Simple ATC Simulator

25/04/2020

—

Angelini Alessandro, Carletti William, Colotti Manuel, Lisotta Marco

Indice

Analisi **2**
1.1 Requisiti 2
1.2 Analisi e modello del dominio 4
Design **6**
2.1 Architettura 6
2.2 Design di dettaglio 8
Carletti William 8
Colotti Manuel 15
Angelini Alessandro 19
Lisotta Marco 21
Sviluppo **25**
3.1 Testing automatizzato 25
3.2 Metodologia di lavoro 25
3.3 Note di sviluppo 27
Commenti finali: **29**
4.1 Autovalutazione e lavori futuri 29
Guida utente **31**

Analisi

1.1 Requisiti

Il software Simple ATC Simulator mira allo sviluppo di un gestore del traffico aereo che permette di impartire ordini ai vari aeromobili che rientrano nell'area di controllo di uno specifico aeroporto. L'obiettivo dell'utente, che impersonerà un controllore del traffico aereo, sarà quello di far decollare gli aerei in attesa nell'aeroporto e di far atterrare quelli che appariranno agli estremi del radar.
Il tutto dovrà essere effettuato seguendo le procedure standard esplicate nell'apposito tutorial presente sia in questo documento che nel menù (sezione tutorial) di gioco.

Requisiti funzionali

- Visualizzazione di ogni aeromobile e del suo movimento sul radar;
- Possibilità di settare la velocità di svolgimento del gioco (time warp);
- Decollo degli aeromobili da una pista attualmente attiva dell'aeroporto;
- Atterraggio nell'aeroporto attuale degli aeromobili che appariranno durante lo svolgimento della simulazione. Uno specifico aereo potrà atterrare solo qualora rispetterà specifiche limitazioni sui suoi parametri, quali altitudine, velocità e direzione (limitazioni spiegate nel tutorial);
- Gestione del movimento (altitudine, velocità e direzione) di ogni aeromobile in volo;
- Possibilità di scegliere tra tre diversi aeroporti (Fiumicino, Milano Malpensa, Bologna Marconi), ognuno con caratteristiche proprie (numero e orientamento piste) derivanti dalla realtà;
- Gestione delle collisioni tra i diversi aeromobili;

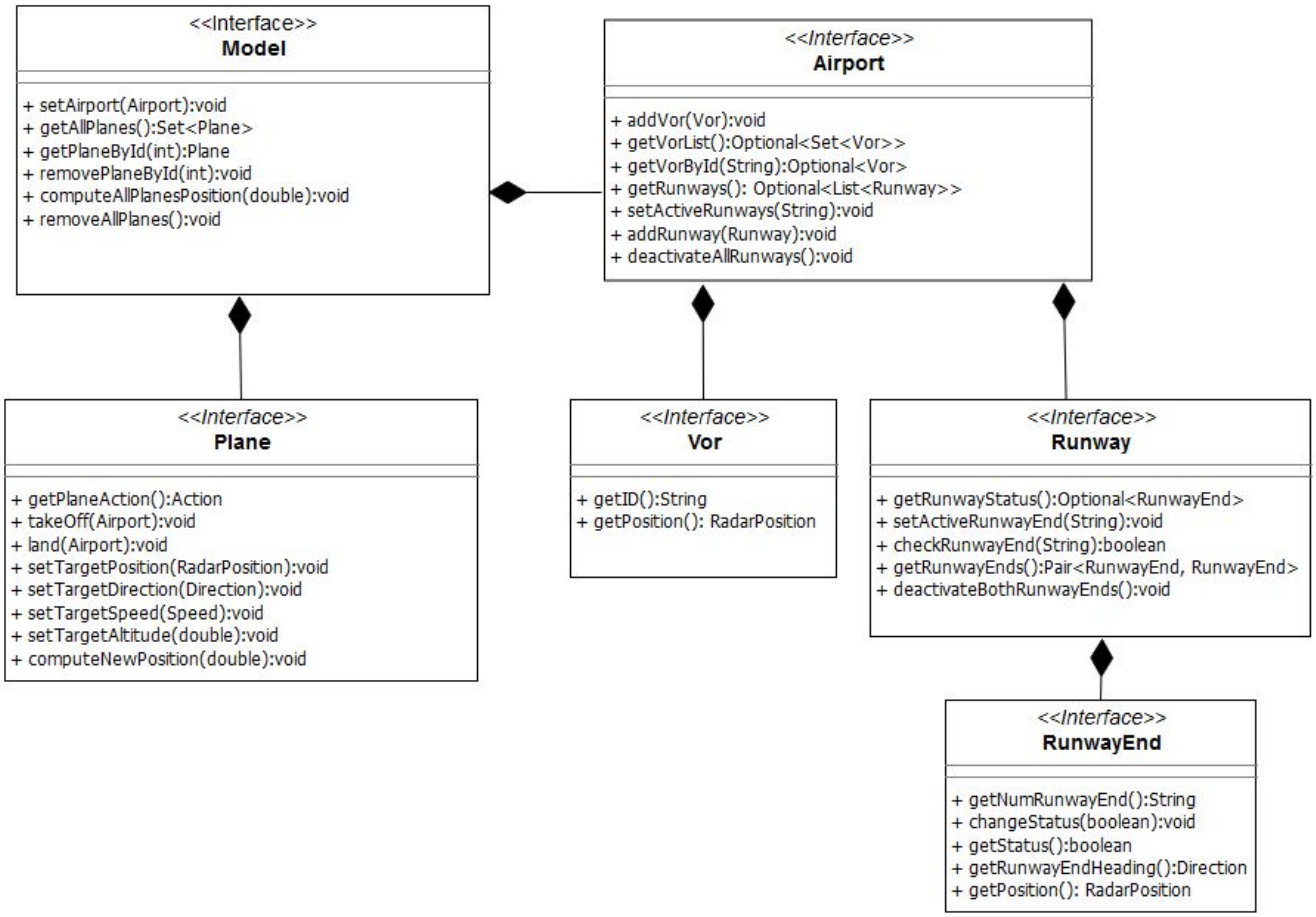
- Visualizzazione di una tabella per ogni aeromobile (strip) contenente id univoco dell'aereo, altitudine, velocità e direzione obiettivo scelta dall'utente;
- Selezione dell'aereo da controllare tramite click sulla corrispondente strip;
- Randomizzazione degli aerei in arrivo ed in partenza;
- Possibilità di scegliere quale pista attivare all'interno dell'aeroporto. I due lati della pista, così come nella realtà, non potranno essere attivati contemporaneamente.

Requisiti non funzionali

- L'applicazione dovrà essere efficiente in una corretta rappresentazione del movimento degli aeromobili e delle interazioni con essi e tra di essi;

1.2 Analisi e modello del dominio

Nell'applicazione Simple ATC Simulator l'utente è in grado di muovere aerei (**Plane**) in uno spazio graficamente bidimensionale, ma a livello di logica tridimensionale, poiché di ogni aereo sarà nota l'altitudine. Il controllo del movimento permetterà ad un aereo di svolgere diverse azioni, tra cui navigazione, decollo e atterraggio.
Questi due ultimi obiettivi risultano essere fondamentali all'avanzamento del gioco; l'utente dovrà decidere in o da quale pista (**Runway**) far atterrare o decollare un aereo. Ogni pista si suddivide in due estremi, chiamati **RunwayEnd**, i quali potranno essere attivati alternatamente dal giocatore in un qualsiasi momento.
Tali piste (una o più) faranno parte dell'aeroporto (**Airport**), attualmente controllato dal giocatore. Inoltre, ogni aeroporto manterrà una collezione che descrive i vari radiofari (**Vor**) presenti nelle aree limitrofe ad esso; questi ultimi potranno essere utilizzati come dei punti di riferimento verso i quali un aereo si potrà dirigere qualora impostato dall'utente.
L'insieme degli aerei attualmente presenti nel radar e dell'aeroporto attualmente selezionato saranno mantenute nel **Model** dell'applicazione.
Una delle maggiori difficoltà sarà quello di gestire il movimento degli aeromobili e di rappresentarne le informazioni relative a runtime.



Rappresentazione del dominio dell'applicazione.

Design

2.1 Architettura

L'applicazione Simple ATC Simulator si basa sul pattern architetturale Model View Controller (**MVC**). All'avvio, la classe Launcher esegue il metodo statico main della View, la quale crea un'istanza di Controller; quest'ultimo inizializza invece il Model del programma.

