# Monopoly

Davide Rossi, Alessio Minniti, Lucas Prati, Martina Ravaioli

19 giugno 2025

# Indice

# Capitolo 1

# Analisi

## 1.1 Descrizione e requisiti

Mettere ref a link Monopoly

Il software mira alla costruzione di una versione virtuale del famoso gioco da tavolo "Monopoly". Quest'ultimo è un gioco di società nel quale più giocatori a turno lanciano i dadi e muovono le proprie pedine su un tabellone composto da caselle di cui la maggior parte sono proprietà, ovvero dei terreni acquistabili dai giocatori. Lo scopo del gioco è di creare un monopolio aumentando i beni o il denaro posseduto. Per fare ciò durante la partita i giocatori possono acquistare le proprietà sulle quali capita la propria pedina pagando una somma alla banca, una volta acquistate il giocatore può anche decidere se eventualmente migliorarle. Se la pedina di un giocatore finisce su una proprietà in possesso di un suo avversario allora sarà costretto a pagargli una somma di denaro per poter transitare su un terreno non suo, in questo caso si dice che il giocatore paga l'affitto per la proprietà.
Sul tabellone saranno presenti anche delle caselle speciali: le più importanti sono il parcheggio e la prigione che impediscono al giocatore di muoversi per un determinato numero di turni e le caselle imprevisti e probabilità che richiedono al giocatore di pescare una carta o dal mazzo degli imprevisti o da quello delle probabilità e seguire le istruzioni dell'effetto scritto sulla carta.
L'obiettivo è mandare gli altri giocatori in bancarotta, e dunque condurli alla sconfitta, cercando di fargli spendere più denaro possibile. Le pedine dei giocatori sconfitti vengono rimosse dal tabellone e i loro beni revocati, si vince rimanendo l'ultimo giocatore sul tabellone.
Il software permetterà di creare e giocare un partita con un determinato numero di giocatori dallo stesso terminale.

### 1.1.1 Requisiti funzionali

- Il gioco potrà essere giocato su uno stesso dispositivo da un minimo di 2 giocatori ed un massimo di 6. L'utente potrà impostare altre opzioni di configurazione relative al gioco prima di avviare la partita come il nome dei giocatori e la modalità di gioco.
- Il giocatore, durante il proprio turno, potrà tirare i dadi per far muovere la propria pedina sul tabellone di un numero di caselle corrispondente alla somma dei dadi lanciati.
- Il giocatore, dopo aver tirato i dadi, avrà la possibilità di acquistare la proprietà sulla quale è capitata la sua pedina, se questa proprietà non appartiene a nessuno. Se la proprietà è già posseduta da qualcun altro il giocatore che vi capita sopra dovrà pagare l'affitto al proprietario per poter continuare a giocare.
- Se il giocatore finisce sopra la casella speciale parcheggio la sua pedina dovrà rimanere ferma per un turno.
- Se il giocatore viene messo in prigione, perchè è capitato sulla casella "Vai in prigione", dovrà rimanere fermo sulla casella "Prigione" per 3 turni. Ogni turno gli verrà concessa la possibilità di liberarsi lanciando i dadi e provando a fare un doppio come risultato.
- Se il giocatore finisce sopra una casella imprevisti o probabilità allora verrà pescata una carta effetto o dal mazzo imprevisti o dal mazzo probabilità. La carta appunto descrive un effetto che va ad agire e modificare gli elementi del gioco (tabellone, pedine, proprietà, denaro posseduto...). L'effetto eseguito è specifico della carta, al giocatore verrà mostrata una descrizione testuale dell'effetto e poi il sistema ne farà partire l'esecuzione.
- La pedina del giocatore potrà finire su altre caselle speciali con un effetto specifico che si attiverà o al passaggio (casella Start) o se ci finisco esattamente sopra (casella tasse e altre)
- Nel corso della partita un giocatore, quando capita sulle sue proprietà, le potrà migliorare spendendo del denaro per posizionare delle case che elevano il costo dell'affitto o un albergo, che è come una casa ma ha un costo di affitto e di acquisto maggiore.
- Durante il proprio turno, se lo desidera, il giocatore potrà vendere alla banca proprietà ed eventuali case precedentemente costruite per riguardagnare denaro qualora fosse a rischio bancarotta.

### 1.1.2 Requisiti non funzionali

- Monopoly dovrà permettere agli utenti di aggiungere e modificare gli effetti speciali delle carte imprevisti e probabilità con facilità e senza toccare il codice, bensì dandogli la possibilità di modificare il file di configurazione.
- Monopoly potrà essere giocato attraverso un'interfaccia grafica che dovrà essere in grado di adattarsi alle dimensioni di vari schermi.
- Monopoly dovrà permettere agli utenti di alterare in facilità la configurazione del gioco (parametri aggiuntivi di configurazione della partita, struttura del tabellone e caselle che lo compongono).
- l'applicazione sarà in grado di girare sui principali SO desktop

## 1.2 Modello del dominio

Nella versione da tavolo del gioco, ai giocatori viene periodicamente concesso il proprio turno d'azione sulla base di una rotazione ciclica ed è durante il proprio turno che ai giocatori viene concesso di "giocare" e dunque controllare i propri beni e la propria pedina. Sempre nella versione da tavolo, solitamente sono i giocatori a coordinarsi tra di loro per gestire la rotazione dei turni e arbitrare la partita verificando che ognuno rispetti le regole principali del gioco (controllare se un compagno ha perso ed è eliminato, verificare che un giocatore possa terminare il suo turno, tenere conto di casistiche particolari in cui il giocatore deve saltare dei turni o non può muovere la propria pedina perchè magari è sulla casella prigione o parcheggio e mantenere il conteggio dei turni da trascorrere su queste caselle...).
Per modellare questo aspetto è stata inserita un'entità esterna che chiameremo arbitro `referee` a cui sono delegate tutte le operazioni di coordinazione del gioco e dei suoi partecipanti. Queste operazioni sono:

- Coordinare i turni dei giocatori concedendo la possibilità di giocare, il giocatore potrà terminare il proprio turno solo una volta che tutte le azioni obbligatorie sono state eseguite (ad esempio pagare un affitto o tirare i dadi).
- Decretare se un giocatore è eliminato oppure no sulla base della sua situazione finanziaria, se il saldo del portafoglio del giocatore è in negativo perde la partita e tutti i suoi contratti di proprietà ritornano alla banca, disponibili per l'acquisto.

- Decretare se il gioco è finito e determinare il vincitore.
- Tenere il conto dei turni che un giocatore deve trascorrere bloccato nel parcheggio o nella prigione e valutare se eventualmente il giocatore potrebbe liberarsi prima

Come è stato detto, a ogni giocatore (Player) viene concesso periodicamente dall'arbitro il proprio turno d'azione sulla base di una rotazione ciclica. Durante il suo turno il giocatore muove sul tabellone (Board) la propria pedina (Pawn) tirando dei dadi (Dices). Il tabellone è composto da una serie di caselle (Tile) disposte su un percorso chiuso. Il giocatore muove la propria pedina spostandola di un numero di caselle corrispondenti alla somma dei dadi tirati, atterrando così su una nuova casella. Questa casella può essere una proprietà (Property) oppure una casella speciale (Special), e sulla base di ciò cambia radicalmente l'interazione con l'utente.
Le proprietà sono caselle alle quali è associato un contratto di proprietà (Title deed). Quando un giocatore capita su una casella di tipo proprietà non ancora in possesso di nessun altro, può richiedere alla banca (Bank) di acquistare il contratto (Title deed) associato alla suddetta proprietà. Dopo aver pagato una somma alla banca il contratto viene consegnato al giocatore che da quel momento in poi possiederà la proprietà. Il contratto di proprietà è un documento che descrive tutte le informazioni monetarie relative alla proprietà come il prezzo d'acquisto, prezzo di vendita e le varie opzioni di affitto, ovvero possibili prezzi di affitto che il proprietario può far pagare per passare sulla proprietà associata. La banca possiede tutti i contratti non acquistati. Se la proprietà su cui è capitato il giocatore era già stata comprata in precedenza da un altro allora si deve pagare l'affitto al proprietario. I giocatori consultano il contratto di proprietà e scelgono una delle varie opzioni di affitto disponibili sul contratto come prezzo concordato per il pagamento. Il giocatore di passaggio procede poi a pagare il giocatore proprietario. Se in un turno successivo all'acquisto il giocatore capita novamente su una casella di sua proprietà può decidere se migliorarla aggiungendo case o alberghi acquistandoli dalla banca. Infine ogni giocatore può rivendere alla banca un contratto di proprietà e ricevere del denaro in cambio. Ogni giocatore ha associato un portafoglio (wallet) contente del denaro. Attraverso il suo portafoglio il giocatore compie tutte le operazioni di compravendita che comportano un addebito o prelievo di denaro (comprare/vendere contratti, pagare affitti...)
Se il giocatore invece capita su una casella speciale (special) si attiverà un effetto caratteristico della casella (Effect) che avrà ripercussioni sullo svolgimento del gioco (guadagno/perdita denaro, saltare un determinato numero

