Relazione Progetto

# "Clean Service Manager"

Programmazione Orientata agli Oggetti

Deadline A - 25 Febbraio 2021

D'Auria Nicola
Di Biasi Vanessa
Laghi Aurora
Spinelli Silvia

# Indice

# Capitolo 1

# Analisi

## 1.1 Requisiti

Il gruppo si pone come obiettivo quello di creare un'applicazione utile per la gestione di un'impresa specializzata nella sanificazione di strutture aziendali.

Ad oggi, ogni azienda deve assicurare la pulizia giornaliera e la sanificazione periodica dei locali, degli ambienti, delle postazioni di lavoro e delle aree comuni. Per fare ciò spesso, ma sopratutto in situazioni di emergenza sanitaria, si affidano a imprese esterne abilitate alla sanificazione.

Questa applicazione, volendo essere utile ai dipendenti dell'azienda, deve innanzitutto essere in grado di gestire tutte le operazioni di inserimento, modifica ed eliminazione di clienti e di dipendenti (con relativo grado di permessi). Secondariamente funge da agenda e storico dei vari appuntamenti e processi di sanificazione effettuati presso il domicilio del cliente. Infine, fornisce uno strumento di analisi attraverso la consultazione di grafici e statistiche, in particolare per i costi dei prodotti e il tempo lavorativo dei vari processi.

**Requisiti funzionali**

- Questa applicazione è in grado di mettere a disposizione dei dipendenti delle funzionalità per gestire la clientela e il magazzino.
- Il dipendente che si interfaccia può gestire le richieste di sanificazione, in questo modo il software sarà utile anche in una prospettiva di agenda digitale interna.
- I dipendenti che giovano di un certo grado di permessi (che li rende amministratori), possono accedere a una sezione di statistica e grafici sui dati interni all'azienda utili per decisioni di marketing.

- L'applicazione mantiene i dati in memoria in modo persistente per far sì che non vengano persi alla terminazione dell'applicazione.

**Requisiti non funzionali**

- L'interfaccia deve risultare semplice, intuitiva, veloce e graficamente interessante.
- L'applicazione in fase di richiesta di nuove sanificazione deve fornire anche un preventivo su costi e tempistiche totali.
- Le fasi di sanificazione che compongono il processo totale devono essere personalizzabili per ogni richiesta e devono essere gestibili (inserimento di nuove o eliminazione di esistenti) dall'azienda nel tempo.

## 1.2 Analisi e modello del dominio

L'applicazione Clean Service Manager si basa sulla gestione della clientela e, di conseguenza, sulle richieste di sanificazioni. Essa sarà l'agenda digitale che i dipendenti andranno a utilizzare per tenere traccia dei dati dei clienti, degli appuntamenti da loro presi e del tipo di sanificazione che desiderano ricevere. L'azienda, infatti, offre una personalizzazione del servizio offerto sulla base dell'obbiettivo dell'intervento stesso, ad esempio se incentrato su una pulizia ordinaria oppure straordinaria. Per ottenere ciò, le sanificazioni si compongono di varie macro-fasi che il cliente può scegliere se richiedere.

Dato che si impiegheranno prodotti propri dell'azienda più o meno costosi, infine, si attua un calcolo dei costi sostenuti dall'azienda e dell'incasso delle sanificazioni, le quali risultano essere fattori decisivi per attuare strategie di marketing. La stima dei costi medi e delle tempistiche delle sanificazioni saranno consultabili attraverso dei grafici utili a questo scopo.

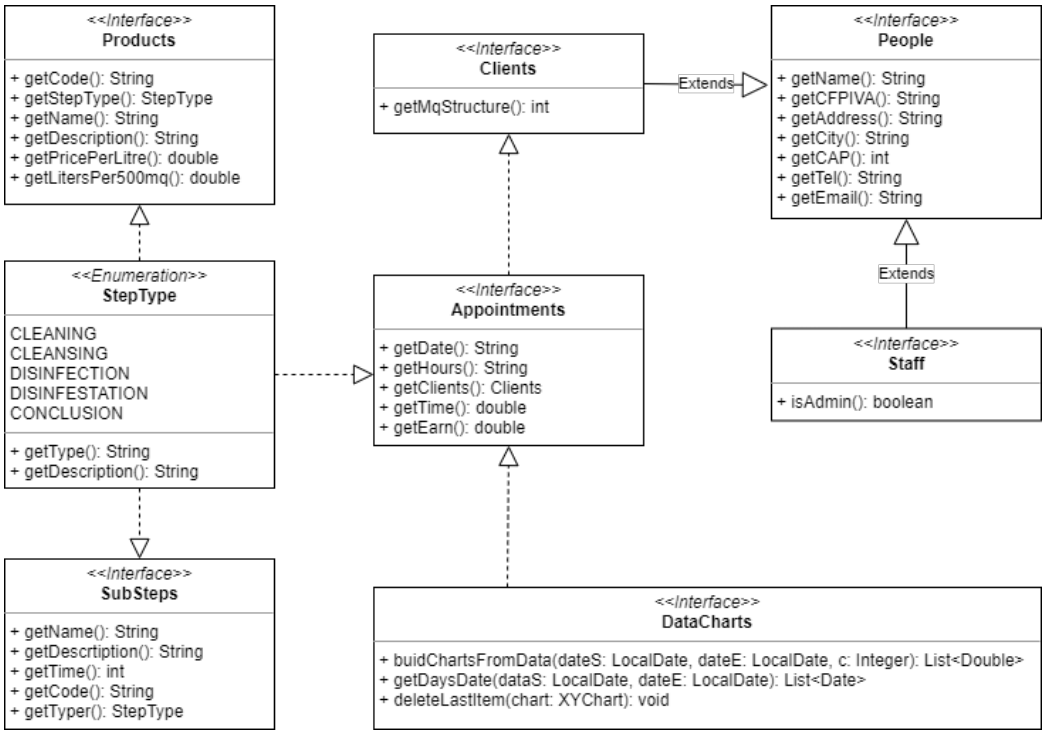


Figura 1.1: Architettura generale

# Capitolo 2

# Design

## 2.1 Architettura

Il gruppo ha sviluppato il progetto seguendo il pattern architetturale MVC, nel quale il Controller fa da intermediario tra il Model e la View.

Il Model contiene i dati e gli algoritmi per la gestione delle entità interne l'applicazione. La View rappresenta graficamente il contenuto del Model specificando come gli oggetti contenuti in esso debbano venire presentati all'utente. Il Controller trasforma gli input dell'utente provenienti dalla View in elementi processabili dal Model, il quale li elabora e restituisce al Controller per essere mostrati nella View.

Si è cercato di far si che la View contenesse meno logica possibile demandandola al Controller oppure sfruttando classi appositamente create, presenti nel package utility. Allo stesso modo, il Model non ha alcun collegamento con la View ma intergisce unicamente con il Controller.

La divisione nei package permette di far si che i soli metodi dichiarati pubblici siano visibili (e quindi accessibili) al di fuori di essi. Il progetto è stato quindi suddiviso in tre package principali, contenenti relativamente le classi di MVC, affiancati ad altri due package: application con il Main.java e utilities con classi a supporto di alcune funzionalità, per la maggior parte inerenti alla View. È presente, inoltre, il package test contenente i vari test di JUnit.