Model:

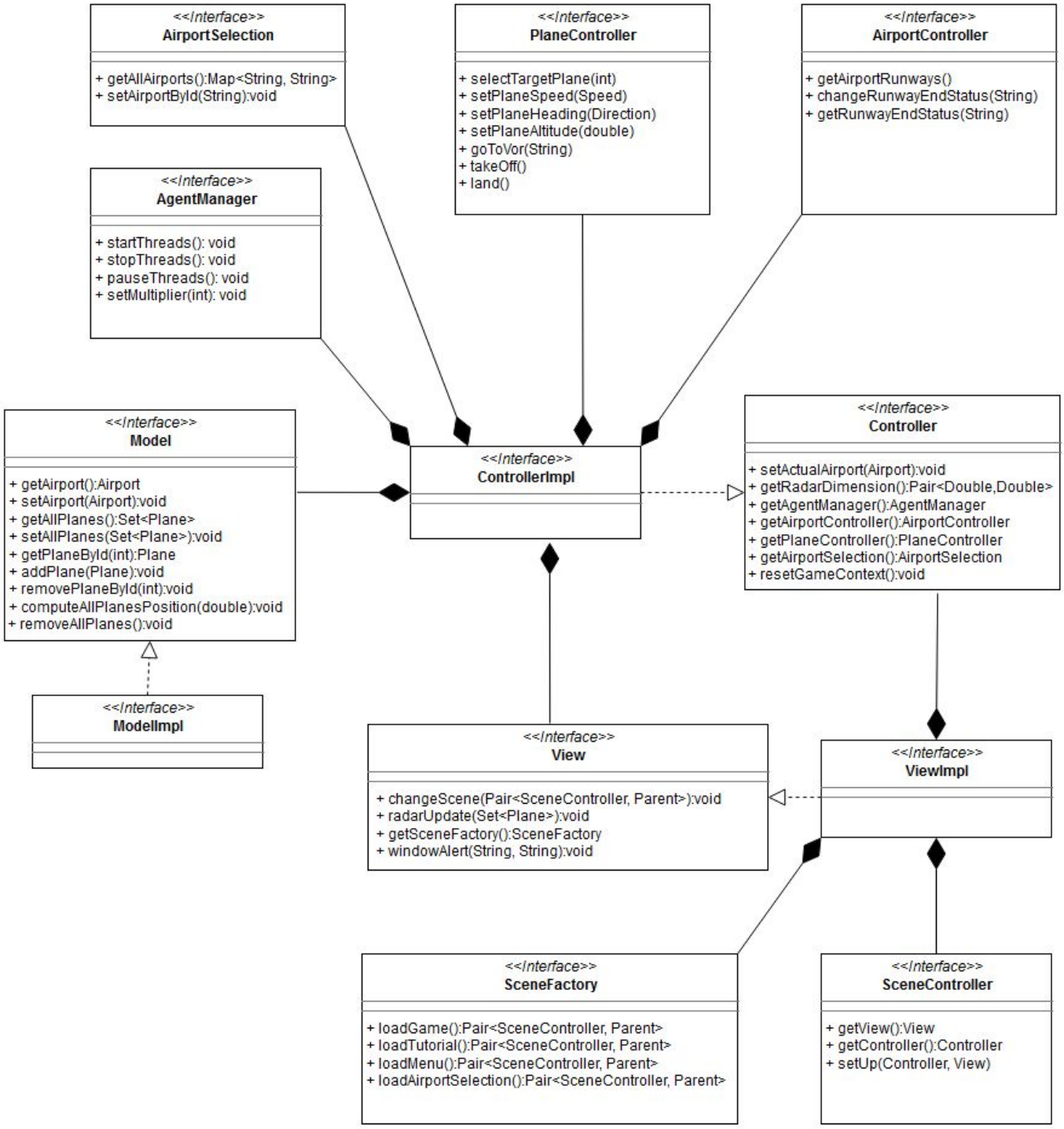
il Model rappresenta la descrizione del dominio dell'applicazione; esso fornisce i mezzi per accedere alle sue varie componenti attraverso il controller preposto.

Controller:

il Controller funge da intermediario tra la View dell'applicazione e il dominio di quest'ultima, comandando al Model modifiche di vario genere sul suo stato attuale, ed alla View la rappresentazione grafica delle informazioni contenute nel Model. Esso si compone a sua volta di diverse sottoparti, ognuna dedicata al controllo di uno specifico aspetto del programma.

View:

la View rappresenta l'interfaccia grafica dell'applicazione, con il quale l'utente può interagire, sfruttando specifiche componenti. Inoltre, essa è in grado, mediante il Controller, di mostrare dinamicamente le informazioni ricevute.
Ogni scena è dotata di un proprio SceneController che gestisce sia l'elaborazione degli input da parte dell'utente, che i dati ricevuti dal Controller.



Rappresentazione dell'architettura dell'applicazione Simple ATC Simulator.

2.2 Design di dettaglio

Carletti William

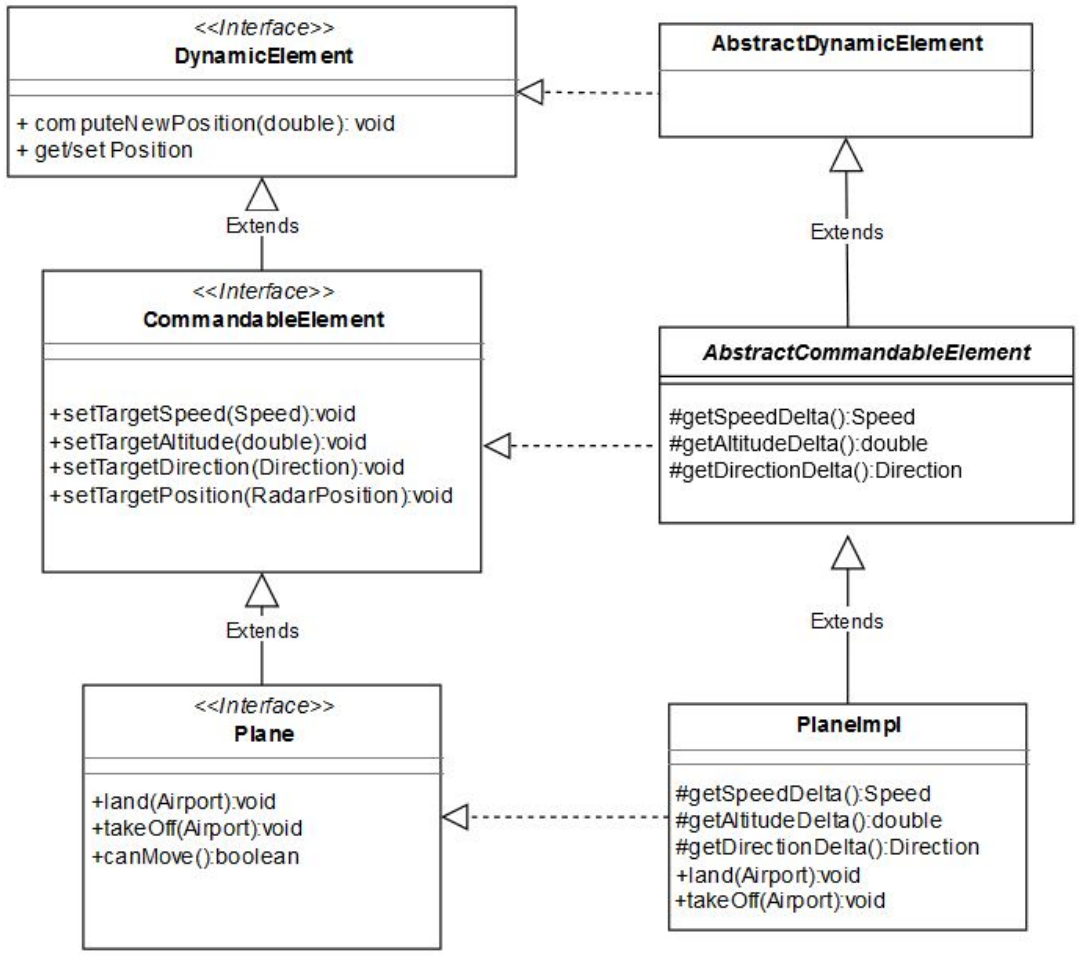
I miei compiti principali sono stati:
(Model)
la creazione di elementi dinamici nel contesto di un radar, l'implementazione dell'entità aereo;
(Controller)
la randomizzazione di aerei nel tempo;
(View)
la realizzazione del radar principale del gioco.

Mi sono infine cimentato nella funzionalità opzionale della scelta dell'aeroporto con cui giocare.

