MONOPOLY

Relazione di progetto
Programmazione e Modellazione ad Oggetti

Borioni Luca
Panaroni Lorenzo
Maurenzi Marco

4 Luglio 2024

INDICE

Capitolo 1
Analisi

Il Team si pone come obiettivo lo sviluppo di un applicativo per giocare a Monopoly.

Il Monopoly è un gioco da tavolo di strategia e compravendita immobiliare che si svolge su una tavola quadrata detta tabellone formata da 24 caselle (nella versione ufficiale sono 40).

1.1 Requisiti

1. Al gioco devono partecipare da 2 a 4 giocatori.
2. Ogni giocatore deve iniziare con una quantità predefinita di denaro (1500 $).
3. I giocatori devono muoversi sul tabellone in base al risultato dei dadi.
4. I giocatori devono avere la possibilità di acquistare proprietà non possedute.
5. I giocatori devono pagare un affitto quando si fermano su una proprietà posseduta da un altro giocatore.
6. I giocatori possono costruire case sulle loro proprietà solamente quando posseggono la serie completa.
7. I giocatori possono mettere all'asta le loro proprietà che possono essere acquistate sia da altri giocatori sia dalla banca.
8. Nel caso in cui i giocatori cadano su una specifica casella, essi verranno mandati in prigione nella quale dovranno passare tre turni senza possibilità di muoversi.
9. Il gioco termina quando tutti i giocatori, tranne uno, sono in bancarotta (saldo negativo).

1.2 Analisi e modello del dominio

Il Monopoly dovrà essere in grado di gestire sessioni di gioco tra i giocatori aggiunte.

Una partita conterrà le principali entità:

- Il gioco (Game).
- I giocatori (Player).
- Il tabellone (Board) composto di caselle (Box) di varie tipologie e da due mazzi di carte (chanceCards/unexpectedCards).
- La banca (Bank).
- I dadi (Dice).

Il **gioco** sarà il regista di tutto ciò che succede all' interno di una partita in quanto esso conterrà tutte le informazioni sullo stato della stessa; più nel dettaglio, conoscerà tutti i **giocatori** che partecipano alla partita, il **tabellone**, i **dadi,** e la **banca**.

Il **giocatore** si muoverà sul tabellone a seconda del risultato di un lancio di dadi, che potrà tirare nuovamente a seconda dell'esito del primo lancio ed infine essere mandato in prigione a fronte di tre risultati doppi consecutivi.
Avrà una liquidità di partenza (1500$) che potrà utilizzare per comprare proprietà, pagare gli affitti agli altri giocatori o eventuali imprevisti dati dalle carte.
Una volta acquistate le proprietà un giocatore potrà mettere queste ultime all'asta nel caso in cui abbia bisogno di soldi.

Il **tabellone** contiene le informazioni riguardo le varie caselle che lo compongono e i due mazzi di carte.

Ogni casella può rappresentare:

- una proprietà acquistabile dai giocatori
- un evento (casella probabilità / casella imprevisti)
- una penalità (casella vai in prigione)
- un passaggio (casella di transito)

