Capitolo 1 - Analisi

1.1 Descrizione e requisiti

Il progetto riguarda verte sulla simulazione del celebre gioco da tavolo Monopoly (soprannominato “java-poly”). Nel gioco tradizionale, i partecipanti competono per accumulare ricchezza attraverso transazioni economiche, acquisto di proprietà e riscossione di affitti, muovendosi su un percorso ciclico determinato dal lancio di dadi.

Il problema principale che il software intende risolvere è la complessità gestionale e la lentezza della versione fisica del gioco. In una partita tradizionale, i giocatori devono calcolare manualmente affitti, gestire resti in banconote, verificare regolamenti su ipoteche e costruzioni, e mantenere traccia dello stato del tabellone. Questo spesso porta a errori di calcolo, interruzioni del flusso di gioco e tempi morti.

L'obiettivo del sistema software è fornire una versione digitale che automatizzi l'applicazione delle regole e la gestione contabile, permettendo ai giocatori di focalizzarsi sulle decisioni strategiche.

Requisiti funzionali

- **Configurazione della partita:** Il sistema deve permettere di impostare una nuova partita o di riprenderne una già iniziata, per un numero variabile di giocatori (da 2 a 4).

- **Gestione del turno**: Il sistema deve garantire il corretto svolgimento dei turni tra i giocatori, gestendo casi particolari come la ripetizione del turno in caso di "doppio" coi dadi o la perdita del turno (es. Prigione).

- **Movimento**: Le pedine dei giocatori devono potersi muovere sul percorso di un numero di passi determinato dalla somma del lancio di due dadi a sei facce.

- **Interazione con le caselle**: Il sistema deve identificare la tipologia della casella di arrivo (Proprietà libera, Proprietà posseduta, Tassa, Imprevisto, Prigione, Via).

- **Gestione Proprietà:** nel caso di proprietà libera, il giocatore deve poter scegliere se acquistarla o meno (in base ai fondi posseduti). Altrimenti in caso di proprietà altrui, il sistema deve calcolare e trasferire automaticamente l'importo dell'affitto dal giocatore corrente al proprietario.

- **Gestione economica:** ogni giocatore deve avere un saldo finanziario aggiornato in tempo reale. Inoltre, il sistema deve rilevare la condizione di bancarotta (saldo negativo non sanabile) ed eliminare il giocatore dalla partita.

- **Gestione Carte:** Il sistema deve simulare la pesca di carte che possono alterare la posizione del giocatore o il suo saldo economico.

- **Vittoria:** Il sistema deve decretare il vincitore quando rimane un solo giocatore attivo (o al raggiungimento di un limite di tempo/turni preimpostato).

Requisiti Non Funzionali

- **Usabilità**: L'interfaccia deve rendere immediatamente visibile lo stato del gioco (posizioni, soldi, proprietà) senza richiedere calcoli mentali all'utente.

- **Estendibilità**: Il modello del dominio deve essere progettato in modo da poter accogliere in futuro nuove regole (es. aste, ipoteche) o nuove tipologie di caselle senza stravolgere la struttura esistente.

- Il sistema deve rispondere **entro 1 secondo** alle azioni dell'utente.

- Il software deve essere **compatibile** con Windows, Linux e MacOS.

1.2 Modello del Dominio

In questa sezione analizziamo le entità che compongono il mondo del gioco java-poly, indipendentemente da come verranno programmate.

Le entità principali identificate sono:

- **Match (Partita):** L'entità aggregatrice che rappresenta lo stato corrente del gioco tiene traccia dei giocatori e di chi è il giocatore di turno.

- **Board (Tabellone):** Rappresenta il percorso di gioco. È composto da una sequenza ordinata e ciclica di Caselle.

- **Tile (Casella):** L'unità minima del tabellone. Può essere di vari tipi (es. Terreno, Stazione, Tassa, Probabilità).

- **Player (Giocatore):** Il partecipante alla partita. È caratterizzato da un nome, un patrimonio (denaro) e un inventario di titoli di proprietà.

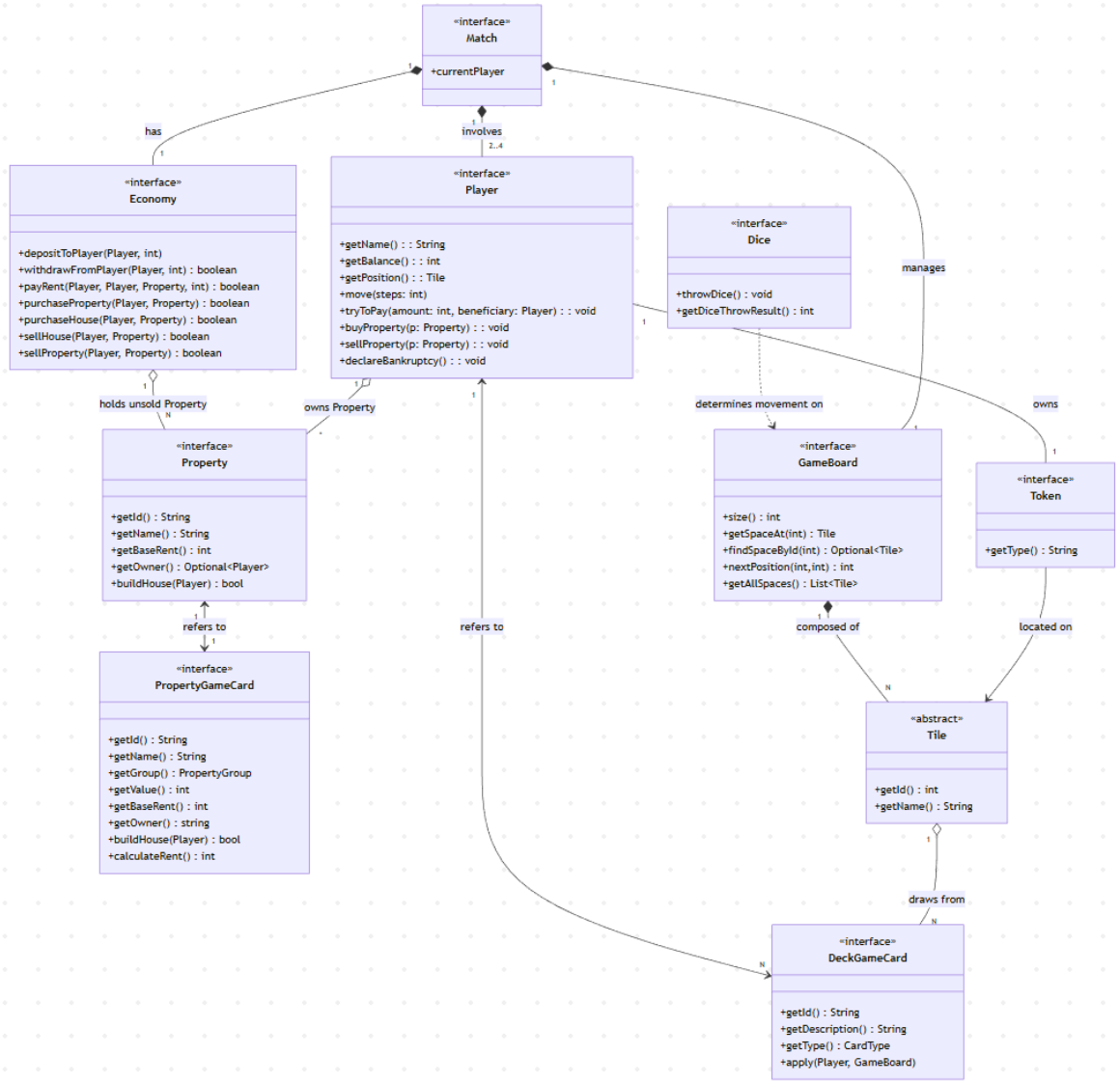
- **Token (Pedina):** La rappresentazione fisica del giocatore sul tabellone. Ha una posizione (riferimento a una Casella). **Nota:** Nel dominio, la Pedina è distinta dal Giocatore (il giocatore *possiede* la pedina).

- **Deed (Titolo di proprietà):** Rappresenta il diritto di possesso su una specifica casella edificabile. Definisce il valore di acquisto e la rendita (affitto).

- **Dice (Dadi):** La fonte di casualità che determina il movimento.

Relazioni principali

- Un **Match** *contiene* 1 **Board** e da 2 a 4 **Player**.
- La **Board** è *composta da* molteplici **Space** (ordinati).
- Un **Player** *possiede* 1 **Token**.
- Il **Token** *si trova su* 1 **Space**.
- Un **Player** *possiede* 0..N **Deed**.
- Un **Deed** *riferisce* a 1 **Space** (di tipo proprietario).


«interface»
Match
+currentPlayer
has
involves
manages
«interface»
Economy
+depositToPlayer(Player, int)
+withdrawFromPlayer(Player, int) : boolean
+payRent(Player, Player, Property, int) : boolean
+purchaseProperty(Player, Property) : boolean
+purchaseHouse(Player, Property) : boolean
+sellHouse(Player, Property) : boolean
+sellProperty(Player, Property) : boolean
«interface»
Player
+getName() : : String
+getBalance() : : int
+getPosition() : : Tile
+move(steps: int)
+tryToPay(amount: int, beneficiary: Player) : : void
+buyProperty(p: Property) : : void
+sellProperty(p: Property) : : void
+declareBankruptcy() : : void
«interface»
Dice
+throwDice() : void
+getDiceThrowResult() : int
holds unsold Property
owns Property
determines movement on
owns
«interface»
Property
+getId() : String
+getName() : String
+getBaseRent() : int
+getOwner() : Optional<Player>
+buildHouse(Player) : bool
«interface»
GameBoard
+size() : int
+getSpaceAt(int) : Tile
+findSpaceById(int) : Optional<Tile>
+nextPosition(int,int) : int
+getAllSpaces() : List<Tile>
«interface»
Token
+getType() : String
refers to
refers to
composed of
located on
«interface»
PropertyGameCard
+getId() : String
+getName() : String
+getGroup() : PropertyGroup
+getValue() : int
+getBaseRent() : int
+getOwner() : string
+buildHouse(Player) : bool
+calculateRent() : int
«abstract»
Tile
+getId() : int
+getName() : String
draws from
«interface»
DeckGameCard
+getId() : String
+getDescription() : String
+getType() : CardType
+apply(Player, GameBoard)

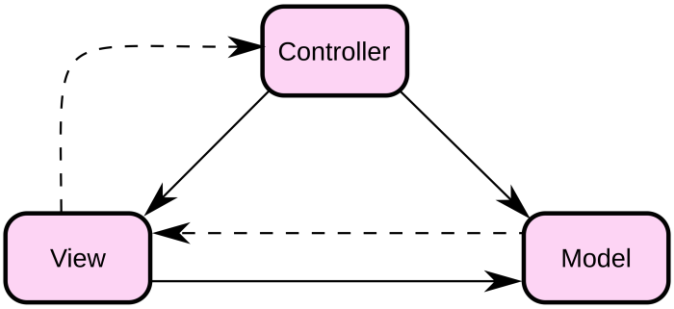
Link UML

Capitolo 2

Design

L'architettura del sistema JavaPoly segue rigorosamente il pattern architetturale **Model-View-Controller (MVC)**. Questa scelta è stata dettata dalla necessità di disaccoppiare la logica di gioco (complessa e ricca di regole) dalla sua rappresentazione grafica, garantendo estendibilità e testabilità.

2.1 Architettura



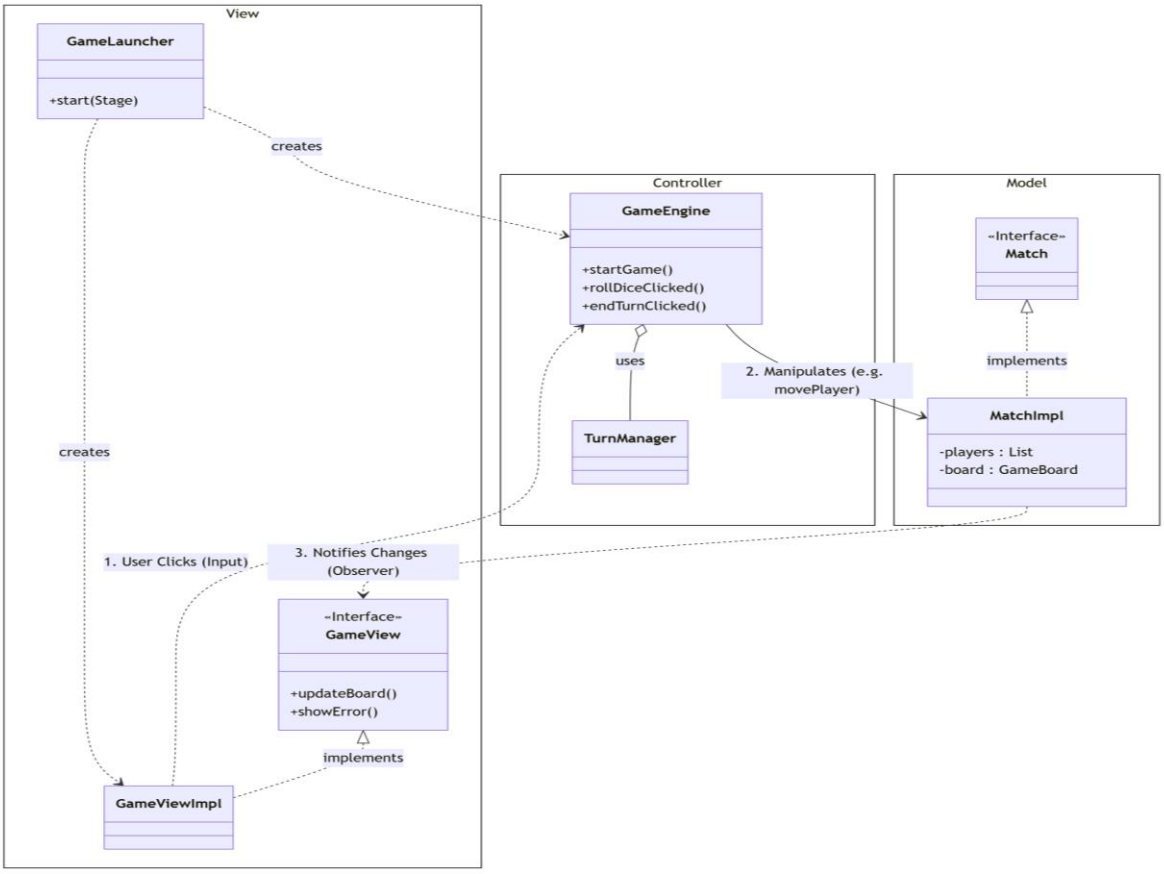
Le responsabilità sono ripartite come segue:

- **Model:** Rappresenta lo stato e la logica di business dell'applicazione. Include le entità Match, Player, GameBoard e Bank. Il Model è completamente all'oscuro dell'esistenza della View e comunica i cambiamenti di stato (es. spostamento di una pedina, variazione del saldo) tramite il pattern **Observer**.

