

Relazione di progetto di Programmazione a Oggetti

“Monopoly edizione Rimini”

Gianmaria Di Fronzo

Giovanni Muccioli

Lisa Vandi

Riccardo Polazzi

23 febbraio 2024

Indice

1. Analisi	2
1.1 Requisiti	2
1.2 Analisi e modello del dominio	3
2. Design	5
2.1 Architettura	5
2.2 Design dettagliato	6
3. Sviluppo	18
3.1 Testing automatizzato	18
3.2 Note di sviluppo	19
4. Commenti finali	22
4.1 Autovalutazione e lavori futuri	22
A Guida utente	25
B Esercitazioni di laboratorio	26
B.0.1 gianmaria.difronzo@studio.unibo.it	27

Capitolo 1

Analisi

Il software si pone come obiettivo l'implementazione in grafica 2D del noto gioco da tavolo "Monopoly", ora ambientato nella città di Rimini.

Il Monopoly è un gioco di società basato sul concetto di monopolio, di acquisto, di vendita e di gestione delle proprietà immobiliari. Lo scopo principale è accumulare ricchezza attraverso strategie finanziarie e abili negoziazioni. I giocatori, infatti, si muovono lungo il tabellone di gioco acquistando e sviluppando proprietà, riscuotendo affitti dagli avversari che si fermano su di esse in modo da poterne trarre profitto.

Il vero obiettivo è quindi eliminare gli avversari finanziariamente, costringendoli alla bancarotta. In particolare, ogni giocatore a turno si muove lungo il percorso sulla base del punteggio totalizzato dal lancio di due dadi. La partita avanza secondo la logica di gioco generale e le strategie personali del giocatore, fino a quando tutti i giocatori, fuorché uno, terminano la propria somma di denaro a disposizione.

L'ultimo giocatore rimasto in gara sarà il vincitore del gioco: il cosiddetto "monopolista".

1.1 Requisiti

Requisiti funzionali

- Il software dà la possibilità all'utente di giocare tramite un apposito tabellone: esso rappresenta il cuore del gioco ed è suddiviso in differenti tipologie di caselle.
 - Caselle acquistabili ed edificabili: esse sono le caselle che permettono al giocatore di ottenere maggior ricchezza e potere, consentendogli di sviluppare le proprie proprietà attraverso la costruzione di case.
 - Caselle acquistabili ma non edificabili: esse aggiungono una componente di investimento strategico al gioco, poiché i giocatori proprietari potranno guadagnare potenziali affitti dagli avversari.
 - Caselle non acquistabili: esse aggiungono varietà e interazioni più complesse al gioco. Tra queste, particolari sono i cosiddetti "imprevisti", ossia caselle in cui il giocatore di turno è obbligato a seguire le indicazioni in esse contenute, positive o negative che siano. Contengono istruzioni o situazioni inaspettate che possono influenzare il gioco, aggiungendone un elemento di imprevedibilità, poiché i giocatori devono adattarsi alle diverse situazioni che possono emergere durante la partita.

- Ciascun giocatore sarà in grado di spostarsi sul tabellone a seguito del risultato ottenuto dal lancio di due dadi. Una volta raggiunta la casella destinazione, allora saranno applicate le regole specifiche della casella.
- Quando un giocatore atterra su una casella di proprietà che non è ancora stata acquistata, ha la possibilità di comprarla. Dopo l'acquisto, il giocatore diventa il proprietario della casella e può iniziare ad edificare costruendo case (se la casella è ovviamente edificabile). Gli altri giocatori, quando atterrano su questa casella, devono pagare un affitto al proprietario in base al numero di case presenti sulla proprietà e al dipendere dalle specifiche regole di affitto per quella particolare proprietà.
- Tutti i giocatori sono posti nelle stesse condizioni economiche iniziali. La disponibilità economica di ciascun giocatore varierà turno dopo turno, al dipendere dall'imprevedibilità del gioco ma anche dall'astuzia e dalla strategia del singolo giocatore.
- I giocatori non hanno alcun limite di acquisto nel numero delle proprietà: se un giocatore può permettersi economicamente l'acquisto, gli è sempre permesso di farlo. D'altra parte, il giocatore è vincolato al costruire al massimo su ogni proprietà edificabile 3 case.
- Un giocatore può scegliere di vendere una proprietà per incassare denaro. Qualora termini su una proprietà altrui o su un imprevisto che non può permettersi economicamente, allora è eliminato dal gioco.

Requisiti non funzionali

- L'applicazione dovrà essere portabile: dovrà essere supportata dai sistemi operativi Windows, MacOS e UNIX/Linux.
- L'applicazione dovrà essere fluida nelle dinamiche di movimento.

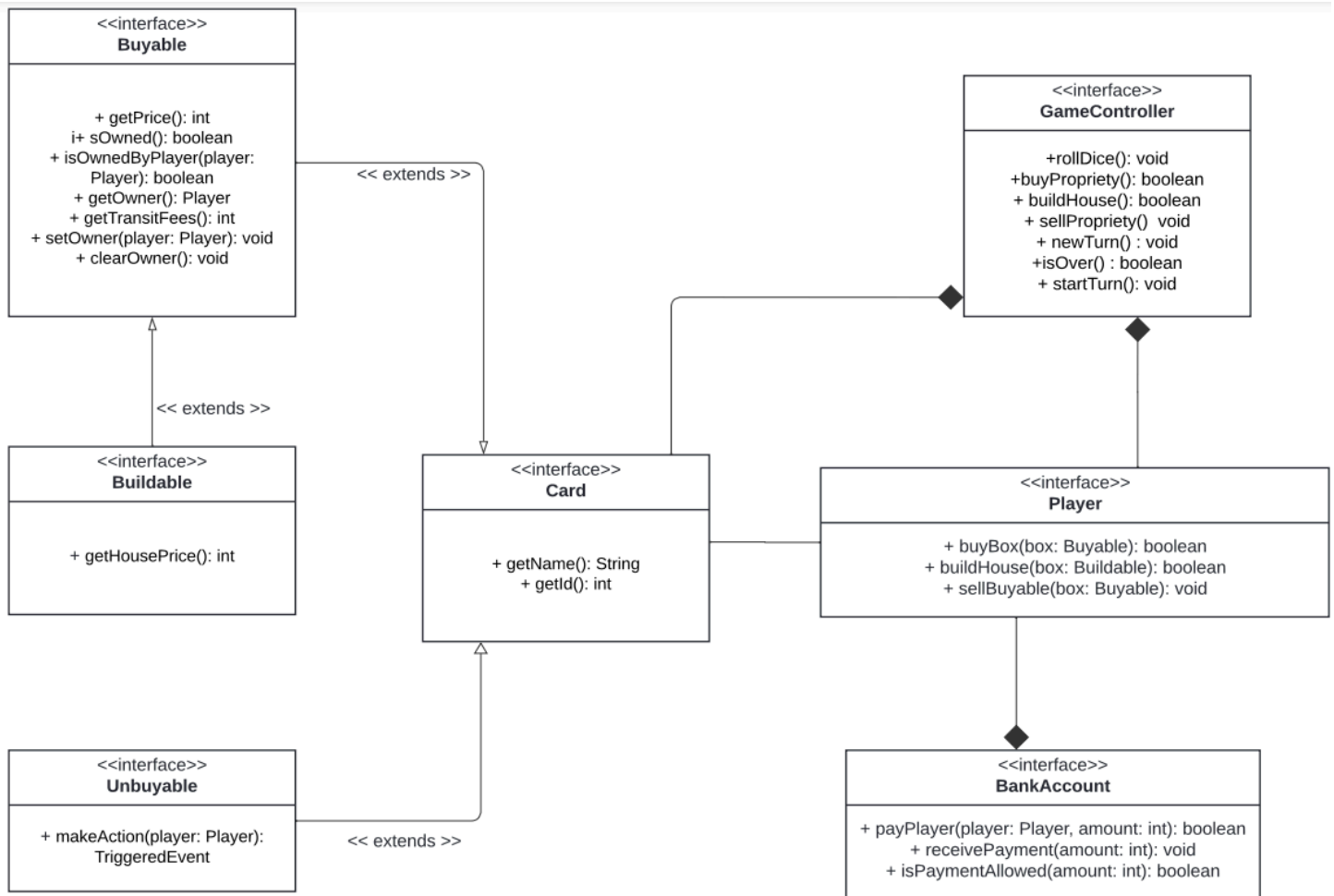
1.2 Analisi e modello del dominio

“Monopoly edizione Rimini” è incentrato sui concetti di Giocatore (Player), Carta (Card) e Turno (Game). In particolare, il dominio di gioco si compone dei domini del giocatore e delle carte.

