Relazione Progetto OOP

di

Lorenzo Maiani

Adis Dema

Indice

Analisi dei requisiti

In questa fase preliminare per la creazione del software IText, seguendo le indicazioni date dall’ingegneria del software, si vanno a descrivere i requisiti fondamentali che l’applicazione deve avere e si fornisce una descrizione del modello applicativo del software.

- Requisiti:

Il software IText mira allo sviluppo di un editor di testo digitale che consente ad un utente generico di sviluppare e scrivere un testo.

Con il termine "editor di testo" si intende un insieme di funzioni e operazioni volte alla stesura di un testo, il quale potrà essere formattato e modificato a seconda delle scelte dell'utente.

Il software dovrà soddisfare i seguenti *requisiti funzionali*:

- Il testo inserito dall'utente deve poter essere formattato a suo piacimento, offrendo quindi la possibilità di modificarne l'aspetto grafico.
Con ciò si intende che l'utente sarà in grado di modificare font-family, dimensione del carattere e colore del testo a piacimento.
Inoltre, all'utente che si interfaccia con IText deve essere fornita la possibilità di modificare l'orientamento della pagina, inteso come orientamento verticale e orientamento orizzontale.

- IText offre all'utente una sezione per la gestione delle impostazioni principali del software: alcune di queste sono la scelta del carattere principale, il tema dell'applicazione e la cartella principale nella quale salvare i documenti da lui redatti.

- Dopo che l'utente ha completato la stesura del proprio testo, potrà salvarlo grazie all'apposita funzionalità offertagli dal sistema.

- L'utente potrà scegliere di importare testo da altri file già presenti sul suo dispositivo. Inoltre, il software consente di riaprire file che sono già stati sviluppati o creati grazie all'editor per apportare ulteriori modifiche.

- Il software consente di ricercare all’interno del testo scritto singole parole.

Qui di seguito sono riportati invece i *requisiti non funzionali*:

- Si desidera che il software operi senza interruzioni nelle fasi di salvataggio del file o durante l'importazione di altri testi.

- Analisi e modello del dominio:

IText dovrà occuparsi di gestire una *sessione* al momento dell'apertura. Nella sessione l'utente dovrà poter interagire con il sistema utilizzando le funzioni che il software gli mette a disposizione: scrittura del testo, formattazione di quest'ultimo, operazioni di salvataggio e apertura di file.

Il software deve sviluppare una sezione per l'inserimento del *testo*, del quale si vuole salvare il font utilizzato che sarà sempre lo stesso durante la sua stesura.

Il testo è organizzato in *pagine*: la sessione in fase di apertura mette a disposizione una pagina all'utente con testo vuoto. Ogni pagina ha un numero massimo di righe e di colonne già predisposte. La pagina mantiene anche l'informazione sull'orientamento attuale.

La sessione si deve anche occupare di gestire le *informazioni* che ogni pagina offre. Queste possono essere sintetizzate come le caratteristiche del testo della pagina e il suo orientamento.

Ogni testo è composto da *parole*. Ogni parola può subire delle *trasformazioni*. Con quest'ultimo termine si intende tutte le modifiche grafiche a cui le parole verranno sottoposte durante la stesura del testo.

Per ogni testo che l'utente andrà a redigere, IText salverà quest'ultimo su un *file.* Ogni file sarà identificato da un nome, una data di salvataggio e avrà un percorso proprio all'interno del filesystem. Il percorso consiste quindi in una stringa testuale che localizza il file nel computer dell'utente. IText deve offrire anche la possibilità di aprire file già scritti.

IText inoltre gestisce le *impostazioni* preferenziali dell'utente le quali sono il tema principale dell'interfaccia, il font preferito e la main directory ovvero la cartella che l'utente sceglierà come predefinita per il salvataggio dei file.

Per quanto riguarda l'analisi dei requisiti non funzionali, ovvero l'uso di tecnologie apposite per limitare le interruzioni nelle fasi di salvataggio e apertura di file, si delega tale sezione a fasi successive della relazione.

La figura *1 Schema modello* riporta uno schema UML che racchiude le principali entità rilevate all'interno del sistema.

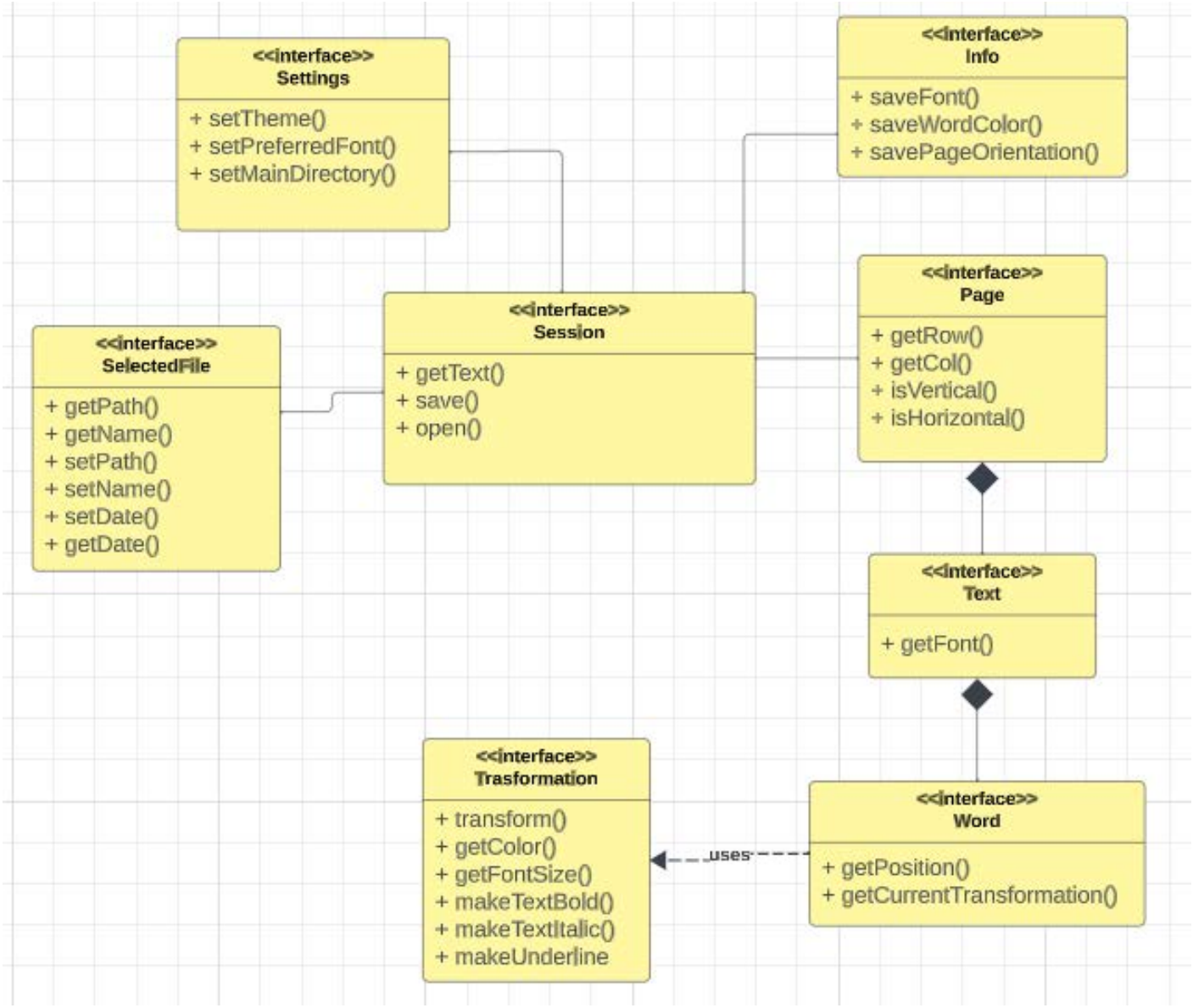


Figura 1 - schema modello di entità per IText, sono presenti le principali entità sviluppate durante l'analisi.