di turni...). In particolare se si capita su una casella Imprevisti (Unexpected) o Probabilità (Probability) l'effetto non è specifico della casella stessa, ma viene letto da uno dei due mazzi di carte effetto corrispettivi(Unexpected deck o Probability deck). Il giocatore pesca una carta (Card) da uno dei due mazzi, legge l'effetto della carta e lo attiva.
Parte della difficoltà consisterà nel riadattare un dominio che non nasce intrinsecamente per un progetto software. Monopoly infatti nasce come gioco da tavolo e questo si riflette in molte interazioni e funzionalità, che non sono pensate nell'ottica di un'architettura software (un esempio è quello dato all'inizio della sezione sulla coordinazione fra i giocatori e l'entità arbitro). Molte azioni nel gioco spesso comportano la comunicazione di più entità (l'arbitro decreta il vincitore sia sulla base dei contratti che possiede sia sul suo portafoglio, migliorare una proprietà ha come conseguenza un cambiamento sulla casella ma per attuarlo il giocatore deve interagire con la banca facendo riferimento al contratto di proprietà, un giocatore può passare il turno solo se ha fatto determinate azioni che dipendono non solo dalla casella in cui si trova ma anche dal suo contratto...). Sarà quindi difficile progettare un'architettura software che permetta di rappresentare il dominio del gioco in maniera efficace e riadattabile.

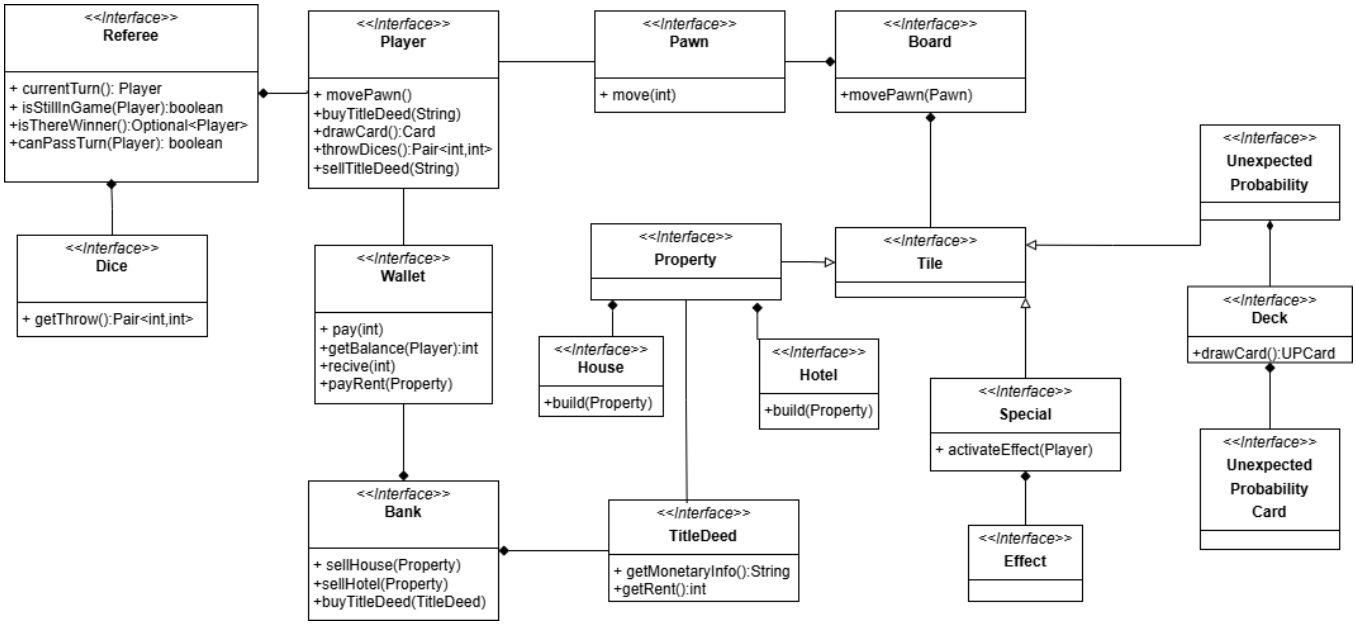


Figura 1.1: Schema UML dell'analisi del problema, con rappresentazione delle entità principali ed i rapporti fra loro

# Capitolo 2

# Design

## 2.1 Architettura

L'architettura di MPoly è basata sul pattern architetturale MVC.
Si è scelto di implementare MVC nella sua forma classica in quanto il gioco ha una modalità di interazione ad eventi: ovvero ogni modifica che avviene sul model è a seguito di un evento generato dall'utente.
L'utente interagisce con l' applicazione facendo scaturire un evento (pressione di un bottone nella view, tasto della tastiera...). Il controller cattura questo evento ed esegue una serie di operazioni specifiche per la sua gestione interagendo con la parte model dell'app, e susseguentemente aggiorna la view mostrando all'utente che cosa è successo.
In riferimento allo schema dell'architettura, è stato predisposto un controller principale denominato GameManager che ha un riferimento a un oggetto che implementa l'interfaccia MainView, sul quale chiamerà l'aggiornamento della view.
In questa versione dell'applicazione la modalità di interazione utente avviene tramite pressione di bottoni della view. Per questo si è deciso di implementare il pattern "Observer": MainViewImpl ha un riferimento al GameController ed è questa classe che notifica il Controller dell' inizio di un evento. Al momento della notifica il controller interroga il model.
A seguito dell'analisi sono stati individuati 3 punti d'entrata per la componente model dell'applicazione, ognuno dei quali si occupa di un macro aspetto del dominio. Nello specifico abbiamo:

- Board, che si occupa di gestire la struttura del tabellone, le caselle con case e alberghi e il movimento delle pedine.
- TurnationManager (che sarebbe il corrispettivo di referee nell'analisi)

che si occupa di gestire l'avvicendarsi dei turni dei giocatori e la fine del gioco.

- Bank, per orchestrare lo scambio di denaro fra i bank account dei giocatori e la compravendita di contratti di proprietà.

Questi punti d' entrata del model offrono delle primitive che incapsulano la logica di funzionamento delle principali azioni che si possono compiere durante il gioco, azioni che sono caratteristiche del gioco stesso Monopoly. Il controller orchestra varie chiamate ai metodi di queste 3 classi del model per far funzionare il gioco. Con questa architettura il model è perfettamente scorporabile e riutilizzabile per costruire un software diverso che risponda allo stesso dominio. L'assenza di un unico punto di entrata per il model previene l'esistenza di una macro classe con eccesiva responsabilità sulla quale dipenderebbe tutto il funzionamento del gioco, facilitando lo sviluppo e l'evoluzione del software.
Il controller coopera con model e view mediante delle interfacce che sono completamente indipendenti dall'implementazione di quest'ultimi. Questo, in particolare, fa sì che le scelte implementative della view non determinino cambiamenti sul controller o sul model in alcun caso rendendo dunque totalmente indipendente e modificabile l'implementazione. In aggiunta, si potrebbe prevedere l'esistenza di uno specifico componente che implementi il pattern observer con il GameController e che chiami i metodi di quest'ultimo permettendo altre modalità di interazione (cattura evento pressione dei tasti del mouse, della tastiera...) Il software prevede anche un menù iniziale di configurazione della partita. Questo menù è a sua volta costituito da una sua architettura MVC più ridotta, modellata tenendo in mente gli stessi principi descritti sovrastante. L'entità GameBuilder è colei che si occupa di instanziare poi le classi dell'MVC principale del software e avviare effettivamente il gioco, costruendo tutti gli oggetti.

