

”Cluedo”

Relazione per il progetto di ”Programmazione
ad Oggetti” A.A 2023/24

Shimaj Kevin
Paggetti Marco
Saponaro Mattia
Brighi Federico

17 settembre 2024

Indice

1	Analisi	3
1.1	Requisiti	3
1.2	Analisi e modello del dominio	6
2	Design	8
2.1	Architettura	8
2.2	Design dettagliato	9
2.2.1	Paggetti Marco	9
2.2.2	Shimaj Kevin	16
2.2.3	Saponaro Mattia	22
2.2.4	Brighi Federico	27
3	Sviluppo	40
3.1	Testing automatizzato	40
3.1.1	Paggetti Marco	40
3.1.2	Shimaj Kevin	41
3.1.3	Saponaro Mattia	41
3.1.4	Brighi Federico	41
3.2	Note di sviluppo	43
3.2.1	Paggetti Marco	43
3.2.2	Shimaj Kevin	43
3.2.3	Saponaro Mattia	44
3.2.4	Brighi Federico	44
4	Commenti finali	45
4.1	Autovalutazione e lavori futuri	45
4.1.1	Paggetti Marco	45
4.1.2	Shimaj Kevin	45
4.1.3	Saponaro Mattia	46
4.1.4	Brighi Federico	46

Capitolo 1

Analisi

1.1 Requisiti

Il software implementato mira alla simulazione di una partita del famoso gioco da tavolo "Cluedo", diversificando l'esperienza di gioco tramite l'aggiunta di elementi non esistenti in quello originale. I 3 giocatori, nei panni di investigatori, dovranno cercare di risolvere un delitto, caratterizzato da un'arma, un sospettato e una stanza. A ciascun giocatore, verranno distribuite le carte dei 3 tipi sopra citati, che sicuramente non fanno parte della soluzione. Sarà poi ruolo del giocatore risolvere il delitto, muovendosi all'interno della mappa e formulando accuse nel momento in cui si trova in una stanza. La condizione di fine partita si verifica nel momento in cui un giocatore, entrando nella stanza centrale, indovina la soluzione, oppure quando tutti gli sfidanti hanno sbagliato l'accusa finale.

Requisiti funzionali

Il numero di giocatori è impostato a 3. Questi parteciperanno alla medesima istanza dell'applicativo alternandosi mediante un sistema di turni.

- L'applicativo si apre con il menù di avvio di gioco, in cui verrà chiesto di indicare i nomi dei giocatori ed il colore della loro pedina. Una volta premuto il tasto di avvio sarà possibile iniziare il gioco.
- Ogni giocatore, all'inizio del proprio turno, dovrà lanciare un dado, il cui risultato stabilirà il numero di passi disponibili.
- Successivamente, al giocatore verrà assegnato un effetto imprevisto, che influenzera il suo turno. Gli imprevisti sono:

- **Nullo:** Non influisce in alcun modo sul giocatore.
- **Rilancio del dado:** Permette al giocatore di rilanciare il dado, incrementando il numero di passi a propria disposizione.
- **Scambio della posizione:** Il giocatore scambia la propria posizione con quella di un altro giocatore ancora in gioco.
- **Scambio di una carta:** Il giocatore scambia, casualmente, una delle proprie carte, con una carta (se possibile dello stesso tipo) di un altro giocatore ancora in gioco.
- **Modifica del numero di passi:** Il giocatore subisce un incremento/decremento del numero di passi a sua disposizione.
- Ora il giocatore avrà la possibilità di muoversi (in relazione ai passi disponibili) attraversando le varie caselle, solamente in direzioni ortogonali, un passo per volta, senza poter attraversare muri o altri giocatori. Alcune di queste caselle possono presentare un effetto:
 - **Malus:** Il giocatore dovrà saltare il turno successivo.
 - **Bonus:** Il giocatore dovrà rilanciare il dado, incrementando i passi a sua disposizione.
- Quando un giocatore entra in una stanza, può formulare una accusa, che consiste nell'indicare, oltre alla stanza in cui si trova, una carta per ciascuno dei tipi rimanenti.
- Se presente, seguendo l'ordine dei turni, il primo giocatore che possiede almeno una delle carte selezionate, la mostra all'accusatore.
- Ogni giocatore possiede un taccuino che, in automatico, tiene traccia delle carte viste.
- Se all'inizio del proprio turno, dopo aver ricevuto l'effetto imprevisto, il giocatore si trova in una stanza in cui è presente una botola, potrà utilizzarla per spostarsi nella stanza ad essa collegata.
- Nel momento in cui il giocatore decide di entrare nella stanza centrale, può formulare l'accusa finale, nella quale indicherà i tre elementi che secondo lui compongono il delitto.
- Se questi sono corretti, il giocatore ha vinto e la partita termina. Al contrario, se l'accusa risulta non coincidere esattamente con la soluzione, il giocatore ha perso.

- Se tutti i giocatori sbagliano la propria accusa finale, il gioco termina ugualmente.
- Al termine della partita viene mostrata la soluzione e le statistiche di gioco:
 - Numero di passi fatti.
 - Numero di accuse fatte.
 - Numero di carte viste.
 - Numero di stanze visitate.
- Nel caso si voglia terminare la partita anticipatamente, è sia possibile terminarla senza effettuare un salvataggio, sia salvarla per riprenderla in seguito.

Requisiti non funzionali

- Il gioco deve funzionare in ciascuno dei 3 sistemi operativi principali: Linux, Windows e MacOs.
- L'interfaccia deve essere progettata in maniera tale da rendere l'esperienza di gioco fluida ed intuitiva.

1.2 Analisi e modello del dominio

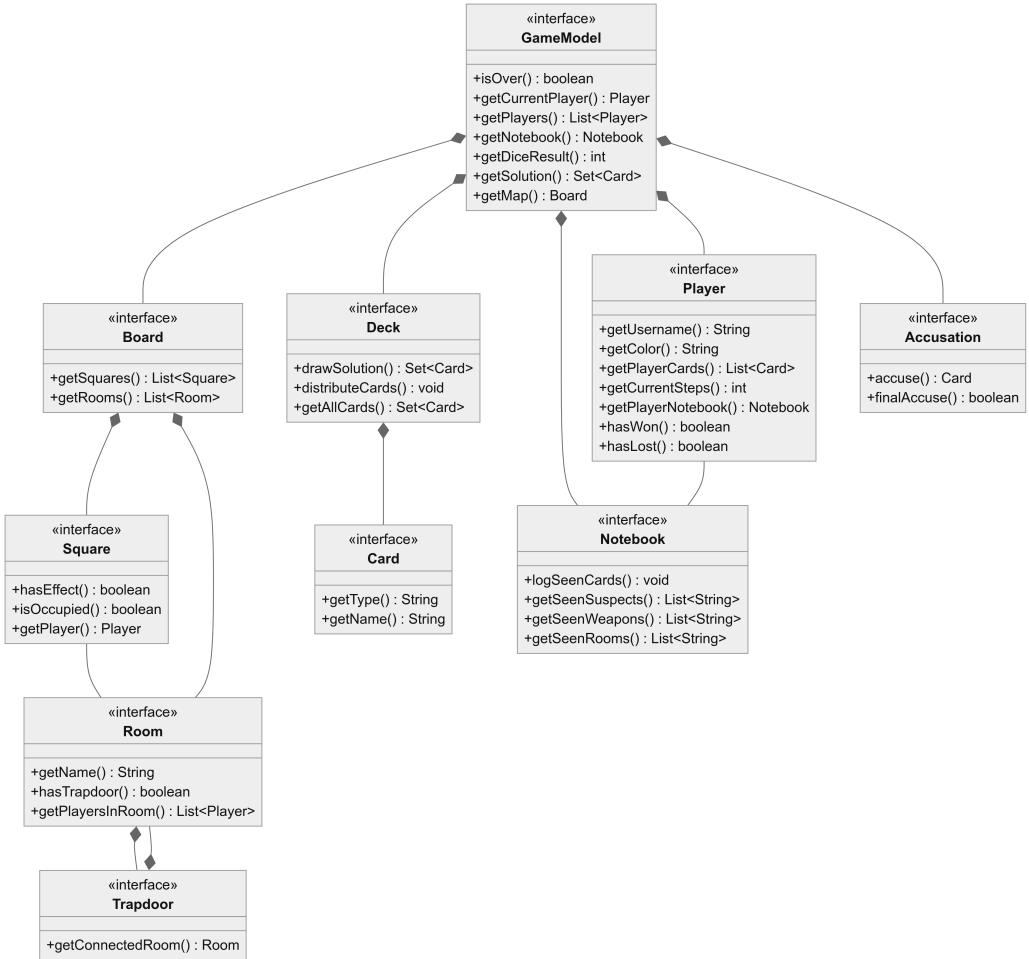


Figura 1.1: Diagramma UML del dominio

Il gioco si caratterizza di entità **Player**. Ciascun player possiede delle **Card**, prelevate dal **Deck** di gioco (dal quale vengono anche prelevate le cards della soluzione). Ciascun player può lanciare il dado e muoversi all'interno della **Board**. La board di gioco si compone di **Square** e **Room**. Gli square caratterizzano i corridoi percorribili per giungere all'interno delle room. Ciascuna room si compone di square e solamente alcune di queste presentano una **Trapdoor**. Ciascuna trapdoor permette ai player di spostarsi dalla room corrente a quella ad essa collegata. Ciascun player possiede un proprio **Notebook**, il quale tiene traccia in modo automatico delle card possedu-

te e viste. Infine, i player possono decidere di effettuare una **Accusation**, aumentando la loro probabilità di intuire la soluzione del gioco.

Capitolo 2

Design

2.1 Architettura

Per la realizzazione del software si è scelto, in fase di progettazione, di utilizzare il pattern architettonale MVC (Model-View-Controller). A livello implementativo, il punto cardine del Model è l'interfaccia **GameModel**. Questa ha il compito di richiamare ed effettuare i vari controlli sui metodi delle classi appartenenti al modello, gestendo così la logica e lo stato di quest'ultimo. Dal lato Controller, il componente principale dell'architettura, è l'interfaccia **MainController**, la quale rende possibile l'utilizzo dei controller minori, ciascuno dei quali si occupa di gestire uno specifico elemento della View. Inoltre ha anche il compito di inizializzare un'istanza del GameModel e di garantire il corretto avvio del gioco. A livello di View, l'elemento principale è il **MainGameFrame**. Questo, istanzia i bottoni, che hanno il compito di richiamare i controller corrispondenti, agendo così sullo stato del modello. Tali cambiamenti, comportano, mediante l'azione del controller principale, l'aggiornamento degli ulteriori elementi grafici. Tramite il corretto utilizzo del pattern MVC, viene agevolata una possibile sostituzione della componente grafica, senza dover modificare le parti di Model e Controller, non essendoci dipendenze con esse.

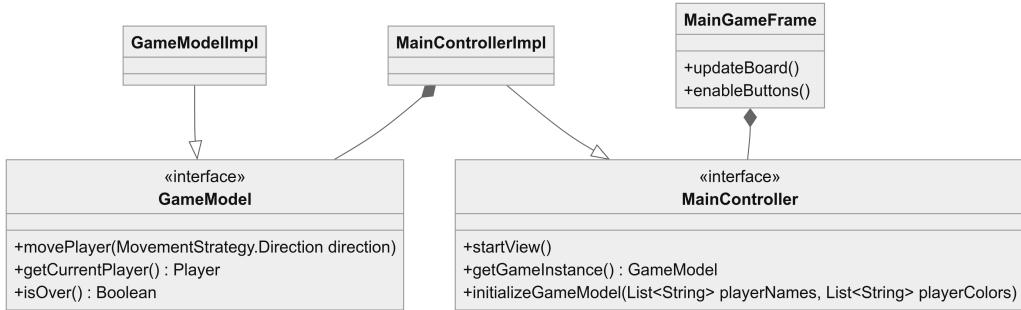


Figura 2.1: Schema UML dell’implementazione del pattern MVC

2.2 Design dettagliato

2.2.1 Paggetti Marco

Diverse tipologie di carte

PROBLEMA

Nel gioco Cluedo, è necessario creare diverse tipologie di carte (personaggi, armi e stanze) in modo centralizzato e standardizzato, oltre che flessibile e manutenibile. La creazione manuale di tali carte, all’interno del codice, può portare a duplicazione e difficoltà di manutenzione.