- **View:** Gestisce l'interfaccia utente e la visualizzazione dei dati. Include le classi JavaFX come GameTablePanel e PlayerDashboard. La View osserva il Model e si aggiorna automaticamente alla ricezione delle notifiche.

- **Controller:** Gestisce il flusso dell'applicazione e interpreta gli input dell'utente. Il GameEngine funge da coordinatore, ricevendo i comandi dalla View (es. click sul bottone "Lancia Dadi") e invocando i metodi appropriati sul Model (es. player.move()).

2.2 Design dettagliato



UML del MVC

Turillo Luca - Gestione del Giocatore e delle Pedine:

In questa sezione ho analizzato le scelte progettuali relative alla modellazione dell'entità **Player** e della sua rappresentazione visiva tramite **Token**.

L'obiettivo principale è stato garantire una netta separazione delle responsabilità tra la logica di gioco e la sua rappresentazione visiva, minimizzando le dipendenze reciproche, facilitando l'estendibilità delle regole di movimento e la personalizzazione delle pedine.

Gestione degli stati del giocatore:

Per modellare la complessa logica di gioco che varia dinamicamente in base alla condizione corrente del giocatore, ho adottato il **Pattern State**. Questa architettura permette all'entità *Player* di alterare il proprio comportamento delegando le operazioni all'oggetto che corrisponde al proprio stato attivo.

Problema: Il comportamento di un giocatore varia notevolmente in base al suo stato corrente. Un giocatore libero può muoversi e acquistare proprietà, un giocatore in prigione vede limitata la sua capacità di movimento pur potendo interagire con il mercato, un giocatore in bancarotta diventa inattivo. Gestire queste variazioni comportamentali attraverso una serie di flag booleani (es. *isInJail*, *isBankrupt ecc.*) all'interno della classe *Player* avrebbe portato a un codice fragile, caratterizzato da numerose istruzioni condizionali *if-else* duplicate in metodi come *move()* o *playTurn()*.

Soluzione: Ho scelto di adottare il **Pattern State** incapsulando il comportamento dipendente dallo stato in un'interfaccia dedicata *PlayerState*. La classe *PlayerImpl* funge da *Context* del Pattern State: essa implementa l'interfaccia *Player*, garantendo che la complessità della gestione degli stati rimanga completamente incapsulata e invisibile al resto del sistema, rispettando il principio di **Dependency Inversion** e delegando allo stato corrente l'esecuzione della logica di turno.

Ho previsto tre implementazioni concrete:

1. **FreeState**: Implementa la logica standard di movimento e interazione.
2. **JailedState**: Gestisce la logica di permanenza e uscita dalla prigione.
3. **BankruptState**: Implementa un comportamento nullo per i giocatori eliminati.

Questa architettura rispetta l'**Open/Closed Principle**: l'introduzione di nuovi stati futuri (es. *Sospeso o Non-Inizializzato*) non richiederà modifiche al *Player*.

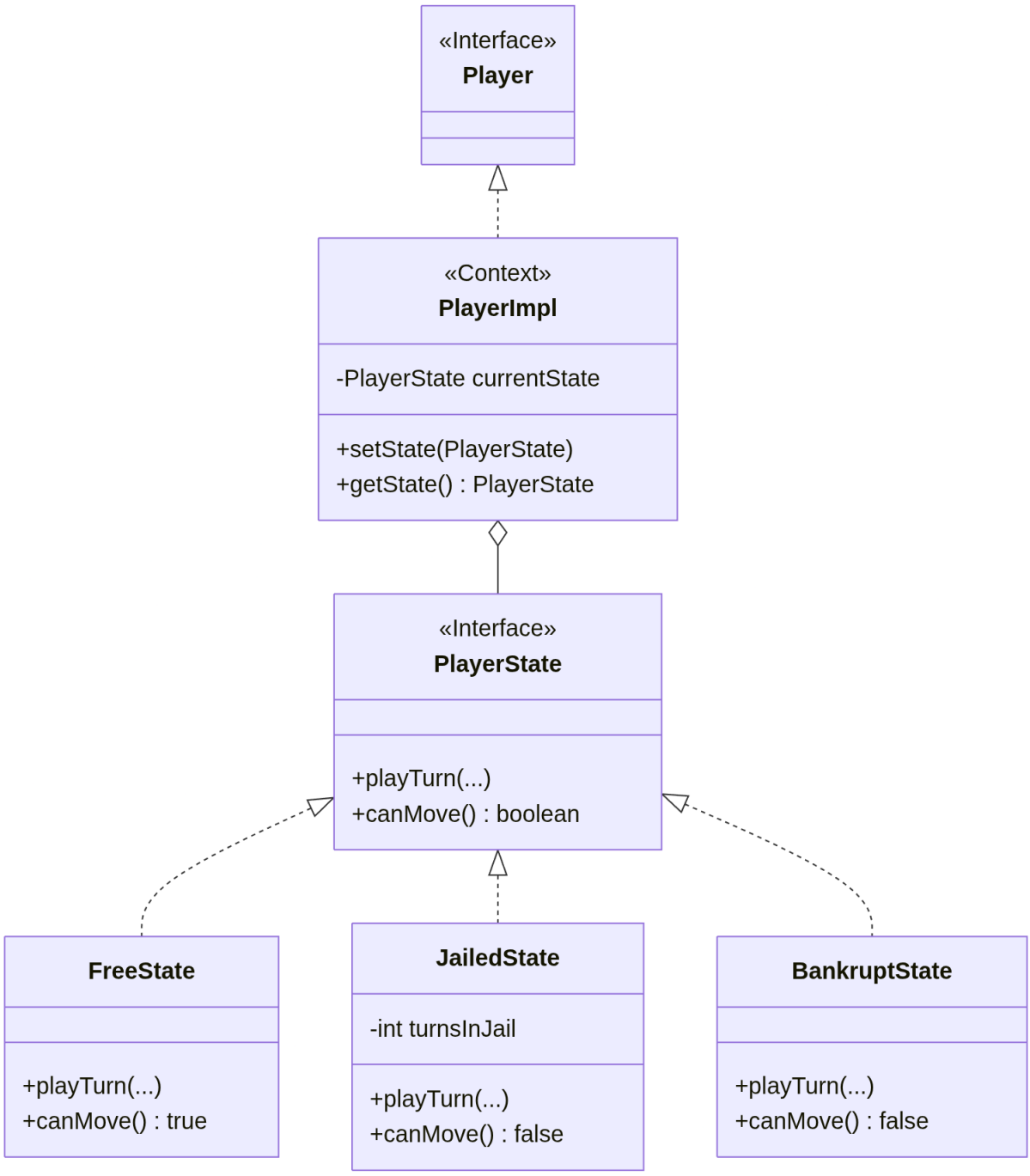


Figura 2.2.1:

Rappresentazione UML del **Pattern State** per la gestione dello stato del *Player*.

Creazione delle pedine:

Per la creazione delle pedine di gioco, i *Token*, ho scelto di adottare il **Pattern Factory Method**. Questa scelta permette di disaccoppiare la logica di creazione degli oggetti dalla loro rappresentazione visiva e utilizzo, garantendo che l'aggiunta di nuove tipologie di pedine non impatti sul codice esistente.

Problema: Il sistema prevede diverse tipologie di pedine (es. Macchina, Cappello, Cane ecc.), ciascuna con risorse grafiche specifiche. Creare queste pedine direttamente nel codice che avvia la partita renderebbe il sistema rigido, per aggiungere o rimuovere una pedina dovrei modificare la logica principale del gioco, violando il **Single Responsibility Principle**.

Soluzione: Ho scelto di applicare il **Pattern Factory Method**. La classe *TokenFactory* centralizza la logica di creazione, nascondendo al *Client* i dettagli di creazione. Quando il *Client* richiede una pedina, specificando un solo identificativo (Enum), la Factory restituisce l'istanza corretta del *Token*.

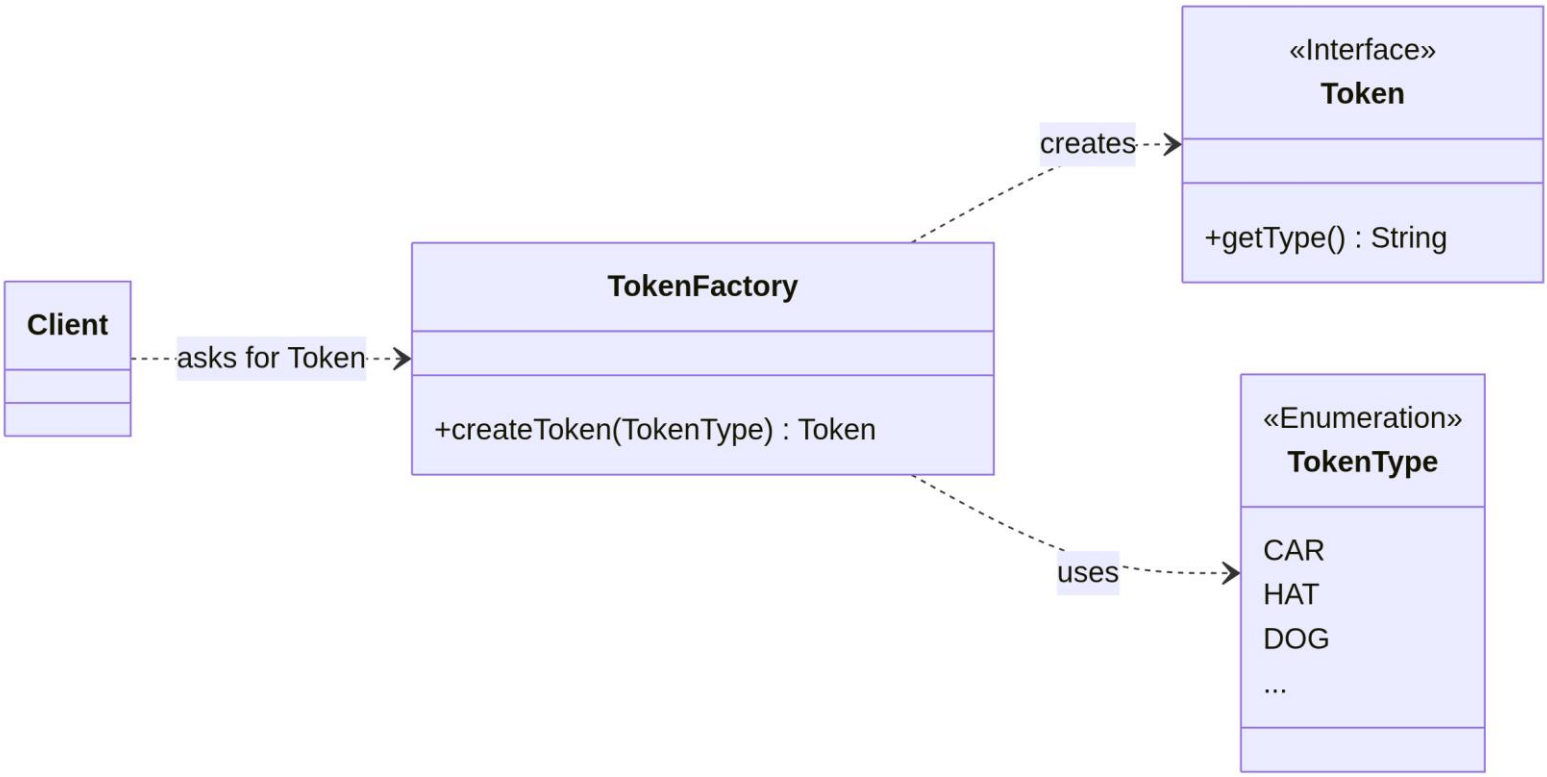


Figura 2.2.2:

Rappresentazione UML del **Pattern Factory Method** per la gestione di creazione delle pedine.

Gestione della comunicazione Model-View:

Per garantire una corretta separazione delle responsabilità secondo l'architettura MVC e permettere alla GUI di reagire in tempo reale ai cambiamenti del *Model* senza introdurre dipendenza, ho scelto di implementare il **Pattern Observer**.

Problema: Il giocatore subisce frequenti variazioni di stato (movimento sul tabellone, transazioni economiche, cambio di stato logico). La *View* e il *Controller* necessitano di essere informati di questi eventi per aggiornare l'interfaccia utente o innescare regole di gioco, ma il *Player*, che è nel *Model* non deve possedere riferimenti diretti a componenti grafici o controller.

