ODD: Object Design Document

IFY-Internship For You

Riferimento	
Versione	1.0
Data	10/01/2020
Destinatario	Prof.ssa F. Ferrucci
Presentato da	Roberto Calabrese, Giusy Castaldo, Geremia Cavezza, Benedetta Coccaro, Simone Civale, Carmine Ferrara, Giacomo Izzo, Alessia Natale
Approvato da	Anna Belardo, Rosanna Coccaro

Revision History

Data	Versione	Descrizione	Autori
03/12/2019	0.1	Aggiunta Introduzione e Trade-offs	Natale Alessia
07/12/2019	0.1	Aggiunta Design Pattern	Ferrara Carmine, Izzo Giacomo
09/12/2019	0.1	Aggiunta Package e Organizzazione del codice in file	Castaldo Giusy, Cavezza Geremia
09/12/2019	0.1	Aggiunta Componenti off-the-shelf e Interfacce delle classi	Coccaro Benedetta, Calabrese Roberto
09/12/2019	0.1	Aggiunta Class Diagram	Civale Simone, Coccaro Benedetta, Calabrese Roberto
10/12/2019	0.1	Aggiunta Interface Documentation guidelines	Ferrara Carmine, Izzo Giacomo
11/12/2019	0.2	Modifica Interface Documentation guidelines	Ferrara Carmine, Izzo Giacomo
11/12/2019	0.2	Modifica Package	Castaldo Giusy, Cavezza Geremia
12/12/2019	0.2	Modifica Class Diagram	Civale Simone, Coccaro Benedetta, Calabrese Roberto
08/01/2020	0.3	Revisione Class Diagram	Civale Simone, Ferrara Carmine
10/01/2020	1.0	Revisione Object design trade-offs, Linee guida per la documentazione delle interfacce e Packages, revisione generale	Civale Simone, Natale Alessia

Sommario

1.Introduzione

L'Object Design Document consente di specificare in modo dettagliato le decisioni prese in fase di analisi e di design; in particolare verranno specificati i principali trade-offs, descritte le componenti off-the-shelfs utilizzate dal sistema, le linee guida per la documentazione delle interfacce e l'individuazione dei Design Patterns. Inoltre, verranno definiti i packages, le interfacce delle classi e i diagrammi delle classi che riguardano importanti decisioni implementative.

1.1 Object design trade-offs

- Sicurezza VS Prestazioni
  Ciascun sistema dovrebbe essere sia sicuro che performante in ugual modo ma, purtroppo, non è possibile avere ad alto livello entrambi e aumentare uno significa diminuire l'altro e viceversa. A tal proposito è importante raggiungere il giusto equilibrio e prendere in considerazione gli aspetti principali di entrambi per capire quale prediligere. Sebbene la sicurezza sia un fattore fondamentale e sebbene il nostro sistema gestirà comunque la privacy, l'autenticazione e l'autorizzazione al meglio, riteniamo dare più importanza a criteri relativi alla performance come throughput, tempo di compilazione o tempo di risposta. Tale scelta si basa sull'ottenere un sistema software che sia quanto più veloce e performante possibile, rispondendo in maniera efficiente alle esigenze degli utenti.

- Spazio di memoria VS Tempo di risposta
  Poiché il nostro sistema software nasce con l'obiettivo di velocizzare quello che è il processo di gestione tirocini rispetto al normale flusso di scambi di documenti cartacei e firme, esso si focalizzerà maggiormente sul tempo di risposta. Le operazioni relative all'accesso e alla gestione del database richiedono molto tempo soprattutto se le richieste di tali operazioni sono elevate e continue; a tal fine una soluzione da poter adottare è l'espansione della memoria per diminuire il tempo di risposta delle richieste e rendere più performanti le operazioni e le funzionalità del software.

- Alta disponibilità VS Tolleranza ai guasti
  Un sistema software che gestisce al meglio gli errori è sicuramente un buon software, ma questo significa spendere molte risorse e molto tempo. La differenza principale tra i due è che la tolleranza ai guasti cerca di minimizzare l'inattività, ma comporta anche un degrado delle prestazioni. Ciò che vogliamo è ottenere un sistema con alta disponibilità che garantisca una continuità dei servizi e delle funzionalità a lungo termine con un alto livello di prestazioni. L'alta disponibilità e la tolleranza ai guasti sono strettamente correlate in quanto avere un'alta disponibilità significa avere un sistema che sia quanto più privo di fallimenti e per ottenere ciò è necessaria molta manutenzione e molti test. Non è possibile effettuare una vera e propria scelta a riguardo, ma possiamo sicuramente orientarci verso una scelta che designi la realizzazione di un sistema software con un'alta disponibilità al fine di ottimizzare le prestazioni a lungo termine.

- Comprensibilità VS Costi
  Un aspetto principale che deve avere un sistema software è la comprensibilità del sistema stesso per rendere il codice comprensibile non solo a chi l'ha realizzato ma anche a coloro che sono esterni al progetto o che magari non sono stati coinvolti in una determinata parte del codice. Per rendere il codice quanto più comprensibile verranno utilizzati dei commenti che potranno permettere una maggiore leggibilità al fine di effettuare eventuali e future modifiche o per il mantenimento del sistema. Quindi daremo maggiore importanza alla comprensibilità anche se ciò comporta un aumento dei costi di sviluppo e di tempo.