Design:

In questa sezione si vuole illustrare quali sono state le principali tecnologie e metodologie di progettazione utilizzate per la stesura di IText.

- Architettura:

L’architettura di IText segue a pieno il modello MVC (Model, View, Controller) nella sua forma principale. Ciò consente di suddividere l’architettura in 3 principali sezioni: la *View*, intesa come interfaccia grafica che, interagendo con l’utente, consente di recepire eventi e, nel caso specifico del software in questione, permette, ad esempio, l’inserimento del testo e l’uso delle principali funzionalità; i *Controller*, componenti il cui compito sarà quello di eseguire determinati compiti sulla base delle interazioni con la View; il *Mode*l, componente che usa come entry point le entità analizzate nella sezione “Analisi e modello applicativo”.

IText, all’apertura, deve fornire una sessione che svolge il ruolo di controller per tutto quello che riguarda la messa in atto delle principali funzioni che il software offre. E’ possibile vederlo come il controller principale per l’applicativo.
Il software sviluppa 2 principali interfacce grafiche (viste): la prima consentirà all’utente di inserire il testo principale, interagendo anche con la sezione per la formattazione del testo; la seconda dovrà gestire le impostazioni. La sessione, quindi, andrà a richiamare a sé molti altri sotto-controller che svolgeranno, andando ad interagire con i modelli previsti, compiti più specifici.

A seguito di un’interazione con l’interfaccia grafica**,** IText prevede la registrazione di un testo e di tutte le sue sotto parti già descritte nel modello di analisi. Sarà quindi compito del controller di sessione andare a prelevare l’organizzazione del testo e le sue principali caratteristiche dalla vista, andando a inserire quest’ultimo nel suo apposito model e salvando le informazioni presenti sul testo, per poi accedervi in momenti successivi al fine di salvare tali modifiche.

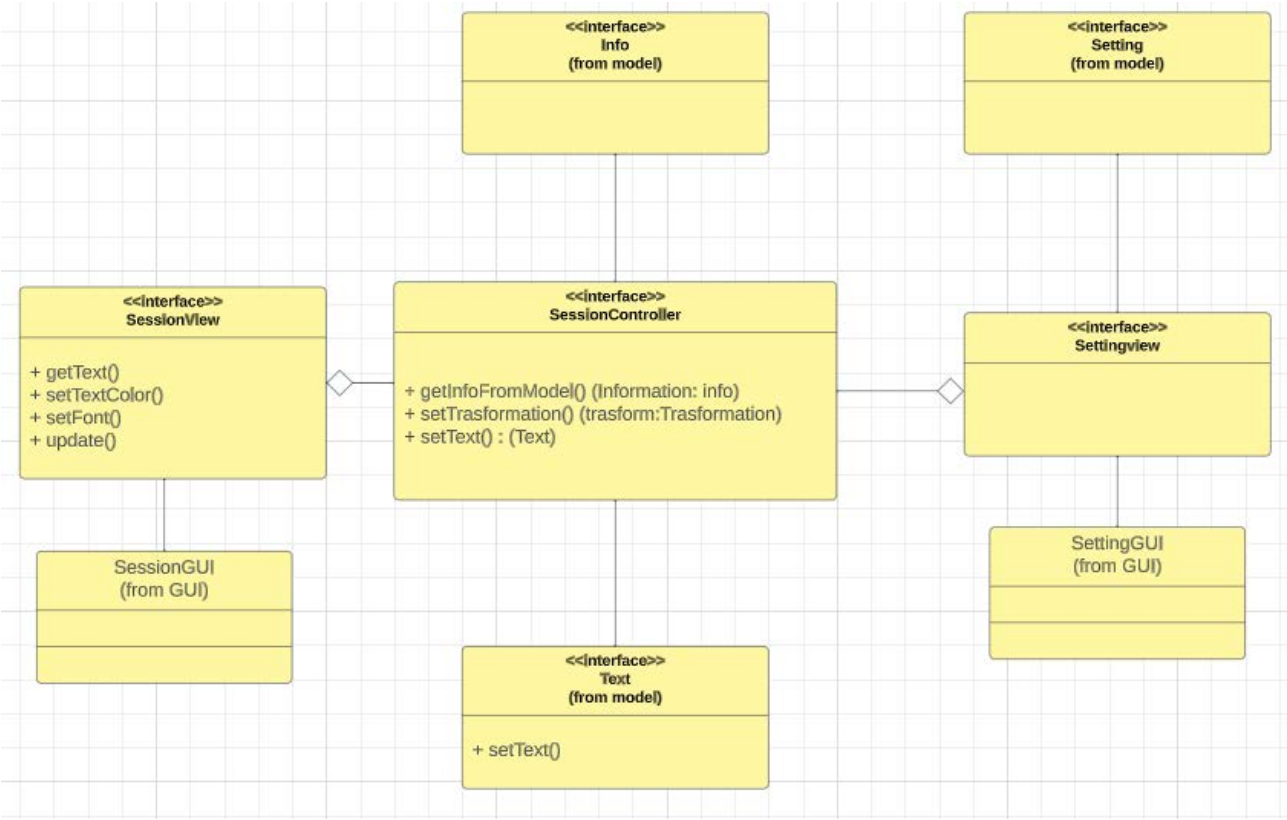


Figura 2- Architettura

Poiché i principali componenti dell'architettura sono ben distinti gli uni dagli altri, come sopra viene descritto e da come può essere dedotto dallo schema riportato, la modifica di una o più viste, intese come componenti View, non va a contrastare il corretto funzionamento degli altri 2 componenti e non implica considerevoli nuove aggiunte.

Nuove funzionalità che potranno essere introdotte nei principali controller andranno ad intaccare la vista, ma solo in minima parte. Infatti occorrerà aggiustare l'interfaccia al fine di poter gestire l'input di un ipotetico utente e comunicarlo al controllore.

Altri aspetti e interazioni più specifiche verranno approfondite durante il design di dettaglio.

- # Design dettagliato

In questa sezione si vuole ulteriormente descrivere la fase di progettazione di IText. Qui di seguito, infatti, sono analizzati i principali problemi e le corrispondenti soluzioni che rappresentano le funzioni che IText dovrà svolgere durante il suo utilizzo. Si vuole quindi sottolineare come tali problemi siano stati affrontati dagli sviluppatori e spiegare le motivazioni delle decisioni prese.