Elementi dinamici

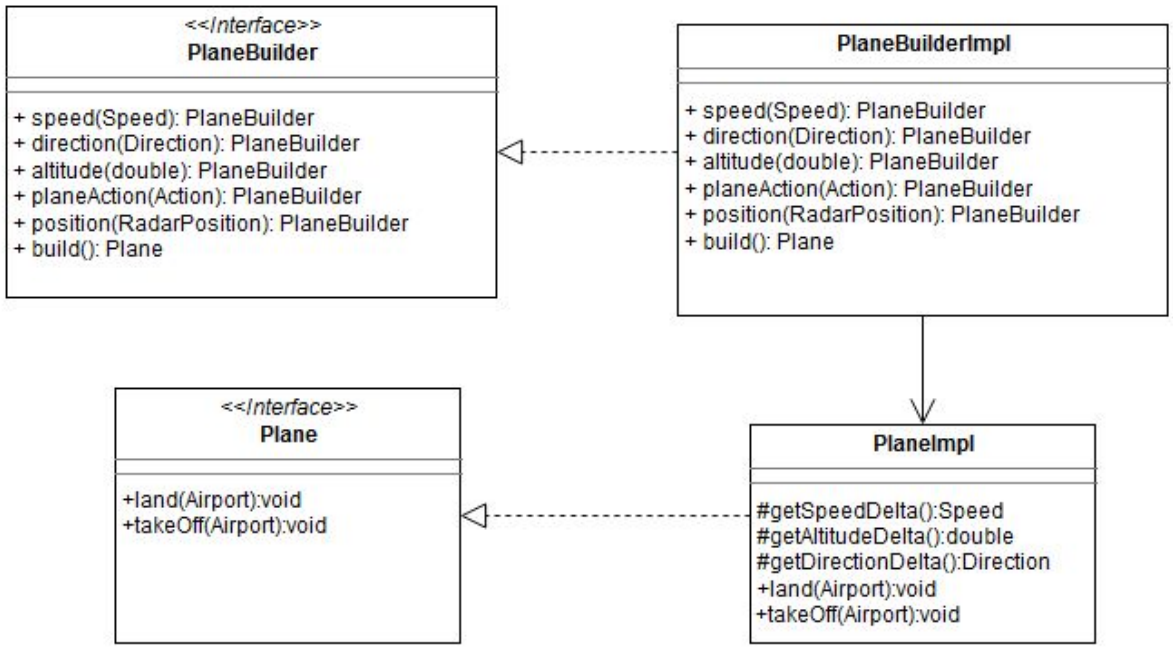
Per realizzare oggetti dinamici all'interno della logica di gioco, è stata modellata l'entità DynamicElement, che rappresenta tutti i possibili elementi dinamici presenti nel radar di gioco. Ogni elemento dinamico è caratterizzato da velocità, direzione, altitudine e posizione nel radar. Per far avanzare uno di questi elementi, viene chiamato il metodo computeNewPosition, che calcola la sua nuova posizione in base a velocità e direzione.
Dall'interfaccia DynamicElement deriva l'interfaccia CommandableElement, che invece modella oggetti dinamici ai quali è possibile indicare posizione, velocità direzione o altitudine da raggiungere.
La rapidità con cui un elemento comandabile raggiunge i propri obiettivi, però, non è nota a priori: infatti, accelerazione, velocità angolare ed accelerazione verticale cambiano a seconda della specifica entità modellata e dalle specifiche condizioni in cui esso si trova. Per questo ho deciso di utilizzare il pattern **template method**.
La classe astratta AbstractCommandableElement include i metodi template computeActualSpeed(), computeActualDirection() e computeActualAltitude(), i quali richiamano un metodo astratto ciascuno.
Così facendo, ogni specializzazione di AbstractCommandableElement potrà specificare questi tre parametri nello specifico momento in cui sono richiesti.

Nella applicazione SimpleATC, è stata modellata un'unica specializzazione di CommandableElement, ossia Plane, che rappresenta gli aerei che possiamo manovrare. Detto ciò, nessuno vieta che in futuro non vengano modellati altri tipi di elementi comandabili dall'utente (ad esempio degli elicotteri), i quali avranno specifici valori di accelerazione (orizzontale e verticale) e velocità angolare.
Con questo pattern l'aggiunta di tali elementi risulterà agevolata.



Rappresentazione della soluzione adottata per rappresentare elementi dinamici, elementi comandabili ed aerei.

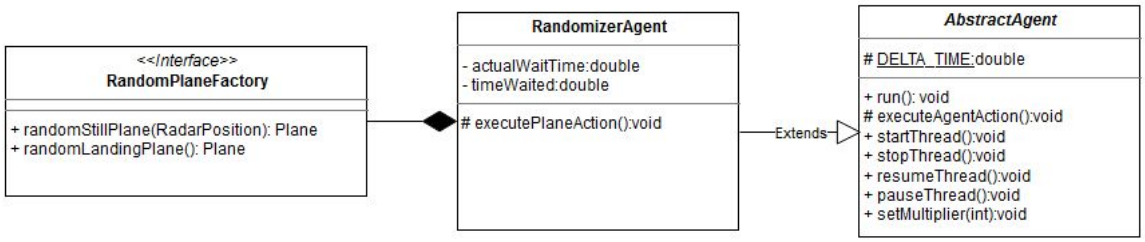
Vista la grande mole di parametri richiesti dal costruttore di un oggetto che aderisce al contratto dell'interfaccia Plane, e considerato il fatto che la generazione randomica degli aeromobili faceva parte delle funzionalità del progetto, ho optato per la creazione di un PlaneBuilder, che rispetta il pattern creazionale **Builder**.
L'implementazione di tale pattern semplifica così la creazione di oggetti che specializzano l'interfaccia Plane, rendendola più leggibile ed evitando eventuali dovuti ad un ordine errato dei parametri necessari.



Rappresentazione del builder di Plane.

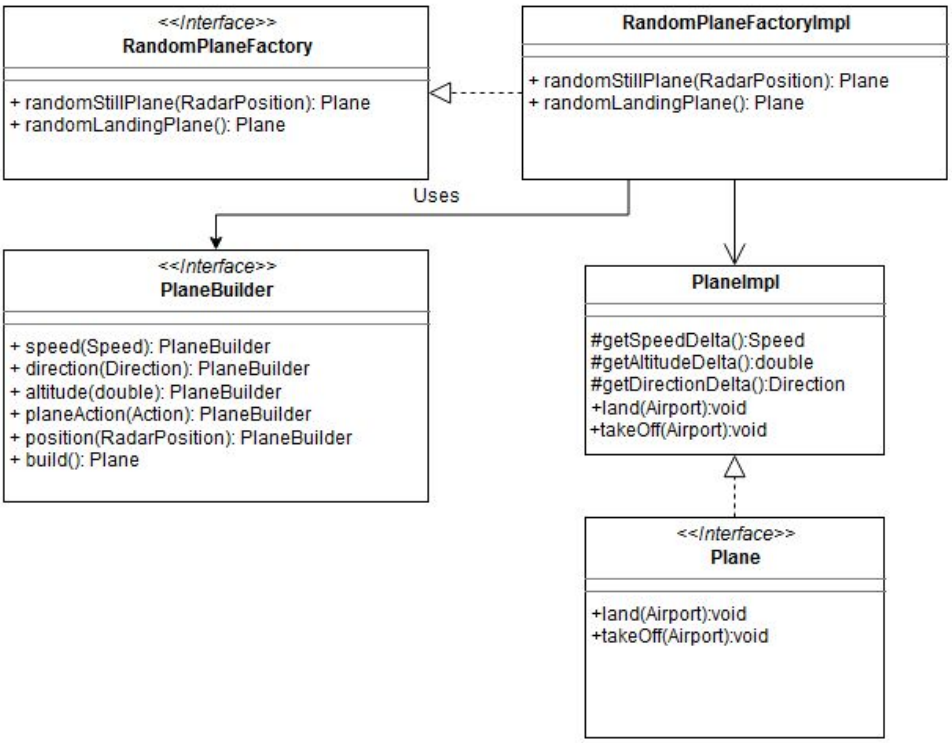
Randomizzazione aeromobili

Per la generazione randomica di oggetti Plane, è stato creata una specializzazione apposita della classe Thread. La scelta di estendere Thread invece di creare un'implementazione dell'interfaccia Runnable sta nel fatto che il periodo di generazione randomica di aeromobili potrebbe potenzialmente non finire mai: in questi casi è preferibile creare una classe Agent, ossia una sottoclasse di Thread.
Da questa classe, in seguito, vista la necessità di ulteriori Agent per altre funzionalità del progetto, è stata ottenuta la generalizzazione AbstractAgent; in questo modo, la classe originale (RandomizerAgent) si riduce all'estensione di quest'ultima, in cui viene definito il metodo astratto executeAgentAction.
Questo metodo viene chiamato nel template method run (implementato da Colotti Manuel), alla base della classe astratta.



Rappresentazione del Agent incaricato della randomizzazione di aerei.

All'interno dell'Agent di randomizzazione, ho deciso di sfruttare un altro pattern creazionale : il **Factory Method**.
La scelta dell'implementazione di tale deriva dal bisogno di creare ripetutamente due tipi diversi di Plane randomici, ossia quelli in partenza e quelli in arrivo. Utilizzando questo accorgimento, all'interno dell'Agent di randomizzazione si avrà solo la chiamata ad un metodo della factory, che si occuperà della creazione dell'aereo.
All'interno dell'implementazione della factory (RandomPlaneFactoryImpl), è stato utilizzato il Builder citato in precedenza per agevolare la creazione degli aerei.



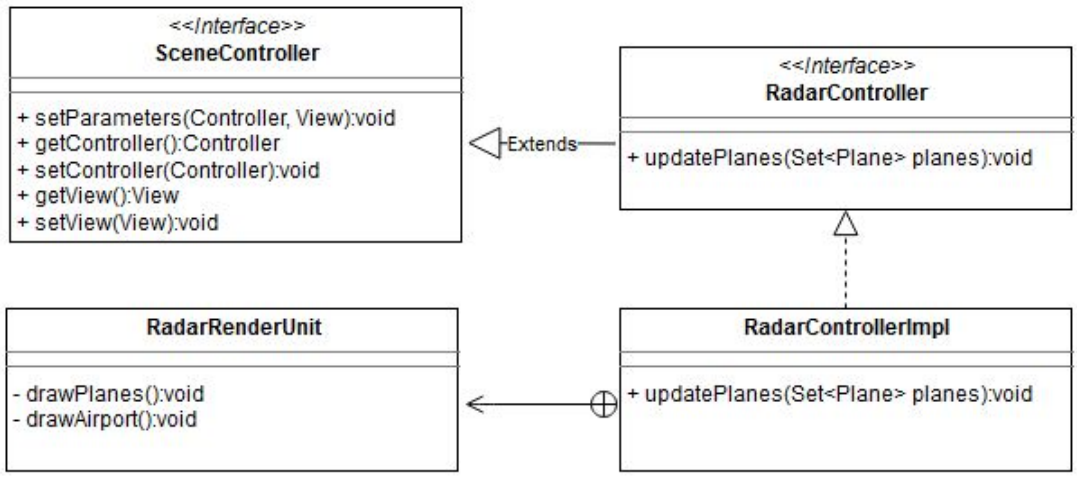
Rappresentazione della factory di aerei con parametri random.