## 2.2 Design dettagliato

I compiti sono stati ripartiti tra i vari componenti a far sì che ognuno possa lavorare su ogni parte del pattern MVC, in parti più o meno equivalenti:

- D'Auria Nicola: Area grafici con statistiche e Home di benvenuto.

- Di Biasi Vanessa: Gestione degli utenti (clienti e dipendenti) e dei prodotti aziendali.
- Laghi Aurora: Area processi di sanificazione con sotto-fasi.
- Spinelli Silvia: Gestione richieste di sanificazione (appuntamenti) e salvataggio dei dati.

### 2.2.1 Di Biasi Vanessa

La parte a me assegnata si incentra nella gestione dei clienti, dei dipendenti e dei prodotti impiegati nelle fasi di sanificazione. A livello di Model ho perciò creato delle strutture adeguate alla modellazione di queste entità permettendone quindi l'inserimento, la modifica e l'eliminazione. Questi elementi, che devono essere visualizzabili in un interfaccia grafica, sono modellabili da un'entità presente nel Controller, a far sì che modifiche eseguite dall'utente nella View non vadano a interagire col Model.

Dato che clienti e dipendenti sono persone ho modellato questo concetto attraverso una classe astratta (*People*) contenente le informazioni anagrafiche utili all'azienda. Ho scelto di dichiarare tale classe astratta affinché non sia possibile istanziarla visto che il suo utilizzo è quello di essere specializzata nelle classi di cliente (*Clients*) e di dipendente (*Staff*). Ciò che è stato aggiunto in queste ultime due sottoclassi sono due semplici informazioni: per i clienti è presente la dimensione in $m^2$ della struttura che sarà luogo degli interventi di sanificazione, mentre per i dipendenti si è optato per una semplice verifica sui permessi che possiede all'interno dell'applicazione al fine di effettuare le operazioni che risultano necessitare di maggiori responsabilità.

L'entità che modella i prodotti è altresì semplice, con una specifica del costo, della quantità da utilizzare secondo una certa metratura e della fase di sanificazione in cui deve essere utilizzato.

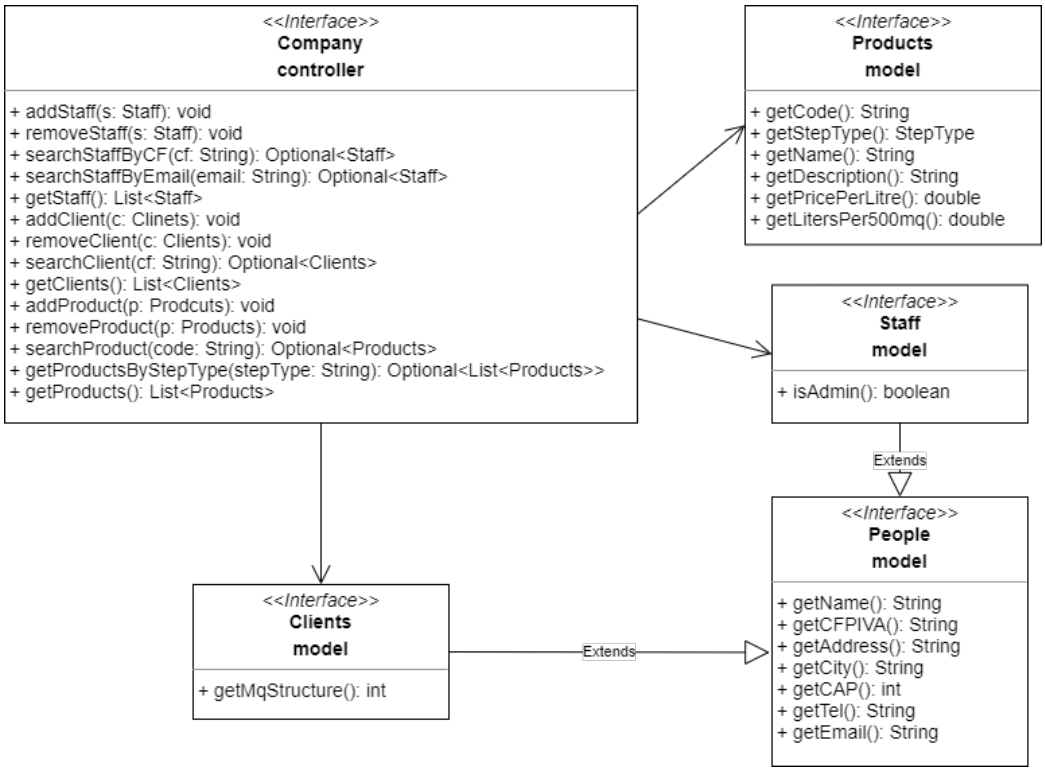


Figura 2.1: Schema UML dalla sottoparte dell'applicazione riguardante il Model degli utenti e dei prodotti e la relazione con il Controller. Dallo schema si nota come *Clients* e *Staff* estendano la classe *People*. Queste due assieme a *Products* vengono poi gestite dentro *Company*.

Le strutture dati per la memorizzazione e i metodi per la loro manipolazione sono stati inseriti in una classe (*Company*) che rappresenta il concetto di azienda. Ciò mi ha permesso di utilizzare un'unica istanza per richiamare i dati e i metodi delle classi sopra descritte. Ho scelto di utilizzare il pattern di progettazione SINGLETON, in modo da avere un'unica istanza della classe accessibile globalmente. Questa classe si pone come unico intermediario tra le classi di Model di mia competenza e le View relative.

Di seguito si mostra lo schema UML rappresentante le implementazioni inerenti Model e Controller, non si specificano nuovamente i metodi delle interfacce che implementano in quanto già pienamente elencati nello schema di cui sopra.