Soluzione: Ho fatto in modo tale che il *Player* agisse come **Observable Subject,** ovvero un soggetto osservabile, mentre la *View* e il *Controller* agissero come **Observer,** ovvero come osservatori del soggetto.

Ho creato l'interfaccia *PlayerObserver*, che espone tre metodi specifici per gestire eventi di cambiamento dello stato:

1. ***onPlayerMoved****(Player, oldPos, newPos)*: Notifica lo spostamento, fornendo sia la vecchia che la nuova posizione permettendo alla *View* di calcolare animazioni fluide e al *Controller* di verificare l'attraversamento di determinate caselle.

2. ***onBalanceChanged(****Player, newBalance)*: Notifica il cambiamento del saldo del *Player*, evitando che la GUI debba fare polling al *model*.

3. ***onStateChanged****(Player, PlayerState old, PlayerState new)*: Notifica il cambiamento di stato gestito dal Pattern State , permettendo alla GUI di fornire riscontri visivi immediati.

Vantaggi ottenuti:

- **Single Responsibility Principle:** Il *Player* fa solo il giocatore, non si mette a disegnare nella *View* e ignora completamente l'esistenza della GUI.
- **Open/Closed Principle:** È possibile aggiungere nuovi osservatori senza modificare il *Player*.
- **Reattività:** La GUI è sempre sincronizzata con il dato reale.

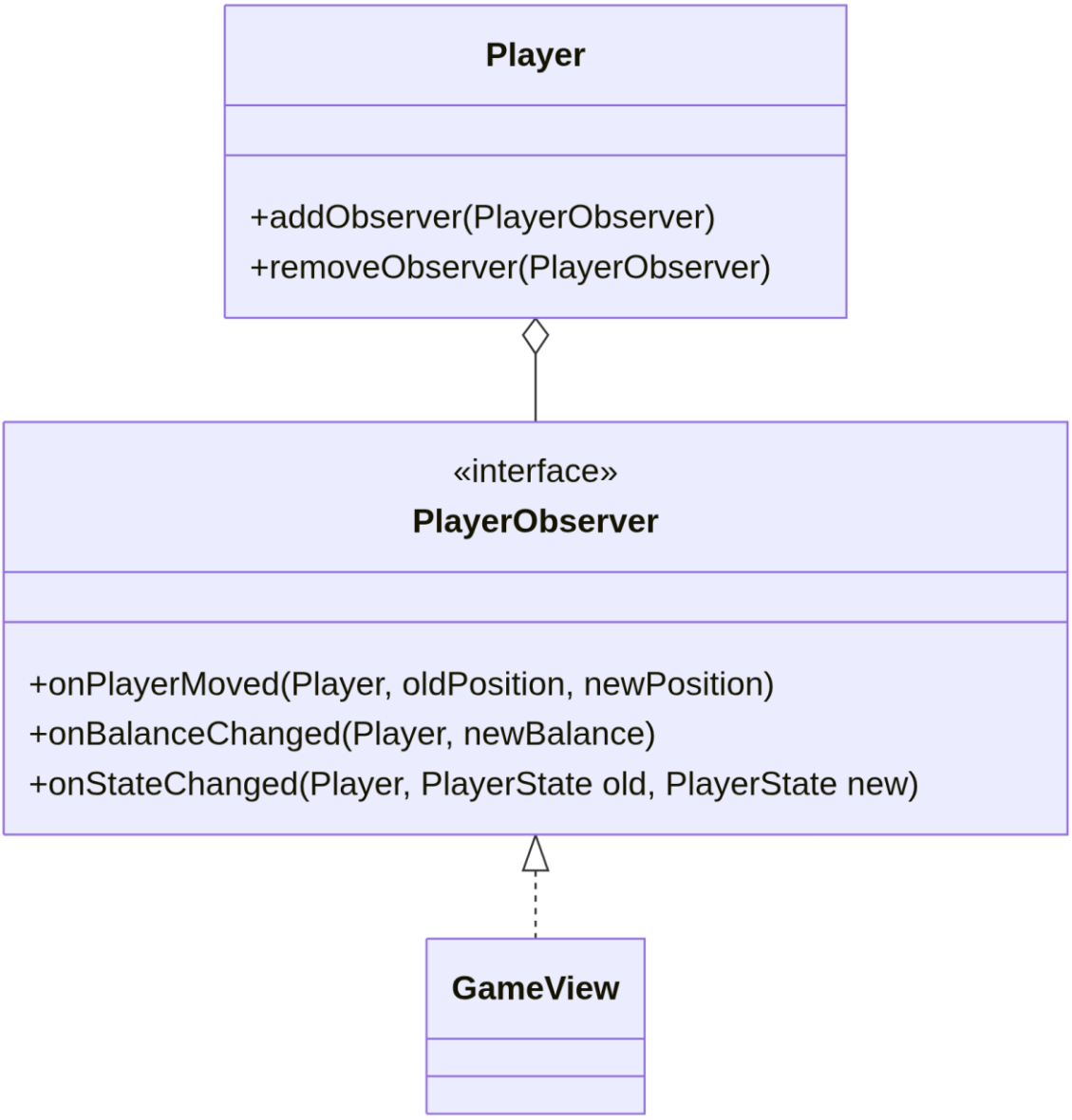


Figura 2.2.3:

Rappresentazione UML del **Pattern Observer** per la gestione delle notifiche ad ogni cambiamento di stato del *Player*.

2.2 Caravita Francesco Gestione Tabellone, proprietà e carte

In questa sezione vengono analizzate le scelte progettuali relative alla gestione del tabellone di gioco e delle sue componenti principali, includendo il movimento sulle caselle, la gestione delle diverse tipologie di *tile* e l'interazione con le carte e le proprietà.

L'obiettivo principale è stato modellare la logica di avanzamento e di interazione sul tabellone in modo coerente e modulare, garantendo una chiara separazione tra il coordinamento delle operazioni di gioco e la rappresentazione dello stato. Questo approccio consente di facilitare l'estendibilità delle regole e l'evoluzione delle meccaniche di gioco.

Entita proprieta:

Nel gioco Monopoly, le caselle di tipo **Proprietà** rappresentano elementi acquistabili dai giocatori, che generano rendita in base a regole specifiche.

Ogni proprietà deve gestire sia informazioni statiche (immutabili per tutta la durata della partita) sia informazioni dinamiche (che cambiano durante il gioco, come il proprietario o il numero di costruzioni presenti).

Problema:

Nel dominio del gioco, ogni casella di tipo "Proprietà" deve rappresentare due aspetti distinti ma correlati:

Dati Fissi (immutabili):

- identificativo,
- posizione sul tabellone,
- carta descrittiva contenente i parametri di costo e le regole di calcolo dell'affitto (AbstractPropertyCard).

Dati Variabili (mutabili):

- proprietario,
- numero di case costruite,
- presenza di un hotel.

Soluzione:

Sono state definite due interfacce distinte:

- Property, dedicata ai dati non mutabili;

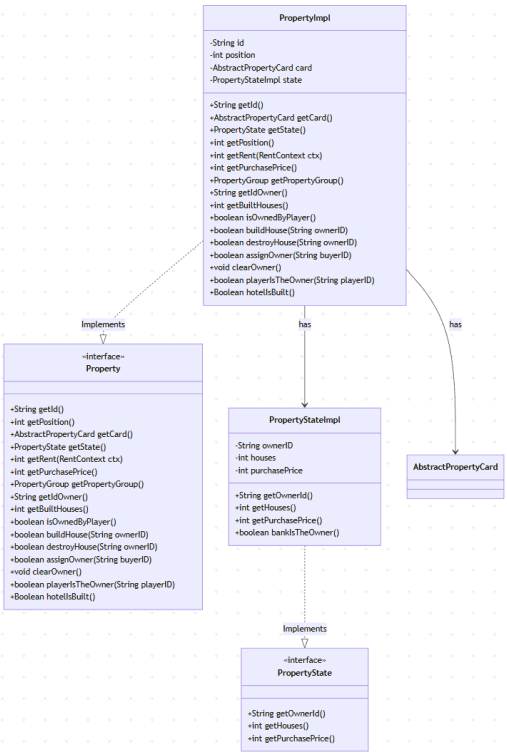
- PropertyState, dedicata ai dati variabili.

L’implementazione concreta di Property gestisce tutte le operazioni relative ai dati fissi e delega la gestione dei dati variabili alla classe concreta di PropertyState.

PropertyState si occupa quindi di:

- restituire i valori correnti (proprietario, numero di case, hotel, ecc.),
- modificarli quando necessario, in base alle regole di gioco.

All’interno di Property vengono inoltre salvate le **carte proprietà**, che fungono da contenitori dei dati descrittivi e delle regole associate a quella specifica proprietà. Questa separazione consente una migliore organizzazione del modello di dominio, favorendo chiarezza, manutenibilità ed estendibilità del sistema.



UML-Property

Archiviazione dati e calcolo rent delle proprietà:

Per salvare tutti i dati relativi alle varie proprietà (ad esempio costo della proprietà, costo di costruzione di una casa, ecc.), ho deciso di creare una classe specifica per ogni tipologia di proprietà esistente (Terreni, Stazioni, Utility).

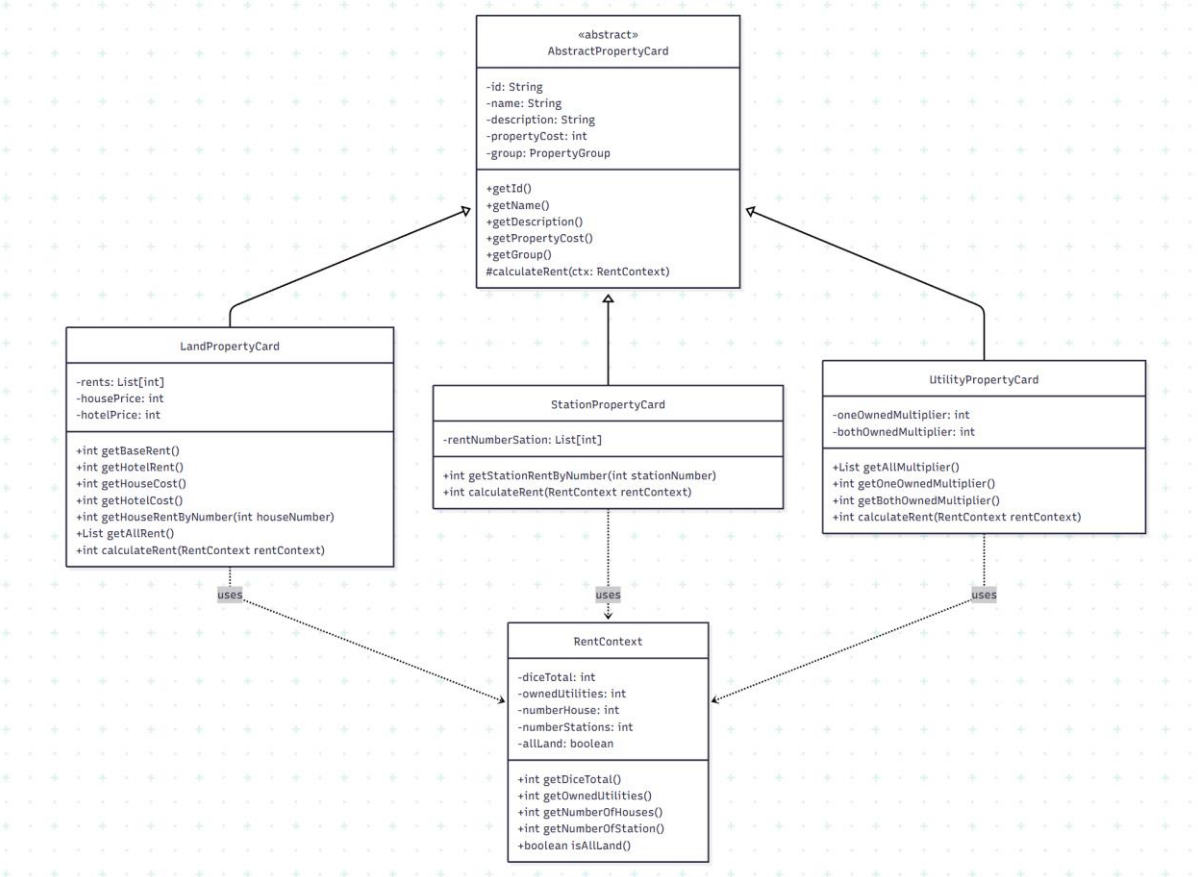
Il termine *card* deriva dal fatto che, nel gioco originale cartaceo, i dati sono riportati su apposite carte.

Problema:

Nel dominio del problema, le diverse *property card* (Land, Station, Utility) condividono la medesima procedura per determinare l'affitto (validazioni, raccolta dei parametri, logging), ma differiscono nella formula specifica di calcolo.

Soluzione:

Ho definito una classe astratta che incapsula lo scheletro dell'algoritmo (validazioni + invocazione del calcolo specifico) e delega alle sottoclassi l'implementazione della parte variabile, ovvero il metodo calculateRent. Inotre ho creato una classe chiamata RentContext dove al suo interno vengono salvare tutte le informazioni centrali per il calcolo del rent.



Link UML

Tabellone e caselle:

Abbiamo due entità principali: **Tabellone** e **Casella**.
La casella non è un elemento unico: nel gioco classico ce ne sono 40, ognuna con caratteristiche e comportamenti specifici.

Problema:

Il tabellone è composto da diverse *tile* (ad esempio: Start, Property, Tax, Jail, GoToJail, FreeParking, Unexpected).