Gli elementi appartenenti al dominio delle Carte rappresentano le singole caselle del tabellone di gioco, su cui si spostano i giocatori durante il corso dei turni. Invece, le entità appartenenti al dominio del Giocatore consistono in coloro che partecipano al turno di gioco: ciascuno ha associato un conto bancario che gli consente di portare avanti le dinamiche di

compravendita applicate alle Carte.

L'interazione tra giocatori e carte è gestita dal dominio del turno, che coordina le azioni che un giocatore può eseguire durante l'avanzamento della partita.



Capitolo 2

Design

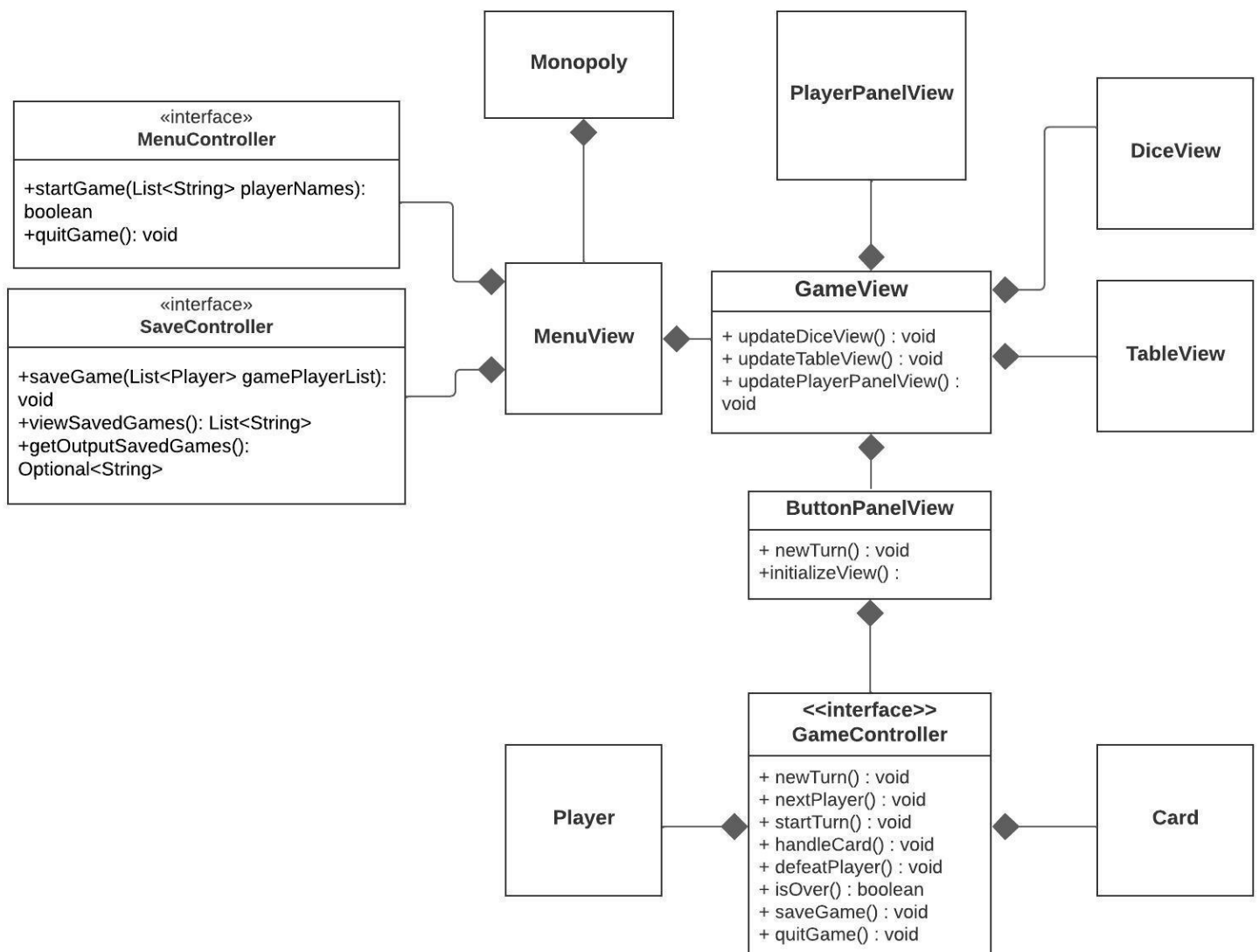
2.1 Architettura

Il software utilizza MVC come pattern architetturale, con l'obiettivo di limitare quanto possibile le dipendenze tra le componenti di model, view e controller.

All'avvio di ogni singola partita, la view del Menù accetta in input dall'utente la lista dei nomi dei giocatori da inserire, gestita dal relativo controller. Inoltre, la view del Menù fornisce in output i dati salvati relativi alle partite precedentemente terminate, operazione anch'essa coordinata dal rispettivo controller. All'interno della view del Menù, viene istanziata la view del Gioco: questa restituisce in output i dati relativi a ogni singola classe di view creata (dadi, tabellone, dati informativi del giocatore corrente).

Il software è coordinato dal controller di gioco: esso modifica le componenti di model, e, in base a questa, viene aggiornata di conseguenza la view.

L'utilizzo vantaggioso e corretto del pattern MVC fa sì che la componente grafica potrà essere sostituita in futuro, senza impattare le parti di Model e Controller, dato che non sono state create dipendenze con esse.



2.2 Design dettagliato

2.2.1 Gianmaria Di Fronzo

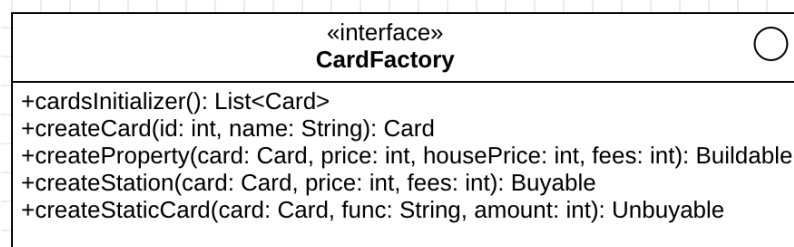
Le soluzioni messe a punto servono a reindirizzare il tabellone completo con le proprietà del gioco, comprese di stazioni, imprevisti, prigione. Le difficoltà emerse erano molteplici, di seguito vengono illustrate le tecniche di approccio alla risoluzione del problema.

Problema: costruire oggetti diversi ma correlati.

Il primo step è quello di salvare una lista di carte ordinata, questa lista rappresenta le caselle del tabellone di gioco.

Soluzione: Factory Method Pattern

La lista di carte ha differenti tipologie, per questo si ricorre al Factory Method pattern per creare diverse istanze dell'interfaccia Card, le quali verranno poi salvate in un'unica lista. La Factory in questione si occupa di creare carte di tipo comprabili, non comprabili, costruibili. Il primo metodo della factory è quello più importante, ovvero restituisce la lista delle carte del tabellone, che potrà poi essere usata per l'intero gioco.



Dipendenze: JSON Library

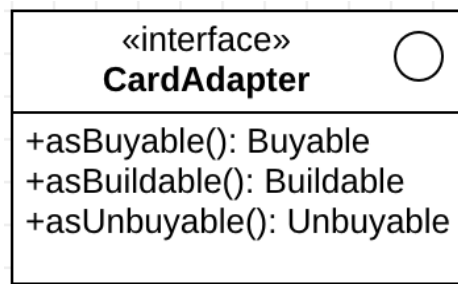
In quanto la lista delle carte è univoca e immutabile nel corso di tutto il gioco, viene utilizzato un file di formato json per leggere ogni singola casella del tabellone con le sue relative proprietà.

Problema: lettura della lista di carte

La lista restituita dalla factory che rappresenta il percorso di caselle del tabellone è di tipo Card, per questo può nascondere in sé dei sottotipi.

Soluzione: Adapter Pattern

Per facilitare l'interagire con gli elementi della lista l'interfaccia Card racchiude dei metodi di default per controllare la tipologia, dopodiché si sfrutta l'Adapter pattern per adattare un oggetto di tipo Carta ai suoi relativi sottotipi. L'uso di questo pattern facilita la compressione delle varie tipologie di carte in un'unica lista e sopprime un abuso di cast nel codice.



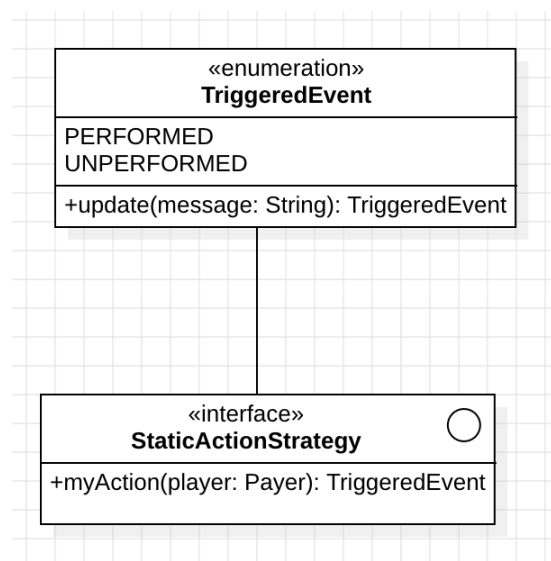
Problema: definire diverse azioni interscambiabili.

