Relazione per progetto
"Monoopoly"

Alion Emini, Emanuele Fabricat, Eugenio Guidi, Federico Mantoni

17/03/2025

Sommario

Questa è la relazione riguardante il progetto di Programmazione ad Oggetti, del gruppo formato dagli studenti Alion Emini, Emanuele Fabricat, Eugenio Guidi, Federico Mantoni. La relazione è stata eseguita in LaTex e segue il template dato dal professore. Tutte le parti in comune sono state revisionate da tutti i membri del gruppo; invece, le parti singole sono di responsabilità del singolo.

Indice

Capitolo 1

1 Analisi

1.1 Requisiti

Il software vuole simulare il famoso gioco Monopoly in una sua versione semplificata. Si tratta di un gioco da tavolo dove i giocatori lanciano a turno due dadi, provocando lo spostamento sul tabellone del proprio segnalino secondo il risultato.
Il gioco è caratterizzato da diversi tipi di celle, alcune funzionali altre acquistabili; tramite l'acquisizione di queste ultime, sarà possibile percepire una rendita, da parte del proprietario, una volta che un giocatore diverso ci finisce sopra; la rendita può essere incrementata da parte del possessore in diversi modi, a seconda della proprietà.
Nel caso in cui un giocatore debba pagare una somma di denaro al di sopra della sua disponibilità economica, può utilizzare alcuni modi, tra cui l'ipoteca (la vendita di una proprietà in cambio di denaro), per sopperire alla mancanza di denaro.

Nel gioco vi è anche la possibilità di essere rinchiusi in prigione, condizione nella quale un giocatore non può capitare tramite il lancio di dadi, ma solo tramite la cella apposita, o tramite una carta, e dalla quale si può uscire pagando una cauzione o con apposita carta "esci gratis di prigione".
Il vincitore sarà l'ultimo giocatore rimasto in gioco una volta che tutti gli altri saranno andati in bancarotta, la quale si verifica nel caso in cui non possa in alcun altro modo effettuare un pagamento richiesto.

Le semplificazioni rispetto al gioco originale sono le seguenti:

- Un giocatore non può eseguire le seguenti operazioni se non dispone di abbastanza denaro liquido:
  - Comprare una proprietà
  - Costruire una casa
  - Disipotecare una proprietà
- La costruzione delle case è abilitata non appena si ottiene una proprietà in cui è possibile costruire
- Non sono presenti alberghi, ma 5 case
- Non sono permessi scambi
- La meccanica che si basa sul confronto delle facce dei dadi per verificare se mostrano lo stesso numero non attiva nessun effetto.
- Nel momento del pagamento di rendita da parte di un giocatore, la banca paga direttamente il beneficiario, poi sarà il debitore a dover riuscire a pagare il debito con la banca
- Quando un giocatore andrà in bancarotta per qualsiasi motivo, le sue proprietà saranno disipotecate e restituite alla banca, per poter essere accessibili ai giocatori rimanenti.

Requisiti funzionali
Il gioco gestisce:

- Lo spostamento del giocatore e la conseguente azione da intraprendere, a seconda della cella in cui termina il suo cammino
- Il meccanismo delle carte imprevisti e i relativi effetti
  (riscossione, pagamento, spostamento della pedina, uscire di prigione gratis...)
- Il game loop, che si compone di:

  - Spostamento del giocatore
  - Risoluzione degli eventi
  - Possibile costruzione di case di proprietà possedute e/o disipoteca
- Il menù iniziale del gioco dove si indicherà il nome e il numero dei giocatori
- La fine del gioco e l'uscita di un giocatore dalla partita una volta in bancarotta
- Il tabellone e il suo aggiornamento

Requisiti non funzionali

- Il software deve essere portabile e quindi eseguibile in più sistemi operativi (Windows, Linux, Mac).
- L'interfaccia sarà realizzata utilizzando tecnologie semplici ma efficaci, come un layout grafico minimale con pulsanti e notifiche testuali.
- Il gioco deve risultare fluido nella sequenza di azioni che vanno a susseguirsi durante il corso del gioco.

1.2 Analisi del modello del Dominio

L'applicazione utilizzerà una GameBoard come stato di fatto del gioco, contenente le celle, i giocatori, il mazzo degli imprevisti, i dadi e il giocatore che sta eseguendo il turno. Ogni turno è composto da diversi sotto-stadi:

1. Prigione: In cui si entra e si esce di prigione.
2. Movimento: In cui si viene spostati a causa di una carta, della casella "vai in prigione" o tramite un lancio dei dadi.
3. Controllo dell'azione da intraprendere: Qui si dovrà gestire cosa accade quando si capita su una determinata cella, e quale stadio susseguirà.
4. Pagamento: In cui si esegue un pagamento, qui dovrà essere gestita la vendita delle case, l'ipoteca in caso di liquidità insufficiente e la bancarotta, nel caso le due azioni precedenti non bastino.
5. Disipoteca e costruzione case: In cui si potrà disipotecare proprietà e costruire case, se ne si ha la possibilità
   (liquidità, possedimento di proprietà ipotecate)

Le celle saranno suddivise nelle varie tipologie: acquistabili, suddivise a loro volta in edificabili o meno, e funzionali. Nel caso in cui solo un giocatore rimarrà in gioco, verrà dichiarato vincitore.

