Smart Station

Il gestionale delle stazioni di servizio

Elaborato per il corso di

Programmazione ad Oggetti

Corso di laurea in

Ingegneria e Scienze dell'Informazione

ANNO ACCADEMICO 2015-2016

Marcin Pabich

Matricola: 0000 718536

Matteo Michelotti

Matricola: 0000 745958

Mami Alessandro

Matricola: 0000 733218

Sommario

1 – INTRODUZIONE

Il progetto "Smart Station" consiste nella realizzazione di una applicazione per la gestione di una fantomatica stazione di servizio.

L'applicazione è stata realizzata in modo da facilitare la sua gestione: essa verrà messa a disposizione su una macchina (indipendentemente dal sistema operativo, purché compatibile con Java 8), permettendo all'amministratore di effettuare operazioni di monitoraggio, inserimento, modifica e cancellazione relative alla stazione.

L'applicativo permetterà di gestire le principali tematiche relative all'amministrazione di un'area di servizio offrendo una esperienza utente piacevole e semplice anche per utenti poco esperti.

La realizzazione del progetto è stata effettuata utilizzando Eclipse *(Mars 2)*, applicando il modello di sviluppo JavaFX, attraverso l'uso di FXML e CSS per la parte grafica e librerie Java per la parte del codice.

2 – ANALISI

2.1 – Requisiti

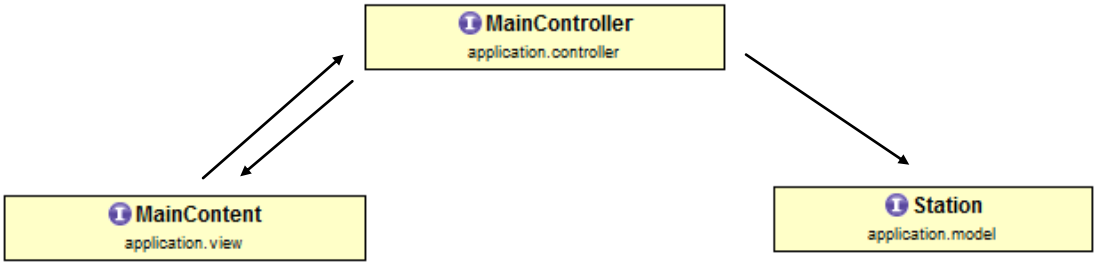
Un gestionale dev'essere completo di tutte le funzionalità relative al proprio ambito, ma anche semplice e immediato nell'utilizzo per poter velocizzare e non ostacolare (o rallentare) la gestione. Una persona addetta alla gestione, chiamata amministratore, sarà colei che gestirà le diverse questioni riguardanti la stazione stessa. Egli deve poter:

- Visionare l'intera area relativa ai distributori,
- Modificare, inserire, cancellare le diverse aree,
- Modificare, inserire, cancellare diversi tipi di carburante,
- Modificare, inserire, cancellare diversi tipi di riserve relative ai carburanti,
- Modificare, inserire, cancellare diversi tipi di pompe,
- Visionare e gestire in modo semplice le entrate e le uscite della stazione.

2.2 – Analisi e modello del dominio

Smart Station dev'essere in grado di gestire diverse questioni relative ad una stazione di servizio, attraverso un ambiente grafico. Essa potrà essere visionata da diversi punti di vista, il principale dei quali offrirà uno sguardo generale. Dovrà gestire quindi l'editing della stazione stessa, offrendo la possibilità di modificare a piacimento le aree all'interno della stazione. Sarà responsabile anche di mantenere traccia dei tipi di carburante messi a disposizione all'interno della stazione e delle relative riserve, nonché delle pompe. Ogni spesa ed entrata dovranno essere consultabili in modo facile e veloce.

Il programma è predisposto ad essere ampliato: è pronto per l'implementazione di una simulazione di macchine in arrivo e altri ulteriori miglioramenti, non previsti in questa release.



