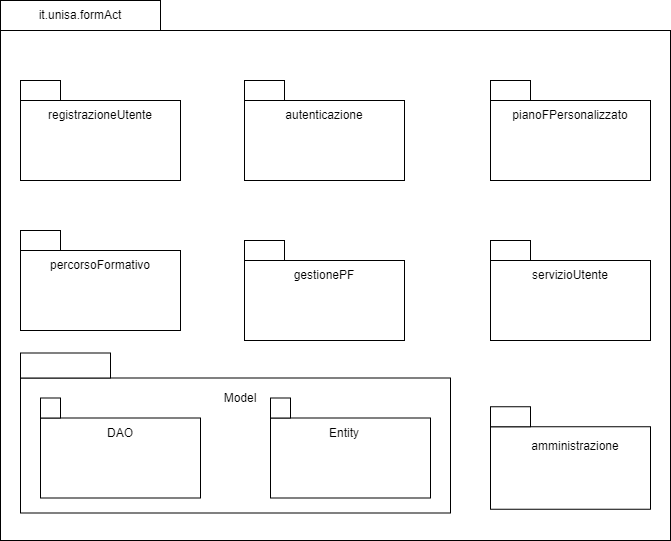
2 Packages

**Package FormAct**

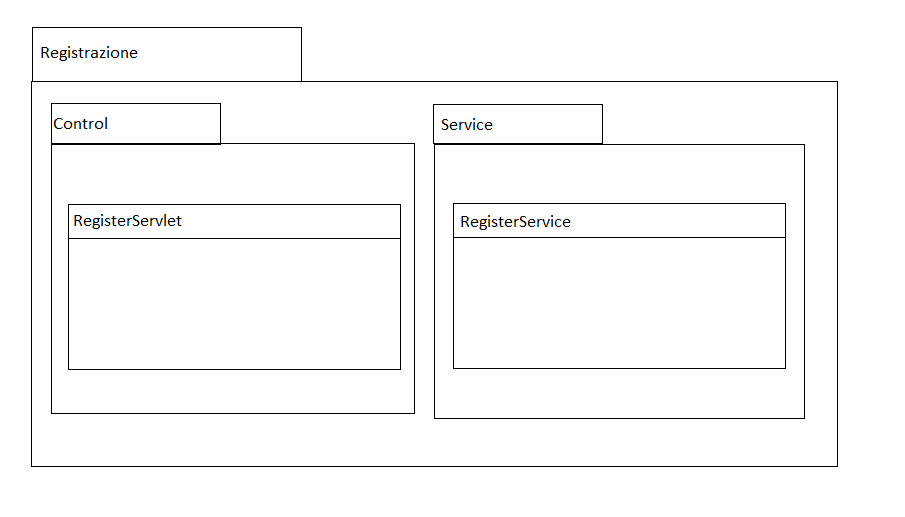
Nella presente sezione si mostra la struttura del package principale di formAct. Le scelte architetturali adottate per la struttura dei package sono:

1. Creare un package separato per ogni sottosistema, contenente le classi service e controller del sottosistema, ed eventuali classi di utilità usate unicamente da esso.

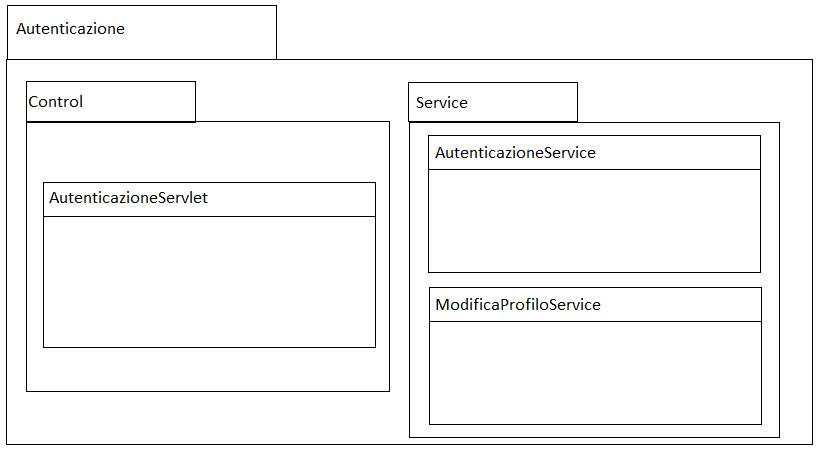
2. Creare un package separato per le classi del model, contenente le classi entity e i DAO per l’accesso al DB. Questo package crea una dipendenza tra esso e tutte le classi