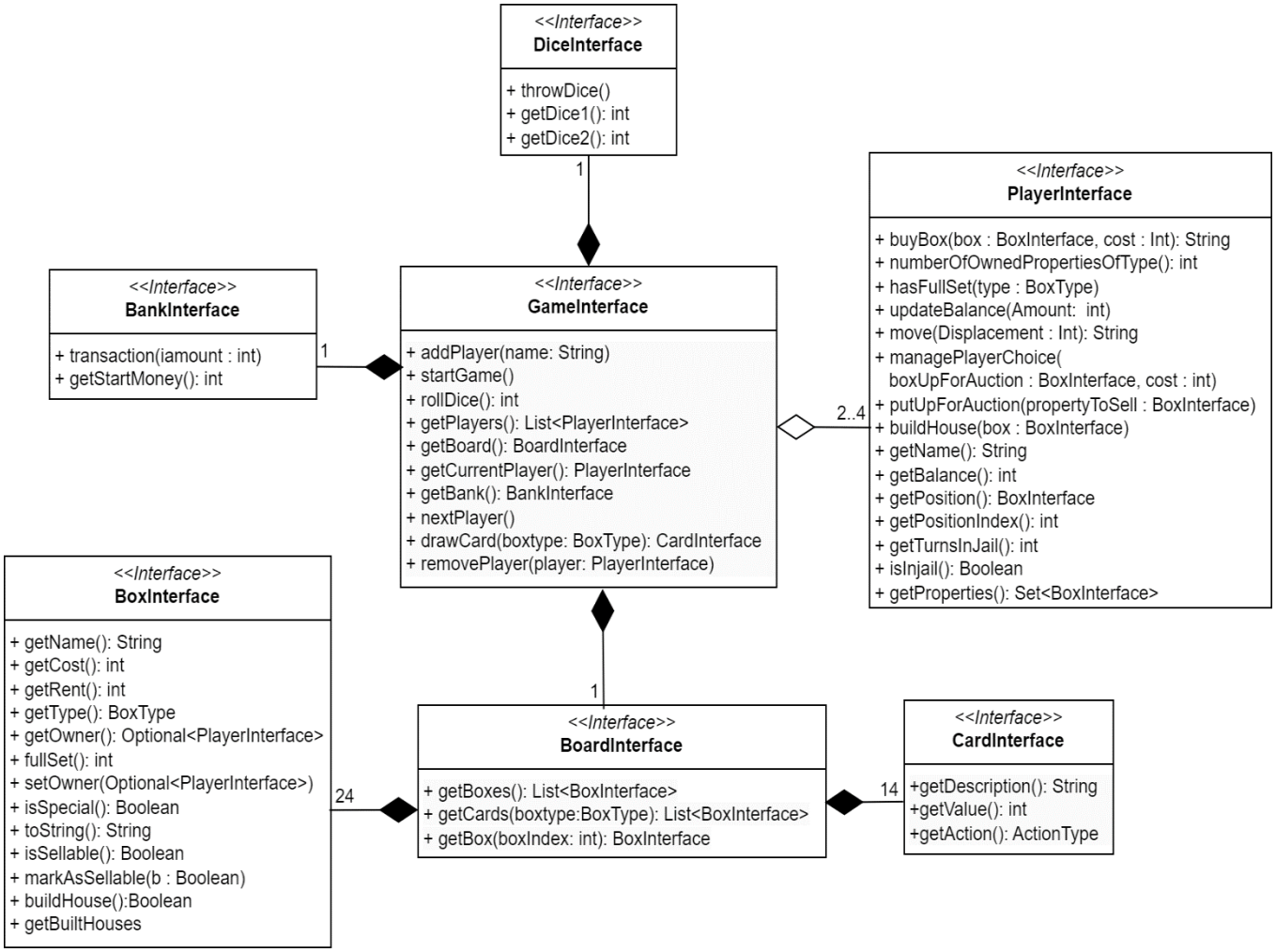
Le proprietà possono essere acquistate dal primo giocatore che capita su una di queste e in caso di acquisto, gli avversari che visiteranno la casella successivamente dovranno pagare l'affitto al proprietario di quella casella, in caso contrario, il giocatore successivo che capiterà sulla casella avrà opportunità di acquistarla.

La casella evento impone al giocatore di pescare una carta o dal mazzo degli imprevisti o da quello delle probabilità per poi fare ciò che la carta pescata indica.
La casella penalità (vai in prigione) impone al giocatore di trasferirsi in prigione fino a pena scontata (3 turni).

La casella di transito non ha nessun effetto nel momento in cui ci si cade sopra.
Tuttavia, anche nel gioco ufficiale corrisponde con la posizione della prigione.

La **banca** si occuperà di comprare le proprietà all'asta che gli altri giocatori non compreranno, di distribuire 200$ ad ogni giocatore che supera il "VIA" e riceverà i soldi di una proprietà acquistata per la prima volta.

Schema UML dell'analisi del problema, con rappresentate le varie entità e come si interfacciano tra di esse.



Capitolo 2
Design

Per lo sviluppo della struttura del monopoly, è stato deciso di seguire il pattern architetturale MVC (Model - View - Controller) in quanto sposa al meglio l'idea di interazione con l'utente per aggiornare il sistema.