3 – DESIGN

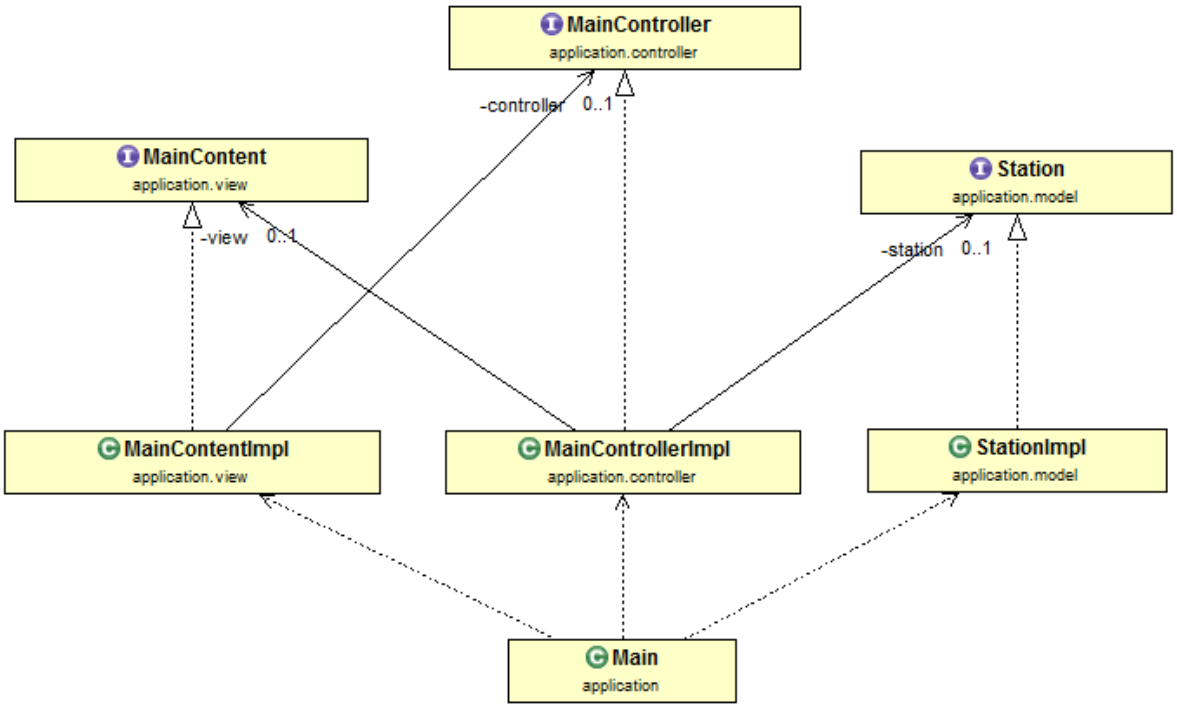
3.1 – Architettura

Il software è stato organizzato seguendo il pattern MVC (Model – View – Controller)

Model *(Mami Alessandro)*
Nel model vengono create tutte le funzioni necessarie al funzionamento del programma, indipendentemente da come vengono poi visualizzati. Esso conterrà tutti i dati relativi alla stazione, le aree che contiene, e rispettivamente, cosa contiene ogni area. Avrà a disposizione la gestione delle riserve e dei vari tipi di carburanti, nonché la gestione dei guadagni e spese.

Controller *(Michelotti Matteo)*
Il controller svolge una funzione importantissima, in quanto non solo collega la view al model, ma gestisce tutta la fase iniziale del caricamento del programma ed effettua tutti i controlli necessari, come il corretto inserimento dei dati provenienti dalla view. Esso ha anche il compito di eseguire gli eventi catturati dalla view.

View *(Pabich Marcin)*
La view è la veste grafica del programma. Essa non sa nulla dei dati: sa che deve visualizzare degli elementi preimpostati o mandati dal controller. Gestisce, in poche parole, tutto quel che riguarda la visualizzazione all'utente dei contenuti, richiamando il controller ad ogni evento. Essa è completamente indipendente dal controller; per questo motivo il programma risulta essere molto flessibile e permette, all'eventualità, di cambiare l'implementazione della view, senza compromettere il funzionamento del programma.



3.2 – Design dettagliato

In questa parte verranno trattate in dettaglio le varie implementazioni delle classi per ogni singola parte del programma, seguendo la logica MVC.

3.2.1 – Model

Nel model si è tentato di suddividere e ricreare nella maniera più articolata possibile tutte le strutture ed i componenti facenti parte di una vera stazione di servizio, senza escludere tuttavia la possibilità di testare ciascuna delle funzionalità tramite l'implementazione di un veicolo con caratteristiche personalizzabili.

BUILDABLES

Parto quindi con la descrizione delle sue componenti più interne e basilari, è stata approcciata questa suddivisione poiché sintetizza al meglio le caratteristiche degli elementi costruibili. Tra questi abbiamo:

Area

Elemento più esterno dopo la stazione, può essere posizionato a piacimento dall'utente grazie ad un sistema di coordinate x ed y, ma pur sempre entro i limiti dello spazio a disposizione.

Ha il compito di contenere un numero definito ed univoco di pompe e inoltre può ospitare la sosta di un solo veicolo per volta, quest'ultimo potrà approfittarne per rifornirsi.

L'area può poi rimuovere le pompe e l'eventuale veicolo al suo interno, mentre per aggiungere, rimuovere o individuare la propria componentistica è stato creato un manager apposito.

Pump

Più internamente troviamo invece le pompe, costruibili solo in numero limitato come accennato in precedenza, possiedono un nome, un tipo di carburante ed una velocità di erogazione.

Inoltre estendono le caratteristiche e le funzionalità di una struttura buildable.

Hanno anche loro un manager personalizzato che si occupa di rimozione ed aggiunta come per le aree in precedenza.

Reserve
La riserva fornisce benzina alle pompe, ma può contenere anche diverse tipologie di carburanti in differenti quantitativi e capacità massime.

Se un veicolo tenta di prelevare più carburante del disponibile gli verranno assegnate le rimanenze.

Tuttavia sarà sempre possibile effettuare un rifornimento della cisterna.

Anche la riserva hai il suo manager ed estende da buildable, che sarà appunto la prossima struttura da definire.

Buildable
Interfaccia con corrispettiva implementazione astratta, contiene attributi e metodi estesi a pompa e riserva, queste hanno infatti in comune la definizione di elemento costruibile.

Anche l'area si trova nella cartella buildables ma pur sembrando azzeccata non le appartengono buona parte di funzionalità e attributi.

Possiamo quindi definire una struttura buildable solo se possiede almeno una durata, un costo di costruzione ed un costo di riparazione.

In definitiva questi costi influenzeranno l'economia della stazione, dato che non sarà sempre possibile acquistare un nuovo rifornimento, sarà necessario rivendere più benzina possibile.

