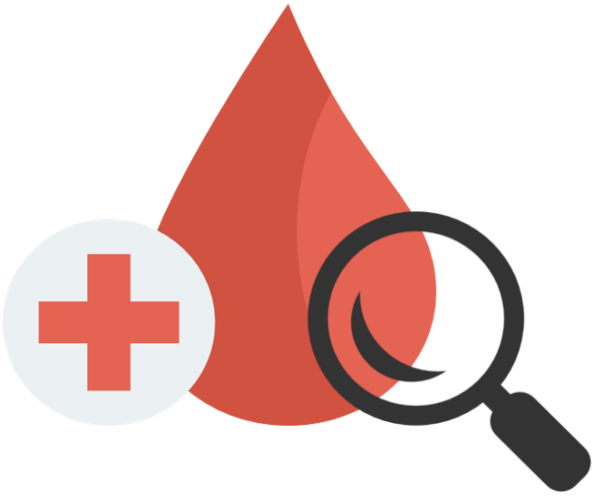
Laurea Magistrale in informatica-Università di Salerno
Corso di *Gestione dei Progetti Software*- Prof.ssa F.Ferrucci



Object Design Document

Digital Donation

Riferimento	
Versione	1.1
Data	2021
Destinatario	Prof.ssa F. Ferrucci
Presentato da	Annamaria Basile, Angela De Martino, Elpidio Mazza, Kevin Pacifico, Mattia Sapere, Fabio Siepe, Marika Spagna Zito
Approvato da	Francesco Abate, Carmine Ferrara

Revision History

Data	Versione	Descrizione	Autori
07/12/21	0.1	Definizione dei trade off di sistema	[Tutti]
07/12/21	0.1	Linee guida per l'implementazione	[Tutti]
07/12/21	0.1	Decisioni preliminari sui design pattern	[Tutti]
24/12/21	0.1	Specifica dei Design Pattern	[Tutti]
24/12/21	0.1	Divisione in Packages	[Tutti]
24/12/21	0.1	Specifica delle interfacce delle classi	[Tutti]
24/12/21	0.1	Definizione del Glossario	[Tutti]
24/12/21	0.1	Aggiornamento del sommario	[Tutti]
24/12/21	0.1	Integrazione e revisione del documento di Object Design	[Tutti]
24/12/21	1.0	Approvazione del documento	FA,CF
11/01/22	1.1	Modifiche Class Diagram	KP, MS
11/01/22	1.1	Modifiche Class Interface	MSZ, MS, KP
11/01/22	1.1	Approvazione del documento	FA, CF

Sommario

Revision History 2
1. Introduzione 4
1.1 Object design goals 4
1.2 Object trade-off 4
1.3 Componenti off-the-shelf 5
1.4 Linee Guida per la Documentazione delle Interfacce 6
1.4.1 Nomenclatura delle componenti 6
1.4.2 Nomi delle classi 6
1.4.3 Nomi dei metodi 6
1.4.4 Nomi delle eccezioni 6
1.4.4 Organizzazione delle componenti 7
1.4.5 Organizzazione del codice 7
1.4.6 Altre linee guida 9
1.5 Definizioni, acronimi e abbreviazioni 11
1.6 Riferimenti 12
2. Package 13
3. Class interfaces 15
3.1 Utente Management 15
3.2 Organizzazione Sedute Management 16
3.3 Tesserino Management 20
3.4 Gestione Sedute Management 22
4. Class diagram 24
5. Design Patterns 25
5.1 Singleton 25
5.2 Facade 26
5.3 Service Layer 27
5.4 Repository 28
6.Glossario 29

1. Introduzione

Dopo aver stilato il documento di Requirements Analysis e il documento di System Design in cui vi è una descrizione sommaria di ciò che sarà il nostro sistema, definendo i nostri obiettivi ma tralasciando gli aspetti implementativi, andiamo ora a stilare il documento di Object Design che ha come obiettivo quello di produrre un modello che sia in grado di integrare in modo coerente e preciso tutte le funzionalità individuate nelle fasi precedenti. Inoltre, verranno definiti i packages, le interfacce delle classi e i diagrammi delle classi che riguardano importanti decisioni implementative.

1.1 Object design goals

Modularità

Il sistema Digital Donation deve basarsi su alta coesione e basso accoppiamento tra le sue componenti, inoltre si farà largo uso dei concetti di ereditarietà e i design patterns.

Robustezza

Il sistema dovrà garantire robustezza reagendo correttamente a situazioni impreviste usufruendo di opportuni controlli degli errori e gestione delle eccezioni.

Incapsulamento

Il sistema garantisce l'occultamento dei dettagli implementativi delle classi grazie alle interfacce in modo da rendere possibile l'utilizzo di funzionalità offerte da diversi componenti o layer sottoforma di blackbox.

Riusabilità

Nell'implementazione del sistema verrà utilizzato un template front-end già realizzato con l'ausilio della libreria Bootstrap.

1.2 Object trade-off

Prestazioni vs Costi

In quanto il nostro progetto è soggetto ad un budget da rispettare, per poter mantenere prestazioni elevate, in determinate funzionalità verranno utilizzati dei template esterni: in particolare il framework Bootstrap; una di queste funzionalità, per la quale lo utilizzeremo sarà un template di sito web lato front-end.
Questo approccio non solo impatterebbe positivamente dal punto di vista delle risorse economiche (costi di sviluppo) ma anche dal punto di vista dell'effort impiegato da parte degli sviluppatori (costi di tempo).

Sicurezza vs Efficienza

La sicurezza, come descritto nei requisiti funzionali del Requirements Analysis, rappresenta uno degli aspetti fondamentali del sistema. Tuttavia, però c'è bisogno di arrivare ad un compromesso: riteniamo dare importanza maggiore a criteri relativi alla performance come throughput, tempo di risposta o tempo di compilazione. Ci limiteremo ad implementare sistemi di sicurezza basati su username e password degli utenti così da avere un sistema software che sia più veloce e performante possibile.