Sono riportate le 2 sezioni proprie per ciascuno studente:

- Lorenzo Maiani:

Come illustrato nell'architettura, IText deve soddisfare una larga gamma di funzioni ed operazioni che interrogano in svariati modi tutti i componenti del pattern MVC.

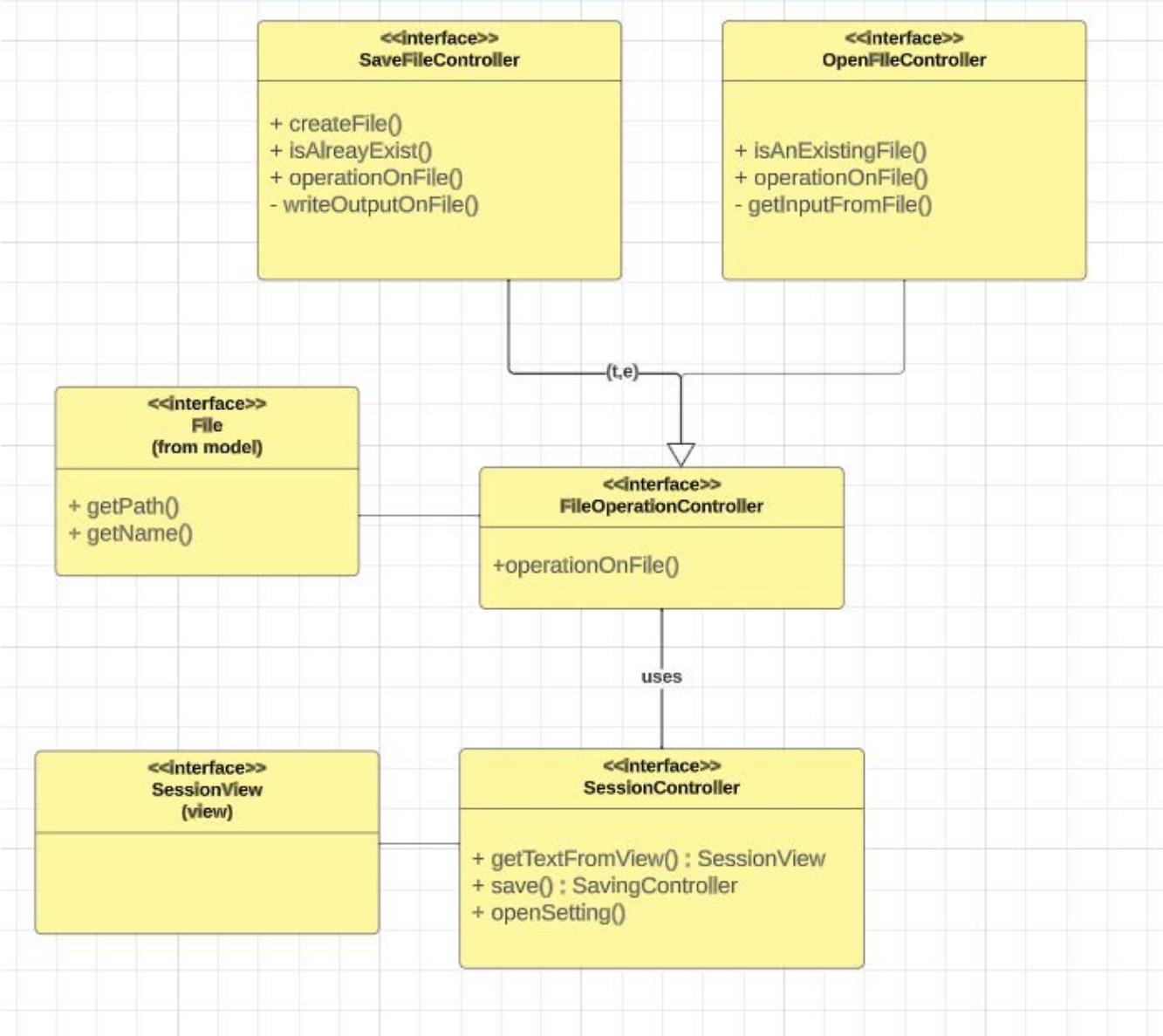
Si è mostrato, tramite la *"Figura 2 - Architettura"*, che IText ha un controllore principale, chiamato SessionController, che richiama a sé sotto-controller al fine di svolgere le principali operazioni.

Un esempio di tali sotto-controller si ritrova nella scelta di utilizzare un FileOperationController, il cui compito sarà quello di inizializzare le comunicazioni con file persistenti all'esterno dell'applicazione. Tale controllore vuole anche essere interfaccia implementabile per sottoclassi, quali FileSaveController e FileOpenController.
Per risolvere quindi il problema dell'interazione con i file esterni al software, si vuole utilizzare il pattern *Strategy*.
Il FileOperationController offrirà un metodo pubblico denominato *operationFile()* il cui compito verrà poi descritto nella classi che implementeranno tale strategia e andranno a sviluppare un metodo proprio. Ad esempio il salvataggio andrà a svolgere tutte le funzioni, quali creazioni di file e salvataggio.

Di seguito viene illustrato uno schema riassuntivo per la parte precedentemente analizzata e un diagramma UML.

Problema: IText deve poter essere in grado di salvare ciò che l'utente ha scritto su file persistenti, i quali potranno anche essere riaperti in seguito e modificati.

Soluzione: Utilizzo del pattern Strategy per sviluppare un'interfaccia FileOperationController che farà da contratto per le sue implementazioni, mettendo a disposizione il metodo operationFile (). Le sue implementazioni avranno il compito di specificare quale sarà il corretto funzionamento di tale metodo.



Un altro problema che si vuole analizzare consiste nel come gestire la sezione delle impostazioni e come tale modello posso essere sviluppato.
Dal testo si deduce che l'utente possa andare a modificare la sezione delle impostazioni a suo piacimento modificando il tema del software, la directory principale e il font base. Queste sono informazioni che tutto il sistema condivide, non solo alcune sezioni. Si supponga che in fase di salvataggio si voglia conoscere quale è la directory principale

scelta dall'utente, oppure che all'apertura dell'applicazione, la sessione debba modificare il font e il tema.

