Relazione di progetto per il corso di Programmazione ad Oggetti

A.A. 2017/2018

UNIBOPOLI

Versione universitaria del celebre gioco “Monopoli”

Autori:

Alesiani Matteo (matteo.alesiani2@studio.unibo.it) – 806476

Rossolini Andrea (andrea.rossolini2@studio.unibo.it) – 794193

Università degli studi di Bologna

Campus di Cesena

Corso di Laurea “Ingegneria e Scienze informatiche”

Indice

Capitolo 1 3
Analisi 3
1.1 Requisiti 3
1.2 Analisi e modello del dominio 4
Capitolo 2 7
Design 7
2.1 Architettura 7
2.2 Design dettagliato 9
Capitolo 3 25
Sviluppo 25
3.1 Testing automatizzato 25
3.2 Metodologia di lavoro 25
3.3 Note di sviluppo 28
Capitolo 4 29
Commenti finali 29
4.1 Autovalutazione e lavori futuri 29
Guida utente 31

Capitolo 1

Analisi

In questo capitolo viene fatta l'analisi dei requisiti e quella del problema, ossia vengono elencate le cose che l'applicazione deve fare (requisiti) e viene descritto l'ambito in cui si colloca (analisi del problema). Si evidenziano quindi le entità principali che costituiscono il dominio del problema e le relazioni che vi sono tra loro, utilizzando anche diagrammi UML per semplificarne la comprensione.

1.1 Requisiti

Il gruppo si propone di realizzare il seguente software come elaborato di progetto per l'insegnamento di Programmazione ad Oggetti; si vuole dunque implementare una versione, leggermente modificata, del celebre gioco da tavola "Monopoly" (in Italia conosciuto come "Monopoli") della casa editrice *Hasbro.*
Di seguito sono elencati i requisiti che l'applicativo deve rispettare.

Requisiti funzionali:

- Il gioco deve permettere di iniziare una partita con un minimo di 2 ad un massimo di 6 giocatori.
- Il gioco, come nelle regole originali del monopoli, attribuirà automaticamente all'avvio un numero prestabilito (che varia in base al numero di giocatori che partecipano ad una partita) di contratti, decurtando dai giocatori il denaro necessario per soddisfare le regole del gioco.
- Ogni giocatore, durante il proprio turno, ha la possibilità di comprare proprietà (secondo le regole del monopoli originale, quindi se e solo se il giocatore si trova nella stessa posizione della proprietà interessata e se quest'ultima non è posseduta da nessun giocatore), eseguire scambi con gli altri giocatori, costruire case, ipotecare proprietà e partecipare ad eventuali aste
- Il gioco non prosegue se durante il turno di ogni giocatore, questo non tira almeno una volta i dadi, seguito da un movimento automatico e da

un'eventuale conseguenza dovuta a questo (se il giocatore finisce in carcere – *jail* – non sarà in grado di muoversi da quella posizione).

- Il gioco sarà accompagnato da una melodia continua che si ripeterà lungo tutto il proseguo del gioco, a meno che l'utente non decida di silenziarla.
- Oltre al sottofondo musicale il gioco gestirà anche dei piccoli effetti sonori che accompagneranno alcune azioni dei giocatori (come ad esempio il lancio dei dadi)
- Gli effetti sonori saranno gestiti separatamente rispetta alla musica di sottofondo, ovvero si potrà scegliere di silenziare o solo la musica o solo gli effetti sonori oppure entrambi (o nessuno ovviamente)
- Sarà possibile salvare lo stato di una partita per poi riprenderla in un secondo momento.
- L'interfaccia grafica sarà in grado di adattarsi a diverse risoluzioni e dimensioni degli schermi.
- Saranno gestiti gli eventi legati alle "PROBABILITÀ" ed agli "IMPREVISTI".
- Sarà presente un menù iniziale che permetterà di impostare numero di giocatori, nomi e relativi avatar (pedine) di questi.

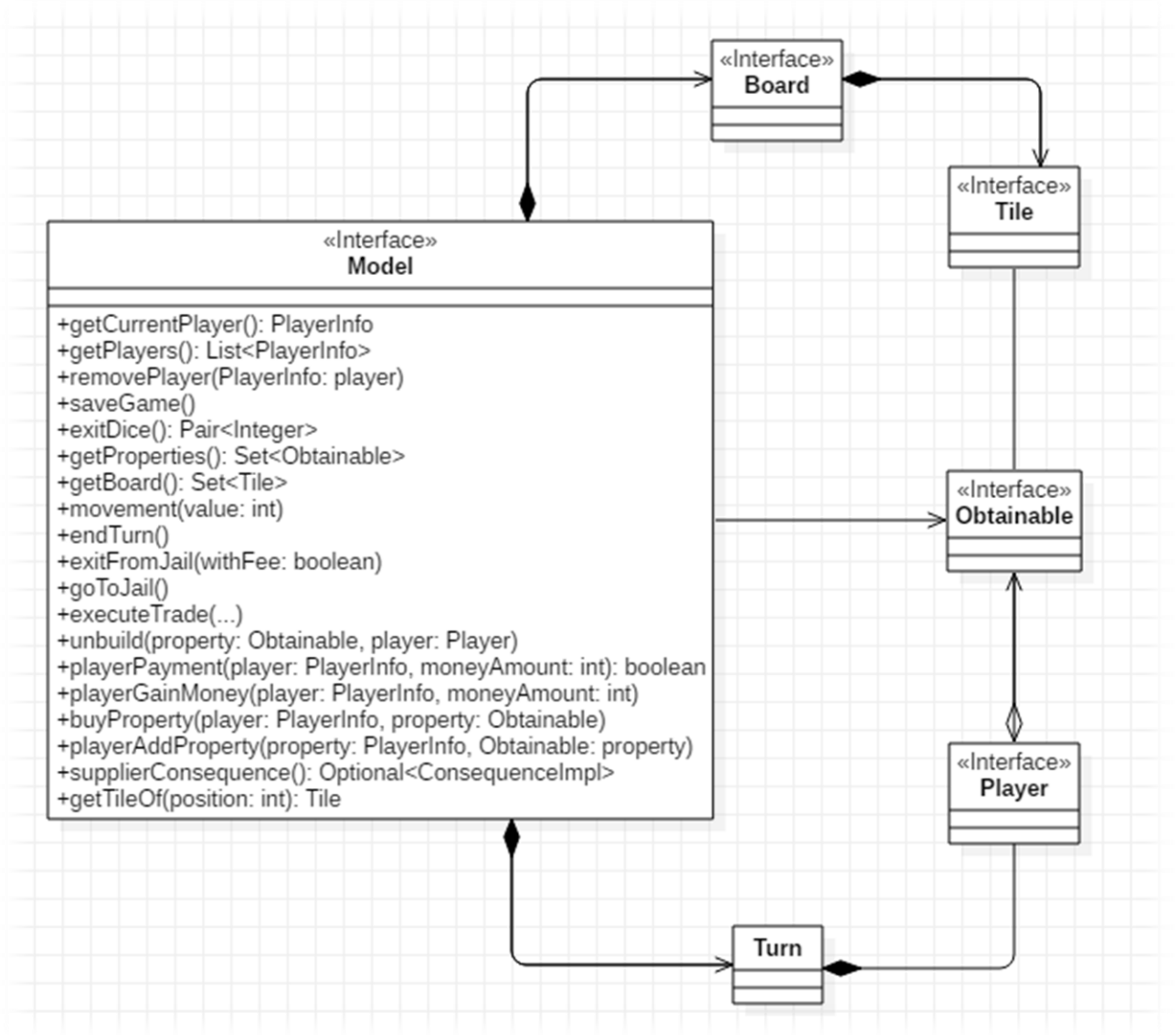
Funzioni ritenute opzionali:

Il gruppo vorrebbe realizzare ulteriori funzionalità, come ad esempio la gestione di mode per la personalizzazione di contratti ed icone. Queste funzioni non sono, appunto, considerate come obbligatorie per la realizzazione completa dell'elaborato.

1.2 Analisi e modello del dominio

Il domino dell'applicazione (**Model**) incapsula al suo interno la logica del gioco, mantenendo lo stato corrente della partita.
Tramite l'utilizzo della classe **Turn** il model si compone da un numero (definito dall'utente) di giocatori (**Player**), gestendo il susseguirsi dei turni di questi (usufruendo di una lista circolare – **CircularList**), e da una plancia di gioco (**Board**), la quale, a sua volta, incorpora l'insieme di tutte le caselle (**Tiles**) presenti nel gioco, definendone posizione e tipo, infatti le caselle del gioco avranno dei tipi differenti, in base alla diversa funzione all'interno del gioco e influiranno sulla partita in maniera

differente. Inoltre è necessario memorizzare la posizione di ogni giocatore durante il corso della partita. Questi “Tiles” vengono letti da file, ciò permetterà in futuro la possibilità di realizzare mode per la personalizzazione del gioco.
L’inizializzazione del gioco, dopo aver scelto il numero dei giocatori, il nome ed il rispettivo avatar, avviene in maniera completamente automatica, “assegnando” ad ogni giocatore un numero prestabilito di Tiles di tipo **Obtainable** e una quantità di denaro relativa al valore dei contratti ricevuti.
Durante il proprio turno il giocatore deve necessariamente poter tirare i dadi almeno una volta e si muoverà nella plancia di gioco del numero di caselle ottenuto dalla somma dei due dadi; inoltre potrà, tramite input dell’utente, eseguire scambi, di proprietà e di denaro, con gli altri giocatori (ma solamente con un giocatore alla volta). Il movimento di un giocatore, nella maggior parte dei casi, determina una conseguenza in base al tipo di Tiles sul quale capita; ad esempio i Tiles di tipo Obtainable possono essere acquistati se un giocatore ci capita sopra, o possono costringere il giocatore a pagare un affitto (determinato dal contratto in sé); oppure, se capita su un altro tipo di Tiles questo può avere diverse conseguenze, come ad esempio il muoversi verso qualche direzione o il finire in carcere, od il pagare una tassa. Se un giocatore finisce in carcere (se lancia i dadi tre volte consecutive, ottenendo sempre dadi doppi, se finisce nella casella “In jail”, o se ottiene l’imprevisto relativo) non sarà in gradi di muoversi per un massimo di 3 turni, al meno di pagare la tassa d’uscita, ma potrà comunque effettuare scambi, partecipare ad aste e lanciare i dadi, i quali gli consentiranno di uscire di prigione se risultano pari.