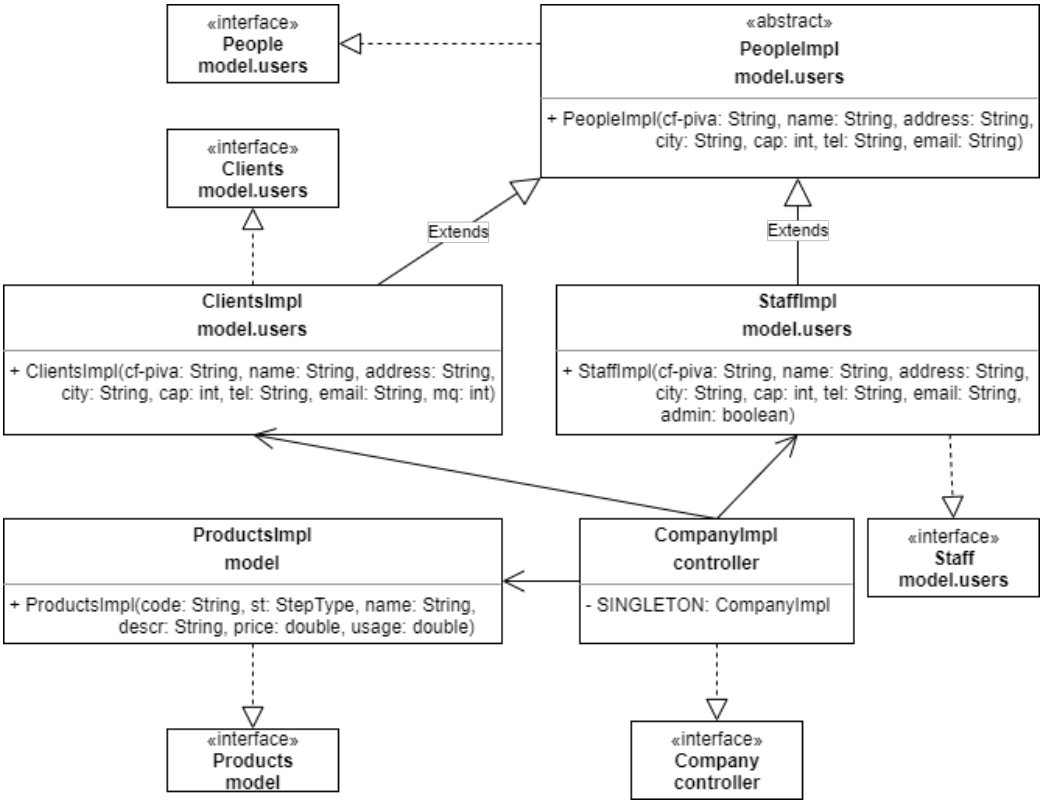


Figura 2.2: Schema UML di dettaglio della sottoparte riguardante Model e Controller inerenti utenti e prodotti. Da questo schema si può intendere come le varie interfacce siano state implementate andando a interagire tra di loro in un modo che rispecchia le relazioni tra interfacce descritte nello schema UML precedente. In particolare, si può vedere il raggruppamento delle classi riguardanti gli utenti in un package *users* contenuto nel *model* e l'utilizzo di SINGLETON in *Company*.

Per quanto riguarda le View da me implementate (*ClientsView, StaffView e ProductsView*), ma in generale per l'intero progetto, si è optato per l'utilizzo della libreria grafica Swing.

Per quanto riguarda la parte grafica ed estetica, la scelta iniziale è stata l'utilizzo del BorderLayout e del GridLayout in quanto molto semplici da utilizzare, tuttavia mi risultava poco flessibile ed esteticamente non soddisfacente. Ciò mi ha spinto nell'utilizzare il GroupLayout con il quale mi sono trovata più a mio agio ma, volendo mantenere un Layout personalizzato nei panel, ho concluso adottando un approccio ibrido che ha portato alla struttura visiva attesa.

Le View, affinché fossero user-friendly ma anche esteticamente coerenti, risultano simili tra loro a causa del posizionamento necessariamente simile degli stessi componenti grafici. Esse si compongono inizialmente di una tabella che mostra i dati presenti in azienda, secondariamente di due sezioni: una che recupera i dati dell'elemento che l'utente seleziona e una che permette l'inserimento di una nuova entità (cliente, dipendente o prodotto) e la manipolazione (modifica ed eliminazione) dell'elemento selezionato nell'area superiore. In particolare, se si effettua l'eliminazione di un dipendente viene richiesto un controllo dei permessi attraverso un pop-up richiamabile da una classe *PopUp* presente nel package *utility*.

Poichè la View necessitava di implementazioni logiche che esulavano dalla sua natura, ho deciso di creare un package ulteriore (*utility*) contenente delle classi che potrebbero intendersi di Controller, tuttavia non essendo di fatto collegate con elementi di Model ho ritenuto essere più apprezzabile mantenerla parallela agli altri packages. In essa sono presenti classi utili a una personalizzazione degli Alert (*PopUp*), al controllo degli input effettuati dall'utente (*InputValidator*) e alla creazione di elementi di View comuni (*ConstantsCleanSvc*). In particolare, *InputValidator* utilizza il metodo *Pattern.matches()* contenuto nella libreria java.util.regex per verificare se l'espressione data in input rispetta la regex (Regular Expressions) dichiarata. Infine, *ConstantsCleanSvc* permette di risolvere il problema dei "magic number" e il riutilizzo continuo di certi elementi nelle view, ad esempio titolo, font, dimensioni standard per i vari componenti grafici, ecc.

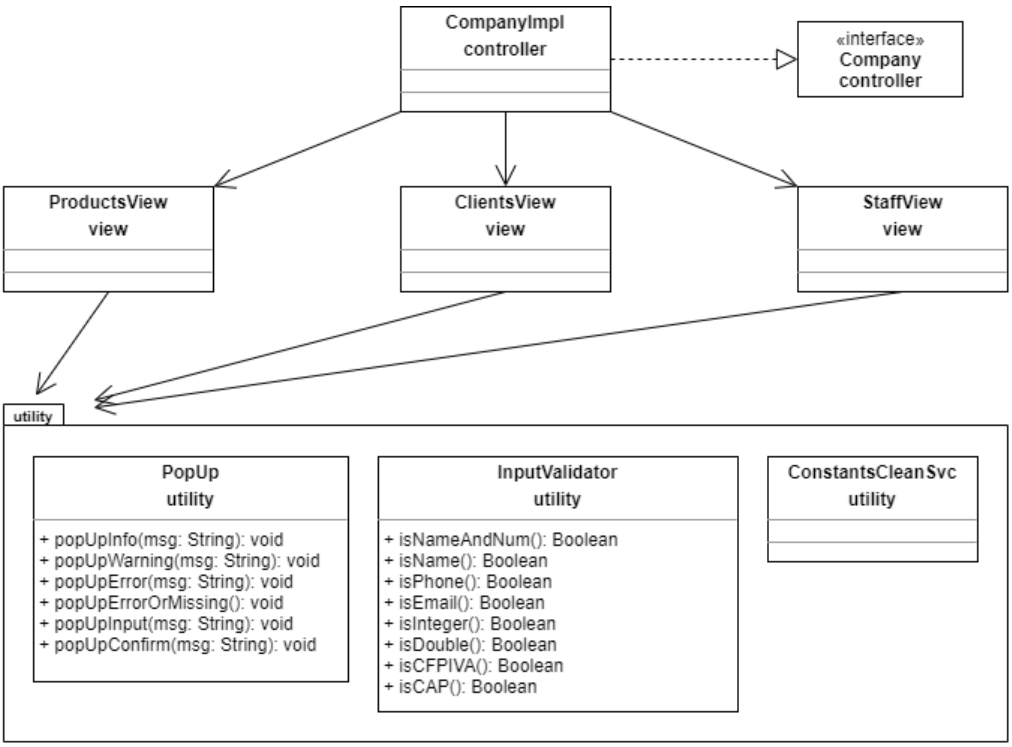


Figura 2.3: Schema UML di dettaglio della sottoparte riguardante Controller e View inerenti utenti e prodotti. Da questo schema si nota che le View interagiscono unicamente con la classe *CompanyImpl* nel controller e utilizzano le classi contenute nel package *utility*.

### 2.2.2 Spinelli Silvia

La parte a me assegnata riguarda il salvataggio e il caricamento dei dati e la gestione delle richieste di sanificazione, quindi degli appuntamenti.