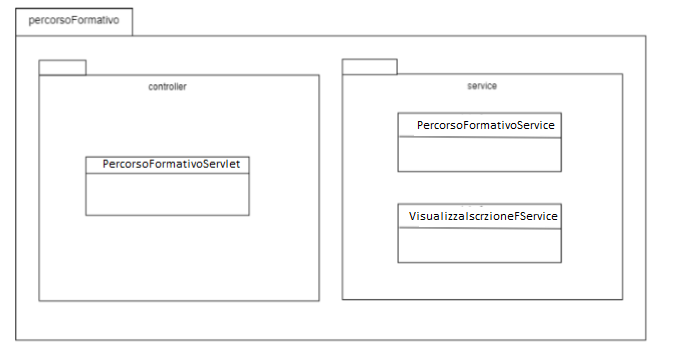
**Package Registrazione Utente**



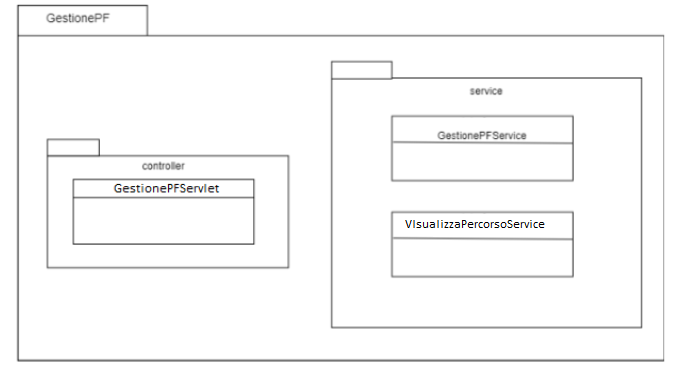
**Package Autenticazione**



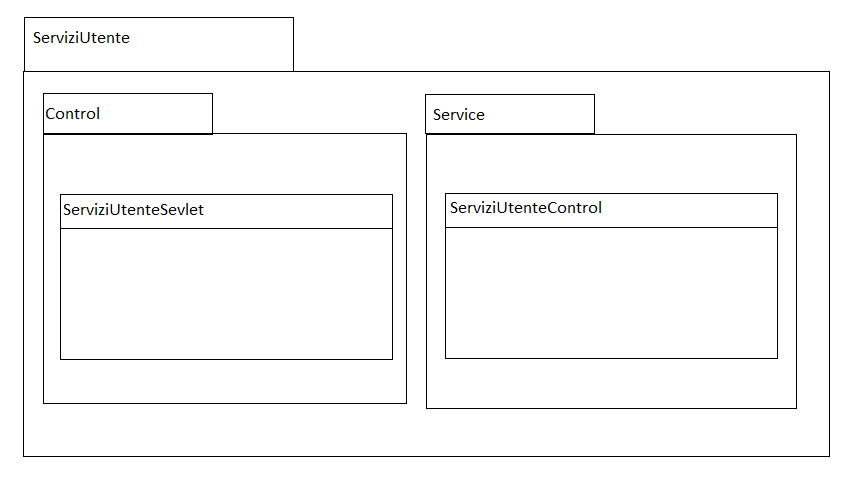
**Package Percorso Formativo**

****

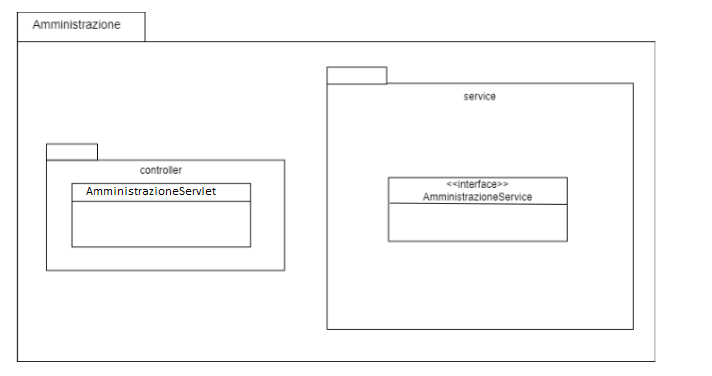
**Package Gestione Percorso Formativo**



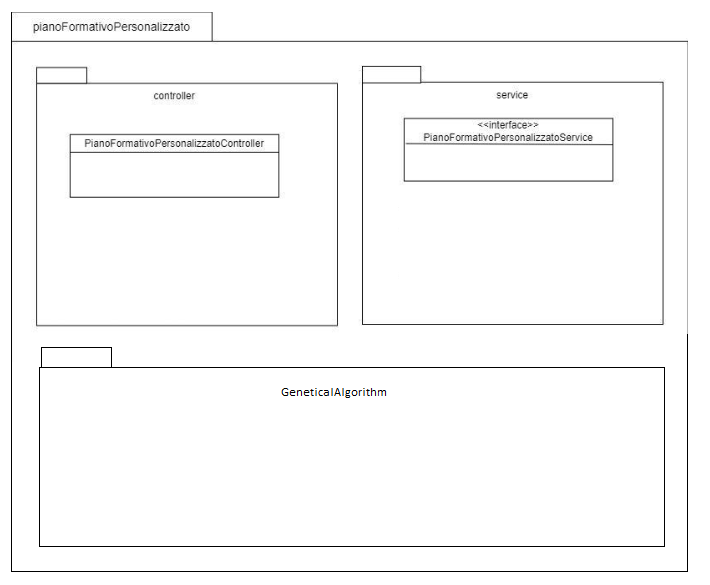
**Package Servizi Utente**

****

**Package Amministrazione**



**Package Piano formativo personalizzato**



3 Class Interfaces

1. AmministrazioneService

| **Nome classe** | AdminService |
| --- | --- |
| **Descrizione** | Questa classe offre operazioni per permettere all’amministratore di gestire la piattaforma |
| **Metodi** | +process(String serviceName, HttpServletRequest req, HttpServletResponse res):Action  +getErrorAction():Action  +getStudenti(HttpServletRequest req):boolean  +getFormatori(HttpServletRequest req):boolean  +getValutazioni(HttpServletRequest req):boolean  +getIscrizioni(HttpServletRequest req):boolean  +deleteUser(HttpServletRequest req):boolean  +createCategory(HttpServletRequest req):boolean |
| **Invariante di classe** | / |

1. AutenticazioneService

| **Nome classe** | AutenticazioneService |
| --- | --- |
| **Descrizione** | Questa classe offre operazioni per permettere ad un utente di effettuare il login e logout |
| **Metodi** | +process(String serviceName, HttpServletRequest req, HttpServletResponse res):Action  +getErrorAction():Action  +checkTrainerLogin(HttpServletRequest req):boolean  +checkStudentLogin(HttpServletRequest req):boolean  +checkAdminLogin(HttpServletRequest req):boolean  +canLogout(HttpServletRequest req):boolean  +disiscrizionePiattaforma(HttpServletRequest req):boolean |