riscrivere menu

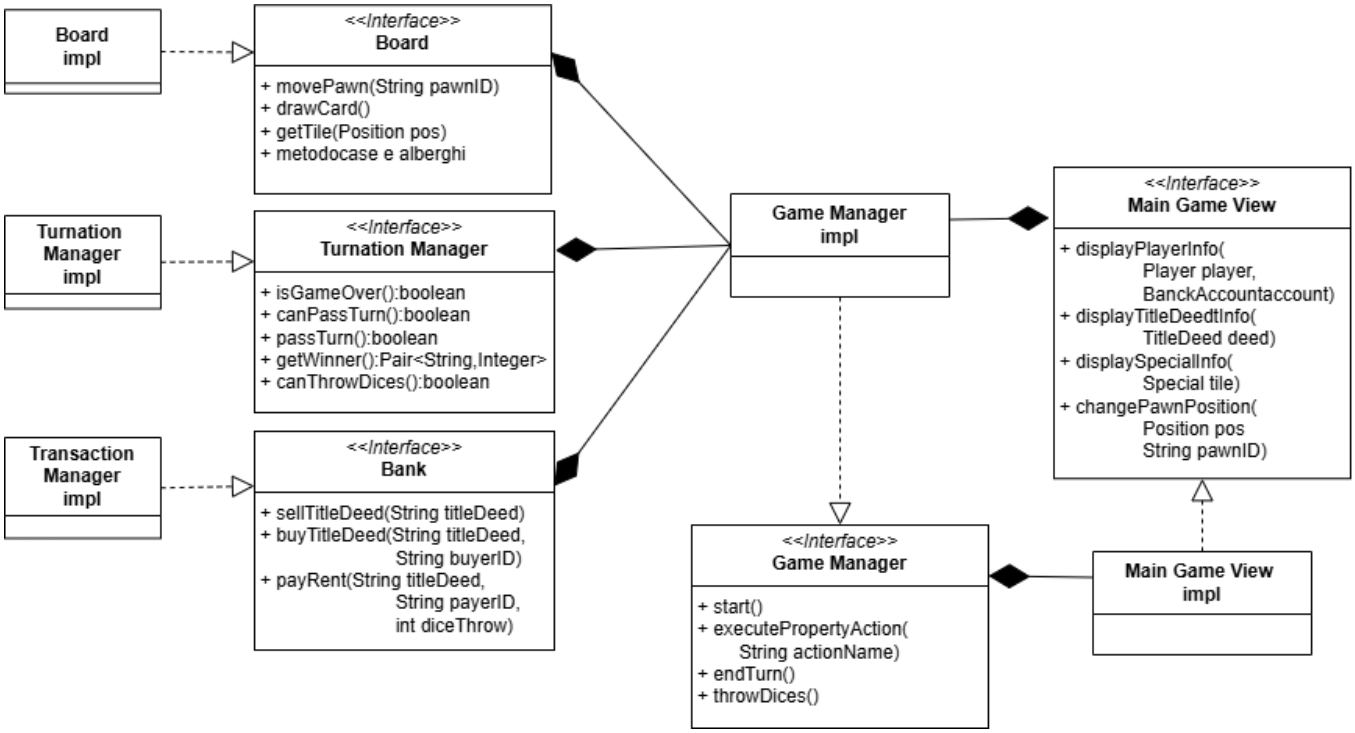


Figura 2.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Cambiare caption

## 2.2 Design dettagliato

### 2.2.1 Davide Rossi

#### Bank e Bankstate

*Problema*
Come detto nell'architettura, `Bank` è l'entita che si occupa della parte di compravendita del gioco, gestendo i `Bank Account` dei giocatori e i `Title deed` associati alle caselle proprietà. `Turnation Manager`, per arbitrare e gestire l'avanzamento del gioco, deve poter comunicare con la banca per richiedere o chiedere di modificare informazioni sulla situazioni finanziaria degli utenti. Ad esempio, quando il giocatore termina il turno bisogna controllare se il suo `Bank Account` è in regola per la prosecuzione del gioco e se ha eseguito tutte le transazioni obbligatorie nel suo turno, se il giocatore ha perso tutti i suoi contratti ritornano disponibili all'acquisto...
`Turnation manager` deve quindi poter fare richieste alla banca per poter ar-

bitrare la partita.
*Soluzione*
Inizialmente era stato delegato al GameController il compito di mediatore fra i due componenti. Tuttavia questo poneva molta responsabilità sul controller aumentandone la complessità e rendendolo più fragile, esponendo l'applicazione a bug vari che si potrebbero originare qualora questo non utilizzasse i componenti in maniera corretta. In più, il controller doveva reperire dalla banca e poi passare a `Turnation Manager` le informazioni necessarie per gestire l'avvicendarsi dei turni di gioco oppure utilizzare queste informazioni per delle operazioni di logica di dominio che non sarebbero di sua responsabilità.
`Turnation Manager` non era quindi progettato per operare in maniera autonoma.
Rifacendoci all'analisi, si è pensato di implementare una qualche sorta di comunicazione fra `Turnation Manager` e `Bank`. Utilizzando il pattern "Adapter" `BankStateAdapter` (Adapter) fa uso dei metodi interni di `BankImpl` (Adaptee) per garantire che questa classe aderisca a un'interfaccia scritta apposta per `Turnation Manager`: `BankState`. In questo modo `BankImpl` aderisce a due interfacce specifiche per i suoi due utilizzatori: il controller usa `Bank` come unico punto d'accesso per l'esecuzione di transazioni, e grazie a `BankStateAdapter` `Turnation manager` può fare uso di un oggetto `BankState` per controllare la situazione bancaria dei giocatori e orchestrare il ciclo di vita della partita senza dipendere dal controller. Infatti, parte delle operazioni di logica del dominio svolte dal controller precedentemente sono state rilocate in `Turnation Manager` come di diritto, in favore di un migliore incapsulamento e una maggiore facilità di utilizzo di questo componente dall'esterno. Allo stesso tempo il fatto che ogni client lavori con un'interfaccia scritta su misura per se limita le operazioni che ognuno può eseguire garantendo maggiore sicurezza e separazione degli interessi.

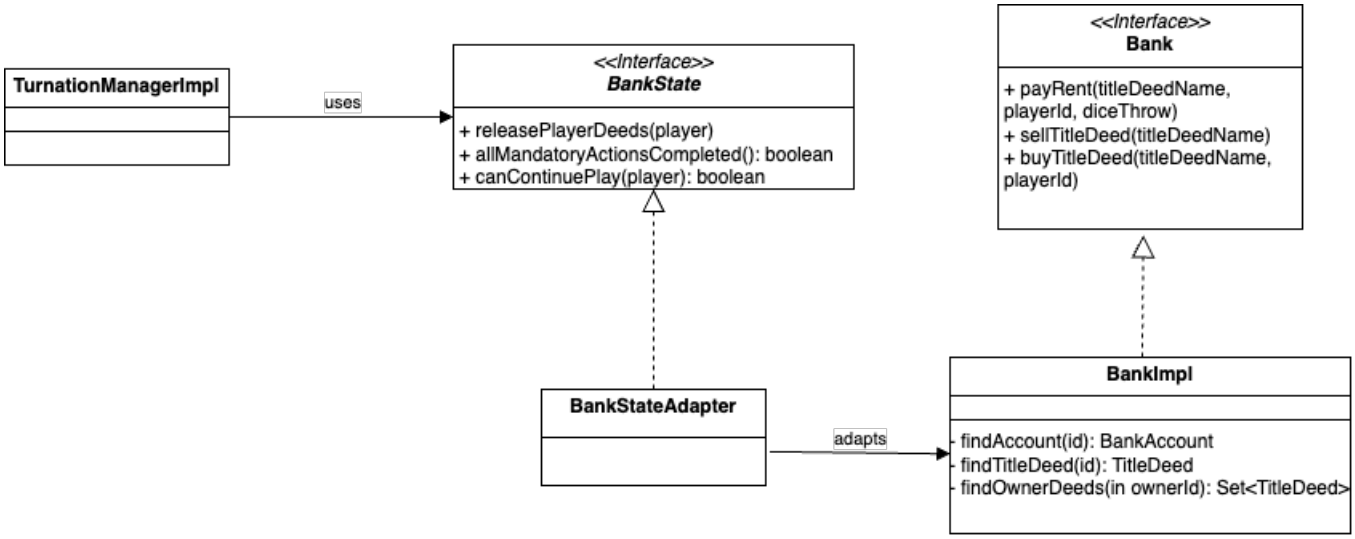


Figura 2.2: Schema UML del pattern Adapter. L'Adapter BankStateAdapter adatta l'oggetto BankImpl all'interfaccia BankState. TurnationManagerImpl utilizza un oggetto BankState per pilotare il gioco. Il controller, qui non riportato per chiarezza, ha un riferimento a un oggetto Bank

### Modellazione dei contratti di proprietà

*Problema*
Rappresentare in un'unico oggetto tutte le informazioni di un contratto di proprietà. In particolare le opzioni di affitto, l'ipoteca e i costi di costruzione, in maniera flessibile e espandibile
*Soluzione*
Si è deciso di adottare il pattern "decorator" per rendere il supporto alle case e alberghi una funzionalità componibile in maniera flessibile, in modo da lasciare agli utilizzatori della classe la scelta di quale implementazione usare permettendo eventualmente di introdurre più modalità di gioco. Lo stesso concetto si può estendere introducendo nuovi decoratori per nuove versioni di `Title deed` per introdurre nuove funzionalità sui contratti.
`TitleDeedDecorator` (decoratore) è una classe abstract che incapsula un oggetto di tipo `Title deed` (decorated) e aderisce alla suddetta interfaccia. `TitleDeedWithHouses` estende questa classe e modifica i metodi riportati nello schema, aggiungendo il supporto per il calcolo del costo di costruzione delle case e dell'albergo, e per il calcolo dell'affitto (`getRent`) considerando anche i miglioramenti (case e albergo) piazzati sulla proprietà.
Analizzando le regole del gioco si è notato che i metodi "costo" (`getHousePrice, getHotelPrice, getMortagePrice`...) calcolano i prezzi basandosi sul prez-

zo di vendita (`getSalePrice`) del contratto. Per implementare questi metodi in maniera flessibile si è deciso di usare il pattern "strategy": i vari metodi "costo" fanno uso di funzioni matematiche che prendono come input il prezzo di vendita. Queste funzioni, o algoritmi, vengono richieste al momento della creazione dell'oggetto lasciando all'utilizzatore la libertà di definirli e personalizzare il contratto.
Ogni implementazione del contratto richiede gli algoritmi necessari per il suo specifico funzionamento