«Interface»
Board
«Interface»
Tile
«Interface»
Model
+getCurrentPlayer(): PlayerInfo
+getPlayers(): List<PlayerInfo>
+removePlayer(PlayerInfo: player)
+saveGame()
+exitDice(): Pair<Integer>
+getProperties(): Set<Obtainable>
+getBoard(): Set<Tile>
+movement(value: int)
+endTurn()
+exitFromJail(withFee: boolean)
+goToJail()
+executeTrade(...)
+unbuild(property: Obtainable, player: Player)
+playerPayment(player: PlayerInfo, moneyAmount: int): boolean
+playerGainMoney(player: PlayerInfo, moneyAmount: int)
+buyProperty(player: PlayerInfo, property: Obtainable)
+playerAddProperty(property: PlayerInfo, Obtainable: property)
+supplierConsequence(): Optional<ConsequenceImpl>
+getTileOf(position: int): Tile
«Interface»
Obtainable
«Interface»
Player
Turn

Capitolo 2

Design

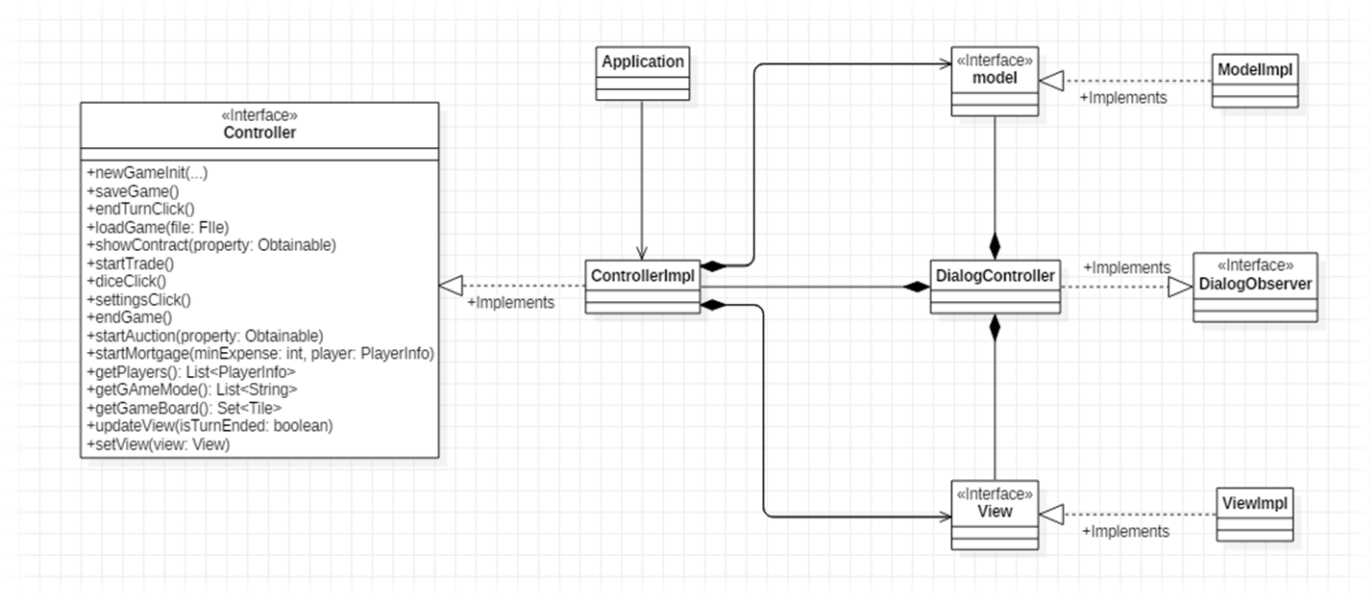
In questo capitolo vengono esposte le tecniche utilizzate per la risoluzione dei problemi riscontrati nella fase di analisi. Si parte da una visione architetturale che informa il lettore sul funzionamento dell'applicazione realizzata ad alto livello, concentrandosi su come i componenti principali del sistema si coordinano tra loro. Successivamente, vengono descritte in maggior dettaglio alcuni parti rilevanti del design al fine di chiarire la logica con cui sono stati risolti i problemi dell'applicazione.

2.1 Architettura

L'architettura del software è basata su MVC. La logica del gioco viene implementata all'interno del Model che si occupa della gestione logica del programma, memorizzando quindi al suo interno le entità che costituiscono il dominio del gioco stesso ed i metodi che ne permettono il corretto funzionamento. La View permette all'utente di interagire con il gioco tramite un'interfaccia grafica fluida ed intuitiva, svolgendo dunque la funzione di input, per registrare le scelte dell'utente, e di output, come ad esempio dialog relazionati alla conseguenza di determinate azioni, oppure messaggi a schermo che comunicano all'utente azioni non corrette oppure contro il regolamento o ancora azioni che hanno necessità di notifica poiché potrebbero passare inosservati, ma sempre in modo da non interferire con la giocabilità dell'applicativo. Il Controller, infine, mette in comunicazione il Model e la View, recependo le richieste effettuate dall'utente tramite l'interfaccia grafica e richiamando nel model i metodi necessari per il loro corretto svolgimento. Ogniqualvolta ce ne sia necessità il controller aggiorna la View permettendo la visualizzazione dei parametri modificati nel Model.

I principali metodi che vengono richiamati dal Controller per il Model sono il lancio dei dadi, il movimento, lo scorrimento della lista che tiene in memoria il susseguirsi dei turni, quindi la successione dei giocatori durante il gioco (in realtà questo viene gestito indirettamente dal model, infatti il susseguirsi dei giocatori nel gioco è contenuto nella classe **Turn**, della quale il model ne contiene un'istanza), e le varie informazioni ad esso rilegate (giocatore di turno, dadi doppi ecc), inoltre ad esso è affidata la logica per l'aggiunta e la rimozione delle proprietà ai vari giocatori, le varie spese, scambi commerciali e costruzione di case ed alberghi.

Il Controller, definito da Singleton, rendendolo quindi raggiungibile universalmente, mantiene al suo interno un'istanza del model e della View; la prima azione rilevante compiuta dal controller infatti tratta l'inizializzazione del gioco, richiamando la classe apposita che ne gestisce la logica, la quale permette di istanziare il model incapsulato all'interno del suddetto controller.
Il Contreller e la View raramente si scambiano richieste, ma spesso sono presenti particolari azioni, come la costruzione di una casa o lo scambio tra due giocatori di una o più proprietà, che necessitano una visualizzazione aggiornata ed intuitiva, inoltre un aggiornamento più generico della view si ha alla conclusione del turno di ogni giocatore, in modo da rendere chiaro agli utenti chi è il giocatore di turno e chi sarà il successivo e chi è stato il precedente.
Come detto in precedenza gran parte delle azioni eseguite dagli utenti, che sia l'acquisto di una proprietà, la partecipazione ad un asta, piuttosto che lo scambio di proprietà e/o denaro, avviene all'interno di particolari dialog con caratteristiche molto simili le une con le altre; è stato dunque deciso di implementare un controller "secondario", per una gestione più specifica di quest'ultime. Una caratteristica rilevante di questo "DialogController" è che questo comunica sia con il controller principale che con il model stesso, in questo modo ha facile accesso ad alcuni metodi utili (come il metodo per conoscere il giocatore di turno) evitando così ripetizioni e ridondanze nel codice.



2.2 Design dettagliato

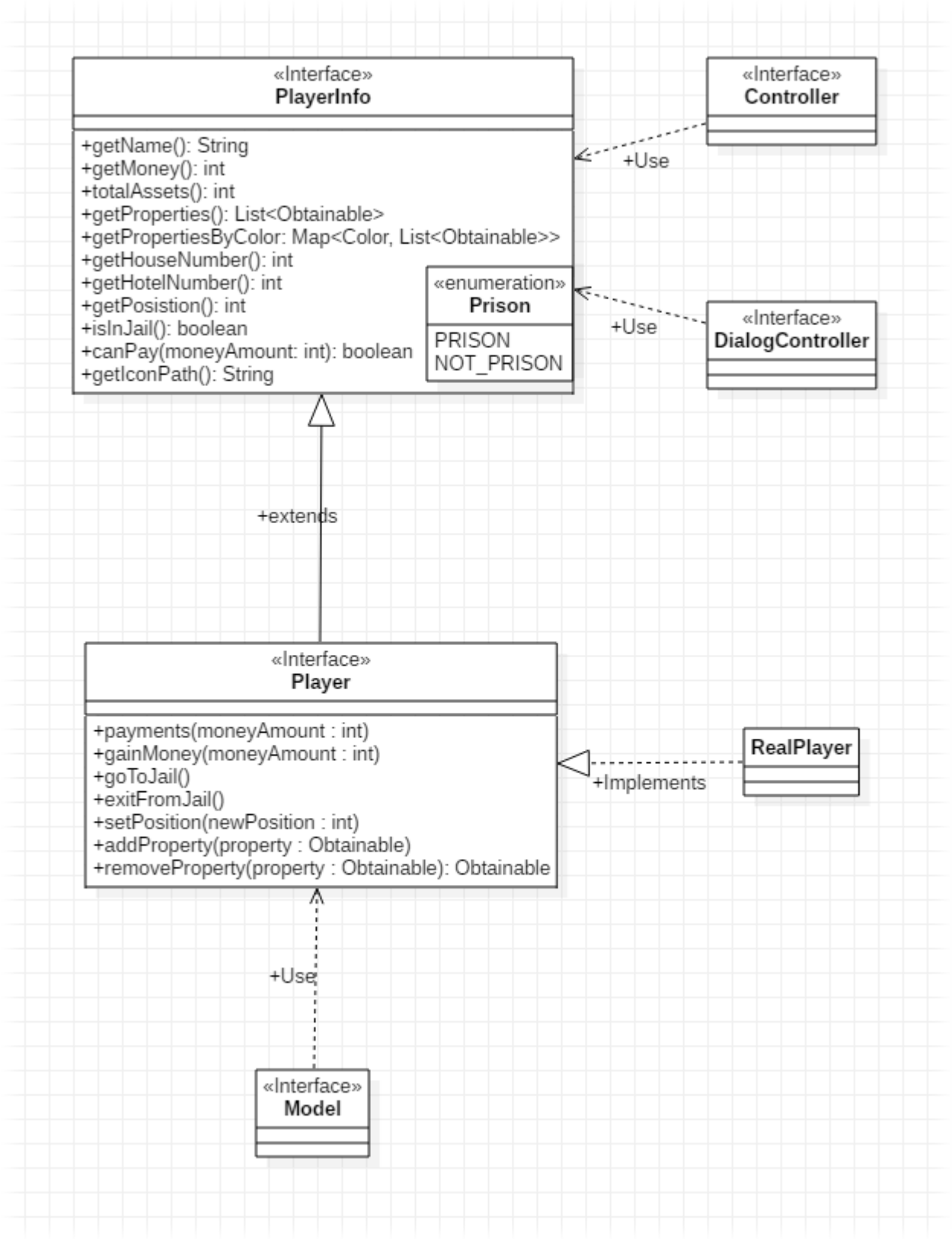
- **Rossolini Andrea (Gestione dei giocatori, Compravendita/asta, salvataggio e caricamento della partita, gestione del menù iniziale, inizializzazione della partita, gestione di musica e suoni, View: grafiche delle dialog e "*pane* informativi")**

All'interno del progetto mi sono occupato della realizzazione delle classi per la gestione dei giocatori, implementata con l'intento poi di realizzare un'intelligenza artificiale, ma poi, per mancanza di tempo, rimasta irrealizzata, la logica dietro la gestione della compravendita di proprietà, delle aste e di conseguenza degli scambi (di proprietà e soldi) che possono avvenire tra due giocatori, gestione del menù iniziale, dove è permesso al giocatore di iniziare una nuova partita o caricarne una precedentemente salvata, l'inizializzazione di una nuova partita, tramite un apposita schermata di gestione dei giocatori, la gestione della musica di sottofondo che accompagna il gioco e degli effetti sonori che vengono riprodotti quando un giocatore effettua determinate azioni (come ad esempio la costruzione di una casa)

- **Gestione dei giocatori**

Essendo il monopoli un gioco dove l'utente ha una grande importanza, dato che la view richiede frequentemente informazioni a suo riguardo, oltre a chiedergli di effettuare delle operazioni, alcune volte obbligatorie (come il pagamento di una tassa) mentre altre no (come la costruzione di una o più case); ho cercato di realizzare una classe che permettesse alla view di raccogliere tutte le informazioni che le interessavano senza però violare l'incapsulamento, quindi facendo in modo che questa non avesse la possibilità di accedere ai metodi che modificano le caratteristiche del giocatore. Il giocatore infatti si compone di una prima interfaccia "*PlayerInfo*", contenente i metodi 'getter' che permettono di accedere alle informazioni utili del giocatore, estesa da una seconda interfaccia, che racchiude i metodi utilizzati per apportare delle modifiche all'entità del giocatore, in fine l'implementazione di quest'ultima avviene tramite la classe "*RealPlayer*", che assunse questo nome poiché all'inizio si era pensato di affiancarla all'eventuale classe che avrebbe gestito l'intelligenza artificiale e ciò avrebbe permesso di distinguere le due classi con più facilità rispetto ad un nome come "*PlayerImpl*".