SOLUZIONE

La soluzione proposta utilizza il **Factory Pattern**. Questo pattern permette di centralizzare la logica di creazione delle carte all’interno della sola classe CardFactory, migliorando la manutenibilità e riducendo la duplicazione del codice. Tale classe fornisce metodi statici per creare carte di tipo personaggio, arma e stanza. Si utilizza CardImpl (classe che implementa l’interfaccia Card) come implementazione per tutte le tipologie di carte, specificando il tipo come parametro.

PRO:

- La logica di creazione delle carte è centralizzata in un’unica classe, rendendo il codice più pulito e manutenibile.
- Aggiungere nuove tipologie di comporterebbe solamente l’aggiunta di un nuovo metodo nella classe CardFactory.

- La creazione degli oggetti è incapsulata all'interno della CardFactory, nascondendo i dettagli di implementazione e promuovendo il principio di separazione delle responsabilità.

CONTRO

- L'aggiunta di una CardFactory può aggiungere un livello di complessità al codice, soprattutto se il numero di tipi di carte è limitato.

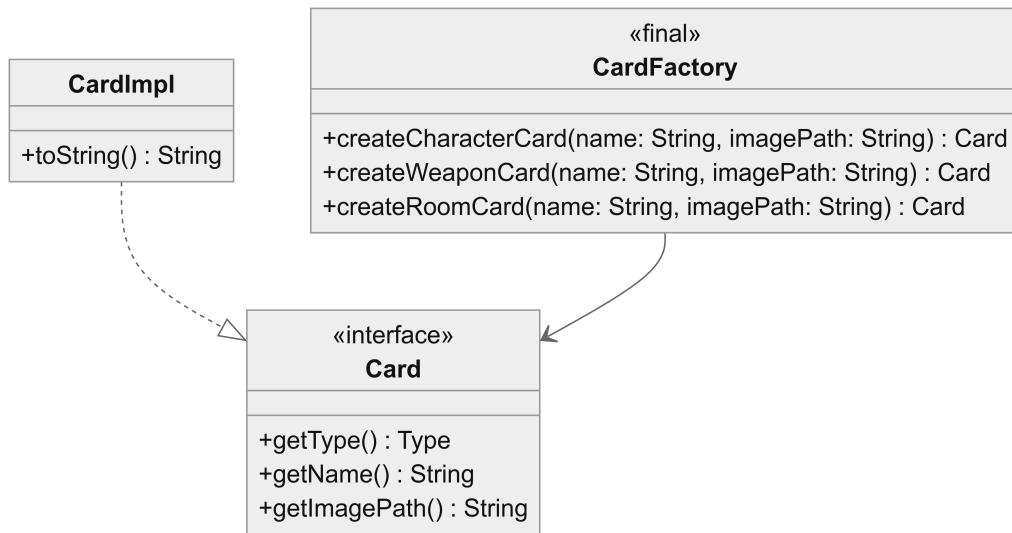


Figura 2.2: Schema UML di Card e CardFactory

Mazzo per la scelta della soluzione e per la distribuzione delle carte rimanenti

PROBLEMA

Nel gioco Cluedo, è necessario gestire un mazzo di carte che include personaggi, armi e stanze. Il mazzo deve permettere di estrarre una soluzione (un personaggio, un'arma e una stanza) e distribuire le carte rimanenti tra i giocatori.

SOLUZIONE

La soluzione proposta utilizza la classe `DeckImpl` per rappresentare il mazzo di carte. Questa classe, prima di tutto, utilizza la `CardFactory` per creare le carte. Definisce poi liste statiche di tutti i nomi delle carte, divise per

tipologia e accessibili mediante metodi statici. Fornisce metodi per estrarre la soluzione e distribuire le carte tra i giocatori (garantisce che i giocatori abbiano un numero equo di carte). Infine, gestisce l'insieme completo delle carte e quelle rimanenti.

PRO:

- Utilizzando CardFactory, la creazione delle carte è centralizzata, migliorando la manutenibilità del codice.
- Le liste statiche facilitano l'accesso ai nomi delle carte divise per tipo.

CONTRO

- La classe DeckImpl deve essere modificata ogni volta che vengono aggiunte nuove tipologie di carte o nuove funzionalità del gioco.

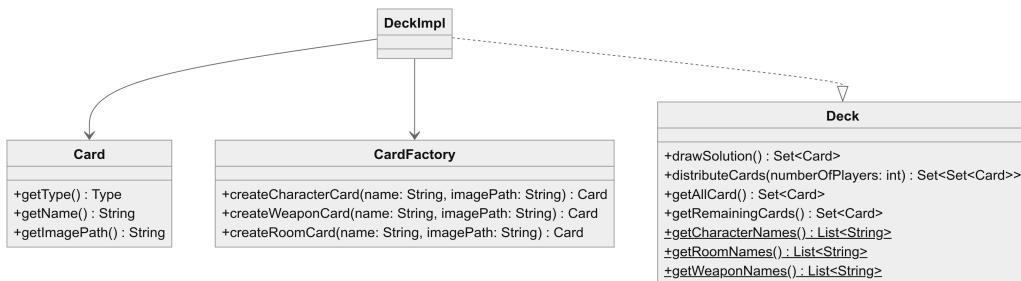


Figura 2.3: Schema UML di Deck

Caselle della mappa di gioco con relativi effetti

PROBLEMA

Nel gioco Cluedo, è necessario gestire le caselle della mappa, ognuna con una posizione specifica e un possibile effetto particolare. Le caselle devono poter applicare effetti ai giocatori che vi si trovano sopra.

SOLUZIONE

La soluzione proposta utilizza la classe SquareImpl per l'implementazione delle caselle della mappa e la classe SquareFactory per centralizzare la creazione delle caselle. La classe SquareImpl implementa l'interfaccia Square e gestisce la posizione, l'effetto e la presenza del giocatore sulla casella. La

classe SquareFactory utilizza il **Factory Pattern** per creare diverse tipologie di caselle (normali, bonus e malus). Gli effetti (nessuno, malus e bonus) sono gestiti utilizzando lo **Strategy Pattern**, con diverse implementazioni dell'interfaccia Effect.

PRO:

- Utilizzando SquareFactory, la creazione delle caselle è centralizzata.
- Utilizzando lo Strategy Pattern, gli effetti delle caselle possono essere facilmente estesi o modificati.

CONTRO

- SquareImpl e SquareFactory devono essere mantenute aggiornate ogni volta che vengono aggiunte nuove tipologie di caselle o effetti.

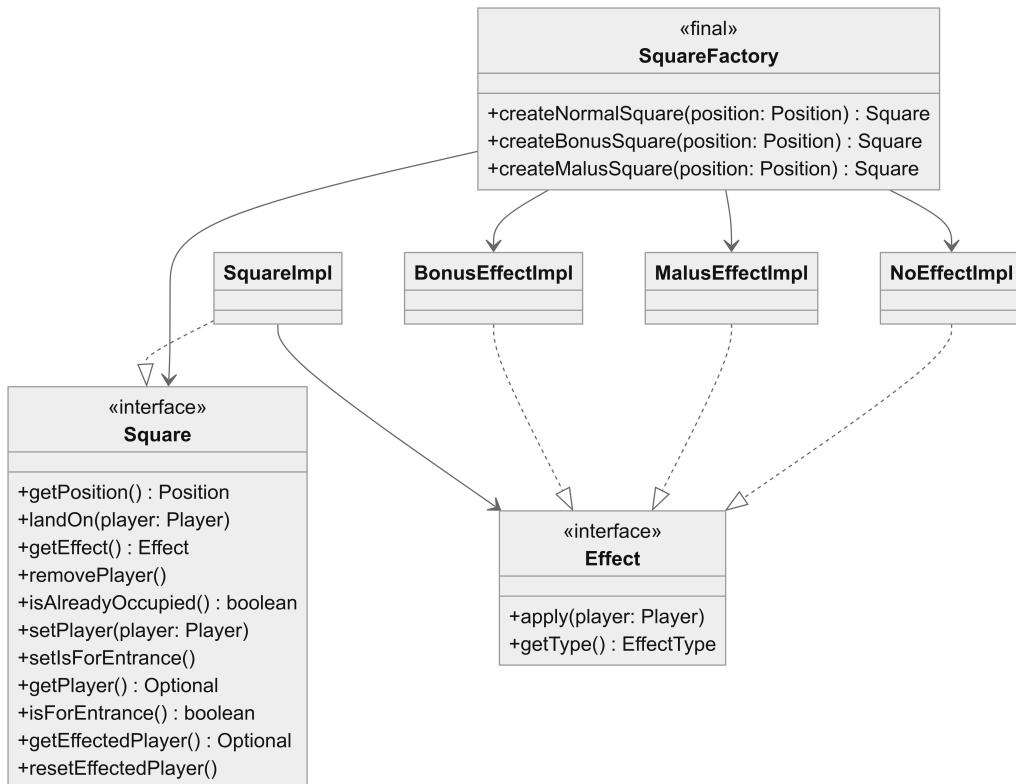


Figura 2.4: Schema UML di Square, SquareFactory e Effect

Stanze e passaggi segreti

PROBLEMA

Nel gioco Cluedo, è necessario rappresentare le stanze e i passaggi segreti tra di esse. Le stanze devono poter contenere giocatori, avere ingressi specifici e possono avere un passaggio segreto che collega ad un'altra stanza.

SOLUZIONE

La soluzione proposta prevede due classi principali. RoomImpl implementa l'interfaccia Room e rappresenta una stanza nel gioco. Gestisce le caselle, gli ingressi, i giocatori presenti e un eventuale passaggio segreto. TrapDoorImpl implementa l'interfaccia TrapDoor e rappresenta un passaggio segreto che collega due stanze.

PRO:

- Separare le stanze dai passaggi segreti rende il codice più modulare e fa assumere a ciascuna classe una responsabilità ben definita.

CONTRO

- La classe RoomImpl dipende da diverse altre classi e interfacce, aumentando in questo modo il numero di dipendenze nel progetto.