Fuel
Si potranno creare diversi tipi di carburante in quanto sarà definibile dall'utente sia per il nome che per il prezzo, prezzo all'ingrosso compreso.

Tuttavia non ci potranno essere due carburanti con lo stesso nome e ad ogni carburante verrà assegnato un colore univoco, visualizzabile sulla pompa che lo rivende.

Anche il carburante ha il suo manager e permette di aggiungere o togliere a piacimento le varie miscele create.

Money Manager
Gestisce il lato economica della stazione, definendo i diversi investimenti e transazioni effettuabili sia sui carburanti che sulle strutture buildable.

Tutto questo tramite un enum MovementType che permette appunto la costruzione, la riparazione, il rifornimento all'ingrosso e la vendita di carburante.

Ogni transazione economica è indicizzata e dotata di una descrizione.

Inoltre è presente un money manager che gestisce transazioni e aggiorna il credito residuo della stazione.

Vehicle
Il veicolo, o cliente della stazione presenta un solo tipo di carburante con cui può essere servito.

I clienti possono inoltre richiedere diversi quantitativi per poter fare il pieno, ma non sempre saranno in grado di ottenere quanto richiesto poiché la riserve può sempre finire prima del dovuto, in questo caso si ottengono le rimanenze.

Stazione
Parte centrale del programma, ha un nome ed un indirizzo ma può definire il numero di aree edificabili e di pompe costruibili all'interno di ciascuna di esse.

Collega a se i manager degli altri componenti che vengono inseriti dentro al costruttore e richiamati grazie ad appositi getters.

3.2.2 – View

Data la particolarità del programma e della libreria grafica scelta per lo sviluppo, si è scelto di separare bene i vari concetti e viste che verranno visualizzate all'utente. Si è deciso di raggruppare le viste in schede, ognuna delle quali avrà una propria implementazione e logica. L'insieme verrà presentato all'interno del contenuto principale. La logica usata per l'implementazione è top-down: da una classe generale, MainContentImpl, si suddivide il lavoro in varie schede, helper e piccoli elementi.

Ogni elemento della view, che presenterà qualcosa a video, (a parte la scheda info mancante di una interfaccia, in quanto possiede soltanto elementi testuali statici) ha dedicato per se:

- Un file <*xxx*>.fxml che contiene la dichiarazione in linguaggio XML della vista,
- Un file <*xxx*>Style.css che contiene lo stile applicato alla suddetta view,
- Un file <*xxx*>.java che contiene l'interfaccia della view,
- Un file <*xxx*>Impl.java che contiene l'implementazione della view.

Il fatto di mantenere ben 4 file per ogni vista rende la visione del progetto abbastanza complessa, se non ben organizzata. Ognuno di questi gruppi, infatti, verrà messo all'interno del package dedicato a lui. Tutti e 4 file sono necessari al corretto funzionamento della view:

File FXML

Esso è il cuore pulsante della view; nel programma si utilizza poco il linguaggio Java per la vera e propria implementazione della grafica. Essa infatti è creata e gestita quasi interamente dal file .fxml, dentro il quale, in modo strutturato e ordinato, è dichiarato ogni singolo elemento della view. All'interno di esso è presente il riferimento al foglio di stile che dovrà applicare su se stesso.

File CSS

Esso contiene lo stile del foglio. Ogni colore, dimensione del testo e quant'altro dipende da questo file, per cui ogni scheda deve avere il suo, in quanto non tutte le schede condividono lo stesso stile. E' comunque possibile che ci siano molte cose in comune tra di esse: tutti gli elementi ripetuti verranno raggruppati in un file generale, "mainStyle.css". Quindi, anche se alcuni file .css delle view potrebbero risultare vuoti, si è scelto di mantenerli per eventuali futuri sviluppi, facilitando l'inserimento di stili mirati soltanto a quella scheda.

File .java (interfaccia)