| **Invariante di classe** | / |

1. ModificaProfiloService

| **Nome classe** | ModificaProfiloService |
| --- | --- |
| **Descrizione** | Questa classe offre operazioni per permettere ad uno studente oppure ad un formatore di modificare il proprio profilo. |
| **Metodi** | +process(String serviceName, HttpServletRequest req, HttpServletResponse res):Action  +getErrorAction():Action  +modificaProfilo(HttpServletRequest req):boolean  +modificaInteressi(HttpServletRequest req):boolean  +modificaPassword(HttpServletRequest req):boolean |
| **Invariante di classe** | / |

1. RegistrazioneService

| **Nome classe** | RegistrazioneService |
| --- | --- |
| **Descrizione** | Questa classe offre operazioni per gestire la registrazione utente |
| **Metodi** | +process(String serviceName, HttpServletRequest req, HttpServletResponse res):Action  +getErrorAction():Action  +executeTrainerFormRequest(HttpServletRequest request):boolean  +executeStudentFormRequest(HttpServletRequest request):boolean  +isCFContent(String cf):boolean  +isEmailContent(String email):boolean  +isEmailContentTrainer(String email):boolean |
| **Invariante di classe** | / |

1. GestionePFService

| **Nome classe** | GestionePFService |
| --- | --- |
| **Descrizione** | Questa classe offre operazioni per gestire i percorsi formativi |
| **Metodi** | +process(String serviceName, HttpServletRequest req, HttpServletResponse res):Action  +getErrorAction():Action  +creatorPercorso(HttpServletRequest request):boolean  +deletePercorso(HttpServletRequest req):boolean |
| **Invariante di classe** | / |