1.2 Componenti off-the-shelf

Nella realizzazione del nostro sistema software andremo ad utilizzare componenti off-the-shelf già disponibili per facilitare lo sviluppo del progetto.

Per la progettazione del lato front-end utilizzeremo Bootstrap 4, che contiene una raccolta di strumenti liberi per la creazione di siti e applicazioni per il Web. Tale framework include esempi di progettazione basati su HTML5, CSS3 e alcune estensioni di JavaScript. Il sito web a cui faremo riferimento per la definizione del nostro template è ESSE3: https://esse3web.unisa.it/unisa/Home.do.

Il framework che utilizzeremo per implementare il codice del nostro software sarà Spring e l'interazione con esso avverrà tramite linguaggio di programmazione Java. Per lo sviluppo del codice verrà utilizzata la libreria jQuery e la tecnica di sviluppo AJAX.

Per la creazione del database ci serviremo del software MySQL che sarà connesso all'ambiente di sviluppo tramite il driver JDBC e usufruiremo del framework Java Persistence API per la gestione dei dati persistenti.

Inoltre, utilizzeremo anche la JavaServer Pages Standard Tag Library (JSTL), una raccolta di tag che gestiscono controllo di flusso condizionale ed iterativo, manipolazione di file XML, accesso a database tramite SQL ed altre comuni funzionalità. In questo modo tutte le scelte saranno gratis per non gravare sui costi.

1.3 Linee guida per la documentazione delle interfacce

Organizzazione del codice in file

Risulta necessario che il codice per l'implementazione del sistema "IFY Internship For You" venga organizzato in file.

Ciascuno dovrà:

- avere un nome che lo identifichi univocamente;
- essere sviluppato in base alla categoria di appartenenza;

- essere suddiviso in più file nel caso in cui la lunghezza diventi tale da rendere difficoltosa la lettura e la comprensione.

Inoltre, dovranno essere utilizzate cartelle per contenere i file delle librerie.
L'implementazione del sistema dovrà essere organizzata in package seguendo le linee guida dell'architettura individuata in fase di progettazione. Ogni nuovo documento creato, dovrà essere nominato in modo da rendere intuitivo l'obiettivo da realizzare e dovrà essere posizionato nel package di riferimento (così come definito nel successivo paragrafo di divisione in package).

Linee guida per classi e interfacce java e Spring

Per ogni sorgente di classe Java implementato, dovranno essere rispettati i seguenti parametri:

- Classi e interfacce:
  - La nomenclatura delle classi dovrà rispettare la notazione UpperCamelCase;
  - L'identificativo delle classi non dovrà essere ambiguo e congruo allo scopo della classe;
  - L'identificativo della classe dovrà essere formulato al singolare;
  - L'identificativo delle classi dovrà essere un nome (Es. Array) o al più un nome frasale (Es.ArrayList);
  - I nomi delle classi di test dovranno riportare il nome della classe testata, seguita dal suffisso "UT" nel caso di test di unità e "IT" nel caso di test di integrazione;
  - I nomi delle classi Repository dovranno riportare il nome dell'entità di riferimento, seguita dal suffisso "Repository";
  - I nomi delle classi Service dovranno riportare il nome dell'entità di riferimento, seguita dal suffisso "Service";
  - I nomi delle classi Controller dovranno riportare il nome del sottosistema di riferimento, seguito dal suffisso "Controller";

- Costanti:
  - I valori immutabili definiti in classi java dovranno essere definiti come "static final";
  - I nomi di costanti dovranno essere definiti in maiuscolo;
  - È ammessa la separazione tramite _ là dove necessario;

- Variabili d'istanza e variabili locali:
  - I nomi di parametri o variabili locali dovranno essere definiti secondo la notazione lowerCamelCase o al più separati dall'underscore;
  - Per ogni parametro d'istanza definito nella classe dovrà essere definito il livello di visibilità;

- Metodi:
  - Gli identificativi dei metodi dovranno seguire la notazione lowerCamelCase o al più separati dall'underscore;
  - Gli identificativi dei metodi dovranno iniziare con un verbo;

- Per ogni metodo di una classe sarà necessario specificare il livello di visibilità;
- Eventuali parametri nella firma del metodo, dovranno seguire le convenzioni adottate per le variabili d'istanza;

- Blocchi e indentazioni:
  - Il codice dovrà essere accuratamente indentato, tramite un Tab per ogni livello d'indentazione;
  - Le parentesi graffe per l'inizio di un nuovo blocco di codice dovranno essere riportate sulla stessa riga della definizione del blocco;
  - Le parentesi graffe di fine blocco dovranno essere allineate con l'inizio della definizione del blocco;

- Blocchi eccezionali:
  - Ogni blocco try catch definito all'interno di un metodo dovrà essere indentato in maniera corretta secondo le specifiche sopra riportate;
  - Le clausole catch dovranno essere riportate in maniera ordinata, dalla più specifica alla più generale, in caso di relazioni di estensione tra le tipologie di eccezioni coinvolte;
  - Ogni messaggio di errore gestito da stampare a video dovrà riportare un messaggio specificato dal programmatore che ne identifichi con chiarezza il tipo di errore e la provenienza;
  - Insiemi di operazioni comuni tra il blocco try e seguenti blocchi catch dovranno essere riportate nel blocco di chiusura finally;

- Annotazioni:
  - Le annotazioni previste per una classe o per un metodo dovranno apparire una per riga, subito dopo il blocco di documentazione;
  - Eventuali annotazioni prima di un parametro della classe dovranno essere definite una per linea, al disopra del parametro stesso, senza lasciare linee vuote;