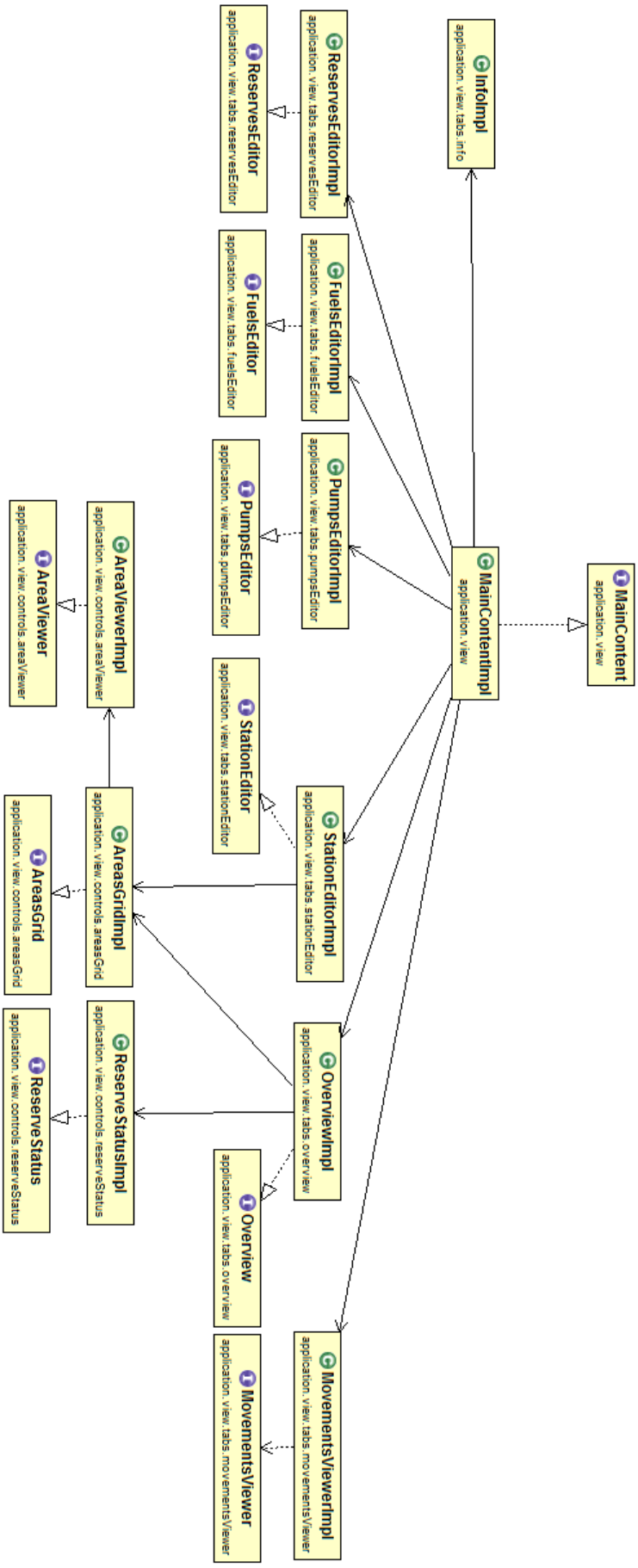
Essa è messa a disposizione per due motivi. Il primo è che sicuramente offre una base per diverse implementazioni della view, che quindi potrà cambiare ma risponderà agli stessi metodi. Allo stesso tempo, l'interfaccia viene usata dal controller per gestire il tutto: getter e setter relativi alla view sono tutti raggruppati nell'interfaccia.

File .java (implementazione)
Svolge un ruolo particolare ed essenziale. Esso da il significato all'interfaccia, che senza l'implementazione non può funzionare. Ha inoltre l'importantissimo ruolo di caricare il file .fxml, collegato direttamente al .css, creando così definitivamente la vista: all'interno del costruttore verrà quindi richiamato il riferimento al file .fxml. Successivamente, per essere congrui al pattern MVC sarà la view che si dichiarerà la radice dell'elemento, ovvero, è lei che ospita quel determinato .fxml e potrà quindi essere riutilizzata come un singolo componente. Per questo dentro la view non è stato possibile usare interfacce: servivano direttamente implementazioni che avessero al loro interno il file .fxml già caricato.

Per essere congrui al modello MVC la view dovrebbe essere in grado di vedere tutti i suoi componenti, lasciando però la gestione degli eventi al controller. Con JavaFX questo principio viene leggermente violato; per vedere tutti i controlli la view deve dichiararsi come il "controller" del file FXML per poter usare il tag @FXML all'interno del codice: esso permetterà di collegare variabili nel codice con componenti dichiarati nel file .fxml, se entrambi saranno dello stesso tipo e nome. Questo tag è però usato anche per catturare gli eventi, per cui la view si vede costretta a doverli gestire. Il disguido viene risolto semplicemente richiamando, ad ogni evento, un metodo dell'interfaccia del controller capace di gestire quell'evento.

C'è una interfaccia default messa a disposizione di tutta la view: tutte le sotto-view e la view principale possono vederla e usarla, e quindi anche il controller è capace di usarla. Essa permette semplicemente di visualizzare messaggi di informazione o di errore in caso ci sia bisogno di comunicare qualcosa all'utente.
Per non rendere caotico il successivo schema UML si omette il fatto che ogni interfaccia della view estende effettivamente da questa interfaccia, chiamata AlertManager.

Ci sono elementi della view, facenti parte del package "controls" che non necessitano di un proprio controller, perché puramente visivi e gestiti interamente dalla view. Hanno comunque una propria interfaccia, ma verranno solo usati all'interno della view stessa, soltanto in alcune schede.


InfoImpl
application.view.tabs.info
ReservesEditorImpl
application.view.tabs.reservesEditor
ReservesEditor
application.view.tabs.reservesEditor
FuelsEditorImpl
application.view.tabs.fuelsEditor
FuelsEditor
application.view.tabs.fuelsEditor
PumpsEditorImpl
application.view.tabs.pumpsEditor
PumpsEditor
application.view.tabs.pumpsEditor
MainContentImpl
application.view
MainContent
application.view
AreaViewerImpl
application.view.controls.areaViewer
AreaViewer
application.view.controls.areaViewer
StationEditor
application.view.tabs.stationEditor
StationEditorImpl
application.view.tabs.stationEditor
AreasGridImpl
application.view.controls.areasGrid
AreasGrid
application.view.controls.areasGrid
OverviewImpl
application.view.tabs.overview
Overview
application.view.tabs.overview
ReserveStatusImpl
application.view.controls.reserveStatus
ReserveStatus
application.view.controls.reserveStatus
MovementsViewerImpl
application.view.tabs.movementsViewer
MovementsViewer
application.view.tabs.movementsViewer

Generalmente parlando, il tutto viene inizializzato dal MainContentImpl, il quale avrà il compito di tenere la suddivisione in schede unita e farà da principale riferimento al controller. All'interno di esso vi saranno dichiarazioni di tutte le sotto-view, quali le diverse schede.
Ognuna delle sotto-schede avrà una propria implementazione dell'interfaccia grafica, per rendere tutto meno caotico e più semplice da leggere e modificare: sono stati usati metodi per creare controlli riusabili. Ogni scheda avrà un compito diverso da gestire.