È necessario rappresentare ciascun tipo di casella con il proprio comportamento specifico (ad esempio: calcolo della tassa, gestione della proprietà, spostamento verso un'altra casella), in modo chiaro, coerente ed estendibile.

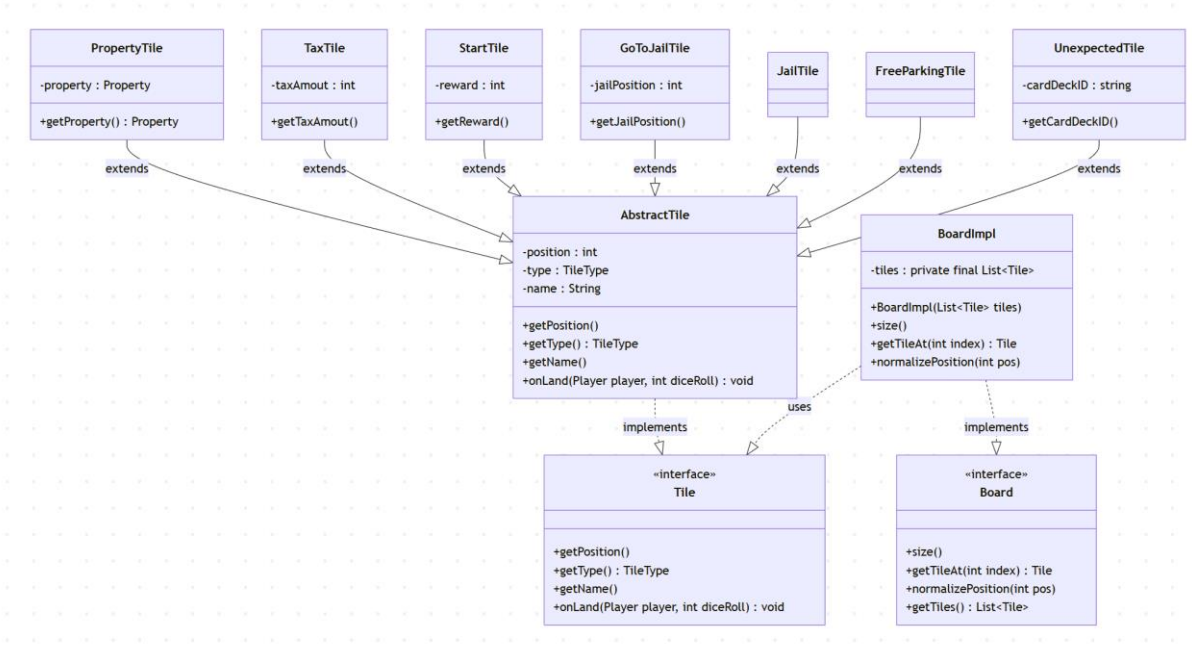
Soluzione:

È stata definita un'interfaccia Board, la cui implementazione contiene una List<Tile> chiamata tiles e fornisce l'accesso alle caselle del tabellone.

È stata inoltre definita un'interfaccia Tile, insieme a un enum TileType che rappresenta tutti i tipi di caselle esistenti.

Sotto l'interfaccia Tile è stata creata una gerarchia di classi concrete. Alla base troviamo la classe astratta AbstractTile, che fornisce le proprietà comuni (posizione, nome, tipo).

Le caselle specifiche estendono il comportamento di AbstractTile.



UML-Board-Tile

Sistema Gestione carte imprevisti e probabilita:

All'interno del gioco Monopoly sono presenti 16 carte di tipo **Imprevisti** e 16 carte di tipo **Probabilità**.

I due mazzi sono stati unificati in un unico mazzo denominato **Imprevisti**.

Ciascuna carta produce un effetto diverso sul giocatore, a seconda della carta pescata.

Problema

Il gioco prevede carte con effetti differenti (pagamenti, spostamenti assoluti, movimenti relativi, "Get Out of Jail Free", ecc.).

È necessario un meccanismo che consenta di:

- rappresentare le carte in modo coerente,
- applicarne correttamente l'effetto al giocatore e/o al tabellone,
- rendere il sistema facilmente estendibile,
- gestire e salvare correttamente le carte e il loro stato (mazzo, scarti, ecc.).

Soluzione:

Sono state definite le seguenti interfacce generali:

- GameCard
- CardDeck

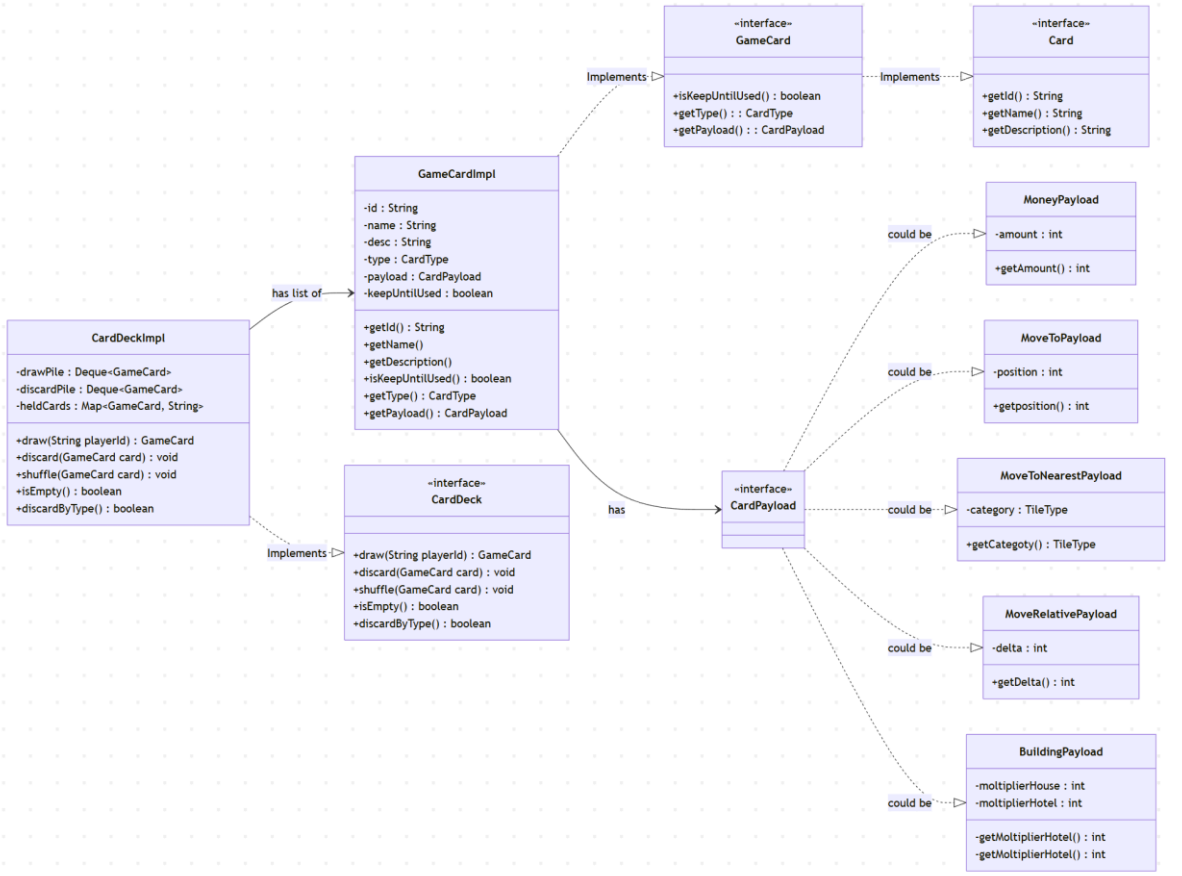
È stato inoltre definito un enum contenente tutti i tipi di carte presenti nel gioco.

Per ogni tipologia di carta è stato creato uno specifico **payload** (ad esempio MoneyPayload, MoveToPayload, ecc.), responsabile della logica dell'effetto associato.

I vari payload sono associati alle implementazioni concrete delle carte (GameCardImpl).

Le istanze di GameCardImpl sono raccolte all'interno di CardDeckImpl, che simula a tutti gli effetti un mazzo di carte, fornendo metodi come: draw(), discard(), ecc.

In questo modo la gestione delle carte risulta modulare, estendibile e coerente con i principi di separazione delle responsabilità.



UML-GameCard-CardDeck

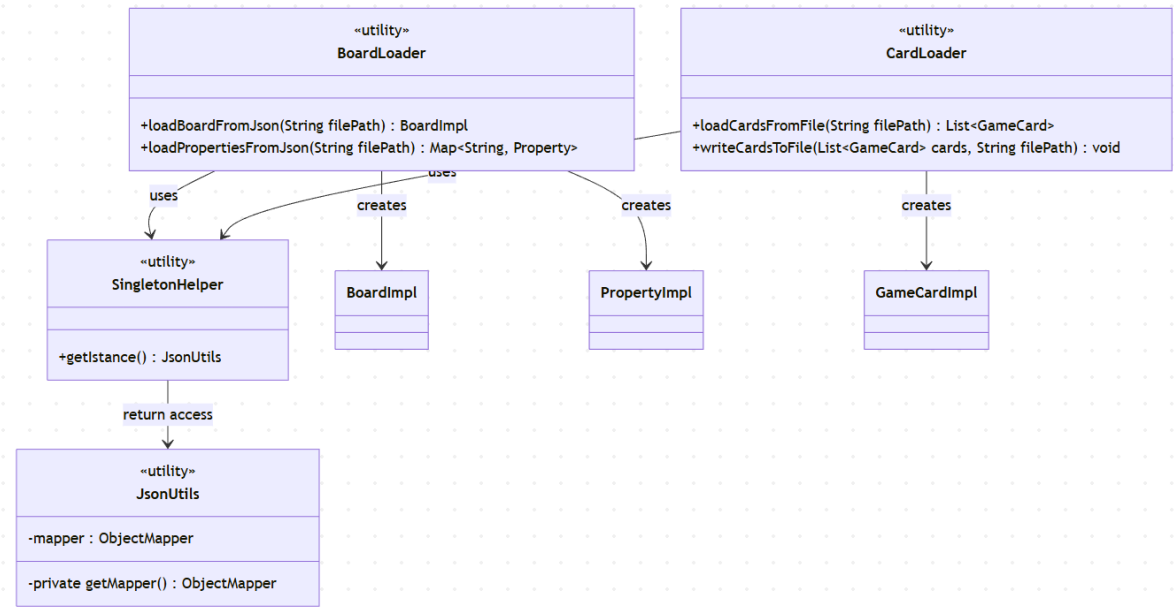
Gestione Serializzazione Deserializzazione Json

Problema:
Il gioco deve essere inizializzabile a partire da file JSON (board, cards, proprietà). Serve un modo coerente e centralizzato per leggere/scrivere questi oggetti rispettando il model domain.
Soluzione:

Ho scelto la libreria Jackson, ho creato una classe JsonUtils che fornisce un ObjectMapper configurato e un metodo mapper() che ritorna una copia della configurazione predefinita, Applicando il pattern Singleton.

BoardLoader e CardLoader sono classi utility con metodi statici che caricano JSON e producono mappe/collezioni di oggetti di dominio (Board, GameCard, Property) usando Jackson.

Le entità del model (es. AbstractPropertyCard e sottoclassi) usano annotazioni Jackson per abilitare la deserializzazione polimorfica (@JsonTypeInfo, @JsonSubTypes).



UML-JsonUtils

Piedimonte Antonio - Architettura del Controller e Gestione dell'Interfaccia Reattiva

Il mio contributo si è concentrato sulla progettazione del core logico-gestionale (**MatchController**) e dell'architettura della **View**, con l'obiettivo di garantire un sistema estensibile, manutenibile e thread-safe.

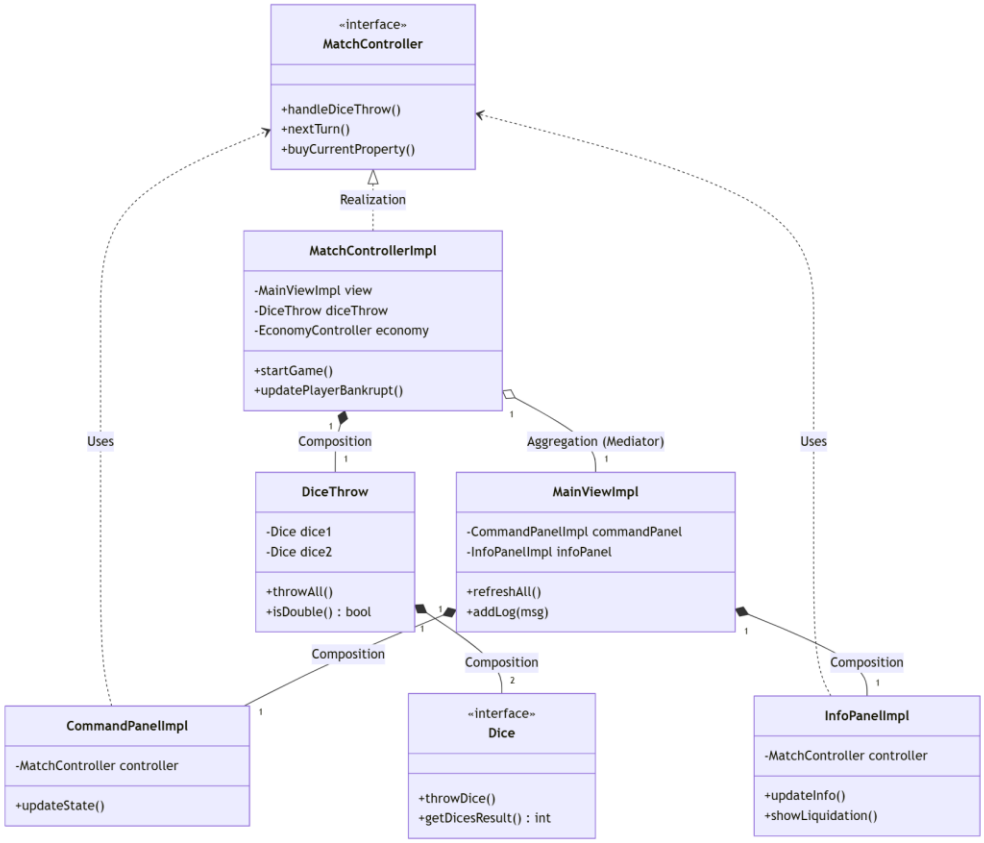


Figura [1]: Diagramma delle classi relativo all'architettura del controller e dei moduli della View. Il diagramma evidenzia l'adozione del **Pattern Mediator** tramite MatchControllerImpl, che centralizza le dipendenze tra la logica di business e i componenti grafici (MainViewImpl). Si nota inoltre l'uso della **composizione** per il sottosistema dei dadi, dove DiceThrow agisce come astrazione di alto livello per le istanze dell'interfaccia Dice.