Per risolvere tale problema si può optare per 2 diverse soluzioni:

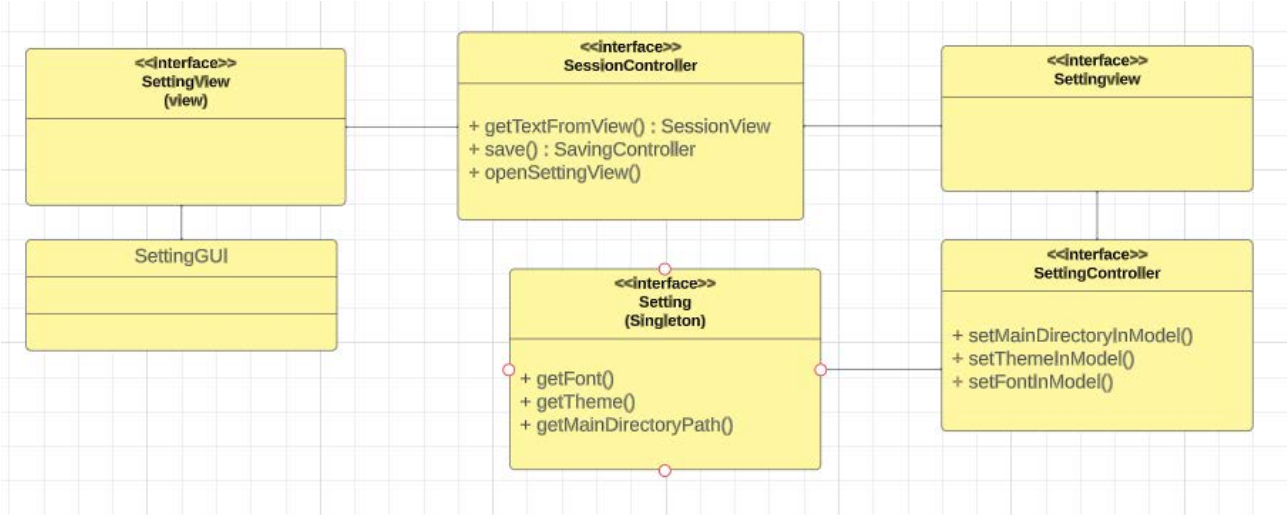
- La prima segue il pattern *Singleton*, il quale rende unico in tutto il programma la classe Impostazioni e consente quindi a chiunque lo necessiti di andare a controllare quale è il suo stato tramite i metodi messi a disposizione.
- La seconda, ovvero l'utilizzo del pattern *Observer* (Osservatore), permette di attendere, o meglio osservare, possibili cambiamenti che possono accadere alle impostazioni ed agire di conseguenza su coloro che osservano.

  Si può offrire un'alternativa a questo secondo punto. Infatti la vista della sessione sarà gestita da un controller apposito che, a seguito dell'interazione dell'utente, modifica il modello. Tale controller potrebbe poi chiamare metodi di altri controllori per modificare la grafica tramite metodi appositi (update() del SessionView ad esempio). Ciò significherebbe creare vari canali di comunicazione tra i controllori e ciò potrebbe rendere il sistema caotico e confusionario in fase di implementazione.

Siccome le impostazioni sono accessibili da molte sezioni, come già descritto, si vuole optare per una soluzione mista. Infatti la classe impostazioni sarà un Singleton accessibile da chiunque lo necessiti per conoscere le informazioni riguardanti il font principale e la main directory. Sarà utilizzato il pattern Observer per quanto riguarda la modifica del tema. La sessione in questo caso svolgerà il ruolo di Osservatore.

Problema: le impostazioni dovranno essere accessibili dalla sessione in qualsiasi momento per estrapolarne il contenuto.

Soluzione: si adotta il pattern Singleton per rendere unica l'istanza delle impostazioni, così da poter accedervi senza passare costantemente l'oggetto tra le varie classi che lo utilizzeranno. Per la questione del tema, si vuole realizzare il pattern Observer, creando un osservatore all'interno della sessione che, alla modifica del tema, si occupi di modificarne la grafica.



Durante la fase di analisi, ci si è accorti di quanto sia necessario gestire diversi tipi di *informazioni* durante la sessione.
Ciò consente, per esempio, di mantenere salvato il nome del file aperto, di salvare le informazioni sulle trasformazioni delle parole interne al testo e il font di quest'ultimo.

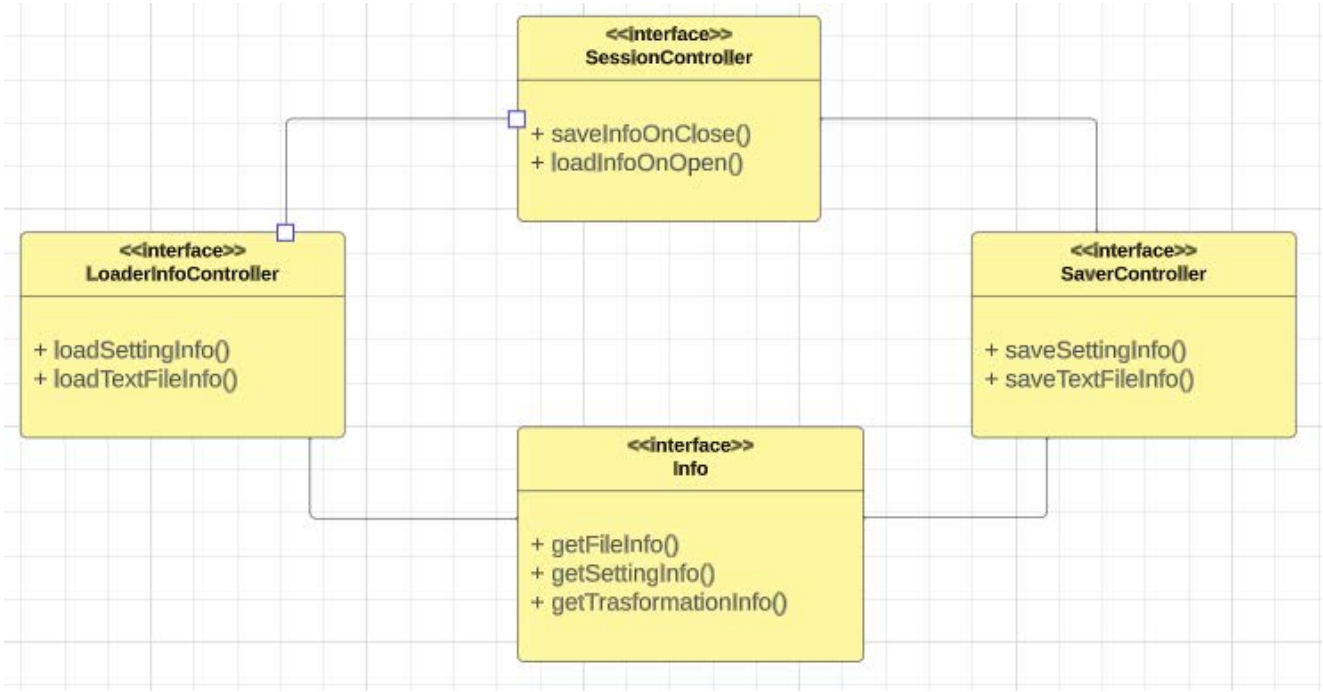
Questa componente risulta fondamentale poiché rappresenta l'entry point di tutte le informazioni all'apertura del software. Per fare un esempio, a seguito dell'apertura di un nuovo documento da parte dell'applicazione, occorrerà caricare le informazioni per quel determinato testo le quali saranno salvate su file appositi. Tutti i dati che verranno letti in fase di apertura, saranno salvati nelle informazioni.

Occorre quindi che tali informazioni possano essere salvate sia a seguito di variazioni apportate dall'interazione dell'utente con l'applicazione sia nel momento in cui si vuole creare un nuovo file.

Le informazioni che occorrono per il salvataggio si possono trovare nei modelli espressi precedentemente, ovvero sul modello dei File, su quello delle Impostazioni e su quello delle trasformazioni sulle parole.
Non occorre quindi creare una nuova sezione apposita che vada a mantenere le informazioni dettagliate, intesa come una gerarchia di informazioni dove le sottoclassi non sono altro che "copie" dei modelli.
Potrebbe essere invece una buona idea e soluzione gestire una interfaccia singola, denominata Informazioni, che collezioni i vari dati in un momento specifico e, tramite un controllore, vada a salvare su file persistenti le informazioni.