Nel caso delle carte statiche e non comprabili, alcune di esse hanno delle azioni da eseguire su dei giocatori, uniche nel loro genere in base alla carta, queste carte non sono solo parte del tabellone ma comprendono anche le carte degli Imprevisti, che vengono pescate in maniera random

Soluzione: Strategy Pattern

Le carte statiche del tabellone sono state implementate attraverso una singola classe che implementa la loro interfaccia, mentre gli imprevisti vengono rappresentati attraverso le enumerazioni.

Ognuna di queste carte è di tipo statico (Unbuyable) perciò richiede l'implementazione di un'interfaccia funzionale di tipo `StaticActionStrategy`, questa interfaccia è basata sulle lambda expression per facilitare l'implementazione o la modifica, vuole rappresentare quindi un'azione richiamata dalla carta di tipo non comprabile su un giocatore passato in input. Le varie strategie vengono implementate in base ad un "nome azione", ovvero una proprietà di tipo stringa della carta stessa, letta dal file di formato json come riportato nel paragrafo sopra.



Problema: modificare graficamente le posizioni dei giocatori.

Ogni giocatore è rappresentato da un codice colore univoco, e deve essere spostato in base agli eventi scatenati nel turno.

Soluzione: Observer Pattern

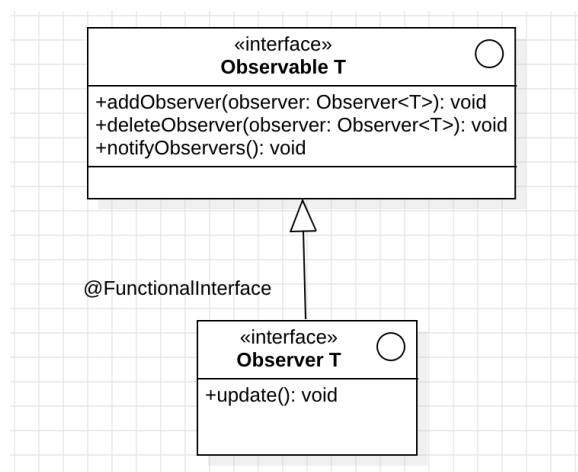
Il primo approccio al problema è stato quello di definire semplicemente due metodi all'interno della View, rispettivamente per cancellare e ridisegnare un giocatore su una posizione diversa, prendendo in input il codice colore del giocatore che si vuole spostare (sufficiente in quanto univoco) e la nuova posizione per disegnare, la posizione attuale se si vuole cancellare il giocatore.

A questo punto entra in aiuto l'observer pattern in quanto si potrebbe desiderare di eseguire una sequenza di eventi tra cui lo spostamento dei giocatori a livello grafico (un esempio di eventi da richiamare è lo spostamento su una casella susseguito dal ricevimento di una somma in denaro).

Viene così definita un'interfaccia Observable con un tipo generico, questa interfaccia si occupa di rappresentare una classe che deve essere osservata, cioè scatena la serie di eventi.

Observable fa uso dell'interfaccia funzionale Observer che definisci l'azione da richiamare.

A livello grafico l'intero pannello del tabellone implementa Observable di tipo Player, questa classe tiene traccia quindi delle azioni da richiamare sui singoli giocatori implementate attraverso lambda expression, un esempio è quindi l'implementazione dello spostamento di uno o più player attraverso il richiamo di notifica da parte della classe osservata.



2.2.2 Giovanni Muccioli

- Menù di gioco

Problema: all'avvio del software, i giocatori devono interagire con un menù di gioco che consente loro di avviare correttamente (o meno) la partita. Quindi, il problema consiste nell'inserimento dei nomi dei giocatori e garantire che siano unici nella partita.

Soluzione: viene implementato un menù di gioco semplice e interattivo (MenuView) che consente ai giocatori di inserire i loro nomi, a seguito della scelta del numero di partecipanti.

La soluzione si caratterizza dalla presenza di controlli per evitare nomi identici e/o “falsi-nomi” (come per esempio il mancato inserimento di un nome) grazie all'utilizzo combinato di Set e Stream nell'apposito controller del menù (MenuController).

La decisione è infatti ricaduta sull'utilizzo di un controller che intercetta gli eventi della view, comunicando di conseguenza gli associati cambiamenti a quest'ultima a seguito dell'aggiornamento del modello.

Così facendo si è deciso di separare le varie responsabilità, in cui il concetto di “Player” nasce ed esiste solo se la partita viene avviata correttamente. Viene quindi creata un'istanza di Player per ogni giocatore inserito, con dati iniziali come la posizione sul tabellone, le proprietà in possesso e la disponibilità economica, i quali saranno aggiornati dinamicamente in base alle dinamiche di gioco durante ciascun turno.

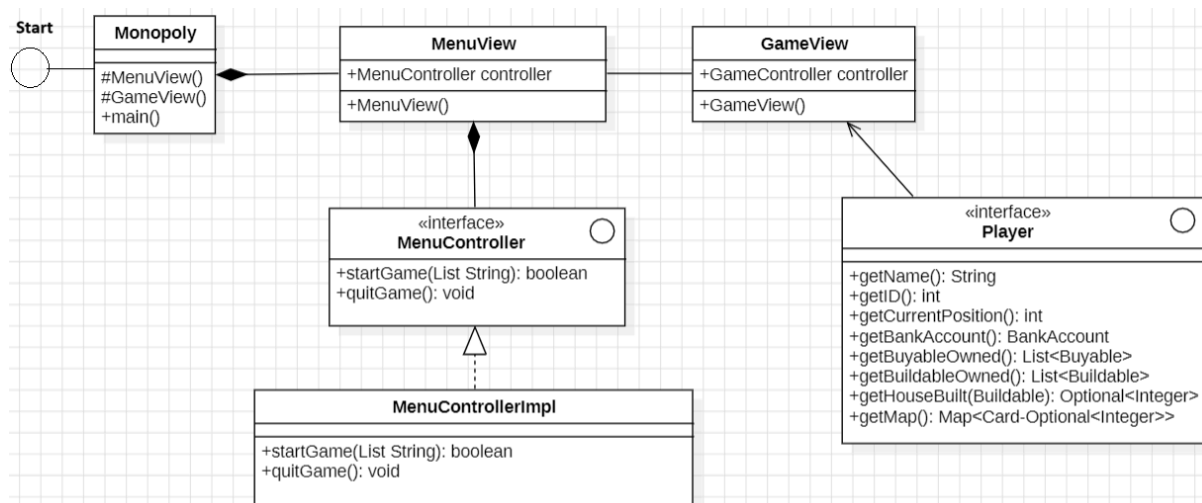
Se quindi l'inserimento è avvenuto correttamente, si transita dalla GUI del menù (MenuView), in cui non esiste l'istanza “Player”, alla GUI di gioco (GameView), in cui ora sussiste il concetto di player.

Pro:

- Gestione efficiente dei nomi dei giocatori con prevenzione di duplicati.
- Separazione del concetto di “potenziale partecipante” da “Player”.
- Separazione delle responsabilità all'interno dell'applicazione, rendendo il codice più modulare, manutenibile ed estendibile.

Contro:

- Limitazioni associate all'operazione di uscita “improvvisa”, la quale dovrebbe essere rivisitata per eventuali estensioni future del gioco che potrebbero richiedere la gestione di sessioni.



- Salvataggio delle partite con resoconto

Problema: gestire il salvataggio dati dei Player. Il software, di conseguenza, dovrà permettere anche la visualizzazione dei dati di gioco salvati.

Il sistema, dovendo quindi supportare output, richiede che vi sia un logger che stampa su file eventuali errori, e un'interfaccia grafica che sia in grado di mostrare la rappresentazione grafica dei dati salvati.

Soluzione: viene adottato un sistema di salvataggio dei dati di gioco che consente ai giocatori di registrare lo stato di ciascun Player della partita corrente, includendo dati come nome di gioco, numero identificativo, posizione corrente, disponibilità economica e proprietà in possesso.

Si è deciso quindi di raggruppare il tutto dietro l'interfaccia SaveController, la quale fa uso delle classi PrintWriter e BufferedReader rispettivamente per salvare i dati e leggere i dati salvati. Si fa quindi utilizzo del costrutto try-with-resources per gestire automaticamente il processo di chiusura delle risorse utilizzate.

Dunque, l'interfaccia SaveController ricopre il ruolo di controller, garantendo da un lato sia la separazione delle responsabilità utili al solo salvataggio e alla visualizzazione dei dati, e dall'altro lato una certa estensibilità, non dovendo ridisegnare completamente l'architettura.