Radar di gioco

Per la parte relativa alla grafica dell'applicazione, mi sono occupato del radar principale del gioco. Elemento fondamentale di questa porzione è stato l'utilizzo del componente JavaFX chiamato Canvas. Tale componente è stato utilizzato sia per la parte statica della rappresentazione del radar, ossia l'aeroporto con le sue piste, che per la rappresentazione degli aerei.
Per evitare di renderizzare ogni volta l'aeroporto, la cui visualizzazione rimane stabile nel tempo, sono stati utilizzati Canvas differenti :

- l'airportCanvas, che è alla base del Pane principale, il quale viene aggiornato solo a fronte di ridimensionamento della finestra;
- il radarCanvas, dove vengono costantemente disegnati gli aerei in movimento nel radar; anche questo componente si ridimensionerà in base alle dimensioni della finestra principale.

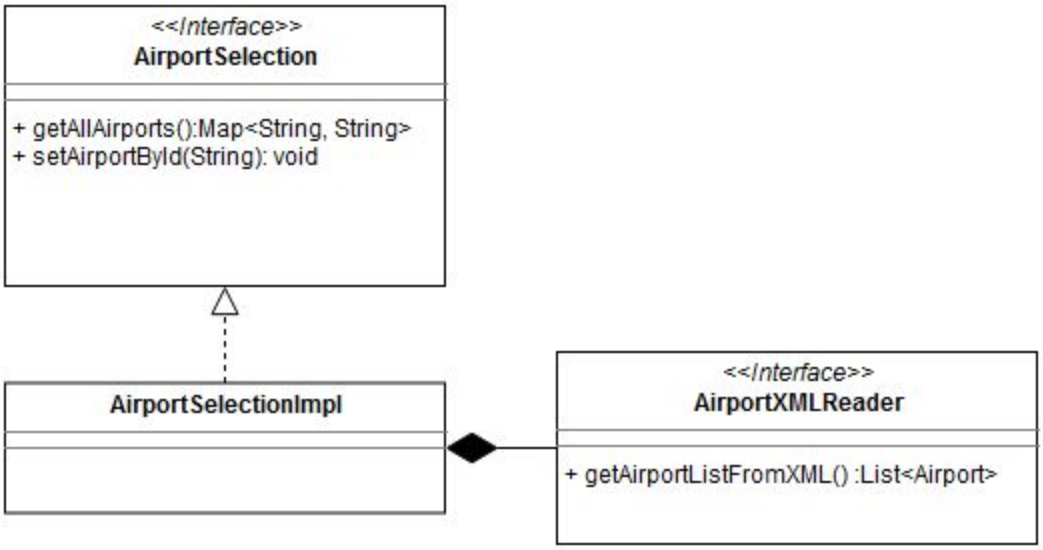
Ad ogni aggiornamento delle posizioni dei diversi aerei presenti nel radar di gioco, verrà chiamato un metodo sulla View principale dell'applicazione, il quale farà in modo di passare tutti gli elementi al controller del radar di view.
Per adempiere al principio di Single Responsibility (SRP), l'effettivo controller della schermata conterrà un'istanza della classe innestata (non statica) RadarRenderUnit, che si occupa dell'aggiornamento della rappresentazione di aeroporto e aerei nel radar. Ho optato per una classe innestata non statica poiché essa verrà utilizzata solo dal controller della schermata di radar; inoltre, RadarRenderUnit utilizza campi privati di quest'ultimo : in questo modo, non sarà obbligatorio passarglieli, visto che saranno comunque visibili al suo interno.



Rappresentazione del controller associato alla scena di visualizzazione del radar di gioco.

Selezione dell’aeroporto

Infine, per la parte opzionale mi sono occupato della selezione dell’aeroporto con cui giocare. Per rendere questa funzionalità espandibile il più possibile evitando la riscrittura di codice, mi sono affidato al parsing di file XML.
In questo modo, la lista di aeroporti disponibili nel gioco sarà rappresentata da un file XML presente tra le risorse del programma e potrà essere espansa in qualunque momento, indipendentemente dal codice scritto.
Per il parsing XML mi sono affidato alle API Document Object Model (DOM), che permettono la conversione di un qualunque documento XML in un albero di oggetti che possono poi essere acceduti.

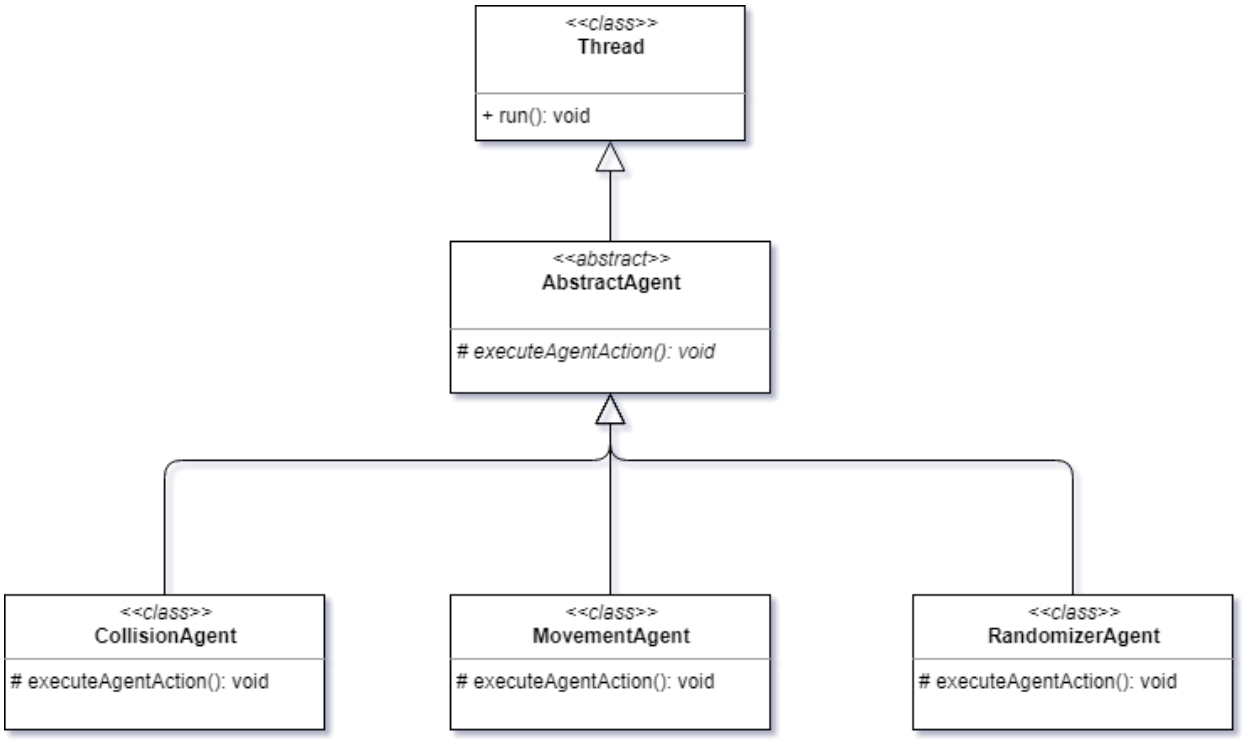


Rappresentazione della sottosezione di controller principale che si occupa della selezione dell’aeroporto di gioco.

Colotti Manuel

Thread impiegati

Dato che per ogni entità dinamica (Plane) è necessario calcolare e aggiornare continuamente le proprietà che lo identificano all'interno dell'area di simulazione nonchè valutare le interazioni con altre entità dinamiche, si è deciso di impiegare tre diversi **Agent** (MovementAgent, CollisionAgent e RandomizerAgent) i quali estendono la classe astratta **AbstractAgent** che estende a sua volta **Thread**.
Il comportamento di ognuno di questi Thread è implementato all'interno di ogni specifico Agent secondo pattern Template Method; in questo specifico caso il template method risulta essere **run()** il quale chiama il metodo astratto **executeAgentAction()**, implementato diversamente a seconda dell'Agent in questione. La scelta di usare un template method per modellare questi aspetti risulta essere estremamente utile nel caso di future aggiunte di nuovi Agent che controllano aspetti di gioco distinti da quelli già realizzati.