Il model gestirà la logica che sta dietro l'acquisto di proprietà, la bancarotta e la conseguente sconfitta del giocatore.

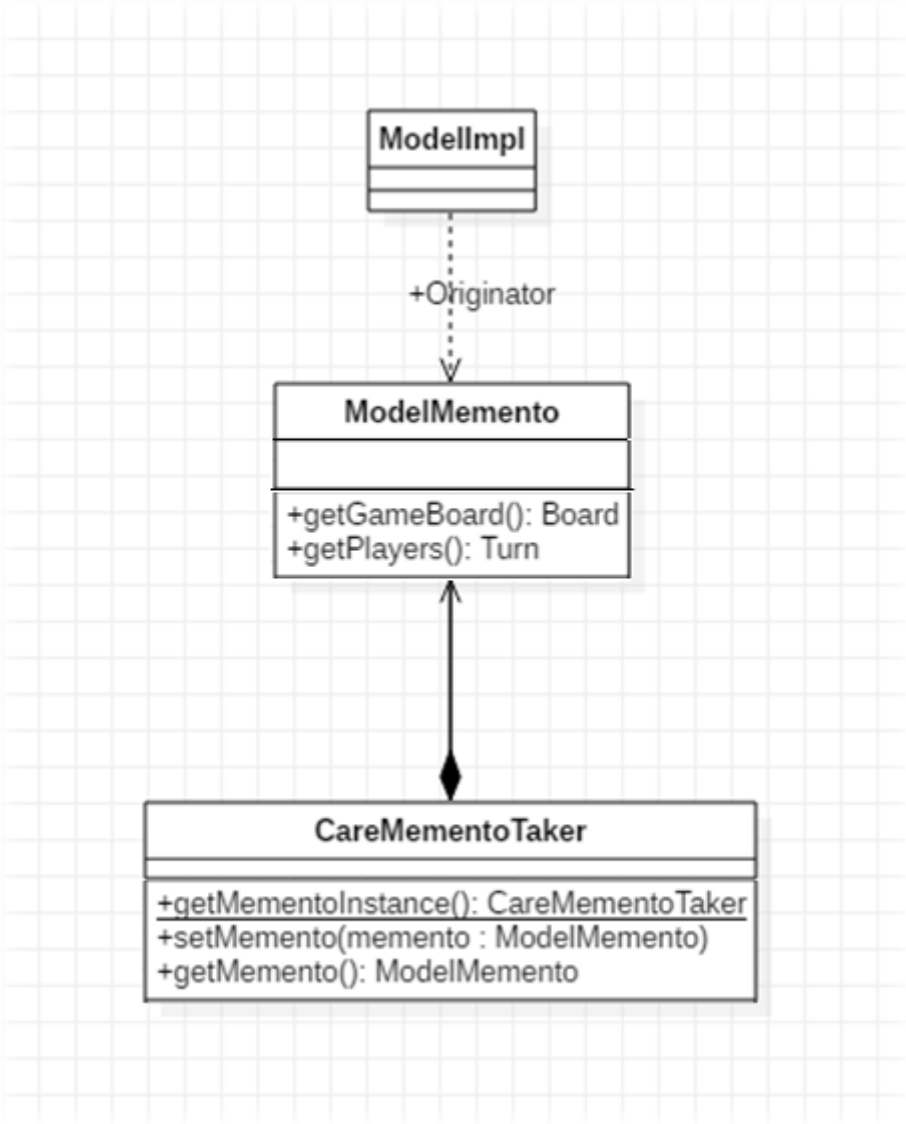

«Interface»
PlayerInfo
+getName(): String
+getMoney(): int
+totalAssets(): int
+getProperties(): List<Obtainable>
+getPropertiesByColor: Map<Color, List<Obtainable>>
+getHouseNumber(): int
+getHotelNumber(): int
+getPosistion(): int
+isInJail(): boolean
+canPay(moneyAmount: int): boolean
+getIconPath(): String
«Interface»
Controller
+Use
«enumeration»
Prison
PRISON
NOT_PRISON
+Use
«Interface»
DialogController
+extends
«Interface»
Player
+payments(moneyAmount : int)
+gainMoney(moneyAmount : int)
+goToJail()
+exitFromJail()
+setPosition(newPosition : int)
+addProperty(property : Obtainable)
+removeProperty(property : Obtainable): Obtainable
RealPlayer
+Implements
+Use
«Interface»
Model

- **Memento pattern**

Per implementare la funzione di salvataggio è necessario estrarre lo stato interno del model, facendo attenzione a non violare l'incapsulamento, in questo modo è possibile ripristinare quest'ultimo in un secondo momento allo stato in cui è stato salvato, il resto della partita (grafica, suoni ecc) possono essere reinizializzati tranquillamente come se si iniziasse una nuova partita. Per realizzare questa soluzione ho deciso di utilizzare il pattern 'Memento' (accennato dal prof durante la lezione riguardante i pattern di progettazione nel periodo di svolgimento del corso – precisamente le slide della settimana 12-13). In questo modo, ogni qualvolta il giocatore vorrà salvare la partita gli basterà fare click sull'apposito pulsante (il procedimento su come raggiungere tale pulsante è spiegato all'interno del manuale di gioco), verrà chiamato il metodo saveGame all'interno del Model (il quale risulterà quindi essere la classe 'Originator' del pattern Memento), tale metodo si occupa di richiedere al "*CareMementoTaker*" di creare un nuovo oggetto di tipo *ModelMemento,* passandogli quelle che sono le risorse essenziali per l'inizializzazione del gioco salvato.

Tramite la serializzazione delle classi interessate dal ModelMemento è possibile scrivere quest'ultimo su file, generando un file di estensione '.ubp' all'interno della cartella '.Unibopoli' (anch'essa creata dal gioco, in caso non fosse già presente); ogni volta che si decide di salvare la partita viene generato un nuovo file che ha per nome la data e l'ora (utilizzando l'orario UTC del sistema) del momento del salvataggio, quando invece si vorrà ricaricare una partita verrà aperto l'esplora risorse di sistema che permetterà di selezionare il file '.ubp' desiderato e conseguentemente caricare la partita. La scrittura e lettura dalla memoria è gestita dalla classe "*ResourceManager*".

Questo tipo di pattern risulta molto efficiente in caso in futuro si volessero realizzare e rendere serializzabili nuove implementazioni, infatti, modificato il solo ModelMemento sarà possibile salvare più informazioni.



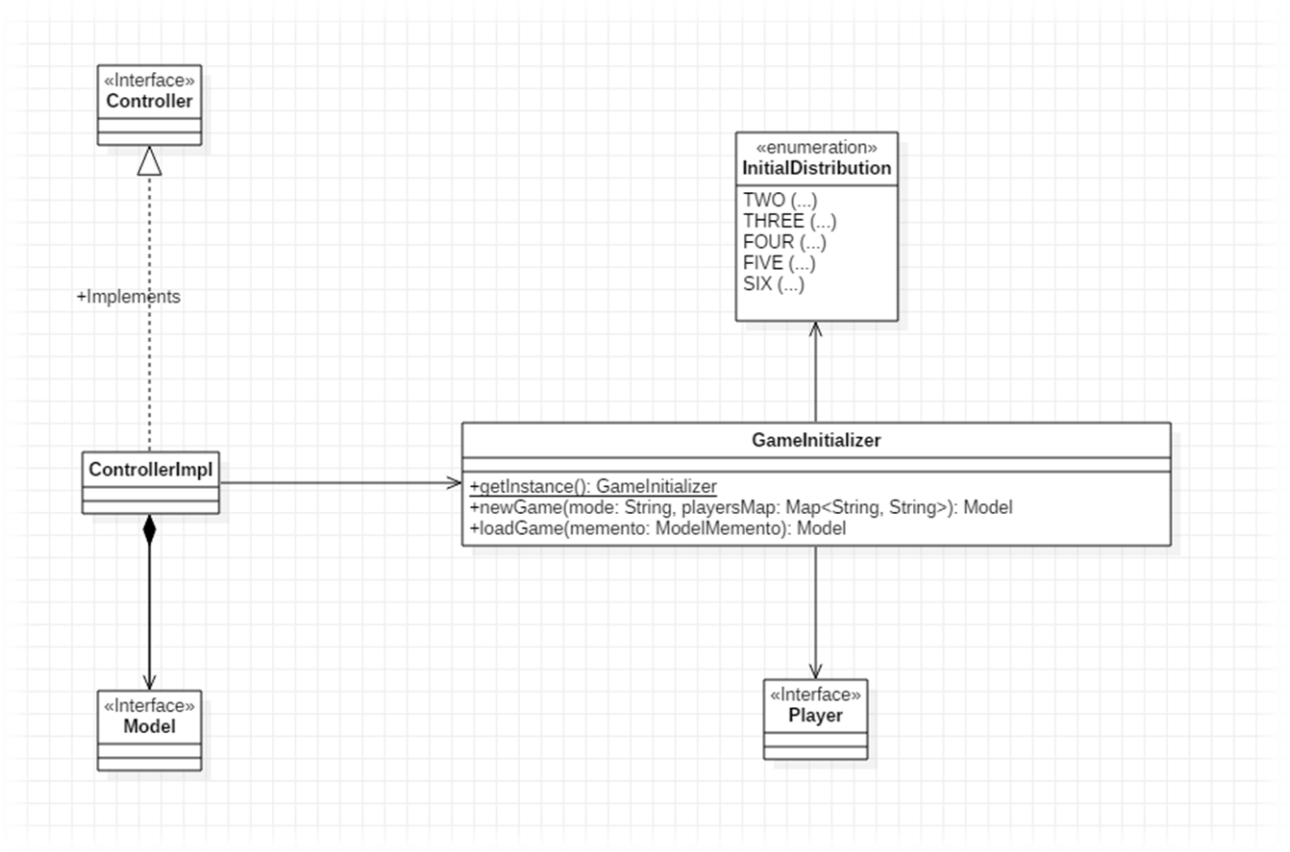
- **Inizializzazione della partita**

Per l'inizializzazione del gioco è stata realizzata un'unica classe adibita unicamente a questo compito, gestita tramite **Singleton**, assicurando in questo modo una sola istanza durante tutta la partita, sia che si tratti dell'inizializzazione di una nuova partita (*newGame(…)*) che il caricamento di una partita precedentemente salvata (*loadGame(…)*); ambedue i metodi hanno lo scopo di inizializzare e restituire il model che andrà poi a gestire la logica di tutto il resto della partita. Una variabile booleana settata a 'true', nel momento in cui viene chiamata la classe, evita che vi possano essere erroneamente inizializzate più partite contemporaneamente.