Spazio di memoria vs tempo di risposta
Tra spazio di memoria e tempo di risposta preferiamo focalizzarci sul secondo requisito, la motivazione

si trova nel fondamento del nostro progetto che si prefigge l'obiettivo di una sostituzione parziale dei documenti cartacei, al fine di avere una gestione migliore e non avere inconsistenze.

Interfaccia vs Usabilità

L'interfaccia grafica è stata progettata in maniera da risultare semplice, chiara e concisa così da garantire una maggiore semplicità di navigazione da parte dell'utente. Come scelta, dunque, abbiamo preferito l'usabilità perché è d'essenziale importanza rispetto ad una interfaccia articolata che non gioverebbe ad un'immediata comprensione per l'utilizzatore del sistema.

1.3 Componenti off-the-shelf

Per il progetto software che si vuole realizzare, facciamo uso di componenti off-the-shelf che sono componenti software disponibili su mercato per facilitare la creazione del progetto. Il framework che andremo ad utilizzare è Bootstrap 5, che è un framework open source che contiene una raccolta di strumenti liberi per la creazione di siti ed applicazioni web. Essa contiene modelli di progettazione basati su HTML e CSS, sia per la tipografia, che per le varie componenti dell'interfaccia, come moduli, bottoni e navigazione, e altri componenti dell'interfaccia, così come alcune estensioni opzionali di JavaScript.

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.

1.4 Linee Guida per la Documentazione delle Interfacce

È richiesto agli sviluppatori di seguire le seguenti linee guida al fine di essere consistenti nell'intero progetto e facilitare la comprensione delle funzionalità di ogni componente, in modo da facilitare l'organizzazione del team di implementazione suddividendo in maniera adeguata i package dell'architettura del software.

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

checkstyle di Google: https://checkstyle.sourceforge.io/google_style.html

1.4.1 Nomenclatura delle componenti

- Ogni nome di un file/componente dovrà avere un nome significativo.
- Ogni componente dovrà terminare con il nome ."estensione" che rappresenta il tipo del file.
- Ogni file dovrà avere un nome auto esplicativo in modo da consentire una facile individuazione della componente anche se generale.
- Ogni metodo e ogni file devono essere preceduti da un commento, o più precisamente da una documentazione che riporti l'obiettivo che si vuole e deve raggiungere con il nome/i dell'autore/i. Inoltre, bisogna commentare, giustificare delle decisioni particolari o dei calcoli.

1.4.2 Nomi delle classi

- Ogni classe deve avere nome in CamelCase
- Ogni classe deve avere nome singolare
- Ogni classe che modella un'entità deve avere per nome un sostantivo che possa associarla alla corrispondente entità di dominio
- Ogni classe che realizza la logica di business per una determinata entità deve avere nome composto da quello del pacchetto per cui espone servizi seguito da "Service"
- Ogni classe che modella una collezione in memoria per una determinata classe di entità deve avere nome composto da quello dell'entità su cui opera seguito da "Repository"
- Ogni classe che realizza un form deve avere nome composto dal sostantivo che descrive il form seguito dal suffisso "Form"
- Ogni classe che realizza un servizio offerto via web deve avere nome composto dal nome del pacchetto per cui espone servizi seguito dal suffisso "Controller"

1.4.3 Nomi dei metodi

- Ogni metodo deve avere nome in lowerCamelCase.
- Ogni metodo è rappresentato da un nome significativo.

1.4.4 Nomi delle eccezioni

- Ogni eccezione deve avere nome esplicativo del problema segnalato.

1.4.4 Organizzazione delle componenti

- Tutte le classi che realizzano un sottosistema devono essere racchiuse nello stesso pacchetto Java.
- Tutte le componenti che realizzano l'interfaccia grafica devono essere collocate in una directory con nome "view".
- Tutte le risorse statiche ovvero fogli di stile, script e immagini devono essere collocate nella stessa cartella suddivise in base alla tipologia.

1.4.5 Organizzazione del codice

- Il codice Java deve essere indentato in maniera appropriata (tramite tabulazione corrispondente a 2 spazi) e l'apertura di un blocco di codice deve avvenire nella stessa riga in cui è specificato il nome della classe o la firma del metodo che quel blocco definisce.
- Ogni classe dovrà avere una documentazione che ne espliciti le sue funzionalità.
- Il codice HTML dev'essere indentato in maniera appropriata (tramite tabulazione corrispondente a 2 spazi) e gli attributi devono essere indicati in minuscolo.
- Il codice deve rispettare la formattazione standard di scrittura, spazio dopo la virgola

```
b = 20;
Int a, b = 20;
```

- L'indentazione deve essere effettuata con un TAB e qualunque sia il linguaggio usato per la produzione di codice, ogni istruzione deve essere opportunamente indentata.
  Questa pratica deve essere usata soprattutto per le istruzioni FOR, IF.
  È buona pratica scendere di livello.

```
<!DOCTYPE html>
<html>
  <head>
    <title>La sezione 'body' della pagina HTML</title>
  </head>
  <body>
    <p>
      Questa e' la sezione body della pagina HTML,
      delimitata dai tag di apertura e chiusura omonimi.
    </p>
    <p>
      Quello sopra e' un paragrafo,
      e io sono un secondo paragrafo.
    </p>
  </body>
</html>
```

- Inizializzare le variabili locali nel punto in cui sono state dichiarate a meno che il suo valore iniziale non dipenda da un calcolo che occorre eseguire prima.

- A prescindere dalle istruzioni che seguono un IF, è necessario, laddove ci fosse anche una sola istruzione, riportare il blocco di istruzioni tra parentesi graffe. Ogni tag di apertura deve essere necessariamente seguito dall'apposito tag di chiusura (eccetto i tag self-closing).