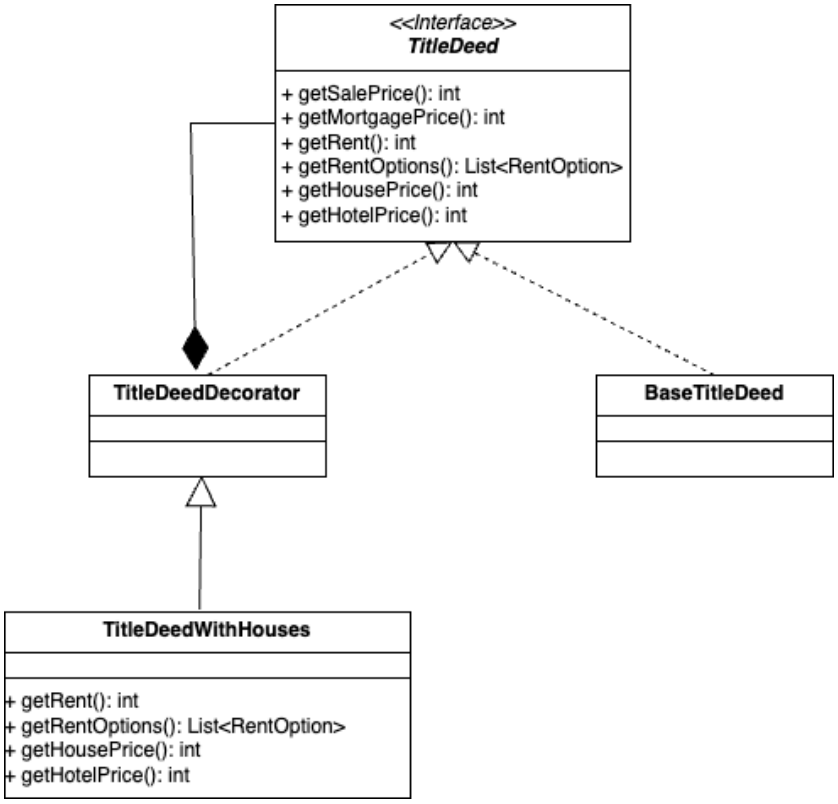


Figura 2.3: Architettura dei contratti di proprietà. BaseTitleDeed è l'implementazione standard dell'interfaccia, è una tipologia di contratto che non ha il supporto per la compravendita delle case. TitleDeedWithHouses reimplementa solo alcuni dei metodi dell' interfaccia mentre i rimanenti chiamano il metodo corrispondente sull'oggetto decorated

### Opzioni di affitto e fabbrica delle opzioni

*Problema*
Rappresentare le varie opzioni d'affitto per ogni contratto di proprietà. Ogni

opzione di affitto ha un suo prezzo e dei criteri di applicabilità. Quando si richiede l'affitto per un contratto di proprietà, generalmente perchè la pedina di un giocatore è capitata su una proprietà non sua ed esso deve pagare, si consulta il contratto e si valuta quali opzioni di affitto sono applicabili. Poi ne viene scelta una (solitamente la più costosa) e il suo prezzo sarà il prezzo concordato per l'affitto.

*Soluzione*

È stata creata un' interfaccia `Rent Option` con dei metodi per richiedere il prezzo e verificare se la suddetta Rent Option può essere applicata o meno, ogni `TitleDeed` ha un elenco di `RentOption`.

Per facilitarne la creazione è stata realizzata una `RentOptionFactory` implementando il pattern "Abstract factory". La factory permette di creare le principali tipologie di opzioni di affitto presenti nel gioco offrendo un certo grado di personalizzazione.

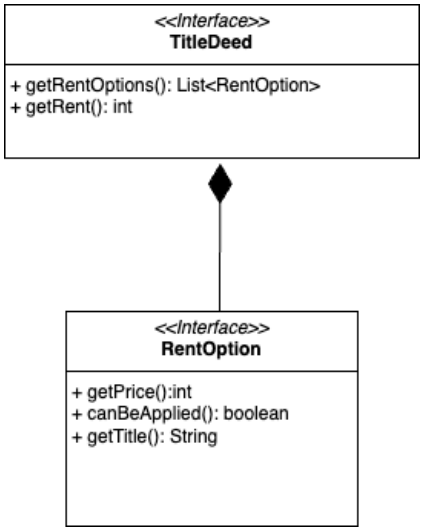


Figura 2.4: Rappresentazione dell'architettura di Title Deed e RentOption

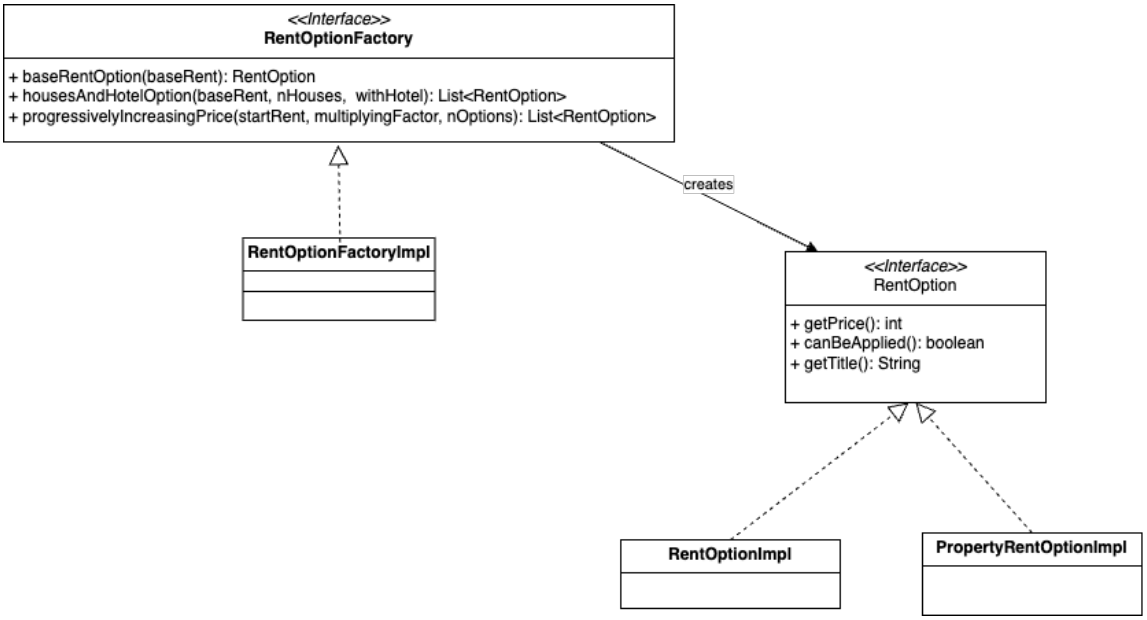


Figura 2.5: Schema UML della factory di RentOption, la factory viene utilizzata dalla parte del sistema che si occupa di creare i contratti. RentOptionFactoryImpl è l'unica implementazione dell'interfaccia RentOptionFactory

### Azioni sulle proprietà

*Problema*
Quando il giocatore finisce su una casella "Property" esso può/deve svolgere determinate azioni nel suo turno. La banca, controllando il `TitleDeed` corrispondente alla casella Property, deve notificare all' utente quali azioni può/deve compiere
*Soluzione*
Si è deciso di applicare il pattern "command". Chiamando `getActionsForTitleDeed` `Bank` restituisce degli oggetti di tipo `PropertyAction`, che sono l'effettivo comando ed incapsulano il codice dell' azione eseguibile. Il controller poi chiede alla view di mostrare le azioni all' utente, che potrà selezionare quale eseguire chiamando `executeAction` sul controller.
In questo modo è estremamente facile estendere il gioco con nuove funzionalità creando nuove `PropertyAction` senza modificare il controller.
Per rendere più facile la creazione sul momento delle azioni e migliorare la riusabilità è stata creata un' "abstract factory" con le azioni principali tipiche del gioco.