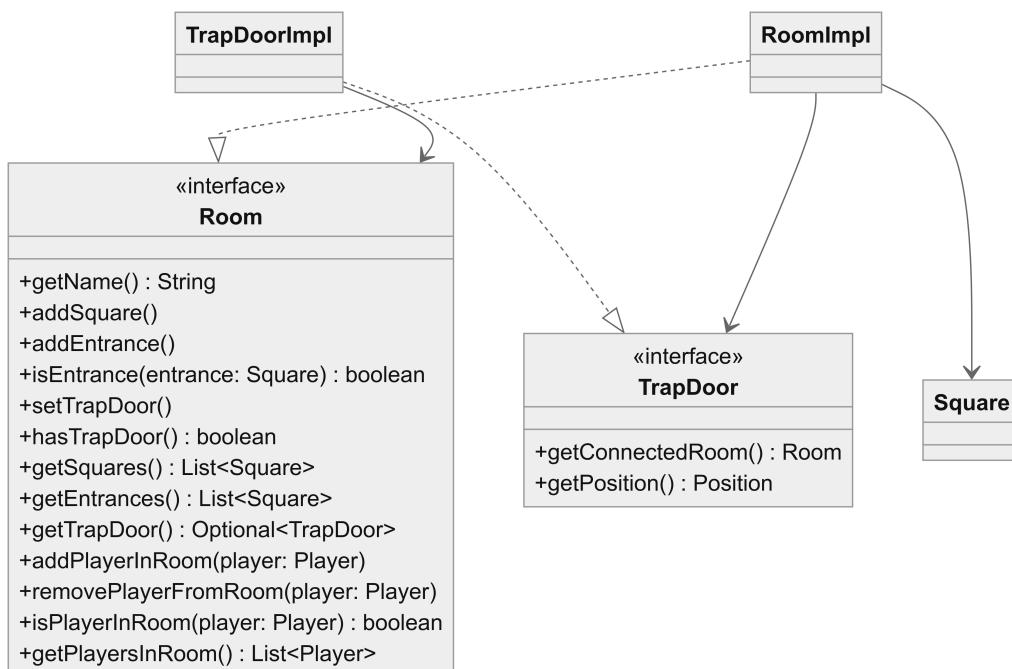


Figura 2.5: Schema UML di Room e TrapDoor

Mappa di gioco

PROBLEMA

La mappa del gioco Cluedo deve rappresentare fedelmente il layout del gioco, includendo stanze (con annesso eventuale passaggio segreto), ingressi e caselle di spostamento con possibilità di presentare effetti. Inoltre, le posizioni delle caselle bonus e malus devono essere casuali e le entrate nelle stanze devono essere gestite correttamente.

SOLUZIONE

La soluzione proposta prevede l'implementazione di una classe `BoardImpl` che gestisce la mappa di gioco. Questa classe inizializza le stanze e le caselle, basandosi su una disposizione predefinita, salvata internamente come matrice. Per garantire la casualità delle posizioni delle caselle con effetto (3 malus e 3 bonus, solo all'esterno delle stanze), è stato implementato il metodo `createRandomSquare`. Questo metodo genera caselle con o senza effetto in modo casuale. Le posizioni valide per queste caselle sono raccolte in una lista e poi mescolate per aumentare la casualità. Durante l'inizializzazione della mappa, le caselle con effetto sono distribuite casualmente tra le posizioni valide, assicurando che non superino il numero massimo predefinito. Per gestire correttamente le entrate nella stanza, oltre a salvare le posizioni degli ingressi all'interno delle stanze, sono salvate anche le posizioni frontali agli ingressi stessi. Questo garantisce che ogni ingresso abbia una posizione univoca sulla mappa da cui poter entrare, eliminando possibili ambiguità sulle posizioni dalle quali è possibile entrare nelle stanze. Infine, vengono mantenute salvate la lista di caselle su cui è possibile spostarsi (che comprendono quelle esterne alle stanze e le caselle di ingresso delle stanze) e la lista delle stanze.

PRO:

- Possibilità di aggiungere facilmente nuovi effetti alle caselle o di aumentare il numero limite di questi.
- Il salvataggio delle liste delle stanze e delle caselle su cui è possibile spostarsi, permette una gestione più efficiente delle interazioni con la mappa.

CONTRO

- La mappa è standardizzata ad una particolare versione del gioco e non è possibile utilizzare una mappa con una diversa disposizione delle stanze senza dover modificare la matrice delle disposizioni e quindi il modello.

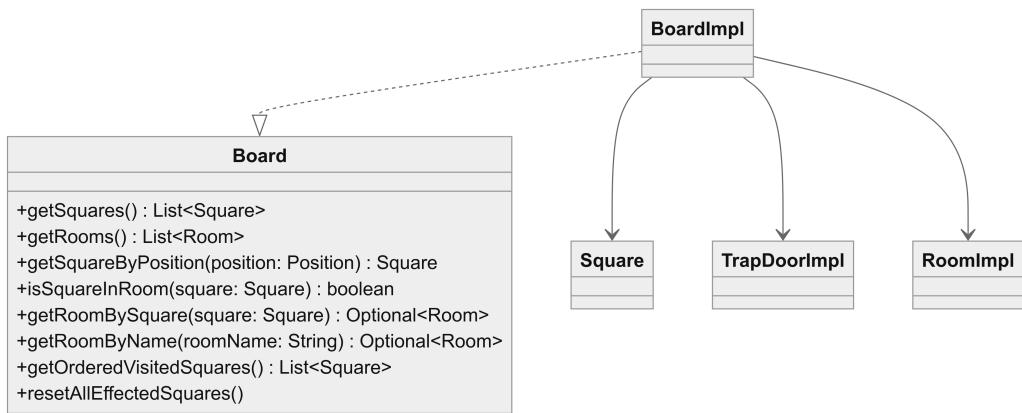


Figura 2.6: Schema UML di Board

2.2.2 Shimaj Kevin

Gestione del movimento del giocatore all'interno della mappa

Problema:

La gestione del movimento di un giocatore all'interno della mappa presenta due aspetti principali che devono essere risolti:

- **Logica e validazione del movimento:** è necessario determinare e validare la nuova posizione del giocatore dopo ogni mossa. Bisogna quindi assicurarsi che il giocatore non esca dai confini della mappa, né si sposti su caselle occupate o bloccate. Inoltre, è importante gestire scenari specifici come l'ingresso in stanze o l'uso di botole.
- **Esecuzione del movimento:** una volta che la mossa è stata calcolata e validata, il sistema deve effettivamente eseguire lo spostamento del giocatore. Nella versione attuale dell'applicativo, il giocatore deve decidere a ogni passo in quale delle quattro direzioni adiacenti (non diagonali) muoversi.

Soluzione:

Per affrontare questi due aspetti, sono stati adottati due pattern distinti, ma complementari:

- **Pattern Strategy:** responsabile della logica di calcolo e validazione del movimento.
- **Pattern Command:** responsabile dell'effettiva esecuzione del movimento, utilizzando la logica fornita dalla strategia.

L'interfaccia **MovementStrategy** è quella che utilizza il **pattern Strategy**, permettendo la separazione della logica del movimento dalla sua esecuzione, rendendo il sistema flessibile per l'introduzione di nuove regole di movimento. La concreta implementazione di tale interfaccia è fornita dalla classe **Board-Movement**.

D'altra parte, l'interfaccia che andrà ad utilizzare il **pattern Command** è **MovementCommand**, quest'ultima si occuperà dell'effettivo spostamento di un giocatore all'interno della mappa, ma solo dopo che la logica di movimento è stata validata dalla strategia. Nello specifico, il metodo **execute()** esegue la mossa, aggiornando la posizione del giocatore se la mossa è valida. Un esempio concreto è la classe **MoveInSingleDirection**, che sposta il giocatore in una singola direzione (passo per passo). Questo tipo di approccio

permette al sistema di gestire in modo flessibile l'introduzione futura di nuovi tipi di movimento (come ad esempio movimenti diagonali o teletrasporti verso caselle specifiche della mappa), senza modificare il codice già esistente. Ogni nuovo tipo di movimento può essere rappresentato da una nuova classe, la quale implementa il metodo **execute()** per eseguire il movimento desiderato. Infine, nella soluzione proposta, viene utilizzato il **record Position** per rappresentare le coordinate bidimensionali all'interno del gioco. Questa struttura semplifica la gestione delle posizioni nel gioco, in quanto genera automaticamente i metodi getter per le coordinate. Lo schema UML riassuntivo di quanto detto è presente in figura 2.7.

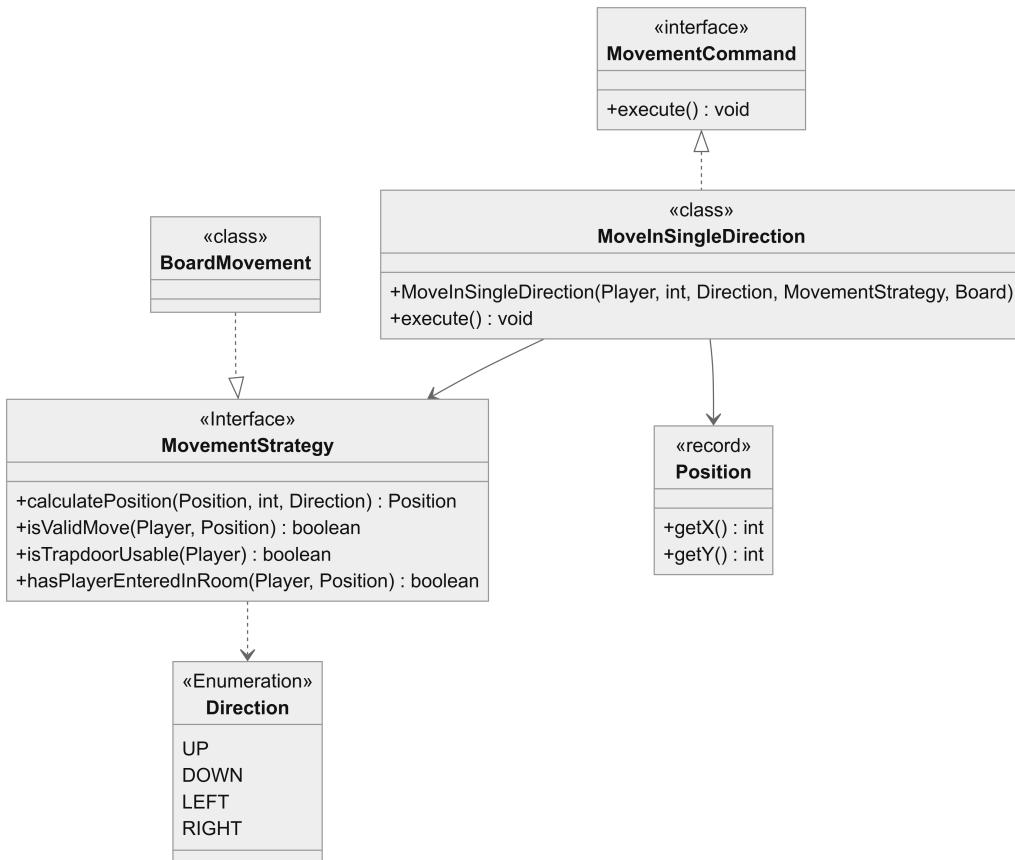


Figura 2.7: Schema UML di MovementCommand e MovementStrategy

Gestione del Player nel sistema

Problema:

Nel gioco Cluedo, è necessario identificare il giocatore tramite attributi fondamentali (come ad esempio username e colore della pedina).

Oltre alla identificazione di base, il giocatore è una risorsa centrale nel sistema di gioco e date le sue numerose iterazioni con altre entità (mappa, carte, turni di gioco), esso deve essere una entità mutabile.

Bisogna quindi consentire la modifica dello stato del giocatore durante il corso della partita.

Soluzione:

Per risolvere il problema della gestione e della identificazione del giocatore sono state adottate due interfacce principali:

- **Player:** rappresenta l'interfaccia di base che definisce gli attributi e i metodi di **sola lettura** delle informazioni del giocatore.
- **MutablePlayer:** un'estensione dell'interfaccia Player, che aggiunge i metodi per **modificare lo stato** del giocatore durante la partita.

L'uso dell'interfaccia MutablePlayer rappresenta una soluzione chiave per gestire il giocatore in modo flessibile e sicuro:

- Presenta alcuni metodi fondamentali per le dinamiche di gioco come **setPosition()**, **setPlayerTurn()** e **setHasWon()**;
- Se in futuro dovessero essere introdotte nuove meccaniche di gioco che richiedono di modificare lo stato del Player, queste possono essere facilmente integrate estendendo l'interfaccia MutablePlayer;
- La distinzione tra Player (sola lettura) e MutablePlayer (lettura e modifica) rende il codice più leggibile e gestibile, separando nettamente le responsabilità. Il diagramma relativo alla gestione del problema è quello a figura 2.8.