- Una convenzione importante, per quanto riguarda l'inserimento di numeri o di valori costanti, è quella di non usare una codifica fissa (hard coding) che è fortemente sconsigliata ma di associare sempre il valore ad una variabile o semplicemente definire una macro che può essere richiamata da eventi ed essere parametrizzata. In questo modo si facilita la modifica, sostituendo solo il valore della variabile o macro, in un unico posto.

- Quando un'espressione supera la lunghezza della linea, occorre spezzarla secondo i seguenti principi generali:

  Interrompere la linea dopo una virgola;
  Interrompere la linea prima di un operatore;
  Preferire interruzioni di alto livello rispetto ad interruzioni di basso livello (interrompere laddove non si interrompe un discorso logico, discorso valido soprattutto per le formule es. (3+4) * 2 interrompere prima della moltiplicazione senza spezzare gli operandi in parentesi);
  Allineare la nuova linea con l'inizio dell'espressione nella linea precedente;
  Se le regole precedenti rendono il codice più confuso o il codice è troppo spostato verso il margine destro, utilizzare solo otto spazi di indentazione.

```
public static class HelloWorldClient {
    public static void main(String[] args)
            throws NamingException {
        Context ctx;
        ctx = new InitialContext();
        HelloBeanRemote helloBean = (HelloBeanRemote)
                ctx.lookup( name: "java:global/HelloBean!hello." +
                        "HelloRemote");
        System.out.println("Ora invoco....");
        //..........//
    }
}
```

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

- 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;

1.4.6 Altre linee guida

- Per avere una documentazione corretta e completa adottiamo l'applicativo Javadoc, più nel dettaglio per ogni classe deve essere definito l'autore e la descrizione.

```
/**
 * Il metodo si occupa di settare il titolo di un libro
 * @param title
 * @exception LibroNonPresenteException
 * @return true
 */
public boolean setTitle(String title) throws LibroNonPresenteException {
    this.title = title;
    return true;
}
```

  Per ogni metodo la descrizione, parametri in entrata e uscita ed eccezioni.

```java
/**
 * La classe modella l'entità persisitente libro
 * @author Marika Spagna Zito
 *
 */
public class Libro {
    public String title;
    /**
     * Il metodo si occupa di settare il titolo di un libro
     *
     * @param title
     * @return true
     * @throws LibroNonPresenteException
     */
    public boolean setTitle(String title)
            throws LibroNonPresenteException {
        this.title = title;
        return true;
    }

}
```

- Per adattare le linee guida selezionate nell'IDE di sviluppo il team prevede di utilizzare il plug-in checkstyle che verrà istruito in modo tale da comprendere tutte, o la maggior parte, delle regole previste in questo capitolo. Come base da fornire al plug-in verrà utilizzato l'insieme di regole fornito da Google all'interno della sua configurazione iniziale per il plug-in.

1.5 Definizioni, acronimi e abbreviazioni

Bootstrap: è un framework open source che contiene una raccolta di strumenti liberi per la creazione di siti ed applicazioni web.

Camel case: "notazione a cammello", consiste nello scrivere parole composte unendo tutte le lettere ma lasciando le iniziali maiuscole.

Design pattern: soluzioni progettuali comuni a problemi che si presentano frequentemente.

LowerCamelCase: convenzione per scrivere i nomi unendo tutte le lettere dove però la prima lettera inizia con una minuscola.

Javadoc: è un applicativo utilizzato per generare automaticamente documentazione al fine di renderla facilmente accessibile e leggibile.

Package: è un meccanismo per raggruppare insiemi logici di classi, interfacce o file al fine di dividere i diversi contesti.

UI: il termine UI rappresenta l'acronimo di "interfaccia utente" (dall'inglese User Interface).
È un'interfaccia uomo-macchina, ovvero ciò che si frappone tra una macchina e un utente, consentendone l'interazione reciproca.

1.6 Riferimenti

- Benrd Bruegge- Allen H. Dutoit, Object-Oriented Software Engineering: Using UML
- RAD Requirement Analysis Document Digital Donation V_1.2
- SDD System Design Document Digital Donation V_1.2

2. Package

In questa sezione mostriamo la suddivisione in package del sistema, come abbiamo previsto nei documenti di Requirement elycitation e System Design.
La scelta di tale suddivisione è motivata dalle scelte architetturali e riprende l'organizzazione in directory definita da Maven.

- .idea intelliJ root folder, contenente file di configurazione del progetto
- .mvn, contiene tutti i file di configurazione per Maven
- src, contiene tutti i file sorgente
    - main
        - java, contiene le classi Java relative alle componenti Control e Model
        - resources, contiene i file relativi alle componenti View
            - resources, contiene i fogli di stile CSS e gli script JS
            - view, contiene i file HTML renderizzati da bootstrap
    - test, contiene tutto il necessario per il testing
        - java, contiene le classi Java per l'implementazione del testing
        - target, contiene tutti i file prodotti dal sistema di build di Maven

Package DigitalDonation

Nella presente sezione si mostra la struttura del package principale di DigitalDonation.
La struttura generale è stata ottenuta a partire da quattro principali scelte:

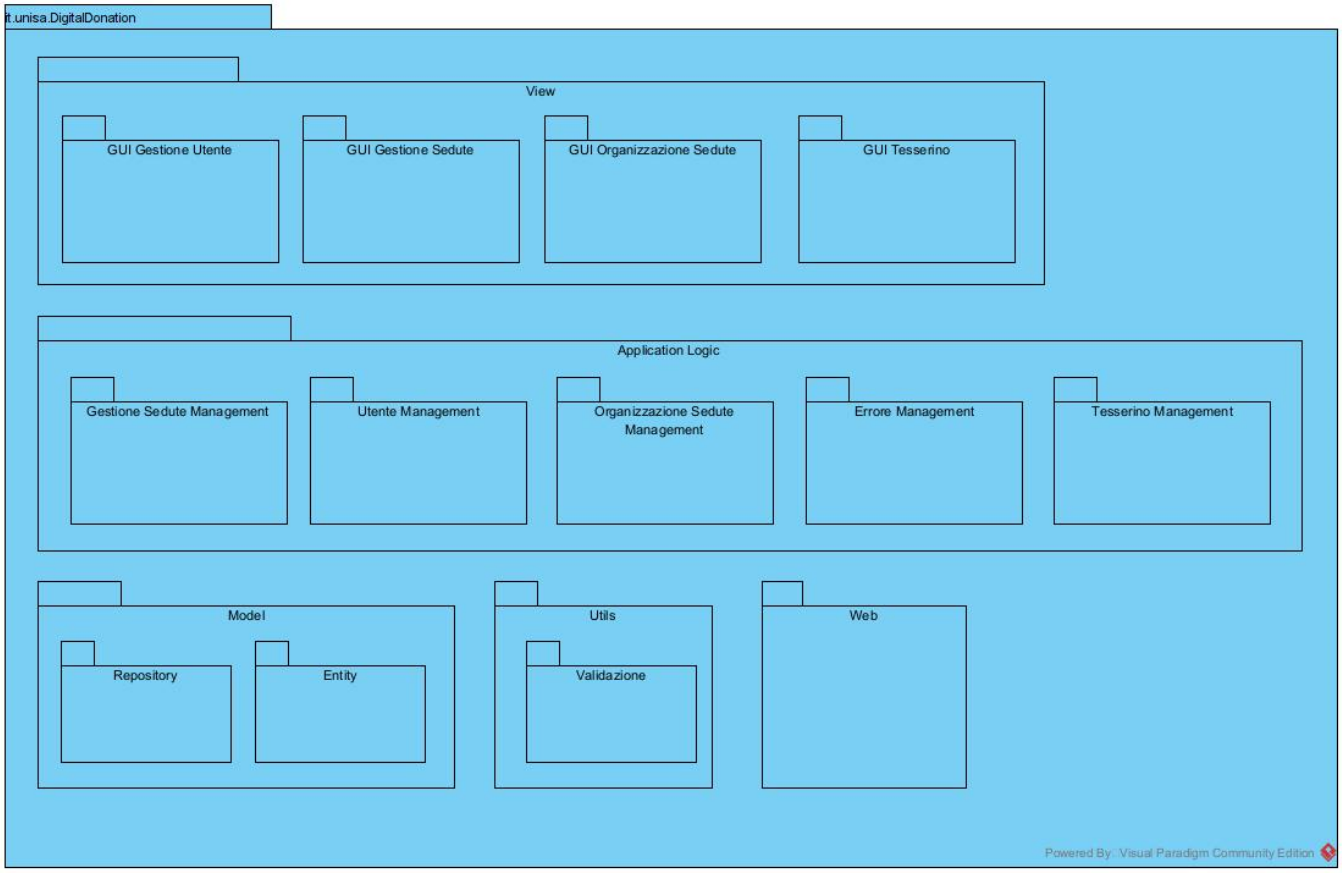
1. Creare un package per la logica di presentazione chiamato "View" ed un package per la logica di applicazione chiamato "Application Logic". All'interno di questi ultimi creare dei package per ogni sottosistema. I quali sono:
    - GUI Gestione Utente: pacchetto che contiene tutte le pagine jsp generiche di utenza
    - GUI Gestione Sedute: pacchetto che contiene tutte le pagine jsp generiche relative alle sedute
    - GUI Organizzazione Sedute: pacchetto che contiene tutte le pagine jsp generiche relative all'organizzazione delle sedute
    - GUI Tesserino: pacchetto che contiene tutte le pagine jsp generiche relative al tesserino
    - Gestione Sedute Managment: pacchetto che contiene classi service per la gestione del tesserino
    - Utente Managment: pacchetto che contiene classi service per la gestione del tesserino
    - Errore Managment: pacchetto che contiene classi service per la gestione degli errori
    - Tesserino Managment: pacchetto che contiene classi service per la gestione del tesserino
2. Creare un package "Web" in cui inserire tutte le classi controller come specificato nel documento di Test Plan.
3. Creare un package separato per le classi del model, contenente le classi entity e i DAO per l'accesso al DB.

4. Creare un package chiamato "Utils" in cui inserire eventuali classi di utilità per il sistema e usabili da più sottosistemi.

Il livello della logica applicativa interagisce con:

- Il package model accedendo alle classi entity e repository.
- Il package Web che contiene riferimenti ai pacchetti contenuti nella view.
- Il package utils e tutte le sue classi al suo interno.

Per ciò che concerne la dipendenza tra i packages, la suddivisione precedentemente illustrata ha portato alla creazione di una relazione tra il package model e tutti gli altri package del sistema



3. Class interfaces

Abbiamo previsto le interfacce solamente per le classi che implementano la logica di business nel sistema.