A livello di Controller ho realizzato la lettura e la scrittura dei dati riguardanti tutte le entità coinvolte, ovvero appuntamenti, clienti, prodotti, dipendenti e processi, e inoltre il salvataggio delle statistiche. A ognuna di queste entità è associato un file testuale in cui sono memorizzati tutti i valori corrispondenti ai campi. Per realizzare la lettura e la scrittura su file è stato utilizzato il pattern STRATEGY che prevede l'implementazione, da parte di specifiche classi nel package *controller.backupFile*, di un'unica interfaccia SAVEANDLOAD. L'utilizzo di questo pattern è fondamentale e utile nel caso in cui si vogliano aggiungere nuove funzionalità: basterà, infatti, implementare ancora una volta l'interfaccia base. Il salvataggio dei dati è effettuato alla pressione del pulsante "Salva ed Esci", mentre il caricamento viene richiamato nel Main, quindi all'avvio dell'applicazione.

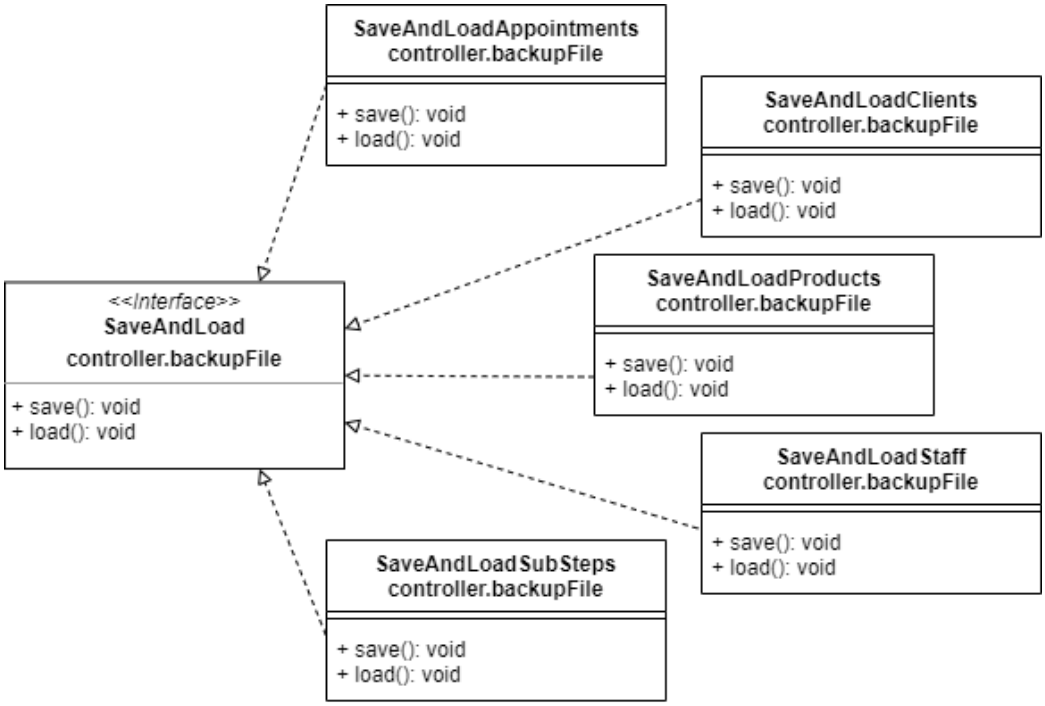


Figura 2.4: Schema UML della sottoparte riguardante il salvataggio e il caricamento dei dati realizzato tramite il pattern *strategy*

A livello di Model ho realizzato la classe che modella gli appuntamenti, la cui peculiarità è la presenza del riferimento alla classe dei clienti, indispen-

sabile al fine di recuperare il cliente associato alla richiesta di sanificazione.

Infine, ho implementato la View, sempre riguardante gli appuntamenti, tramite la libreria Swing. Per gestire il layout mi sono avvalsa dell'utilizzo del GroupLayout per ottenere un soddisfacente posizionamento dei pannelli, unitamente all'uso del BorderLayout e del GridLayout, quest'ultimo utilizzato per l'allineamento dei vari campi testuali, cercando allo stesso tempo di avere uniformità e coerenza con tutte le altre view del progetto. La finestra principale mostra inizialmente una tabella in cui sono presenti i dati delle varie richieste di sanificazione, in seguito due pannelli in cui è possibile ricercare, tramite data e ora, l'appuntamento desiderato ed in seguito eliminarlo. Infine, dall'ultima sezione di questa View è possibile accedere a un'altra finestra in cui poter inserire un appuntamento e visualizzare il riepilogo, realizzato dalla collega Laghi. Per l'inserimento è necessario scegliere una data, un orario, un'azienda o un cliente (definiti tramite nome e Codice Fiscale/Partita IVA per evitare omonimie), uno o più processi e inserire il numero di dipendenti. L'inserimento è possibile se la data e l'ora non sono già state prenotate e se tutti i dati immessi sono validi, in caso contrario vengono mostrati degli Alert. Se l'inserimento è avvenuto correttamente, si salvano direttamente i dati necessari per le statistiche (data, minuti ed entrate), utili per lo svolgimento della parte riguardante i grafici del collega D'Auria.

L'intermediario tra il Model e la View è la classe sopracitata COMPANY, che utilizza il pattern SINGLETON. Nella classe realizzata in collaborazione con la collega Laghi, ovvero *NewAppointmentView*, è presente un'altra classe che utilizza questo pattern, ovvero PROCESS. Inoltre, per indicare all'utente la presenza di dati errati o mancanti, vengono richiamati i metodi della classe POPUP presente nel package *utility*. Per evitare, infine, l'utilizzo dei "magic numbers" si richiamano le costanti contenute, sempre nello stesso package, nella classe CONSTANTSCLEANSVC.

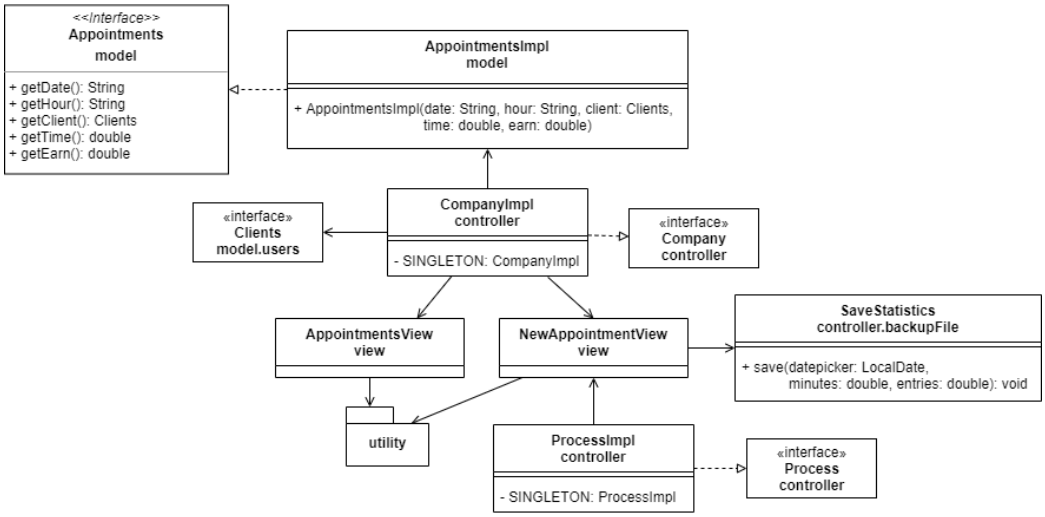


Figura 2.5: Schema UML della sottoparte riguardante l'interazione tra Model e View degli appuntamenti

### 2.2.3 Laghi Aurora

La realizzazione della mia parte è incentrata sulla gestione del processo di sanificazione con le rispettive sotto-fasi. L'analisi del design è partita dall'implementazione del Model, ho preferito organizzare il mio lavoro creando due packages:

- *model.step.enumerations*, contenente l'enumerazione StepType che rappresenta le macro-fasi del processo. Ho deciso di utilizzare questo tipo di classe poiché si dispone di un numero limitato di macro-fasi, non soggette ad evoluzione nel tempo.