2.1 Architettura

2.1.1 Model

Il Model è il funzionamento base del software, gestisce tutta la parte di funzionamento e aggiornamento del sistema.
Successivamente questo componente comunicherà con il controller per poter aggiornare la View oppure essere aggiornato lui stesso.
Importante specificare che il Model funzionerebbe anche se View e Controller non ci fossero.

2.1.2 View

La View è la parte grafica del sistema, servirà quindi ad interagire con i vari giocatori che potranno, tramite bottoni e menu a tendina, modificare di fatto lo stato del sistema e fare scelte che influenzeranno la partita.
Per l'implementazione della View abbiamo utilizzato Java-Swing e per rendere tutto il più snello possibile abbiamo semplificato la rappresentazione del tabellone facendone una sua versione "stilizzata" ma comunque funzionale.

2.1.3 Controller

Il Controller è quel componente che registra ciò che è accade nella View e lo comunica al Model aggiornando così lo stato del sistema come prima accennato, facendo da tramite tra l'utente e lo stato interno del programma.
Una volta registrata l'interazione dell'utente e aggiornato il Model di conseguenza, viene aggiornata anche la View che mostrerà i cambiamenti causati dall'interazione con l'utente.

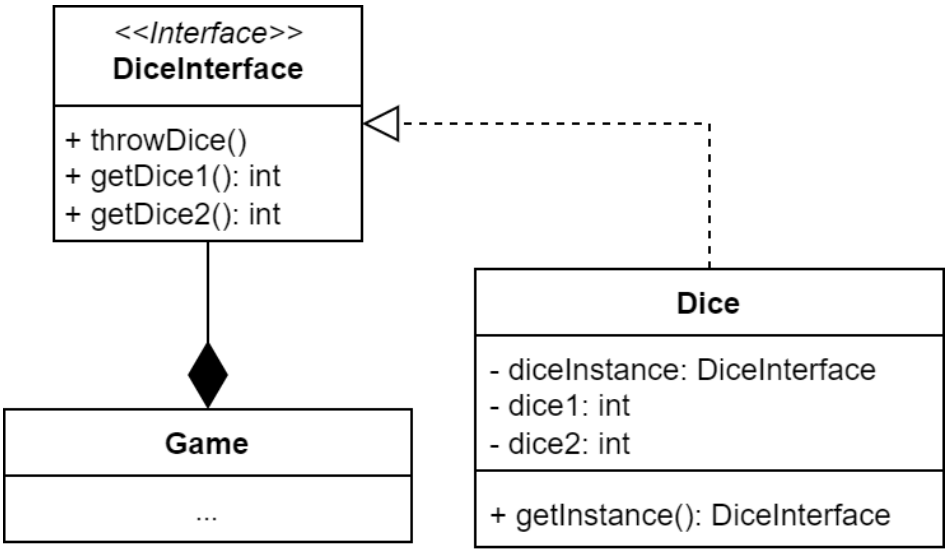
2.2 Design dettagliato

2.2.1 Dice

I dadi sono una entità più concettuale che di effettiva utilità in quanto il loro compito è semplicemente generare due numeri pseudocasuali entrambi da uno a sei.

Questi sono stati realizzati seguendo il pattern architetturale singleton, il quale prevede che possa esistere una sola istanza di questa classe tramite costruttore privato ed un campo privato statico.
Quest'ultimo può essere inizializzato solo una volta chiamando il metodo statico della classe "getInstance()".