3.1 Utente Management

Nome classe	UtenteService
Descrizione	Questa classe permette di gestire le operazioni inerenti ad un Utente.
Metodi	+login(String email, String password): Utente +logout(Utente utente): void
Invariante di classe	/

Nome Metodo	+login(String email, String password): Utente
Descrizione	Questo metodo consente di loggare un Utente registrato.
Pre-condizione	**context**: UtenteService::login(String email, String password) **pre:** email != null && password != null
Post-condizione	**context**: UtenteService::login(String email, String password) **post**: utente != null && utente instanceof Donatore || utente instanceof Operatore
Eccezioni	UserNotLoggedException, NoSuchAlgorithmException

Nome Metodo	+logout(Utente utente):
Descrizione	Questo metodo permette di effettuare il logout dalla piattaforma.
Pre-condizione	**context**: UtenteService:: logout(Utente utente) **pre**: utente != null
Post-condizione	/
Eccezioni	AccessNotAuthorizedException

3.2 Organizzazione Sedute Management

Nome classe	OrganizzazioneSeduteService
Descrizione	Questa classe offre le funzionalità di organizzazione per le sedute di donazione in base al tipo di Utente che ha effettuato accesso alla piattaforma.
Metodi	+feedbackPositivo(Donatore donatore, Long idSeduta): void +feedbackNegativo(Donatore donatore, Long idSeduta): void +monitoraggioSeduta(Long idSeduta): ArrayList<Object> +inserimentoGuest(Long idSeduta, Guest guest): Guest +schedulazioneSeduta(Seduta seduta): Seduta +modificaSeduta(SedutaForm sedutaForm, Long idSeduta, Utente utente): Seduta +eliminaSeduta(Long idSeduta): void +visualizzaSeduta(Long idSeduta): Seduta +visualizzaSeduteDisponibili(String codiceFiscale): List<Seduta> +visualizzaElencoSedute(): List<Seduta>
Invariante di classe	/

Nome Metodo	+feedbackPositivo(Donatore donatore, Long idSeduta): void
Descrizione	Questo metodo permette al donatore di confermare la sua partecipazione alla seduta di donazione.
Pre-condizione	**context**: OrganizzazioneSeduteService::feedbackPositivo(Donatore donatore, Feedback feedback) **pre**: donatore != null
Post-condizione	**context**: OrganizzazioneSeduteService::feedbackPositivo(Donatore donatore, Feedback feedback) **post**: sedutaRepository.save(seduta)
Eccezioni	CannotRelaseFeedbackException

Nome Metodo	+feedbackNegativo(Donatore donatore, Long idSeduta): void
Descrizione	Questo metodo permette al donatore di rifiutare la partecipazione alla seduta di donazione.
Pre-condizione	**context**: OrganizzazioneSeduteService:: feedbackNegativo(Donatore donatore, Feedback feedback) **pre**: donatore != null
Post-condizione	**context**: OrganizzazioneSeduteService:: feedbackNegativo(Donatore donatore, Feedback feedback) /
Eccezioni	CannotRelaseFeedbackException

Nome Metodo	+monitoraggioSeduta(Long idSeduta): Seduta
Descrizione	Questo metodo permette di recuperare i dettagli della seduta
Pre-condizione	**context**: OrganizzazioneSeduteService:: monitoraggioSeduta(Long idSeduta) **pre**: idSeduta != null
Post-condizione	/
Eccezioni	CannotLoadDataRepositoryException

Nome Metodo	+inserimentoGuest(Seduta seduta, Guest guest): Guest
Descrizione	Questo metodo permette di aggiungere guest alla seduta di donazione.
Pre-condizione	**context**: OrganizzazioneSeduteService:: inserimentoGuest(Seduta seduta, Guest guest) **pre**: seduta != null AND guest != null
Post-condizione	**context**: OrganizzazioneSeduteService:: inserimentoGuest(Seduta seduta, Guest guest) **post**: guestRepository.save(guest) sedutaRepository.save(seduta)
Eccezioni	CannotSaveDataRepositoryException

Nome Metodo	+schedulazioneSeduta(Seduta seduta): Seduta
Descrizione	Questo metodo permette di creare una nuova seduta.
Pre-condizione	**context**: OrganizzazioneSeduteService:: schedulazioneSedute(Seduta seduta) **pre**: seduta != null
Post-condizione	**context**: OrganizzazioneSeduteService:: schedulazioneSedute(Seduta seduta) **post**: sedutaRepository.save(seduta)
Eccezioni	CannotSaveDataRepositoryException

Nome Metodo	+modificaSeduta(Seduta seduta, Utente utente): Seduta
Descrizione	Questo metodo permette di modificare una seduta
Pre-condizione	**context**: OrganizzazioneSeduteService:: modificaSeduta(Seduta seduta) **pre**: seduta != null && utente != null
Post-condizione	**context**: OrganizzazioneSeduteService:: modificaSeduta(Seduta seduta) **post**: sedutaRepository.save(seduta)
Eccezioni	CannotUpdateDataRepositoryException

Nome Metodo	+eliminaSeduta(Long idSeduta)::
Descrizione	Questo metodo permette di eliminare una seduta
Pre-condizione	**context**: OrganizzazioneSeduteService:: eliminaSeduta(Long idSeduta) **pre**: idSeduta != null
Post-condizione	**context**: OrganizzazioneSeduteService:: eliminaSeduta(Long idSeduta) **post**: sedutaRepository.deleteById(idSeduta)
Eccezioni	CannotDeleteDataRepositoryException

Nome Metodo	+visualizzaSeduta(Long idSeduta): Seduta
Descrizione	Questo metodo permette di recuperare una seduta dato il suo id.
Pre-condizione	**context**: OrganizzazioneSeduteService:: visualizzaSeduta(Long idSeduta) **pre**: idSeduta != null
Post-condizione	/
Eccezioni	CannotLoadDataRepositoryException

Nome Metodo	visualizzaElencoSedute(): List<Seduta>
Descrizione	Questo metodo permette all'operatore di recuperare la lista delle sedute.
Pre-condizione	/
Post-condizione	/
Eccezioni	CannotLoadDataRepositoryException

Nome Metodo	visualizzaSeduteDisponibili(String codiceFiscale): List <Seduta>
Descrizione	Questo metodo permette al donatore di visualizzare l'elenco delle sedute disponibile.
Pre-condizione	/
Post-condizione	/
Eccezioni	CannotLoadDataRepositoryException