- Documentazione:
  - Le classi e i metodi dovranno essere corredati da documentazione javadoc adeguata, in modo tale da rispettare tutte le specifiche prefissate in fase progettuale;
  - Nella documentazione della classe dovranno essere riportati l'autore e lo scopo della classe;
  - La definizione dei contratti specificata in fase di Object Design dovrà essere riporta anche nella documentazione javadoc tramite i costrutti per precondizioni, postcondizioni e invarianti della classe;
  - Per ogni metodo della classe, tramite appropriati costrutti Javadoc, dovranno essere specificati scopo del metodo, visibilità, parametri e tipo di ritorno;

- Commenti:

- Per ogni metodo della classe dovranno essere riportati blocchi di commenti che aiutino a capire il corretto flusso di operazioni del metodo;
- Sarà necessario chiarire tramite commenti operazioni innestate o eventuali blocchi poco chiari in prima lettura;
- La specifica di commenti nel codice dovrà essere conforme ai seguenti esempi:

```
/*
 * This is          // And so           /* Or you can
 * okay.            // is this.          * even do this. */
 */
```

Linee guida per pagine HTML 5

Per ogni documento HTML 5 creato, dovranno essere rispettati i seguenti parametri:

- Ogni documento creato dovrà riportare il tag <!doctype html> per identificare la tecnologia HTML 5 utilizzata;
- Ogni tag aperto nel corpo del documento HTML, a meno di tag singoli, dovrà riportare il rispettivo tag di chiusura;
- La struttura base di una pagina html (head, body), a meno di include JSP, dovrà essere rispettata;
- Ogni documento html, in particolare nel corpo del documento, dovrà essere indentato (preferibilmente tramite 1 tabulazione per livello) ad ogni definizione di un nuovo tag;
- Non potranno essere definiti più tag HTML sulla stessa riga;
- È preferibile definire tag in minuscolo.

Esempio accettabile

```
<!doctype html>
<html>
        <head>
                <title> Titolo </title>
        </head>
        <body>
                <ol>
                        <li>
                                primo
                        </li>
                </ol>
        </body>
</html>
```

Esempio non accettabile

```
<!doctype>
<html>
        <head>
                <title> Titolo </title>

        <body>
                <ol>
                        <li>primo</li>
        </ol>
        </body>
</html>
```

Linee guida per script JavaScript

- Ogni funzione Javascript dovrà essere riportata in un documento diverso dalla pagina html inclusa;
- Ogni funzione Javascript dovrà essere correttamente documentata in stile Javadoc;
- I nomi di funzioni, variabili e costanti dovranno seguire le stesse specifiche definite per i documenti Java;
- Ogni script dovrà essere incluso alla fine del body del relativo documento HTML.

Linee guida per Fogli di stile CSS 3

- Ogni regola CSS non inline dovrà essere riportata su un documento differente rispetto a quello della pagina html di riferimento, in modo da garantire un più facile riuso senza duplicazione;
- Ogni regola CSS dovrà iniziare all'inizio di una nuova linea, con la specifica dei selettori della regola;
- L'ultimo selettore di una regola CSS dovrà essere seguito dall'apertura del blocco con {.
- L'indentazione dovrà seguire i seguenti criteri:
    - Inizio di una nuova regola (#...{) e fine del blocco della regola (}) livello di indentazione 0;
    - Le proprietà di ogni regola CSS dovranno essere indentate di un Tab rispetto all'inizio del blocco, e dovranno essere riportate 1 per riga.

1.4 Design Pattern

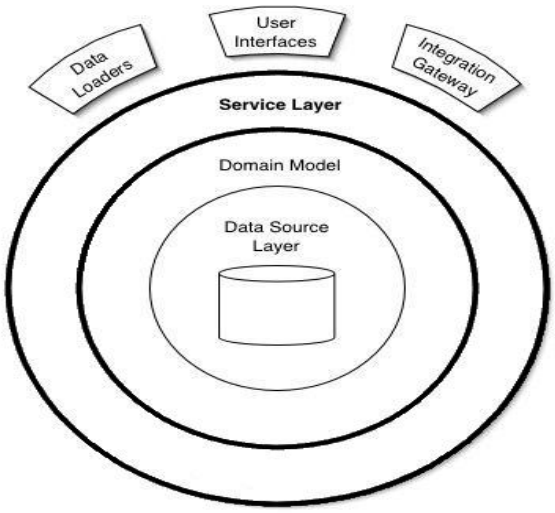
Service Layer

Problema

In sistemi complessi l'accesso alle informazioni è effettuato da una serie eterogenea di componenti con scopi differenti, ciononostante le richieste vengono effettuate attraverso interazioni simili con le sorgenti dei dati. Il design pattern Service Layer introduce un'interfaccia fra il modello dei dati e il resto del sistema, raggruppando le funzionalità per la visualizzazione e la modifica dei dati in un inventario di servizi comuni.