La visualizzazione dei dati salvati è permessa tramite un'apposita GUI (SavedGamesView) la quale permette di visualizzare i dati provenienti dal modello a seguito dell'interazione con l'apposito controller sopra descritto.

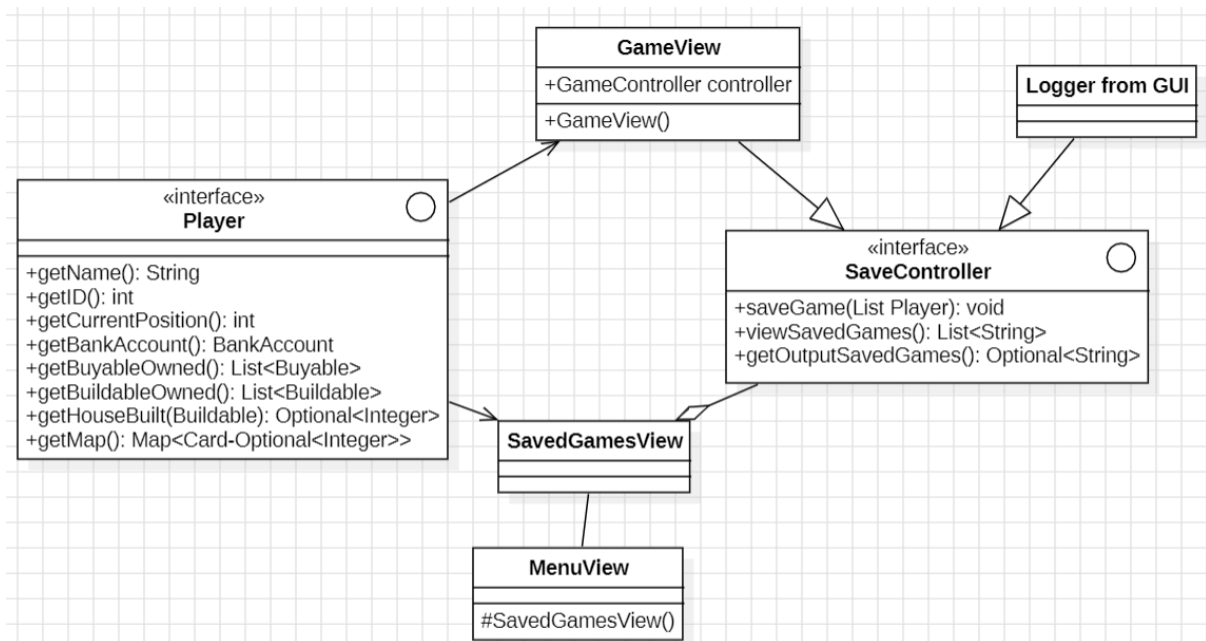
Pro:

- Separazione di responsabilità e codice estendibile.
- Gestione efficiente del recupero dei dati di gioco tramite buffering dei dati, che ne migliora le prestazioni.
- Stampa formattata dei dati per rendere l'output facilmente leggibile e comprensibile.

- Gestione e controllo degli errori/eccezioni tramite logger.

Contro:

- Non vi è un'elevata efficienza nella scrittura (e quindi nel salvataggio) di dati, la quale è passata in secondo piano a seguito della scelta di salvare dati in modo formattato e anche a causa della presenza di una quantità di dati piuttosto limitata per il nostro contesto.
- L'attuale implementazione non include la serializzazione dei dati di gioco, il che significa che (volutamente) i salvataggi sono limitati alla sessione di gioco corrente. La mancanza di serializzazione impedisce ai giocatori di riprendere la partita in un momento futuro, quindi non esiste il concetto di "ripresa della partita interrotta".
- Per motivi di semplicità, i dati di gioco vengono salvati in un file di testo. Questo approccio può introdurre di certo forti limitazioni in termini di efficienza e sicurezza (se comparato per esempio con un database).
- Mancata adozione del formato JSON per salvataggio e visualizzazione dei dati.



- Lancio dei dadi

Problema: ciascun Player necessita di lanciare due dadi per determinare il suo spostamento e quindi il suo avanzamento sul tabellone di gioco durante il proprio turno.

Soluzione: la scelta è ricaduta sull'implementazione di una funzione di lancio dadi che generasse in modo randomico due numeri tra 1 e 6, simulandone il lancio. Il risultato viene poi sommato di conseguenza per determinare il numero di caselle su cui avanzare nel tabellone.

Tale azione infatti è la responsabile del movimento e della dinamicità di gioco, la quale permetterà quindi variazioni ai dati non statici dei Player (dove con "dati statici" si intendono

nome e id, mentre i restanti dati come posizione, disponibilità economica, numero di proprietà in possesso, ecc. sono tutti “dati non statici”, ossia dinamici).

Inizialmente avevo preso in considerazione l’uso di un algoritmo pseudo-randomico per la generazione dei due numeri, ma alla fine ho optato per una soluzione più semplice per garantire in particolare chiarezza di codice e semplicità.

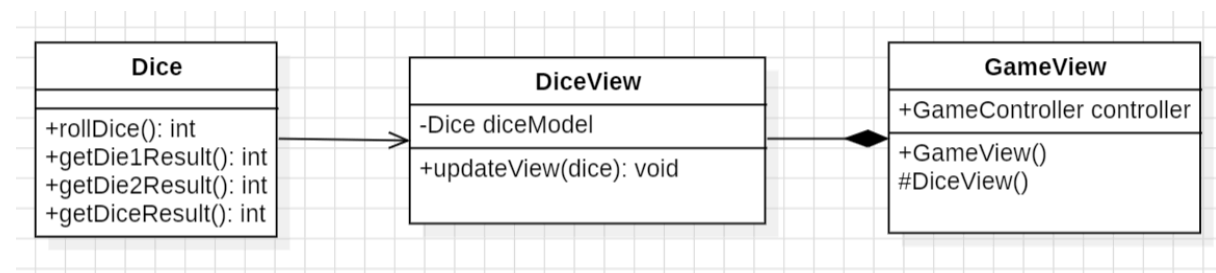
Anche in questa soluzione, la view dei dadi (DiceView) si aggiorna sulla base del modello (Dice) che viene aggiornato a seguito dell’interazione dell’utente con l’apposita dinamica di gioco che ne permetterà poi lo spostamento.

Pro:

- Semplicità nell’implementazione.
- Efficienza nel generare numeri casuali.
- Conseguente chiarezza nella gestione del movimento del giocatore.

Contro:

- Possibile mancanza di realismo nella simulazione del lancio.

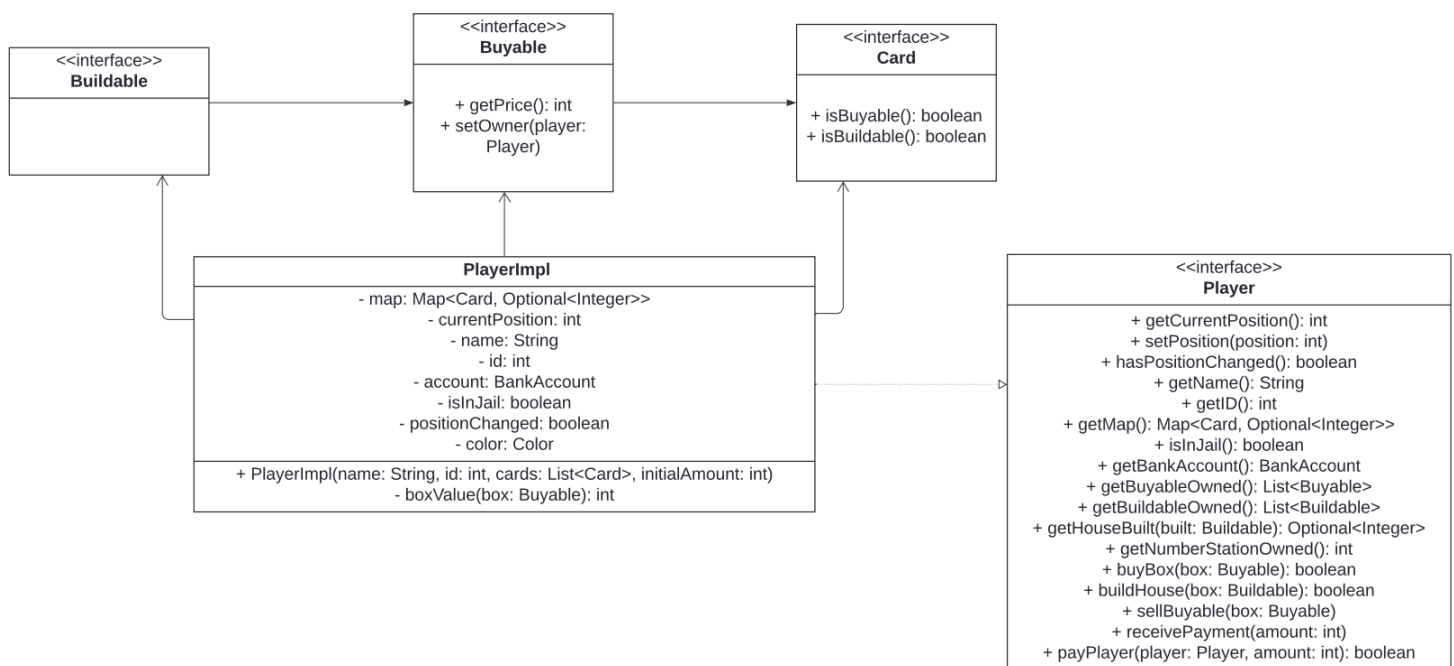


2.2.3 Lisa Vandì

- Gestione delle proprietà di un giocatore

Problema: in fase di analisi, si è dovuto pensare a come associare le proprietà possedute da un giocatore allo stesso. Infatti, un Player può acquistare caselle di tipo Buyable, ma può edificare case solo se la casella acquistata è di tipo Buildable. Buyable è superclasse di Buildable, Card è superclasse di Buyable.