3.3 Tesserino Management

Nome classe	TesserinoService
Descrizione	Questa classe permette di gestire le operazioni inerenti alla gestione del tesserino digitale.
Metodi	+creazioneTesserino(Donatore donatore, Tesserino tesserino, Donazione donazione): Tesserino +autodichiarazioneIndisponibilita(Indisponibilita indisponibilita): Indisponibilita +aggiornaTesserino(Utente utente):Tesserino
Invariante di classe	/

Nome Metodo	+creazioneTesserino(Donatore donatore, Tesserino tesserino, Donazione donazione): Tesserino
Descrizione	Questo metodo permette all'operatore di creare un nuovo tesserino.
Pre-condizione	**context**: TesserinoService:: creazioneTesserino(Donatore donatore, Tesserino tesserino, Donazione donazione) **pre**: donatore != null && tesserino != null
Post-condizione	**context**: TesserinoService:: creazioneTesserino(Donatore donatore, Tesserino tesserino, Donazione donazione) **post**: utenteRepository.save(donatore) tesserinoRepository.save(tesserino)
Eccezioni	CannotSaveDataRepositoryException

Nome Metodo	+ autodichiarazioneIndisponibilita(Indisponibilita indisponibilita): Indisponibilita
Descrizione	Questo metodo permette di dichiarare l' indisponibilità di un donatore a donare.
Pre-condizione	**context**: TesserinoService:: autodichiarazioneIndisponibilita(Indisponibilita indisponibilita) **pre**: indisponibilita.getDataProssimaDisponibilita != null
Post-condizione	**context**: TesserinoService:: autodichiarazioneIndisponibilita(Indisponibilita

	indisponibilita) **post**: indisponibilitaRepository.save(indisponibilita)
Eccezioni	CannotSaveDataRepositoryException

Nome Metodo	+aggiornaTesserino(Utente utente):Tesserino
Descrizione	Questo metodo permette di aggiornare il Tesserino.
Pre-condizione	**context**: TesserinoService::aggiornaTesserino(Utente utente) **pre**: tesserino!=null
Post-condizione	**context**: TesserinoService::aggiornaTesserino(Utente utente) **post**:tesserinoRepository.findByDonatore UtenteCodiceFiscale(utente.get CodiceFiscale)
Eccezioni	CannotSaveDataRepositoryException

3.4 Gestione Sedute Management

Nome classe	GestioneSeduteService
Descrizione	Questa classe permette di gestire le operazioni inerenti alla seduta.
Metodi	+salvataggioDonazione(String codiceFiscaleDonatore, Long idSeduta, String tipoDonazione): Donazione +salvataggioIndisponibilita(String codiceFiscaleDonatore, Long idSeduta, IndisponibilitaDonazioneForm indisponibilitaDonazioneForm): Indisponibilita
Invariante di classe	/

Nome Metodo	+salvataggioDonazione(String codiceFiscaleDonatore, Long idSeduta, String tipoDonazione): Donazione
Descrizione	Questo metodo permette di registrare una nuova donazione da parte di un donatore e aggiorna il tesserino del donatore.
Pre-condizione	**context**: GestioneSeduteService:: salvataggioDonazione(String codiceFiscaleDonatore, Long idSeduta, String tipoDonazione) **pre**: codiceFiscaleDonatore != null && donatore != null && seduta != null
Post-condizione	**context**: GestioneSeduteService:: salvataggioDonazione(String codiceFiscaleDonatore, Long idSeduta, String tipoDonazione)) **post**: donazioneRepository.save(donazione) tesserinoRepository.save(tesserino)
Eccezioni	CannotSaveDataRepositoryException

Nome Metodo	+ salvataggioIndisponibilita(String codiceFiscaleDonatore, Long idSeduta, IndisponibilitaDonazioneForm indisponibilitaDonazioneForm): Indisponibilita
Descrizione	Questo metodo permette di registrare una indisponibilità per un donatore
Pre-condizione	**context**: GestioneSeduteService:: salvataggioIndisponibilita(String codiceFiscaleDonatore, Long idSeduta, IndisponibilitaDonazioneForm indisponibilitaDonazioneForm) **pre**: donatore != null && seduta != null
Post-condizione	**context**: GestioneSeduteService:: salvataggioIndisponibilita(String codiceFiscaleDonatore, Long idSeduta, IndisponibilitaDonazioneForm indisponibilitaDonazioneForm) **post**:indisponibilitaRepository.save(indisponibilita)
Eccezioni	CannotSaveDataRepositoryException

4. Class diagram

Il Class diagram è presente in un documento apposito: 2021_CD_C9_DigitalDonation_Abate-Ferrara_Vers1.1 per ragioni di visibilità.
Lo stesso documento in formato PDF sarà allegato alla consegna finale.

5. Design patterns

In questa sezione verranno mostrati i Design Patterns utilizzati nello sviluppo del sistema, per ognuno di essi verrà mostrato il problema che doveva risolvere, la soluzione al problema e le conseguenze dovuto all'utilizzo dello specifico pattern.