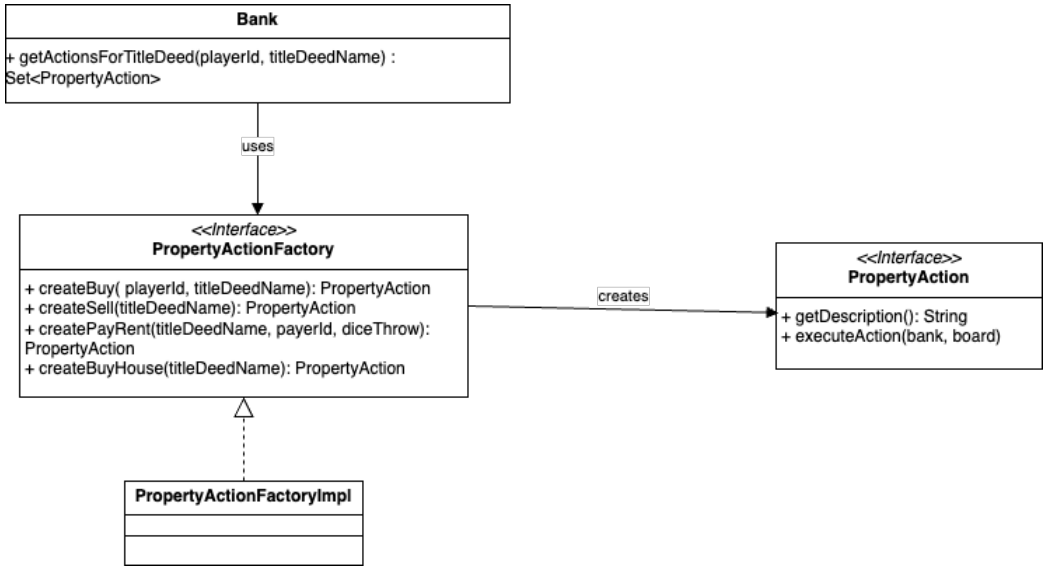


Figura 2.6: PropertyActionFactory è la factory che si occupa di creare le PropertyAction. Bank controlla quali azioni sono concesse all'utente selezionato per il determinato contratto selezionato e adopera la factory per creare i comandi

### 2.2.2 Alessio Minniti

#### Turnation Manager

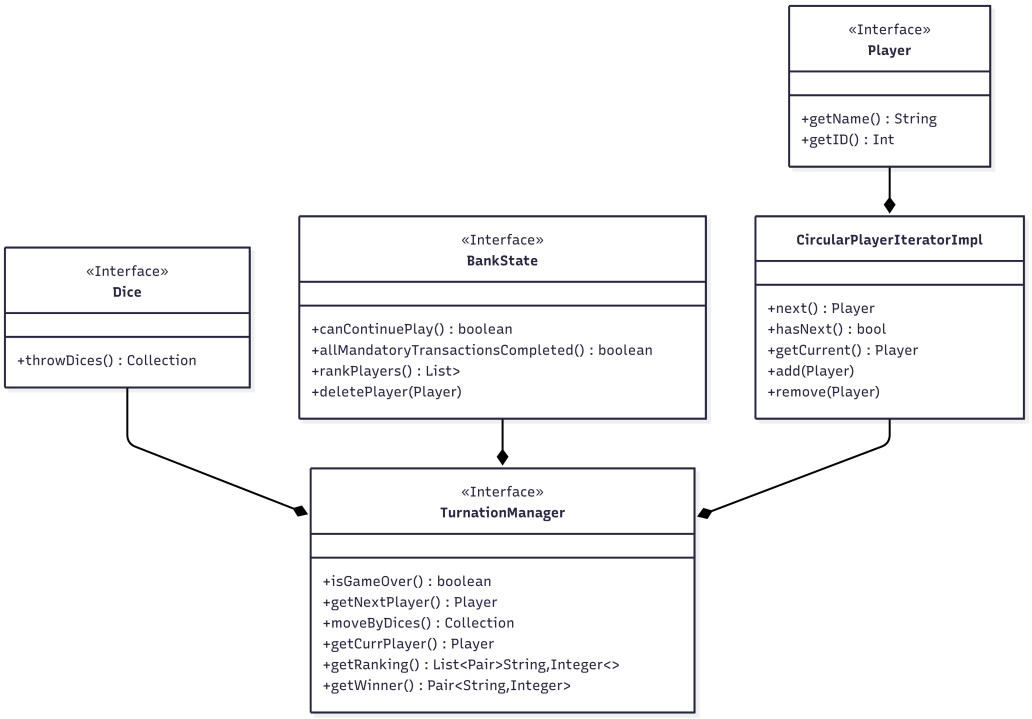


Figura 2.7: Schema UML della struttura del Turnation Manager, il Turnation Manager gestisce una lista di player che viene iterata ciclicamente, il Dice che permette il lancio dei dadi e Il BankState che permette la comunicazione con la Bank

Il **Turnation Manager** ha il compito di gestire lo stato dei **player** e i loro turni, infatti permette e controlla il lancio dei dadi, la terminazione del turno, gli stati di prison e park del player e lo stato del ranking dei players.

Per farlo il Turnation Manager usa una **lista circolare di player** usata per ricavarne il **current player** per capire chi deve eseguire il turno e quando la lista arriva all'ultimo player ricomincia dall'inizio senza però contenere più i player che sono morti nel turno precedente.
Inoltre ha un oggetto **Dice** che all'interno possiede una serie di random in modo che si possano lanciare un numero a piacere di dadi da un numero di facce a piacere che viene definito alla creazione. Con questa realizzazione è quindi possibile avere future implementazioni del gioco dove si usano quanti

dadi si vuole e con quante facce si vuole.
Infine utilizza il **BankState** per controllare che un player abbia fatto tutte le azioni di compravendita obbligatorie prima di terminare il turno. Inoltre bankState viene utilizzato dal Turnation Manager anche per determinare il ranking finale e il vincitore alla fine del gioco.

### Circular Player Iterator

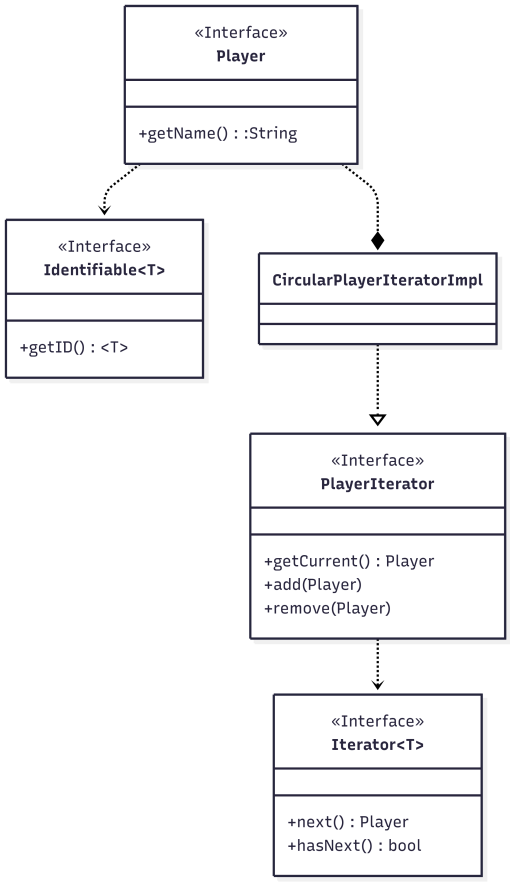


Figura 2.8: Schema UML che definisce il funzionamento della struttura dati circolare dei player, Il Circular Player Iterator è un'implementazione ciclica del Player Iterator, un' estensione di iterator pensata per gestire i player

PROBLEMA: Il Turnation Manager per gestire i turni dei player ha bisogno di una struttura dati che passa sempre il prossimo player in modo ciclico, quindi una volta che arriva all'ultimo player deve ricominciare ad assegnare i turni dal primo player, tuttavia di base non c'è una lista circolare e cercando

tra le librerie esterne non ne ho trovata una che ne forniva una adeguata.

SOLUZIONE: Per risolvere questo problema quindi ho utilizzato il pattern **iterator**, andando a creare un' interfaccia **PlayerIterator** che estende iterator e serve a definire le azioni cardine del circular player iterator che oltre ad avere i metodi di un iteratore normale possiede anche funzioni aggiuntive per gestire la lista di player, l'implementazione del player iterator **Circular Player Iterator** wrappa la lista di player e itera la lista ritornando sempre il prossimo player che deve svolgere il turno in maniera ciclica, quindi quando vede che è arrivato all'ultimo player ricomincia ad iterare la lista dall'inizio.