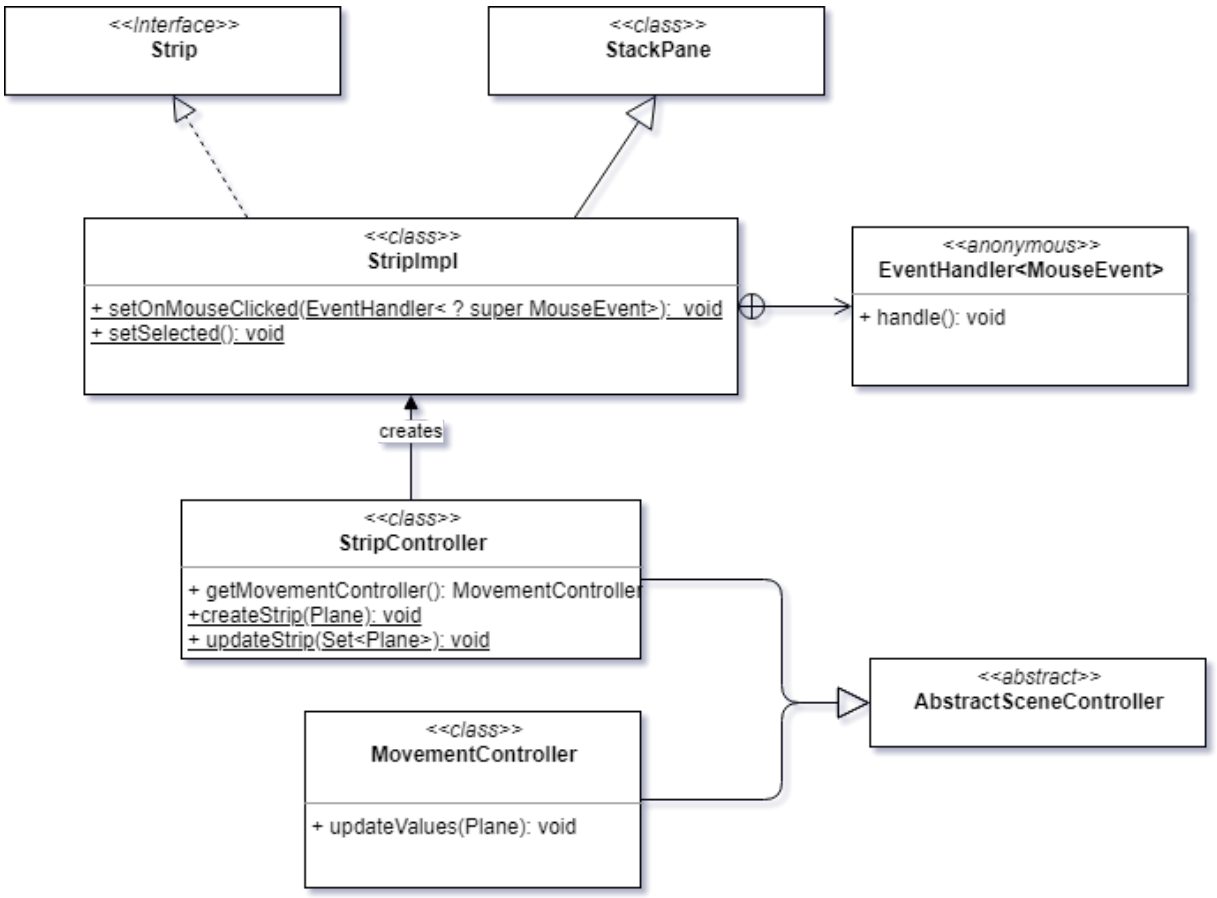


Aggiornamento delle proprietà di un aeromobile

Una delle funzionalità più importanti del gioco risulta sicuramente essere la gestione del movimento dei vari aeromobili da parte dell'utente. Per garantire un tempestivo aggiornamento della posizione degli elementi è stato necessario sfruttare le funzionalità di vari **Listener** agganciati ai diversi elementi di **View** (Come Slider, ChoiceBox e Button), ma anche un Listener associato ad ogni **Strip** (elemento descritto nella parte di analisi).

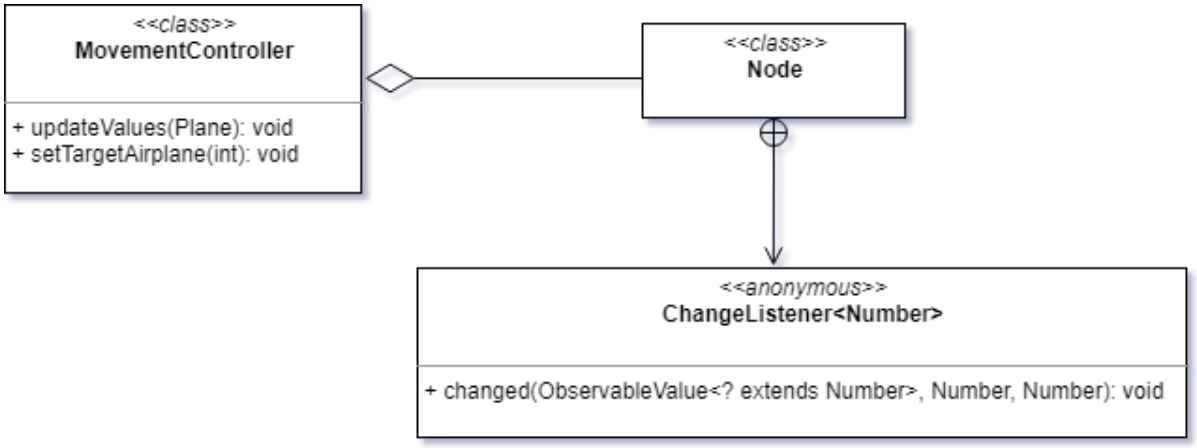
Listener sulle strip

Questo listener (uno per ogni strip) rappresenta la componente **Observer**, la quale al click sulla Strip(**Observable**) viene notificata di aggiornare i valori delle componenti grafiche del gestore di movimento (Slider, Label, ChoiceBox, TextField) relativamente allo stato attuale dell'aereo con cui si vuole interagire.



Listeners sulle componenti grafiche

Anche per quanto riguarda i Listener sulle componenti di View si è optato per l'utilizzo del pattern comportamentale Observer, l'unica differenza è che in questo caso gli Observable, identificati dalle componenti di view presenti nel gestore di movimento, notificano la loro variazione al proprio Listener (Observer) che a sua volta, attraverso il Controller, aggiornerà le informazioni di Model relative allo stato del velivolo.



Queste scelte di progettazione garantiscono la possibilità di aggiungere in futuro nuove funzionalità legate al comportamento delle entità di gioco; un esempio rappresentativo potrebbe essere l'implementazione di percorsi standard che un aeromobile sarà in grado di seguire dopo il decollo e prima dell'atterraggio (in gergo aeronautico standard departure e standard arrival); uno di questi percorsi potrebbe essere scelto tramite un elemento di view che si occuperà di notificare l'aggiornamento dello stato dell'aereo attraverso il proprio Listener.

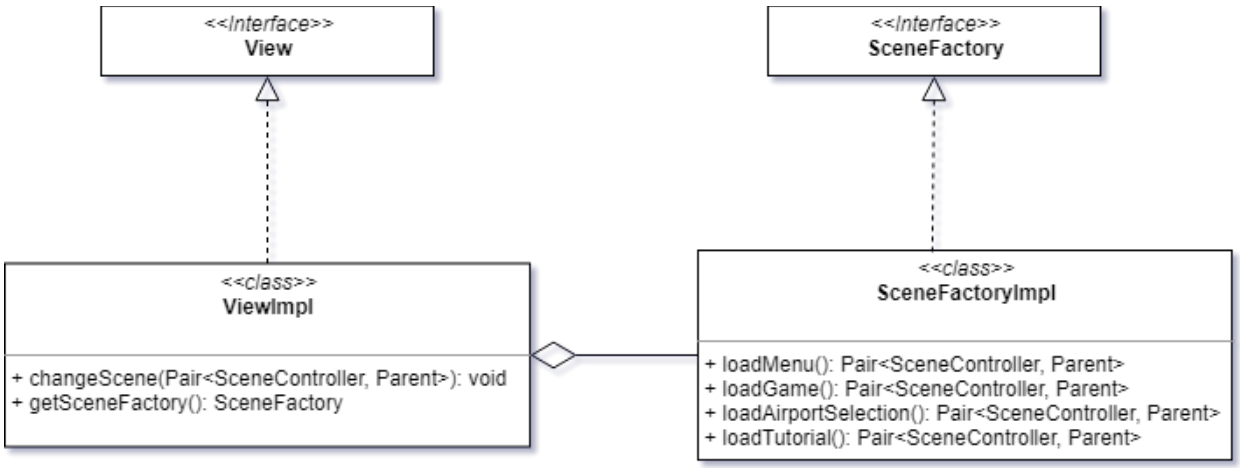
Gestione delle scene

Come per la maggior parte dei software si è anche dovuto provvedere a una gestione dinamica delle scene mostrate all'utente che nel caso di Simple ATC Simulator si suddividono in:

- **Menù principale**
- **Menù tutorial**
- **Menù selezione aeroporti**
- **Schermata di gioco**

Per poter reperire una specifica schermata al momento della richiesta è stata implementata una **Factory** di scene, la quale tramite uno dei metodi pubblici chiamati sulla stessa , identifica la scena che si vuole aprire e restituisce un Pair contenente un **Parent** (parte grafica caricata da un file fxml) e un controller della scena (**SceneController**) relativo al Parent ricevuto.
Una volta reperita la coppia di valori costituita da SceneController e Parent, la View aggiorna la scena corrente e il campo che identifica il SceneController attuale.