Gli elementi sopra elencati sono sintetizzati in Figura 1.1

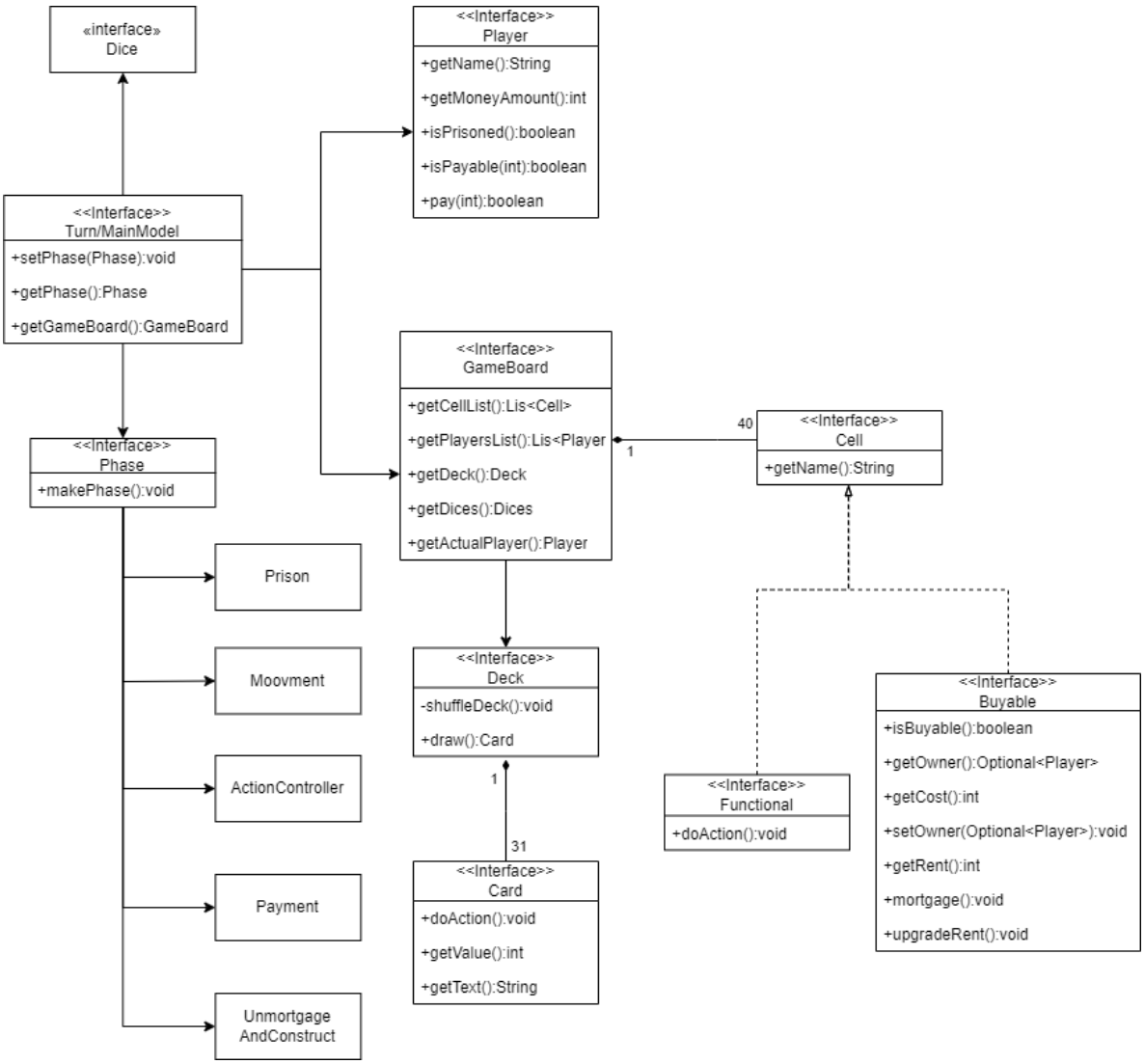

Figura 1.1: UML Model

Capitolo 2

2 Design

2.1 Architettura

Monoopoly utilizza il pattern architetturale MVP (Model-View-Presenter), caratterizzato dal completo distacco tra Model e View per il passaggio dei dati. In particolare Monoopoly utilizza due pattern:

1. Pattern State in tutti e tre gli stati del MVP.
2. Pattern Builder per la pacchettizzazione e il passaggio dei dati.

Il primo viene utilizzato per dividere le varie fasi del gioco (movimento, costruzione case, pagamento, ecc.) ed incapsularne metodi, campi e logica. In particolare, eseguendo lo stesso stato contemporaneamente su tutti e tre gli stadi del MVP, si può facilmente lavorare solo sulla parte di codice che ci interessa senza intaccare il resto. Quindi, malgrado la struttura si irrigidisca, diviene più facile la manutenzione e l'ampliamento del progetto, cambiando uno Stato già esistente o aggiungendone uno nuovo con limitate modifiche per collegarlo ai successivi e precedenti.
In particolare, utilizzare il pattern su tutti e tre i componenti è stato possibile rispettare al meglio il tipo MVP:

- Model
  In cui è stata incapsulata tutta la logica, separandolo completamente dalla View.
- Presenter
  Che ha un ruolo di scambio di informazioni e gestione dei rapporti tra View e Model, estraendo e modellando le informazioni in maniera autonoma dal Model.
- View
  Completamente passiva, una volta che il Presenter le passa le informazioni già formattate, questa mostra gli insiemi di pannelli in base ai dati ricevuti, e restituisce il controllo al Presenter, fornendogli le possibili scelte del giocatore.

Il secondo viene utilizzato nel Presenter per creare un Record, con le informazioni essenziali già elaborate per la View, così che non ci sia dialogo tra Model e View. Viene inoltre utilizzato nella View, per passare i dati selezionati dal giocatore.
In figura 2.1 è semplificato il diagramma UML architetturale

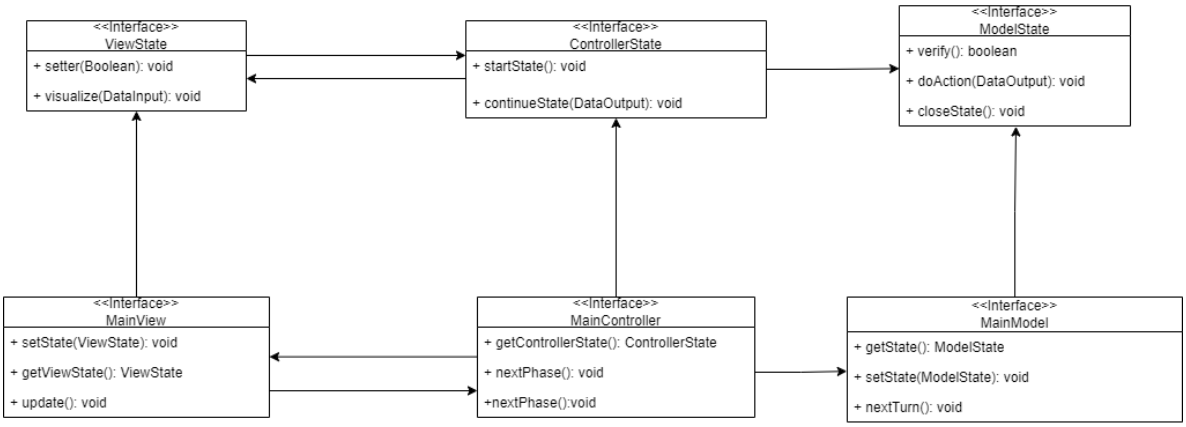

Figura 2.1: UML MVP

2.2 Design dettagliato

2.2.1 Alion Emini

Gestione del giocatore e delle proprietà

Problema Nel gioco, il giocatore può acquistare e vendere proprietà, ma è importante evitare che possa farlo in modo scorretto, ad esempio costruire case senza pagare o rimuovere proprietà senza motivo. Questo potrebbe rompere l'equilibrio del gioco.

Soluzione Per risolvere questo problema, abbiamo creato un'interfaccia Player che incapsula tutte le operazioni legate alle proprietà. Questo significa che il giocatore non può modificare direttamente lo stato delle proprietà, ma deve passare attraverso metodi specifici come addProperty() o removeProperty(). Questi metodi includono controlli interni: ad esempio, il giocatore può costruire una casa solo se possiede tutte le proprietà dello stesso gruppo e ha abbastanza soldi. Inoltre, le proprietà sono gestite tramite un Set di Buyable, che garantisce che non ci siano duplicati e che ogni proprietà sia unica.

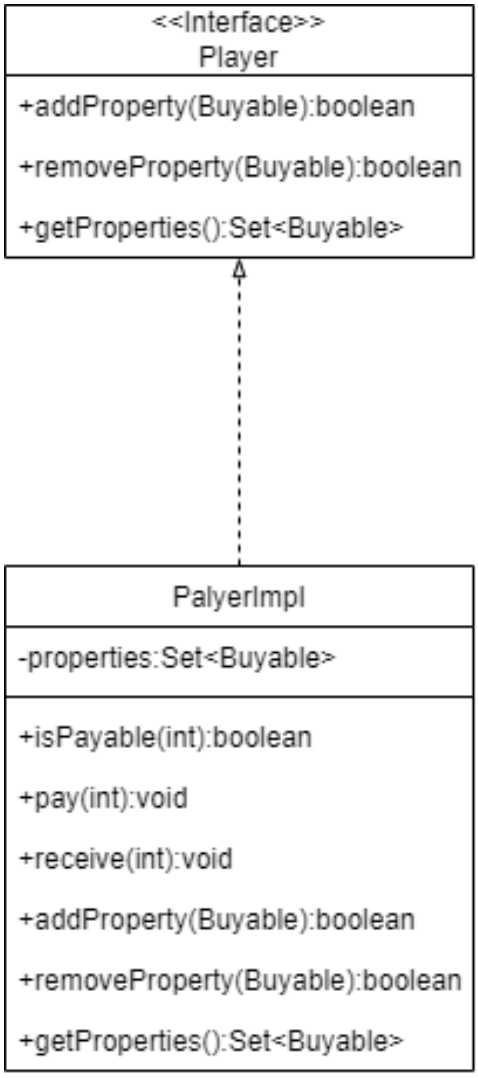

Figura 2.2: UML risoluzione problema: gestione del giocatore e delle proprietà

Uscire dalla prigione

Problema Quando un giocatore finisce in prigione, deve avere opzioni chiare per uscire: pagare una multa, usare una carta "Esci di prigione" o attendere. Senza una gestione precisa, il giocatore potrebbe rimanere bloccato in prigione senza sapere come procedere, creando frustrazione.