Soluzione: ho sfruttato una Map<Card, Optional<Integer>>, la quale associa ad ogni casella del tabellone di gioco un Optional<Integer>. Se esso è Optional.empty(), allora il giocatore non possiede tale casella, altrimenti tale valore indica il numero di case che il Player ha edificato sulla Card acquistata.

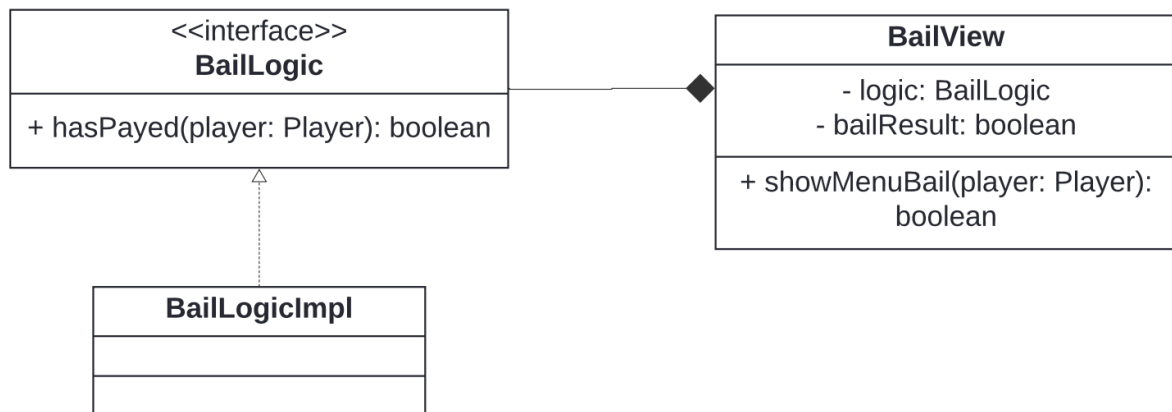


- Gestione della cauzione e del rispettivo popup informativo.

Problema: gestione del meccanismo di cauzione legato all'imprigionamento di un giocatore al passaggio sulla casella "Prigione", con riscontro a livello grafico mediante un popup che propone al player il pagamento della cauzione.

Soluzione: al fine di mantenere la coerenza con il pattern architetturale MVC, ho sfruttato il **Dependency-inversion principle**, con l'obiettivo di separare la logica che regola l'uscita di prigione dalla rispettiva implementazione grafica. In particolare, il DIP è uno dei cinque principi SOLID della programmazione a oggetti, il quale afferma che *"i moduli di alto livello non devono dipendere da quelli di basso livello, entrambi devono dipendere da astrazioni"*. Questo mi ha portata alla creazione di un'interfaccia **BailLogic**, contenente i metodi necessari alla gestione della cauzione. Ho implementato i suddetti metodi e li ho utilizzati all'interno

della classe di grafica BailView, richiamandoli su un oggetto di tipo BailLogic, al fine di scindere il *model* dalla *view*.



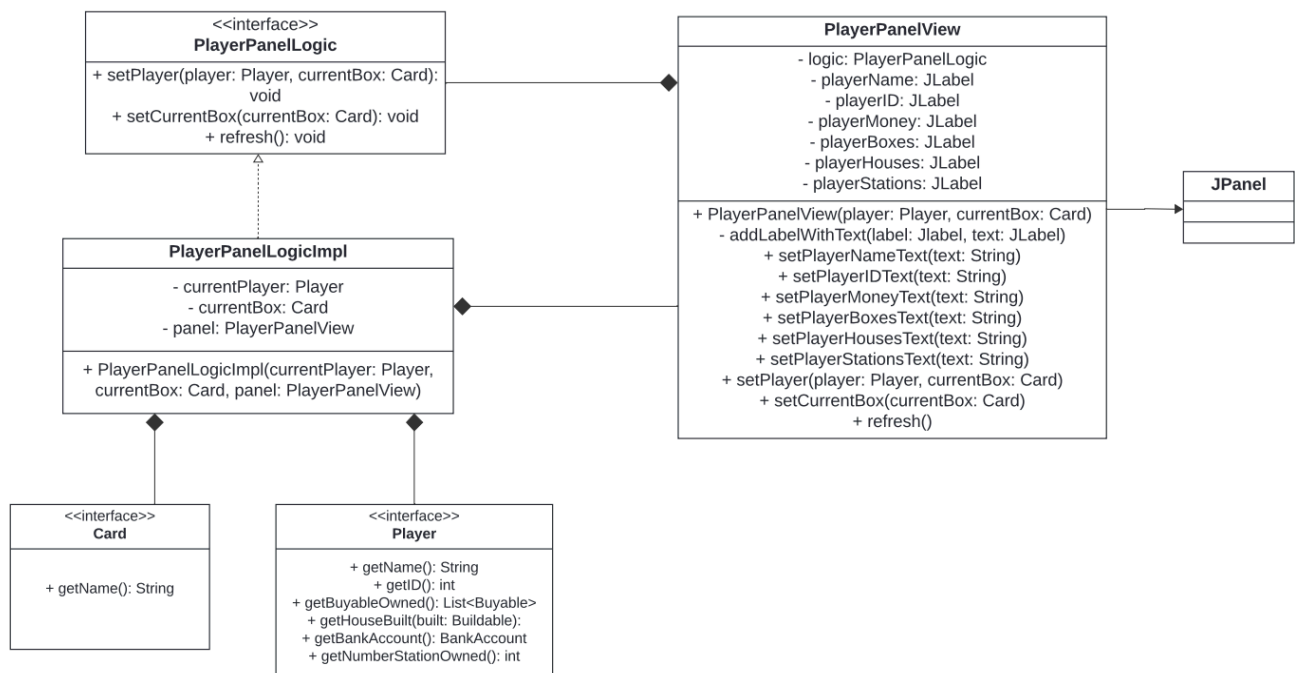
- Aggiornamento delle proprietà possedute dal giocatore in turno.

Problema: aggiornamento dei dati inerenti ai possedimenti di un Player durante il turno, con il rispettivo riscontro a livello grafico. Di conseguenza, la maggiore sfida è stata rispettare il pattern architetturale MVC.

Soluzione: l'obiettivo da perseguire è stato quello di fornire un'astrazione del comportamento della logica che governa la *view* del pannello "Player's report". Dunque ho sfruttato il **Dependency-Inversion Principle**: la classe **PlayerPanelView** dipende da **PlayerPanelLogic**, che è un'interfaccia e non una classe. Ciò consente alla classe **PlayerPanelView** di essere meno dipendente dall'implementazione specifica di **PlayerPanelLogic**.

Inoltre, la classe **PlayerPanelView** si occupa solo di visualizzare i dati del giocatore graficamente, mentre la logica relativa alla manipolazione dei dati è gestita da **PlayerPanelLogic**: questo *modus operandi* segue il **principio di Singola Responsabilità**, uno dei cinque principi SOLID della programmazione a oggetti. Esso afferma che *"afferma che ogni elemento di un programma deve avere una sola responsabilità, e che tale responsabilità debba essere interamente incapsulata dall'elemento stesso"*.

Un punto di debolezza della soluzione adottata consiste nell'aver creato una dipendenza tra *model* e *view*: infatti, il costruttore del pannello contenente i dati relativi al giocatore ha come parametro il giocatore che è attualmente di turno. D'altra parte, ho creato questa dipendenza proprio per la necessità di aggiornare il pannello grafico con le proprietà del Player di turno.