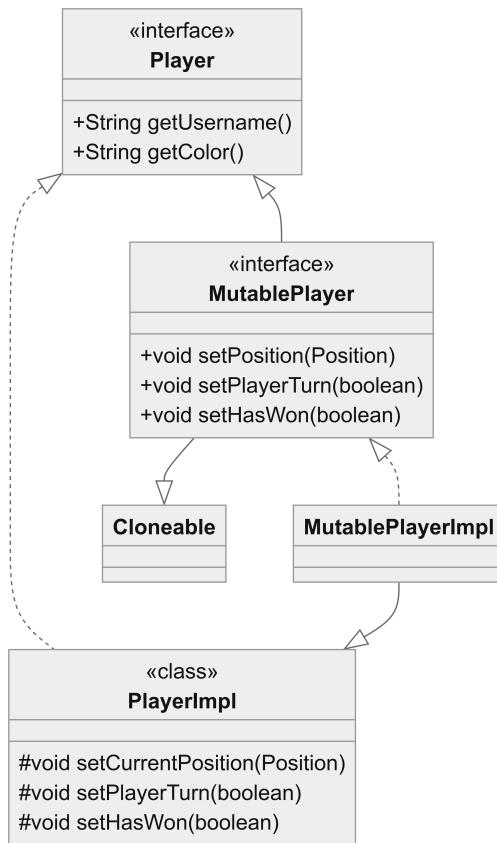


Figura 2.8: Schema UML di Player e MutablePlayer

Creazione del Player

Problema:

Nel contesto del gioco, è necessario creare istanze di giocatori con diverse proprietà (come username e colore).

Soluzione:

Per risolvere il problema della creazione di un Player in modo centralizzato è stato adottato il **Factory Pattern**. L'interfaccia **SimplePlayerFactory** definisce il metodo `createPlayer()`, che permette di creare un nuovo Player senza esporre l'implementazione concreta. Le classi che desiderano creare un Player non devono preoccuparsi di quale classe concreta sia effettivamente utilizzata, devono solo chiamare questo metodo. Anche se la classe **SimplePlayerFactoryImpl** crea un'istanza di `MutablePlayerImpl`, il metodo

restituisce un oggetto di tipo Player. Questo consente di trattare i giocatori attraverso la loro interfaccia pubblica, senza esporre i dettagli dell'implementazione interna. Se in futuro si volesse cambiare l'implementazione concreta (ad esempio, sostituendo MutablePlayerImpl con un'altra classe), sarebbe sufficiente modificare solo l'implementazione della factory, senza toccare il resto del codice. Lo schema riassuntivo di quella che è la soluzione si trova nell'immagine 2.9.

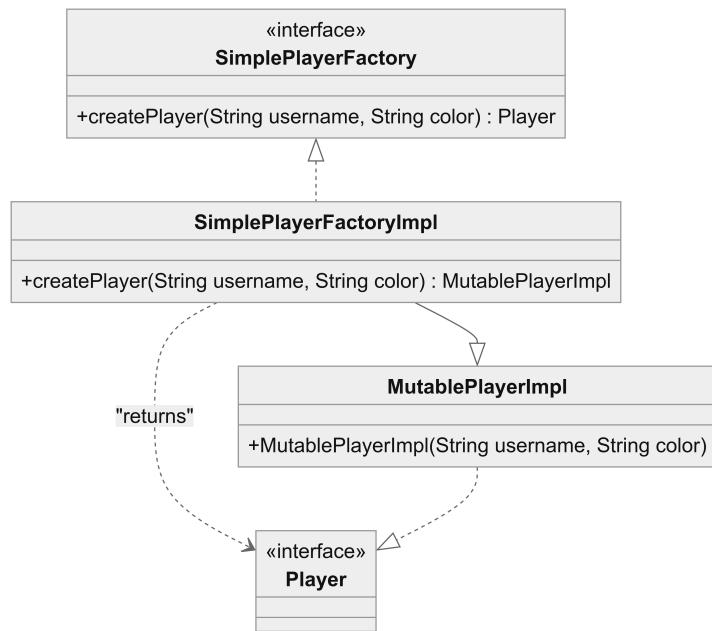


Figura 2.9: Schema UML di SimplePlayerFactory

Creazione di una nuova Partita e ripristino di una partita Salvata

Problema:

La creazione di un modello di gioco per una partita di Cluedo richiede l'inizializzazione di diverse componenti cruciali come i giocatori e la soluzione finale del gioco. La complessità aumenta ulteriormente nel caso della ripristinazione della partita salvata con dati già preimpostati. Sorge quindi la necessità di dover gestire numerosi parametri.

Soluzione:

In fase di implementazione ho deciso di adottare il **Pattern Builder**, che consente di costruire il modello di gioco in modo modulare e sicuro, procedendo passo per passo. Il pattern utilizzato permette di costruire oggetti complessi come il GameModel in maniera incrementale, assicurandosi che tutte le componenti necessarie siano fornite prima di restituire un'istanza completa di gioco. L'interfaccia **GameModelBuilder** cattura i metodi utili per creare una nuova istanza di modello come **withGameSolution()** per impostare la soluzione del gioco, e **addPlayer()**, che garantisce la presenza esatta di una delle regole fondamentali del sistema: la presenza esatta di 3 giocatori. Lo schema UML di quanto detto è presente nell'immagine 2.10.

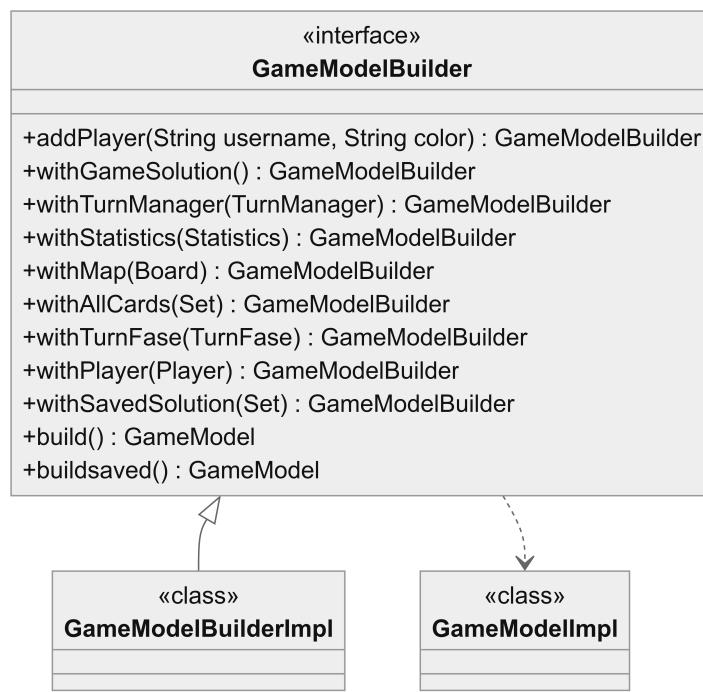


Figura 2.10: Schema UML della creazione del GameModel

2.2.3 Saponaro Mattia

PROBLEMA:

unire le varie classi di model per permetterne un facile uso da parte dei vari controller.

SOLUZIONE:

si è deciso di creare un'interfaccia **GameModel**, implemetata in **GameModelImpl**, che, usando i vari metodi creati dalle classi appartenenti al model, crea quello che sarà l'entry point del **pattern MVC**.

Costruendo una classe così fatta, i **controller** possono utilizzare i metodi del model direttamente da qui, semplificandone molto la struttura. In questa classe sono presenti vari metodi che permettono di ottenere informazioni sul **Player** che sta giocando in un determinato momento, o sulla **Board**. Queste, una volta lette permetteranno al controller di far comportare la View di conseguenza. Sono poi stati creati altri metodi che rappresentano le azioni compibili dall'utente durante il suo turno e che aggiornano la condizione della **Board** e del **Player** in base a ciò che succede durante la partita. L'intera classe è indipendente dall'effettivo numero di giocatori, ciò permette di poter, in futuro, aggiornare il software per giocare con un numero arbitrario di **Player**.

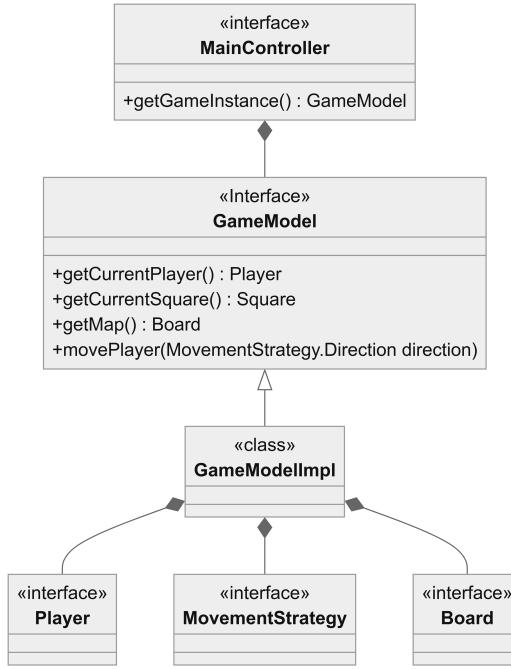


Figura 2.11: Rappresentazione UML dell’interazione tra GameModel e Controller.

PROBLEMA:

si rende necessario modellare la sequenzialità del turno di ogni giocatore e l’impossibilità di compiere determinate azioni che possano disturbare l’andamento del gioco.

SOLUZIONE:

per evitare che il **Player** compia più volte le stesse azioni, se non richiesto dal gioco (ad es. il lancio del dado), e per impedire che vengano eseguite in una sequenza non concessa dalle regole, si è deciso di creare un sistema di **fasi** per ogni turno. Le fasi, descritte nella enum di utility **TurnFase**, sono:

- **ROLL DICE**: fase del turno in cui il Player può lanciare il **Dice**.
- **DRAW UNFORESEEN**: qui viene assegnato al Player l’**Unforeseen Effect** che influenzera il suo turno.
- **MOVE PLAYER**: fase di movimento, oppure momento in cui, se possibile, si può usare una **Trapdoor**.

- **MAKE ACCUSATION**: momento in cui è possibile fare una **Accusation**, che sia solo un'ipotesi o quella finale.
- **END TURN**: fine del proprio turno.

Dopo aver creato questo sistema, è stato inserito un controllo sulla TurnFase in ogni metodo del **GameModel**, in modo tale che questi lanciassero un'eccezione nel caso di uso in un momento non opportuno.

È inoltre necessario che i vari metodi settino correttamente la successiva fase del turno a seconda di ciò che accade nel gioco, onde evitare un blocco totale del software.

Tramite il **GameModelImpl** è possibile ottenere la fase corrente del turno, la quale viene utilizzata anche per disabilitare i tasti, non utilizzabili in quel momento, nella GUI.

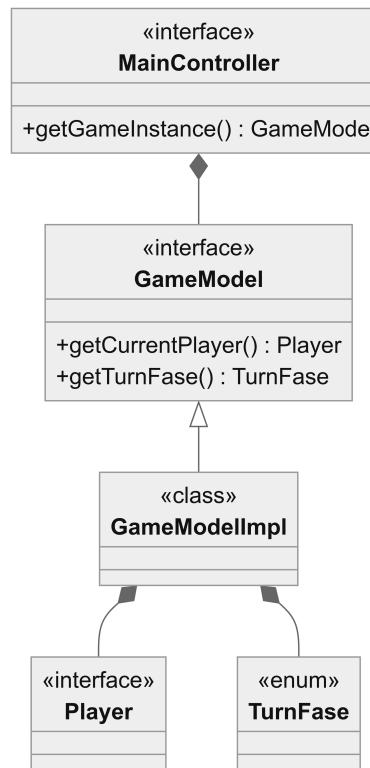


Figura 2.12: Rappresentazione UML dell'implementazione del GameModel con le sue fasi.

PROBLEMA:

Gestire le accuse compiute dai vari Player.