1. Orchestrazione del Dominio (Pattern Mediator & Observer)

Per gestire l'interazione tra i moduli (economia, proprietà, movimento), ho implementato il **Pattern Mediator** tramite MatchControllerImpl. Invece di permettere alla View di accedere direttamente alla logica interna, il Controller funge da unico punto di accesso (**Facade**). Inoltre, estendendo PlayerObserver, ho applicato un meccanismo di **Inversion of Control (IoC)**: il controller reagisce ai cambiamenti del modello (es. variazione del saldo) e notifica automaticamente la View, mantenendo la logica di business isolata dalla rappresentazione.

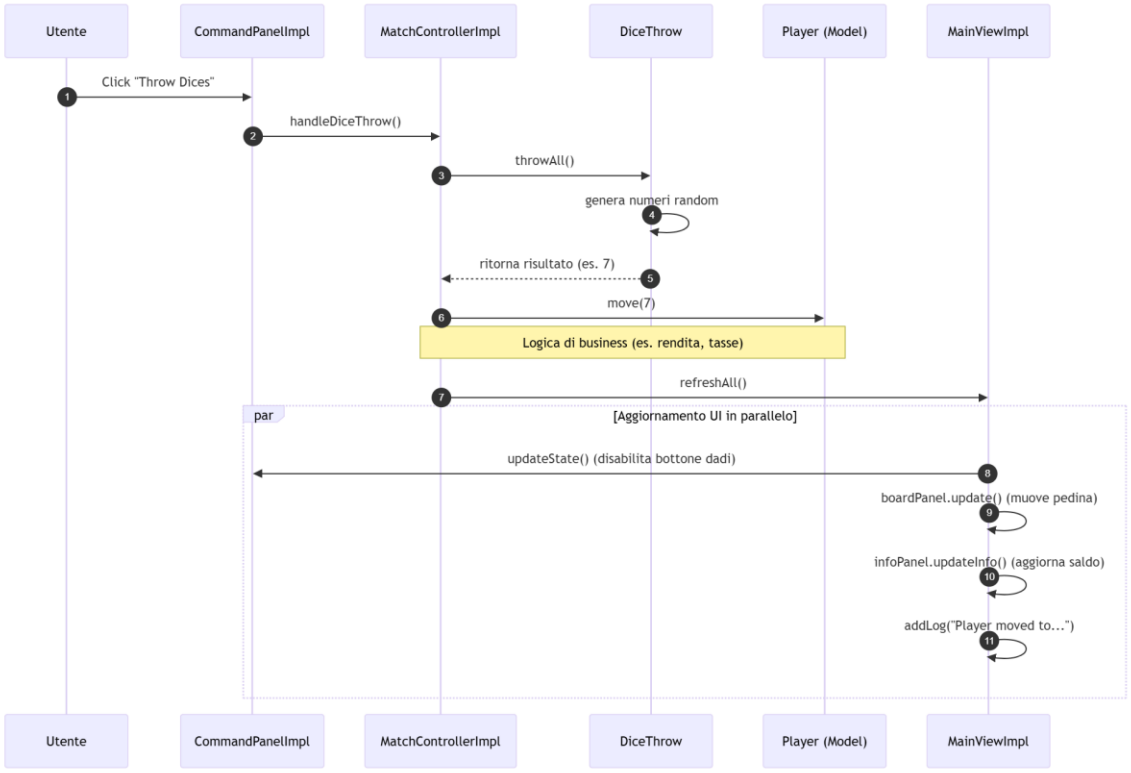


Figura [2]: Diagramma di sequenza che illustra la dinamica di un evento di gioco tipico (lancio dei dadi). Il flusso evidenzia il **disaccoppiamento** tra l'input dell'utente (scatenato nel CommandPanelImpl) e l'aggiornamento del modello. Il MatchController coordina la sequenza garantendo che il refresh dei componenti della View (refreshAll) avvenga solo dopo la mutazione dello stato del sistema, assicurando la coerenza tra logica e rappresentazione.

2. Modellazione della Casualità e Persistenza

Il sottosistema dei dadi separa la generazione casuale dalla logica di gioco. Tramite DiceThrow, ho centralizzato la gestione dei lanci doppi e dei turni extra. Per la **persistenza**, e' stata configurata la serializzazione JSON con **Jackson**. La sfida principale è stata la gestione di oggetti complessi e contatori (come jailTurnCounter): sono stati utilizzati @JsonCreator e @JsonProperty per definire costruttori di ricostruzione precisi, e @JsonIgnore per evitare ridondanze nei metodi derivati.

3. Architettura della View e Feedback Dinamico

La MainViewImpl è stata concepita come orchestratore di pannelli specializzati, favorendo la **Separation of Concerns**:

Interfaccia Operativa (CommandPanelImpl): Gestisce l'input utente e include la logica di salvataggio sessione (saveStateGame()). Ho implementato una gestione del

file system **cross-platform** tramite System.getProperty("user.home"), garantendo la portabilità del file di salvataggio.

Gestione degli Stati UI: Il metodo updateState() agisce come una macchina a stati che abilita/disabilita i bottoni in base al contesto (es. fase di liquidazione). Per ottimizzare il layout, ho combinato setVisible e setManaged, permettendo all'interfaccia di riadattarsi dinamicamente alla scomparsa dei pulsanti non pertinenti.

Visualizzazione Dinamica: L'InfoPanelImpl gestisce la visualizzazione dei player tramite card generate a runtime, utilizzando styling CSS dinamico per fornire feedback immediato sull'avanzamento dei turni.

Mularoni Luca – Coordinamento delle operazioni economiche

In questa sezione vengono analizzate le scelte che si basano sull’economia di Monopoly che comprende molteplici operazioni come: trasferimenti di denaro, acquisto/vendita di proprietà, costruzione di edifici, pagamento di affitti.

Problema:
Il sistema di gestione economica e delle proprietà del Monopoly coinvolgono molteplici sottosistemi complessi:

- Gestione delle transazioni economiche Bank.

- Gestione delle proprietà attraverso PropertyController.
- Costruzione di case.
- Pagamento di affitti

L'esposizione diretta di questi componenti introdurrebbe un'eccessiva complessità all'interno del MatchController e delle View, rendendo il sistema rigido, difficile da utilizzare e oneroso da mantenere.

Soluzione:

Si è optato per ragruppare dei componenti interni attraverso la classe EconomyController, la quale fornisce un'interfaccia semplificata per interagire con i sottosistemi complessi. Questo controller nasconde i dettagli implementativi delle operazioni, coordinando internamente classi come Bank e PropertyManager. Nello specifico, EconomyController esporrà unicamente i metodi necessari che fanno un uso congiunto di Bank e PropertyManager.

Vantaggi:

1. Interfaccia semplificata: Operazioni complesse esposte tramite metodi singoli.
2. Disaccoppiamento: il client non dipende dalle implementazioni dei sottoinsiemi.
3. Manutenibilità: Modifiche ai sottosistemi non impattano i client.

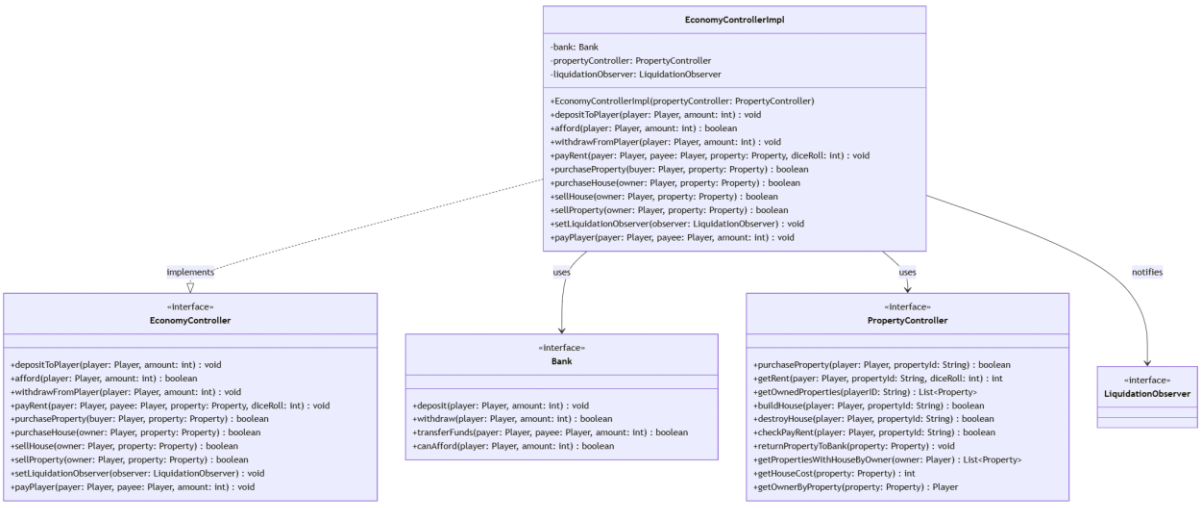


Figura 2.2.11

Rappresentazione del UML della gestione dell'economia

In dettaglio: L'implementazione EconomyControllerImpl coordina al suo interno:

- Bank (BankImpl) per tutte le operazioni monetarie.
- PropertyController per la gestione e l'assegnazione delle proprietà.

Sistema di liquidazione

Problema:

L'EconomyController deve notificare la View quando un giocatore non possiede fondi sufficienti per saldare un debito e deve, conseguentemente, vendere proprietà o edifici. La View deve essere aggiornata in tempo reale in base alle decisioni del giocatore, mantenendo un rigoroso rispetto del pattern architetturale MVC (evitando quindi che il Controller dipenda direttamente o conosca la View).

Soluzione:

Si è scelto di implementare il Pattern Observer. Quando il giocatore, all'interno di EconomyController, non dispone di sufficiente liquidità, il LiquidationObserver entra in azione e aggiorna la View in base alle scelte intraprese dall'utente.

Flusso di esecuzione:

1. L'EconomyController rileva l'insufficienza di fondi sul conto del giocatore.
2. Viene notificato il LiquidationObserver registrato, tramite il metodo onInsufficientFunds().
3. L'Observer coordina la View per mostrare la schermata dedicata (SellAssetView).
4. Il giocatore interagisce con la View per procedere alla vendita degli asset.
5. La View utilizza una callback (liquidationCallBack) per comunicare l'esito al livello logico.
6. L'Observer gestisce la conclusione del processo (successo e ripianamento del debito, oppure bancarotta).

Vantaggi:

1. Disaccoppiamento completo tra la logica di controllo (Controller) e l'interfaccia grafica (View).

2. Estensibilità: è possibile aggiungere nuovi Observer in futuro senza dover modificare i sottosistemi esistenti.

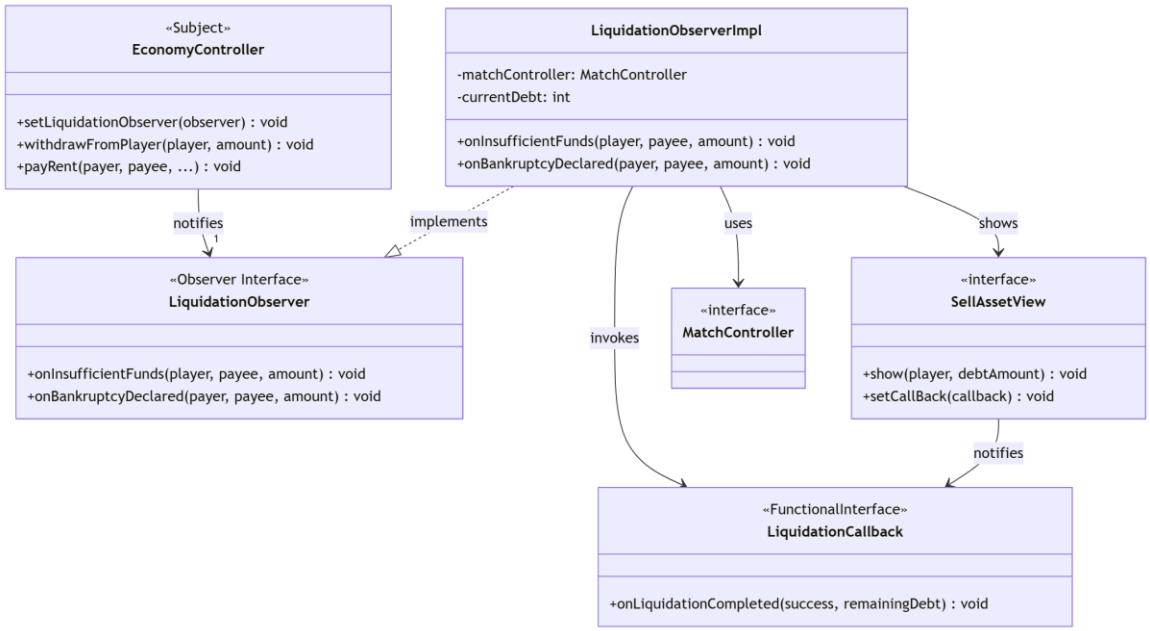


Figura 2.2.12

Rappresentazione del UML del pattern Observer per la comunicazione della Liquidazione con la View

Salvataggio e caricamento del gioco

Problema:

Il sistema deve garantire all'utente la possibilità di salvare la partita in corso e di caricarla in un momento successivo, ripristinando fedelmente l'esatto stato di gioco.