All'avvio, l'applicazione verrà lanciata grazie al supporto di un package, mnmlwindow, che gestirà l'aspetto della finestra caricando il MainContentImpl con le varie sotto-view associate ad esso. Il suddetto package ha un capitolo dedicato a lui successivamente.

3.2.3 – Controller

La logica del controller è anchessa top-down, parte dalla classe generale MainControllerImpl che setta i riferimenti del model e della view e ogni tab ha un proprio tab controll che setta il riferimento alla propria tab.

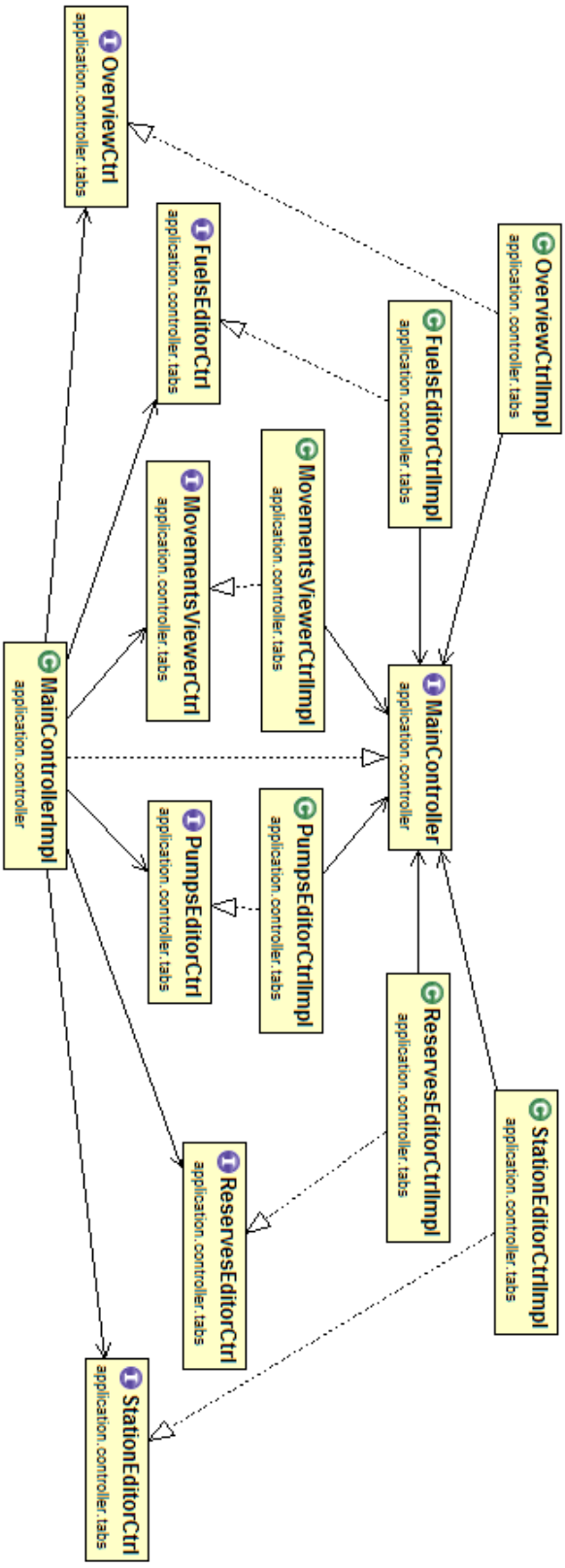
Nel MainControllerImpl si trova il salvataggio su file .xml preso la configurazione dal model, il caricamento dal file che vengono inviati al model man mano che viene letto.

File .XML
Siccome nel programma i dati sono volatili si è deciso di salvarli su file. Nel file viene salvato il nome della stazione, se è aperta, il massimo delle aree che può contenere, il massimo delle pompe che può contenere un'area e il bilancio. Tutti i tipi di fuels che vengono raggruppati in fuel con i rispettivi dati: nome, prezzo, prezzo all'ingrosso e il colore; poi tutte le riserve salvando il tipo, la capacita e il rimanente; successivamente alle pompe prendendo il nome, il tipo di fuel che usa, la velocità, la durabilità, il prezzo, il costo di riparazione e la durabilità rimanente; in fine le aree salvando la posizione x e y e i nomi delle pompe dell'area.

File .java (interfaccia)
È messa a disposizione per diversi motivi. Offre una base per diverse implementazioni del controller, che quindi potrà cambiare ma risponderà lo stesso alla chiamata della view. Poi l'interfaccie vengono utilizzate dalla view per far vedere all'utente quello che succede.

File .java (implementazione)
Ha un ruolo importante, ovvero quello di implementare la propria interfaccia perché senza non può funzionare. Ogni file .java implementato utilizza interfacce messe a disposizione del model per caricare o modificare i dati e quelli della view per decidere cosa modificare, aggiungere o eliminare nel model, in più indica alla view cosa visualizzare dentro ogni tabella. Ogni tab ha un proprio controller per rendere il lavoro comprensibile il più possibile.