SOLUZIONE:

Si utilizza il **pattern MVC** per gestire le possibili **Accusation** fatte dai **Player**. Il giocatore, tramite la **View**, può selezionare due o tre **Card**, a seconda del tipo di accusa, che verranno passate all'**AccusationController** o al **FinalAccusationController**. Questi andranno a richiamare i metodi presenti in **GameModel** per effettuare l'accusa. Nel caso di un'ipotesi di accusa, viene ritornata, se questa è andata a buon fine, una delle Card che verranno mostrate a video nell'interfaccia utente. Nel caso invece sia stata effettuata l'accusa finale, si è scelto di mostrare a video la soluzione nel caso questa sia corretta, altrimenti verrà mostrato un messaggio che informa il giocatore del fatto che ha perso.

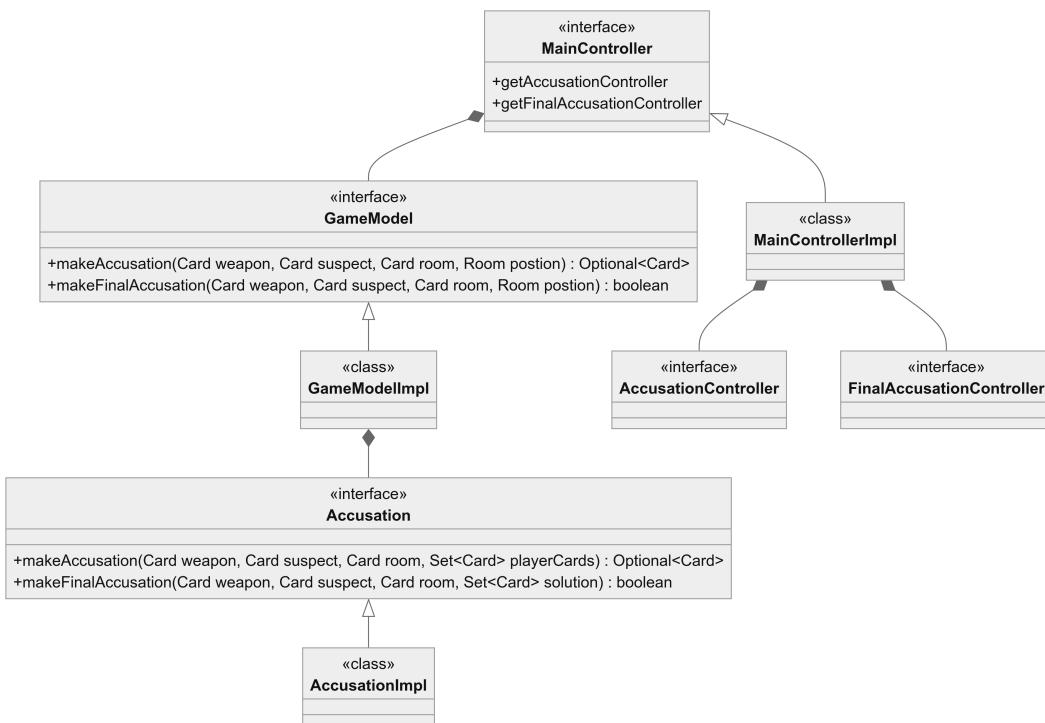


Figura 2.13: Rappresentazione UML dell'implementazione dell'Accusation.

PROBLEMA:

gestione delle statistiche del gioco e visualizzazione delle varie classifiche.

SOLUZIONE:

anche le statistiche di gioco sono state implementate usando il **pattern MVC**. Si istanzia la classe **StatisticsImpl** che implementa l'interfaccia **Statistics** la quale ha più funzioni:

- Fornire metodi per incrementare le varie statistiche di gioco.
- Restituire, quando richieste, le classifiche.
- Immagazzinare i dati di queste.

Il **GameModel** usa in maniera appropriata i vari metodi per incrementare le statistiche e, alla fine del gioco, lo **StatisticsController** fornirà alla **View** le classifiche, già ordinate in modo decrescente dalla classe **Accusation**, pronte per essere visualizzate a video. Per quanto riguarda l'ordinamento, siccome sono state usate delle mappe, ho creato un metodo dentro ad **Accusation** che trasforma una mappa in una coppia di liste ordinate in modo decrescente.

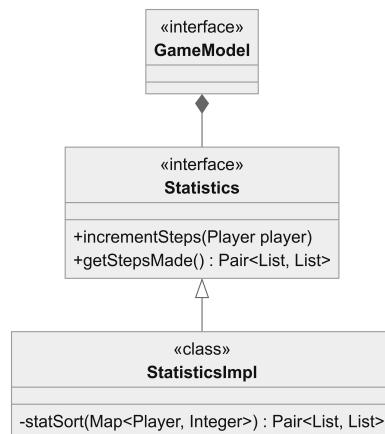


Figura 2.14: Rappresentazione UML dell'implementazione delle Statistcs.

2.2.4 Brighi Federico

LANCIO DEL DADO

PROBLEMA:

Necessario implementare una funzione che, simulando il lancio di un dado tradizionale utilizzato nei giochi da tavolo, restituisca un valore casuale che sia compreso tra il numero minimo di facce, che è 1, e il numero massimo di facce, fissato a 6. Questo valore deve rappresentare il risultato del lancio del dado e riflettere in modo realistico il comportamento di un dado fisico, dove ogni faccia ha la stessa probabilità di essere ottenuta.

SOLUZIONE:

Questa è una funzione semplice ma fondamentale per la simulazione di giochi da tavolo, perché da essa dipende il movimento dei giocatori. Per risolvere il problema ho sviluppato la classe **DiceImpl**, la quale implementa l’interfaccia **Dice**, che tramite il metodo **rollDice()** restituisce un intero che è un numero casuale compreso tra 1 e 6. Questo numero rappresenta il risultato del lancio del dado. L’implementazione di **DiceImpl** sfrutta l’incapsulamento e la separazione delle responsabilità attraverso l’uso dell’interfaccia **Dice**. Per aumentare il carico di lavoro relativo a questa funzionalità, ho cercato di implementare a livello grafico una simulazione di lancio di dado, dove le varie facce di esso ruotano casualmente per terminare poi nel risultato ottenuto.

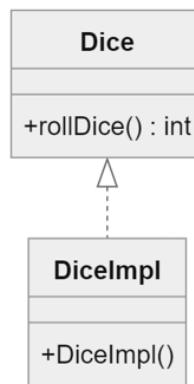


Figura 2.15: Rappresentazione UML dell’implementazione di Dice.

TACCUINO DI GIOCO

PROBLEMA:

Consentire al giocatore di visualizzare in maniera automatica tutte le informazioni rilevanti sulle carte che possiede, organizzandole in modo ordinato e suddividendole in categorie specifiche, come sospetti, armi e stanze. Questo sistema deve permettere di mostrare non solo le carte in possesso del giocatore all'inizio della partita, ma anche quelle che vengono progressivamente scoperte o rivelate durante il corso del gioco, garantendo così un aggiornamento continuo delle informazioni e una facile consultazione, al fine di supportare il giocatore nella formulazione di accuse e strategie.

SOLUZIONE:

Inizialmente l'idea era quella di creare uno spazio di scrittura per il giocatore dove si poteva provare effettivamente l'esperienza di uso di un taccuino. Tuttavia, insieme al gruppo, ho preferito scegliere un' alternativa in cui le informazioni vengono aggiornate in modo automatico dal taccuino stesso.

La soluzione definitiva prevede, tramite la classe **NotebookImpl**, la creazione di tre insiemi separati per le carte riguardanti i sospettati, le armi e le stanze che il giocatore ha visto durante la partita. Una volta che il gioco inizia al giocatore vengono assegnate 6 carte che vengono immediatamente segnate nel taccuino tramite il metodo **initialize()**. Ogni volta che una carta viene scoperta (tramite l'imprevisto di scambio carte o in seguito alla formulazione di accuse) viene inserita nel Set corrispondente tramite il metodo **logSeenCards()**. Viene inoltre fornito un metodo per determinare ognuno dei diversi tipi di carte (sospettato, arma, stanza) e aggiornarne le sezioni del taccuino di conseguenza (metodi getter). L'uso di Set piuttosto che List mi garantisce che ogni carta sia registrata solo una volta, evitando possibili duplicati.

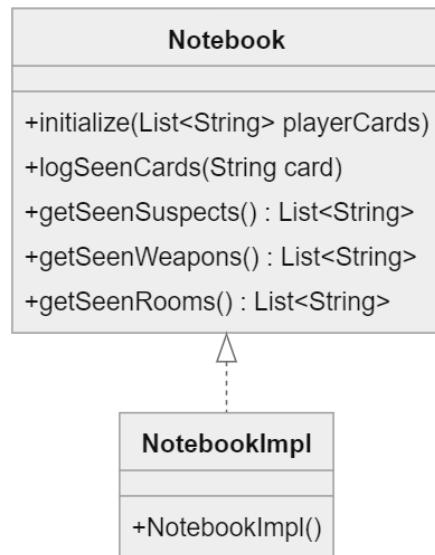


Figura 2.16: Rappresentazione UML dell’implementazione di Notebook.

GESTIONE DEI TURNI

PROBLEMA:

Gestire in modo efficace e fluido il flusso del gioco, garantendo che il passaggio del turno da un giocatore all'altro avvenga senza intoppi e in conformità con le regole stabilite. Un corretto flusso di gioco implica non solo il semplice passaggio del turno al giocatore successivo, ma richiede anche un controllo costante e preciso delle meccaniche di gioco. Questo include, ad esempio, la verifica di eventuali condizioni speciali che potrebbero influenzare l'ordine dei turni, come l'eliminazione di giocatori che sono stati sconfitti o l'applicazione di effetti che possono influire sull'ordine di gioco. Oltre a ciò, è fondamentale gestire correttamente la terminazione del gioco quando vengono soddisfatte determinate condizioni di vittoria o sconfitta, in modo da assicurare che la partita si concluda al momento giusto e secondo le regole prestabilite.

SOLUZIONE

La classe **TurnManagerImpl** risolve il problema attraverso un meccanismo ciclico che controlla lo stato di avanzamento del gioco e assicura che solo i giocatori attivi partecipano allo scambio di turni.

La classe mantiene traccia di una lista di giocatori e utilizza un indice (**currentPlayerIndex**) per monitorare il giocatore corrente.

Durante ogni turno l'indice viene aggiornato e alla fine del turno si passa al successivo giocatore in lista. La classe fornisce anche meccanismi per rimuovere giocatori dal gioco in caso di sconfitta e la verifica della terminazione della partita. In particolare:

- **Passaggio del turno:** Il metodo **switchturn()** è responsabile di passare il turno al giocatore successivo, ignorando eventuali eliminati. Questo garantisce che solo i giocatori ancora attivi in gioco possano partecipare al ciclo dei turni.
- **Rimozione dei giocatori:** Il metodo **removeplayer()** rimuove un giocatore dalla lista, in seguito ad una accusa finale errata, e aggiorna correttamente l'indice del giocatore corrente, verificando che questo non porti alla terminazione della partita.
- **Fine Partita:** Il metodo **checkGameEndCondition()** Tiene conto del numero di giocatori ancora attivi e verifica le condizioni di fine partita, controllando quanti giocatori non hanno ancora vinto o perso.

La partita termina quando uno dei giocatori raggiunge la vittoria o quando tutti i giocatori sono stati eliminati. Questo passaggio è fondamentale per garantire che il gioco non prosegua inutilmente, assicurando che la partita si concluda correttamente una volta che è stato decretato un vincitore o che tutti i giocatori abbiano perso.