Soluzione:

Per la serializzazione e il salvataggio degli oggetti si è scelto di utilizzare il formato JSON, avvalendosi della libreria esterna Jackson. A tale scopo, è stata implementata MatchControllerDeserializer, una classe di utilità che fornisce il metodo deserialize(File). Questo metodo si occupa di ricostruire in memoria l'istanza di MatchControllerImpl a partire dal file JSON di salvataggio.

Per facilitare le operazioni di I/O, si è fatto uso delle JsonUtils fornite nel progetto Java. Inoltre, le entità collegate al MatchControllerImpl impiegano le apposite annotazioni di Jackson per abilitare e configurare correttamente la serializzazione.

Dinamica di salvataggio:

1. Il file di salvataggio può essere generato manualmente dall'utente, tramite il pulsante Save situato nel CommandPanelImpl, oppure in maniera automatica al passaggio del turno (azione Next Turn).
2. Il CommandPanelImpl chiama matchController.serialize() che:
    - Prende l'istanza JsonUtils configurata che può restituire un mapper
    - Serializza lo stato completo (esclusi i campi con @JsonIgnore)
    - Salva nel file "javapoly_save.json " nella root dell'utente.

Flusso di caricamento:

1. MenuController chiama loadGame(File)
2. MatchControllerDEserializer.deserialize(file):
    a. Legge file JSON
    b. Estrae il nodo "MatchControllerIMpl"
    c. Deserializza usando costruttore @JsonCreator
3. Otterremo un MatchController pronto per iniziare il gioco

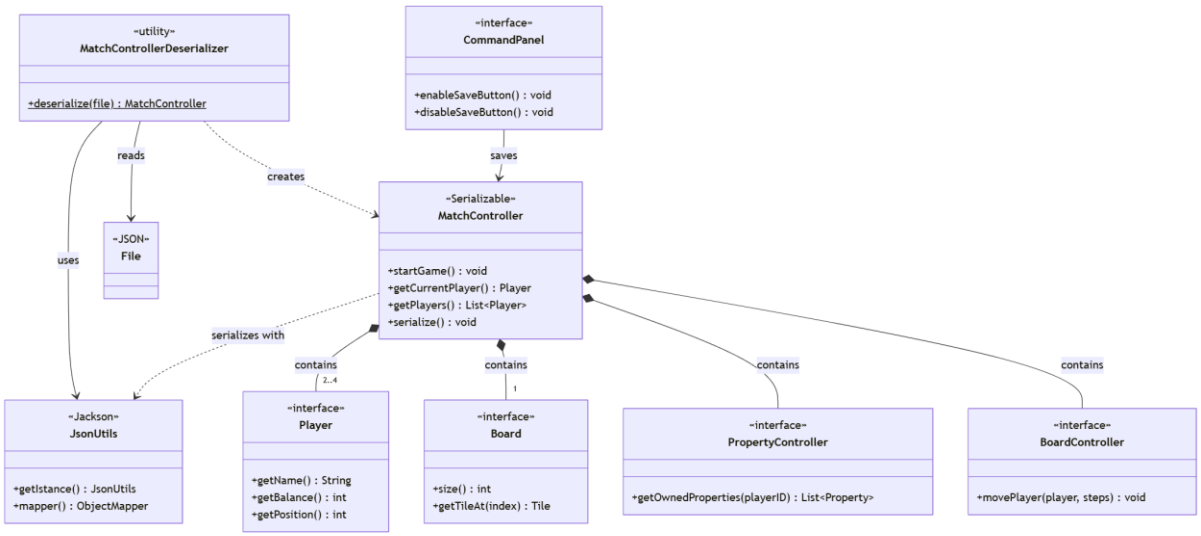


Figura 2.2.13

Rappresentazione Uml per il salvataggio del gioco

Sistema Menu

Problema:

Il sistema necessita di gestire l'inizializzazione del gioco attraverso due modalità distinte:

- Nuova partita: Raccoglie input utente (nomi giocatori, token) e configurare lo stato iniziale
- Carica partita: Deserializzare uno stato salvato e ripristinare la partita

Inoltre, è necessario coordinare le transizioni tra le interfacce MenuView, PlayerSetupView, MainView) gestendo ilo spostamento dei dati presi in input.

Soluzione:

Il **MenuController** agisce come coordinatore centrale che:

1. **Gestisce le interfacce**: Controlla la navigazione tra MenuView, PlayerSetupView e la vista principale del gioco.
2. **Raccoglie e valida input**: Processa dati provenienti da diverse fonti (form utente in PlayerSetupView, file JSON).
3. **Inizializza il gioco**: Una volta raccolti tutti i dati necessari, crea e configura il MatchController.
4. **Delega il controllo**: Passa il comando al MatchController e carica l'interfaccia principale della partita.

Vantaggi:

- Separazione responsabilità: Il MenuController si occupa solo dell'inizializzione e il MatchController gestisce la partita.
- Prima validazione: Tutti gli input passano dal MenuController prima di essere dati ad altri componenti.
- Il cambio interfaccia gestito dal MenuController.

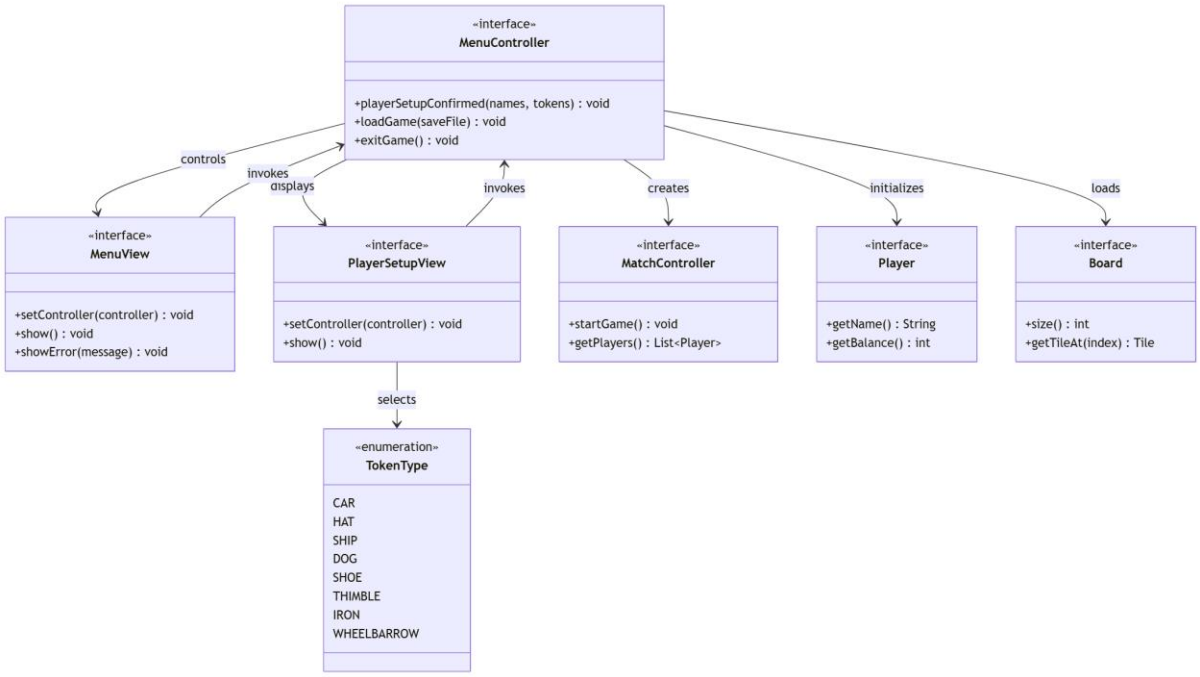


Figura 2.2.14

Uml che spiega il collegamento delle interfacce

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per garantire la robustezza del codice e prevenire regressioni durante lo sviluppo delle nuove funzionalità, è stata realizzata una suite di test unitari completamente automatizzati utilizzando il framework **JUnit 5**.

La strategia di testing si è focalizzata sui componenti critici per la logica di dominio e per la validazione dei dati:

- **Utility di Validazione *ValidationUtilsTest*:** È stata testata esaustivamente la classe *ValidationUtils*, che rappresenta il nucleo della programmazione difensiva del progetto. I test verificano che vengano sollevate le corrette eccezioni quando vengono passati input non validi, come stringhe vuote, riferimenti nulli o valori numerici fuori dai range consentiti.
- **Logica del Giocatore *PlayerImplTest*:** Sono stati implementati test per verificare il corretto comportamento del *PlayerImpl*. In particolare, i test automatici coprono:
  - **Inizializzazione:** Verifica del saldo iniziale, della posizione di partenza e dell'assegnazione corretta del *TokenType*.
  - **Movimento:** Controllo della logica di aggiornamento della posizione sul tabellone.
  - **Gestione Finanziaria:** Verifica delle operazioni di addebito e accredito, assicurando che il saldo non diventi inconsistente.
  - **Token Custom:** Verifica specifica per i costruttori che supportano i token personalizzati, controllando che il percorso del file immagine venga memorizzato correttamente quando si utilizza il tipo *TokenType.CUSTOM*.
- **Logica della Banca BankImplTest**: Sono stati implementati test automatici per verificare il corretto comportamento della classe BankImpl, responsabile della gestione delle operazioni economiche tra i giocatori. I test coprono i seguenti aspetti:
  - **Deposito di denaro:** verifica del corretto accreditamento di importi positivi sul saldo del giocatore e del lancio di un'eccezione in caso di importi negativi.
  - **Prelievo di denaro:** controllo del corretto addebito di importi validi (inferiori o uguali al saldo disponibile), della gestione dei casi di fondi insufficienti (senza alcuna modifica del saldo) e del comportamento del sistema in presenza di importi negativi.
  - **Trasferimento di fondi:** verifica della corretta transazione di denaro tra due giocatori quando l'importo è valido e disponibile, **nonché** della gestione dei casi in cui gli importi siano negativi o superiori al saldo del giocatore mittente.
  - **Verifica della disponibilità economica:** accertamento che venga restituito il valore corretto in base al saldo attuale del giocatore e che vengano **gestiti** in modo adeguato eventuali importi non validi.
- **BoardTest**: I test per la classe BoardImpl si concentrano su aspetti fondamentali come:
  - Verifica del corretto comportamento di metodi come size(), getTileAt() e normalizePosition().

  - Assicurazione che la posizione venga correttamente normalizzata e che l'accesso alle tessere avvenga senza errori.
- **CardDeckTest**: La suite di test per CardDeckImpl verifica:
  - Che il metodo getAllCards() restituisca una copia difensiva della lista.
  - La corretta gestione delle carte che devono essere mantenute, come nel caso della carta "GET_OUT_OF_JAIL_FREE".
  - Funzionalità di scarto delle carte, sia per tipo che individualmente.
- **GameCardTest**: I test per GameCardImpl verificano:
  - La correttezza dei getter della carta e la rappresentazione testuale.
  - L'integrità del comportamento di una carta quando si disegna e si esegue l'operazione di scarto.
- **PropertyImplTest**: I test per PropertyImpl si concentrano su:
  - Verifica della corretta gestione dell'assegnazione dei proprietari e delle modifiche di stato (ad esempio, l'aggiunta e la rimozione di case).
  - Verifica che l'assegnazione dell'affitto funzioni correttamente in base alla logica della proprietà.
- **PropertyStateImplTest**: I test per PropertyStateImpl includono:
  - Verifica delle proprietà iniziali di una proprietà, inclusi il prezzo di acquisto e il numero di case.
  - Comportamento del metodo addHouse() e removeHouse(), con test per i limiti sul numero massimo di case e il comportamento quando si rimuovono case.
- **LandPropertyCardTest**: I test per LandPropertyCard verificano:
  - La corretta gestione dell'affitto in base al numero di case o alberghi posseduti.
  - La logica di calcolo dell'affitto, con un comportamento diverso in base al numero di case.
- StationPropertyCardTest: I test per StationPropertyCard si concentrano sulla gestione dei canoni in base al numero di stazioni possedute.
- **Utility Property Card Test**
  UtilityPropertyCardTest: I test per UtilityPropertyCard verificano:
  - Il calcolo corretto dell'affitto in base al numero di utility possedute e al totale del lancio dei dadi.
  - Comportamento con validazioni quando non si hanno le giuste condizioni per calcolare l'affitto.
- **DiceTest:** I test per DiceImpl e DiceThrow verificano:
  - Il numero dei dadi e' compreso nell'intervallo [1, 6];
  - Verifica la correttezza del risultato del metodo getLastThrow(), confrontando il risultato del metodo con la somma algebria dei due dadi interni, e che il metodo isDouble() si attivasse correttamente.

Tutti i test sono progettati per essere eseguiti senza alcun intervento manuale e costituiscono un vincolo di integrità per ogni build del progetto.