Per quanto riguarda il caricamento e il salvataggio della configurazione, esse sono state implementate. Soltanto il caricamento viene utilizzato, mentre il salvataggio è rimasto solo implementato in attesa di una futura release che permette anche il salvataggio dello stato della stazione dentro un file.


OverviewCtrl
application.controller.tabs
FuelsEditorCtrl
application.controller.tabs
MovementsViewerCtrl
application.controller.tabs
PumpsEditorCtrl
application.controller.tabs
ReservesEditorCtrl
application.controller.tabs
StationEditorCtrl
application.controller.tabs
MainController
application.controller
MainControllerImpl
application.controller
OverviewCtrlImpl
application.controller.tabs
FuelsEditorCtrlImpl
application.controller.tabs
MovementsViewerCtrlImpl
application.controller.tabs
PumpsEditorCtrlImpl
application.controller.tabs
ReservesEditorCtrlImpl
application.controller.tabs
StationEditorCtrlImpl
application.controller.tabs

3.2.4 – MinimalWindow

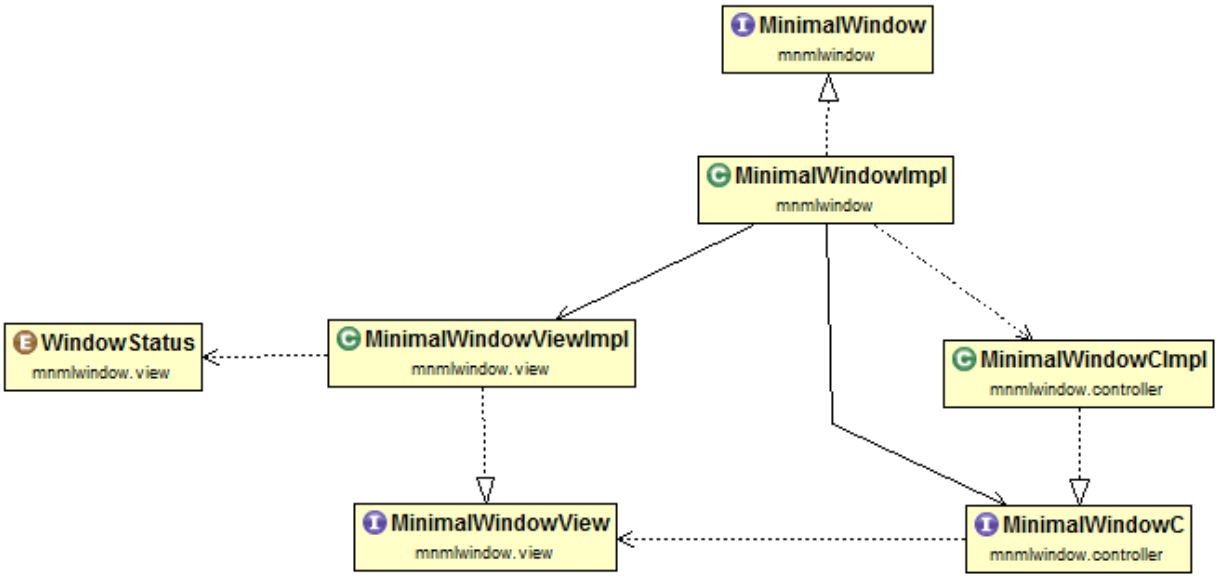
Sviluppando quest'applicazione, soprattutto per quanto riguarda la parte della "view", si è posti subito il problema di fare un primo impatto visivo slegato alla classica grafica di Java. L'idea di usare una libreria per personalizzare la finestra era una delle prime idee, ma su internet i vari risultati portavano a delle librerie non troppo organizzate o comunque non troppo facili o poco o difficilmente personalizzabili. Per questo motivo Marcin Pabich e Michelotti Matteo hanno contributo alla realizzazione del package mnmlwindow che contiene la logica per una finestra completamente personalizzata.

Questo package ha cercato di seguire al meglio la logica del MVC, ma in mancanza di un vero e proprio model si è basato soprattutto tra una distinzione netta tra la view e il controller.

La disposizione delle classi è semplice e segue la logica valida per tutto il programma: .fxml, .css, interfaccia e implementazione. L'interfaccia è usata dal controller, mentre l'implementazione dalla view. Lo stile e la grafica vengono create e personalizzate dai file .fxml e .css che vengono entrambi caricati all'interno della view. È presente un'enumerazione di supporto, immagini ed un file colors.css contenente la definizione dei colori che potranno essere quindi riutilizzati all'interno del package.

Tutto questo è coordinato dall'interfaccia e classe generale MinimalWindow, fatta in modo da poter essere utilizzata al posto del classico "stage" al lancio dell'applicazione.

Il funzionamento è semplice: anziché usare in modo classico lo stage che proviene dal punto di partenza di un'applicazione JavaFX si utilizza la classe MinimalWindow. Essa per essere creata ha bisogno in ingresso di tre parametri: il riferimento allo stage e una lunghezza e altezza minima. All'interno di sé provvede a instanziare la view e il controller e a passare i relativi riferimenti.



Avendo quindi preso lo "stage" dal metodo di lancio, la classe personalizza in dettaglio la finestra, rimuovendo i bordi, aggiungendo l'effetto ombra e permettendo lo spostamento e il ridimensionamento della finestra: da adesso è lei che gestisce interamente tutti gli aspetti che deve avere una finestra generica.