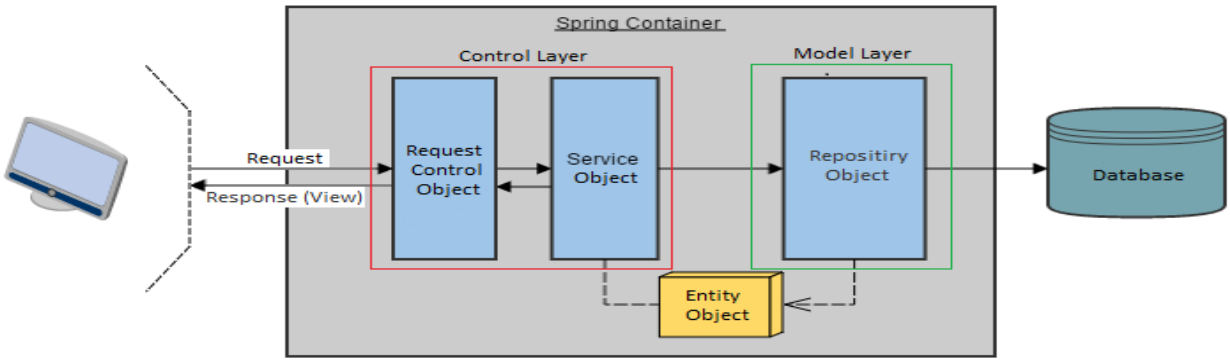
Soluzione

L'applicazione di questo pattern richiede la creazione di un inventario di servizi, con le relative funzionalità associate. Successivamente si raggruppano i servizi in layers secondo funzionalità affini. I servizi definiscono elementi di codice riusabile e componibile, semplificando il design del servizio e fornendo un'interfaccia adattabile a richieste differenti.



Conseguenze

- La suddivisione in servizi comuni facilita la riusabilità del codice;
- Riduzione del livello di accoppiamento tra layer Control e layer Model.



Repository

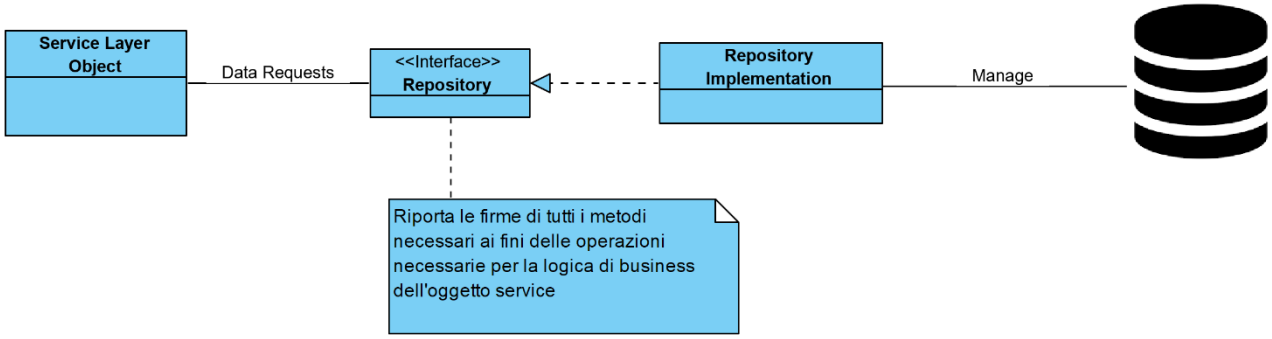
Problema

La logica di gestione dei dati persistenti in un'applicazione di tipo enterprise, ed in particolare quando si fa utilizzo del framework Spring abbinato alla tecnologia JPA (Java persistence api), è sfruttabile al massimo quando la logica di business non è legata da un forte accoppiamento con la logica di persistenza, inoltre, la tecnologia utilizzata nel nostro sistema, prevede che gran parte della logica di business venga demandata al container, e ciò risulterebbe complicato, laddove non ci sia un meccanismo di separazione netto tra queste due realtà.

Soluzione

Da un'attenta analisi della documentazione del framework spring, e dallo studio dei pattern di sviluppo di tipo enterprise, si è valutato di gestire la logica di persistenza, tramite l'utilizzo del pattern strutturale repository, che si divide in:

- Un'interfaccia pubblica per oggetti che necessitano di ricevere dati memorizzati su un qualsiasi meccanismo di gestione della persistenza (sia esso file o database), la quale riporta le firme dei metodi per la gestione dei dati persistenti;
- Una classe che implementa i metodi dell'interfaccia, che si occupa della gestione vera e propria.



Conseguenze

- Riduzione accoppiamento tra le classi di business logic e la base dati;
- Gestione delle componenti persistenti migliore e maggiormente mirata ai fini della logica di business;
- Operazioni di gestione della base dati ben definite e raccolte in un unico punto vincolante (dato l'utilizzo di un'unica interfaccia dedita a tale scopo);
- Maggior attinenza alla logica di gestione dei dati persistenti con il meccanismo Spring/JPA;
- Implementazione della logica di persistenza demandabile al container, dati i meccanismi implementativi utilizzati.

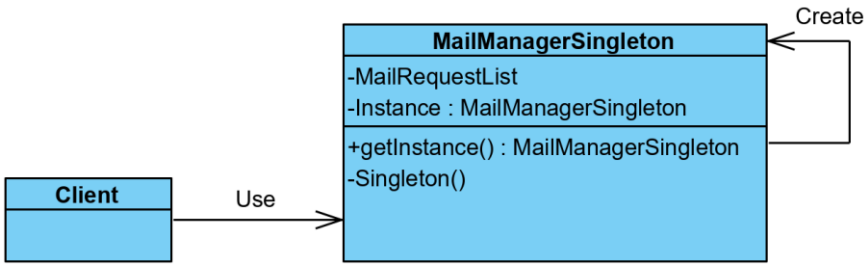
Singleton

Problema

La progettazione delle funzionalità in fase di analisi prevede l'invio di una notifica di avvenuto cambiamento di stato su più oggetti identificati nel dominio del problema, a tal proposito, si sceglie di optare nel dominio della soluzione per l'invio di notifiche e-mail. La gestione d'inoltro di e-mail è un servizio identificabile da una serie di processi comuni, quindi una gestione troppo specifica, in termini di oggetti del dominio, potrebbe aumentare il numero di classi con scopo comune in maniera errata.