3.2 Note di sviluppo

Turillo Luca:

Di seguito ho elencato le funzionalità avanzate del linguaggio Java e le librerie esterne utilizzate per l'implementazione delle componenti di mia competenza.

- **Progettazione con Generici:**
  - Permalink: ValidationUtils#L45-L47
  - Permalink: ValidationUtils#L290-L295
  - Permalink: ValidationUtils#L322-L327
- **Uso di Lambda Expressions:**
  - Permalink: PlayerSetupViewImpl#L167
  - Permalink: PlayerSetupViewImpl#L266
  - Permalink: PlayerImplTest#L141-L176
  - Permalink: PlayerImplTest#L212
  - Permalink: PlayerImplTest#L301
  - Permalink: PlayerImplTest#L334
  - Permalink: PlayerImplTest#L489-L508
  - Permalink: ValidationUtilsTest#L85-L109
  - Permalink: ValidationUtilsTest#L129-L138
  - Permalink: ValidationUtilsTest#L157-L172
  - Permalink: ValidationUtilsTest#L196-L211
  - Permalink: ValidationUtilsTest#L230
- **Uso di Stream:**
  - Permalink: PlayerSetupViewImpl#L239
- **Librerie di terze parti: JavaFX (Advanced IO & Graphics):**
  - Permalink: PlayerSetupViewImpl#L139-L140
  - Permalink: PlayerSetupViewImpl#L142-L145
  - Permalink: PlayerSetupViewImpl#L147-L150
  - Permalink: PlayerSetupViewImpl#L167-L191
  - Permalink: PlayerSetupViewImpl#L196-L197
  - Permalink: BoardPanel#L117
  - Permalink: BoardPanel#L131
  - Permalink: BoardPanel#L143-L147
  - Permalink: BoardPanel#L151

    - Permalink: BoardPanel#L160-L162
    - Permalink: BoardPanel#L164
- **Librerie di terze parti: JUnit 5**
    - Permalink: ValidationUtilsTest#L6-L9
    - Permalink: PlayerImplTest#L3-L10
- **Codice riadattato e Fonti esterne:**

    - Conversione Path in URI:

      La logica per convertire multipiattaforma un percorso file locale in una stringa URI tramite *file.toURI().toString()* è stata implementata seguendo le linee guida della documentazione **Oracle JavaFX** e discussioni su **StackOverflow**.

      Permalink: PlayerSetupViewImpl#L178

    - Pattern di Validazione:

      La struttura della classe di utilità *ValidationUtils* e i metodi generici di controllo come: requireAtLeast e requireNonBlank, sono state realizzate ispirandosi alle librerie di **Apache Commons Lang 3** Validate class e **Google Guava** Preconditions class, adattandole alle specifiche esigenze del progetto senza importare l'intera dipendenza. Il nome dei metodi, invece, è stato ispirato a: ***Objects.requireNonNull()***.

      Permalink: ValidationUtils

    - Applicazione Effetti Grafici, DropShadow:

      L'effetto di DropShadow applicato alle pedine dei giocatori per migliorarne la visibilità sul tabellone l'ho implementato seguendo gli esempi forniti nei tutorial ufficiali di **Oracle JavaFX** Working with Effects.

      Permalink: BoardPanel#L148-L152

Caravita Francesco:

Di seguito ho elencato le funzionalità avanzate del linguaggio Java e le librerie esterne utilizzate per l'implementazione delle componenti di mia competenza:

Uso di libreria di terze parti — Jackson (parsing/serializzazione JSON, ObjectMapper, JsonNode).

- BoardLoader : Link

- CardLoader : Link
- JsonUtils : Link

Uso di annotazioni esterne (Jackson annotations) per serializzazione / deserializzazione avanzata.

- @JsonAutoDetect:
  - BoardControllerImpl : link
  - CardControllerImpl : link
  - PropertyControllerImpl : link
  - BoardImpl : link
  - AbstractPropertyCard : link
  - CardDeckImpl : link
  - LandPropertyCard : link
  - StationPropertyCard : link
  - UtilityPropertyCard : link
  - PropertyImpl : Link
  - PropertyStateImpl : Link
- @JsonCreator e @JsonProperty:
  - BoardControllerImpl : Link
  - CardControllerImpl : Link
  - PropertyControllerImpl : link
  - BoardImpl : Link
  - AbstractPropertyCard : Link
  - CardDeckImpl : Link
  - LandPropertyCard : link
  - StationPropertyCard : link
  - UtilityPropertyCard : link
  - PropertyImpl : Link
  - PropertyStateImpl : Link
  - BuildingPayload : Link
  - MoneyPayload : Link
  - MoveRelativePayload : Link
  - MoveToNearestPayload : Link
  - MoveToPayload : Link
  - FreeParkingTile : Link
  - GoToJailTile : Link
  - JailTile : Link
  - PropertyTile : Link
  - StartTile : Link
  - TaxTile : Link

    - UnexpectedTile : Link
    - GameCardImpl : Link
- @JsonDeserialize e @JsonSubType:
    - BoardController : Link JsonSubType
    - CardController : Link JsonSubType
    - PropertyController : Link JsonSubType
    - Board : Link JsonSubType
    - Tile : Link JsonSubType
    - CardDeck : Link JsonDeserialize
    - GameCard : Link JsonSubType
    - CardPayload : Link JsonSubType
    - Property : Link JsonSubType
    - AbstractPropertyCard : Link JsonSubType
- @JsonRootName:
    - BuildingPayload : Link
    - MoneyPayload : Link
    - MoveRelativePayload : Link
    - MoveToNearestPayload : Link
    - MoveToPayload : Link
    - FreeParkingTile : Link
    - GoToJailTile : Link
    - JailTile : Link
    - PropertyTile : Link
    - StartTile : Link
    - TaxTile : Link
    - UnexpectedTile : Link
    - AbstractPropertyCard : Link
- @JsonIgnore:
    - BoardControllerImpl: Link 1
    - BoardControllerImpl: Link 2
    - CardControllerImpl: Link
    - CardDeckImpl: Link
    - LandPropertyCard: Link
    - StationPropertyCard: Link

Uso di TypeReference (idioma Jackson) per gestione di tipi generici durante la deserializzazione e Stream.

- BoardLoader : Link typereference
- BoardLoader : Link stream
- CardLoader : Link stream

- CardController : Link stream

Uso di lambda expressions:

- PropertyImplTest.java : Link
- PropertyControllerImpl : Link
- PropertyStateImplTest.java : Link

Uso di libreria di terze parti per i test: JUnit:

- Org.junit.jupiter.* :
  - BoardTest : Link
  - PropertyImplTest : Link
  - PropertyStateImplTest : Link
  - LandPropertyCardTest : Link
  - UtilityPropertyCardTest : Link
  - StationPropertyCardTest : Link

Piedimonte Antonio:

- Di seguito ho elencato le funzionalità avanzate del linguaggio Java, le librerie esterne e i pattern utilizzati per l'implementazione delle componenti di mia competenza:

- **Gestione della Concorrenza e UI Threading (JavaFX)**

- In un'applicazione grafica, la gestione dei thread è critica. Ho implementato la sincronizzazione tra logica e grafica tramite:
- **Platform.runLater():**
  - MainViewImpl: Per l'aggiornamento thread-safe del log di gioco e la visualizzazione di Alert.
  - MatchControllerImpl: Utilizzato per scatenare aggiornamenti visivi (come showWinner) in seguito a mutazioni dello stato del modello.

- **Uso di Lambda Expressions e Functional Interface**

- Ho sfruttato la programmazione funzionale per rendere il codice più snello e leggibile:
  - **Event Handling:**
    - Gestione dei click sui bottoni nel CommandPanelImpl.
  - **Property Listeners:**
    - Utilizzati in MainViewImpl per implementare l'auto-scrolling del log basato sulla heightProperty.

- **Stream API:**
    - Per il filtraggio dei giocatori e la gestione dei dati nelle fasi di caricamento della partita nel MatchControllerImpl.

- **Analisi Statica e Qualità del Software (SpotBugs)**

    - Per garantire l'aderenza agli standard di qualità e la sicurezza dell'incapsulamento:
        - **@SuppressFBWarnings:**
            - MainViewImpl : Per giustificare l'esposizione controllata di nodi grafici (es. getRoot()).
            - MatchControllerImpl : Per gestire i riferimenti circolari necessari al pattern Mediator tra View e Controller.

- **Uso di librerie per la Gestione delle Risorse**

    - **Composizione e Astrazione:**
        - DiceThrow e DiceImpl : Implementazione della logica tramite composizione di oggetti, separando il calcolo del risultato dalla logica di business del controller.

    - **java.util.logging:**
        - Implementazione di un sistema di logging professionale per tracciare il caricamento degli asset grafici nel InfoPanelImpl.
- **Test**
    - **Test sui dadi:**
        - Utilizzo di test parametrici e cicli intensivi (1000 iterazioni) su DiceImpl e DiceThrow per assicurare che la distribuzione dei risultati sia confinata nel range [1, 6] (per dado singolo) e [2, 12] (per la somma), verificando contestualmente la corretta attivazione del flag isDouble. (DiceTest)

Mularoni Luca:

Di seguito ho elencato le funzionalità avanzate del linguaggio Java e le librerie esterne utilizzate per l'implementazione delle componenti di mia competenza.

Uso di Lambda Expressions:

- Permalink: LiquidationObserverImpl.java#L112-L115

- Permalink: MenuViewImpl.java#L144-L146

- Permalink: MenuViewImpl.java#L176-L179

- Permalink: PlayerSetupViewImpl.java#L110-L116

- Permalink: PlayerSetupViewImpl.java#L229

- Permalink: PlayerSetupViewImpl.java#L233

- Permalink: PlayerSetupViewImpl.java#L267-L273

- Permalink: SellAssetViewImpl.java#L141

- Permalink: SellAssetViewImpl.java#L159

- Permalink: BankImplTest.java#L51

- Permalink: BankImplTest.java#L69

- Permalink: BankImplTest.java#L109

- Permalink: BankImplTest.java#L133

Uso di Stream:

- Permalink: LiquidationObserverImpl.java#L112-L115

- Permalink: MenuViewImpl.java#L71-L74

-Pemalink: MenuViewImpl.java#L106-L113

- Permalink: PlayerSetupViewImpl.java#L208-L210

- Permalink: PlayerSetupViewImpl.java#L220

- Permalink: PlayerSetupViewImpl.java#L229

- Permalink: PlayerSetupViewImpl.java#L233

- Permalink: PlayerSetupViewImpl.java#L255-L258

Uso JsonUtils:

- Permalink: MatchControllerDeserializer.java#L26-L36

Uso di annotazioni esterne (Jackson annotation):

@JsonAutodetect e @IgnoreProperties

- Permalink: MatchControllerImpl.java#L44-L45
- Permalink: MatchControllerImpl.java#L58
- Permalink: MatchControllerImpl.java#L68

@JsonCreator e @JsonProperty:

- Permalink: MatchControllerImpl.java#L114-L125
- Permalink: CardDeckImpl.java#L64-L68

- Permalink: GameCardImpl.java#L38-L44
- Permalink: DiceThrow.java#L24-L26

- Permalink: PlayerImpl.java#L146-L150

@JsonDeserialize e @JsonSubType:

- Permalink: PlayerState.java#L28-L36
- Permalink: BoardController.java#L13
- Permalink: CardController.java#L12
- Permalink: CardDeck.java#L10
- Permalink: GameCard.java#L10

Librerie di terze parti: JUnit

- Permalink: BankImplTest.java#L3-L13

Librerie di terze parti: JavaFX

- Permalink: MenuViewImpl.java#L52C2-L52C2
- Permalink: MenuViewImpl.java#L61-L63
- Permalink: MenuViewImpl.java#L81
- Permalink: MenuViewImpl.java#L88-L94
- Permalink: MenuViewImpl.java#L103-L105
- Permalink: MenuViewImpl.java#L109-L110
- Permalink: MenuViewImpl.java#L115-L117
- Permalink: MenuViewImpl.java#L119
- Permalink: MenuViewImpl.java#L130-L132

- Permalink: MenuViewImpl.java#L140-L149
- Permalink: MenuViewImpl.java#L158-L164
- Permalink: MenuViewImpl.java#L171-L177
- Permalink: MenuViewImpl.java#L207-L213
- Permalink: MenuViewImpl.java#L219-L229
- Permalink: MenuViewImpl.java#L235-L237
- Permalink: MenuViewImpl.java#L243-L245
- Permalink: PlayerSetupViewImpl.java#L39-L47
- Permalink: PlayerSetupViewImpl.java#L53-L63
- Permalink: PlayerSetupViewImpl.java#L68-L73
- Permalink: PlayerSetupViewImpl.java#L80-L87
- Permalink: PlayerSetupViewImpl.java#L94-L100
- Permalink: PlayerSetupViewImpl.java#L280-L296
- Permalink: SellAssetViewImpl.java#L26-L29
- Permalink: SellAssetViewImpl.java#L42-L49
- Permalink: SellAssetViewImpl.java#L54-L60
- Permalink: SellAssetViewImpl.java#L66-L72
- Permalink: SellAssetViewImpl.java#L77-L79
- Permalink: SellAssetViewImpl.java#L84-L124
- Permalink: SellAssetViewImpl.java#L133-L143
- Permalink: SellAssetViewImpl.java#L152-L161
- Permalink: SellAssetViewImpl.java#L169-L184
- Permalink: SellAssetViewImpl.java#L192-L207
- Permalink: SellAssetViewImpl.java#L212-L216
- Permalink: SellAssetViewImpl.java#L223-L227
- Permalink: SellAssetViewImpl.java#L240-L242

Per la generazione di immagini è stata utilizzata intelligenza artificiale (Gemini con NanoBanana).