Il metodo per l'inizializzazione di una partita da zero vuole in ingresso due parametri, il primo indica il tipo di modalità con cui la board di gioco dovrà inizializzarsi, mentre il secondo parametro è una mappa che ha nel set di chiavi i nomi dei giocatori scelti dagli utenti tramite l'apposita schermata di setup, mentre come valori, ognuno legato alla rispettiva chiave, il percorso dove sono contenute le

immagini degli avatar scelte dai giocatori (anch'esse nella medesima schermata). Il compito di questo metodo dunque è quello di attribuire il giusto numero di soldi e di proprietà ad ogni giocatore, questo numero è contenuto all'interno di un **enum**, chiamata '*InitialDistribution'* contenuta all'interno del package "utilities.enumerations". Poi tramite le stream di java vengono scelte casualmente delle proprietà (inizializzate ed instanziate all'interno della board) e di conseguenza vengono tolte al giocatore le giuste quantità di denaro dalla quantità inizale (come definito dalle regole originali del monopoli).

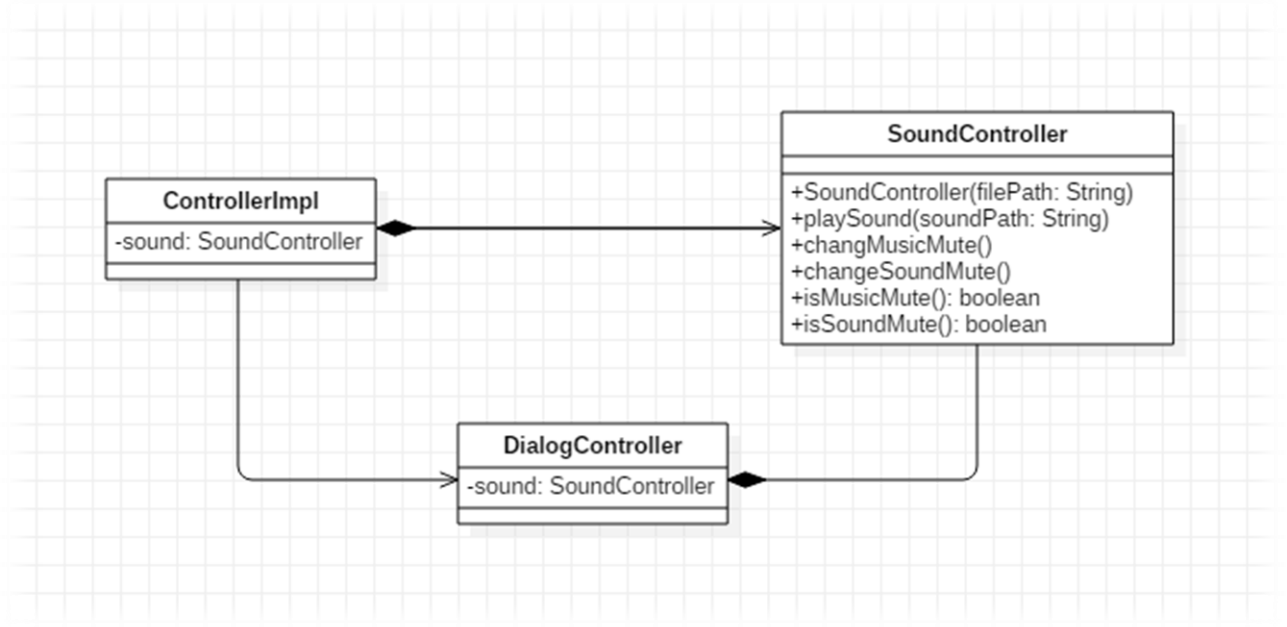
L'inizializzazione di una partita salvata precedentemente avviene tramite il metodo che prende in ingresso un oggetto di tipo *ModelMemento* (vd. Paragrafo "Memento pattern"), da questo estrapola le informazioni necessarie per poter generare un model identico a quello che è stato salvato.



- **Gestione di musica e suoni**

La gestione della musica di sottofondo ed i suoni che si possono incontrare lungo il corso della partita (lancio dei dadi, aggiunta o rimozione di una casa, acquisto di una proprietà, finire in carcere, essere eliminati o vincere la partita), sono tutti gestiti dalla classe "*SoundController*", gestendo dunque il multithreading per la riproduzione di più file audio contemporaneamente (possono essere riprodotti

esclusivamente file di tipo '.wav'). La classe viene inizializzata passandole il path dove si colloca la canzoncina di sottofondo (della durata di circa 5 o 6 minuti) e viene riprodotta in loop, a meno che l'utente non decida di interromperla, in questo caso viene settata una variabile booleana che mette in pausa il loop fino ad una nuova richiesta dell'utente. Per la riproduzione di suoni il modus operandi è molto simile, infatti l'utente può decidere a priori se riprodurre suoni o meno, sempre settando una seconda variabile booleana; in caso si volesse riprodurre un suono viene chiamato un particolare metodo che, dopo aver controllato che la suddetta variabile booleana sia settata a true, lancia il thread che riprodurrà il suono desiderato.



- **Finestre di dialogo e schermate di gioco**

Partendo dalle prime schermata che ci si presentano all'avvio dell'applicativo (package "view.gameSetup"), vi è presente il menù iniziale ("*MainMenu*"), dove, come già ripetuto più volte, è possibile scegliere se cominciare una nuova partita o caricarne una già salvata; conseguentemente si può, cominciando una nuova partita, accedere alla schermata di setup dei giocatori ("*PlayerSetupMenu*"), la quale utilizza la classe "*PlayerSetupBox*" per creare i "Box" nel quale l'utente inserisce il nickname e sceglie l'icona che lo rappresenterà nella board di gioco, l'unica particolarità che può essere interessante è l'implementazione di una "*CellFactory*" self made per poter visualizzare in maniera piacevole tutti gli avatar di gioco disponibili all'interno della comboBox utilizzata dal giocatore, per scegliere, appunto, un avatar.

Il package "gameDialog" contiene tutte le dialog che, durante il gioco, permetteranno agli utenti di interagire integramente con l'applicativo, tutte quante utilizzano il **Singleton** pattern, poiché hanno necessità di essere raggiunte con

facilità da Controller e View, inoltre non sono state utilizzate interfacce od altri pattern particolare, poiché la loro realizzazione, a mio parere, avrebbe reso il codice più complesso da analizzare e da manutenere, oltre al fatto che i numerosi metodi diversi avrebbero reso completamente inutile un interfaccia. Dato che durante la progettazione abbiamo pensato di attribuire a tutte queste dialog dimensioni simili e simili caratteristiche, abbiamo deciso di effettuare una sorta di standardizzazione tramite l'utilizzo di una classe *"Dialog"*, la quale determina dimensioni generali, dimensioni dei bottoni, decorazioni, sfondo eccetera; questa classe verrà poi estesa da tutte le Dialog che di seguito sono presentate:

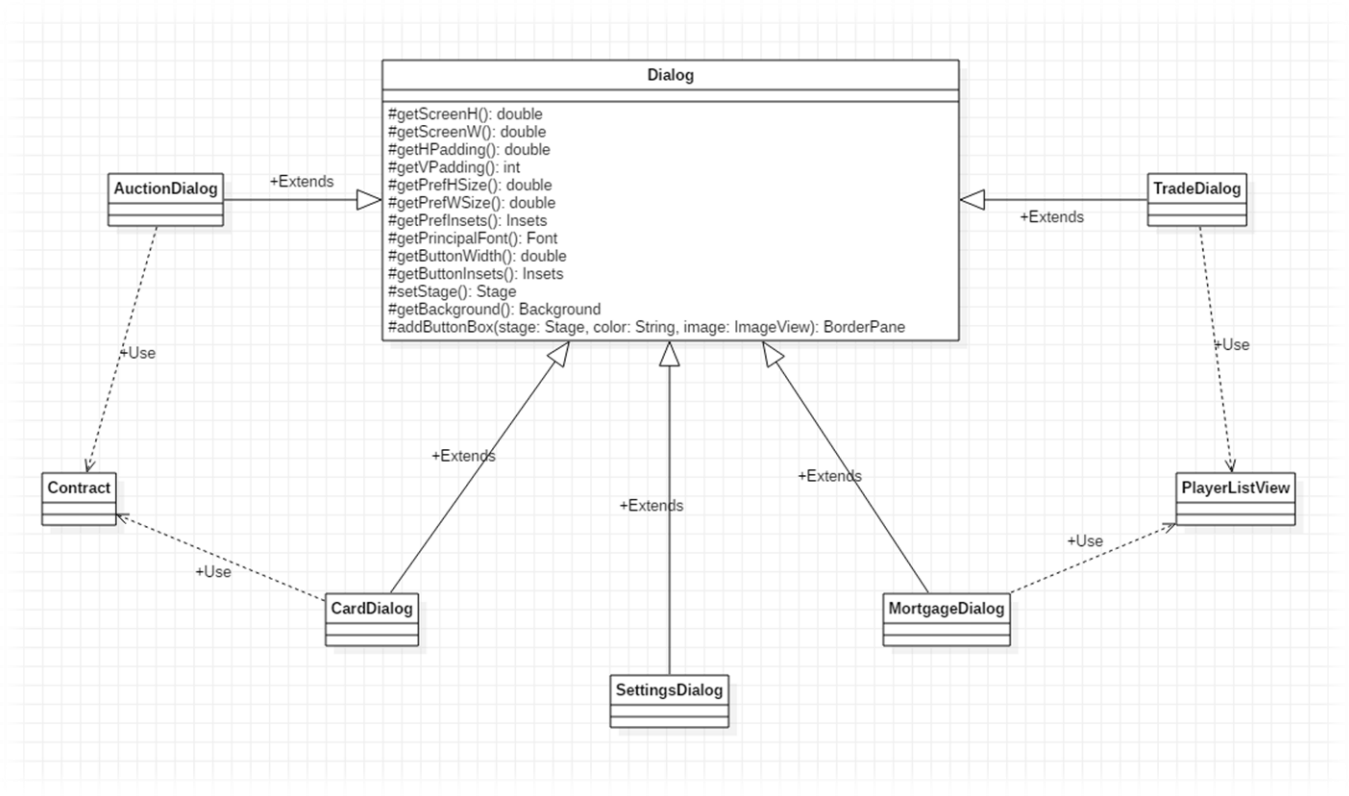
- *AuctionDialog*: Dialog chiamata solo nel particoalre caso in cui un giocatore abbia rifiutato l'acquisto di una casa o un giocatore perde la partita, questa classe utilizza anche la classe "*Contract*" che spiegherò subito dopo.
- *CardDialog*: Questa è la dialog più frequente, ti permette di visualizzare, oltre che il contratto di una determinata carta, anche le sue caratteristiche meno auto evidenti, come ad esempio il proprietario, il numero di case e se è ipotecata o meno; inoltre permette l'interazione con la proprietà stessa, infatti oltre all'acquisto permette (se il giocatore al momento ne è in grado) di costruire o rimuovere case e di ipotecare la proprietà. Anche questa utilizza Contract.
- *MortgageDailog*: Creata quando un giocatore non è in grado di affrontare una spesa, costringendolo ad ipotecare alcune delle sue proprietà. Questa Dialog utilizza la classe "*PlayerContractListView*" che, come Contract, spiegherò nelle prossime righe.
- *TradeDialog*: Viene utilizzata per permettere ai giocatori di effettuare scambi tra di loro all'interno del gioco, mentre il giocatore corrente è preselezionato, questo, tramite una comboBox, dovrà selezionare l'altro giocatore con cui vorrà interagire. Anche questa utilizza PlayerContractListView
- *SettingDialog*: L'unica dialog che si distingue particolarmente dalle altre, ma per comodità è stato deciso di implementarla all'interno di questo package, pur estendendo sempre "*Dialog*", molto semplicemente ti permette di interagire con la musica ed i suoni del gioco anche durante la partita, oltre che il salvataggio della partita. Al di là dei metodi che vengono richiamati, di questa dialog è quasi più interessante un analisi sullo Style CSS che javaFX permette di collegargli.