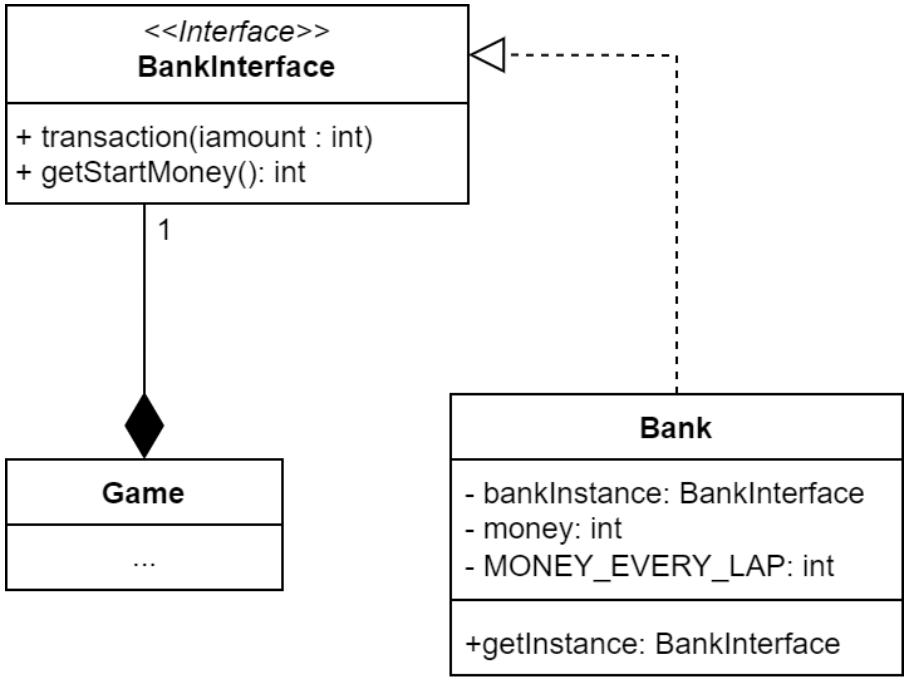
La classe si sposa perfettamente con questo tipo di pattern poiché anche in una partita reale esiste soltanto un paio di dadi, ed il singleton realizza esattamente questa idea.



2.2.2 Bank

La banca riprende quello che è il design dei dadi appena presentato; infatti, anch'essa segue il pattern architetturale del singleton.

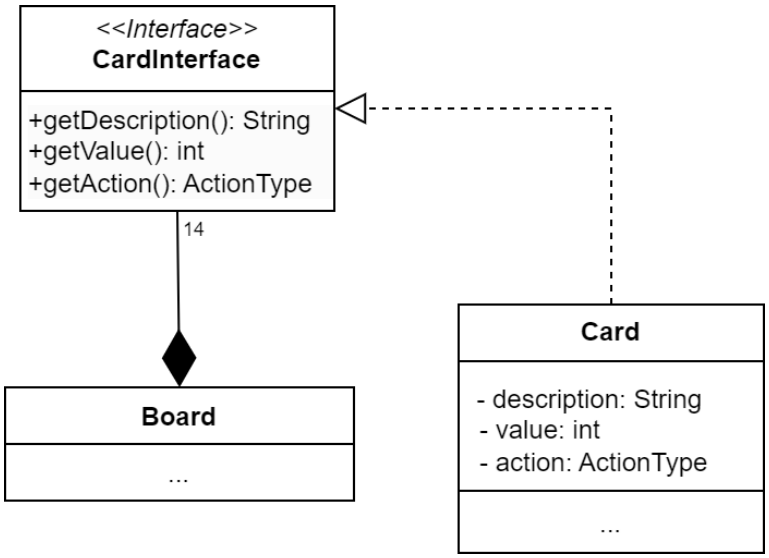
Il suo scopo è quello di fornire i soldi necessari ai vari giocatori durante lo svolgimento del gioco come, ad esempio, il passaggio dal via.
Inoltre può anche ritirarli in situazioni come imprevisti dati dalle carte, che consistono nel pagamento di un giocatore alla banca oppure al primo acquisto di una proprietà.
La banca ha disponibilità iniziale di un miliardo di $.



2.2.3 Card

La carta è stata modellata pensando al pattern architetturale strategy, in quanto esistono tre tipi di azioni possibili che la carta obbliga a compiere, ovvero, un cambio di saldo, un cambio di posizione oppure un cambio di stato (in prigione).
In questa classe, infatti, è stato riservato un campo che con l'utilizzo di un tipo di dato enumerato da noi definito, esplicita quale azione quella determinata carta impone al giocatore che l'ha pescata.
Questo campo verrà poi utilizzato all'interno della classe player recuperandolo con l'utilizzo del getter corrispondente, poi a seconda del tipo di azione da compiere verrà eseguito un algoritmo che andrà a modificare lo stato di player.