- *model.step*, racchiude lo scheletro per modellare le sotto-fasi del processo di sanificazione fornendo la possibilità di eliminazione e d'inserimento. Per questa entità vengono specificati lo StepType di appartenenza e la rispettiva durata, che verrà successivamente utilizzata per il calcolo relativo al tempo impiegato per lo svolgimento del processo, presso il domicilio del cliente.

Per evitare un'interazione diretta fra Model e View, ho deciso di elaborare l'entità SubSteps attraverso un'altra classe presente nel Controller (*Process*).

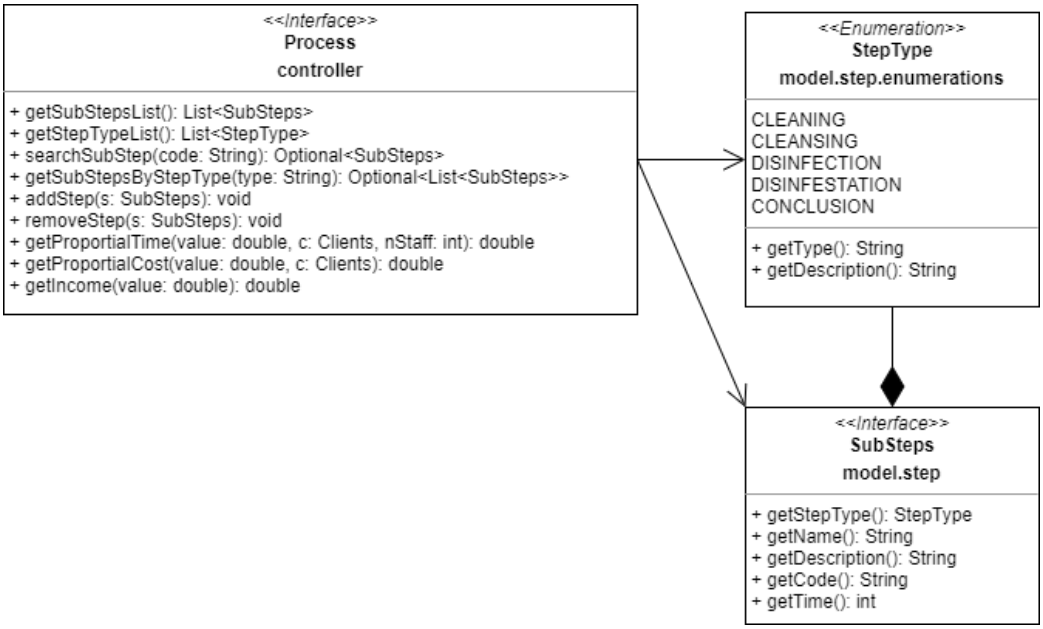


Figura 2.6: Schema UML della sottoparte riguardante l'implementazione del Model delle rispettive macro-fasi e sotto-fasi e la loro interazione con il Controller. Si può notare dallo schema che *SubSteps* è legata all'enumerazione *StepType* in quanto ogni sotto-fase è legata ad una sola macro-fase. Inoltre, entrambe vengono gestite unicamente all'interno della classe *Process* nel Controller.

Per la parte di Controller, quindi per la gestione delle strutture dati e dei metodi, è stata pensata la classe *PROCESS* che racchiude il concetto vero e proprio di processo di sanificazione; ho scelto di implementare una classe differente da quella implementata della collega Di Biasi in quanto ritengo che riguardino due concetti diversi ed inoltre ho voluto evitare un appesantimento della sua classe con ulteriori metodi implementativi. In questo modo, mi è risultato più semplice gestire la chiamata dei dati e dei metodi della classe descritta attraverso un'unica istanza; infatti, ho optato per l'impiego di SINGLETON, il pattern di progettazione che mi ha garantito l'utilizzo di una singola istanza accessibile in maniera globale.

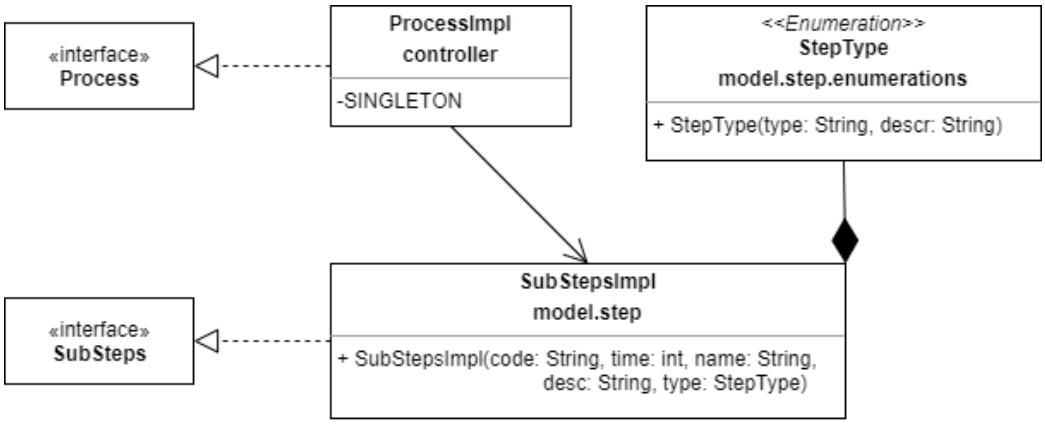


Figura 2.7: Schema UML dettagliato che rappresenta l'interazione Model e Controller delle sotto-fasi. Si noti, come queste interfacce sono state fatte interagire mantenendo lo schema delle relazioni presentate nell'UML sopra riportato. Infine l'impiego di SINGLETON in *Process*.

Per concludere, nella parte di View a me dedicata è stato necessario lo studio e l'utilizzo della libreria Swing di Java. Per l'implementazione della classe *SubStepView*, come gestore di layout, ho impiegato GroupLayout, questo mi ha consentito di gestire separatamente, con i rispettivi layout orizzontale e verticale, i differenti gruppi di componenti che ho inserito nella finestra per la creazione della View da me pensata.

In primo luogo, è composta da una tabella mostrante tutte le sotto-fasi presenti al momento nel software aziendale, poi da due panelli posti parallelamente alla tabella ma sequenzialmente fra loro. Il primo è stato creato per l'inserimento di una nuova sotto-fase, mentre il secondo per l'eliminazione di una di queste scegliendola da una lista. Per la realizzazione della View si sono resi necessari Alert e controlli sugli input, sono state così utilizzate le classi presenti all'interno del package *utility*. Attraverso la creazione della

classe *ConstantCleanSvc* è stato più semplice gestire il problema dei "magic number".