La classe "*Contract*" estende AnchorPane di JavaFX e permette la visualizzazione "real like" di un contratto del monopoly, distinguendo tra i vari contratti (ad esempio le stazioni sono visivamente diverse da "Vicolo Stretto"), in questo modo è facile per l'utente ottenere informazioni riguardo ad un particolare contratto, oltre al fatto che permette una visualizzazione efficace e gradevole in

caso si volessero implementare delle 'mod' per il gioco, l'unica difficoltà che ho riscontrato durante il suo sviluppo è stata l'adattabilità alle diverse risoluzioni dello schermo.

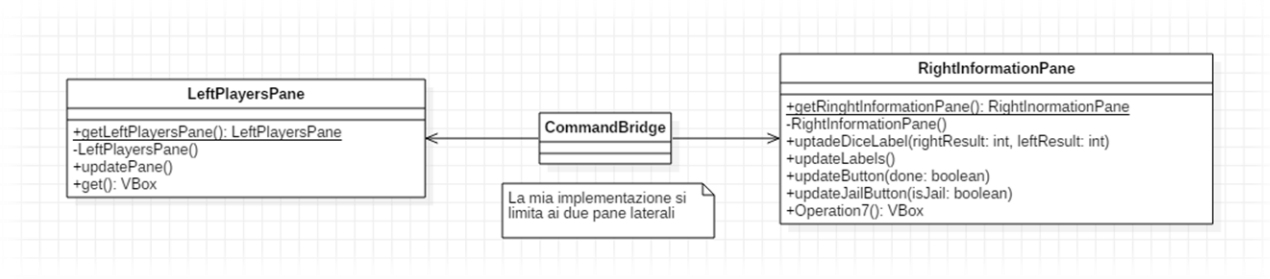
La classe "*PlayerContractsListView*" estende ListView di JavaFX, permettendo una visione piacevole della lista dei contratti appartenenti al giocatore, suddivisi per colore di appartenenza. Il suo funzionamento consiste nel "collezionare" gli elementi cliccati dell'utente (facendoli quindi divenire di colore grigio), all'interno di una lista, che verrà poi restituita in base alle esigenze (il fatto che il ritorno sia una lista opzionale di stringhe è puramente zucchero sintattico, dato che un giocatore può anche non scegliere nulla, ma in quel caso sarebbe tornata una lista vuota).

** Il completo funzionamento delle Dialog esposte è spesso esplicitato nel manuale di gioco.*



In ultima analisi considero i Panel laterali ("*LeftPlayersPane*" e "*RightInformationPane*") che accompagnano la board di gioco; la loro realizzazione è piuttosto banale:
Nel Pane di sinistra viene utilizzato un Effect di JavaFX per indicare chi è il giocatore di turno, questo effetto viene aggiornato ad ogni turno, inoltre un Tooltip mostra le proprietà appartenute da ogni giocatore (L'utilizzo di Tooltip è

stato spesso trascurato nella stesura della relazione poiché lo ritengo relativamente importante).
Nel Pane di destra invece vengono riportate le informazioni riguardanti il giocatore (nome, soldi ecc), il bottone che permette il lancio dei dadi (indicato dal disegno di due grandi dadi), il bottone che apre gli scambi tra i giocatori, il pulsante per il cambio di turno o il pagamento della tassa di prigione e quello per i settings.

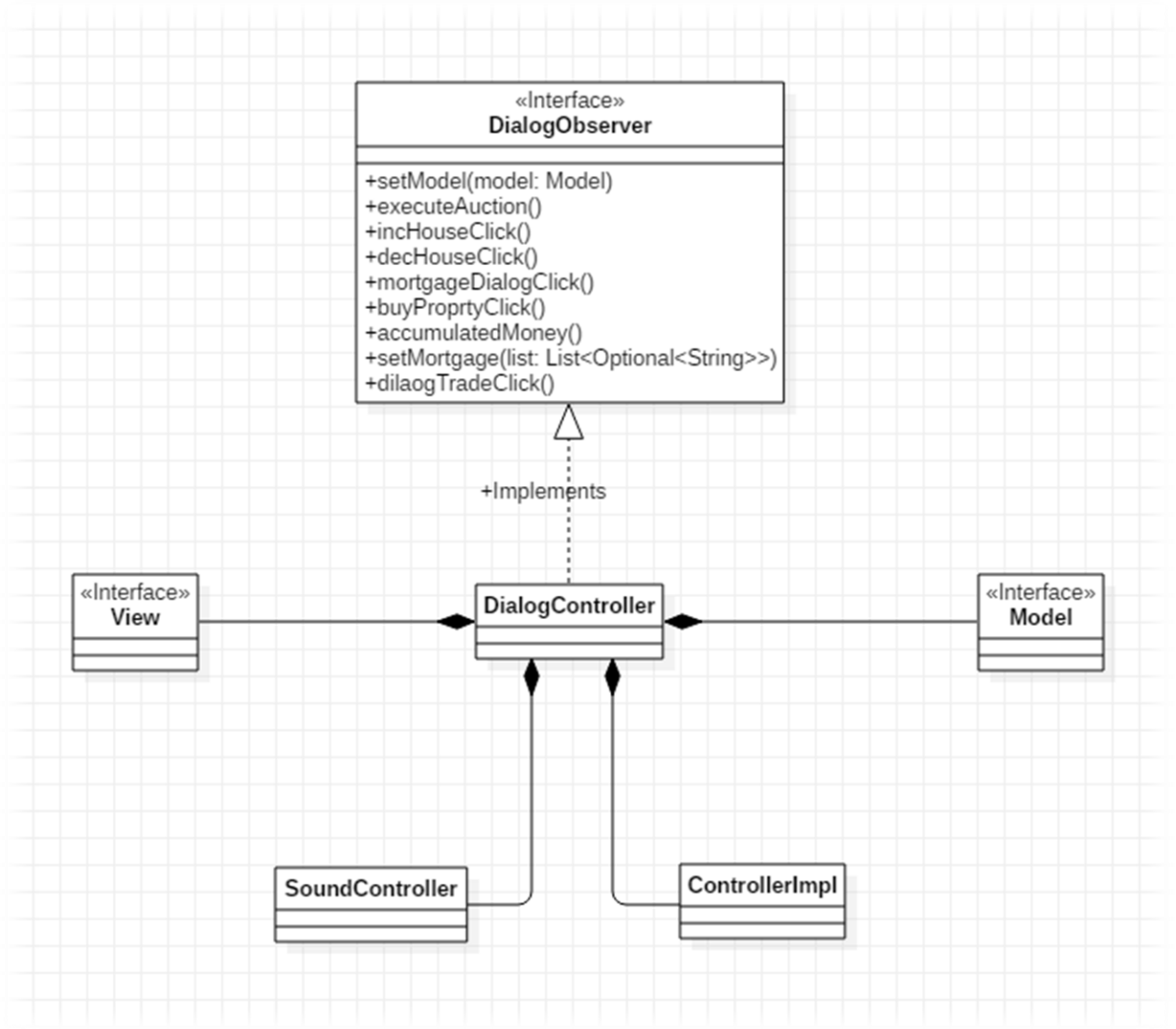


- **DialogController**

Per la gestione dei numerosi dialog per l'interazione con l'utente è stato deciso di implementare un controller dedicato unicamente alla loro gestione, questo perché aver utilizzato il controller principale lo avrebbe reso di difficile comprensione. Questo controller, gestito anch'esso tramite **Singleton,** implementa un interfaccia che funge come una sorta di Observer per le dialog, le quali, a richiesta dell'utente (quindi generalmente tramite la pressione su dei bottoni) ,verranno richiamate ed eseguite, al di là di azioni illegali. Quasi tutti questi metodi fanno uso del model, che viene settato dal controller e quindi incapsulato al momento dell'inizializzazione della partita, eccezion fatta per quei metodi che elaborano una richiesta di aggiornamento diretto della view, come ad esempio il metodo accumulatedMoney() che aggiorna in tempo reale i soldi accumulati dal giocatore nel momento in cui sceglie le proprietà da ipotecare all'interno della MortgageDialog (vd. Paragrafo sulle finestre di dialogo o il manuale di gioco). È possibile notare come tutti i metodi presenti nell'interfaccia non abbiano un valore di ritorno, in luce del fatto che la view (gameDialog) manda unicamente dei comandi al DialogController e mai richieste. Nonostante questo controller sia adibito alla comunicazione tra le dialog ed il model, i metodi che visualizzano a schermo queste dialog sono invece presenti unicamente nel Controller principale, i pochi valori comuni che hanno il Controller e questo DialogController ci hanno fatto arrivare alla conclusione che un eventuale estensione del Controller primario sarebbe stato controproducente e concettualmente sbagliato. Le sue funzioni principali sono dunque:

-*executeAuction*: per eseguire le aste.
-*setMortgage*: per permettere di ipotecare le proprietà.
-*inc/decHouse:* Che permette di costruire o demolire le case su di una proprietà.
-*buyPropertyClick:* Per acquistare le proprietà.