Nota di sviluppo: all'interno della classe `PlayerPanelLogicImpl`, mi sono trovata a sopprimere uno warning individuato da SpotBugs, con dicitura “*may expose internal representation by storing an externally mutable object*”. Infatti, ho optato per la soppressione perché i metodi con tale warning hanno come obiettivo l'aggiornamento dei valori relativi al giocatore. Se avessi ritornato una copia difensiva, allora tale aggiornamento non sarebbe avvenuto.

2.2.4 Riccardo Polazzi

Durante lo sviluppo del software la parte che è risultata più difficile è stata quella di far comunicare view e controller creando meno dipendenze possibili fra i due.

Problema:

Un giocatore può scegliere quali azioni eseguire direttamente premendo i bottoni sulla view ma, in base all'andamento del turno, questi bottoni dovranno essere abilitati o disabilitati. La logica gestisce infatti la posizione del player e in base alla casella su cui quest'ultimo finisce potrà fare solo determinate azioni; il controller dovrebbe quindi abilitare e disattivare i bottoni presenti nella view;

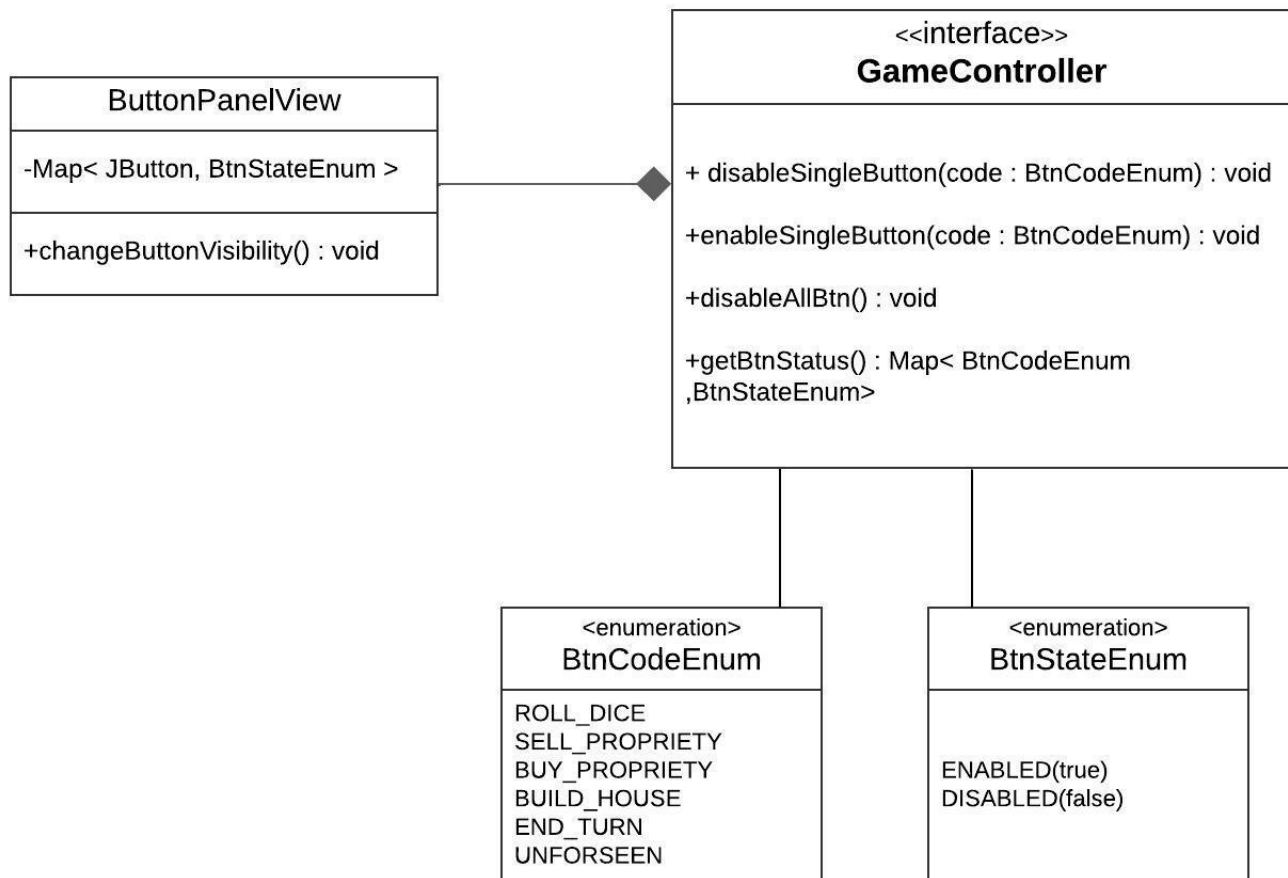
Soluzione:

Per non creare una dipendenza diretta tra view e controller non passo al controller i riferimenti diretti dei bottoni, ma ho istanziato una mappa all'interno di `GameControllerImpl` formata da due campi composti da due enumerazioni differenti.

Il `keySet` della mappa è composta da un campo `ButtonCodeEnum`, un' enumerazione che contiene codici che rappresentano ognuno un bottone, mentre il `valueSet` è composto da un campo `ButtonStateEnum`, un' enumerazione che rappresenta lo stato del bottone.

In questo modo quando il controller deve cambiare lo stato di uno specifico bottone cambia semplicemente il valore dello stato corrispondente ad un codice-bottone all' interno della sua mappa;

Sarà poi ButtonPanelView che si fa ritornare questa mappa e di conseguenza aggiornerà i suoi bottoni;



Problema:

Questo problema consiste nell' aggiornare tutta la restante parte di view, sempre in base all'andamento dei turni gestiti all'interno del controller.

Una volta lanciati i dadi per esempio la grafica di essi o la posizione del player dovranno venire aggiornate di conseguenza nella view, oppure in base a certe dinamiche di gioco dovranno comparire a schermo diversi pop-up che aiuteranno a capire a chi sta giocando cosa succede effettivamente al proprio player

Soluzione:

Per risolvere questo problema ho utilizzato il pattern observer.

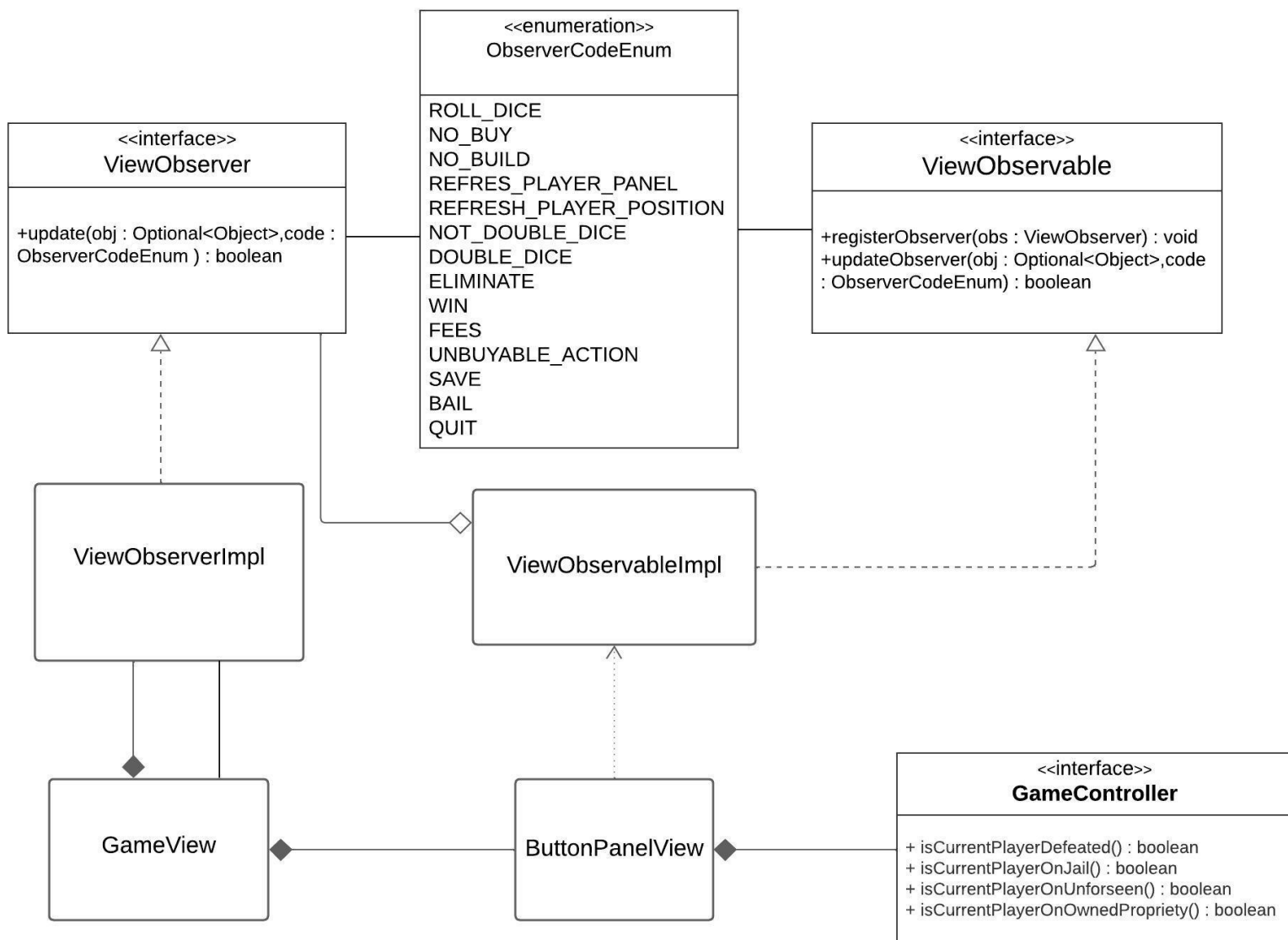
Ho iniziato istanziando un oggetto observer (ViewObserver) all' interno della classe GameView passandogli la GameView stessa come parametro. L' observer deve infatti avere un suo riferimento poiché utilizzerà i suoi metodi per aggiornare la grafica durante il gioco; Successivamente, la classe buttonPanelView funge da componente observable, essa infatti estende ViewObservable nella quale sarà registrato l'observer creato in precedenza.