Soluzione Per gestire questa situazione, abbiamo introdotto uno stato specifico chiamato ModelPrisonState. Questo stato si occupa di verificare se il giocatore deve andare in prigione o se può uscire. Se il giocatore è già in prigione, lo stato gestisce le opzioni disponibili: pagare una multa di 50€, usare una carta "Esci di prigione" (se ne possiede una), o attendere il turno successivo. La vista (ViewPrisonState) mostra al giocatore un messaggio chiaro con le opzioni disponibili, rendendo l'esperienza di gioco più intuitiva. In questo modo, il giocatore sa sempre cosa fare e non rimane bloccato in uno stato di incertezza.

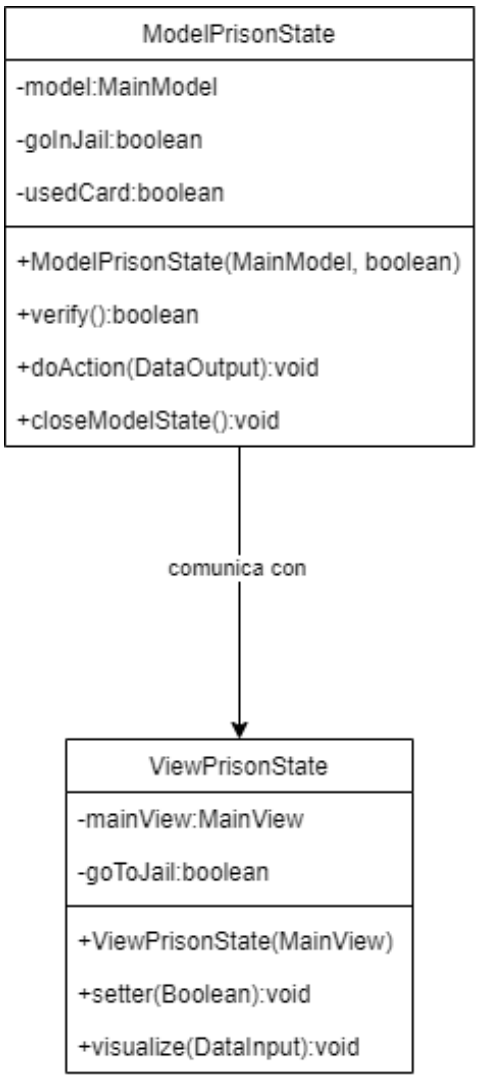

Figura 2.3: UML risoluzione problema: uscire dalla prigione

Costruire case sulle proprietà

Problema Il giocatore può costruire case sulle proprietà, ma deve farlo rispettando le regole: deve possedere tutte le proprietà dello stesso gruppo e avere abbastanza soldi. Senza controlli, potrebbe costruire case in modo illegittimo, rompendo l'equilibrio del gioco.

Soluzione Per gestire la costruzione di case, abbiamo creato uno stato dedicato chiamato ModelBuildHouseState. Questo stato verifica se il giocatore può costruire case su una proprietà, controllando che:

1. Il giocatore possieda tutte le proprietà dello stesso gruppo.
2. Abbia abbastanza soldi per pagare il costo della casa.
3. La proprietà non abbia già il numero massimo di case (5).

Se tutte le condizioni sono soddisfatte, il giocatore può costruire una casa. La vista (ViewBuildHouseState) mostra al giocatore solo le proprietà su cui può costruire, evitando confusione e garantendo che le regole del gioco siano rispettate. Inoltre, il controller (ControllerBuildHouseState) coordina l'interazione tra il modello e la vista, assicurando che tutto funzioni in modo fluido.

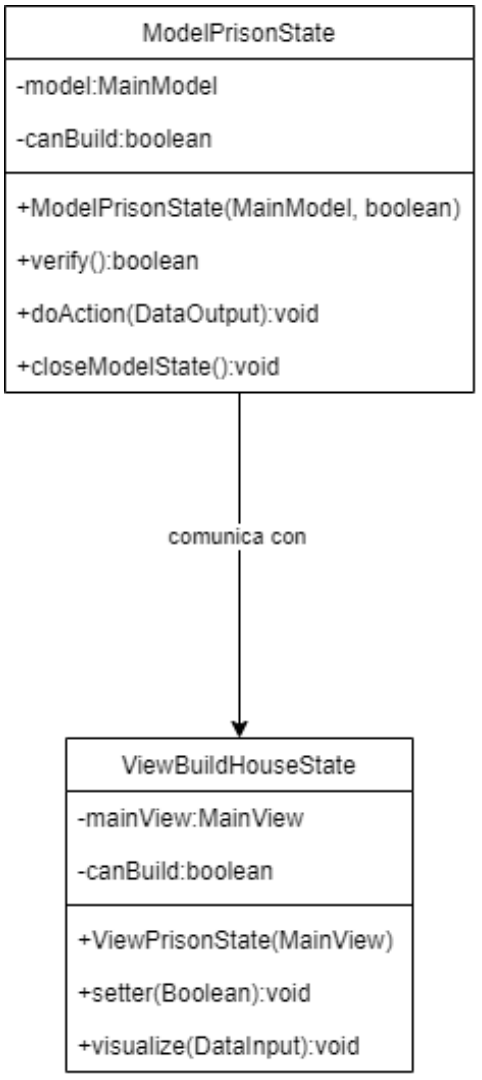

Figura 2.4: UML risoluzione problema: costruire case sulle proprietà

2.2.2 Emanuele Fabricat

Passaggio informazioni Model-Presenter-View

Problema Passare informazioni tra Model, Presenter e View, in modo che fossero il minimo di informazioni possibili, e soprattutto evitando di passare parti grosse di ogni parte.

Soluzione Utilizzare due pattern Builder, così che il Presenter possa utilizzare un DataBuilderInput, per creare una classe DataInput, ovvero un pacchetto di campi Optional che saranno inizializzati a seconda delle esigenze dei vari State. In modo simile si comporta la View, utilizzando un DataBuilderOutput. Una prima implementazione del passaggio delle informazioni, è stata fatta tramite metodi dei vari State con l'utilizzo dei generici, ma complicava molto il codice e le logiche, a causa dei diversi tipi di ritorno a seconda dello specifico Stato, e in più veniva violata la regola per la quale il Model non debba passare dati al Presenter, soprattutto già "formattati"