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Turillo Luca:

Ruolo all'interno del gruppo:

Il mio ruolo all'interno del gruppo si è concentrato principalmente sulla modellazione del *Model*, con particolare riferimento al ***Player*** e al suo ecosistema. Ho avuto la responsabilità di progettare e implementare la logica che governa lo stato del giocatore tramite il *Pattern State*, la gestione delle risorse finanziarie e l'identità visiva *Token*. Parallelamente, mi sono occupato di garantire la robustezza dell'applicazione implementando utility per la validazione dei dati. Ho inoltre curato la *View* relativa a *PlayerSetupView*, introducendo funzionalità di I/O per la personalizzazione delle pedine.

Punti di forza:

Ritengo che il mio principale punto di forza risieda nell'architettura del *Model*. Ho disaccoppiato la logica di stato: Libero, In Prigione o in Bancarotta dalla classe principale del giocatore, e questo ha reso il codice più pulito e conforme ai principi *SOLID*. L'utilizzo del *Pattern Observer* per comunicare alla *View* i cambiamenti che avvenivano nel *Player* senza che quest'ultimo conoscesse minimamente l'implementazione visiva. Anche l'utilizzo del *Pattern Factory* ha facilitato di non poco la creazione e la gestione dei *Token.* Un altro elemento positivo è la robustezza: La classe *ValidationUtils* e l'uso di controlli sui dati in ingresso hanno ridotto la possibilità che il sistema si trovi in stati inconsistenti, facilitando il lavoro per l'intero gruppo.

Punti di debolezza:

Riconosco che la mia parte di *Model*, sebbene ben strutturata, presenta una scarsa complessità algoritmica: la maggior parte della logica di calcolo e delle regole di gioco è stata delegata al *Controller* o ad altri manager, rendendo le mie classi talvolta dei semplici contenitori di dati o macchine a stati, senza una vera intelligenza computazionale. Inoltre, considero una criticità il mio scarso coinvolgimento nello sviluppo del *Controller* e, in parte, della *View* principale. Essendomi focalizzato molto sul *Model* e sul *Setup*, ho delegato quasi interamente la gestione del flusso di gioco agli altri membri della squadra, questo mi ha portato, in alcune fasi, ad avere una visione meno chiara di come i miei componenti venissero utilizzati durante l'esecuzione della partita.

Sviluppi futuri:

Ho intenzione di continuare lo sviluppo di questo progetto anche dopo la consegna. Essendo la prima applicazione di elevata complessità che affronto in un contesto di lavoro di gruppo, ritengo *Javapoly* un elemento di grande valore per il mio portfolio personale. La mia volontà è quella di perfezionare il codice esistente ed implementare gli obiettivi opzionali che sono stati esclusi per limiti di tempo. In particolare, mi piacerebbe lavorare sull'introduzione di giocatori controllati dalla CPU, dei *Bot*, sfruttando la struttura a stati che ho già predisposto, e implementare una logica *Multiplayer Cross-Platform*, permettendo a più istanze del gioco di comunicare in rete, prima in locale, e poi magari anche non solo.

Caravita Francesco:

Ruolo:
Il mio ruolo si è concentrato principalmente sulla progettazione e implementazione del model e del controller. In particolare mi sono occupato della definizione della struttura del tabellone e della gestione delle interazioni ad esso collegate, curando la logica che regola il comportamento delle varie componenti. Parallelamente ho gestito il caricamento dei dati iniziali fissi del tabellone, lavorando direttamente con i file JSON. Ho progettato e realizzato un oggetto unico di gestione dei dati che si occupa dell'organizzazione delle informazioni provenienti dal JSON, oggetto che è stato poi utilizzato anche dagli altri membri del gruppo per interagire correttamente con i dati. Questo ha permesso di centralizzare l'accesso alle informazioni, evitare duplicazioni di codice e mantenere coerenza nell'utilizzo delle risorse condivise.

Punti di forza:

Penso che il punto forte sia la struttura del model, il quale mi ha richiesto la maggior parte del tempo e dell'attenzione durante lo sviluppo. Ho cercato di costruirlo in

modo solido e coerente, definendo con precisione le entità principali del dominio e le loro relazioni, evitando sovrapposizioni di responsabilità. La suddivisione tra interfacce e implementazioni mi ha permesso di mantenere una buona separazione tra definizione dei comportamenti e dettagli concreti, rendendo il codice più leggibile ed estendibile. Anche la gestione dei dati tramite JSON è stata pensata in modo centralizzato, così da evitare duplicazioni e garantire coerenza nell'accesso alle informazioni. Nel complesso considero il model la parte più solida del progetto, sia dal punto di vista strutturale sia da quello concettuale.

Punti di debolezza:

Nonostante la struttura del model sia solida, ritengo che alcune parti possano essere ulteriormente migliorate dal punto di vista della semplicità e della chiarezza. In alcuni casi la progettazione risulta piuttosto articolata, con diversi livelli di astrazione che, se da un lato rendono il sistema flessibile, dall'altro possono aumentare la complessità percepita e richiedere più tempo per essere compresi da chi legge il codice per la prima volta. Alcune classi potrebbero essere rifattorizzate per ridurre la lunghezza dei metodi o per rendere ancora più esplicite le responsabilità interne. Inoltre, la documentazione potrebbe essere ampliata, ad esempio spiegando meglio le scelte architetturali adottate e il flusso delle principali interazioni tra le componenti. Con più tempo a disposizione avrei potuto dedicarmi a una fase di revisione finale più approfondita, mirata a migliorare ulteriormente leggibilità, coerenza stilistica e ottimizzazione di alcune parti del codice.

Piedimonte Antonio:

Ruolo all'interno del gruppo:

Il mio contributo si è focalizzato sulla progettazione del **MatchController**, l'elemento centrale di coordinamento dell'intero sistema, e sullo sviluppo dell'architettura della **View principale**. Ho avuto la responsabilità di orchestrare il flusso dei turni, gestendo l'interazione tra la logica di movimento, l'economia e le proprietà. Ho curato la componente estetica e funzionale dell'interfaccia (log di gioco, pannelli di controllo e card dei giocatori), garantendo un'esperienza utente reattiva e coerente.

Punti di forza:

Considero l'adozione del **Pattern Mediator** nel controller il principale punto di forza del mio lavoro. Questa scelta ha permesso di disaccoppiare nettamente la View dai micro-controller specialistici, centralizzando la logica di validazione delle azioni. Un altro aspetto positivo è la gestione della **thread-safety** in JavaFX: l'uso sistematico di Platform.runLater() ha garantito la stabilità dell'applicazione anche durante aggiornamenti complessi della GUI. Infine, l'integrazione di Jackson per la persistenza

è stata implementata con un alto livello di precisione, superando le difficoltà legate ai riferimenti circolari tra oggetti grazie a un uso oculato delle annotazioni.

Punti di debolezza:

Riconosco una criticità nella gestione della **complessità della View**. Con l'aumentare delle funzionalità, alcune classi legate all'interfaccia (come MainViewImpl) sono diventate eccessivamente corpose, violando parzialmente il principio di singola responsabilità (SRP). Sebbene funzionanti, queste componenti avrebbero beneficiato di una scomposizione ulteriore in micro-pannelli ancora più indipendenti. Inoltre, la gestione del CSS dinamico, pur essendo efficace, è stata gestita in parte direttamente via codice Java; un approccio basato esclusivamente su file .css esterni avrebbe garantito una separazione ancora più netta tra logica e stile.

Sviluppi futuri:

Per il futuro, intendo raffinare l'architettura della View adottando un framework di gestione degli stati più evoluto per eliminare del tutto le dipendenze residue. Dal punto di vista delle funzionalità, vorrei implementare un sistema di **animazioni avanzate** per il movimento delle pedine sul tabellone, migliorando il feedback visivo durante il lancio dei dadi. Inoltre, mi piacerebbe estendere il sistema di persistenza per supportare il **Cloud Saving**, permettendo agli utenti di riprendere la sessione di gioco da dispositivi diversi.

Mularoni Luca:

Ruolo all'interno del gruppo:

Il mio ruolo all’interno del gruppo si è contrato principalmente sulla modellazione del Controller, in particolare alla classe EconomyController e al relativo ecosistema. Ho avuto la responsabilità di progettare in parte (Bank) la logica incaricata di coordinare le diverse classi e di gestire le transazioni economiche (EconomyController), nonché’ il collegamento con la View per la gestione delle situazioni di insufficienza di fondi tramite il LiquidationObserver

Inoltre, mi sono occupato della gestione del menu, implementandone tutte le funzionalità principali. In quanto ho usato la libreria Jackson per la serializzazione e deserializzazione dello stato di gioco, permettendo il cariacemento delle partite direttamente dal menu e il salvataggio delle partite.

Ho curato la View relativa al menu relativa al menu principale del gioco, occupandomi inoltre del cambio di vista dalla MenuView alla PlayerSetupView. In quest'ultima ho gestito sia l'interfaccia grafica sia i controlli per l'inserimento e la validazione dei nomi dei giocatori. Infine, ho collegato questa vista alla MainView, garantendo una corretta integrazione con la View principale dell'applicazione.

Punti di forza:

Analizzando il lavoro svolto, ritengo che i principali punti di forza risiedano nella struttura del controller. L'utilizzo della creazione della classe EconomyController mi è servito per fare chiarezza architetturale e fornire alla View un insieme di metodi pronti all'uso. Un altro elemento che mi è stato di grande aiuto è stata l'implementazione del pattern Observer, per gestire efficientemente la fase di liquidazione.

Infine, considero un grande punto di forza nell'essere riuscito a portare a termine con successo un obiettivo opzionale del progetto, cioè l'implementazione di un sistema di salvataggio automatico (autosave).

Punti di debolezza:

In una revisione critica e oggettiva, riconosco la possibile presenza di race condizioni e la mancanza di rollback per garantire l'atomicità delle operazioni finanziarie, la presenza di poco model e una grafica minimale senza tante grafiche complesse

Sviluppi futuri:

Anch'io ho intenzione di proseguire lo sviluppo di questo progetto.

In futuro, ho intenzione di proseguire lo sviluppo di questo progetto. In particolare, mi piacerebbe implementare l'obiettivo opzionale escluso a causa dei limiti di tempo: la gestione delle statistiche di gioco. Avendo già predisposto un sistema di tracciamento delle transazioni, questo potrebbe essere esteso per generare grafici in grado di mostrare l'andamento del bilancio dei giocatori nel tempo.

In conclusione, ritengo che questo progetto, sviluppato in un contesto collaborativo fortemente assimilabile a un ambiente aziendale, abbia rappresentato un'esperienza formativa di grande rilevanza, fornendomi competenze pratiche preziose per il mio futuro percorso professionale.

Appendice A

Guida utente

Di seguito vengono illustrati i passaggi principali per avviare e condurre una partita.

4.1 Menu Principale:

All'avvio dell'applicazione, l'utente viene accolto dalla schermata del Menu Principale, che presenta tre opzioni fondamentali:

- **Nuova Partita:** Avvia la procedura di configurazione per una nuova partita.
- **Carica Partita:** Permette di riprendere una partita precedentemente salvata. Selezionando questa opzione, si aprirà una finestra di dialogo del sistema operativo che richiederà all'utente di selezionare il file di salvataggio in formato .json.
- **Exit:** Chiude l'applicazione.

4.2 Configurazione della Partita:

Se si sceglie di iniziare una nuova partita, si accede alla schermata di Player Setup. La procedura si articola in due passaggi:

1. **Selezione numero giocatori:** Nella parte superiore della schermata è presente un menu a tendina che consente di selezionare il numero di partecipanti da un minimo di 2 a un massimo di 4.
2. **Dettagli Giocatori:** Al centro della schermata ci sono i campi per l'inserimento dei dati. Per ogni giocatore è necessario:
    a. Inserire un **Nome** obbligatorio e univoco.
    b. Selezionare una **Pedina** dal menu a discesa. È possibile scegliere tra le pedine predefinite.

Una volta completati i campi, premere il tasto **Confirm** in basso per accedere al tavolo da gioco.

4.3 Interfaccia di Gioco:

La schermata principale della partita è suddivisa in quattro aree funzionali distinte.

- **Tabellone (Centro):** Occupa la parte centrale della finestra e mostra la disposizione delle caselle e la posizione attuale delle pedine dei giocatori.
- **Pannello Giocatori (In alto a destra):** In quest'area sono riepilogate le informazioni essenziali per ogni partecipante, in particolare il **Nome** e il **Saldo attuale**, aggiornati in tempo reale dopo ogni transazione.
- **Pannello di Vendita degli Asset** (In centro a destra): In questa sessione sono riepilogati il debito da saldare e gli asset che possono essere venduti. La vendita avviene secondo un ordine prestabilito: inizialmente vengono cedute le abitazioni, qualora queste non siano più disponibili, si procede con la vendita delle altre proprietà possedute.
- **Log Attività (Sinistra):** Una console testuale che registra la cronologia degli eventi (lanci di dadi, acquisti, pagamenti di affitto, imprevisti), permettendo ai giocatori di tenere traccia di quanto accaduto nei turni precedenti.
- **Pannello Comandi (In basso a sinistra):** Contiene i pulsanti interattivi per eseguire le azioni di gioco. I comandi principali includono:
  - **Lancia Dadi:** Per lanciare i dadi e muovere la pedina.
  - **Compra:** Per acquistare la proprietà sulla quale si è atterrati o costruirvi edifici, il tasto si attiva solo se l'azione è permessa.
  - **Fine Turno:** Per passare il turno al giocatore successivo, fa un salvataggio in automatico.
  - **Salva:** Per salvare lo stato della partita, quest’ultimo verrà salvato nel root dell’utente.