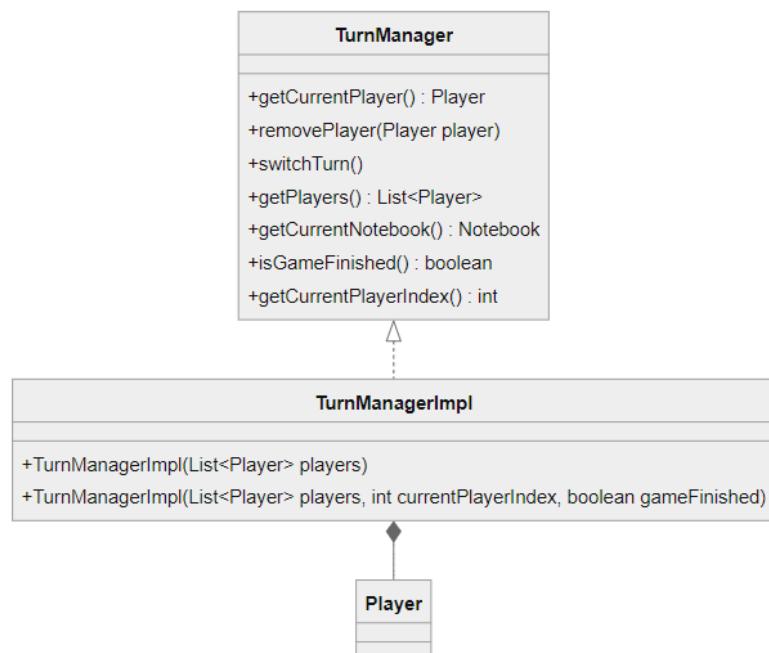


Figura 2.17: Rappresentazione UML dell’implementazione di TurnManager.

EFFETTI IMPREVISTO

PROBLEMA:

Generare in modo casuale un effetto imprevisto tra quelli già implementati, applicarlo automaticamente a un giocatore all'inizio del suo turno e garantire che ogni effetto abbia una diversa probabilità di essere selezionato, in base alla sua natura o rilevanza nel gioco.

Il sistema deve assicurare che l'effetto scelto venga eseguito correttamente, modificando lo stato di gioco in accordo con le regole definite per l'effetto stesso. Inoltre, è fondamentale mantenere un meccanismo flessibile e facilmente estendibile, che consenta l'aggiunta di nuovi tipi di effetti imprevisti senza dover apportare modifiche strutturali al codice esistente. Ogni nuovo effetto deve poter essere integrato con una probabilità di occorrenza personalizzabile, offrendo una gestione dinamica degli effetti.

SOLUZIONE:

In questo caso la nostra meccanica di gioco si differenzia da quella dell'originale gioco da tavolo: all'inizio di ogni turno, dopo aver eseguito il proprio lancio di dado, al player viene applicato un effetto imprevisto che può influenzare positivamente o negativamente lo svolgimento del suo turno. Sono stati implementati 5 tipi diversi di effetti :

- **NullEffect:** Effetto nullo che non influisce in alcun modo sul giocatore.
- **ReRollDice:** Effetto che permette al giocatore di rilanciare il dado, incrementando il numero di passi a propria disposizione.
- **SwapPosition:** Effetto che permette al giocatore di scambiare la propria posizione con quella di un altro giocatore ancora in gioco.
- **SwapCard:** Effetto che permette al giocatore di scambiare casualmente una delle proprie carte con una carta (se possibile dello stesso tipo) di un altro giocatore ancora in gioco.
- **MoveExtraStep:** Effetto che fa subire al giocatore un incremento/decremento del numero di passi a sua disposizione già determinati dal lancio del dado.

Tutti questi effetti implementano la struttura di base di un effetto imprevisto, descritta nella classe **UnforeseenEffect**:

di ogni effetto vengono presi il tipo (metodo **getType**), una breve descrizione (metodo **getDescription**), e in più viene implementata la funzione che lo caratterizza, applicata al player corrente (metodo **applyEffect**).

La Factory **UnforeseenEffectFactory** si occupa di creare gli effetti imprevisti in modo casuale, basandosi su un sistema di probabilità predefinito su ciascun effetto: ogni tipo di effetto ha una probabilità specificata, più alta negli effetti più "leggieri" a livello di gioco (come assegnare più o meno passi) e più bassa negli effetti più "pesanti" (come lo scambio di carte, che aiuta i giocatori ad avvicinarsi alla soluzione).

Ho deciso di utilizzare il Pattern Factory per la creazione degli oggetti perché esso centralizza la logica e evita di gestire manualmente la probabilità per ciascun effetto nella propria classe. Anche a livello di estendibilità risulta utile perché nel momento in cui si volesse aggiungere un nuovo imprevisto basterebbe creare la classe che implementa l'interfaccia e aggiornare la factory con la relativa probabilità.

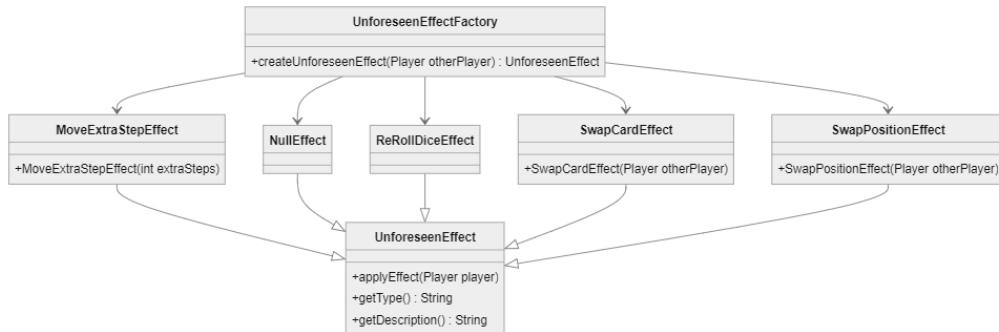


Figura 2.18: Rappresentazione UML dell'implementazione di **UnforeseenEffect** con i 5 tipi di effetti e la relativa Factory.

MENU' DI GIOCO

PROBLEMA:

Creare un menù di avvio del gioco che offra un'interfaccia intuitiva, semplice ed efficiente per l'utente, permettendogli di configurare facilmente i parametri della partita. Questo menù deve consentire la selezione e l'inserimento dei nomi e dei colori per i 3 giocatori, garantendo che l'utente possa personalizzare la partita in modo rapido e senza complicazioni. Una volta inseriti i dati, il sistema dovrà convalidare l'input ricevuto, verificando che i nomi non siano vuoti o troppo lunghi e che non ci siano duplicati sia nei colori scelti che nei nomi dei giocatori. Il menù di configurazione dovrà inoltre fornire messaggi di errore chiari e immediati in caso di input non valido, offrendo all'utente la possibilità di correggere le informazioni inserite. Solo dopo che tutti i parametri saranno stati validati correttamente, il gioco potrà essere avviato con i dati corretti, permettendo ai giocatori di iniziare la partita con le configurazioni personalizzate.

SOLUZIONE:

La soluzione proposta è implementata nella classe **GameMenuControllerImpl**, che segue le specifiche dettate dall'interfaccia corrispondente e contiene l'intera logica necessaria per gestire le operazioni del menù di avvio del gioco. Questa classe rappresenta il punto di controllo centrale per tutte le operazioni legate alla configurazione iniziale della partita e alla gestione delle informazioni inserite dagli utenti. Il suo obiettivo principale è quello di fornire un'interazione fluida e senza errori, permettendo di configurare correttamente i parametri fondamentali della partita come nomi e i colori dei giocatori. In aggiunta, la classe garantisce che tutte le informazioni raccolte siano verificate e validate prima di procedere all'avvio del gioco. Questo processo di configurazione include non solo la validazione dell'input dell'utente, ma anche la possibilità di recuperare potenziali partite salvate e la corretta chiusura del menu tramite un pulsante di uscita.

Funzioni principali:

- **Validazione dei dati:** Il metodo **startGame()** verifica che le condizioni di gioco siano soddisfatte e, quando un utente inserisce i nomi e i colori dei giocatori, verifica che l'input digitato sia corretto.
 - Il numero di giocatori deve essere tassativamente 3;
 - I nomi dei giocatori non deveono superare i 20 caratteri ognuno;

- I nomi e i colori non devono essere duplicati (verificato tramite l’uso dei Set);
- **Avvio del gioco:** Una volta che i dati dei vari giocatori sono stati correttamente validati, il metodo `setPlayer()` si occupa di creare le istanze dei giocatori, utilizzando i nomi e i colori forniti dall’utente. Il metodo verifica che le liste di nomi e colori contengano esattamente il numero di giocatori richiesto e che i nomi rispettino il limite di lunghezza prestabilito. Se tutte le condizioni sono soddisfatte, le istanze dei giocatori vengono create e aggiunte alla lista dei giocatori attivi. Ogni giocatore viene associato in modo univoco al proprio nome e colore, garantendo che tutte le informazioni siano configurate correttamente per l’inizio della partita.
- **Gestione dei salvataggi:** La classe fornisce il metodo `viewSaved-Games()` per visualizzare un eventuale partita salvata e caricarla utilizzando il controller dei salvataggi, il quale recupera lo stato precedente del gioco e lo inizializza, ricreando esattamente la partita salvata. Nel caso non fossero presenti salvataggi verrà mostrato un messaggio tramite un pop-up, e, inoltre, se verranno eseguiti più salvataggi diversi verrà ripristinata solo l’ultima partita, che andrà a sovrascrivere quelle precedenti.
- **Chiusura del gioco:** Il metodo `quitGame()` chiude tutte le finestre di gioco aperte, assicurandosi che tutte le risorse siano correttamente rilasciate.

Nel realizzare questa classe non ho esplicitamente implementato dei Pattern, ma il metodo `startGame()` ricorda il Validation Pattern, dato che crea delle condizioni di validazione sugli input dell’utente, mentre il metodo `setPlayer()` ricorda il Factory Pattern, dato che inizializza istanze di giocatori.

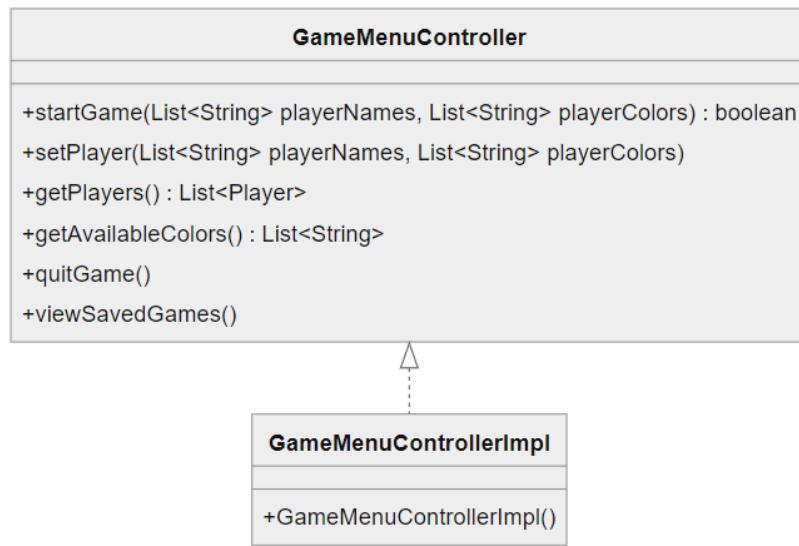


Figura 2.19: Rappresentazione UML dell’implementazione di GameMenuController.

SALVATAGGIO DELLA PARTITA

PROBLEMA:

Assicurarsi che tutte le informazioni rilevanti della partita, inclusi i dati dei giocatori, la mappa, la soluzione del gioco, le statistiche, le carte e ogni altra componente fondamentale, siano correttamente memorizzate in un formato persistente e che possano essere recuperate in modo sicuro e affidabile in qualsiasi momento. È necessario garantire che il sistema di salvataggio sia robusto e che i dati vengano conservati senza corruzione. Inoltre, deve essere possibile ripristinare con precisione lo stato della partita esattamente com'era al momento del salvataggio, permettendo ai giocatori di continuare il gioco senza perdere progressi o dettagli importanti.