Ultima cosa degna di nota nella classe card è l'utilizzo di una tecnica chiamata "overloading dei costruttori" che permette di invocare un costruttore piuttosto che un altro a seconda degli argomenti passati al momento dell'invocazione.
Il primo dei due costruttori permette di creare carte che variano saldo oppure posizione di una quantità che viene specificata come argomento del costruttore.
Mentre al secondo sarà sufficiente passare la descrizione e verrà automaticamente invocato il primo costruttore con tipo di azione "prigione" e un valore predefinito per quanto riguarda la quantità prima citata. (sarà poi il valore che costringerà il giocatore ad andare in prigione)

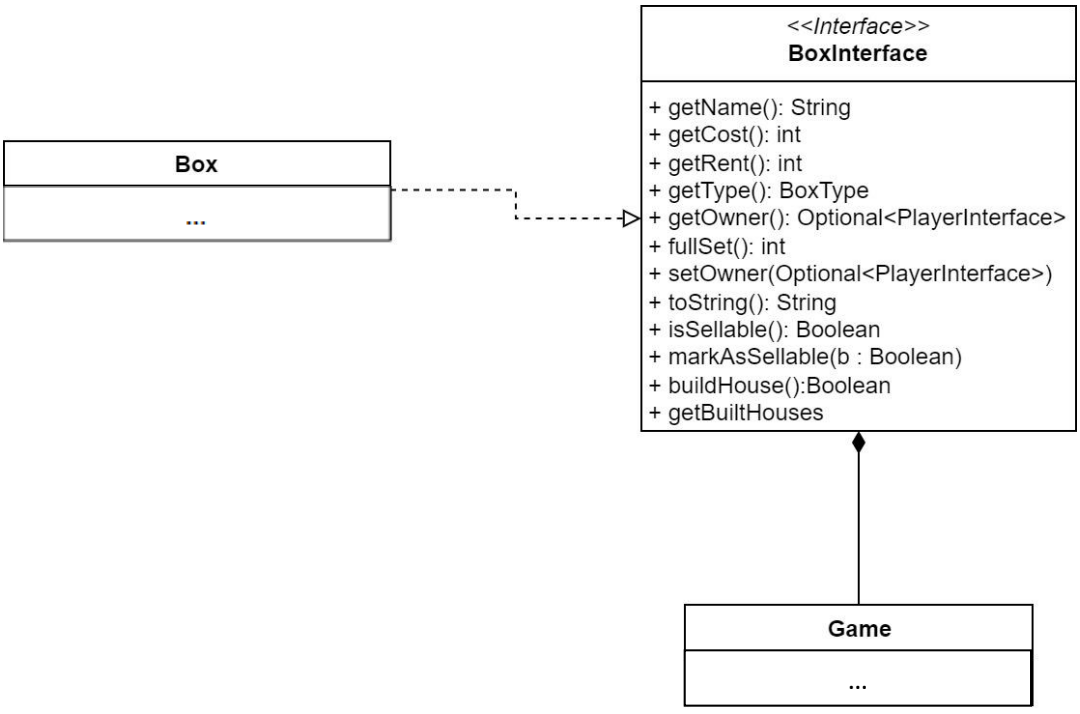


2.2.4 Box

Box è la classe in cui le caselle che formano il tabellone di gioco vengono modellate.
Oltre a campi come nome e costo della casella, abbiamo deciso di utilizzare un “Optional” per inizializzare il proprietario in quanto esso può esserci o meno.
Questo verrà poi modificato nel momento in cui un giocatore comprerà la proprietà, la quale tornerà allo stato originale qualora quello stesso giocatore decida di venderla o venga eliminato dal gioco.

Una casella che per ovvie ragioni non è acquistabile, come quella per pescare carte probabilità e imprevisto, possiede un campo che specifica questa caratteristica; inoltre queste rimarranno sempre senza proprietario.
Come per la classe card anche qui si è fatto uso dell’overloading dei costruttori per discriminare la creazione di caselle vendibili e non.
Infine, per differenziarle in modo “pulito” è stata usata un’enumerazione, che permette di distinguere le varie caselle per tipo e per numero.