Soluzione

A tal proposito si prevede di delegare la gestione delle e-mail di notifica ad un singleton object univoco per il sistema capace di memorizzare una serie di notifiche da inviare, e gestirle tramite l'invio di messaggi mirati (individuati dall'oggetto della richiesta e il relativo stato) agli utenti interessati.



Conseguenze

- Riduzione accoppiamento tra le classi di business logic e il server di inoltro e-mail;
- Riduzione di codice non prettamente coinvolto nella logica di business dei controller e relativo aumento di coesione;
- Singola istanza di un oggetto per un servizio specifico.

1.5 Definizioni, acronimi e abbreviazioni

Definizioni

- **Off-the-shelf**: servizi esterni al sistema di cui viene fatto utilizzo;
- **Bootstrap:** raccolta di strumenti liberi per la creazione di siti e applicazioni per il web;
- **JavaScript:** linguaggio di scripting orientato agli oggetti ed agli eventi, comunemente utilizzato nella programmazione web lato client per la creazione di effetti dinamici interattivi;
- **Spring:** framework open source per lo sviluppo di applicazioni su piattaforma Java;
- **JQuery:** libreria JavaScript per applicazioni web;
- **MySQL:** database open source che utilizza il linguaggio SQL;

Acronimi

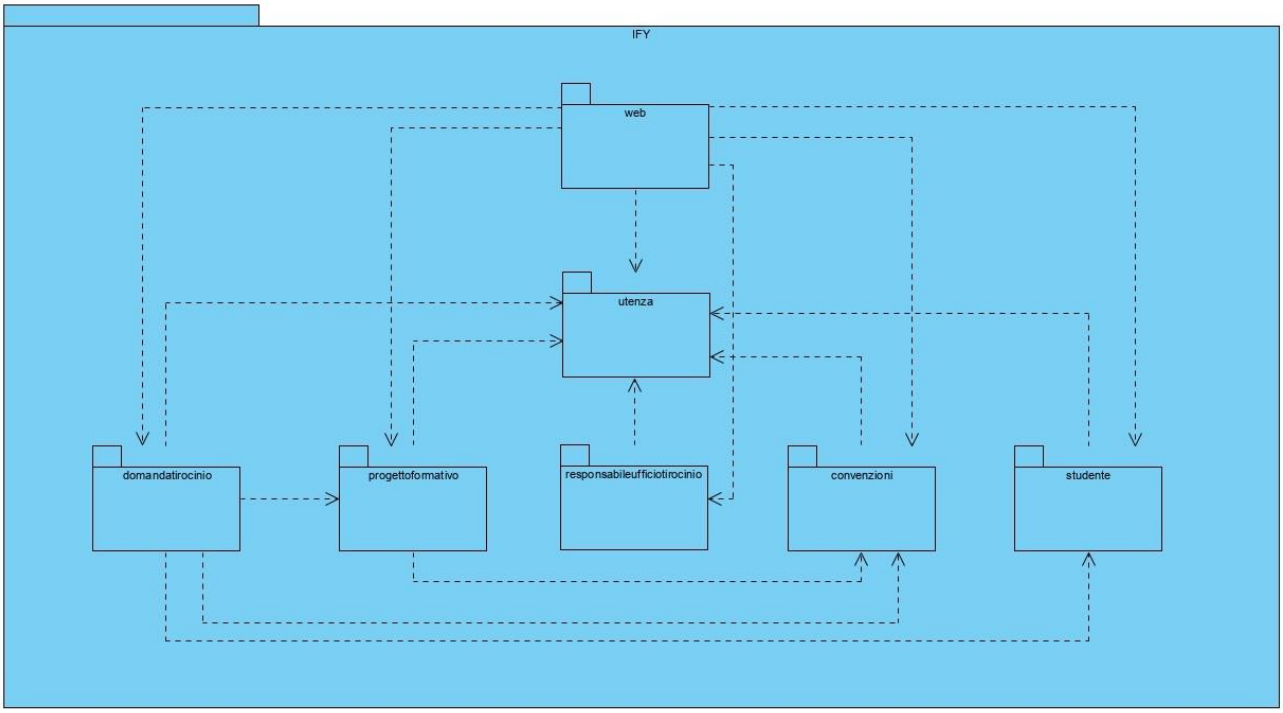
- **RAD:** Requirements Analysis Document;
- **SDD:** System Design Document;
- **ODD:** Object Design Document;
- **HTML:** Hyper Text Markup Language;
- **CSS:** Cascading Style Sheet;
- **XML:** eXtensible Markup Language;
- **AJAX:** Asynchronous JavaScript and XML;
- **JDBC:** Java Database Connectivity;
- **JSTL:** JavaServer Pages Standard Tag Library
- **JSP:** JavaServer Pages
- **SQL:** Structured Query Language

- **JPA:** Java Persistence API

2. Packages

La divisione in package proposta segue la divisione in sottosistemi individuata nella fase di system design, a tal proposito abbiamo individuato un package per sottosistema nel quale sono presenti le entità coinvolte nel sottosistema, e per ognuna di esse possiamo avere una classe repository (per la logica di persistenza) e una classe service (per la logica di business) del sottosistema.