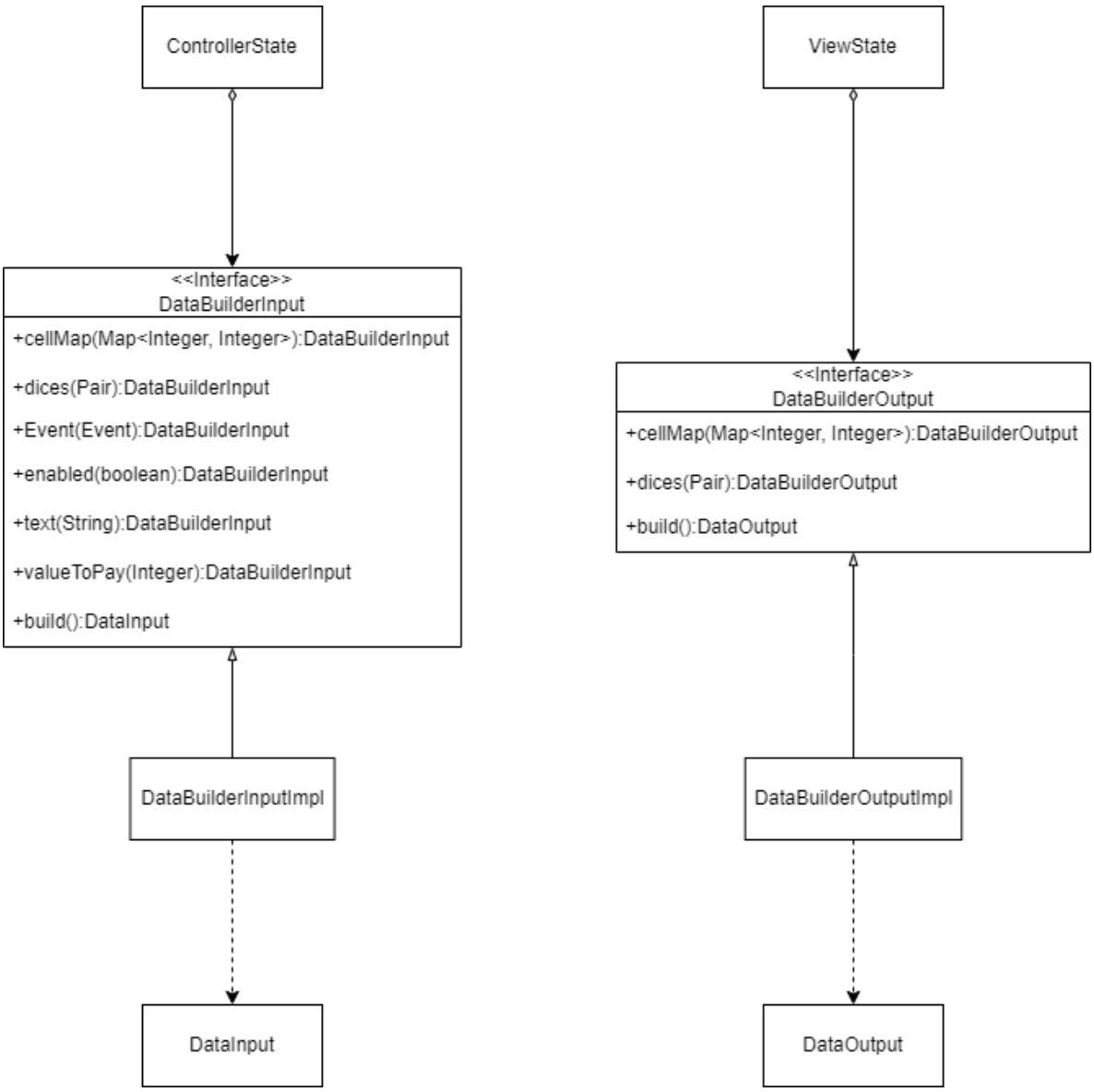

Figura 2.5: UML risoluzione problema: passaggio informazioni Model-Presenter-View

Passaggio azione delle carte

Problema Trasmettere sinteticamente il comportamento logico di ciascuna carta alla parte del modello, ovvero allo State relativo, che dovrà gestirne l'esecuzione.

Soluzione Un record contenente un Enum, o meglio un Event, classe creata e implementata dal collega Mantoni, per il tipo di azione da eseguire e un Optional di Integer per i dati da utilizzare. Grazie a questa soluzione, il metodo closeState del ModelCardState può impostare il corretto State successivo con i relativi dati e precederlo con parti di codice funzionali alla logica, come far pagare una tassa al giocatore e poi impostare il ModelBankerState.

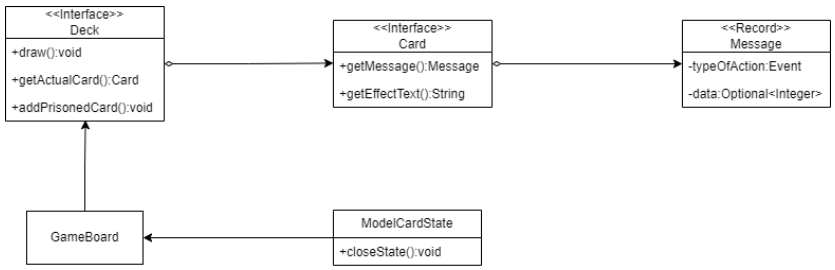

Figura 2.6: UML risoluzione problema:

Passaggio sicuro del Deck

Problema Il passaggio del Deck all'interno del costruttore del ControllerDeckState esponeva l'interno della classe DeckImpl, ma non era possibile passare solo la carta, dato che questa viene pescata all'interno dello ControllerState

Soluzione Utilizzare il Pattern Proxy per creare un wrapper per il Deck, in cui solo il metodo getActualCard è implementato in modo che ritorni una copia della carta, e i restanti metodi lanciano un'eccezione.

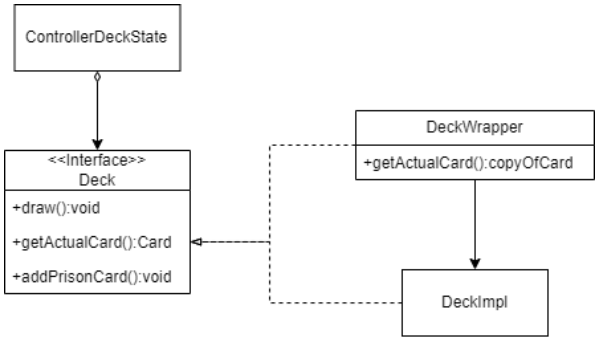

Figura 2.7: UML risoluzione problema: passaggio sicuro del Deck

2.2.3 Eugenio Guidi

Prevenzione dello Stack Overflow

Problema Il problema si è verificato a causa di una gestione inefficace del flusso di esecuzione del gioco, in cui le chiamate di metodi si susseguivano senza che ogni metodo terminasse prima che fosse completato quello richiamato. Questo comportamento causava un accumulo eccessivo di chiamate sullo stack, portando infine a un stack overflow. Durante il turno di gioco, quando si lanciavano i dadi, il gioco simulava il lancio mostrando un JDialog che bloccava temporaneamente l'esecuzione. Il pulsante "Lancia dadi" chiudeva il JDialog, ma senza svuotare effettivamente lo stack, creando un possibile accumulo di chiamate.

Soluzione Per risolvere ciò, è stata introdotta una modifica che consente all'utente di fare una scelta (lanciare i dadi). La modifica consiste nel sostituire il JDialog con un semplice pulsante "Lancia dadi", il quale è associato a un action listener. In questo modo, l'utente può proseguire con il gioco senza interruzioni, e, a differenza della soluzione precedente, lo stack viene svuotato correttamente almeno una volta per turno, evitando il rischio di stack overflow.

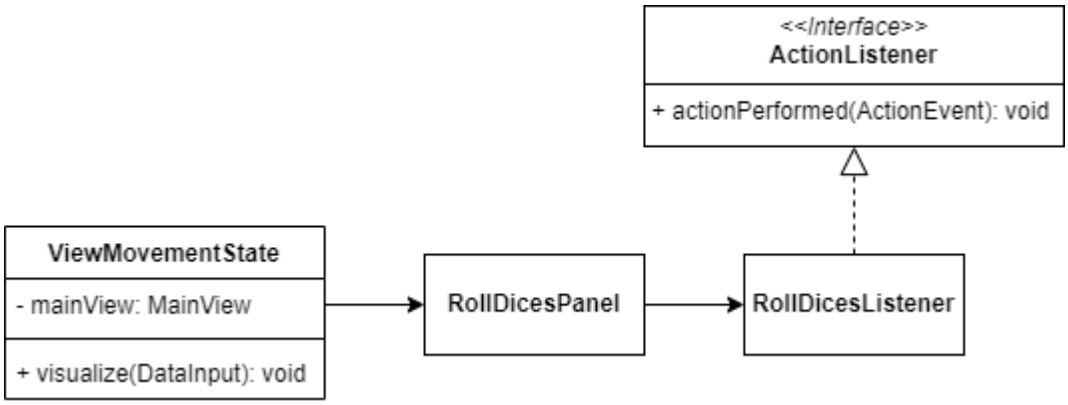

Figura 2.8: UML risoluzione problema: prevenzione dello Stack Overflow

Unificazione delle Posizioni tramite il Pattern Builder

Problema Il problema si verificava nella classe PositionAllocatorImpl, dove era necessario creare due oggetti separati per rappresentare i pallini e i numeri sulla game board. Venivano così create due liste distinte, una per ogni tipo di oggetto (pallini e numeri), che venivano poi restituite alla game board. Quest'ultima, attraverso due iterazioni separate, si occupava di disegnarli sulla board. Tuttavia, questa struttura risultava poco compatta e inefficiente.