A tali informazioni potranno accedere controller differenti durante lo sviluppo del software, ad esempio il SessionController il quale, al caricamento di un testo, dovrà precedentemente aver già ottenuto le informazioni che definiranno le trasformazioni che questo subirà.

Occorrerà quindi predisporre un Loader e un Saver per le informazioni laterali all'esecuzione del software.



L'ultima sezione che si vuole trattare riguarda la grafica. IText si pone l'obiettivo di offrire all'utente 2 principali GUI (o view per il pattern MVC): una schermata principale e una di impostazioni. Per quanto riguarda la seconda, essa è già stata trattata precedentemente nella apposita sezione; si vuole ora descrivere come e da quali principali componenti verrà composta la prima.
La SessionView, interfaccia con il compito di ricevere i vari input dall'utente e di comunicarli al SessionController, fa riferimento alla SessionGUI la quale offre tutti i componenti grafici: area per il testo, pulsanti per la modifica di quest'ultimo e altri per eseguire specifiche funzioni.
Sarà compito della SessionView andare a captare tali segnali mandati dalla GUI e richiamare a sua volta il SessionController per lo svolgimento dei compiti.
Con ciò si risolve il problema, espresso in analisi, di sviluppare un'interfaccia grafica per l'utente e di realizzare una barra rapida per l'esecuzione delle principali operazioni.

- Adis Dema:

Come mostrato nella figura del design di architettura IText deve essere in grado di soddisfare diversi sotto-controller.

Un esempio di tale sotto-controller si ritrova nel textTransformerController il quale Principale compito sarà quello di comunicare le modifiche avvenute nel testo.

Tale controller vuole essere in grado di fare delle modifiche su parole. Per fare ciò come sarà riportato sullo schema UML è stato utilizzato una combinazione di pattern Bridge, Builder e MVC.

1.BRIDGE PATTERN

Scopo:

Il Bridge pattern è stato utilizzato per separare l'astrazione dalle implementazioni, consentendo di variare in modo indipendente

Utilizzo:

Ho definito un'interfaccia WordTransformer per le trasformazioni sulle parole, consentendo di creare diverse implementazioni concrete come ColorTrasform, Bold Trasform ecc.…

Ho seguito lo stesso approccio per le trasformazioni sul testo completo, con un'interfaccia TextTransformer.

Questi due insiemi di implementazioni (trasformazioni sulle parole e trasformazioni sul testo) possono essere variati separatamente e combinate liberamente, ad esempio, puoi applicare una trasformazione sulle parole a un testo composto da più parole.

2.Builder Pattern:

Scopo:

Il Builder Pattern è stato utilizzato per creare oggetti complessi passo dopo passo, consentendo di costruire configurazioni complesse in modo più leggibile e mantenibile.

Utilizzo:

Hai creato due builder: WordTransformationBuilder per le trasformazioni sulle parole e TextTransformationBuilder per le trasformazioni sul testo completo. I builder permettono di aggiungere trasformazioni in modo sequenziale e quindi costruire un oggetto complesso. Inoltre, ho utilizzato il builder per combinare diverse trasformazioni in un'unica funzione di trasformazione.

3.MVC (Model-View-Controller): pattern strutturale

Scopo: Il pattern MVC separa il codice in tre componenti principali per garantire la separazione delle preoccupazioni e la facilità di manutenzione.

Utilizzo:

Ho creato il TextTrasfomrModel come modello che rappresenta i dati dell'editor di testo e include campi per il testo, lo stato del font, le opzioni di formattazione, il conteggio delle parole, ecc.

Il TextTrasformController è responsabile di gestire gli aggiornamenti del testo, delle trasformazioni e del font. Ho utilizzato il controller per aggiornare il modello quando il testo o il font cambiano, inoltre funge anche come intermediario tra il TextTrasformModel e SessionController.

Un'altra opzione per l’utente deve essere anche quella di cambiare l’orientamento della pagina. Per fare ciò la pagina è stata progettata come una tabella con righe e colonne. Per fare ciò è stato utilizzato un pattern MVC.

Scopo:

Il pattern MVC separa il codice in elementi diversi per facilitare la manutenzione del codice.

Utilizzato:

Ho creato il PageModel come modello che rappresenta il numero delle righe e delle colonne.

Il PageController è responsabile di cambiare l’orientamento della pagina, inoltre funge anche come intermediario tra il PageModel e SessionController.

L’utente non sarà in grado di decidere lui le dimensioni della pagine, siccome il numero di righe e colonne sarà dato senza la possibilità di cambiarle. La funzionalità del cambiamento dell’orientamento della pagina dal verticale(default) a quella orizzontale è stata reso possibile grazie al cambiamento deli valori delle righe con quelle delle colonne.

Sviluppo:

In questa sezione si vogliono analizzare alcune sottosezioni che riguardano lo sviluppo di IText.

- Testing automatizzato

IText offre una sezione apposita di testing utilizzando la suite JUnit, la quale consente di provare le principali funzionalità del software in maniera totalmente automatizzata.

Si vuole quindi sottoporre a testing le seguenti funzionalità:

- Dato un testo fittizio, creare un file, scrivere dentro a quel file.

- Aprire un dato file e prenderne il contenuto, confrontarlo con il testo fittizio: dovranno corrispondere.

• Metodologia di lavoro

Nella seguente sezione si analizza come si è proceduti con lo sviluppo, evidenziando le sotto parti che si sono svolte in maniera autonoma.

• Lorenzo Maiani:

Come descritto nella sezione di design, i principali compiti che dovevo implementare sono: la creazione delle GUI e la gestione dei loro eventi, la gestione del salvataggio e apertura di nuovi file dal filesystem, la creazione delle impostazioni riguardanti applicazione, tra cui il tema, la main directory e il font preferito, la gestione delle informazioni inerenti sia all'applicazione che al testo attualmente aperto e l'acquisizione di testo da altri file.

Il progetto è stato strutturato in diversi package che rispettano il pattern MVC. Si sono quindi inseriti package generali per model, view e controller con all'interno sotto package più specifici. Il package *utils* è stato aggiunto al fine di contenere classi ed enumerazioni utili per la gestione del software, come ad esempio NumericConstants e StringConstats che contengono rispettivamente le dimensioni dell'interfaccia e i principali percorsi ai file.

Mi sono occupato della realizzazione delle 2 principali GUI, sviluppate tramite l'utilizzo di SceneBuilder e inserite all'interno della directory layout nella sezione "resource". Dopo aver fatto ciò, mi sono dedicato allo sviluppo delle View, intese come classi che hanno il compito di intercettare eventi generati dall'utente e di conseguenza chiamare in causa specifici controller.

E' questo il caso delle classi SessionViewImpl e SettingViewImpl.

Completata la gestione degli eventi, mi sono dedicato allo sviluppo dei principali controllori del sistema. Ho iniziato con il SessionController, controller principale per tutta l'applicazione, creando l'interfaccia che descrivesse il suo "contratto", ovvero i metodi principali.