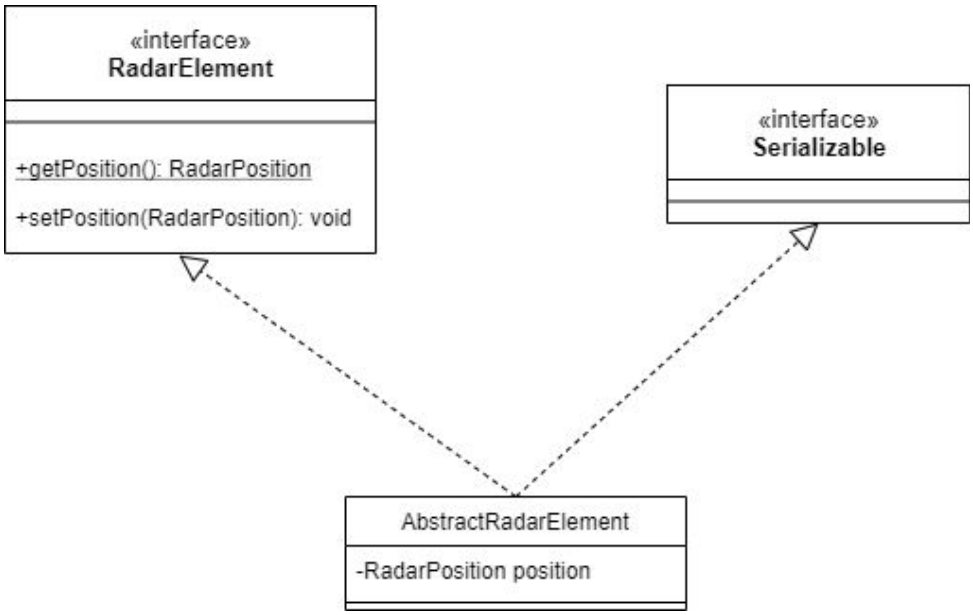
In questo modo la **View**, chiamando un singolo metodo sulla factory, sarà sempre in grado di reperire una qualsiasi delle scene disponibili mantenendo di fatto una radicale indipendenza dalle altre componenti del pattern **MVC** (M e C).
Questo approccio risulta particolarmente efficace nel caso si opti in futuro per l'aggiunta di una o più nuove schermate che arricchiscono la user experience con nuove funzionalità.



Angelini Alessandro

RadarElement

Ho deciso di fare come base di tutti gli elementi del radar la classe RadarElement, che è la base di tutti gli elementi, e contiene la loro posizione nel radar, che abbiamo deciso di utilizzare siccome nel nostro programma ci sono elementi dinamici (aerei), e statici(vor e aeroporto), quindi i secondi implementano direttamente questa classe, mentre i primi usano un'altra classe chiamata DynamicElement, ma che a sua volta implementa questa.



Model

Siccome abbiamo deciso di seguire il pattern MVC come descritto precedentemente, ho scritto il Model che è una parte centrale del pattern e del progetto, che consiste nel contenere i dati più importanti, come la lista di tutti gli aerei presente nel radar, e dell'aeroporto attivo in quel momento insieme ai suoi vor.
Il model oltre che contenerli, gli espone alla view per farli mostrare a schermo grazie a tutti i loro dati contenuti qui, poi grazie al controller gli tiene sempre aggiornati.

SceneFactory

Ho fatto il file SceneFactory, che so occupa nel caricamento di ogni scenario del progetto, carica il menù, il gioco e il tutorial.
Quindi ogni qual volta che qualche classe gli serve cambiare scena, chiamerà questa classe passandogli il nome della scena da caricare e gli ritornerà lo SceneController e il Parent del file caricato.

StripController

Poi per la parte di View, ho fatto le strip che sono spiegate precedentemente, ho deciso di creare uno StripController che passandogli ogni volta nella funzione updateStrip la lista di tutti gli aerei, ha il compito guarda se ci sono aerei da eliminare, aggiungere e modificare.
In seguito modifica la lista degli strip contenuta in un VBox.

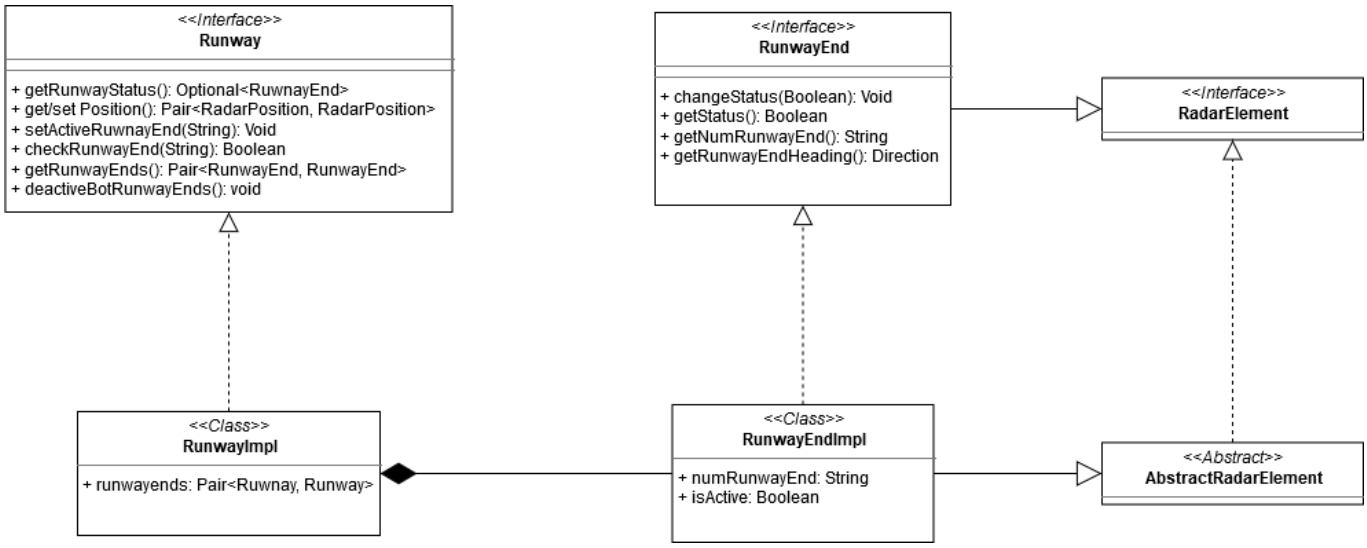
CollisionAgent

Infine CollisionAgent, un thread che estende AbstractAgent, che passandogli la lista di aerei, controlla se ci sono degli aerei vicini tra di loro nel radar o se si stanno scontrando.

Lisotta Marco

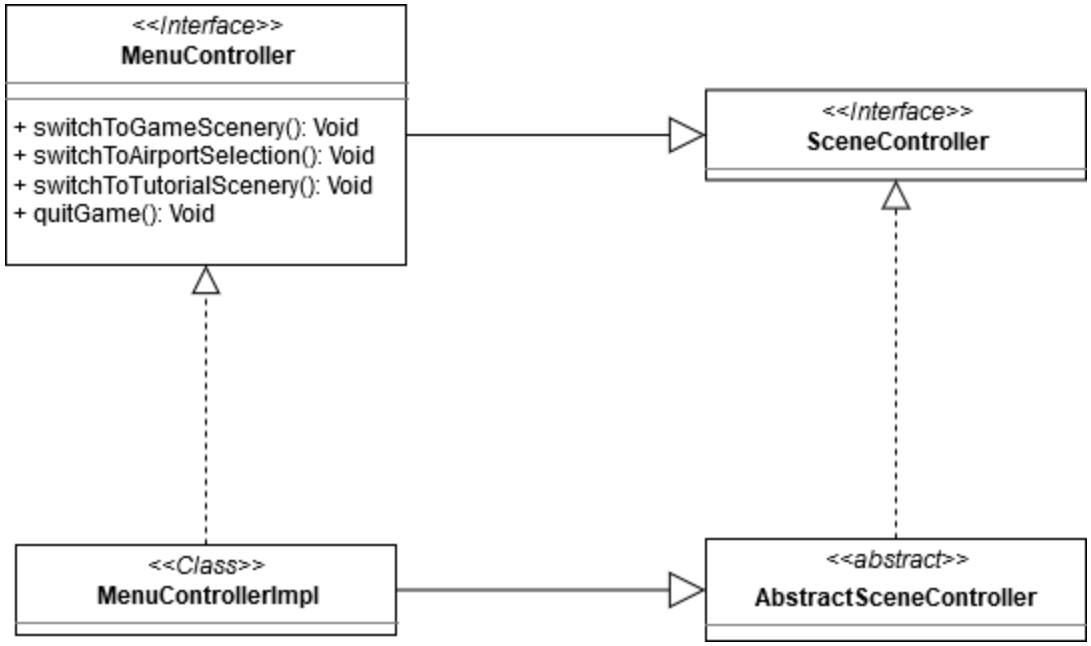
Runway

Per la realizzazione di elementi all'interno della logica di gioco, si è optato per definire ogni oggetto presente nella schermata di gioco come estensione della classe base RadarElement già esplicata in precedenza, attraverso un Template Method.
In quanto la pista è definita da due estremi in ambito grafico e del gameplay, ho deciso di definire la classe Runway attraverso i suoi estremi e quindi di comporre ciascuna pista da un Pair di RunwayEnd.
Ciascuna runwayend è definita: per il suo numero/ID, per il suo stato attuale, per la sua posizione nel Radar già definita in AbstractRadarElement e da una direzione in cui indica. Nel Model si è optato per definire la direzione di un elemento all'interno del gioco attraverso una classe Direction, che non necessità di essere contenuta in runwayend in quanto facilmente deducibile dal suo numero/ID che indica la direzione di utilizzo della pista in gradi.
In questo modo l'intero programma si interfaccerà con le Runway, le quali gestiranno le rispettive RunwayEnd secondo la logica da loro implementata.