### Board

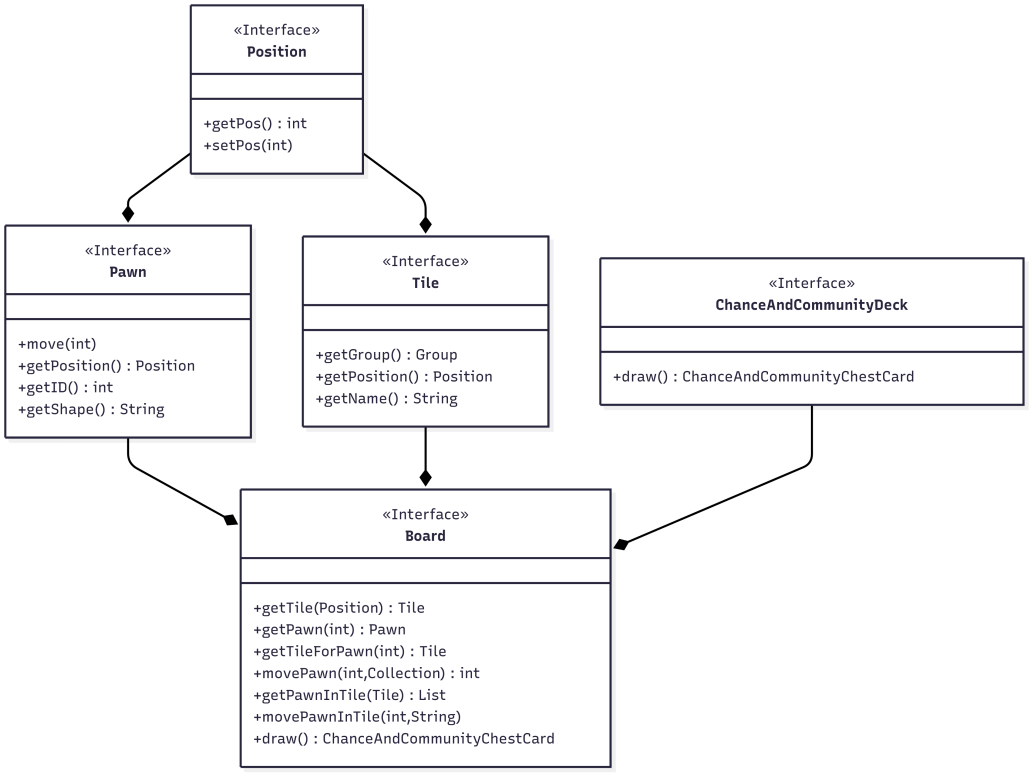


Figura 2.9: Schema UML che definisce il funzionamento di Board, esso gestisce tutte le azioni da svolgere sulle pawns, sulle tiles e sul deck

La **Board** ha il compito di gestire le **Tiles** e le **Pawns** con le loro posizioni, e il **deck** delle carte chance e community, gestisce il movimento delle pawns

ricavato dal lancio del dado che gli viene passato e tiene traccia di tutti i loro cambiamenti di posizioni, tiene traccia di tutte le posizioni delle caselle ed inoltre gestisce anche la creazione e rimozione delle case e alberghi nelle proprietà.

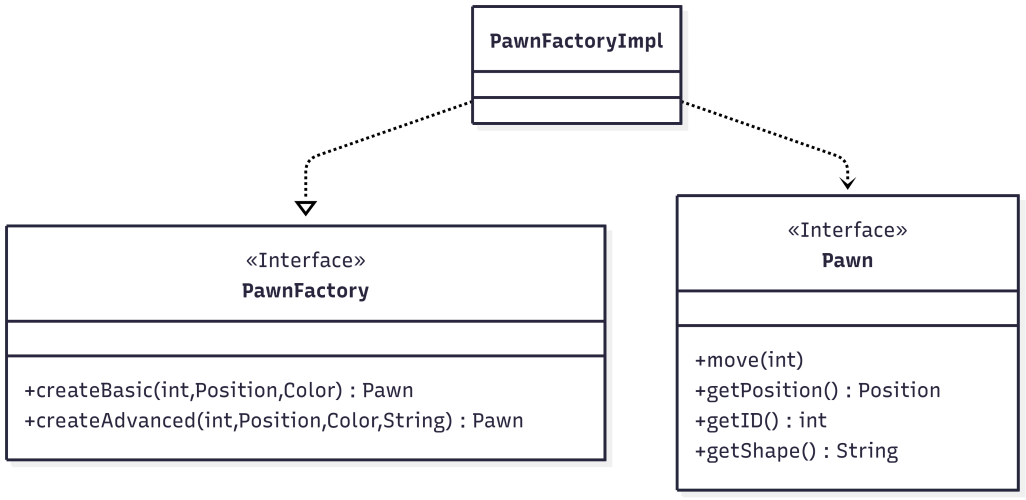


Figura 2.10: Schema UML che definisce la struttura della factory di pawn utilizzata dalla board

PROBLEMA: Quando la board deve restituire le pawns, essendo che non deve permettere il loro cambio di posizione al di fuori della board, essa deve restituire oggetti immutabili per prevenire eventuali modifiche. Per risolverlo si potrebbe richiamare sempre il costruttore delle pawn ma non mi sembrava il metodo più efficiente

SOLUZIONE: Quindi Per facilitare tale operazione ho usato il pattern **factory method**, andando a creare l'interfaccia **PawnFactory** che con la sua relativa implementazione possiede i metodi **createBasic** che permette di creare la pawn normale senza shape e **createAdvanced** che permette di creare la pawn con la shape in caso in cui in future versioni del gioco si voglia far scegliere la forma della propria pedina al giocatore. Usando quindi una factory evito di dover richiamare sempre il costruttore per ritornare copie immutabili.

### Tiles e Property

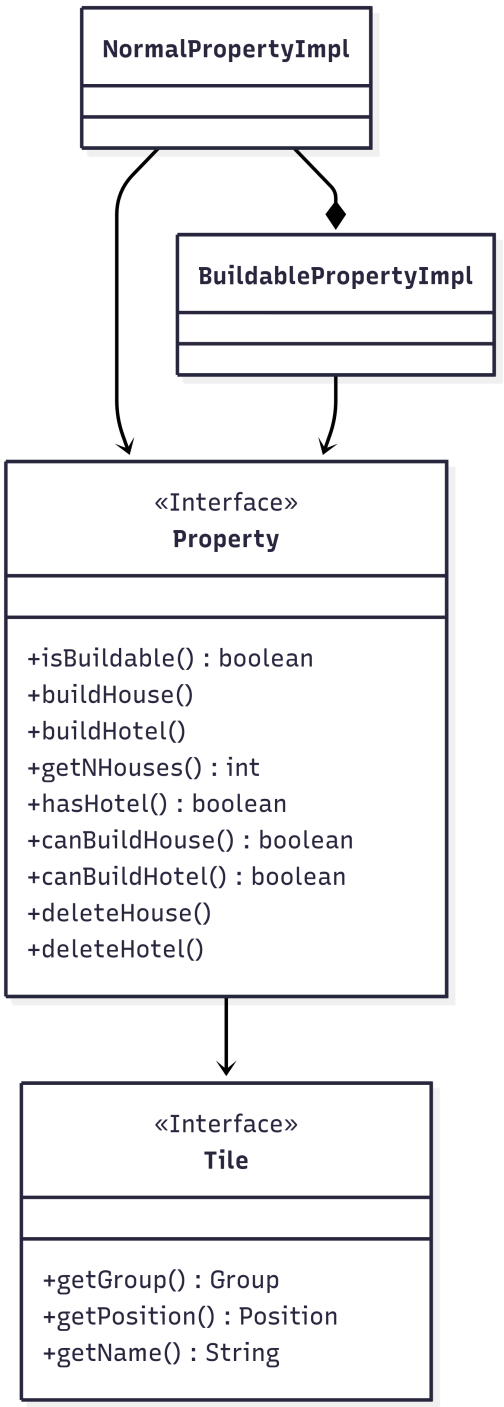


Figura 2.11: Schema UML che definisce la struttura delle tile e più nello specifico delle property, le property normali sono quelle che non posso costruire appalti mentre le buildable property sono i decorator che contengono all'interno le proprietà normali e gestiscono la creazione e rimozione di case e alberghi

Le **Tiles** si suddividono in **Property** e **Special**, queste due tipologie di Tile sono state rappresentate creandole come figlie di Tile, io mi sono occupato della gestione delle proprietà.

PROBLEMA: Le proprietà si dividono in proprietà normali in cui si possono costruire case e hotel e proprietà come stazioni e società che non prevedono tale azione. Tuttavia tutte le altre azioni si possono svolgere in entrambe di esse e quindi risulta che la proprietà con le case e alberghi diventa una proprietà normale con più opzioni, perciò differenziarle in due classi distinte non risultava la scelta ottimale.

SOLUZIONE: Per risolvere questo problema quindi ho scelto di utilizzare il pattern **decorator**, in cui si va a creare una proprietà normale (**NormalPropertyImpl**) che non può avere case e hotel e un decoratore chiamato **BuildablePropertyImpl** che decora la proprietà con le case e gli alberghi, andando quindi a definire le funzionalità in più che non possono svolgere le stazioni e le società.