La classe della view è abbastanza corposa, perché contiene molti eventi che dovevano essere gestiti: alla fine, in una finestra capitano moltissimi eventi, e per essere precisi, ne mancherebbero molti altri. Infatti, la classe è predisposta ad essere ampliata e diventare un progetto a se, che potrà essere continuato separatamente anche più avanti. Come prima nel progetto, anche qui per avere un contenuto riusabile si deve usare direttamente la sua implementazione, non interfaccia. L'implementazione infatti estende direttamente da un componente, che non possiede una sua interfaccia: visto che i metodi per impostare una scena, che finirà dentro lo stage, vogliono in ingresso un nodo, ovvero un componente di grafica implementato e pronto per l'utilizzo, usare l'implementazione mi è parsa come la soluzione più valida e veloce.

Sono disponibili vari metodi per la personalizzazione della finestra: uno per mettere del contenuto al suo interno, uno per il titolo, il footer, e un altro per l'icona. La classe è predisposta ad avere ulteriori future migliorie e personalizzazioni.

Per quanto riguarda l'implementazione stessa è comunque difficile realizzare in poco tempo una classe perfettamente funzionante e rispondente ad ogni esigenza. L'implementazione attuale di MinimalWindow non supporta pienamente il ridimensionamento (soltanto un bottone in basso a sinistra lo permette), gli eventi di spostamento non sono ottimizzati e alcune feature, come quelle di Windows che permettono di affiancare due finestre, non sono implementate. Questo (e molto altro) saranno obiettivi successivi del progetto MinimalWindow che, si spera, continui.

4 – Sviluppo

4.1 – Testing automatizzato

Il testing del model è situato in model.MyTest, eseguito mediante JUnit. Esso testa alcune delle funzionalità del programma, ma risulta essere parzialmente implementato e non funzionante.

4.2 – Metodologia di lavoro

Il progetto è nato da un'unione di varie idee sui gestionali da creare. L'incontro iniziale è stato cruciale per definire i ruoli e le attività che ogni membro doveva svolgere. Nei successivi incontri sono state stabilite le varie interfacce ed ognuno ha proceduto nella loro relativa prima implementazione. La suddivisione del lavoro è la seguente:

Model	Mami Alessandro
View	Pabich Marcin
Controller	Michelotti Matteo

Durante il progetto erano molto frequenti i nostri successivi incontri, chiamate tramite i servizi VOIP e/o comunicazione tramite messaggeria istantanea. Il lavoro è stato sempre coordinato e collaborativo, ma ognuno ha pensato alla propria parte del progetto: ciò spiega i commit su BitBucket effettuati solo nelle parti che interessavano al singolo. Solo verso la fine la suddivisione del lavoro si è mischiata per permettere un "bugfixing" efficiente.

4.3 – Note di sviluppo

Si ringrazia fortemente il creatore della libreria Undecorator e UndecoratorBis. Essa è stata lo spunto principale per la creazione e perfezionamento del package mnmlwindow.

"in-sideFX"
Undecorator: https://github.com/in-sideFX/Undecorator
UndecoratorBis: https://github.com/in-sideFX/UndecoratorBis

5 – Commenti finali

5.1 – Autovalutazione

Il progetto poteva essere svolto in maniera più accurata. Molte parti del progetto sono state un po' trascurate o sottovalutate, per cui le difficoltà incontrate non erano poche. Generalmente però siamo riusciti a svolgere un programma decente, completo nelle sue funzionalità. L'implementazione e il risultato finale non è perfettamente ciò che ci aspettavamo di ottenere, ma eravamo sicuramente su una strada buona per un programma valido in tutte le sue parti.

Mami Alessandro *(model)*
Per quanto mi riguarda ero più ottimista ad inizio progetto, abbiamo avuto una buona partenza, ma poi lo sviluppo è andato a rilento. Ho avuto qualche problema ad organizzare la progettazione e a fornire al controller i corretti metodi.
Un'altra difficoltà riguarda l'UML: quello che avevamo pianificato inizialmente sembrava molto promettente, ma si è rivelato molto articolato e complesso. Abbiamo quindi dovuto consultarci diverse volte prima di giungere alla versione definitiva e completa.
Questa ultima settimana avremmo dovuto fare più progressi e sono sorti parecchi problemi inaspettati, quindi penso che si sarebbe potuto ottimizzare o forse anche implementare meglio.
Anche se l'argomento del progetto non risulta particolarmente entusiasmante, mi ha fatto molto piacere poterlo vedere crescere ed evolversi.
Pur essendo la mia prima esperienza di programmazione in gruppo, ritengo si sia dimostrato un modello di progettazione valido, anche se non sempre si è tutti d'accordo o motivati allo stesso modo, qualcosa siamo riusciti a tirar fuori.

Michelotti Matteo *(controller)*
Sono soddisfatto abbastanza per il lavoro. Fare la parte del controller mi ha fatto capire il disagio che si prova programmando in gruppo; facendo la propria parte senza essere dipendenti da altri, per poi unire il tutto e vedere se il progetto funziona.
Ho scoperto la utilità delle interfacce, perché avendo lavorato con C# e Visual Studio non ne avevo visto la grande utilità e vantaggio.
Essendo il controller mi è piaciuto riuscire finalmente a caricare i dati da file e inviarli al model, ordinare alla view cosa visualizzare e, siccome avevo molti controlli simili, mi è bastato fare un metodo privato per ogni file implementato, rendendo il lavoro poco più semplice.