Il sistema sarà distribuito in packages come segue:



- Il package "utenza" conterrà tutte le classi utili alla gestione dell'utenza.
- Il package "studente" conterrà tutte le classi utili alla gestione delle richieste di iscrizione.
- Il package "convenzione" conterrà tutte le classi utili alla gestione delle richieste di convenzionamento.
- Il package "progettoFormativo" conterrà tutte le classi utili alla gestione dei progetti formativi.
- Il package "domandaTirocinio" conterrà tutte le classi utili alla gestione delle domande di tirocinio.
- Il package "responsabileUfficioTirocinio" conterrà tutte le classi utili alla gestione dei dati del responsabile ufficio tirocini.
- Il package "web" conterrà i controller.

2.1 Package utenza

-utenza
Utente.java
UtenzaService.java
UtenteRepository.java
MailSingletonSender.java

Classe	Descrizione
Utente.java	Classe che rappresenta le informazioni di un utente.
UtenzaService.java	Fornisce tutti i servizi utili per la gestione dell'utenza.
UtenteRepository.java	Si occupa del recupero e del salvataggio dai dati relativi agli utenti.
MailSingletonSender.java	Si occupa dell'invio di notifica e-mail per ogni service che ne richiede l'utilizzo

2.2 Package studente

-studente
Studente.java
RichiestaIscrizione.java
StudenteService.java
StudenteRepository.java
RichiestaIscrizioneRepository.java

Classe	Descrizione
Studente.java	Classe che rappresenta le informazioni di uno studente.
RichiestaIscrizione.java	Classe che rappresenta le informazioni di una richiesta di iscrizione.
StudenteService.java	Fornisce tutti i servizi utili per la gestione delle iscrizioni.
StudenteRepository.java	Si occupa del recupero e del salvataggio dai dati relativi agli studenti.
RichiestaIscrizioneRepository.java	Si occupa del recupero e del salvataggio dai dati relativi alle richieste di iscrizione.

2.3 Package convenzione

-convenzione
Azienda.java
DelegatoAziendale.java
RichiestaConvenzionamento.java
ConvenzioneService.java
AziendaRepository.java
DelegatoAziendaleRepository.java
RichiestaConvenzionamentoRepository.java

Classe	Descrizione
Azienda.java	Classe che rappresenta le informazioni di una azienda.
DelegatoAziendale.java	Classe che rappresenta le informazioni di un delegato aziendale.
RichiestaConvenzionamento.java	Classe che rappresenta le informazioni di una richiesta di convenzionamento.
ConvenzioneService.java	Fornisce tutti i servizi utili per la gestione delle convenzioni.
AziendaRepository.java	Si occupa del recupero e del salvataggio dai dati relativi alle aziende.
DelegatoAziendaleRepository.java	Si occupa del recupero e del salvataggio dai dati relativi ai delegati aziendali.
RichiestaConvenzionamento Repository.java	Si occupa del recupero e del salvataggio dai dati relativi alle richieste di convenzionamento.

2.4 Package progettoFormativo

-progettoFormativo
ProgettoFormativo.java
ProgettoFormativoService.java
ProgettoFormativoRepository.java

Classe	Descrizione
ProgettoFormativo.java	Classe che rappresenta le informazioni di un progetto formativo.

ProgettoFormativoService.java	Fornisce tutti i servizi utili per la gestione dei progetti formativi.
ProgettoFormativoRepository.java	Si occupa del recupero e del salvataggio dai dati relativi ai progetti formativi.

2.5 Package domandaTirocinio

-domandaTirocinio
DomandaTirocinio.java
DomandaTirocinioService.java
DomandaTirocinioRepository.java

Classe	Descrizione
DomandaTirocinio.java	Classe che rappresenta le informazioni di una domanda di tirocinio.
DomandaTirocinioService.java	Fornisce tutti i servizi utili per la gestione delle domande di tirocinio.
DomandaTirocinioRepository.java	Si occupa del recupero e del salvataggio dai dati relativi alle domande di tirocinio.

2.6 Package responsabileUfficioTirocinio

-responsabileUfficioTirocinio
ResponsabileUfficioTirocini.java
ResponsabileUfficioTirociniRepository.java

Classe	Descrizione
ResponsabileUfficioTirocini.java	Classe che rappresenta le informazioni di un responsabile dell'ufficio tirocini.
ResponsabileUfficioTirocini Repository.java	Si occupa del recupero e del salvataggio dai dati relativi ai responsabili ufficio tirocini.

2.7 Package web

-web
UtenzaController.java
ConvenzioneController.java

DomandaTirocinioController.java
StudenteController.java
ProgettoFormativoController.java

Classe	Descrizione
UtenzaController.java	Si occupa di login e logout degli utenti.
ConvenzioneController.java	Si occupa della gestione delle convenzioni.
DomandaTirocinioController.java	Si occupa della gestione delle domande di tirocinio.
StudenteController.java	Si occupa della gestione delle richieste di iscrizione.
ProgettoFormativoController.java	Si occupa della gestione dei progetti formativi.