Successivamente ho sviluppato tutta la sezione del salvataggio, apertura e creazione di un nuovo file.

I controller messi in atto per questa sezione sono: FileOperationController, SaveFileController ed OpenFileController. Come visto anche in design, il primo definisce una principale operazione che poi verrà implementata nella classi specifiche e verrà richiamata dal Save o Open controller a seconda dell'operazione richiesta

dall'utente.
Nel FileOperationController ho deciso di utilizzare, come parametro e valore di ritorno dell'unico metodo, i generici. Tale scelta è stata fatta poiché le classi implementative dovranno ritornare diversi valori e avranno tipi di parametri differenti. Un'alternativa che si era presa in considerazione era l'utilizzo della classe Optional. Il salvataggio avviene con la creazione di una directory con il nome del file nel percorso scelto dall'utente. Al suo interno si troverà il file .txt, apribile dall'applicazione, ed un file .ini, che verrà trattato in seguito.
Il salvataggio del testo è impostato di base come salvataggio di un documento di testo (.txt), ma ho implementato anche la possibilità di salvare tale file in altri formati come quello .pdf.

L'apertura di un file avviene dopo che l'utente ha selezionato il file nell'apposita grafica. Il software supporta solo l'apertura di file di testo e nessun altro formato può essere selezionato grazie ad appositi filtri.
Il testo, sia in fase di salvataggio che in fase di apertura è codificato con la codifica carattere UTF-8. Si è scelto di agire in questo modo al fine di poter offrire una più vasta gamma di caratteri rispetto a quello di default offerto da JavaFx.

Dopo aver testato la parte precedentemente descritta, mi sono concentrato in autonomia su come implementare la porzione dei Setting.
Già in fase di design ci si era accorti di quanto fosse importante che i Setting fossero accessibili a tutta l'applicazione in qualsiasi momento.
Tale scelta ha portato a sviluppare la classe SettingImpl seguendo il SingletonPattern. La classe ha un costruttore privato che inizializza i campi della classe con valori di default per poi essere ottenuto tramite il metodo statico *getInstance()*.
Al suo interno è contenuto lo stato della directory principale che verrà visualizzata sia in fase di salvataggio che in fase di apertura di un nuovo file, il font principale con il quale i testi verranno scritti e il tema dell'applicazione.

Riguardo al tema, come accennato in design, si ha l'intenzione di implementare la gestione del cambio di quest'ultimo utilizzando un Observer. Si sono valutate diverse opzioni implementative consultando sul web e sull'apposita documentazione e ho deciso di svilupparlo nel seguente modo:

- La classe SettingImpl ha al suo interno un campo privato di tipo *PropertyChangeSupport.* Sono stati definiti 2 metodi per aggiungere e rimuovere i listener all'oggetto sopracitato.
- Alla chiamata del cambio di tema tramite il metodo setAppTheme () di Setting, il support segnala a tutti i listener che qualcosa è cambiato.
- Le classi che desiderano essere notificate hanno implementato il *PropertyChangeListener* che consente di stare in ascolto della modifica.
- Il tema viene cambiato caricando e rimuovendo un file .css dalle resource.

Ho anche aggiunto alcuni elementi che fanno da corredo al corretto svolgimento dell'applicazione, ad esempio ho gestito la chiusura dell'applicazione andando a mostrare un AlertDialog con ButtonType personalizzati che vadano a richiedere all'utente, in caso di cambiamento del testo, se si vuole salvare prima di chiudere e, in caso affermativo, eseguono la procedura di salvataggio, in caso negativo, invece, chiudono l'applicazione.

Infine ho implementato la porzione di codice riguardante le informazioni. Queste, come detto in fase di design, devono mantenere i dati riguardanti il file aperto, le impostazioni e tutto ciò che riguarda il testo. Perciò si è creata una classe Info la quale ha, al suo interno, campi privati che riguardano i dati sopraelencati.

Le informazioni sono quindi il punto d'incontro principale tra il mio lavoro e quello del mio collega, poiché dovrò accedere alla sua porzione di codice per estrapolare le informazioni necessarie.

Una volta ottenute le informazioni, queste andranno salvate alla chiusura dell'applicazione per quanto riguarda i Setting, mentre alla chiusura del file per tutto ciò che riguarda lo stile del testo.
Una situazione analoga la si trova in fase di apertura dell'app o di un file, ma occorrerà caricare talune informazioni.
Per fare ciò, sono state implementate 2 classi SaverImpl e LoaderImpl le quali rispettivamente salvano/caricano le informazioni su/da file persistenti.
In ciascuna classe sono stati pensati due metodi differenti per le operazioni di carico e scarico delle informazioni su impostazioni e su file. Ciò è dato dal fatto che si vogliono creare differenti file persistenti.
Il metodo per il salvataggio delle impostazioni crea una directory nel percorso "user.home/Documents" dell'utente chiamata ITextSetting e all'interno salva un file con estensione ".bin". Per quanto riguarda il salvataggio delle informazioni sul file, queste vengono inserite nella cartella del salvataggio del file stesso all'interno di un file .ini.

Per sviluppare il software in maniera collaborativa, abbiamo utilizzato Git come DVCS. L'idea iniziale era quella di lavorare con un repository administrator sviluppando su branch separati utilizzando poi la funzione di merge per unificare i lavori e creare il prodotto finito.
Essendo il gruppo formato da soli 2 componenti, si è però scelto di optare per una soluzione più comoda e rapida, dove uno dei due studenti si occupa della creazione del repository pubblico consentendo all'altro collega di effettuare le funzioni "push" e "pull" e sviluppando così il codice in maniera ben strutturata.

- Note sullo sviluppo

Qui di seguito sono riportare le informazioni su alcune sezioni particolari del codice, ognuna suddivisa per ogni componente.

- Lorenzo Maiani:

  - Uso di JavaFx in combinazione allo strumento SceneBuilder per la creazione delle interfacce grafiche e l'uso della libreria anche nel codice per modificare, aprire e chiudere sottosezione della GUI

  - Uso di "ITextPDF" come libreria esterna per il salvataggio del testo in formato PDF. Se ne fornisce un esempio al seguente link:

    https://github.com/lorenzomaiani/OOP22-itxt/blob/75e07f342a47942c93bbd68eeb04ebe4547d9fab/IText/src/main/java/org/example/controller/file/SaveFileControllerImpl.java#L94

  - Uso di lamba nella forma methd reference, di seguito riportato un esempio link: https://github.com/lorenzomaiani/OOP22-itxt/blob/75e07f342a47942c93bbd68eeb04ebe4547d9fab/IText/src/main/java/org/example/view/sessionview/SessionViewImpl.java#L70 e nel metodo *initTextAreOnChangeMethods ()* nella forma più comune. Link: https://github.com/lorenzomaiani/OOP22-itxt/blob/75e07f342a47942c93bbd68eeb04ebe4547d9fab/IText/src/main/java/org/example/view/sessionview/SessionViewImpl.java#L197

  - Uso della classe Stream, per generare un lista che contiene le possibili dimensioni del carattere.

    Permalink: https://github.com/lorenzomaiani/OOP22-itxt/blob/75e07f342a47942c93bbd68eeb04ebe4547d9fab/IText/src/main/java/org/example/view/sessionview/SessionViewImpl.java#L69

  - Progettazione con l'uso dei generici per quanto riguarda il FileOperationController.