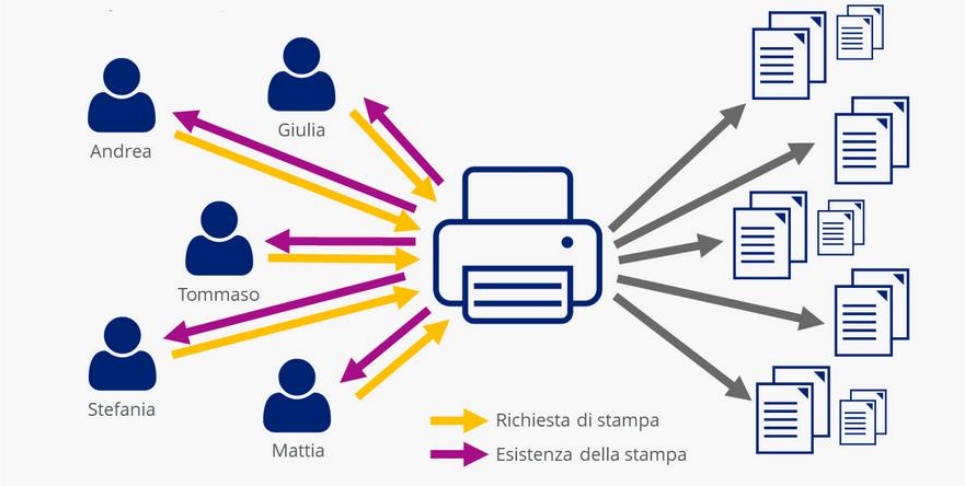
5.1 Singleton

Problema

In ogni tipo di sistema ci possono essere delle risorse delle quali per garantire il corretto funzionamento deve esisterne una singola istanza. In genere, quando si scrive una classe il suo costruttore viene dichiarato public in modo da poterla istanziare. Avendo un costruttore pubblico un metodo è in grado generare "infinite" istanze di quella classe.

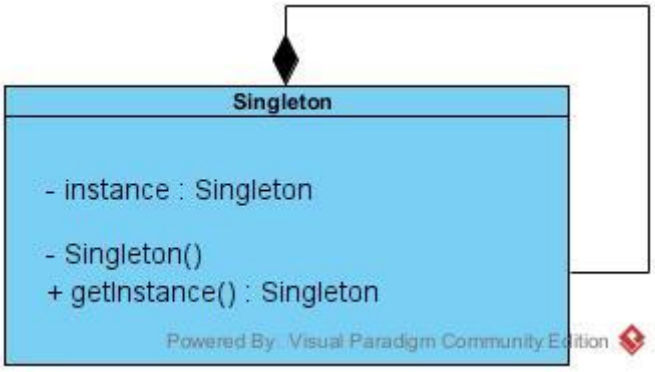
Soluzione

Il Singleton assicura al programmatore che della classe voluta esisterà sempre e solo un'unica istanza. Incapsula la propria creazione così da poter esercitare pieno controllo su quando e come sia possibile accedervi.
In Spring Boot il pattern Singleton può essere utile quando si gestiscono risorse condivise o si forniscono servizi trasversali.
Il pattern Singleton verrà utilizzato nello specifico per il servizio di mailing previsto dal sistema.



Conseguenze

- Scrivere un singleton è un'operazione veloce e poco complicata, in quanto non consta di un numero elevato di variabili (globali).
- Un singleton viene istanziato solamente quando richiesto. Questa caratteristica chiamata viene lazy loading

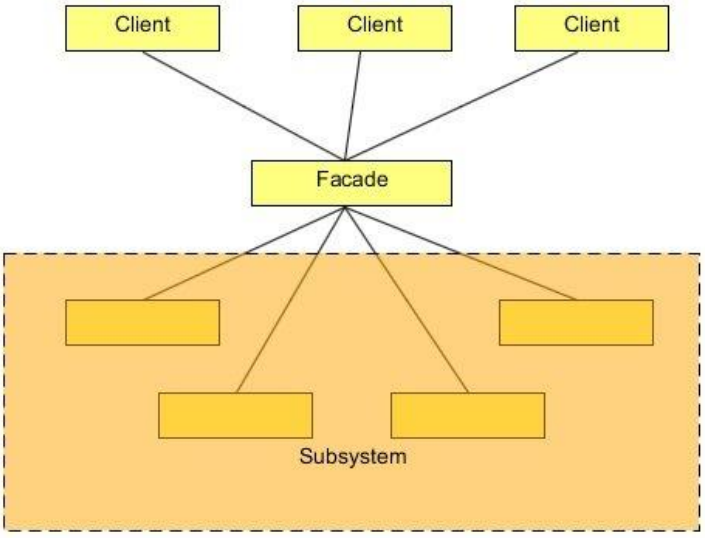


5.2 Facade

Problema

L'Ente Nazionale Ricerca Sangue essendo un sistema molto complesso le operazioni di manutenibilità ed aggiornamento risultano essere complicate.
Spesso si hanno tante classi che svolgono funzioni correlate e l'insieme delle interfacce può essere complesso e difficile capire qual è l'interfaccia essenziale ai client per l'insieme di classi.
C'è bisogno di ridurre le comunicazioni e le dipendenze dirette fra i client ed i sottosistemi.

Soluzione
Il Design Pattern Facade verrà applicato alle classi di servizio, permettendo di implementare una propria logica di business, rendendo più facile l'interfacciarsi ad essa.
Il Pattern Facade fornisce un'unica interfaccia semplificata ai client e nasconde gli oggetti del sottosistema, questo riduce la complessità dell'interfaccia e quindi delle chiamate.
Inoltre invoca i metodi degli oggetti che nasconde ed il client interagisce solo con l'oggetto Facade.
Per implementare delle modifiche basta cambiare l'implementazione dei metodi rendendo più facili le operazioni di manutenibilità ed aggiornamento del sistema.



Conseguenze

- Basso accoppiamento tra sottosistemi e client
- Operazioni di aggiornamento e manutenzione semplificate
- Nasconde ai client l'implementazione del sottosistema
- Riduce le dipendenze di compilazione

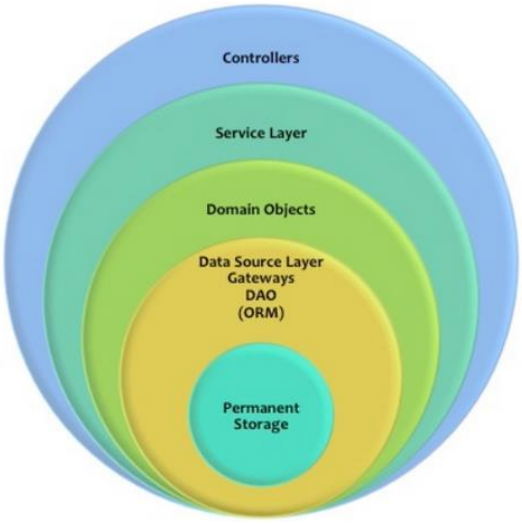
5.3 Service Layer

Problema

Tipicamente le applicazioni richiedono differenti tipi di interfacce ai dati che salvano e la loro logica che implementano. Anche se lavorano su dati differenti le interfacce spesso richiedono interazioni comuni con l'applicazione per accedere, manipolare i dati che invocano e la logica di business.
Servirebbe un'interfaccia che permette il riuso di funzionalità e interazione con i dati raggruppandoli in servizi comuni.