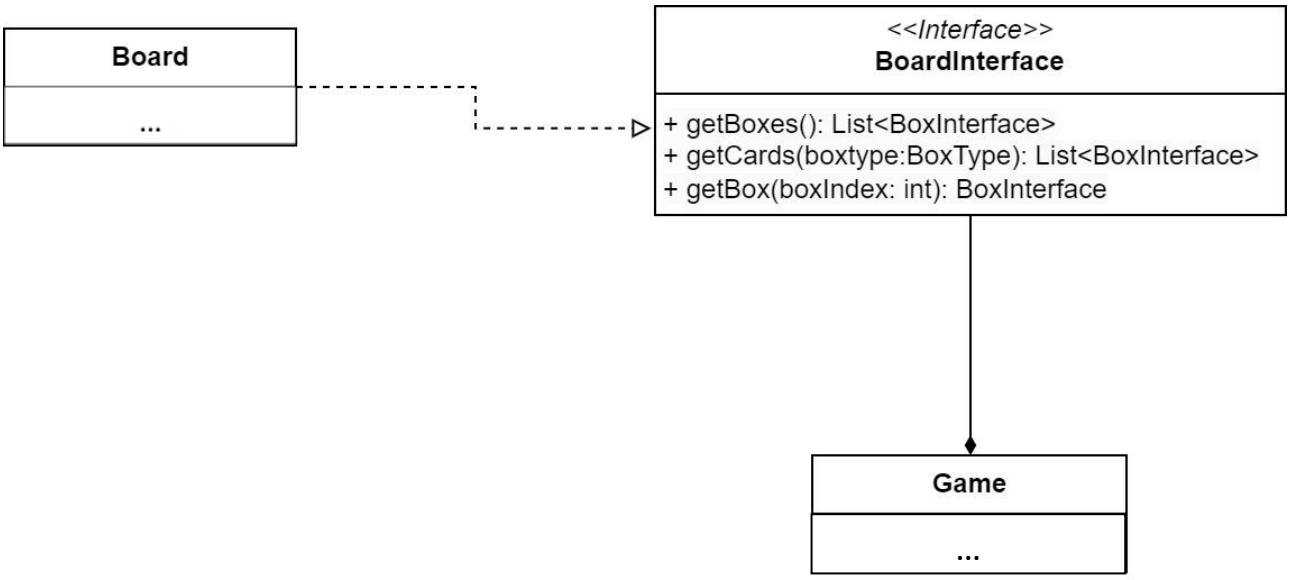
Volendo garantire tra le funzionalità anche la possibilità di costruire case sulle celle acquistate, è necessario sapere quante ne sono state costruite su ogni casella, perciò un campo della classe è stato dedicato a questo scopo.



2.2.5 Board

La classe board contiene l'inizializzazione delle caselle che compongono il tabellone di gioco, ma anche delle carte probabilità e imprevisti (le seconde sono più negative delle prime), e questi tre elementi sono contenuti nelle strutture dati che rappresentano i campi della classe.

Gli unici metodi della classe sono metodi getter, uno dei quali in grado di adattarsi qualora volessimo la lista di carte imprevisto piuttosto che quella di carte probabilità.
Questo metodo segue il pattern strategy anche se l'algoritmo è praticamente lo stesso ma cambia il campo restituito.