1. VisualizzaPercorsoService

| **Nome classe** | VisualizzaPercorsoService |
| --- | --- |
| **Descrizione** | Questa classe offre operazioni per visualizzare i percorsi formativi |
| **Metodi** | +process(String serviceName, HttpServletRequest req, HttpServletResponse res):Action  +getErrorAction():Action  +getPercorsiByFormatore(HttpServletRequest req):boolean  +getSchedaPercorso(HttpServletRequest req):boolean |
| **Invariante di classe** | / |

1. PercorsoFormativoService

| **Nome classe** | PercorsoFormativoService |
| --- | --- |
| **Descrizione** | Questa classe offre operazioni per interagire con i percorsi formativi |
| **Metodi** | +process(String serviceName, HttpServletRequest req, HttpServletResponse res):Action  +getErrorAction():Action  +iscrizione(HttpServletRequest req):boolean  +disiscrizionePF(HttpServletRequest req):boolean  +ricercaPercorsoFormativo (HttpServletRequest req):boolean  -RemoveDuplicate(ArrayList<PercorsoFormativoEntity> list):ArrayList<PercorsoFormativoEntity> |
| **Invariante di classe** | / |

1. UserService

| **Nome classe** | UserService |
| --- | --- |
| **Descrizione** | Questa classe offre operazioni per gli utenti |
| **Metodi** | +process(String serviceName, HttpServletRequest req, HttpServletResponse res):Action  +getErrorAction():Action  +createValutazione(HttpServletRequest req):boolean  +getAgenda(HttpServletRequest req):boolean  +getProfilo(HttpServletRequest req):boolean |
| **Invariante di classe** | / |

1. PianoFormativoService

| **Nome classe** | UserService |
| --- | --- |
| **Descrizione** | Questa classe offre operazioni per ottenere un piano formativo personalizzato. |
| **Metodi** | +process(String serviceName, HttpServletRequest req, HttpServletResponse res):Action  +getErrorAction():Action  +pianoFormativo(HttpServletRequest req):boolean  +ottieniPianoFormativo(HttpServletRequest req):boolean  +ottieniSoluzione(ArrayList<Stato> spazioStati,ArrayList<PreferenzaStudenteEntity> giorniLiberi, ArrayList<String> interessi, Map<Integer, String> categorie):Soluzione |
| **Invariante di classe** | / |

**4 Design patterns**

Nella presente sezione si andranno a descrivere e dettagliare i design patterns utilizzati nello sviluppo dell’applicativo FormAct. Per ogni pattern si darà:

• Una brevissima introduzione teorica.

• Il problema da risolvere all’interno di FormAct.

• Una brevissima spiegazione di come si è risolto il problema in FormAct.

**Façade Pattern:**

**Introduzione teorica:**

Il Façade è un design pattern strutturale che permette di accedere a sottoinsiemi più complessi attraverso una interfaccia semplificata. In questo modo, nascondiamo al sistema la complessità dei framework, dei set di classi e delle librerie utilizzate. Utilizzando un'unica interfaccia per accedere a tutto il sottosistema favoriamo un alto disaccoppiamento e rendiamo la piattaforma più manutenibile e più aggiornabile, poiché basterà cambiare l’implementazione dei metodi dell’interfaccia per implementare le modifiche.

**Problema da risolvere:**

Abbiamo bisogno di un modo per rendere il sistema più manutenibile favorendo un alto disaccoppiamento.

**Come abbiamo risolto il problema:**

Sfruttiamo il design pattern Façade per implementare tutta la logica di business e rendiamo i vari sottosistemi poco accoppiati.

Utilizzeremo il Façade per ogni sottosistema, implementando delle interfacce che sono utilizzate per accedere ai metodi interni dei sottosistemi.

**DAO Pattern (Data Access Object Pattern):**

**Introduzione teorica:**

Un DAO è un design pattern architetturale che offre un’interfaccia astratta per alcuni tipi di database. Mappando le chiamate dell’applicazione allo stato persistente, il DAO fornisce alcune operazioni specifiche sui dati senza esporre i dettagli del database. I DAO sono utilizzabili nella maggior parte dei linguaggi e la maggior parte dei software con bisogni di persistenza.