SOLUZIONE:

La soluzione proposta è implementata nella classe **GameSaveControllerImpl**, la quale gestisce il salvataggio e il caricamento dello stato di gioco, garantendo che tutte le informazioni rilevanti siano memorizzate in modo sicuro e possano essere recuperate correttamente in qualsiasi momento. Questa classe implementa l'interfaccia corrispondente e fornisce un sistema robusto per la persistenza dello stato del gioco, permettendo di ripristinare esattamente la partita da dove era stata interrotta. Il funzionamento si basa sulla serializzazione e deserializzazione dello stato di gioco, rappresentato dalla classe interna **GameState**, che contiene tutte le informazioni necessarie per ripristinare il contesto di gioco.

Funzioni principali:

- **Salvataggio del gioco:** Il metodo **saveGame()** serializza lo stato attuale del gioco in un file. Crea un nuovo oggetto **GameState** che rappresenta lo stato del gioco corrente e viene serializzato tramite l'uso di un **ObjectOutputStream**.
- **Caricamento del gioco:** Il metodo **loadGame()** deserializza quel file e ripristina la partita salvata. Viene letto il file e deserializzato in un oggetto **GameState** che viene poi restituito come **Optional GameState**. Se il file non esiste viene restituito un **Optional.empty()** e viene loggato l'errore.

All'interno di **GameSaveControllerImpl** viene definita la classe **GameState**, la quale rappresenta lo stato del gioco ed è serializzabile.

Essa contiene tutte le informazioni rilevanti per ripristinarlo, ovvero :

- **Players** : i 3 giocatori che partecipano alla partita;
- **Solution** : la soluzione finale del gioco;
- **TurnManger** : la gestione del turno della partita, che comprende anche l'ordine dei giocatori;
- **Statistics** : le attuali statistiche della partita che poi aumenteranno con l'andare avanti del gioco e verranno mostrate alla fine di esso;
- **Map** : la mappa del gioco con i relativi bonus/malus;
- **AllCards** : tutte le carte in mano ai giocatori;
- **TurnFase** : la fase di gioco che c'era nel momento in cui la partita è stata salvata.

Miglioramenti possibili :

Attualmente la classe **GameSaveControllerImpl** gestisce un unico salvataggio. Questo approccio limita la possibilità di avere più salvataggi contemporaneamente, poiché ogni salvataggio sovrascrive il precedente. Avrei quindi potuto cercare di implementare diversamente questa funzionalità, aggiungendo un elenco di salvataggi. L'utente avrebbe così avuto la possibilità di ricaricare una delle partite salvate a sua scelta.

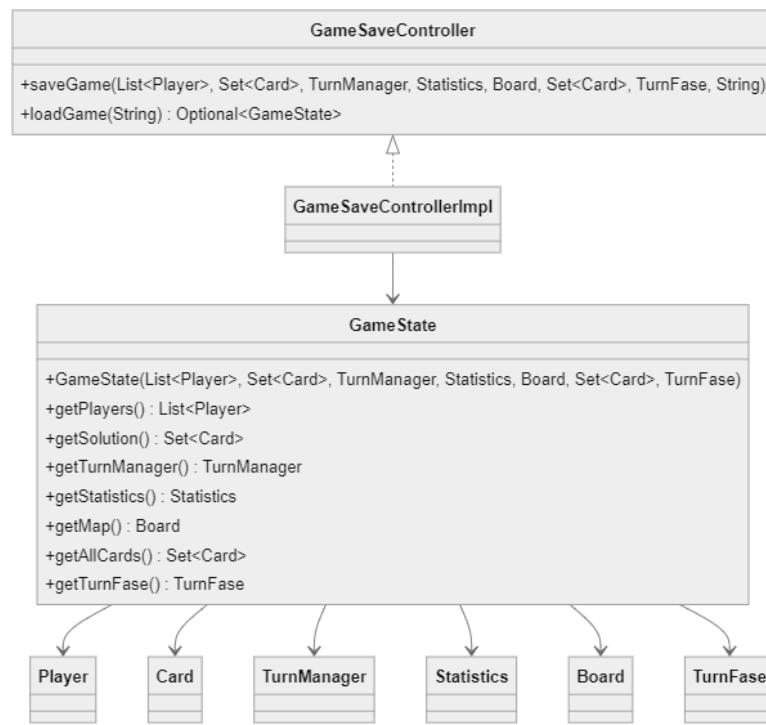


Figura 2.20: Rappresentazione UML dell’implementazione di GameSaveController e della inner class GameState, collegata agli aspetti principali della partita.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per verificare il corretto funzionamento delle diverse parti del progetto, sono stati creati Test automatizzati usando JUnit.

3.1.1 Paggetti Marco

- **BoardTest:** Testa la corretta inizializzazione della mappa, la corretta creazione di tutti gli elementi che vi appartengono e verifica l'esistenza delle caselle con effetto.
- **CardTest:** Testa la corretta creazione delle varie tipologie di carte mediante la CardFactory.
- **DeckTest:** Testa l'inizializzazione del mazzo di carte, la scelta della soluzione e la distribuzione delle carte rimanenti.
- **RoomTest:** Testa l'aggiunta di caselle, entrate e botole nelle stanze, la restituzione di questi elementi e la presenza del giocatore all'interno delle stanze.
- **SquareTest:** Testa la creazione delle caselle con e senza effetto, tramite la SquareFactory. Testa anche la presenza del giocatore sopra di esse.
- **TrapDoorTest:** Testa la stanza connessa dalla botola e la sua posizione.

3.1.2 Shimaj Kevin

- **GameModelBuilderImplTest:** testa che il builder aggiunga in modo corretto i Player e che venga settata la soluzione, prima di effettuare l'effettivo build. Inoltre viene testato il building della partita salvata.
- **PlayerImplTest:** testa l'inizializzazione corretta dei campi del Player e i relativi getter.
- **MutablePlayerImplTest:** testa il corretto funzionamento di tutti i setter
- **SimplePlayerFactoryImplTest:** testa che l'istanza del Player è correttamente creata, con lo specificato username e colore.

3.1.3 Saponaro Mattia

- **AccusationTest:** Testa il corretto funzionamento delle varie accuse.
- **StatisticsImpl:** Testa il corretto funzionamento delle varie statistiche e il corretto ordine delle classifiche.
- **GameModelTest:** Testa alcune funzioni del game model e alcune eccezioni lanciate.

3.1.4 Brighi Federico

- **DiceImplTest:** Testa che il lancio del dado avvenga correttamente e che ritorni come risultato un numero casuale nell'intervallo prefissato.
- **NotebookImplTest:** Testa il corretto funzionamento del taccuino, con l'inizializzazione e l'aggiunta automatica delle carte viste dal giocatore.
- **TurnManagerImplTest:** Testa il corretto scambio di turno di giocatori in vari casistiche di gioco (come eliminazione di giocatori).
- **MoveExtraStepEffectTest:** Testa la corretta applicazione dell'effetto al giocatore, con l'aumento/diminuzione del suo numero di passi disponibili.
- **ReRollDiceEffectTest:** Testa la corretta applicazione dell'effetto al giocatore, il quale può rilanciare il dado.

- **SwapCardEffectTest**: Testa la corretta applicazione dell'effetto tra due giocatori che si scambiano una carta.
- **SwapPositionEffectTest**: Testa la corretta applicazione dell'effetto tra due giocatori che si scambiano di posizione.
- **GameMenuControllerImplTest**: Testa il corretto funzionamento del menu di gioco, con i vari controlli sui parametri scelti dall'utente per iniziare la partita.
- **GameSaveControllerImplTest**: Testa il corretto salvataggio della partita e la corretta ripresa di una partita salvata. I test avvengono su un diverso file rispetto a quello del salvataggio della partita vera e propria per evitare eventuali sovrascritture.

3.2 Note di sviluppo

3.2.1 Paggetti Marco

- **Utilizzo di Optional:** Utilizzato in vari punti.

Un esempio è: <https://github.com/mpaggio/OOP23-Cluedo/blob/39bce14831d5977f0d4a2db65a42b04bcbe54f44/src/main/java/it/unibo/cluedo/model/square/impl/SquareImpl.java#L98-L100>

- **Utilizzo di Stream:** Utilizzato in vari punti.

Un esempio è: <https://github.com/mpaggio/OOP23-Cluedo/blob/39bce14831d5977f0d4a2db65a42b04bcbe54f44/src/main/java/it/unibo/cluedo/model/board/impl/BoardImpl.java#L402-L406>

- **Utilizzo di Lambda expression:** Utilizzato in vari punti.

Un esempio è: <https://github.com/mpaggio/OOP23-Cluedo/blob/39bce14831d5977f0d4a2db65a42b04bcbe54f44/src/main/java/it/unibo/cluedo/model/deck/impl/DeckImpl.java#L54-L56>

Per quanto riguarda l'implementazione della mappa di gioco all'interno del modello, durante lo sviluppo dell'applicazione ho provato a pensare ai possibili modi in cui tenere traccia della disposizione specifica dei vari elementi della mappa (caselle, spazi vuoti, botole e stanze). Ho quindi optato per utilizzare una matrice di interi, specificando per ogni valore utilizzato il suo significato in termini di elementi della mappa. Per fare ciò, ho preso spunto e riadattato la matrice ripresa dal codice della repository GitHub al seguente link: <https://github.com/k3vonk/UCD-Cluedo/blob/master/TileGrid.java>. Per quanto riguarda l'implementazione della mappa di gioco all'interno della view, per rappresentare la griglia delle celle di spostamento e dell'interno delle stanze ho preso spunto dal metodo drawTile implementato in una porzione di codice all'interno della medesima repository GitHub (<https://github.com/k3vonk/UCD-Cluedo/blob/master/Tile.java>), aggiungendo la parte riguardante il disegno della pedina del giocatore posizionato sopra la cella che deve essere rappresentata.

3.2.2 Shimaj Kevin

- **Utilizzo di Stream:** utilizzato in una sola classe: BoardMovement.

Un esempio è: <https://github.com/mpaggio/OOP23-Cluedo/blob/master/src/main/java/it/unibo/cluedo/model/movement/impl/BoardMovement.java#L64-L69>

3.2.3 Saponaro Mattia

- **Utilizzo di Optional:** utilizzato in vari punti, un esempio è: <https://github.com/mpaggio/OOP23-Cluedo/blob/master/src/main/java/it/unibo/cluedo/model/accusation/impl/AccusationImpl.java#L19-L23>.
- **Utilizzo di Stream:** utilizzato in vari punti, un esempio è: <https://github.com/mpaggio/OOP23-Cluedo/blob/master/src/main/java/it/unibo/cluedo/controller/statisticscontroller/impl/StatisticsControllerImpl.java#L21-L27>.
- **Utilizzo di Lambda expression:** utilizzato in vari punti, un esempio è: <https://github.com/mpaggio/OOP23-Cluedo/blob/master/src/main/java/it/unibo/cluedo/model/GameModelImpl.java#L312-L316>

3.2.4 Brighi Federico

- **Utilizzo di Optional:** Utilizzato in vari punti, soprattutto nella classe GameSaveControllerImpl.
Un esempio è: <https://github.com/mpaggio/OOP23-Cluedo/blob/39bce14831d5977f0d4a2db65a42b04bcbe54f44/src/main/java/it/unibo/cluedo/controller/gamesavecontroller/impl/GameSaveControllerImpl.java#L61-L90>
- **Utilizzo di Stream:** Utilizzato in vari punti.
Un esempio è: <https://github.com/mpaggio/OOP23-Cluedo/blob/39bce14831d5977f0d4a2db65a42b04bcbe54f44/src/main/java/it/unibo/cluedo/controller/gamemenucontroller/impl/GameMenuControllerImpl.java#L48-L51>
- **Utilizzo di Lambda expression:** Utilizzate solo in GameMenuImpl e SwapCardEffect.
Un esempio è: <https://github.com/mpaggio/OOP23-Cluedo/blob/39bce14831d5977f0d4a2db65a42b04bcbe54f44/src/main/java/it/unibo/cluedo/model/unforeseen/impl/SwapCardEffect.java#L44>

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Paggetti Marco

Questa è stata la mia prima esperienza di lavoro in gruppo. Il mio ruolo all'interno del team era focalizzato sulla modellazione della mappa di gioco, dei suoi elementi e delle carte. Ritengo di essermi impegnato molto e di aver svolto un ruolo significativo nella progettazione dell'architettura, nella comunicazione e nella collaborazione all'interno del gruppo. Tra i miei punti di forza, evidenzierei la determinazione nel cercare soluzioni ai problemi proposti e la conoscenza del gioco originale. Tuttavia, tra i miei punti di debolezza, riconosco la mia scarsa esperienza nel lavoro di gruppo e nella gestione della suddivisione degli incarichi. Durante il progetto, le principali difficoltà incontrate sono state l'utilizzo corretto di GitHub, l'organizzazione del dominio e della architettura del progetto nelle fasi iniziali, l'applicazione corretta del pattern MVC e l'integrazione finale del lavoro dei singoli membri del gruppo per far funzionare tutto assieme. Nonostante queste sfide, sono soddisfatto del lavoro svolto, dell'impegno profuso nel progetto e del risultato finale ottenuto.