In questo modo la buttonPanelView riceve gli output e l'andamento del turno dalla classe GameControllerImpl e in base a questi può inviare un segnale all' observer specificando quale parte di grafica deve essere aggiornata.

Per fare ciò, il metodo update della classe observer richiede un codice da una enumerazione chiamata ObserverCodeEnum in cui ogni codice è univoco per un cambiamento nella view, e che l'observer utilizzerà in uno switch.

In questo modo tramite un unico metodo update, può selezionare quale delle molteplici modifiche in base al codice ricevuto.

Oltre a un codice richiede anche un oggetto generico che varia in base alla parte di grafica che deve richiamare e serve a far comparire un certo valore nella view.



Nota di sviluppo : Ho deciso di sopprimere un warning nella creazione del ViewObserver perchè gli passo come campo la GameView (this), ho infatti bisogno del frame di gioco e non di una copia difensiva perchè devo chiamare dei metodi che andranno a modificare i pannelli presenti sul frame stesso.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Diverse parti del software sono state sottoposte a testing automatizzato, tramite la suite JUnit versione 5.10.1. Nel dettaglio, sono state realizzate classi di testing per i seguenti componenti:

- Meccanismi di versamento, pagamento e ricevimento di somme di denaro;
- acquisto e vendita di caselle, costruzione di case e conseguente conteggio delle proprietà del giocatore;
- inserimento dei giocatori;
- avvio della partita;
- uscita dalla partita;
- salvataggio dei dati dei giocatori;
- visualizzazione dei dati salvati dei giocatori;
- lancio dei dadi;
- pattern Adapter per adattare le carte;
- creazione delle carte tramite factory;
- observer con spostamento grafico dei giocatori;
- imprevisti.

Per quanto riguarda la logica del gioco verranno simulate varie situazioni di una partita vera e propria tramite la manipolazione dell'andamento del turno di un giocatore.

In questo modo verranno verificate varie situazioni di gioco quali:

- il corretto passaggio al turno del giocatore successivo;
- la corretta creazione di una lista di carte ordinata;
- il corretto lancio dei dadi e la nuova posizione del player;
- il modo corretto di richiamare le azioni delle caselle Unbuyable;
- la corretta attivazione e disattivazione dei bottoni in base alla casella su cui il player atterra;
- la corretta eliminazione dei player;
- la corretta terminazione del gioco.

3.2 Note di sviluppo

Di seguito vengono elencate (e successivamente mostrate) parti di codice ben realizzate che fanno uso di funzionalità avanzate del linguaggio Java.

3.2.1 Gianmaria Di Fronzo

- Progettazione con generici di un Observer pattern da utilizzare lato grafico

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/cbbf6f89d45e2cb4f6bbff49c40836be4b3659af/src/main/java/app/card/api/Observable.java#L8-L28>

- Uso di lambda expression attraverso interfacce funzionali per azioni di carte statiche

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/cbbf6f89d45e2cb4f6bbff49c40836be4b3659af/src/main/java/app/card/api/StaticActionStrategy.java#L10-L17>

- Uso di Optional per identificare proprietari di una carta

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/cbbf6f89d45e2cb4f6bbff49c40836be4b3659af/src/main/java/app/card/impl/BuyableImpl.java#L15>

- Uso della libreria JSON-JAVA per leggere file json che identificano le carte del tabellone di gioco

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/cbbf6f89d45e2cb4f6bbff49c40836be4b3659af/src/main/java/app/card/utils/JsonReader.java#L23-L41>

3.2.2 Giovanni Muccioli

- Utilizzo di stream e lambda expression

L'utilizzo di stream e lambda expression è stato utilizzato per manipolare prevalentemente collezioni di dati:

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/0d1cbe8ea1639cd98ffe6fb70c78adf8dac4ca31/src/main/java/app/game/controller/MenuControllerImpl.java#L35-L38>

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/0d1cbe8ea1639cd98ffe6fb70c78adf8dac4ca31/src/main/java/app/game/controller/SaveControllerImpl.java#L61-L64>

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/0d1cbe8ea1639cd98ffe6fb70c78adf8dac4ca31/src/main/java/app/game/controller/SaveControllerImpl.java#L82-L92>

- Utilizzo di Optional (codice)

L'utilizzo di Optional è servito ed è stato utilizzato per gestire la possibilità di valori nulli, in particolare quando si desidera leggere dei dati salvati che potrebbero non essere ancora stati ancora salvati per la prima volta.

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/0d1cbe8ea1639cd98ffe6fb70c78adf8dac4ca31/src/main/java/app/game/controller/SaveControllerImpl.java#L142-L149>

Precisazione per processo di uscita dal gioco e chiusura dell'applicazione.

Nel corso dello sviluppo dell'applicazione "Monopoly edizione Rimini", è emersa la necessità di implementare una soluzione per la chiusura dell'applicazione a seguito della pressione dell'apposito bottone.

Inizialmente è stata presa in considerazione l'uso della tradizionale istruzione "System.exit(0)", ma, anche al fine di evitare potenziali problemi di rilevamento errori da parte degli strumenti di analisi statica come SpotBugs, si è deciso di optare per una soluzione alternativa, decidendo quindi di non sopprimere l'errore.

Scoprendo quindi che l'utilizzo di "System.exit(0)" per uscire dall'applicazione viene considerata una pratica non ottimale, potendo causare effetti collaterali indesiderati, ho deciso di evitare tale istruzione e allo stesso tempo ho deciso di aderire alle migliori pratiche di programmazione, in modo da poter garantire una chiusura dell'applicazione più controllata.

Ho quindi optato per una soluzione alternativa che prevede l'utilizzo del metodo "dispose()" su tutte le finestre attive dell'applicazione, qualunque sia il tipo di finestra. Con tale approccio si consente di chiudere l'applicazione in modo pulito, coerente e di sicuro in modo meno drastico senza terminare bruscamente i processi in esecuzione.

Per questo ho deciso di *prendere spunto* dall'apposito forum presente su Stack Overflow (<https://stackoverflow.com/questions/258099/how-to-close-a-java-swing-application-from-the-code>) riadattando il codice al nostro contesto, in cui non si fa utilizzo di multithreading, quindi senza far uso di Timer e senza distinguere il tipo di finestra utilizzata.

3.2.3 Lisa Vandi

- Uso di stream e lambda expressions. Seguono esempi:

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/86803d27b693a431adf4bc8f8505a45d193553/src/main/java/app/player/impl/PlayerImpl.java#L180-L185>

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/86803d27b693a431adf4bc8f8505a45d193553/src/main/java/app/player/impl/PlayerImpl.java#L191-L196>

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/86803d27b693a431adf4bc8f8505a45d193553/src/main/java/app/player/impl/PlayerImpl.java#L212-L216>

3.2.4 Riccardo Polazzi

- Uso di Optional di un generico come campo di un metodo nelle interfacce ViewObserver e ViewObservable che di conseguenza verrà usato ogni volta che questo metodo viene richiamato nella classe BtnPanelView.

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/86803d27b693a431adf4bc8f8505a45d193553/src/main/java/app/game/api/ViewObserver.java#L21>

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/86803d27b693a431adf4bc8f8505a45d193553/src/main/java/app/game/api/ViewObservable.java#L30>

- Uso di stream per ordinare la lista di carte

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/86803d27b693a431adf4bc8f8505a45d193553/src/main/java/app/game/controller/GameControllerImpl.java#L77-L79>

<https://github.com/GiammaBigFishEngineer/OOP23-monopoly/blob/86803d27b693a431adf4bc8f8505a45d193553/src/main/java/app/game/view/GameView.java#L173-L177>

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Gianmaria Di Fronzo

Avendo avuto passate esperienze a livello lavorativo nello sviluppo software, l'approccio al metodo di lavoro mi era già familiare, spero però che questa sarà un vantaggio nella valutazione e che non porti a sbagliate procedure e quindi a negative valutazioni, nel contesto ho seguito comunque tutto il corso e cercato di rispettare tutte le linee guida fornite dal docente per il lavoro da svolgere.