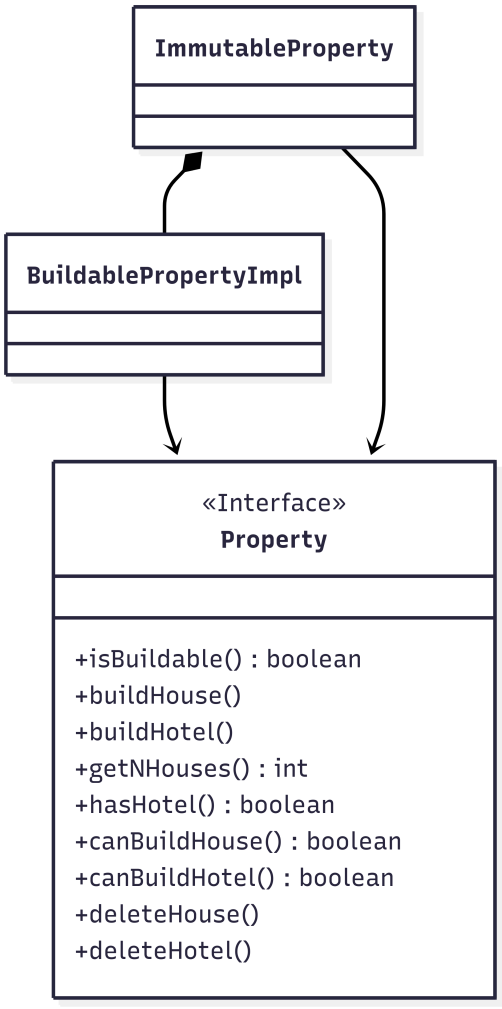


Figura 2.12: Schema UML che definisce la struttura delle immutable property, le immutable property incapsulano le proprietà edificabili per far ritornare solo le informazioni necessarie al contratto

PROBLEMA: Tutte le proprietà hanno il prezzo dell'affitto che bisogna pagare quando ci passa sopra un giocatore che non è owner, tuttavia per le proprietà che possono avere degli appalti tale prezzo cambia in base a quante case sono presenti e se è presente l'hotel. Quindi la banca dovrà possedere tali informazioni relative alla proprietà ma senza entrare in possesso di altri dati non di sua pertinenza. Per farlo quindi si potrebbe ricreare sempre un nuovo contratto in cui alla creazione si specifica il nuovo numero di case e la presenza di hotel, ma tale soluzione risulterebbe molto poco efficiente.

SOLUZIONE: Per risolvere questo problema invece ho utilizzato il pattern **proxy**, andando a creare una copia immutabile della proprietà definita come **ImmutableProperty**, che incapsula la proprietà e restituisce solo le informazioni necessarie per il title deed, mentre le altre sono inaccessibili.

### 2.2.3 Martina Ravaioli

**caselle speciali**

**Problema**
Nel gioco di monopoli è necessario creare delle carte speciali ognuna con il proprio effetto specifico. La creazione di tali carte può portare a proliferazione di classi e difficoltà di manutenzione.
**Soluzione**
In questa soluzione si utilizza il Factory Pattern per centralizzare la creazione all'interno della classe `SpecialFactory` e nasconderne i dettagli implementativi. La factory di speciali ritorna una carta speciale con un effetto specifico, la creazione dell'effetto la delega a sua volta ad una factory di effetti (`EffectFactory`). Limitando la creazione alla factory si evita di violare il principio DRY(Don't Repeat Yourself) per la ripetizione del codice e si rende più semplice l'aggiunta di altre eventuali carte in quanto basta aggiungere un metodo alla classe SpecialFactory promuovendo il SRP (Single Responsibility Principle)

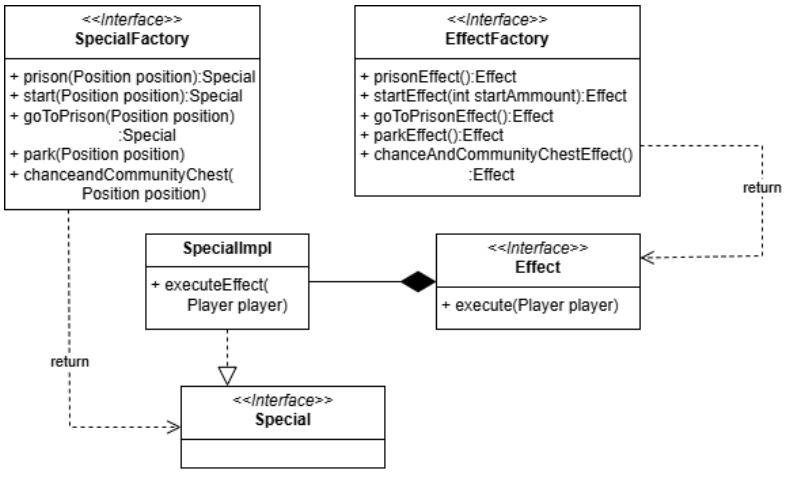


Figura 2.13: Schema UML del pattern factory. SpecialFactory nasconde dentro ogni metdo la creazione delle relative carte che a loro volta delegano la creazione degli effetti al relativo metodo in Effect Factory

**Problema**
Nel gioco ci sono le società, delle proprietà che quando ci arrivi sopra l'affitto da pagare è il tiro del dado moltiplicato per un certo fattore dato da quante società si possiedono. La creazione di una carta a sé può portare a duplicazione di codice.
**Soluzione**
in questa soluzione mi sono ricondotta al pattern decorator, per composizione un `SpecialTitleDeed` ha un (`BaseTitleDeed`) a cui modifica le opzioni di affitto attraverso vari fattori moltiplicativi. Così facendo non ci sarà duplicazione del codice implementativo delle proprietà normali con la sola variazione dell'affitto.

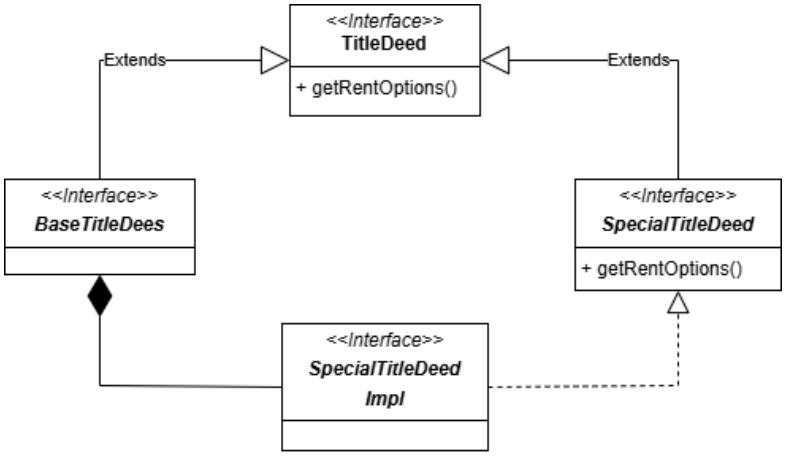


Figura 2.14: Gli SpecialTitleDeed fanno override del metodo getRentOptions

### Player

**Problema**
Nel gioco i giocatori devono poter essere messi in prigione o nel parcheggio e di conseguenza perdere l'abilità di giocare un numero predefinito di turni. Nel caso del parcheggio viene saltato solo il turno successivo ma non c'è modo di evitare questo effetto. Nel caso della prigione i turni che vengono saltati sono 3 ma il giocatore ha la possibilità, prima di perdere il turno, di tirare i dadi per provare ad uscire di prigione facendo un tiro di almeno 2 numeri uguali. Per poter creare dei player che possono essere messi in prigione ma non nel parcheggio e viceversa si dovrebbero creare tutte le possibili combinazioni e dunque duplicare molto codice.
**Soluzione**
per risolvere questo problema mi sono ricondotta al pattern decorator. Nell'implementazione base del player (`PlayerImpl`) sono presenti tutti i metodi

per gestire questi due eventi ma la loro implementazione è delegata a classi specializzate che verranno usate per decorare il player all'inizio del gioco (`ParkablePlayer, PrisonablePlayer`). In questo modo si mantengono in classi separate le implementazioni dei metodi per la gestione di questi due problemi e si rende il gioco più adattivo nel caso in cui si volesse giocare una versione dove l'effetto della prigione o del parcheggio risultino disabilitati

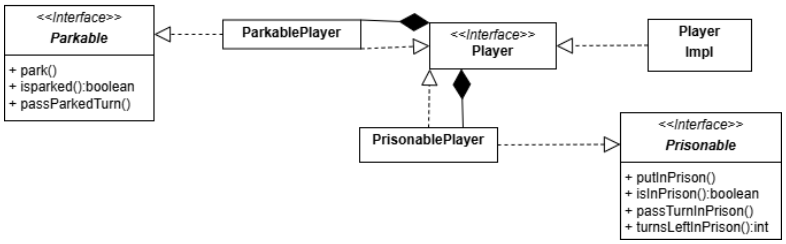


Figura 2.15: Prisonable e parkable fanno entrambi override dei metodi nelle rispettive interfacce

### Imprevisti e Probabilità

#### Problema

Nel gioco ci sono delle caselle speciali di tipo imprevisto o probabilità che fanno pescare una carta da uno fra due mazzi per poi svolgere le azioni presenti su quella carta. Le azioni sono una composizione di altre azioni più semplici che hanno ripercussioni sul turno del giocatore. Scrivere tutto l'effetto all'interno della classe carta porterebbe a rendere il codice poco espandibile.