Mi sono poi interfacciata con la collega Spinelli Silvia per la realizzazione della View in comune, cioè *NewAppointmentView*. Qui ho inserito attraverso l'utilizzo di JCheckBox tutte le macro-fasi disponibili per il processo, alcune selezionabili e altre obbligatorie poiché necessarie di base; inoltre, è stato posto anche un campo per specificare il numero di dipendenti che verranno impiegati dall'azienda per svolgere il lavoro di sanificazione. Nella sezione di riepilogo sono indicati i tempi complessivi delle singole macro-fasi, dati dalla somma dei tempi specificati nella creazione delle sotto-fasi per quella determinata macro-fase e tenendo anche conto del numero di dipendenti precedentemente specificato; è, inoltre, possibile vedere il tempo totale del processo in modo da poterlo riferire al cliente e l'incasso totale per quel determinato processo.

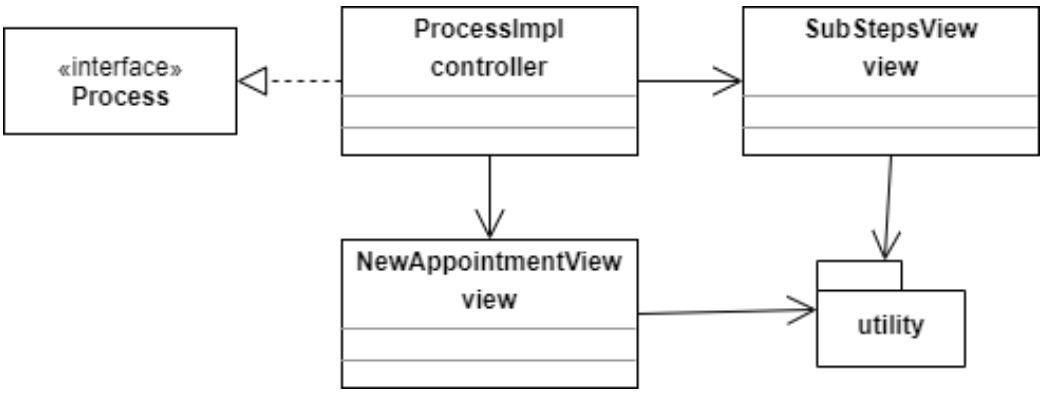


Figura 2.8: Schema UML dettagliato che rappresenta la View delle sotto-fasi. Si noti, come queste interfacce sono state fatte interagire unicamente con la classe del Controller *ProcessImpl*.

### 2.2.4 D'Auria Nicola

La parte di cui mi sono occupato è stata l'implementazione e quindi la realizzazione dei grafici relativi ai tempi di lavoro complessivi giornalieri e alle corrispettive entrate in determinati intervalli di tempo scelti su volontà dell'utente, successivamente lo sviluppo dell'interfaccia grafica di benvenuto. Dopo un'analisi iniziale ho deciso di utilizzare il pattern Strategy essendo più flessibile e permettendo una riusabilità, in particolare gli oggetti sono liberi nell'implementazione. Quindi di conseguenza la mia idea è stata quella di avere una classe principale nel model *DataChartImpl* che implementa

*DataChart* la quale si occupa di tutta la parte di logica, ad essa collegata come controller la classe *ControllerAdministratorChartsImpl* che fa da ponte per la view *AdministratorChartView*. La View è stata sviluppata adottando più border layout in modo da poter separare le sezioni, in particolare per poter visualizzare il grafico nel modo corretto. Ho cercato di rendere l'interfaccia quanto più user friendly e pulita possibile. La view è stata pensata per poter permettere la visualizzazione di più linee nel grafico, in periodi di tempo indipendentemente più o meno lunghi, questo per dare all'utente la possibilità di poter mettere a confronto per ciascuna scelta i relativi dati. AdministratorChartView comunica con il model attraverso il controller, come vuole il pattern MVC, nel mio caso la classe *AdministratorChartControllerImpl*, che implementa l'interfaccia *AdministratorChartController*, fa da collante tra queste due parti e si occupa di tutto ciò che riguarda le segnalazioni per l'aggiornamento del grafico: aggiunta, rimozione, reset. Ad ogni interazione corrisponde un relativo comportamento del model. Alla richiesta di aggiunta di una linea nel grafico ho pensato fosse utile creare un'eccezione, *DataException*, che si occupasse di segnalare le varie anomalie che possono verificarsi quando l'utente andrà a settare l'intervallo di tempo tra le date desiderato, in particolare può capitare la mancanza di una delle due date oppure che la data da cui partire sia successiva alla data di arrivo, quindi a mio avviso è stato necessario avere un eccezione simile per segnalare problemi di questo tipo. Successivamente ho ritenuto necessario anche la creazione di una nuova Exception, *ChartException*, che mi indicasse quando non fosse possibile effettuare un operazione di aggiunta o di rimozione di una linea dal grafico. Quindi le operazioni logiche vere e proprie vengono eseguite dal model attraverso la classe *DataChartImpl*, che implementa DataChart. La classe si occupa innanzitutto di distinguere la richiesta effettuata dall'utente, per cui fa riferimento ad un enum *DatiDaVisualizzareEnum* per distinguere le due scelte. Il core vero e proprio della classe è la capacità di poter svolgere le operazioni di aggiornamento in modo reattivo indipendentemente da quanto avviene al di fuori del suo modello, per cui seppur cambiando la view il model reagirà sempre allo stesso modo. L'aspetto più difficile di questa sezione è stato venire incontro alle implementazioni dei miei colleghi, in particolare per la lettura di dati da file e il conseguente aggiornamento del grafico secondo quanto richiesto in modo da rendere più preciso possibile il disegno raffigurato dal grafico.

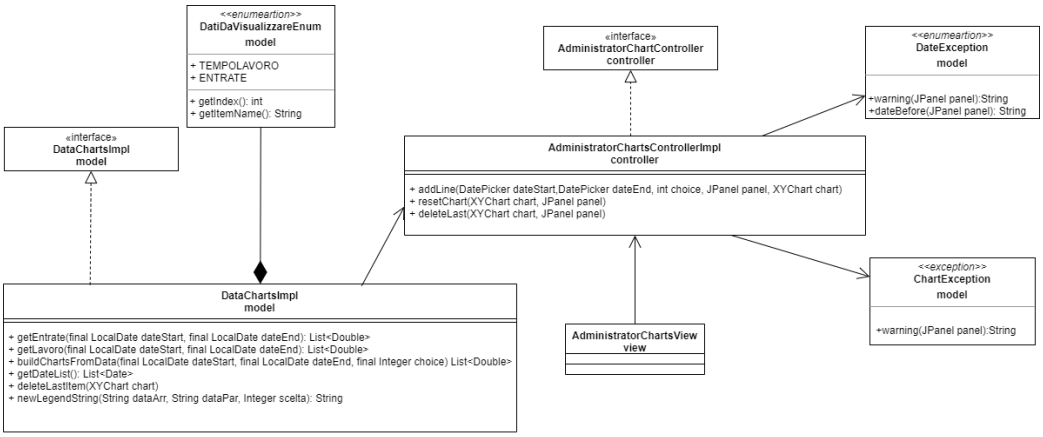


Figura 2.9: Dallo schema uml risulta più chiaro l'approccio del pattern Architetturale MVC, da notare come il model non comunichi mai direttamente con la view, a fare da intermediario per le operazioni di input e di output la classe *AdministratorChartControllerImpl*

# Capitolo 3

# Sviluppo

## 3.1 Testing automatizzato

Per il test automatizzato dell'applicazione abbiamo impiegato delle classi di test utilizzando il framework JUnit cercando di sottoporre a controlli più metodi possibili (quasi ogni classe del model e del controller posseggono un test corrispondente). Per la View invece si è reso più efficiente effettuare dei controlli manuali atti a verificare il corretto funzionamento dell'interfaccia e dei meccanismi di eccezione oltre al posizionamento dei vari componenti.