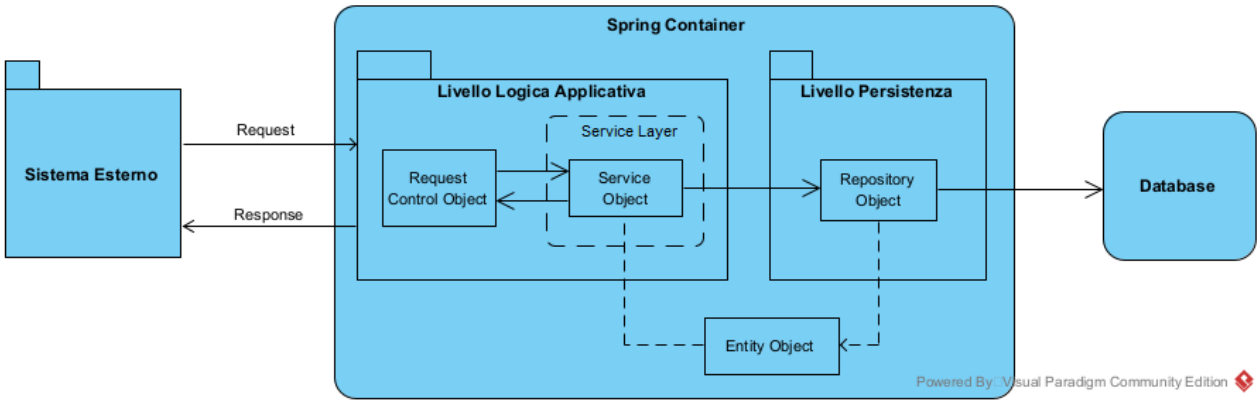
Soluzione

Il service Layer è un'astrazione sopra il dominio logico. Definisce i confini di un'applicazione con livello di servizi che stabilisce un insieme di operazioni e coordina le risposte di ogni operazione.
In Spring Boot il service Layer definisce quali funzionalità vengono fornite, come si accede, cosa passare e cosa ottenere in cambio indipendentemente da qualsiasi porta e dal funzionamento interno.
E' molto utile per servizi web, code di messaggi ed eventi programmati.



Conseguenze

- Maggiore riusabilità del codice
- Minore accoppiamento tra i livelli del controller e il Dominio degli Oggetti.



5.4 Repository

Problema

L'ENRS dato che è una web application che punta a gestire tutte le funzionalità necessarie per le donazioni, presenta un Database molto vasto in modo da permettere al sistema di effettuare operazioni rapide e sicure per la molteplicità di dati da gestire.

E' necessario quindi aumentare la granularità e diminuire l'accoppiamento tra classi, aumentando la modularità del sistema per la persistenza.

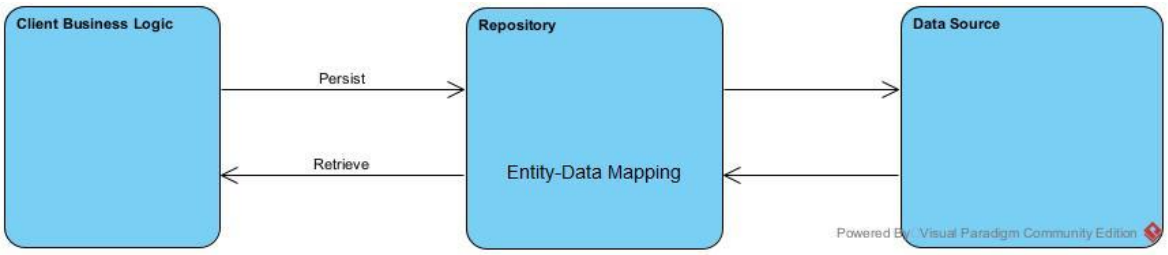
Dato che ogni classe ha bisogno di interfacciarsi con il Database, al posto di fare operazioni con classi di servizio lo si fa con classi Repository.

Soluzione

Repository ci permette di creare un'interfaccia con le firme di metodi per la richiesta di dati persistenti che risiedono o su un database o su un file statico.

Oltre all'interfaccia ci permette di creare una classe che implementa i metodi definiti che si occupano della gestione vera e propria.

In Spring Boot il pattern Repository può essere utilizzato per la definizione di interfacce, semplificando il lavoro nella creazione delle query.



Conseguenze

- Riduzione della duplicazione del codice delle query
- Codice delle query in un singolo spazio
- Riduzione accoppiamento tra le classi di business logic e la base dati
- Aumento della modularità del sistema per la persistenza.

6.Glossario

D

Directory

È una cartella che contiene documenti.

C

Classe

Una classe è un costrutto usato come modello per creare oggetti.

E

Eccezione:

Condizione che altera il normale flusso di controllo.

F

Form:

È la parte di interfaccia utente che permette all'utente di inserire ed inviare dati digitati tramite tastiera.

L

Logica di business:

Logica delle applicazioni che risolve un problema.

I

Implementazione:

Realizzazione di una procedura di elaborazione automatica dei dati.

T

Trade off:

Scelte e compromessi tra design goals dissonanti.