Soluzione Per risolvere questo problema si è utilizzato il pattern Builder per la creazione di un unico elemento per rappresentare entrambi gli oggetti (pallini e numeri), sfruttando la classe NumberAndCirclePosition, al cui interno è presente la classe statica Builder, quest'ultima sfruttando una serie di chiamate di metodo che ritornano lo stesso builder permettono la creazione chiara dell'oggetto.

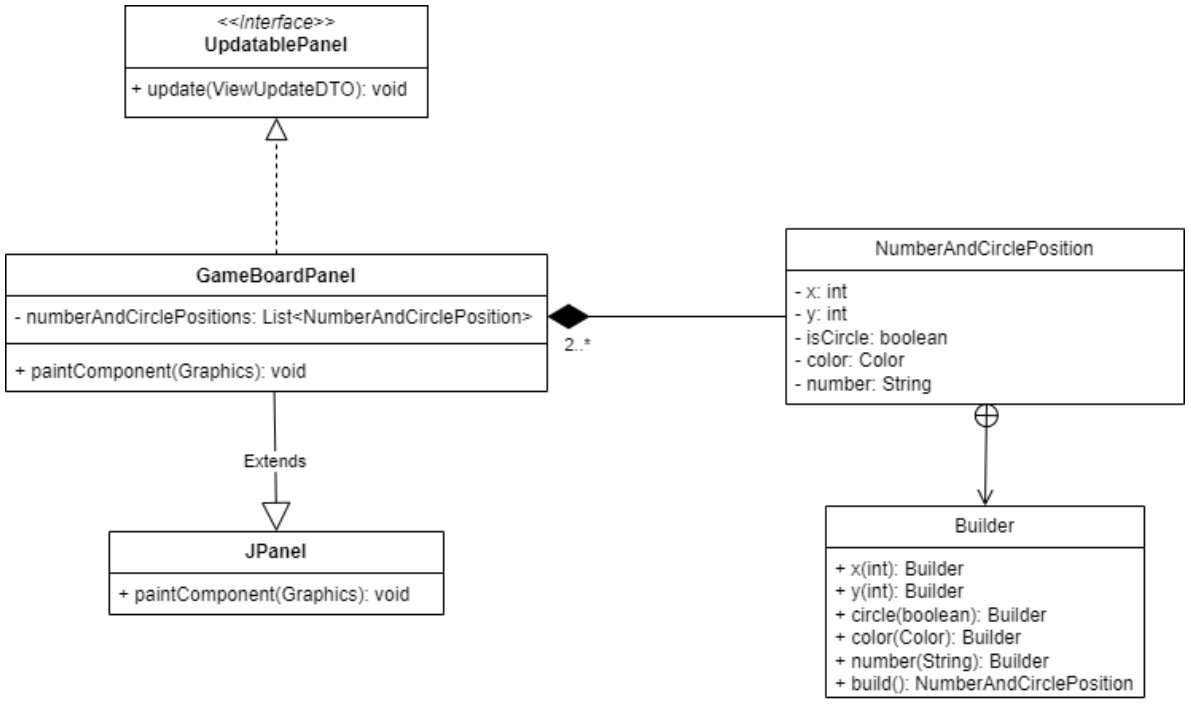

Figura 2.9: UML risoluzione problema: unificazione delle Posizioni tramite il Pattern Builder

Ottimizzazione della Gestione delle Posizioni sulla game board

Problema Il problema si verificava all'interno della classe GameBoardPanel, che aveva il compito di inizializzare i dati relativi alle possibili posizioni di pallini e numeri sulla game board. Questo veniva fatto tramite un metodo di aggiornamento che, una volta avviato, creava due liste separate di oggetti. Successivamente, nel metodo paintComponent, queste due liste venivano utilizzate per disegnare gli oggetti sulla board. Tuttavia, questa struttura caricava troppo lavoro sulla classe, violando così il principio DRY (Don't Repeat Yourself).

Soluzione La soluzione è stata quella di suddividere la classe in tre classi separate aggiungendo: PositionAllocatorImpl e PositionsFactoryImpl. PositionsFactoryImpl, utilizzando il pattern Factory, si occupa di inizializzare i dati relativi alle posizioni dei cerchi e dei numeri, estraendo le informazioni dai file JSON. PositionAllocatorImpl, invece, sfrutta la Factory di PositionsFactoryImpl per generare le posizioni effettive che devono essere visualizzate durante il gioco. Con questa nuova architettura, la GameBoardPanel si limita a ricevere i dati da PositionAllocatorImpl e a disegnarli sulla game board.

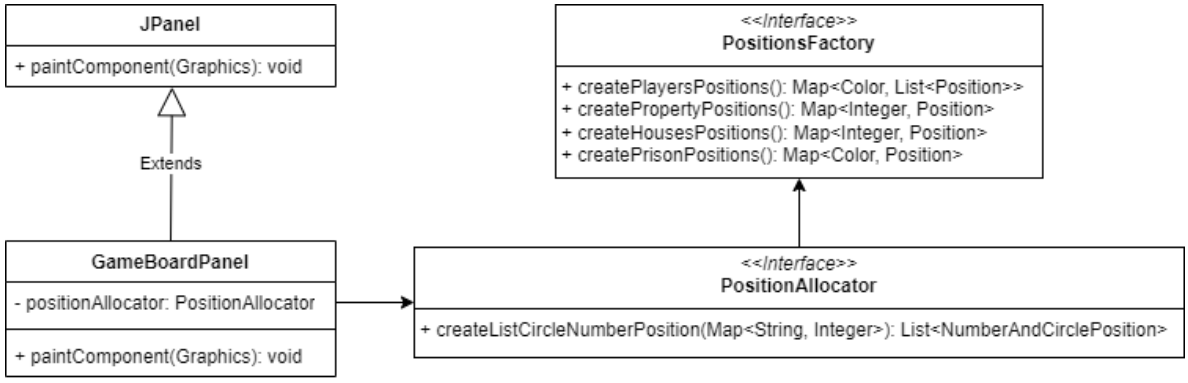

Figura 2.10: UML risoluzione problema: ottimizzazione della Gestione delle Posizioni sulla game board

2.2.4 Federico Mantoni

Creazione delle caselle del tabellone

Problema Creare le istanze delle 40 caselle del tabellone. Ogni casella è istanza di una classe diversa (Buildable, Functional, Company ...) e anche caselle dello stesso tipo differiscono tra loro. Si vuole riuscire a creare tutte le celle in modo semplice e mantenendo un codice pulito, privo di variabili che servono solo all'inizializzazione delle caselle.

Soluzione Sono stati salvati i dati di ogni cella in un unico file Json. É stata utilizzata la libreria Jackson per mappare correttamente ogni cella alla sua classe specifica, utilizzando i tag adatti forniti dalla libreria. La classe JsonConverter si occupa della deserializzazione data in input una classe, così che possa essere riutilizzata in futuro per la creazione di altri elementi utili per l'applicazione. É stato usato il pattern Simple Factory per la creazione delle celle: la classe CellFactory va a chiamare correttamente JsonConverter e restituisce una lista di Cell che verrà poi gestita dal MainModel

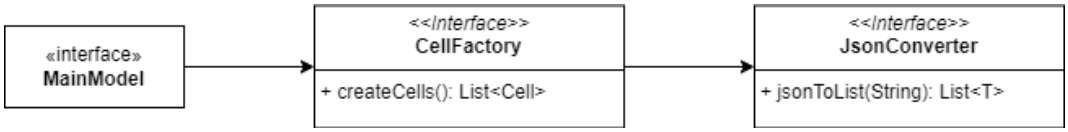

Figura 2.11: UML risoluzione problema: Creazione delle caselle del tabellone

Controllo dell'azione da eseguire

Problema A seconda della cella in cui un giocatore finisce il suo spostamento, possono accadere diversi eventi. L'evento deve essere riconosciuto e portato a termine correttamente.