3. Interfacce delle classi

Nome classe	Utente
Descrizione	Questa classe rappresenta le informazioni relative ad un utente del sistema.
Pre-condizione	**context** Utente::addNotifica(n:Notifica) **pre:** n!=null
Post-condizione	**context** Utente::addNotifica(n:Notifica) **post:** self.notifiche.size==self@pre.notifiche.size+1
Invarianti	**context** Utente **inv:** checkEmail(email)==true

Nome classe	Responsabile Ufficio Tirocini
Descrizione	Questa classe rappresenta le informazioni relative al responsabile ufficio tirocini.
Pre-condizione	-
Post-condizione	-
Invarianti	**context** ResponsabileUfficioTirocini **inv:** self.utenteID!=null

Nome classe	Delegato Aziendale
Descrizione	Questa classe rappresenta le informazioni relative al delegato aziendale.
Pre-condizione	-

Post-condizione	-
Invarianti	**context** DelegatoAziendale **inv:** self.utenteID!=null

Nome classe	**Studente**
Descrizione	Questa **classe** rappresenta una specializzazione della classe Utente, contenente le informazioni specifiche per un utente immatricolato all'università presso il dipartimento.
Pre-condizione	**context** Studente::addDomandaTirocinio(d: DomandaTirocinio) **pre:** d!=null; **context** Studente::setMatricola(m: Matricola) **pre:** m.length() == 10; **context** Studente::setData_nascita (dt: data) **pre:** dt.get(YEAR) <= new Date().get(Year) - 19; **context** Studente:: addDomandaTirocinio (d: DomandaTirocinio) **pre:** d != null; **context** Studente:: removeDomandaTirocinio (d: DomandaTirocinio) **pre:** DomandeTirocinio.contains(d);
Post-condizione	**context** Studente::addDomandaTirocinio(d: DomandaTirocinio) **post:** DomandeTirocinio.size()==DomandeTirocinio.size() += 1; **context** Studente::removeDomandaTirocinio(d: DomandaTirocinio) **post:** DomandeTirocinio.size()==DomandeTirocinio.size() -= 1;
Invarianti	**context** Studente **inv:** UtenteID != null;

Nome classe	**Richiesta Iscrizione**
Descrizione	Questa classe rappresenta le informazioni e le funzionalità relative ad una richiesta di iscrizione.
Pre-condizione	**context** Richiesta Iscrizione:: setStato(s: Stato) **pre:** s="In Attesa" | | s="Accettata" | | s="Rifiutata";
Post-condizione	-
Invarianti	**Context** Richiesta Iscrizione **inv:** Stato!= null;

Nome classe	**Azienda**
Descrizione	Questa **classe** contiene le informazioni inerenti ad un'azienda convenzionata con il dipartimento.

Pre-condizione	**context** Azienda::setPIVA (piva: Partita Iva) **pre:** p.length() == 11; **context** Azienda:: addDomandaTirocinio (d: DomandaTirocinio) **pre:** d != null; **context** Azienda:: removeDomandaTirocinio (d: DomandaTirocinio) **pre:** DomandeTirocinio.contains(d); **context** Azienda:: addProgettoFormativo (p: ProgettoFormativo) **pre:** p != null; **context** Azienda:: removeProgettoFormativo (p: ProgettoFormativo) **pre:** ProgettiFormativi.contains(p);
Post-condizione	**context** Azienda::addDomandaTirocinio(d: DomandaTirocinio) **post:** DomandeTirocinio.size()==DomandeTirocinio.size() += 1; **context** Azienda::removeDomandaTirocinio(d: DomandaTirocinio) **post:** DomandeTirocinio.size()==DomandeTirocinio.size() -= 1; **context** Azienda::addProgettoFormativo(p: ProgettoFormativo) **post:** ProgettiFormativi.size()==ProgettiFormativi.size() += 1; **context** Azienda::removeProgettoFormativo(p: ProgettoFormativo) **post:** ProgettiFormativi.size()==ProgettiFormativi.size() -= 1;
Invarianti	**context** Azienda **inv:** PIVA != null;

Nome classe	**Domanda Tirocinio**
Descrizione	Questa **classe** contiene le informazioni inerenti ad una singola domanda di tirocinio inviata da uno studente.
Pre-condizione	**context** DomandaTirocinio::setDataInizio (d: data Inizio) **pre:** d > new Date(); **context** DomandaTirocinio::setDataFine (df: data fine) **pre:** df > getDataInizio(); **context** DomandaTirocinio::setCFU (c: #CFU) **pre:** 6 <= c <= 12 **context** DomandaTirocinio::setIDProgetto(id: IDProgetto) **pre:** azienda.ProgettiFormativi -> exist(p: ProgettoFormativo | p.getId_Progetto() == id) **context** DomandaTirocinio::setAziendaPIVA (api: Partita Iva Azienda) **pre:** exist(a: Azienda | a.getPIVA () == api)