4.1.2 Shimaj Kevin

Nonostante questa fosse la mia prima esperienza all'interno di un progetto di tale portata, mi ritengo abbastanza soddisfatto del mio contributo. Le difficoltà maggiori sono state principalmente due:

- 1.La fase di progettazione iniziale;
- 2.La coordinazione del lavoro tramite GitHub.

In particolare, la fase di progettazione si è rivelata impegnativa poiché richiedeva un’analisi approfondita delle esigenze del sistema e una pianificazione accurata delle componenti software. All’inizio, per me, è stato complicato definire chiaramente le responsabilità delle varie classi e interfacce, oltre a prevedere l’evoluzione futura del progetto. Tuttavia, con il supporto del gruppo e un confronto costante, siamo riusciti a delineare una struttura solida che ha semplificato le fasi successive. Per quanto riguarda GitHub, la sfida principale è stata imparare a gestire i branch. Nonostante qualche difficoltà iniziale nel coordinare i contributi di tutti i membri del team, alla fine abbiamo sviluppato un flusso di lavoro collaborativo. Un aspetto che ho trovato particolarmente interessante è stata la creazione del model relativo alla mia parte del progetto. Questa fase mi ha permesso di approfondire l’uso di alcuni design pattern fondamentali e di riflettere su come modularizzare il codice per facilitare future estensioni. Nel complesso, ritengo che il progetto abbia rappresentato un’importante esperienza formativa, sicuramente un qualcosa di davvero utile per i miei futuri progetti.

4.1.3 Saponaro Mattia

Ho già avuto piccole esperienze di lavoro di gruppo prima di questo progetto. Il mio ruolo, all’interno del gruppo, riguardava l’implementazione delle accuse, delle statistiche e infine la gestione dell’intera classe contenente il model del gioco. Penso di essermi impegnato a dovere, cercando di rendermi sempre disponibile e ascoltando i bisogni dei vari membri del gruppo. Tra i miei punti di forza evidenzierei: la capacità di analisi dei problemi e la determinazione al completamento del progetto. I punti di debolezza sicuramente riguardano la difficoltà di visione di insieme iniziale, la quale può rallentare lo sviluppo, specialmente in fase di suddivisione dei compiti. Le sfide maggiori di questo progetto sono state: il corretto uso di git, la suddivisione del lavoro e l’implementazione del pattern MVC. Mi sento di poter dire di aver appreso da queste sfide e di aver messo impegno in questo lavoro.

4.1.4 Brighi Federico

Sono complessivamente soddisfatto del lavoro svolto dal nostro gruppo su questo progetto. È stata la mia prima esperienza con un progetto di questa portata e con una metodologia di lavoro collaborativa basata sull’utilizzo di Git, e ciò ha inizialmente comportato alcune difficoltà. Tuttavia, dopo un’attenta fase di analisi e modellazione iniziale svolta in presenza con tutti i componenti del gruppo, siamo riusciti a superarle con successo. Abbiamo lavorato per tutto il mese di agosto e la parte iniziale di settembre singolar-

mente, tenendoci sempre in contatto per chiarire dubbi o gestire e risolvere eventuali errori, fissandoci varie deadline per completare le diverse parti di lavoro.

Personalmente, sono soddisfatto del contributo che ho dato al progetto: ero consapevole di avere alcune lacune, ma non mi sono demoralizzato anzi, ho recuperato le lezioni e ho concentrato il mio lavoro principalmente sull'implementazione delle funzionalità assegnatemi, assicurandomi che esse funzionassero correttamente, piuttosto che andando ad adottare tecniche di programmazione avanzate non mie che avrebbero potuto rallentare e compromettere il progresso di lavoro complessivo, sia personale che del gruppo. Concludendo, questa esperienza mi ha permesso di crescere notevolmente, migliorando sia la mia conoscenza della materia, sia la mia capacità di lavorare in gruppo con altre persone utilizzando strumenti come Git.

Ripensandoci, avrei preferito affrontare questo progetto prima di sostenere l'esame, poiché mi ha aiutato a comprendere meglio diversi aspetti della programmazione object oriented.

Appendice A

Guida utente

All'avvio del software, ci si trova davanti un menù interattivo che richiede di inserire username dei giocatori e colore della relativa pedina (gli username devono essere lunghi non più di 20 caratteri, non vuoti e diversi l'uno dall'altro, così come per i colori).



Figura A.1: Menù iniziale di gioco

Per avviare la partita è possibile sia premere il tasto "Start game", il quale ne inizierà una nuova, sia premere il tasto "Resume", il quale riprenderà l'ultima partita salvata dall'utente (se presente).



Figura A.2: Schermata principale di gioco

Nel momento in cui viene presentata la schermata principale di gioco, saranno abilitati solamente i pulsanti relativi alle azioni disponibili al giocatore. Allo stesso modo, man mano che le fasi di gioco cambiano, i pulsanti non di interesse alla fase corrente vengono disabilitati e gli altri attivati. Nella parte in basso a destra della schermata, è comunque sempre presente una sezione che elenca le possibili azioni disponibili al giocatore in quel momento. Ciò che contraddistingue il reale inizio del turno è il lancio del dado, mediante il pulsante "Roll dice".

Per poter muovere la propria pedina, è necessario utilizzare il joystick presente nella parte destra della schermata, dove i 4 pulsanti rappresentano le possibili direzioni di movimento.

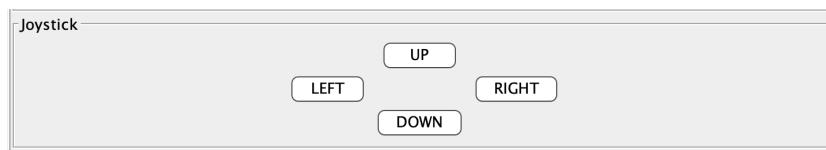


Figura A.3: Joystick per il movimento

All'interno delle stanze, per formulare un'accusa, sarà necessario cliccare l'apposito pulsante ("Make final accusation" in caso ci si trovi nella stanza centrale o "Make normal accusation", negli altri casi). Alla pressione del pulsante, verrà presentata una finestra in cui bisognerà specificare gli elementi dell'accusa.

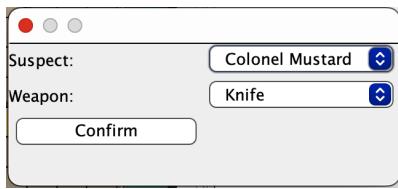


Figura A.4: Finestra per l'accusa normale



Figura A.5: Finestra per l'accusa finale

Nel caso di accusa normale, dopo aver confermato la scelta, se nessuno degli altri giocatori possiede una delle carte specificate nell'accusa, verrà segnalato tramite avviso. In caso contrario, verrà mostrata una finestra contenente una delle carte trovate.

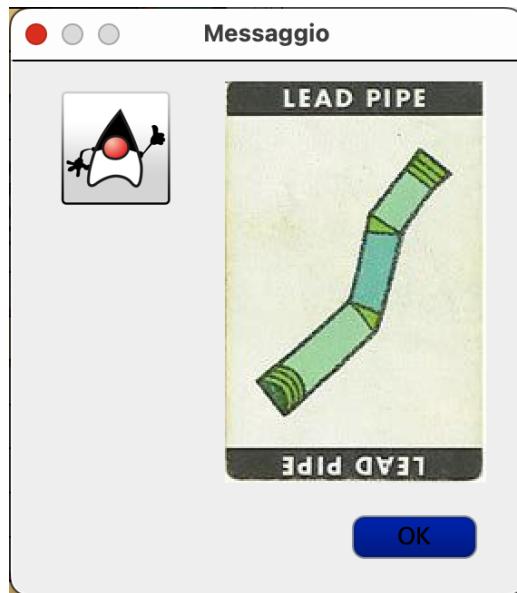


Figura A.6: Finestra con la carta posseduta da un'altro giocatore

Nel caso di accusa finale, dopo aver confermato la scelta, se l'accusa si rivela sbagliata, verrà segnalato tramite avviso e il giocatore non potrà più partecipare alla partita. Se l'accusa dovesse rivelarsi corretta (o nel caso in cui tutti i giocatori avessero perso), verrà presentata una finestra contenente la soluzione della partita.



Figura A.7: Accusa finale corretta

Dopo esser stata mostrata la soluzione del gioco, verrà presentata una finestra contenente le statistiche della partita.

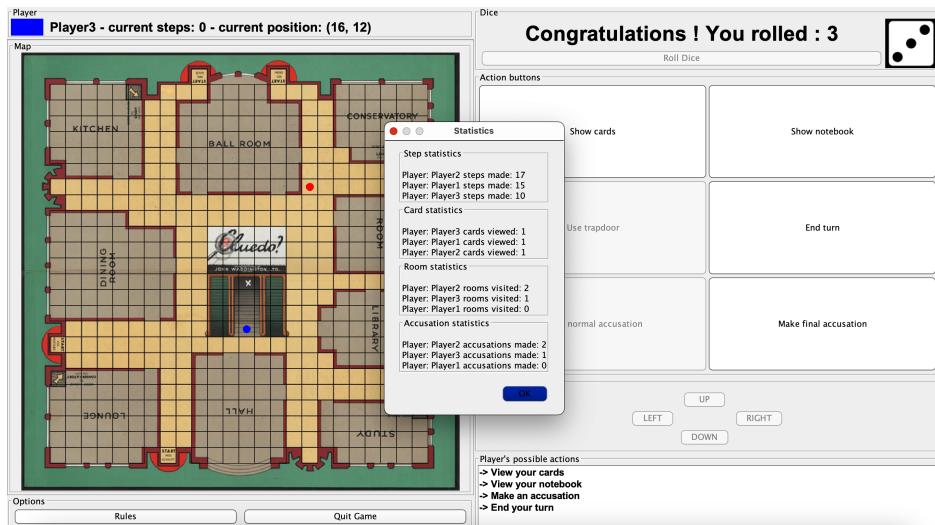


Figura A.8: Statistiche della partita

Durante la partita è sempre possibile visualizzare il regolamento, mediante l'apposito bottone "Rules", posizionato sotto alla mappa di gioco e vicino al bottone "Quit Game", che permette, in ogni momento della partita, di uscire normalmente o salvando lo stato del gioco.