    Link: https://github.com/lorenzomaiani/OOP22-itxt/blob/75e07f342a47942c93bbd68eeb04ebe4547d9fab/IText/src/main/java/org/example/controller/file/FileOperationController.java#L

Per quanto riguarda la mia sezione non ho utilizzato né pezzi di codice provenienti da altri progetti del corso di OOP né classi e metodi copiati da internet. Usando per la prima volta la libreria JavaFx per lo sviluppo grafico, mi sono ispirato a tutorial online su come creare interfacce grafiche reattive e su come strutturare il progetto in maniera efficiente.

Commenti finali:

- Autovalutazione e lavori futuri:

Qui di seguito è riportata la mia sola valutazione, poiché il collega non ha potuto scrivere la propria.

- Lorenzo Maiani:

All'inizio del progetto, la comunicazione e il confrontarsi con il mio collega è risultato fondamentale, poiché era la prima volta per entrambi che ci si accingeva ad un progetto ed ad una sfida come questa.

Durante la fase di progettazione, assecondando anche il fatto che il mio collaboratore avesse un impiego esterno nel frattempo, mi sono occupato io principalmente della stesura della relazione e della creazione degli schemi, cercando di arricchire la mia sezione aggiungendo schemi UML e Pattern di design.

Durante la fase di sviluppo ho cercato di utilizzare al meglio tutte le risorse acquisite sia nel corso di OOP che negli altri corsi durante la triennale e specialmente ho cercato di integrare nel progetto elementi conosciuti durante il tirocinio.

Nel complesso ritengo di aver svolto un buon lavoro, portando avanti un progetto quasi interamente in autonomia, e occupandomi di studiare da zero nuovi costrutti come JavaFx del quale conoscevo solo poche nozioni.

Punti di debolezza nella mia sezione di progetto si possono trovare nella struttura di quest'ultimo, anche se credo di aver seguito abbastanza il pattern architetturale MVC previsto dal design, e forse una scarsa ottimizzazione del codice, nel quale potrebbero esserci delle ripetizioni, nonostante io abbia cercato di limitarle il più possibile.

Nel complesso ritengo sia stata creata una discreta applicazione, non troppo complessa date le circostanze, ma concorde con l'idea "poco ma buono".

Il software potrebbe essere ampliato completando la sezione del mio collega, sviluppando una buona gestione del testo e ampliando alcune funzionalità su quest'ultimo. Anche un salvataggio in Cloud, ad esempio OneDrive, potrebbe essere una delle funzionalità da aggiungere.

- Difficoltà incontrate e commenti per i docenti

Lorenzo Maiani:

La principale problematica che ho riscontrato riguarda l'elaborato. Il corso è uno dei migliori in tutto il triennio, ma introduce una grande quantità di argomenti, dal momento che il mondo ad oggetti è vastissimo. Di conseguenza molti studenti non hanno la possibilità di acquisire al meglio i contenuti proposti, come ad esempio Gradle e a che cosa serve, GIT e il suo funzionamento. Per quanto riguarda, invece, l'esame di laboratorio, ritengo che sia ben tarato sui 12 CFU: esso affronta tutti gli aspetti visti a lezione e, sebbene a volte possa risultare molto complesso, con una buona preparazione è possibile ottenere risultati soddisfacenti.

A proposito dell'elaborato, sinceramente penso che il regolamento ponga alcuni vincoli eccessivamente limitanti, tra cui il numero minimo di componenti di ciascun gruppo. Mi sono trovato personalmente nella situazione di dover attendere per mesi la risposta di un compagno per poter creare un gruppo, a cui non si sono poi aggiunti ulteriori partecipanti. Altri corsi come quello di Basi di Dati prevedono di svolgere un elaborato, il quale però può essere realizzato in singolo o in gruppo con un numero di componenti variabile tra 1 e 4. Ciò implica un livello di difficoltà incrementale in base al numero di studenti e, allo stesso tempo, offre a ciascuno la possibilità di svolgere il progetto autonomamente se egli lo desidera.

Comprendo, d'altra parte, la vostra intenzione di stimolare la collaborazione tra gli studenti e di simulare un lavoro coordinato come quello che effettivamente sussiste tra ingegneri nel mondo del lavoro. Tuttavia, credo sia difficile da realizzare, dal momento che non è previsto l'obbligo di frequenza nei laboratori e gli studenti potrebbero avere obiettivi differenti.

Dal mio punto di vista, delle possibili soluzioni per rendere il corso più accessibile e comprensibile potrebbero essere quella di suddividerlo in più moduli, per consentire agli studenti di apprendere al meglio tutte le nozioni fondamentali della progettazione e programmazione ad oggetti, oppure quella di farli collaborare già durante i laboratori in gruppi, affinché possano iniziare a conoscersi e a creare gruppi di lavoro più uniti e con obiettivi comuni.

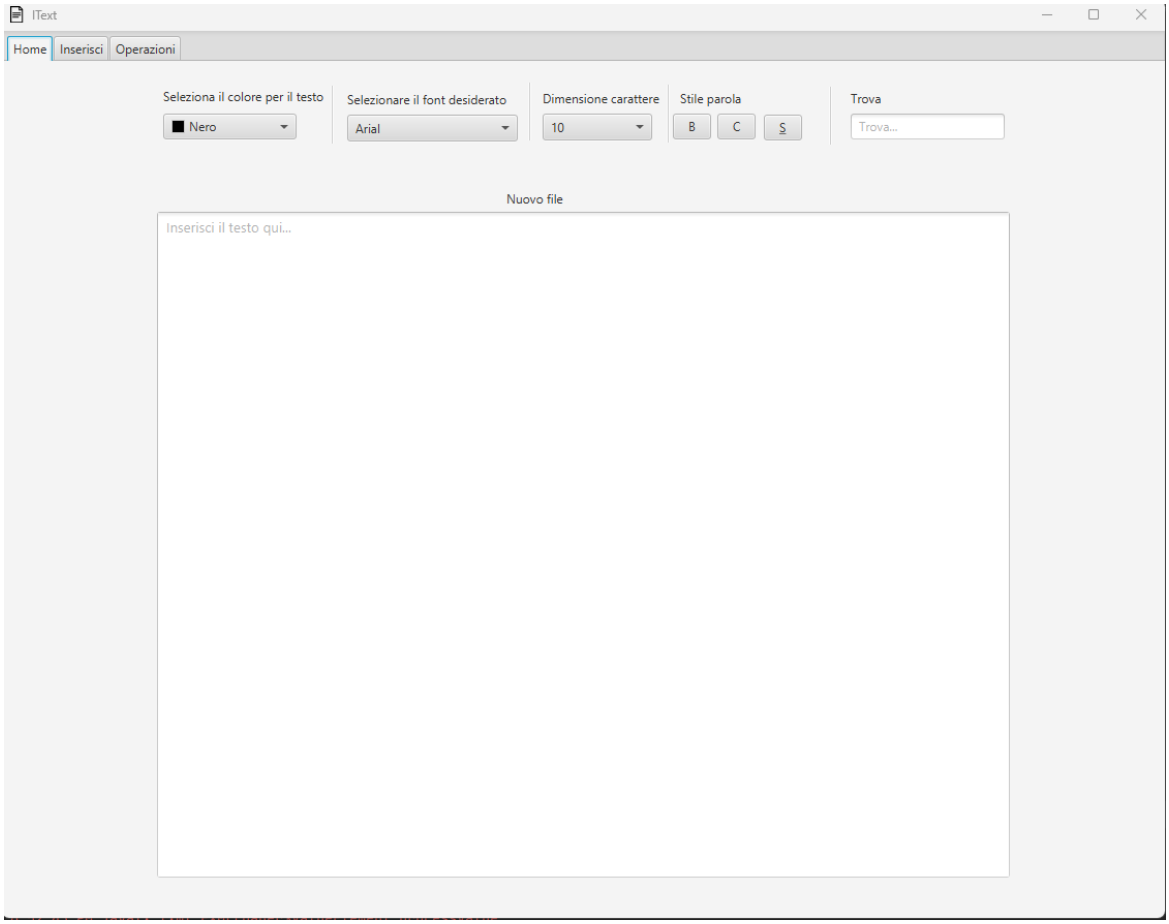
Appendice A:

Guida utente:

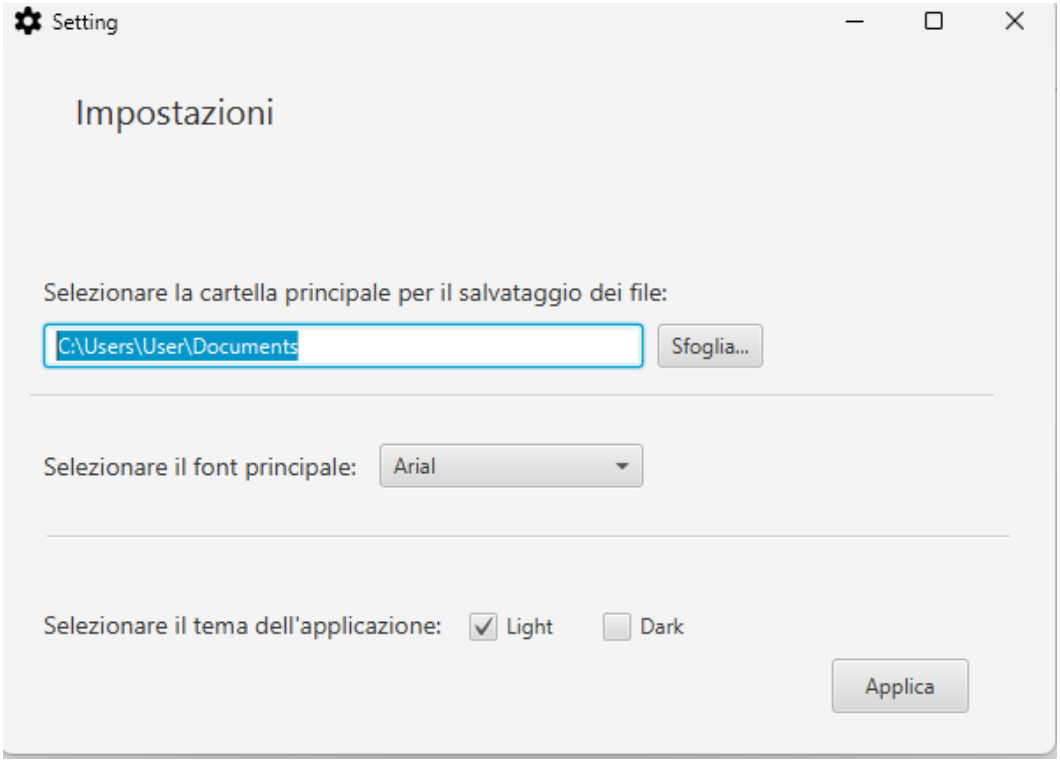
Per avviare il software occorre eseguire il file IText.jar contenuto all'interno della main directory del progetto.

Il software è suddiviso in 2 schermate principali: l'homepage, dove l'utente può inserire il testo e apportare tutte le modifiche che desidera su quest'ultimo e le impostazioni, le quali configurano le variabili di sistema.

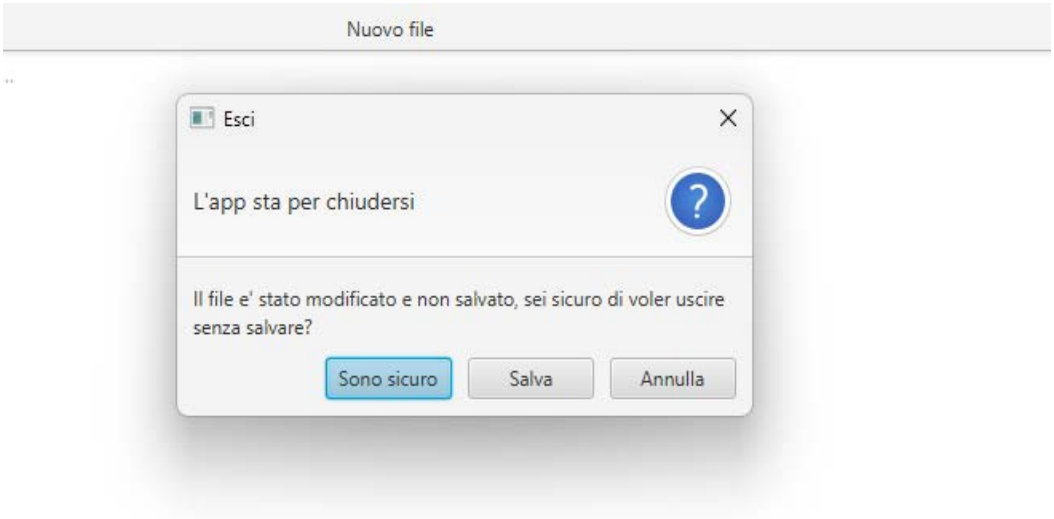
Nell'homepage è stata predisposta una barra degli strumenti rapidi nella parte superiore della schermata. Al suo interno sono state sviluppate 3 sotto sezioni: Home, che contiene i comandi per la modifica del testo, Inserisci, che contiene funzionalità per aggiungere elementi al testo provenienti all'esterno dell'applicazione, e una sezione Operazioni, le cui funzionalità consentono di salvare, aprire e creare nuovi file. Inoltre quest'ultima apre anche la sezione delle impostazioni.



Per quanto riguarda le impostazioni, troviamo una schermata che consente di modificare le variabili di sistema. Con questo termine si intende la gestione e il controllo da parte dell'utente di scegliere la locazione desiderata per il salvataggio dei propri file, la scelta di un tema dell'applicazione e il font principale con il quale il testo verrà scritto (ovviamente modificabile in seguito).



Alla chiusura dell'applicazione ed all'apertura di un nuovo file, se l'utente ha modificato il testo e non ha effettuato alcun salvataggio, il software chiede conferma per procedere con la funzionalità scelta tramite un pop-up a schermo. Se l'utente desidera procedere, allora deve selezionare "Sono sicuro".



Appendice B:

B.01 **lorenzo.maiani3@studio.unibo.it**