Usata principalmente in applicazioni web sia di tipo Java EE sia di tipo EJB, per stratificare e isolare l'accesso ad una tabella tramite query (poste all'interno dei metodi della classe) ovvero al *data layer* da parte della *business logic* creando un maggiore livello di astrazione ed una più facile manutenibilità. I metodi del DAO con le rispettive query dentro verranno così richiamati dalle classi della business logic. Il vantaggio relativo all'uso del DAO è dunque il mantenimento di una rigida separazione tra le componenti di un'applicazione, le quali potrebbero essere il "Modello" e il "Controllo" in un'applicazione basata sul paradigma MVC.

**Problema da risolvere:**

Abbiamo bisogno di un modo per interagire con il nostro database in modo rapido e sicuro con numerose query.

**Come abbiamo risolto il problema:**

Utilizziamo varie interfacce DAO all’interno del nostro sistema per interagire con il nostro database con l’utilizzo di varie query.

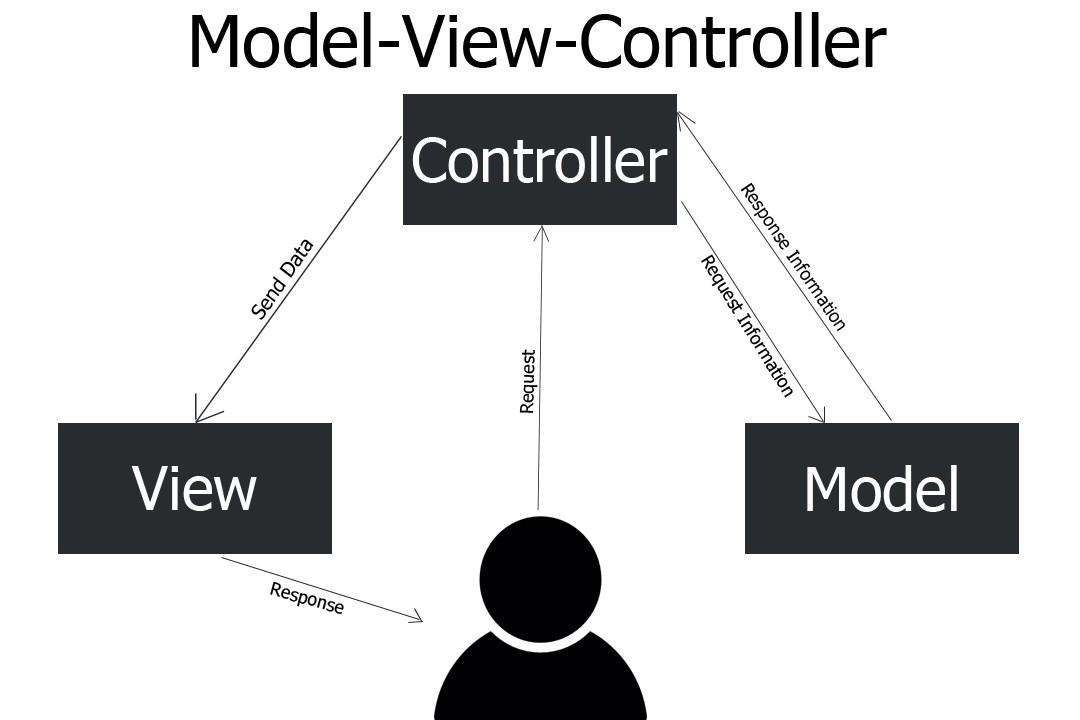
**MVC Pattern (Model-View-Control):**

**Introduzione teorica:**

Model-View-Controller (MVC) è un pattern utilizzato per dividere il codice in blocchi dalle funzionalità ben distinte.

lo schema che abbiamo identificato è esattamente quello proposto dal pattern MVC. In particolare:

* Model: contiene i metodi di accesso ai dati.
* View: si occupa di visualizzare i dati all'utente e gestisce l'interazione fra quest'ultimo e l'infrastruttura sottostante.
* Controller: riceve i comandi dell'utente attraverso il View e reagisce eseguendo delle operazioni che possono interessare il Model e che portano generalmente ad un cambiamento di stato del View.



**Problema da risolvere:**

Separare la logica di business, dalla visualizzazione in modo da migliorare il riuso e facilitare la manutenibilità

**Come abbiamo risolto il problema:**

Separando in package le varie classi, raggruppandole nei sottogruppi definiti prima.