Relazione progetto di Programmazione ad Oggetti

Scotland Yard

Francesco Barzanti
Irene Borri
Raffaello Fraboni

1 febbraio 2026

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il software mira a riprodurre una versione digitale per singolo giocatore del gioco da tavolo "Scotland Yard". In questo gioco da tavolo sono presenti due fazioni: una è composta da un solo giocatore, chiamato Mister X, e l'altra è composta da un detective e, eventualmente, uno o più bobbies (aiutanti del detective). Nell'applicazione si fa scegliere al giocatore umano la difficoltà di gioco e la modalità di gioco:

- Modalità Detective: l'obiettivo è catturare il fuggitivo, Mister X
- Modalità Mister X: l'obiettivo è scappare dal detective e dai suoi aiutanti (se presenti), non facendosi catturare fino alla fine dell'ultimo round

I giocatori si muovono all'interno di una mappa utilizzando specifici mezzi di trasporto. I mezzi di trasporto disponibili sono: taxi, bus, metropolitana, traghetto. La posizione del fuggitivo non è nota al detective e ai suoi aiutanti: viene rivelata solo in pochi turni specifici. Tuttavia, si può visualizzare il mezzo che ha usato ad ogni round.

Per effettuare uno spostamento con un certo mezzo, ogni giocatore deve utilizzare un biglietto relativo a quel mezzo: il detective ha un numero finito di biglietti per ogni categoria, mentre i bobbies e il fuggitivo hanno numero infinito di biglietti per ciascun mezzo. Mister X, inoltre, possiede due tipi di biglietti speciali: biglietto "Doppia Mossa", che gli consente di effettuare due spostamenti in un solo turno, e biglietto "Black" che gli permette di utilizzare il traghetto. Ques'ultimo, infatti, è l'unico mezzo che può essere usato solo da Mister X.

Una partita di gioco termina se Mister X è stato catturato, oppure se il detective non può più effettuare mosse (per le possibili destinazioni che può raggiungere ha finito la tipologia di biglietto richiesta), oppure se si è superato l'ultimo round. Il software, inoltre, permette all'utente di visualizzare delle statistiche sulle partite di gioco svolte:

- Partita di gioco più lunga
- Numero di vittorie e numero di sconfitte

Dunque, sarà presente un menù che permetterà all'utente di selezionare ciò che desidera fare: effettuare una partita di gioco oppure visualizzare le statistiche sulle partite già giocate.

1.1.1 Requisiti funzionali

- In creazione di una nuova partita l'utente dovrà poter selezionare la modalità di gioco tra Mister X e Detective e la difficoltà della partita.
- Il numero di bobbies è variabile: cambia a seconda della modalità di gioco e della difficoltà di gioco selezionate.
- Ad ogni turno il giocatore deve poter selezionare una nodo destinazione in mappa e in caso di multipli mezzi utilizzabili deve poter selezionare quale vuole usare.
- Il computer controlla l'avversario in entrambe le modalità: in modalità Detective controlla Mister X; in modalità Mister X controlla il detective e i suoi aiutanti.
- Il software dovrà rilevare quando le condizioni di fine partita sono soddisfatte e dovrà mostrare una schermata di game over.
- L'applicazione deve gestire l'IA, ovvero l'avversario controllato dal computer, impostando algoritmi di diversa complessità e efficienza a seconda della difficoltà di gioco scelta dall'utente.

1.1.2 Requisiti non funzionali

- Il software dovrà essere compatibile con i principali sistemi operativi sul mercato.
- La finestra di gioco dovrà permettere di essere ridimensionata per essere compatibile con qualsiasi monitor.

1.2 Modello del Dominio

Il cuore di Scotland Yard è costituito dallo stato di gioco *GameState*. Al suo interno vengono memorizzate varie informazioni, tra cui : modalità di gioco *GameMode*, livello di difficoltà *GameDifficulty* e pedine di gioco. Ogni pedina di gioco è rappresentata da un giocatore *Player*, ogni giocatore ha un inventario *Inventory* contenente i vari tipi di biglietto *TicketType* : sia i biglietti necessari per utilizzare il trasporto pubblico che i biglietti speciali per effetture mosse doppie. Ogni giocatore si trova in una posizione *MapNode* della mappa *MapData*.