Penso che la mia parte fosse una delle basi da cui partire per realizzare il gioco e penso di avere facilitato i miei colleghi nel riuso della mia parte. Se dovessi trovare imperfezioni nel mio sviluppo penserei ad uno scarso utilizzo di stream e di Generici, analizzando il problema non penso ci fossero problematiche che richiedessero l'implementazione di una di queste due cose ma comunque può darsi mi sbagli. Il resto del lavoro penso comunque sia ben fatto soprattutto sotto un punto di vista del design.

4.1.2 Giovanni Muccioli

La realizzazione di "Monopoly edizione Rimini" è stata la mia prima esperienza di sviluppo di un software completo.

Il lavoro in team è stato svolto in maniera collaborativa, aiutandosi e consultandosi sempre nelle (non poche) difficoltà che emergevano nello sviluppo. Nonostante la buona collaborazione e la buona unione del gruppo, è stato un lavoro che mi ha richiesto notevoli sforzi e una grande quantità di tempo per la realizzazione del lavoro finale. Con una maggiore quantità di tempo si sarebbe potuto implementare, per esempio, la funzionalità di ripresa di una partita interrotta, riguardante in primis la mia parte relativa al salvataggio dei dati dei

giocatori. Ovviamente le difficoltà sarebbero aumentate per implementare correttamente questa feature, ma la giudicavo come stuzzicante e intrigante, in quanto mi avrebbe permesso di esplorare e conoscere anche cose che non avevo mai visto durante il corso.

Inoltre, sempre con una maggiore quantità di tempo, mi sarebbe piaciuto poter esplorare altre librerie grafiche, magari più potenti e moderne (come JavaFX) rispetto all'utilizzata Swing.

Infine, nonostante a seguito della suddivisione dei task mi sia stata attribuita (di comune accordo!) una parte di lavoro non centrale e cruciale (come per esempio la gestione del main frame del gioco con la relativa logica per far avanzare i vari giocatori durante la partita), mi ritengo comunque utile alla realizzazione del software. Infatti, la parte riguardante il menù di gioco è la prima parte che permette di avviare il gioco correttamente, avendo quindi fin da subito diverse responsabilità "nelle mie mani".

Per questo, anche se non sono andato a toccare direttamente le dinamiche centrali e cruciali del gioco, ho cercato di dare comunque il mio aiuto e il mio contributo dall'inizio alla fine, sforzandomi di realizzare un buon codice per i task a me attribuiti.

Giudico come punto di forza la chiarezza del codice e la sua corretta organizzazione anche grazie alle diverse ore spese nella parte di analisi. Allo stesso tempo, reputo come punto di debolezza la scarsa complessità di codice scritto, ma pur sempre con lo scopo di favorire la chiarezza. Tant'è che in merito a quanto affermato, osservo e sono cosciente che avrei potuto utilizzare il pattern Observer per tracciare i cambiamenti/aggiornamenti nei giocatori utili per il salvataggio. Allo stesso modo sono anche cosciente che avrei potuto utilizzare un salvataggio dati basato su JSON, il quale avrebbe portato ad un risultato di certo più strutturato e compatibile, ma che d'altra parte mi avrebbe richiesto ulteriore tempo aggiuntivo, a causa della mia non conoscenza di tale linguaggio.

Sono comunque soddisfatto del lavoro svolto! Magari in futuro sarebbe interessante apportare i suggerimenti sopra citati, oltre a rendere le varie interfacce grafiche caratterizzate da animazioni, portando quindi il software ad avere una grafica ed un'estetica migliore.

La realizzazione di questo progetto mi ha di certo portato allo sviluppo di nuove competenze, tra cui il lavoro in team, l'utilizzo di un DVCS e una buona/avanzata conoscenza del linguaggio java.

4.1.3 Lisa Vandì:

Il mio ruolo all'interno del gruppo ha consistito nello sviluppo del Player, con annesso il suo BankAccount, il meccanismo di cauzione e la rappresentazione grafica del report dello stato del giocatore durante i turni.

Un significativo punto di forza del nostro gruppo è stato l'aver svolto in maniera dettagliata e, per quanto possibile, congiunta, la fase di analisi, il che ha facilitato la fase implementativa.

Al netto del progetto svolto, posso considerarmi soddisfatta di quanto fatto. Nello specifico, essendo stato questo il mio primo progetto nell'ambito della programmazione con una quantità di lavoro considerevole e portato avanti in gruppo, credo di aver gestito con abilità la distribuzione della mole di lavoro nel tempo. Inoltre, ritengo di essermi sempre mostrata

disponibile nel cercare di risolvere problematiche, mettendomi al servizio dei miei compagni quando necessario e ascoltando ciò che invece avrei potuto e dovuto migliorare. Valuto positivamente il mio lavoro, infatti, avendo il ruolo di fornire l'implementazione del Player, i miei compagni sono riusciti a sfruttare quanto fornito con scioltezza.

Sebbene tutti i componenti del gruppo siano stati aperti all'ascolto dei problemi altrui e delle rispettive soluzioni pensate, ciò in cui ho riscontrato maggiore difficoltà è stato proprio adattare se stessi agli altri, sia dal punto di vista logico che di tempistiche.

4.1.4 Riccardo Polazzi

Questa è stata la mia prima esperienza nello sviluppo di un software e, soprattutto all'inizio, ho incontrato diverse difficoltà che sono però riuscito a risolvere sia da solo che insieme ai compagni.

Una nota positiva di questo progetto è stata infatti secondo me il lavoro di squadra, infatti sono riuscito a comunicare e a scambiare consigli in modo ottimale con i miei compagni.

Credo di aver imparato molto da questo progetto, soprattutto per quanto riguarda l'uso di Git, di Gradle e di aver migliorato le mie abilità nel linguaggio java.

Ho inoltre compreso l'importanza di pianificare un'architettura solida prima di iniziare a scrivere effettivamente il codice. Questo mi ha infatti fatto risparmiare molto tempo una volta iniziato a scrivere le mie parti di software e sono sicuro che me ne farà risparmiare altrettanto in progetti futuri.

Appendice A

L'avvio del software è immediato e il suo funzionamento è guidato dall'interfaccia grafica. Le azioni che i giocatori possono svolgere durante la partita sono eseguibili mediante la pressione dei rispettivi bottoni presenti sulla grafica.

Grafica Menù:

The image shows a screenshot of a software menu for the game MONOPOLY. The word "MONOPOLY" is displayed in large, bold, red capital letters in the center of the screen. Below it, the text "'Edizione Rimini'" is shown in a smaller, bold, red font. At the bottom, there are three rectangular buttons stacked vertically, each with a thin border and bold black text. The buttons are labeled "Avvia Partita", "Visualizza Salvataggi", and "Esci" from top to bottom.

'Edizione Rimini'

Avvia Partita

Visualizza Salvataggi












Esci

Grafica di gioco :

PLAYER'S REPORT

Nome del giocatore:
ID del giocatore:
Monete sul conto bancario:
Case costruite su qualla casella corrente:
Numero di stazioni possedute:
Caselle possedute:

Giacatore3
3
335
Non possiedi questa casella
0
Borgo di Sangiuliano

GO 	Via Monaco  40\$	Imprevisti 	Stazione Sud 200\$	Viserba  100\$	Via IV Novembre 120\$	Transito 
Gran Hotel 5 stelle 300\$	<div>Message  Sei atterrato su una proprieta' di Giocatore1, devi pagare! OK</div>					Via caduti di Nassi... 140\$
Piazza Tre Martiri 230\$						Piazza Cavour
Stazione Est 200\$						Imprevisti 
Le Befane Shoppin... 170\$						Stazione Ovest 200\$
Corso d'Augusto 260\$						Teatro Galli 160\$
Vai in Prigione 	Via Balbla 220\$	Stazione Nord 200\$	Imprevisti 	Borgo di Sangiulia...  210\$	Via Tiberio 6 200\$	Parcheggio 

Lancia i Dadi!

Fine Turno

Compra Proprieta'

Pesca Imprevisto

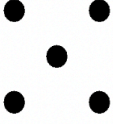
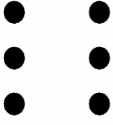
Vendi Proprieta'

Costruisci Casa

Salva Gioco

Esci Dal Gioco

Risultato: 11



Appendice B

B.0.1 gianmaria.difronzo@studio.unibo.it

- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p209964>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211304>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212807>