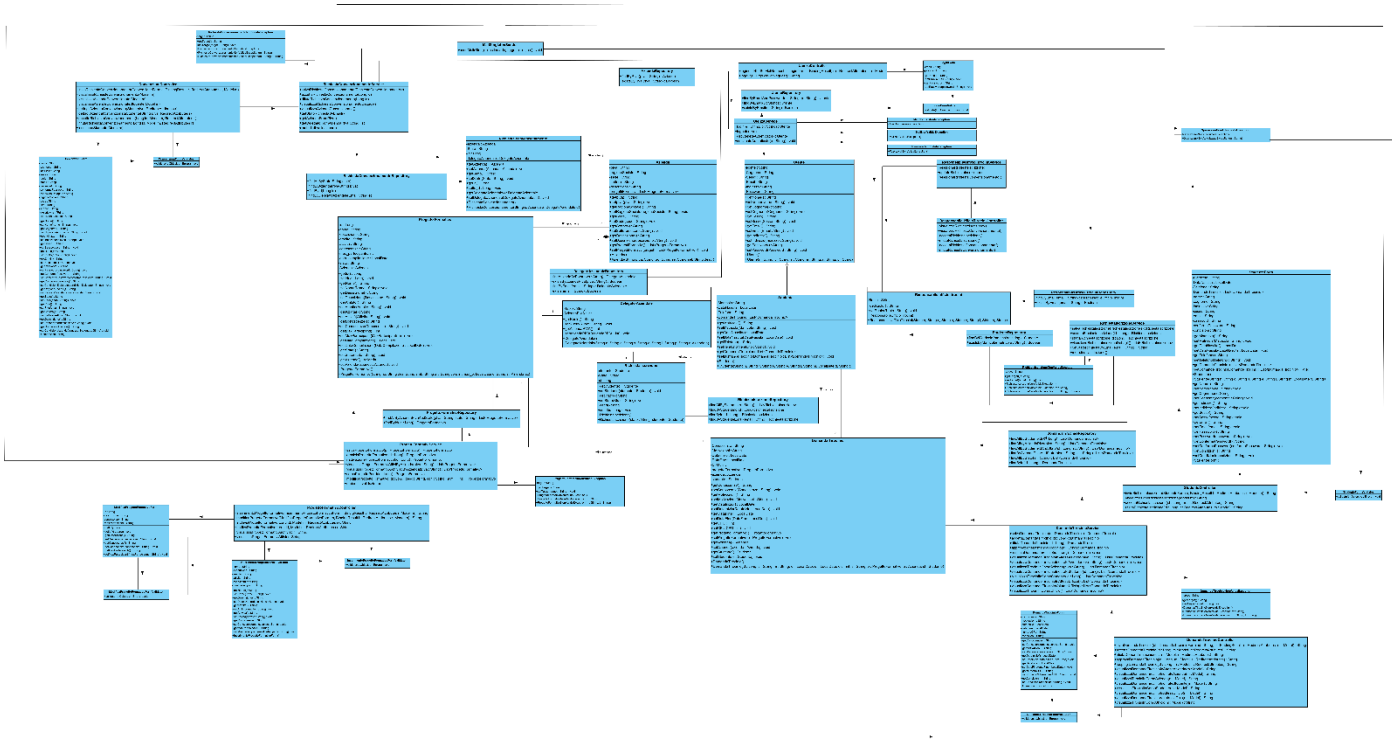
	context DomandaTirocinio::setStato(s:Stato) :s=="in attesa"|| s == "accettata"||s == "rifiutata"||s=="approvata"||s=="respinta"
Post-condizione	-
Invarianti	**context** DomandaTirocinio **inv:**id_DomandaTirocinio != null; **context** DomandaTirocinio **inv:**idProgettoFormativo != null; **context** DomandaTirocinio **inv:**AziendaPIVA!= null; **context** DomandaTirocinio **inv:**Data Inizio!= null; **context** DomandaTirocinio **inv:** Data Fine!= null;

Nome classe	**Richiesta Convenzionamento**
Descrizione	Questa classe rappresenta le informazioni e le funzionalità relative ad una richiesta di convenzionamento.
Pre-condizione	**Context** Richiesta Convenzionamento:: setIDRichiesta(ir: IDRichiesta) **pre:** ir!=null; **Context** Richiesta Convenzionamento:: setStato(s: Stato) **pre:** s="In Attesa"|| s="Accettata" || s="Rifiutata"; **Context** Richiesta Convenzionamento:: setAziendaPIVA(piva: Partita Iva) **pre:** p.length() == 11;
Post-condizione	-
Invarianti	**Context** Richiesta Convenzionamento **inv:** RichiestaID != null; **Context** Richiesta Convenzionamento **inv:** PIVA != null;

Nome classe	**Progetto Formativo**
Descrizione	Questa classe rappresenta le informazioni e le funzionalità relative ad un progetto formativo.
Pre-condizione	**Context** Progetto Formativo:: setMaxPartecipanti(m: MaxPartecipanti) **pre:** m!=null && m>0&&m<=999; **Context** Progetto Formativo:: setDataCompilazione(d: Data) **pre:** d!=null && d==DataCorrente; **Context** Progetto Formativo:: setAziendaPIVA(piva: Partita Iva) **pre:** p.length() == 11; **Context** Progetto Formativo: addDomandaTirocinio(d: Domanda) **pre:** d!=null; **Context** Progetto Formativo:: removeDomandaTirocinio(d: Domanda) **pre:** d!=null && DomandeTirocinio.contains(d)==true;
Post-condizione	**Context** Progetto Formativo:: addDomandaTirocinio(d: Domanda) **post:** List.size == List.size+1 && d⊂List; **Context** Progetto Formativo:: removeDomandaTirocinio(d: Domanda) **post:** List.size == List.size-1 && d⊄List;

Invarianti	-

4. Diagramma delle classi



Nota: si allega il documento pdf del class diagram