#### Soluzione

Nella risoluzione di questo problema mi sono rifatta al pattern command. Ci sono due tipi di comandi che incapsulano il concetto di effetto, i comandi con più azioni da eseguire (`ComplexCommand`) e quelli che incapsulano una sola delle azioni di quelli composti `BaseCommand` estendono entrambi la stessa interfaccia. Quelli complessi si salvano una lista di comandi base associati agli argomenti con cui devono essere eseguiti. L'interfaccia `command` ha solo il metodo execute che prende in ingresso il player che ha fatto attivare l'effetto e gli argomenti con cui quell'effetto deve essere attivato. Le carte imprevisto o probabilità si salveranno un comando complesso che verrà attivato quando viene pescata la carta.

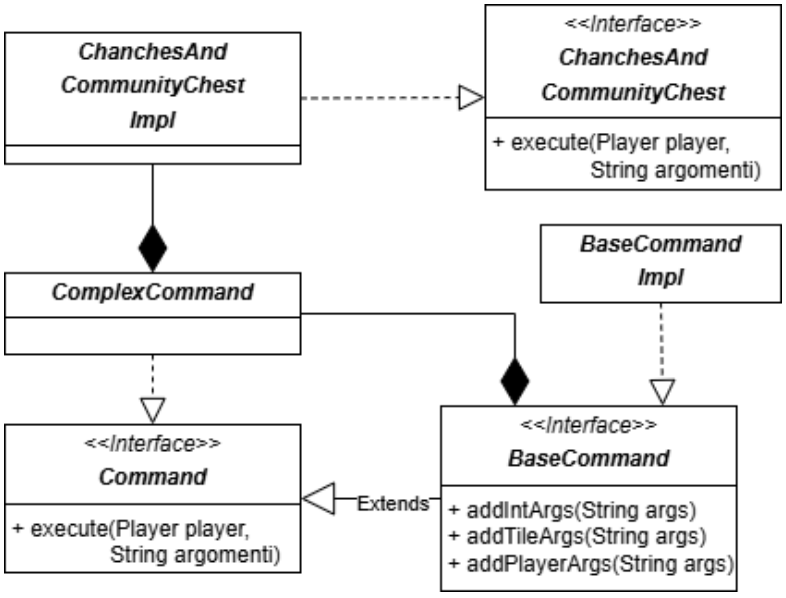


Figura 2.16: schema UML della gestione dei comandi delle carte imprevisto e probabilità

**Problema**

La creazione dei comandi delle carte imprevisti può portare ad un enorme spreco di memoria e una grande proliferazione del codice visto che sono tanti effetti composti dalle stesse azioni base.

**Soluzione**

Per alcune parti di questa soluzione mi sono ispirata al pattern interpreter ma per la maggior parte ho dovuto aggiungere parti per adattarla al problema. L'interpretazione dei comandi delle carte imprevisti e probabilità avviene su 4 livelli e prende le informazioni su come comporre i detti effetti da un file nella cartella di risorse. I 4 interpreter sono fatti per rendere più semplice e più vicina al linguaggio naturale la scrittura del file. Il primo interpreter (`DeckCreator`) prenderà dal file il blocco di informazioni che definisce l'intero effetto della carta e con quel blocco delega la creazione dell'effetto all'interpreter successivo. Una volta ottenuto l'effetto sotto forma di un `ComplexCommand` (Vedi soluzione sopra) creerà una nuova carta imprevisto-probabilità che una volta attivata eseguirà le istruzioni contenute dentro il comando. Dopo aver creato tutte le carte si occuperà di aggiungere il deck Il secondo interpreter (`ComplexInterpreter`) ricevuto il blocco con tutto l'effetto lo scompone, riga per riga nelle varie azioni base previste, ogni riga viene a sua volta divisa in due parti. La prima parte viene passata all'interpreter successivo per sapere quale comando base usare, una volta ottenuto il tipo di comando base questo verrà memorizzato assieme alla seconda parte

della stringa che verrà poi passata al metodo execute quando l'effetto dovrà essere eseguito. Il terzo interpreter (`BaseInterpreter`), attraverso un sistema di keyword univoche per ogni tipo di comando base, ritorna il riferimento al comando base relativo alla stringa passata dall' interpreter precedente. Il quarto interpreter (`ArgsInterpreter`) situato dentro ogni comando base, si attiva quando il comando viene eseguito e si occupa di elaborare la stringa con gli argomenti e aggiungerli nei vari campi per fare in modo che l'azione venga eseguita correttamente. Per ogni tipo di comando base viene creato un solo oggetto contenente le istruzioni da eseguire, quando il terzo interpreter "crea" un nuovo comando base in realtà sta solamente indicando quale dei comandi già creati usare per eseguire l'azione desiderata. Questo è fatto per evitare di creare una moltitudine di oggetti di tipo base command, tutti uguali tra loro tranne che per gli argomenti con cui vengono eseguiti.

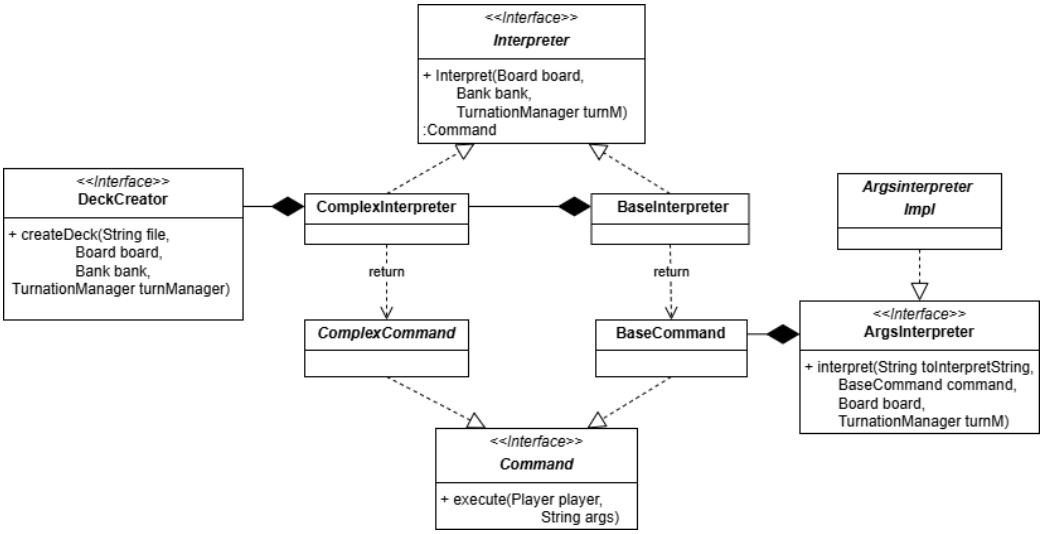


Figura 2.17: Schema UML degli interpreti a cascata per la creazione dei comanndi

## 2.3 Commenti finali

### 2.3.1 Autovalutazione e lavori futuri

#### Davide Rossi

Personalmente ritengo di aver affrontato il lavoro di gruppo con professionalità. Ho sempre cercato di orientare le mie scelte progettuali e implementative favorendo riusabilità e estendibilità, sfruttando le conoscenze apprese nel

corso per giudicare il codice da me prodotto. Ho sempre cercato di immedesimarmi nella realizzazione di un'applicazione in un contesto lavorativo, e quindi lavorare con l'ottica che un giorno le mie parti avrebbero potuto subire delle modifiche o aggiunte che potrebbero essere fatte anche da un'altra persona. Questo è stato per me uno stimolo molto utile per realizzare del codice di qualità e impegnarmi per fare una buona progettazione perché credo che questo vada ad impattare tanto sulla facilità con la quale si possano realizzare le suddette modifiche. Penso di essere stato anche molto bravo a rianalizzare il mio codice, identificare i punti di debolezza, e produrre nuove soluzioni più congeniali. A volte tuttavia tendo a prediligere soluzioni eccessivamente complesse, realizzando componenti che fanno ben più di ciò che è richiesto e ho notato che questo si è spesso rivelato un problema perché seppure lo facessi per favorire l'espandibilità spesso semplicemente il risultato era un sistema complesso e difficile da utilizzare. Una direzione più congeniale sarebbe quella di ideare soluzioni che rispondano al problema che si sta trattando, e che tuttavia lascino la possibilità di essere modificate per il proprio scopo. All'interno del gruppo ho assunto un ruolo un po' centrale e di riferimento per gli altri membri. Sebbene io mi sia esclusivamente occupato della parte di progetto a me assegnata spesso gli altri ragazzi si rivolgevano a me per pareri ed avevo sempre un'idea chiara dello stato generale di avanzamento dei lavori. Se si dovesse portare avanti il progetto spenderei maggiori energie per rivedere la progettazione dell'architettura del model. Credo che parte della debolezza dell'architettura attuale sia dovuta al fatto che i componenti principali hanno necessità di scambiare dati tra di loro e per fare ciò sono state prodotte delle soluzioni funzionanti e accettabili che tuttavia avrebbero del margine di miglioramento. Penso che rivisitare l' architettura darebbe la possibilità di costruire un model ancora più facile da utilizzare, espandere e più resiliente ai problemi.