## 3.2 Metodologia di lavoro

Il progetto è stato suddiviso in funzionalità tali da far sì che ogni componente del gruppo potesse sviluppare elementi di Model, di Controller e di View. In questo modo è stato possibile ottenere uno sviluppo individuale particolarmente indipendente dagli altri colleghi, rendendo il coordinamento necessario solo in pochi contesti.

Per quanto riguarda il DVCS è stato utilizzato Git con hosting BitBucket. Ognuno ha sviluppato le proprie funzionalità su branch differenti fino ad effettuare il merge col branch master a completamento oppure per necessità dei colleghi. Nell'utilizzo di tale strumento si è cercato di effettuare numerosi commit che esponessero significativamente i cambiamenti apportati.

- **Vanessa Di Biasi:** durante lo sviluppo del progetto ho cercato di mantenere uno stile coerente e una disposizione semplice e chiara degli elementi di View, al fine di garantire una migliore esperienza per i vari target. A livello di logica mi sono occupata di creare un unica classe che interagisse con le view e i relativi oggetti nei model, evitando così di

attribuire all'interfaccia compiti tipici del controller, oltre a predisporre un package *utility* avente lo stesso scopo e garantendo una riusabilità di codice. Infine, ho implementato le interfacce e le classi proprie del model relative a clienti, dipendenti e prodotti, cercando di rendere la loro struttura di facile manipolazione.

- **Silvia Spinelli:** mi sono occupata dello sviluppo della parte del Controller che permette di salvare e caricare i dati senza che questi vengano persi alla chiusura dell'applicazione e ho creato e gestito i file testuali. Ho implementato, inoltre, la parte di Model e View relativa alle richieste di sanificazione, in particolare ciò che riguarda la visualizzazione degli appuntamenti, la loro eliminazione e la parte di inserimento comprendente la possibilità di scelta del cliente, della data e dell'orario, che è stata in seguito integrata a quella della collega Laghi.

- **Aurora Laghi:** il mio compito è stato quello di gestire la creazione del processo sviluppando una classe nel Controller in maniera da riuscire a organizzare al meglio il lavoro senza demandare controlli, tipici di questa parte, ad altri. Ho poi sviluppato la classe di Model riguardante la creazione delle macro-fasi per le quali ho deciso di impiegare gli enum in modo da evitare di rendere ripetitivo il codice e semplificare le chiamate a funzione. Per quanto riguarda la parte di Model e View delle sotto-fasi, questa è stata successivamente integrata in una View in collaborazione con la collega Spinelli. Tale View é relativa all'inserimento di un nuovo appuntamento, la parte da me sviluppata permette la scelta delle fasi che andranno a comporre il processo, il tutto per poi calcolare sia la stima del tempo lavorativo sia quella del costo relativo all'appuntamento che si sta prenotando per un certo cliente. Quest'ultima stima verrà impiegata dal collega D'Auria nella sezione dei grafici. Durante l'intero sviluppo del codice ho cercato di utilizzare un linguaggio semplice da interpretare affinché sia più agile effettuare sviluppi futuri.

- **Nicola D'Auria:** La sezione a me assegnata tratta lo sviluppo e quindi la rappresentazione di grafici in un periodo deciso dall'utente. Utilizzando MVC e adottando inoltre il pattern strategy, il lavoro è risultato più semplice da un punto di vista logico in quanto la parte puramente implementativa del model è separata dalla parte di gestione delle interazioni dalla view che avviene nel controller. Tuttavia mi sono dedicato anche allo sviluppo di un HomeView nel modo più semplice possibile e di una View per i grafici quanto più dettagliata per fornire all'utente tutte le possibili opzioni per la visualizzazione di dati. Ho sviluppato

il controller per catturare le interazioni dell'utente con la view e per far fronte ai vari errori che possono incorrere da parte dell'utente ho ritenuto necessario lo sviluppo di Exception specifiche. Infine Il model dal punto di vista di sviluppo è stato quello più corposo poichè è stato necessario far fronte a più esigenze, tra cui dover leggere i dati da file e poterne permettere la corretta visualizzazione, fornendone un ordine e cercando di essere coerente con i dati precedentemente calcolati e salvati dalle mie colleghe Laghi e Spinelli.

## 3.3 Note di sviluppo

Il nostro lavoro ha avuto inizio con lo stabilire di comune accordo l'architettura del sistema per procedere successivamente nello sviluppo individuale delle parti di competenza. Ci siamo tuttavia resi conto di aver pensato superficialmente alla struttura completa del progetto, ciò ci ha richiesto di effettuare modifiche a funzionalità o classi che ritenevamo concluse.

### 3.3.1 Di Biasi Vanessa

- La gestione degli elementi di model mi ha permesso di utilizzare costrutti quali gli Optional. Risultato essere l'approccio migliore in particolare in fase di ricerca di un certo elemento, ad esempio all'interno delle liste di oggetti.

- Per quanto riguarda il testing automatizzato ho fatto uso di JUnit 5, il quale mi ha dato certezza della corretto funzionamento dei metodi implementati sia nel model che nel controller.

- A livello implementativo delle view ho fatto uso della libreria AWT e Swing (facendo spesso riferimento alla documentazione Oracle) e per alcune implementazioni ho invece cercato migliorie su Stack Overflow e su tutorialspoint. Un componente che ho voluto personalizzare è stato JOptionPane per creare degli Alert utili all'utente ma, dato il loro frequente utilizzo, ho preferito in un secondo momento creare una classe (presente nel package *utility*).

- In *utility* è presente anche una classe per effettuare controlli sugli input dell'utente, in particolare per clienti e dipendenti si è reso necessario verificare che i dati anagrafici corrispondessero alla realtà (Codice Fiscale, Partita IVA, Indirizzo, Telefono, Email, ecc); per fare ciò mi

sono appoggiata al metodo *Pattern.matches()* dichiarando le Regular Expression di interesse.

### 3.3.2 Spinelli Silvia

- In ogni classe del Controller che implementa l'interfaccia SaveAndLoad ho utilizzato gli Stream associati a lamba expressions per poter scorrere il file corrispondente e leggere i dati di interesse. Ciascun file, infatti, viene trattato come uno Stream di linee in cui ricercare le informazioni necessarie.

- Per il testing automatizzato mi sono avvalsa di JUnit 5, al fine di testare il corretto funzionamento dei metodi implementati.