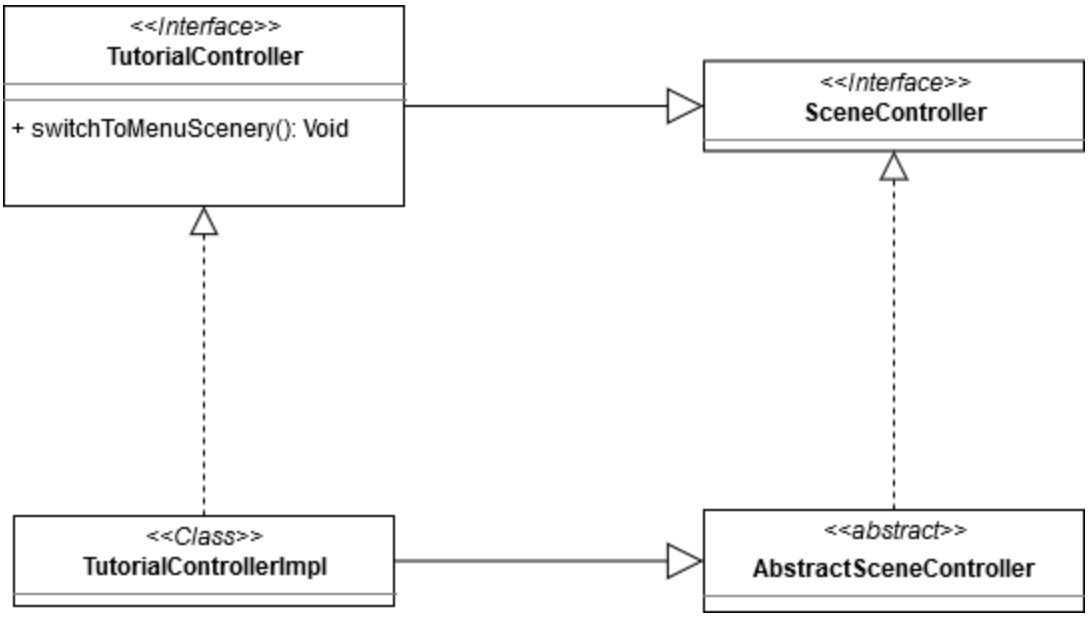
Menu

Per quanto riguarda il menù inziale ho sviluppato la scena con SceneBuilder, formata da un semplice FlowPane per riuscire ad incolonnare i pulsanti in modo chiaro e corretto. Ho collegato il file FXML al proprio controller MenuController in modo da controllare gli eventi nel momento dell'interazione da parte dell'utente con i pulsanti.
Per mettere in relazione ciascun metodo con il proprio pulsante ho utilizzato l'interfaccia di SceneBuilder e il metodo onAction() in ciascun pulsante per lanciare il metodo prestabilito al momento dell'interazione da parte dell'utente.
Ciascun di essi carica un diverso Stage utilizzando la View e lo SceneBuilder come già esplicato in precedenza nella Gestione delle scene.



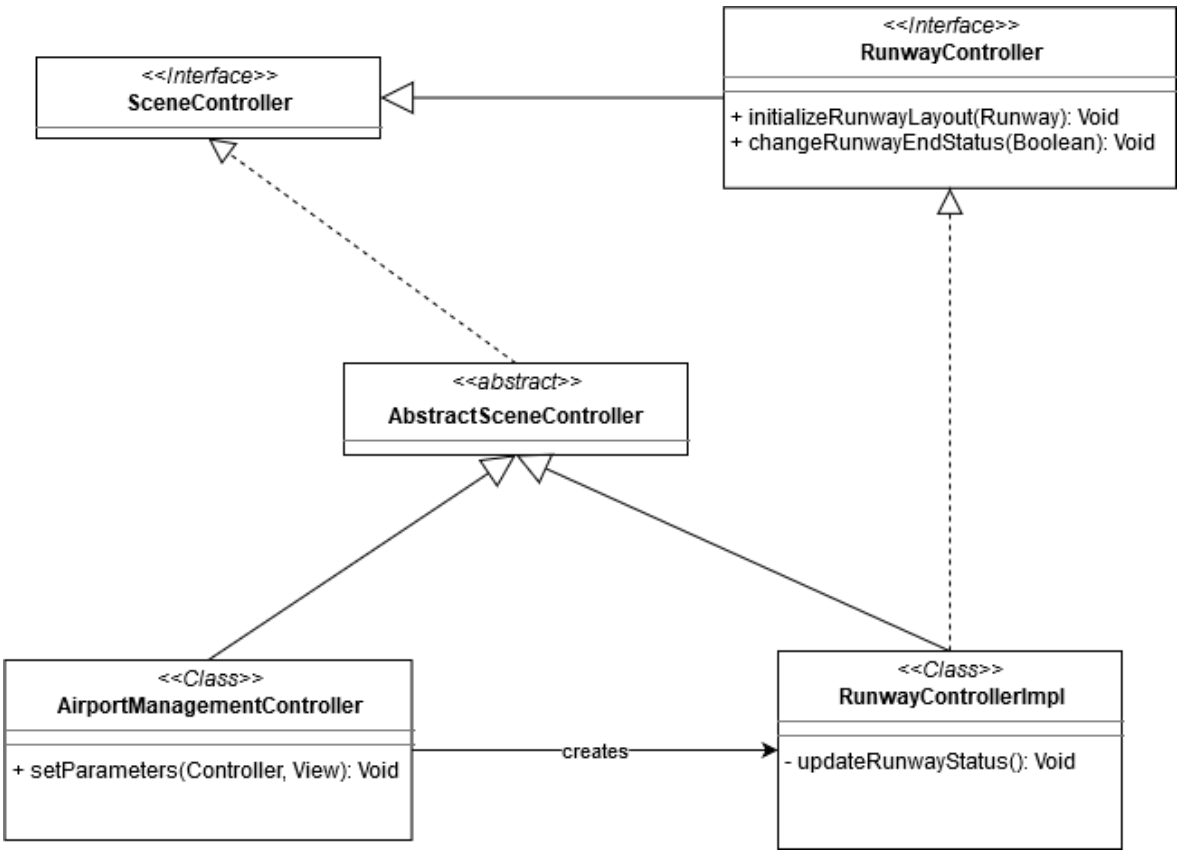
Tutorial

Mi sono occupato di sviluppare la scena del Tutorial e in quanto parte prettamente statica quindi nulla di interattivo con l'utente o con il modello MVC, ho scelto di implementarla completamente attraverso SceneBuilder.
Ho suddiviso il Tutorial in capitoli tramite un TabPane e caricato le immagini per ciascuna parte attraverso un ImageView e il metodo setImage() presente nell'interfaccia grafica di SceneBuilder.
Il file FXML è collegato al proprio TutoriaController per permettere l'interazione con il pulsante <Menù> per ricaricare la Menu Scene in qualsiasi momento.



Airport Layout

Per la parte grafica relativa al gioco, visto che mi sono occupato dell'implementazione delle runway e delle runwayend io e il mio gruppo abbiamo pensato che fosse giusto che implementassi anche della parte grafica relativa alle Runway e alla loro gestione.
Considerando le runway come elemento statico all'interno della logica di gioco ho optato per suddividere la gestione in due file FXML uno specifico per la gestione di ciascuna runway, RunwayController, e uno per la loro creazione, AirportManagementController.
La scelta di lavorare con file FXML invece di creare una classe, come nel caso delle Strip visto in precedenza, è stata dettata dalla loro indipendenza e la loro staticità intrinseca, in quanto una volta definite non possono essere aggiunte o eliminate durante la stessa istanza di gioco.
Per istanziare le Runway ho fatto l'Override del metodo presente nell'AbstractSceneController setParameters() per aver accesso al Controller, in questo modo posso conoscere le piste dell'aeroporto selezionato e istanziarle in ciascun RunwayController, dopo averlo caricato nel proprio FXML attraverso un FXMLLoader.



Sviluppo

3.1 Testing automatizzato

Per il testing automatizzato del programma, è stata utilizzata la libreria JUnit.
Gli aspetti dell'applicazione che abbiamo deciso di testare sono quelli relativi al Model ed al Controller; più precisamente, abbiamo valutato il funzionamento degli aeroporti, degli aerei e della factory di questi ultimi.
Le classi di test si possono trovare nel percorso "src\test\java\".

3.2 Metodologia di lavoro

Per il controllo del versionamento abbiamo usato il software Git, seguendo i consigli appresi durante le lezioni: ossia, la suddivisione in feature del progetto e lo sviluppo di queste da parte di uno, o in alcuni casi, più membri, per poi concludere, una volta raggiunta una versione stabile, con il merge sul branch Master.
L'integrazione delle diverse funzionalità dell'applicazione è risultata abbastanza complessa per quanto concerne la parte di View: infatti, nella schermata di gioco principale, si possono trovare più funzionalità sviluppate dai singoli membri del gruppo. Per mettere insieme tali porzioni di codice, ci siamo affidati ad una caratteristica della libreria JavaFX che permette di innestare file FXML all'interno di un file FXML: in questo modo, ciascun componente ha sviluppato un proprio layout FXML (con annesso controller) che in seguito è stato inserito nel layout principale (in questo caso il RadarLayout).
Nella parte di controller, invece, ogni membro ha inserito i metodi necessari per l'interazione tra il Model è la propria sezione di View.