Pabich Marcin *(view)*
Personalmente sono abbastanza soddisfatto della parte che ho svolto. Buttarsi in JavaFX è stata sia una follia, sia un passo in avanti rispetto al corso svolto durante l'anno. Mi piace essere al passo e avevo sin da subito dichiarato di non voler proseguire sulla strada "swing" per la sua ormai non-modernità.
FXML mi risulta molto comodo da usare, è molto più bello da vedere, anche se ancora ha moltissimi problemi da risolvere. Provenendo da un ambiente C#, Visual Studio e basandomi sullo sviluppo con XAML mi sono ritrovato da una parte con mani molto più legate nella fase iniziale di sviluppo, riuscendo però moltissimo ad apprezzare alcuni elementi che mancavano in XAML.
Certamente, non potendo approfondire più di tanto lo sviluppo in JavaFX mi sono dovuto accontentare di guide online, video e altre fonti su internet. Credo di aver svolto un lavoro abbastanza decente, anche se mi rendo conto di alcune problematiche legate all'uso di FXML che potevano essere migliorate (come il riuso di variabili "double" definite in ogni file FXML), ma sicuramente questa applicazione è una base molto solida e una partenza certa verso i futuri sviluppi di altre applicazioni: JavaFX sarà comunque la mia libreria preferita, ciò però non la rende priva di problemi.

Ritornando al programma, me lo sono immaginato diversamente, ma nel complesso non è stato così drammatico. Certamente, volevo mettere più personalizzazioni, slegarmi completamente dallo stile "java", ma per il tempo che ci siamo ritrovati ad avere mi sono concentrato più sulle funzionalità che sull'aspetto. Sull'aspetto generale direi che ci sarebbe da ridire, ma per quanto riguarda il mio vero primo progetto in gruppo sono ottimista per il futuro, sapendo che ho potuto sperimentare il lavoro con più membri.

Per quanto riguarda un piccolo commento sulla classe MinimalWindow, direi che è questo il punto di forza che non vorrei abbandonare. Le implementazioni su internet mi sembrano un po' caotiche e complesse, per cui una reimplementazione, un rinfresco alle librerie presenti credo non faccia male. I risultati su internet portavano sempre a delle guide poco chiare o rimandavano a "undecorator", non sempre consigliato dalla comunità. Spero che questa implementazione possa portare un po' di rinfresco su internet per quanto riguarda una libreria per la personalizzazione della finestra, visto che ho intenzione di pubblicarla e renderla disponibile alla comunità.

5.2 – Difficoltà incontrate e commenti per docenti

Tra le varie difficoltà incontrate nello sviluppo del programma, la prima in assoluto fu quella di determinare univocamente i ruoli dei singoli componenti, seguendo lo schema MVC; non è ben chiaro infatti sin dall'inizio quali sono i compiti di ogni membro, per cui la primissima cosa da affrontare era definire per bene chi doveva fare cosa.

Buttandosi su JavaFX eravamo più che certi di incontrare difficoltà: inizialmente non è stato facile capire il funzionamento di questa libreria, come legare il file .fxml alla classe, come rendere i controlli riusabili, dinamici e meno confusi.

Molte volte durante lo svolgimento del progetto ci è capitato di confrontarsi sulle utilità e funzionalità del programma su cui non eravamo sicuri, riuscendo però a trovare sempre una via di uscita.

Il programma, pur teoricamente completo, potrebbe non risultare adatto ad una vera e propria gestione della stazione di servizio. Molti dettagli e la realtà trattata sono stati semplificati: il programma è una release effettivamente non completa di un possibile progetto più grande.

6 – Guida Utente

L'interfaccia aiuta l'utente in tutto il percorso, in quanto risulta essere semplice da gestire. All'apertura del programma verrà caricato l'ultimo salvataggio disponibile (in fase di consegna ne è presente già uno con qualche pompa/riserva/area disponibile).

La scheda Overview non permette modifiche, ma offre uno sguardo generale alla stazione, facendo vedere la disposizione delle aree e lo stato delle riserve.

La scheda Station Editor permette di modificare e aggiungere nuove aree.
Per modificare un'area bisogna selezionare dalle caselle disponibili le coordinate di quell'area. Successivamente, ogni modifica effettuata va confermata con il relativo bottone. Per aggiungere un'area basta cliccare sul bottone "Switch to adding panel", che porterà sul pannello di aggiunta. Una volta inseriti i campi premere il tasto "insert" per inserire l'area.

La scheda Fuels Editor permette di aggiungere e/o modificare i tipi di carburanti disponibili nella stazione. La gestione è praticamente uguale alle aree.

La scheda Reserves Editor permette di aggiungere, gestire e modificare le riserve della stazione. Oltre alla modifica e aggiunta, permette di riempire la riserva comprando la benzina e di ripararla, vista l'usura della stessa ad ogni operazione.

La scheda Pumps Editor permette di aggiungere, gestire e riparare le pompe disponibili nella stazione. Il funzionamento è uguale alle schede precedenti.

Nella scheda Movements è possibile visualizzare i movimenti effettuati ed aggiungerli eventualmente manualmente. La scheda mostra anche l'attuale guadagno (o perdita) della stazione.

La scheda Information è puramente informativa.