- Per quello che riguarda la View, invece, non sono stati eseguiti test automatici poiché la verifica del funzionamento è stata eseguita manualmente tramite la GUI. A livello implementativo ho utilizzato vari componenti delle librerie grafiche AWT e Swing. Inoltre, mi sono avvalsa dei componenti DatePicker e TimePicker per la selezione di data e ora.

### 3.3.3 Laghi Aurora

- Nell'implementazione del Controller l'utilizzo di Optional è stato preferito come tipo di ritorno di metodi, ad esempio delle liste, in quanto ove necessario è stato possibile tornare come valore *Optional.empty()*.

- Per riuscire a verificare nel giusto modo i Test creati per la mia parte di lavoro ed essere sicura che questi fossero verificati correttamente, ho fatto uso di JUnit5.

- Per l'implementazione grafica ho fatto uso delle librerie AWT e Swing, in particolare da quest'ultima sono riuscita ad impiegare differenti componenti attraverso i quali l'interfaccia grafica mi è sembrata più intuitiva e veloce nelle scelte da fare.

### 3.3.4 D'Auria Nicola

- Per poter graficare i dati in modo adeguato ho fatto riferimento ad una libreria esterna XKnowmChart, ho preferito l'utilizzo di questa libreria piuttosto della classica JFreeChart poichè come prima impressione mi è sembrata più accattivante e completa in quanto a possibilità di utilizzo

fornendo ulteriori tipi di grafici e in particolare è stata la completa possibilità di personalizzazione del grafico. Inoltre per poter avere un menù a tendina relativo alla scelta delle date ho fatto riferimento alla libreria JDatePicker.

- Ho utilizzato raramente stream per poter leggere i dati raffigurati nel grafico, e quando possibile ho fatto utilizzo di lambda in modo da rendere il codice più pulito e comprensibile.

# Capitolo 4

# Commenti finali

## 4.1 Autovalutazione e lavori futuri

### 4.1.1 Nicola D'Auria

Il progetto non era di difficile implementazione, necessitava di uno studio iniziale e un po' di attenzione per quanto riguardava la logica applicata alle varie parti che dovevano comunicare tra loro. Almeno per la mia parte mi sono divertito nello sviluppare qualcosa che mi restituisse un feedback grafico, ho cercato di svilupparla al meglio per renderla il più performante possibile secondo le mie conoscenze. Tuttavia non è stato facile far fronte alle varie esigenze di tutti, si era partiti con altre prospettive e si è terminati con tutt'altro. Il tempo a disposizione era abbastanza ma non c'è stata un adeguata organizzazione. Ho visto la mia sezione dedicata snellirsi andando avanti con lo sviluppo del progetto e ho dovuto effettuare cospicue modifiche soprattutto a pochi giorni dal termine. Non c'è stata molta comunicazione almeno per quanto mi riguarda. Sicuramente si poteva fare di meglio, il gestionale aveva del potenziale non sfruttato o non capito.

### 4.1.2 Vanessa Di Biasi

Il progetto ha avuto una fase di design non particolarmente approfondita, a mio parere, ciò ha comportato alcune incomprensioni nella fase di sviluppo, in particolare sull'interazione tra le parti, le quali sono state risolte richiedendo modifiche a codice già concluso oppure aggiunte di nuove funzionalità. Per la mia parte questa problematica è stata risolta in tempi brevi e senza particolari complicazioni. Nel complesso ritengo di essere riuscita a sviluppare tutte le funzionalità da me richieste con una certa attenzione alla view e un modesto studio sulle tecniche migliori da utilizzare nel controller. Sono

soddisfatta dell'aiuto e della comprensione che si è creata tra i membri del gruppo anche se alcune problematiche sorte poco prima della consegna sono state particolarmente sentite. Per quanto riguarda il progetto in esame, ci sono miglioramenti su molti aspetti che sarebbe interessante realizzare ma oltre ciò ritengo di non avere particolare interesse nel suo utilizzo al di fuori del corso.

### 4.1.3 Aurora Laghi

Partendo dal fatto che ho sempre pensato che lavorare in un team fosse stimolante e di aiuto per la creazione di progetti, questa volta mi è risultato tutto più complicato, in parte forse anche a causa dell'emergenza sanitaria in cui ci troviamo e in parte perché la conoscenza di questo linguaggio mi risulta sotto certi aspetti complicata e poco intuitiva. Per quanto riguarda lo sviluppo del design sono stati riscontrati diversi problemi di comunicazione fra i componenti del gruppo, i quali hanno portato al necessario cambiamento del codice ormai concluso anche se, per fortuna, si sono risolti in breve tempo. Nel complesso sono soddisfatta della parte da me creata e della cooperazione che in certi momenti dello sviluppo è stata la chiave per riuscire a risolvere problemi. Con maggior tempo a disposizione si sarebbero riusciti a curare molti più aspetti al fine di creare un progetto più accattivante e sicuramente più strutturato.

### 4.1.4 Silvia Spinelli

Lo sviluppo di questo progetto è stato, dal mio punto di vista, difficoltoso, tuttavia anche utile ed istruttivo. La suddivisione dei compiti è risultata immediata ed approvata in maniera unanime, mentre la fase di design non è stata del tutto ben studiata e ha portato delle difficoltà inerenti allo sviluppo del progetto. Tuttavia, la collaborazione da parte di tutti i componenti del gruppo è stata, per quello che mi riguarda, significativa e questo aspetto è risultato fondamentale per riuscire a portare a termine il lavoro, nonostante le difficoltà sorte proprio nei momenti finali. In generale sono abbastanza soddisfatta del lavoro svolto. La difficoltà più evidente per la mia parte di sviluppo risiedeva nel fatto che fosse necessaria una costante coordinazione tra tutte le parti, consistendo il mio compito in una parte importante del Controller. Tuttavia, la buona cooperazione da parte di tutto il team mi ha permesso di lavorare ad essa con tranquillità. Un altro punto critico è stato l'uso iniziale del DVCS, ma con il tempo sono riuscita ad acquisire la padronanza sufficiente per poterlo utilizzare. Sicuramente, con un tempo più ampio e una conoscenza maggiore del linguaggio, almeno sotto il mio punto

di vista, il progetto avrebbe potuto essere sviluppato meglio, con un utilizzo più preciso di pattern, features e librerie avanzate, e un'interfaccia grafica visivamente più accattivante.

## 4.2 Difficoltà incontrate e commenti per i docenti

**Di Biasi Vanessa e Laghi Aurora** Ritentiamo di comune accordo che questo corso sia il più impegnativo della triennale. In quanto essere l'ultimo da sostenere prima della laurea possiamo ritenere che possa essere più sostenibile dagli studenti se fosse successivo ad altri insegnamenti. Ad esempio, per una migliore strutturazione degli UML e in generale della fase di design, sarebbe più efficiente avere delle conoscenze pregresse di uno dei due corsi relativi a Basi di Dati oppure Ingegneria del Software.

# Appendice A

# Guida Utente

All'avvio, l'applicazione mostrerà la HomeView, quindi la schermata principale. Da qui, si potrà accedere a una qualsiasi schermata di visualizzazione o interazione con l'utente. Da ognuna di questa sarà possibile tornare alla Home in cui è presente la possibilità, prima di uscire dall'applicazione, di effettuare il salvataggio delle modifiche tramite il pulsante "Salva ed Esci".