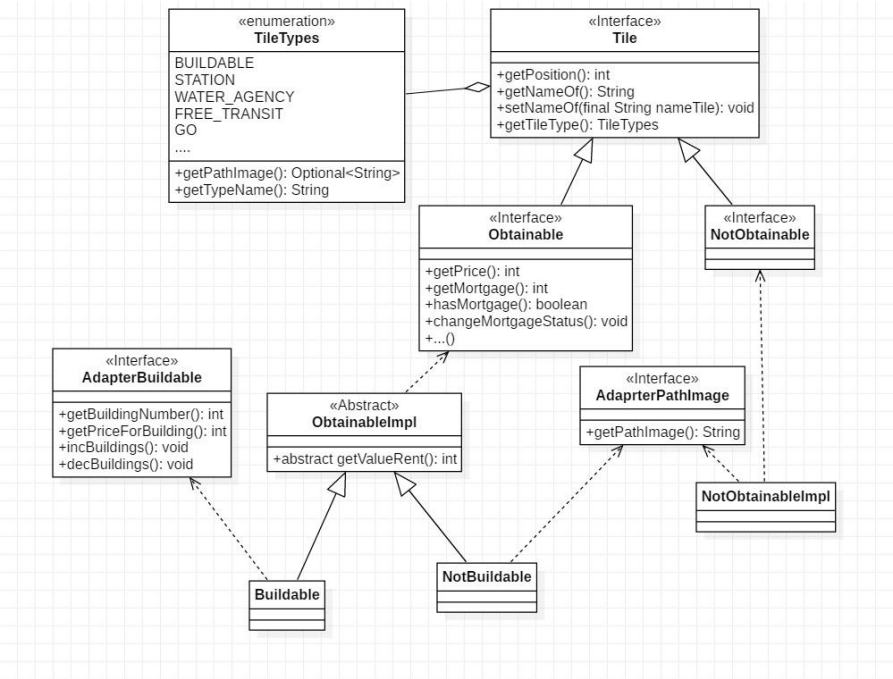
-*dialogTradeClick:* Che esegue lo scambio di proprietà e denaro tra due giocatori.



- **Alesiani Matteo (Gestione del movimento della sua conseguenza, Sviluppo dei contratti da gioco e degli imprevisti e probabilità, gestione del turno dei giocatori e grafica della board di gioco)**

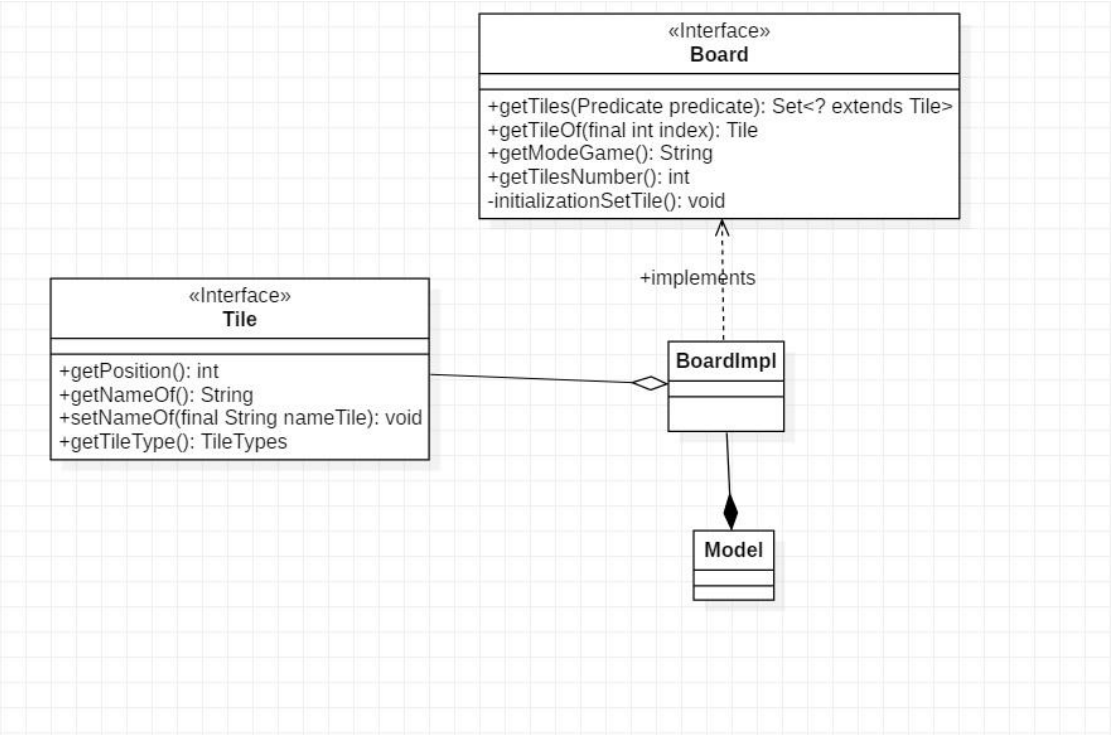
  - **Tiles**

Nel normale svolgimento del gioco, ogni giocatore si troverà all'interno di una casella con apposite proprietà. Il seguente paragrafo ha l'intento di esplicare come vengono modellate le caselle della board e i rispettivi contratti di proprietà. La natura eterogenea di quest'ultime ha portato il team a suddividerle in categorie e successivamente creare una gerarchia per gestire ogni loro aspetto.
Come prevedibile che fosse, la prima proposta implementativa generava una gerarchia di classi troppo ampia ed estesa con il rischio di grandi difficoltà per una futura modifica del codice. Analizzando alcuni patterns si è ridotta la gerarchia a due soli livelli e partendo dall'interfaccia padre ("**Tile**"), essa si estende in ottenibili dal giocatore ("**Obtainable**"), ovvero proprietà, stazione e società, e non ottenibili ("**NotObtainable**"). La suddivisione permette di inserire all'interno dell'interfaccia padre tutti i metodi comuni, quali posizione e nome della casella e demanda alle sottoclassi la specializzazione. Per completare l'operazione vengono implementate due ulteriori interfacce: **AdapterPathImage** e **AdapterBuildable**, che come si evince dal nome, appartengono all'Adapter Pattern.
La prima strutta l'ereditarietà e, grazie all'attributo di tipo **TileTypes**, restituisce il path dell'immagine che la casella dovrà contenere nella board. Non tutti i contratti hanno bisogno dell'immagine, basti pensare alle proprietà, quindi analizzando l'attributo è possibile sapere anche la natura.
La seconda, invece, sfrutta la delegazione della classe **Rent** per restituire l'affitto della proprietà in base al numero di case e/o hotel costruiti su di essa. L'unica classe che la implementa è Buildable.
Infine va precisato che la classe ObtainableImpl e ti tipo astratto: cosi facendo si ha la possibilità di definire un algoritmo (o nel nostro caso un metodo) lasciando alle sottoclassi la possibilità di modificare alcune parti senza alterarne la struttura della classe. Quindi il metodo **getValueRent**() è di ausilio alle sottoclassi per restituire l'affitto corretto.



- **Board**

La plancia sulla quale si svolge l'attività di gioco, viene inizializzata e gestita all'interno della classe BoardImpl. Al suo interno si memorizza l'insieme di Tile che costituiscono la plancia e la mode corrente di gioco. All'interno del costruttore avviene l'inizializzazione automatica del gioco, consentita grazie ad alcune classi collocate, all'interno del package utilities, come ReadFile, ClassicType e Parse.
ReadFile mette a disposizione un metodo statico che restituisce uno Stream<String> in base al percorso passatogli.
ClassicType contiene tutti i path utilizzati all'interno del progetto. La sua presenza è di primaria importanza e porta numerosi benefici al progetto, tra cui la leggibilità del codice e la possibilità di concentrare in un unico punto componenti che potrebbe subire modifiche. I cambiamenti apportati al suo interno ricadranno a cascata in tutto il progetto, senza dover modificare tutte le classi che lo utilizzano.
Parse, infine, dispone di metodi statici per trasformare le sterili righe dei file da String a rispettivi oggetti.
Come ultimo compito, la board dovrà inizializzare gli imprevisti e le probabilità, che pescheranno i giocatori durante la gara, attraverso la classe CardEffectSupplier (verrà spiegata in seguito).

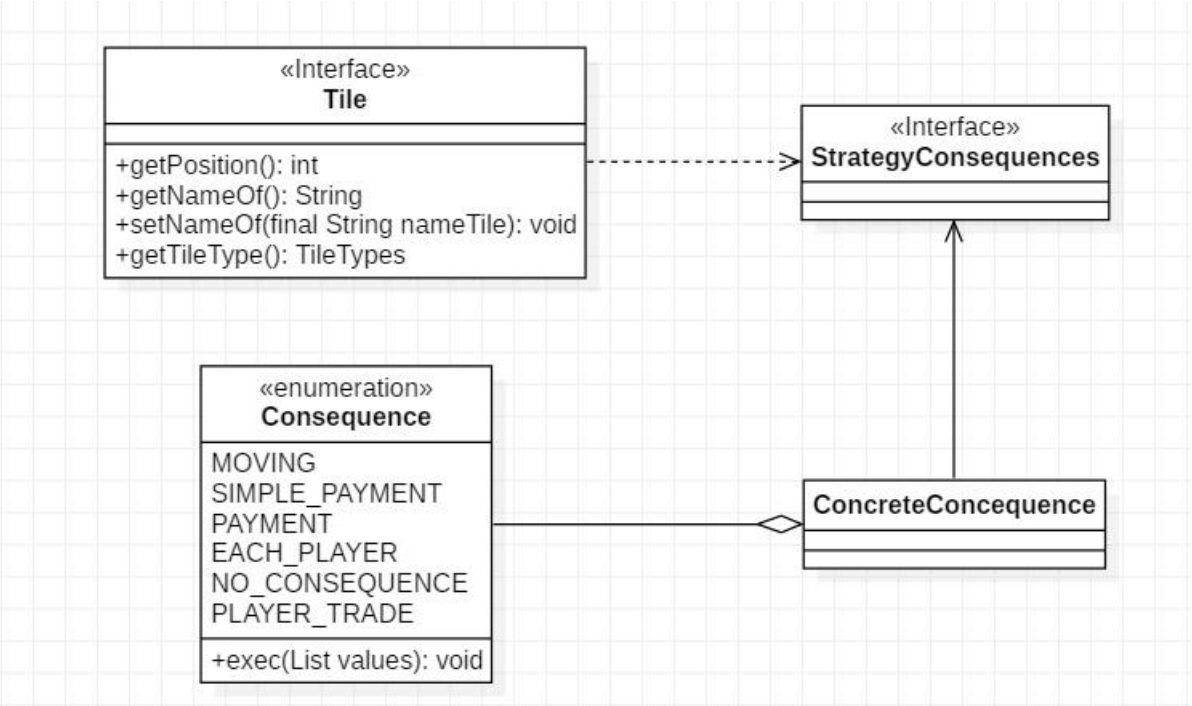


- **Conseguenze**

La caratteristica peculiari dei giochi da tavolo è la presenza di possibili conseguenze; ovvero l'esecuzione di un'azione porta il giocatore a di fronte a sanzioni o ricompense. Anche in questo caso la loro natura è di vario genere, soprattutto generati in differenti circostanze da oggetti diversi. La soluzione utilizzata è quella di adoperare il Strategy Pattern implementando le seguenti strutture:

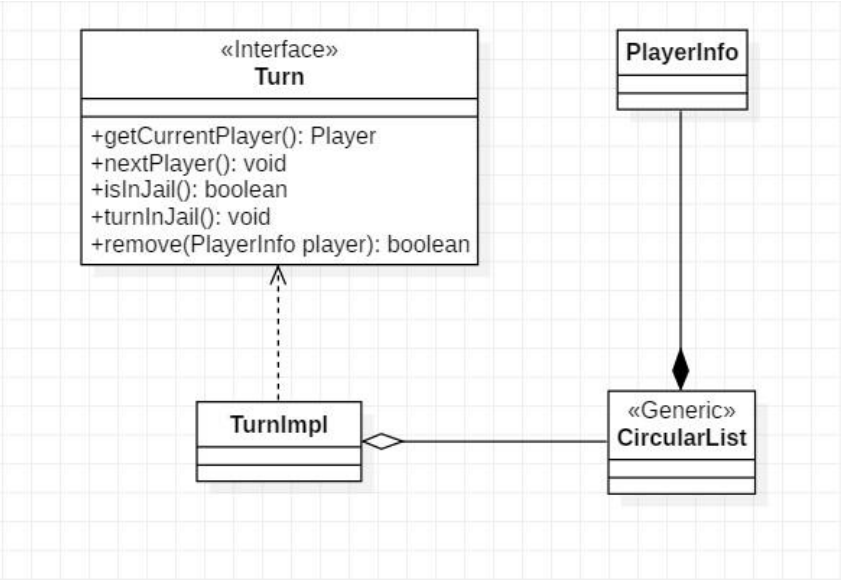
1. StrategyConsequences è l'interfaccia comune alla famiglia di algoritmi.
2. Gli oggetti che estendono Tile sono le strutture su cui agisce il pattern.
3. L'enumeratore Consequences contiene elementi contenenti l'algoritmo specifico
4. ConcrateConsequences può essere considerato come un wrapper di Consequences ed inoltre invoca l'esecuzione del relativo codice.

Cosi facendo si separa in oggetti diversi il comportamento che agisce sullo stato permettendo grande flessibilità di combinazione, anche a runtime, cosa non possibile con l'ereditarietà.
Una nota di riguardo va fatta alla classe CardEffectSupplier sopracitata. Essa è costituita da un Singleton che permette, quindi, di avere un'unica istanza della classe. Contiene le probabilità e gli imprevisti sfruttando due liste circolari, dalle quali verrà estratto un elemento come conseguenza al movimento di un giocatore.



- **Turn**

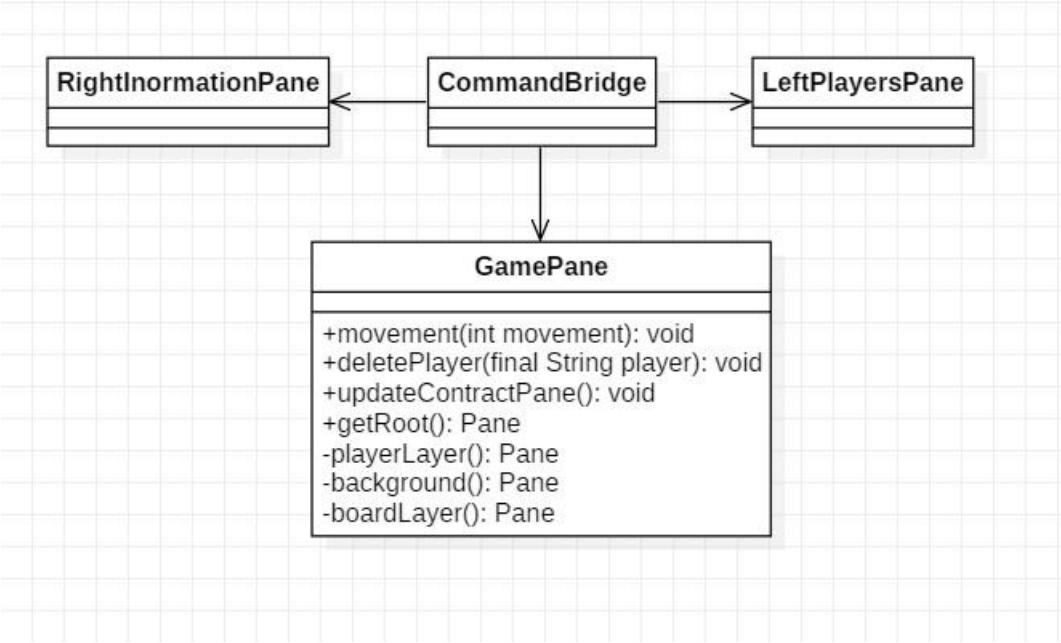
La dinamica di gioco consiste nel continuo passaggio da un giocatore all'altro in maniera ciclica. Partendo da questo presupposto, la risposta al problema si basa su una lista circolare contenuta nella classe generica CircularListImpl<E>. Quest'ultima viene istanziata in TurnImpl, dove si eseguiranno operazioni solo al giocatore in testa alla lista; colui che nel progetto prenderà il nome di CurrentPlayer. La classe TurnImpl svolge altre funzioni quali la memorizzazione dei giocatori man mano che vengono eliminati dal gioco, verificare se il giocatore si trovi il prigione e garantire che ci finisca qualora vengono tirati 3 volte consecutive dadi doppi.



- **GamePane e CommandBridge**

In fase di creazione di una nuova partita, il processo incontrerà innanzitutto la classe CommandBridge. Essa organizza lo Stage suddividendolo in tre parti distinte, LeftPlayersPane e RightInormationPane lateralmente (discussi in precedenza), e GamePane, collocato al centro e contenente la board di gioco.
Per ricreare la plancia originaria e tipica del gioco, GamePane sfrutta le potenzialità del layout StackPane e dispone gli elementi impilandoli uno sopra l'altro, come in una pila. Gli strati del pane sono restituiti da background(), il quale è collocato sul fondo e rendo il tavolo del tipico colore, playerLayer() che inserisce e gestisce le Icon del player, spostandole di volta in volta, ed infine viene inserito boardLayer(). Quest'ultimo con l'ausilio della classe LandAbstractFactory genera ogni cella e la dispone lungo il perimetro. La classe LandAbstractFactory mette a disposizione il metodo createLand(), il quale elabora le informazioni contenute in ogni tile, allo scopo di realizzare la corrispondente cella. In particolare, dopo aver effettuato un controllo sulla natura della tile, restituisce la cella della dimensioni e dell'aspetto opportuno.
La descrizione di questa procedura termina con la presentazione della classe ComponentFactory. Essa mette a disposizione dei metodi statici che restituiscono i singoli elementi che compongono ogni cella.

I compiti svolti dalla classe comprendono anche la visualizzazione, nella parte centrale, dei contratti posseduti dal giocatore corrente, i quali alla fine del turno vengono rifrescati, inserendo quelli del giocatore successivo.



- **Movimento**

Il lancio dei dadi genera, oltre alla modifica della posizione del giocatore e il conseguente spostamento della propria pedina, il quale rappresenta l'essenza del gioco.

Il primo problema nato durante la risoluzione è che esso non segue un comportamento determinato e periodo, e quindi l'ipotesi di implementare un GameLoop, che controllava periodicamente lo stato delle pedine, fu scartata sul nascere.

Analizzando le caratteristiche della libreria JavaFx, si è arrivati alla conoscenza della classe PathTransition. Essa effettua lo spostamento grafico di un nodo dello Stage in base a specifiche indicazioni e coordinate, che compongono il percorso che si intende effettuare. Da questo ne deriva l'implementazione dell'enum Direction. Ogni suo elemento restituisce la coordinata successiva da dover raggiungere con la pedina del giocatore. La logica che si segue è quella di divide lo spostamento in piccoli tratti, ognuno dei quali ha un inizio ed una fine, questo agevola l'implementazione quando, inevitabilmente, la pedina giungendo all'angolo, non deve continuare il suo percorso, bensì ruotare il verso che sta percorrendo e proseguire.

Ultimata la creazione del percorso, la classe Movement genera lo spostamento grafico della pedina.

Nel normale svolgimento dell'azione appena esplicata, viene utilizzata la classe Pawn. Essa estende Icon, ed oltre ad incorporare un ImageView corrispondente alla pedina possiede un attributo di tipo Pair<> che tiene traccia della posizione nella plancia.

Alla classe Movement viene passato il riferimento a Pawn e al Path che deve essere percorso ed infine viene creato ed eseguito un PathTransition della durata di un secondo.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per il testing automatizzato è stata utilizzata la suite di JUnit per verificare il corretto funzionamento di alcune particolari classi. I test implementati riguardano le parti critiche del gioco, come la gestione dei giocatori, l'inizializzazione della partita, il salvataggio e caricamento del gioco tramite il memento pattern.

I test disponibili sono all'interno del package "*test*":

- PlayerTest: Testa la gestione dei giocatori.
- MementoTest: Testa il funzionamento del pattern memento e del salvataggio/caricamento di questo.
- GameInitTest: Mostra il funzionamento dell'inizializzazione del gioco, simulando anche una superficiale partita per valutare le azioni eseguite dal model.

Il corretto funzionamento dell'interfaccia grafica è stato eseguito (per forza di cose) manualmente.

3.2 Metodologia di lavoro

Il progetto inizialmente era stato pensato per essere svolto da un gruppo di 3 persone, infatti inizialmente abbiamo iniziato ad organizzarci, analizzando il dominio dell'applicativo tutti insieme, scomponendo il problema in problemi più piccoli ed analizzandoli volta per volta, costruendo delle linee guida da seguire su tutti gli ambiti, in modo che ognuno potesse realizzare la propria parte avendo comunque un idea su cosa avrebbero fatto i suoi compagni, oltre che a dare la possibilità ad ognuno di noi di scambiare opinioni su ogni argomento e qualche tip implementativo. Attribuendo a grandi linee un livello di difficoltà ad ogni problema abbiamo cercato di dividerci equamente i compiti da svolgere: Matteo Alesiani si sarebbe occupato della creazione della board, della gestione del movimento e le sue conseguenze, quindi dei contratti, degli imprevisti e delle probabilità, Andrea Rossolini avrebbe gestito la logica dei giocatori, il sistema che sta dietro la compravendita e l'asta dai contratti, parte del compartimento grafico (caselle e

dialog per la comunicazione con l'utente), il salvataggio ed il caricamento da file della partita, in fine Edoardo Doglioni avrebbe dovuto gestire il Menù iniziale e tutto ciò che ne consegue, come l'inizializzazione della partita o la riproduzione di musiche e suoni, e la gestione del turno (quindi il susseguirsi dei giocatori durante la partita). Per lavorare al meglio decidemmo di incontrarci periodicamente, almeno una volta a settimana, o fisicamente o tramite videochiamate, organizzando delle sorte di 'briefing' di circa un'ora o due dove ogni membro del gruppo presentava il proprio lavoro svolto dal briefing precedente, esponendo eventuali dubbi personali agli altri membri del gruppo, oltre che a consigli o critiche costruttive; questo approccio è stato utile anche all'integrazione della parte del software realizzata insieme, ovvero la parte riguardante View, Controller e model, oltre che altri elementi della View come i Pane Informativi (Andrea Rossolini) con la board di gioco (Matteo Alesiani). Ma col passare del tempo lo studente Doglioni, presentava magari domande o chiedeva consigli senza mai presentare nulla di fatto. Infatti, nonostante allo studente siano stati offerti differenti aiuti, continuava a procrastinare. Purtroppo, Edoardo decise di lasciare il gruppo (e quindi il progetto) in prossimità della deadline da noi originariamente scelta, lasciandoci completamente a mani vuote per quanto riguardasse la sua parte. Abbiamo quindi deciso di continuare il lavoro senza Doglioni, avvisando al docente che avremmo consegnato il lavoro in ritardo. In seguito a questo avvenimento i 'briefing' si sono intensificati aumentandone anche la durata.