Angelini Alessandro:

- sviluppo ed implementazione degli elementi base di radar;
- sviluppo ed implementazione dell'interfaccia Model;
- interfaccia per il caricamento di una scena della GUI;
- implementazione delle strip della GUI;
- gestione delle collisioni tra gli aeromobili.

Carletti William:

- sviluppo ed implementazione degli elementi dinamici di radar;
- sviluppo ed implementazione degli aerei;
- randomizzazione degli aeromobili che compaiono nel radar di gioco;
- sviluppo della rappresentazione grafica del radar di gioco, con possibilità di gestire il time warp di gioco;
- caricamento degli aeroporti con cui è possibile giocare.
- implementazione dei test relativi agli aeromobili;

Colotti Manuel:

- sviluppo ed implementazione dell'interfaccia relativa ad una posizione nel radar;
- sviluppo ed implementazione di un generico aeroporto e dei Vor;
- implementazione della classe che definisce un generico thread;
- implementazione thread di movimento degli aeromobili;
- sistema per la rilevazione di aerei in rotta di collisione;
- sviluppo ed implementazione dell'interfaccia View;
- gestore di movimento nella GUI e suo interfacciamento con il relativo Controller e con le Strip;
- implementazione dei test su un generico aeroporto.

Lisotta Marco:

- sviluppo ed implementazione di runway e runway end;
- implementazione del menù di gioco;
- implementazione del tutorial;
- gestione grafica delle runway ed interfacciamento con il Controller.

3.3 Note di sviluppo

Angelini Alessandro:

- Nella classe CollisionAgent ho deciso di fare uso delle Stream per la semplicità con cui si possono filtrare gli elementi all'interno di una collezione e anche per la agevolazione che dà alla lettura del codice.
- Sempre nella precedente classe ho utilizzato le Lambda Expression dentro alle Stream, per facilitarmi ancora di più nella funzione di collisione.
- Siccome abbiamo deciso di usare JavaFx per la componente grafica, che non è stato spiegato tanto nel dettaglio durante il corso, mi sono studiato abbastanza la struttura per la realizzazione delle strip.

Carletti William:

Nella mia parte di progetto sono state utilizzate le seguenti feature:

- Optional come valore di ritorno dei getter dei target di un elemento comandabile; i campi target non sono degli optional per permettere la serializzazione delle istanze, ma possono essere null.
  Quindi, invece di tornare direttamente il loro valore, nei getter viene restituito un Optional contenente lo specifico target.
- Stream: le Stream sono state utilizzate spesso nella realizzazione del progetto; si possono trovare utilizzi in PlaneImpl e RadarControllerImpl.
- Lambda expression: data la comodità di utilizzo (soprattutto insieme alle stream), le lambda expression sono state utilizzate molteplici volte nel progetto: se ne possono trovare usi principalmente nella classe PlaneImpl, RadarControllerImpl.
- StringBuilder: dato il massiccio numero di campi presenti in AbstractDynamicElement e AbstractControllableElement, si è preferito l'utilizzo di StringBuilder per i metodi toString.
- Utilizzo di JavaFX: la parte grafica del progetto è stata creata mediante la libreria grafica JavaFX; infatti, la grafica del radar di gioco ha un suo controller specifico, ossia RadarController;

- Utilizzo del parser XML DOM per la lettura degli aeroporti con cui è possibile giocare da file.

Il metodo che determina se è più rapido girare in senso orario oppure antiorario per raggiungere un certo angolo, è stato ricercato ed ottenuto nel forum math stackexchange; in seguito il link: (https://math.stackexchange.com/questions/110080/shortest-way-to-achieve-target-angle).

Colotti Manuel:

Nell'implementare la mia parte di progetto mi sono concentrato molto nell'approfondimento di strumenti fino ad ora parzialmente trattati tra cui JavaFX, MVC, JUnit ed i vari design pattern visti durante le lezioni. Uno degli aspetti su cui mi sono maggiormente concentrato è stato lo studio di come far interagire, sfruttando JavaFX, componenti di View tra di loro e con il Model attraverso gli appositi Controller. Ho inoltre cercato di dedicare particolare attenzione all'ottimizzazione del codice e del suo funzionamento attraverso l'utilizzo di Stream, Lambda , classi anonime e Opzionali; come pure anche tramite un corretto impiego dei Listener relativi alla parte di gestione del movimento.

Lisotta Marco:

Con la realizzazione del progetto ho compreso meglio aspetti implementativi quali pattern, template e altri aspetti avanzati o più specifici come lambda, stream e optional anche se non direttamente implementati da me. Ho utilizzato molto, visto la mia suddivisione del lavoro, la libreria JavaFX e ho lavorato con file FXML; strumenti che ho approfondito personalmente.

Nel progetto è stata utilizzata la classe Pair fornita nelle esercitazioni d'esame del corso, la quale modella una coppia di oggetti di due tipi generici X e Y.

Commenti finali:

4.1 Autovalutazione e lavori futuri

Angelini Alessandro:

Questo progetto è stata una buona esperienza per il mio miglioramento nel lavoro di gruppo, anche per le mie conoscenze nell'ambito di java. Avevo già fatto un'esperienza simile alle superiori, ma con persone che già conoscevo, invece per questo progetto ho lavorato con persone che ho conosciuto da poco, quindi abbiamo avuto qualche problema di comunicazione, e anche con l'emergenza sanitaria non abbiamo potuto ritrovarci a programmare e discutere insieme per risolvere eventuali problemi riscontrati. Tutto sommato è stata una bella esperienza che diventerà molto utile per lavori futuri sempre in questo ambito.

Carletti William:

Sono parzialmente soddisfatto di come è stato svolto il progetto: infatti, almeno inizialmente, mi è rimasto abbastanza complicato integrare tutte le conoscenze apprese dal corso di Programmazione ad Oggetti (più altre acquisite individualmente) nella realizzazione di un progetto di queste dimensioni.
Una delle principali difficoltà che ho incontrato è stata l'utilizzo del pattern architetturale MVC combinato con la libreria grafica JavaFX.
Considero i punti di forza della mia parte di progetto la modellazione degli oggetti comandabili, che permette in maniera agevole la creazione di un qualsiasi elemento dinamico controllabile in ambito del radar di gioco, e la generazione randomica degli aeromobili. Quest'ultima, con l'estensione del concetto di Factory di Plane in Abstract Factory, potrebbe in futuro rendere possibile la generazione di un qualsiasi elemento dinamico, senza particolari modifiche al codice sorgente.
Un aspetto dal mio punto di vista migliorabile è l'efficienza della rappresentazione degli elementi nel radar di gioco.

Nonostante tutto, credo che questo progetto sia servito sia a fornire una visione completa per quello che concerne la realizzazione di un lavoro di gruppo (essendo personalmente abituato a svolgere lavori individuali), che a consolidare le mie conoscenze nell'ambito di programmazione ad oggetti.

Colotti Manuel:

Sono abbastanza soddisfatto del lavoro svolto nello sviluppare il progetto; ammetto che all'inizio, non conoscendo bene JavaFX e MVC, ho avuto difficoltà a capire come strutturare la mia parte di lavoro. Tuttavia tramite confronto con il gruppo e studio personale sono riuscito a raggiungere i miei obiettivi nonché anche a migliorare alcuni aspetti in varie parte del software. Tra gli aspetti più importanti del mio lavoro si può individuare:

- lo sviluppo degli aeroporti e dei relativi Vor l'implementazione del Thread di movimento
- l'implementazione del gestore di movimento TCAS (Traffic Collision Avoidance System) per la segnalazione all'utente di un imminente collisione Interazioni tra gestore di movimento e Strip

Sarebbe molto interessante continuare in futuro lo sviluppo di questa applicazione aggiungendo nuove importanti feature: più tipologie di aeromobili con caratteristiche diverse, gestione di un velivolo in emergenza attraverso l'applicazione di specifiche procedure ed implementazione di percorsi standard per l'arrivo e la partenza da un aeroporto.

Lisotta Marco:

Personalmente sono soddisfatto del lavoro svolto dal mio gruppo, abbiamo creato un codice abbastanza leggibile ed estendibile, inizialmente abbiamo speso un po' di tempo sulla progettazione ma ci siamo subito buttati a capofitto sul codice e la progettazione è andata di pari passo con l'implementazione.
Sfortunatamente non penso di aver dato il meglio di me in questo progetto visto anche magari la parte del lavoro quasi prettamente grafica da me sviluppata. Scelta non voluta da parte mia o del mio team, ma semplicemente l'evoluzione degli eventi ha portato a questo tipo di suddivisione e magari una scarsa comunicazione, dovuta purtroppo al periodo storico attuale, che a mio parere ha influenzato molto la nostra interazione e la visione globale sul progetto.

Nel complesso sono soddisfatto dell'esperienza che ho avuto e del lavoro svolto con i miei compagni; se potessi gli sceglierei anche per progetti futuri, mi sono trovato molto bene e nonostante tutto siamo sempre riusciti a trovarci d'accordo e sulla stessa lunghezza d'onda. Sempre pensando al futuro, le feature che potrebbero continuare a espandere il progetto sono diverse: tipi di aeromobile differenti, piste con diversa priorità, condizioni meteorologiche avverse che intaccano il volo e la possibilità di atterraggio degli aerei.

Guida utente

La guida utente dettagliata dell'applicazione Simple ATC Simulator è presente nella sezione Tutorial, accessibile dal menù di gioco.