Soluzione É stato creato un enum Event dove ogni valore corrisponde ad un evento che dovrà essere portato a termine. L'enum viene impostato nel MainModel dal ModelCheckActionState. L'Event è impostato a seconda della casella in cui si trova il giocatore di turno. L'evento verrà interpretato dal ControllerCheckActionState per comandare la View che cosa visualizzare e dallo state del model per passare alla prossima fase.

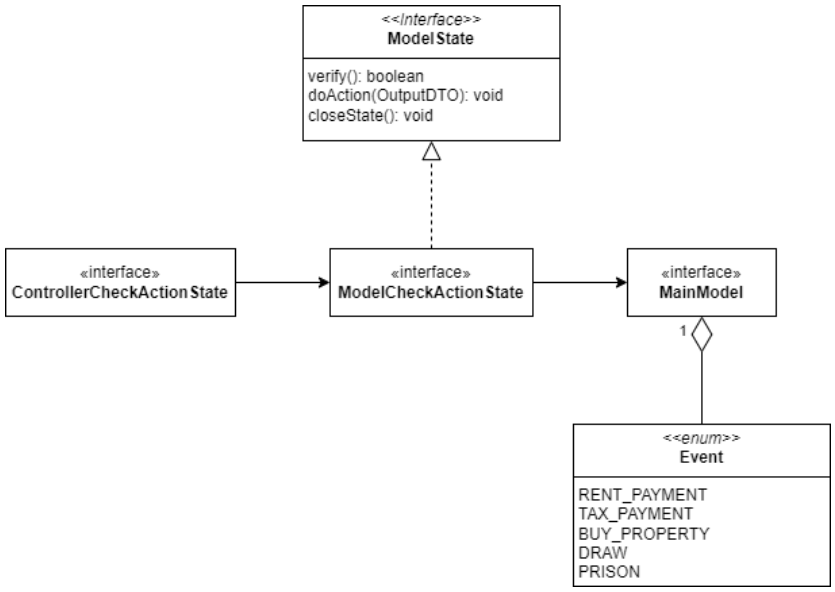

Figura 2.12: UML risoluzione problema: Controllo dell'azione da eseguire

Passaggio di dati incapsulati alla View

Problema Alla fine di ogni fase, la view deve essere aggiornata, passando dei dati in modo che gli elementi del Model non siano esposti.

Soluzione Dovendo passare un numero elevato di parametri, prendendo ispirazione dal record usato per il passaggio di dati in architettura, è stato creato utilizzando un Record, un DTO (DataTransferObject) che contiene sotto forma di stringhe ed interi tutti i dati necessari per l'aggiornamento della View.

Così facendo non è il controller a passare i dati alla View, e la logica di update può essere facilmente cambiata, e si va ad evitare la creazione di più metodi getter separati nel controller.

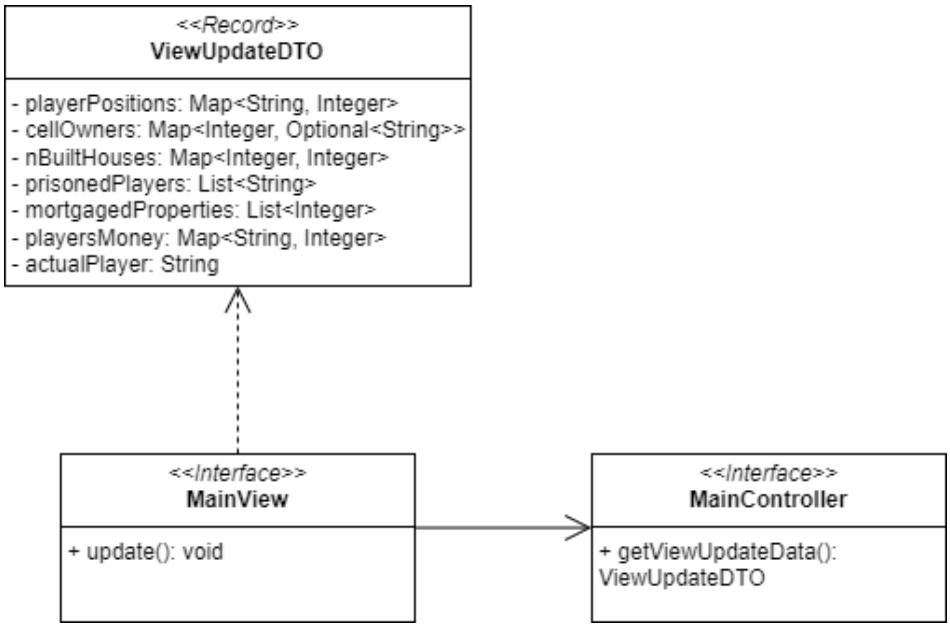

Figura 2.13: UML risoluzione problema:Passaggio di dati incapsulati alla View

Passaggio al Controller in modo sicuro e incapsulato di un elemento di gioco

Problema Il passaggio della classe Cell al Controller espone un elemento del Model a possibili cambiamenti indesiderati.

Soluzione Attraverso il pattern Proxy è stata creata la classe CellWrapper che implementa le interfacce Functional e Company così da avere tutti i metodi di una possibile cella implementati. CellWrapper implementa solo i metodi getter così da garantire la correttezza del pattern MVP

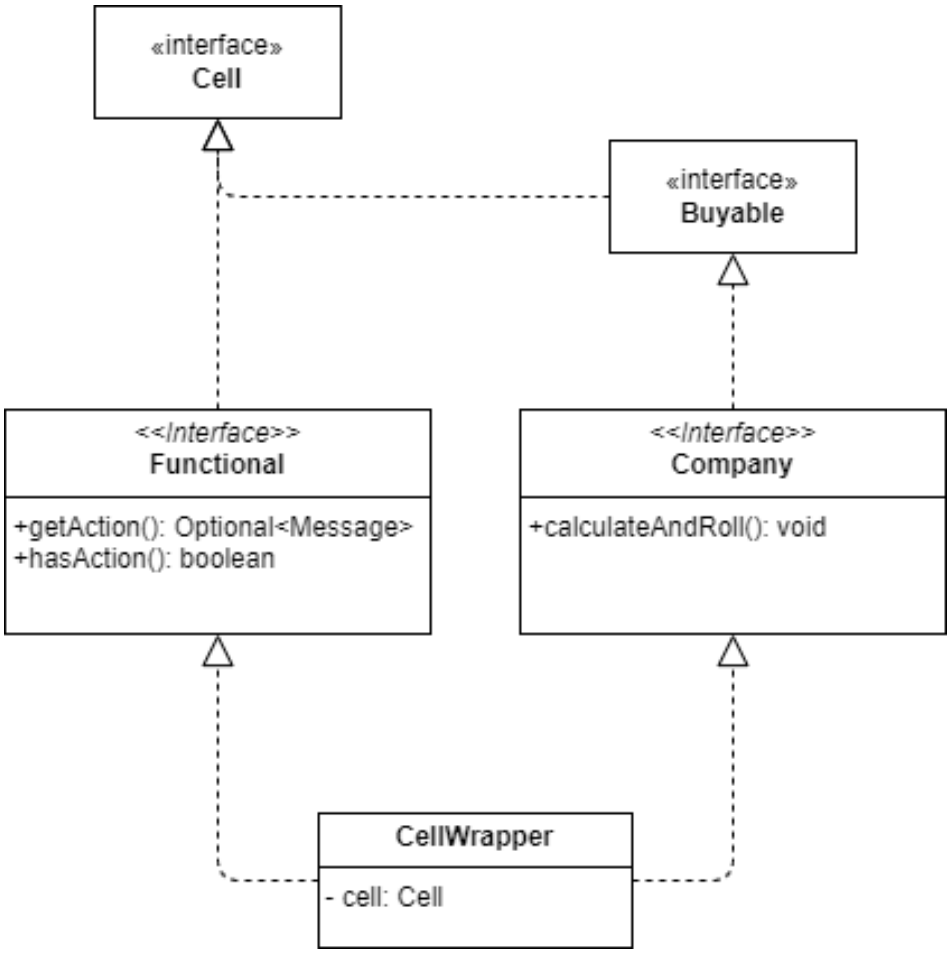

Figura 2.14: UML risoluzione problema: Passaggio al Controller in modo sicuro e incapsulato di un elemento di gioco