Di seguito è riportato l'effettiva suddivisione del lavoro nel dettaglio.

- **Andrea Rossolini**

Si è interessato della realizzazione della parte di lavoro a lui assegnatogli, iniziando il lavoro studiando la logica che gestisce i giocatori, teorizzando anche i metodi che sarebbero stati implementati nel model e nel controller (poiché si era deciso di implementarli tutti insieme). La realizzazione iniziale in realtà verteva anche sullo studio delle statistiche che si sarebbero dovute mostrare a fine partita, poi rimosse poiché, per mancanza di tempo, non vennero mai concretamente implementate. In seguito ad uno studio della libreria di JavaFX ed un utilizzo propedeutico del programma 'SceneBuilder' (mai utilizzato concretamente per la realizzazione del programma) per conoscere meglio classi e metodi di tale libreria, si è dedicato alla realizzazione dei pane e dei dialog per la comunicazione con l'utente. Infine, dopo essersi informato su internet come implementare al meglio la questione della serializzazione su file, si è dedicato al salvataggio ed il caricamento della partita tramite pattern memento. Ha iniziato il lavoro riguardante la parte di programma non di sua competenza subito dopo l'avviso che il compagno avrebbe abbandonato il gruppo, realizzando il prima possibile le parti mancanti, dopo una veloce progettazione, ma con le conoscenze già ottenute nella realizzazione della propria

parte, eccezion fatta per la gestone dei suoni. Quindi nel suo lavoro va aggiunto l'inizializzazione della partita, il menù iniziale ed i suoni. È responsabile anche della stesura delle regole di gioco (accordate con gli altri membri del gruppo) e della guida all'interfaccia grafica.

- **Matteo Alesiani**

Lo studente aveva inizialmente il compito di modellare l'ambiente di gioco attraverso lo studio dei contratti e delle celle disposte lungo la board di gioco, di consentire ad ogni giocatore il normale svolgimento del gioco, concedendogli la possibilità di spostare automaticamente le rispettive pedine e porre in essere conseguenze in base alla locazione di destinazione. Ha inserito i file esterni (principalmente .txt) di ausilio al programma, sviluppandoli attraverso una metodologia che consente l'inserimento futuro di nuove mode di gioco.
Ha reso disponibile ai componenti del team numerose classi all'interno del package utilities, utili e necessarie all'esecuzione dell'applicativo. Ha collaborato alla scrittura e alla messa a punto del Model e del Controller, rendendoli il più possibile atomici allo scopo di rispettare il pattern architetturale MVC utilizzato.
Per far fronte all'uscita di un componente del gruppo, lo studente si è incaricato di gestire l'avvicendamento dei giocatori e delle dinamiche che accadono all'interno del proprio turno, facendo in modo che la logica di gestione rimanga costantemente aggiornata e consistente delle informazioni del gioco.
Per quanto riguarda gli aspetti grafici, il primo passo compiuto è stato quello di informarsi sulle caratteristiche della libreria che si intendeva utilizzare e sfruttando a favore del progetto le potenzialità. In termini più tecnici ha disposto la condivisione di un unico Stage utilizzabile da tutte le schermate del programma e ha suddiviso in parti differenti la finestra principale di gioco rendendole modificabili senza alterare le altre.
L'intero processo è stato sempre preceduto da una meticoloso fare di progettazione, sia di gruppo che individuale, allo scopo di rendere il codice il più fruibile e riutilizzabile possibile.

3.3 Note di sviluppo

- **Andrea Rossolini**

Nella realizzazione della mia parte di lavoro ho fatto largo uso di lambda expressions e stream (cosa che non ho fatto durante l'esame pratico per semplice ignoranza, pentendomene conseguentemente) per la manipolazione di mappe, set e/o liste e altre strutture dati. Ho utilizzato la libreria di Google Guava per l'utilizzo di Optional serializzabili, dato che elementi che venivano scritti su file avevano in sé attributi opzionali, purtroppo non ho approfondito ulteriormente questa libreria che vanta di numerosi pareri positivi in rete. L'esplorazione della libreria di JavaFX ha permesso di espandere la mia visione sull'implementazione di interfacce grafiche oltre che ad avvicinarmi al linguaggio CSS (o quantomeno una sua forma). È stata inoltre realizzata una nuova Exception per segnalare, durante il gioco, quando un giocatore che deve eseguire un "pagamento rischioso" non riesce in alcun modo a coprire la spesa (viene quindi rimosso dal gioco).

- **Matteo Alesiani**

Durante la realizzazione mi sono affacciato all'immenso mondo dell'informatica cercando di carpirne più informazioni e conoscenze possibili. Ho cercato di mettere in pratica tutte le informazioni studiate durante il corso che vanno dalla gestione delle classi, all'utilizzo di interfacce per identificare gli aspetti comuni di un oggetto, arrivando ad utilizzare tutti i modificatori di accesso in base alle loro peculiarità.

Per concepire un progetto con un discreto livello di qualità, ho approfondito gli aspetti tecnici di alcuni pattern, applicandoli (spero correttamente) all'interno del progetto. Per completare l'opera, mi sono dilettato più volte all'utilizzo di stream, lamba expressions e Function, utili ad evitare istante di strutture dati temporanee e consentono una maggiore leggibilità.

Infine, insieme al collega, abbiamo adoperato fogli di stile CSS per rendere la grafica più gradevole ed accattivante per gli utenti del gioco.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

- **Andrea Rossolini**

Essendo per me la prima volta che affronto il linguaggio java o ad ogni modo un progetto così corposo, mi ritengo piuttosto soddisfatto, in quanto, se non avessi dovuto affrontare anche altri impegni (come esami universitari, altri progetti o lavoretti estivi) magari avrei potuto trovare degli elementi nella mia parte di lavoro da poter migliorare o da poter integrare ancora di più; inoltre ho un piccolo rimpianto riguardo ai pattern che non ho potuto usare, dato che, almeno così mi è sembrato, non avrebbero preso troppo piede nelle parti a me dedicate. Magari avrei potuto utilizzare in maniera più approfondita la libreria di Guava. Ad ogni modo anche lo studio della libreria di javaFX mi ha dato modo di fare esperienza quanto meno per farmi capire cosa significa studiare da autodidatta una libreria così ampia come, appunto, quella di JavaFX. Sono inoltre contento di aver potuto lavorare con lo studente Matteo Alesiani, molto competente, paziente e bravo nel districarsi fra i vari problemi incontrati, inoltre (per quanto già lo conoscessi di persona) mi ha fatto capire cosa significa lavorare in un team, riuscirsi ad organizzare, rispettando impegni personali e altrui, integrare il proprio lavoro e le proprie idee con quelle di altri; oltre che è sempre un piacere scambiarsi pareri e consigli in questo ambito. Inoltre, anche se inizialmente con qualche difficoltà, ho imparato ad utilizzare Git, ormai per me indispensabile, infatti l'ho già usato in altri progetti svolti all'interno dell'università. Purtroppo, mi dispiace per lo studente che ha lasciato il gruppo mi sarebbe piaciuto lavorare in un gruppo più numeroso.

- **Matteo Alesiani**

Ritengo che sia la prima volta per me realizzare un progetto che, seppur non eccessivamente ampia, dispone di tutte le potenzialità per poter essere utilizzabile all'interno del mondo video-ludico. Ho riscontrato le classiche difficoltà iniziali, soprattutto nel progettare il sistemare tenendo conto non solo del mio operato, ma anche di quello degli altri componenti. Man mano che il lavoro proseguiva, però, le idee si chiarirono e le classi iniziavano ad operare all'unisono facendo intravedere quello che poi sarebbe stato il prodotto finito. Arrivati alla conclusione del progetto

è opportuno fare delle valutazioni oggettive ed imparziali. Mi sono avvicinato al mondo dell'informatica perchè ritenevo grandiosa la possibilità di poter realizzare qualsiasi cosa si voglia, partendo dal basso ed arrivando a grandi applicazioni di utilizzo quotidiano. Quella realizzata non ha l'intendo di stravolgere il mondo ludico, perché come in ogni progetto, si potrebbero trovare tantissimi punti di miglioramento e modifica, ma va osservato per quel che è. Esso mi ha permesso di accrescere molto le competenze e conoscenze di un linguaggio (Java) utilizzato in numerosi ambiti, mi comprendere il flusso di lavoro di un progetto ed interfacciarmi professionalmente con altre persone, aventi un obiettivo prefissato e comune.

Un aspetto, sicuramente da migliorare, è quella della documentazione del codice. In più di una occasione i compagni, si sono trovati in difficoltà nell'utilizzare i metodi che ho realizzato, poiché a volte il semplice prototipo del nome non basta per comprendere come lavori il metodo.

Inoltre, una consideravazione va fatta sull'utilizzo della libreria Javafx: il suo utilizzo ha sicuramente impreziosito il progetto, ma il suo studio ha comportato grande perdita di tempo, utilizzabile per altri scopi.

Il software Git, conosciuto per la prima volta in questa circostanza (e ovviamente durante il corso), ha consentito una rapida condivisione del codice tra i componenti, a volta situati a km di distanza, e una flessibile modalità di lavoro. Il codice risultata costantemente aggiornato e i componenti potevano suggerirsi piccole modifiche per incorporare i metodi implementati.

Un ultima considerazione, nonché un enorme ringraziamento, va ad Andrea, il quale con grande caparbietà e saggezza ha saputo mantenere la calma, anche quando la mia insana pazzia portava a cambiare parti particolarmente consistenti di codice. La sua predominante voglia di imparare e curiosità, unite ad un'oggettiva vivacità intellettuale, alimentavano continuamente una comune voglia di perfezionare e migliorare il progetto.

Il grande dono che sicuramente ci porteremo per sempre con noi, è quello di non mollare di fronte alle difficoltà e di trovare (da futuri ingegneri informatici) sempre e comunque una soluzione al problema.

Guida utente

Per chiunque non dovesse conoscere le regole del monopoli o volesse comunque confrontare le lievi differenze con il gioco originale, è possibile accedere al manuale di gioco direttamente da applicativo. Il menu iniziale mette infatti a disposizione un pulsante (riconoscibile con un “?”) il quale aprirà un pdf contenente, appunto, le regole di gioco ed una rapida guida all’interfaccia grafica.