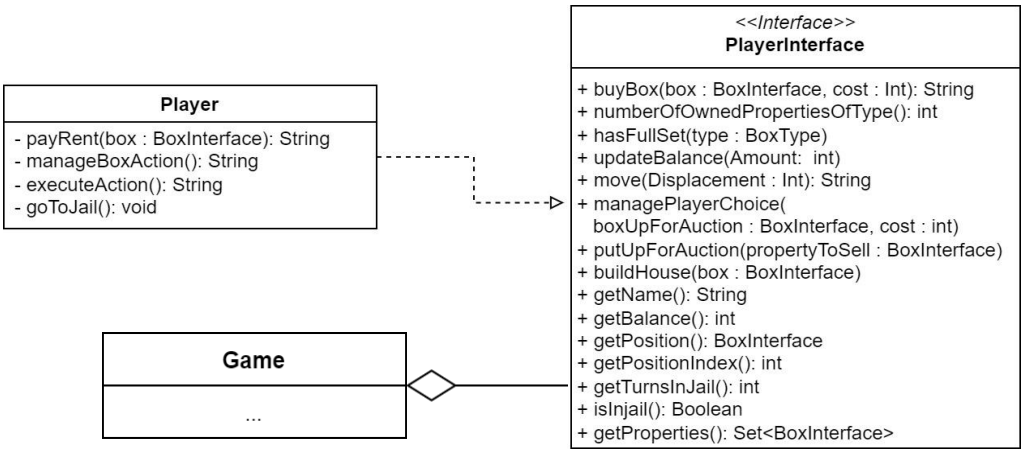
2.2.6 Player

Per questa classe l'idea è stata quella di modellare un generico giocatore in grado di compiere tutte le azioni a lui concesse in una partita di monopoly.
Qui si tiene traccia del saldo del giocatore (modificabile attraverso un metodo che controlla la sua liquidità e nel caso in cui vada in negativo lo rimuove dal gioco), della sua posizione sul tabellone e della lista delle sue proprietà.
È anche necessario tenere presente se il giocatore si trovi attualmente in prigione e per quanti turni ci debba rimanere, in quanto questo avrà effetto sulla possibilità del giocatore di muoversi.

Oltre a metodi per poter recuperare i campi caratterizzanti lo stato della classe anche al suo esterno (per esempio la lista di proprietà del giocatore), il player può muoversi attraverso le varie caselle, è in grado di acquistare proprietà oppure di mettere all'asta quelle in suo possesso e costruire case sui suoi possedimenti (rispettate le giuste condizioni).
La classe implementa anche un metodo in grado di simulare una scelta del giocatore per quanto riguarda l'acquisto delle proprietà messe all'asta da altri, la condizione di acquisto è che manchi una proprietà per avere la serie completa (tutte le proprietà dello stesso colore) e che si abbia tre volte il costo della proprietà come liquidità.

Come prima accennato nella classe card è presente un metodo che una volta identificato il tipo di azione di una carta pescata, ne affida l'esecuzione ad altri metodi che renderanno effettiva l'azione di quella carta.

Ultima cosa da far notare in questa classe è la scelta per i valori dello spostamento del giocatore, che nel caso in cui non sia più autorizzato (da game) a tirare il dado verrà eseguito il metodo per muoversi con valore zero, mentre nel caso in cui debba essere mandato in prigione verrà eseguito con parametro passato uguale a meno uno.
Inoltre, questo metodo si occuperà di ridurre i turni in prigione del giocatore.