Capitolo 3

3 Sviluppo

3.1 Testing automatizzato

Sono stati testati in modo automatico tramite JUnit, tutti gli elementi principali del Model:

- Tutti gli state del Model
- Tutti i tipi di celle
- La game board
- I dadi
- Il deck e le carte
- Il notaio
- Il banker
- La factory delle celle
- Il convertitore dei Json

3.2 Note di sviluppo

3.2.1 Emanuele Fabricat

Utilizzo org.apache.commons.lang3.tuple.Triple
Permalink: https://github.com/ElManto03/OOP-monoopoly/blob/a9a3de1446b608a183bbdd790f6d9c345a5ea5cb/src/main/java/it/unibo/monoopoly/view/main/impl/MainViewImpl.java#L92-L98

Utilizzo Stream
Svariati utilizzi, ecco un esempio: https://github.com/ElManto03/OOP-monoopoly/blob/a9a3de1446b608a183bbdd790src/main/java/it/unibo/monoopoly/controller/state/impl/ControllerBankerState.java#L125-L129

Utilizzo Optional
Svariati utilizzi, ecco un esempio: https://github.com/ElManto03/OOP-monoopoly/blob/e39256a8ee9c7bf071b940b11src/main/java/it/unibo/monoopoly/controller/data/impl/DataInput.java#L28C1-L33C48

3.2.2 Eugenio Guidi

Utilizzo della libreria org.apache.commons.lang3.tuple.Pair
Permalink: https://github.com/ElManto03/OOP-monoopoly/blob/5623c791a08d03ed536ce53eafaee2c63ef87184/src/main/java/it/unibo/monoopoly/model/gameboard/impl/DicesImpl.java#L32C5-L46C6

Utilizzo di Stream e lambda expression
Usate molteplicemente come da esempio seguente.
Permalink: https://github.com/ElManto03/OOP-monoopoly/blob/5623c791a08d03ed536ce53eafaee2c63ef87184/src/main/java/it/unibo/monoopoly/view/position/impl/PositionAllocatorImpl.java#L89C5-L99C6

3.2.3 Federico Mantoni

Utilizzo di stream e lambda functions
Utilizzato in varie parti. Un esempio: https://github.com/ElManto03/OOP-monoopoly/blob/f0a60a6b49885d5e20fa0f
src/main/java/it/unibo/monoopoly/controller/main/impl/MainControllerImpl.java#L143C5-L160C6

Utilizzo di Optional
Permalink: https://github.com/ElManto03/OOP-monoopoly/blob/0f62fe38d27bc1d7d2ba3384cf3b37e004b9b715/
src/main/java/it/unibo/monoopoly/model/gameboard/impl/FunctionalImpl.java#L30

Utilizzo di Jackson
Permalink: https://github.com/ElManto03/OOP-monoopoly/blob/a9a3de1446b608a183bbdd790f6d9c345a5ea5cb/
src/main/java/it/unibo/monoopoly/utils/impl/JsonConverterImpl.java#L39-L49
Uso dei tag Jackson: https://github.com/ElManto03/OOP-monoopoly/blob/0f62fe38d27bc1d7d2ba3384cf3b37e004b9
src/main/java/it/unibo/monoopoly/model/gameboard/api/Cell.java#L16C1-L22C3

Capitolo 4

4 Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Alion Emini

Una delle principali difficoltà incontrate è stata la gestione del tempo e delle scadenze. La necessità di coordinare il lavoro con gli altri membri del gruppo e di rispettare le linee guida ha richiesto un grande sforzo di pianificazione. Dal punto di vista del corso, ho trovato molto utili le linee guida sulla progettazione software e l'uso degli strumenti di versionamento come GitHub. Ritengo che questo progetto sia stato un'ottima esperienza di apprendimento, che mi ha permesso di migliorare sia le mie competenze tecniche sia quelle di collaborazione in team.

4.1.2 Emanuele Fabricat

Ritengo di aver svolto un buon lavoro per quanto riguarda le implementazioni della mia parte; è probabile che ci siano delle criticità in alcune parti del codice dovute alla fretta di concludere il progetto. Per esempio, non sono sicuro che il metodo utilizzato per evitare l'esposizione del Deck forse non sia il più ottimale, dato che si è dovuto comunque utilizzare più di un SuppressWarning; un altro punto critico della mia parte è sicuramente Message, che a parer mio poteva, con tempo a disposizione, essere del tutto modellato in un pattern Command, quando attualmente è semplicemente una soluzione che viene utilizzata per eseguire la stessa cosa, ovvero passare delle informazioni a una classe che poi dovrà eseguire l'azione dedotta dai dati passati.

Per quanto riguarda il mio ruolo all'interno del gruppo, insieme ai compagni Eugenio Guidi e Federico Mantoni abbiamo pensato e implementato la maggior parte dell'applicazione, a mio parere nel trio non c'è stato un elemento di spicco, ma un buon equilibrio dato da iterazioni che hanno portato a risolvere in maniera buona e, in alcuni casi, ottimale (a parer mio) la maggior parte dei problemi riscontrati.

Non so se continuerei a sviluppare l'applicazione, ma nel caso cercherei di implementare tutte le parti che la porterebbero a seguire tutte le regole del gioco, e correggerei i problemi rilevati precedentemente in questa sezione.

4.1.3 Eugenio Guidi

Il mio ruolo all'interno del gruppo è stato, con buona sicurezza, equivalente a quello degli altri membri. Ho partecipato attivamente alla progettazione delle parti comuni del progetto e alla risoluzione di problemi che ci hanno bloccato per settimane! Mi rendo conto che per quello che riguarda le mie parti personali, c'è sicuramente margine di miglioramento, come sempre, ma comunque mi ritengo soddisfatto del lavoro che ho svolto. All'inizio, ammetto di non aver compreso appieno la complessità delle attività che avremmo dovuto affrontare, in particolare per quanto riguarda il rispetto dei pattern e delle architetture di progettazione. Questo iniziale errore di valutazione ha causato un rallentamento nello sviluppo della progettazione, ma, guardando indietro, mi rendo conto che questi errori sono stati fondamentali per la mia crescita personale e per la mia consapevolezza, specialmente per eventuali esperienze lavorative future. Un altro aspetto che avevo sottovalutato è stata la difficoltà nel lavorare in gruppo, considerando che ciascuno ha il proprio approccio e il proprio modo di ragionare, ho capito che il lavoro di squadra che abbiamo dovuto metterci è stata una vera e propria palestra. Per quanto riguarda il futuro, non credo che proseguirò con questo progetto, ma ne porto con me un'importante esperienza che mi ha fatto sicuramente crescere sia dal punto di vista tecnico che personale.

4.1.4 Federico Mantoni

La realizzazione di questo progetto è stata un'esperienza sicuramente formativa. Mi ha fatto comprendere veramente cosa vuol dire progettare un'applicazione e non solo programmarla, e come questa sia una parte cruciale. Inoltre, le numerose linee guida da seguire come gli strumenti di analisi statica, o il corretto uso di GitHub, mi hanno fatto capire quanto lavoro ci sia dietro ad un software che possa essere compreso e utilizzato da altre persone.

Anche l'esperienza di lavorare in gruppo mi ha insegnato quanto possa essere difficile comunicare un'idea o un concetto agli altri per quanto nella propria testa venga considerato semplice. In generale penso di essermi impegnato duramente per questo progetto.

Ritengo di aver avuto un buon spirito critico, di essere riuscito a comprendere un'idea altrui valutandone pregi e difetti, e molte volte non mi sono accontentato quando ritenevo che qualcosa potesse essere fatta meglio. É sicuramente necessario rivedere l'organizzazione generale del progetto e la valutazione del tempo necessario, poiché hanno portato alla consegna in ritardo di questo. Parte di ciò che era di mia competenza, probabilmente poteva essere implementato seguendo più accuratamente i pattern se non fosse stato per il motivo sopra.

A mio onesto parere non porterei avanti il progetto, ma di sicuro quest'esperienza è servita per affrontare progetti futuri con maggiore consapevolezza.

4.2 Difficoltà incontrate e commenti per i docenti

Si sono riscontrate difficoltà nella gestione dell'MVP, malgrado si siano fatte delle ricerche riguardanti le informazioni, anche utilizzando il libro "Design Patterns Elements of reusable Object-Oriented Software.". Da prima è stato difficile capire i vincoli del pattern MVC, che si voleva utilizzare in primis; poi nell'avvicendarsi dell'implementazione, si è cercato di incapsulare il più possibile i tre componenti, a causa di una malinterpretazione dei vincoli del pattern. Nelle fasi finali del progetto, si è voluto essere sicuri della giusta implementazione del pattern, e eseguendo ulteriori ricerche che vertevano su uno scopo diverso, si è identificato il pattern come MVP a causa della completa disconnessione diretta tra Model e View.

A Guida utente

Appena verrà eseguita l'applicazione, verrà mostrata la seguente schermata.

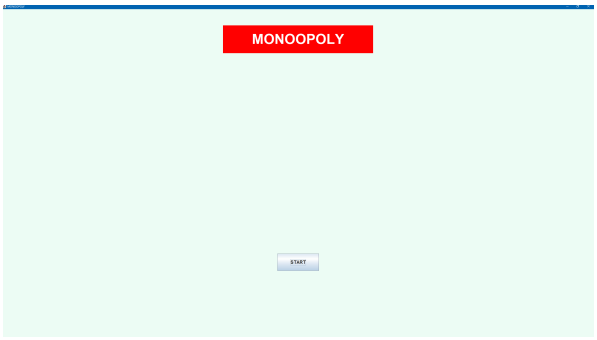

Figura A.1: Schermata iniziale del gioco

Qui basterà cliccare sul bottone "START". Si passerà alla selezione del numero dei giocatori, da 2 a 4.

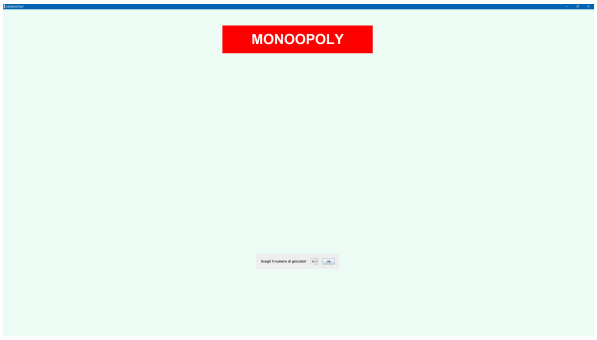

Figura A.2: Schermata scelta numero giocatori

Si selezioni il numero dei giocatori utilizzando le freccette o inserendo un numero valido, e premere il pulsante "Ok". Apparirà la seguente schermata:

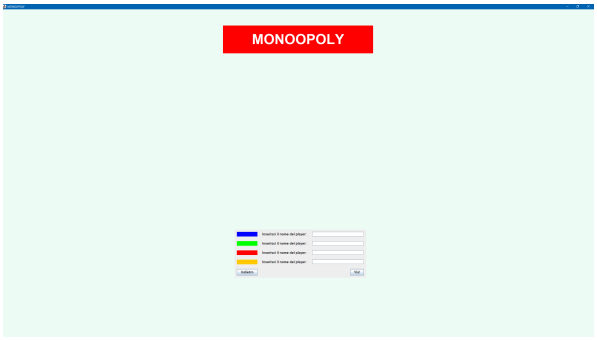

Figura A.3: Schermata nome dei giocatori

Qui si inseriscono i nomi dei giocatori dove si hanno a disposizione fino a 20 caratteri per giocatore, una volta immessi tutti i nomi in modo permesso, si prema "Via" per far partire il gioco, "Indietro" se si vuole tornare alla schermata precedente.

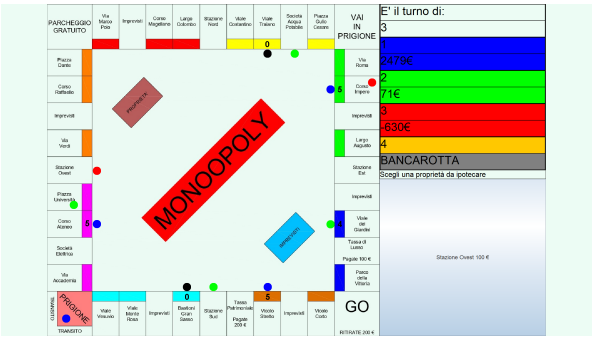

Figura A.4: Schermata esempio game board

Questa è la game board divisa in tre parti principali:

- Parte sinistra: composta dalla plancia di gioco, qui i giocatori rappresentati dai pallini colorati saranno mossi a seconda di ciò che accade nel gioco.
  - Quando durante il gioco si acquisteranno proprietà verrà visualizzato un pallino del colore del giocatore al di sopra della proprietà.
  - Quando verranno costruite delle case, verrà visualizzato il numero relativo sulla banda colorata della proprietà.
  - Nel caso una proprietà sia ipotecata il pallino al disopra della proprietà diverrà nera.
  - Appariranno durante il gioco alcuni messaggi di notifica o per eseguire una semplice scelta si/no
- Parte destra alta: qui sarà visualizzato il giocatore che ha il turno, e la lista dei giocatori con la propria situazione economica sempre aggiornata.
- Parte destra bassa: qui appariranno nel momento del bisogno due tipi di schermate:
  - Lancio dei dadi, che permetterà di lanciare i dadi.
  - Selezione di proprietà, qui verranno visualizzate tutte le proprietà tra le quali scegliere nei vari momenti del gioco che lo richiedono (costruzione/vendita case, ipoteca, disipoteca)
  - Quando non si dovranno eseguire nessuna di queste azioni, verrà visualizzata una breve descrizione della game board.
- Se non viene eseguita nessuna selezione nei vari avvisi e schermate, il gioco non andrà avanti
- Se si vuole chiudere il gioco, basterà aspettare che nella parte in basso a destra apparirà una selezione da eseguire, dopo di che premere sulla tastiera "esc" e selezionare "yes" sul messaggio che apparirà sul monitor.

Quando il gioco finirà sarà visualizzata una schermata con il nome del giocatore vincente; per chiudere del tutto l'applicazione premere sulla "x" in alto a destra.

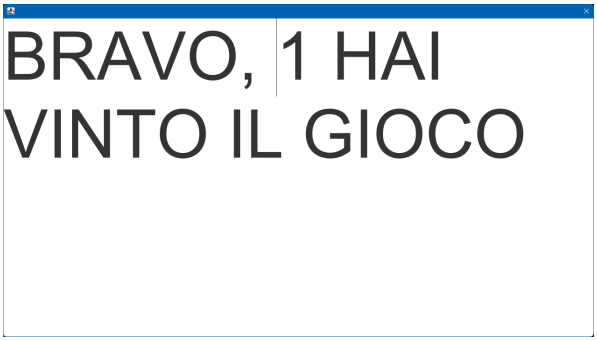

Figura A.5: Schermata di vincita

B Esercitazioni di laboratorio

B.0.1 Eugenio Guidi

- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=178723#p247437
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p248367
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=180101#p249066
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=181206#p250800

B.0.2 Federico Mantoni

Le seguenti esercitazioni si riferiscono all'anno 23/24

- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p210218
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211633
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212827