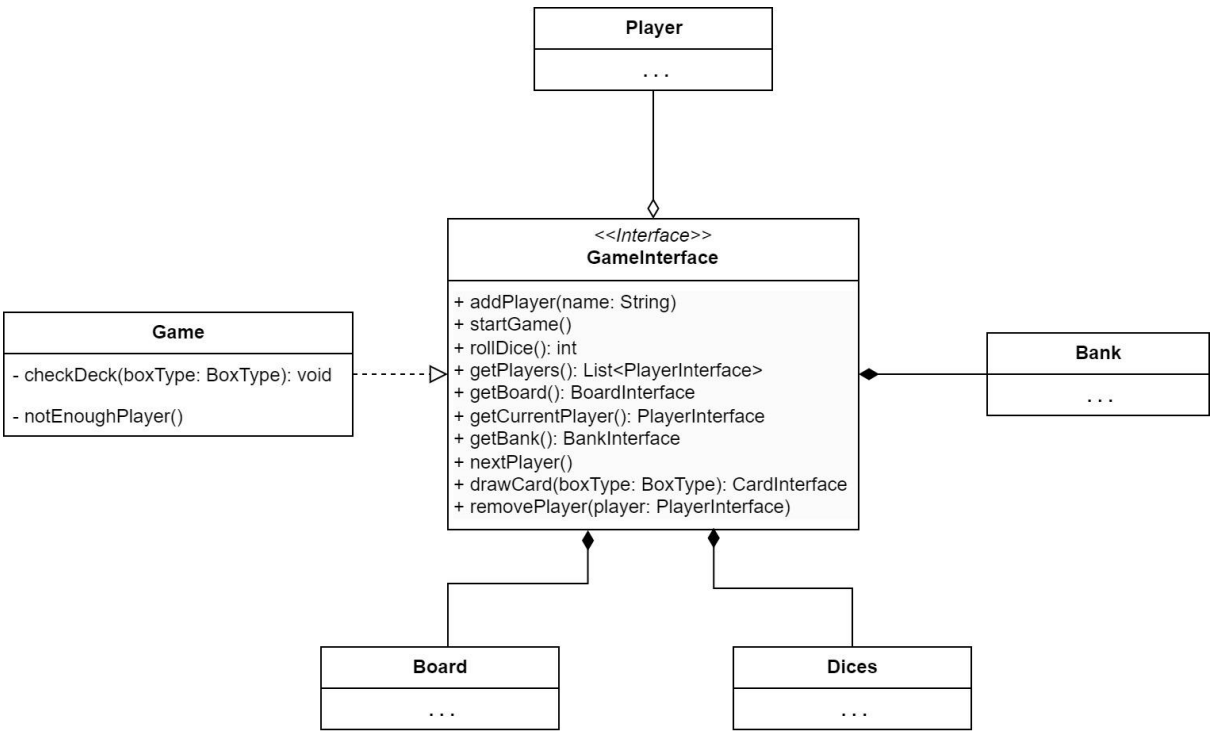
2.2.7 Game

Il game è il cuore del sistema che raccoglie tutte le classi precedentemente elencate tramite composizioni pure, aggregazioni e gestisce tutta la parte di regole del gioco. Inoltre, tutti i tipi di dati con cui game si aggrega/compone sono tutti dichiarati di tipo interfaccia permettendo così, successivamente, di creare varianti del gioco costruendo classi diverse che implementano quelle interfacce, come ad esempio un player che è più oculato nell'acquisto di una proprietà all'asta piuttosto che dei dadi che vanno da uno a dieci. (Polimorfismo con interfacce -----> subtyping)

Questa classe permette di aggiungere/rimuovere giocatori, gestire i dadi, i turni e i due mazzi di carte.
Si occupa inoltre di far cominciare o terminare la partita ed assicurarsi che i requisiti siano rispettati (almeno due giocatori per cominciarla e uno rimanente per terminarla), il tutto coordinato da un metodo che ha può lanciare un'eccezione.

La regola che concerne i dadi dice che, se con entrambi si ottiene lo stesso numero, il giocatore li può rilanciare, questo può avvenire per un massimo di tre volte.
Nel caso in cui uscisse "doppio" tre volte di fila il giocatore verrà mandato in prigione.
Se non uscisse doppio la possibilità del giocatore di rilanciare i dadi verrebbe bloccata.

I due mazzi vengono mescolati alla creazione degli stessi e le carte vengono pescate una per volta (tramite il tabellone) fino all'esaurimento di uno dei due, il qual viene rimesso in gioco dopo essere stato mescolato nuovamente.



Capitolo 3
Sviluppo

3.1 Testing automatizzato

In questo progetto è stata utilizzata la libreria “Junit 5” per testare ogni unità del sistema man mano che questa veniva sviluppata oppure modificata.
Una cosa di cui prendere consapevolezza è la presenza di “Flaky Tests”, dovuti all’aleatorietà (dadi/mazzi mescolati) del sistema e come tali, non daranno sempre lo stesso esito a fronte di esecuzioni diverse.

3.2 Metodologia di lavoro

La strategia di integrazione è stata quella di partire dalle singole unità del sistema già testate per poi unirle tra loro usando il modello “Bottom-up”, quindi testare classi contenenti queste unità e infine unire tutto il sistema.
Un esempio di utilizzo di questa tecnica è nella classe “GameTest” dove al suo interno vengono usate altre classi già testate come player, box...

La divisione del lavoro in questo progetto non è stata netta a sufficienza da creare sottosezioni di lavoro per ogni membro, tuttavia è stato un lavoro di team-building grazie al confronto continuo tramite Google Meet, Git e Github.

3.3 Note di sviluppo

- Stream per creare una lista con i nomi delle caselle (Controller)
- Lambda expressions per la creazione di ActionListener (Controller)
- Stream per la ricerca di una proprietà a partire dal nome (Controller)
- Optional per contenere la proprietà cercata nel caso in cui fosse presente (Controller)
- Optional per contenere il proprietario di una casella se presente (Box)
- Stream per contare il numero di proprietà dello stesso colore (Player)
- Enumerazione utilizzata per distinguere l’azione imposta da una carta (ActionType)
- Enumerazione per discriminare il tipo di casella (BoxType)

GITHUB: https://github.com/luke99310/PMOProject.git