Ogni giocatore durante il suo turno potrà effettuare delle azioni *GameCommand* tra cui l'utilizzo dei biglietti di trasporto pubblico per effettuare una mossa, l'utilizzo del biglietto speciale per permettere di effettuare due mosse in un singolo turno o passare il turno. A seconda del ruolo scelto in creazione della partita alcune pedine saranno pilotate automaticamente dal computer *PlayerBrain* che ogni round potrà scegliere tra le stesse azioni disponibili al giocatore umano.

Ogni mappa è composta da dei nodi *MapNode*, ognuno con un id *NodeId*, collegati tra loro da delle connessioni *MapConnection* ognuna con un tipo veicolo *TransportType* di collegamento.

La mappa contiene inoltre la lista di nodi *NodeId* che potranno essere assegnati ad ogni giocatore all'inizio della partita e l'insieme dei turni dove la posizione del fuggitivo, Mister X, sarà visibile a tutti i giocatori.

In ogni momento del gioco il gruppo dei detective potrà consultare lo storico di biglietti utilizzati da Mister X tracciati da *RunnerTurnTracker* nei turni precedenti per cercare di predirre la posizione attuale del player ricercato.

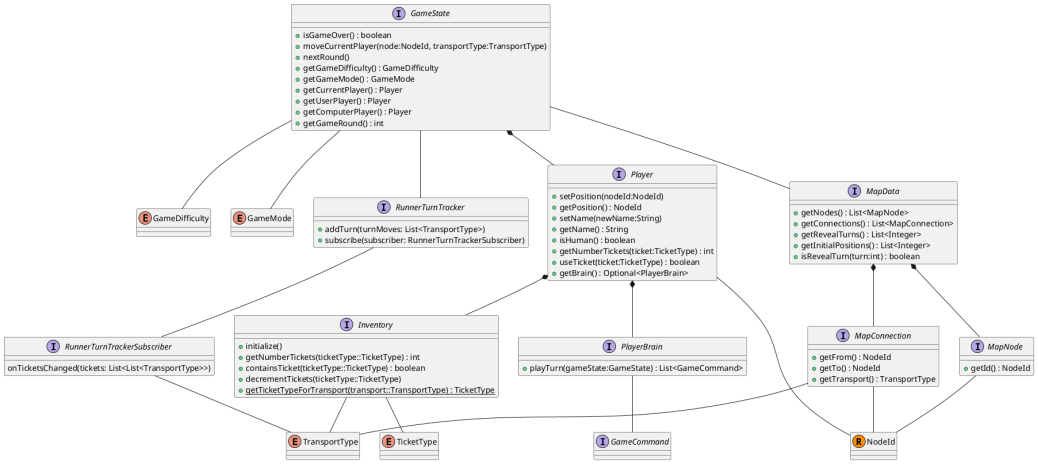
GameState
+ isGameOver() : boolean
+ moveCurrentPlayer(node:NodeId, transportType:TransportType)
+ nextRound()
+ getGameDifficulty() : GameDifficulty
+ getGameMode() : GameMode
+ getCurrentPlayer() : Player
+ getUserPlayer() : Player
+ getComputerPlayer() : Player
+ getGameRound() : int
GameDifficulty
GameMode
RunnerTurnTracker
+ addTurn(turnMoves: List<TransportType>)
+ subscribe(subscriber: RunnerTurnTrackerSubscriber)
Player
+ setPosition(nodeId:NodeId)
+ getPosition() : NodeId
+ setName(newName:String)
+ getName() : String
+ isHuman() : boolean
+ getNumberTickets(ticket:TicketType) : int
+ useTicket(ticket:TicketType) : boolean
+ getBrain() : Optional<PlayerBrain>
MapData
+ getNodes() : List<MapNode>
+ getConnections() : List<MapConnection>
+ getRevealTurns() : List<Integer>
+ getInitialPositions() : List<Integer>
+ isRevealTurn(turn:int) : boolean
RunnerTurnTrackerSubscriber
onTicketsChanged(tickets: List<List<TransportType>>)
Inventory
+ initialize()
+ getNumberTickets(ticketType:TicketType) : int
+ containsTicket(ticketType:TicketType) : boolean
+ decrementTickets(ticketType:TicketType)
+ getTicketTypeForTransport(transport:TransportType) : TicketType
PlayerBrain
+ playTurn(gameState:GameState) : List<GameCommand>
MapConnection
+ getFrom() : NodeId
+ getTo() : NodeId
+ getTransport() : TransportType
MapNode
+ getId() : NodeId
TransportType
TicketType
GameCommand
NodeId

Figura 1.1:

Capitolo 2

Design

2.1 Architettura

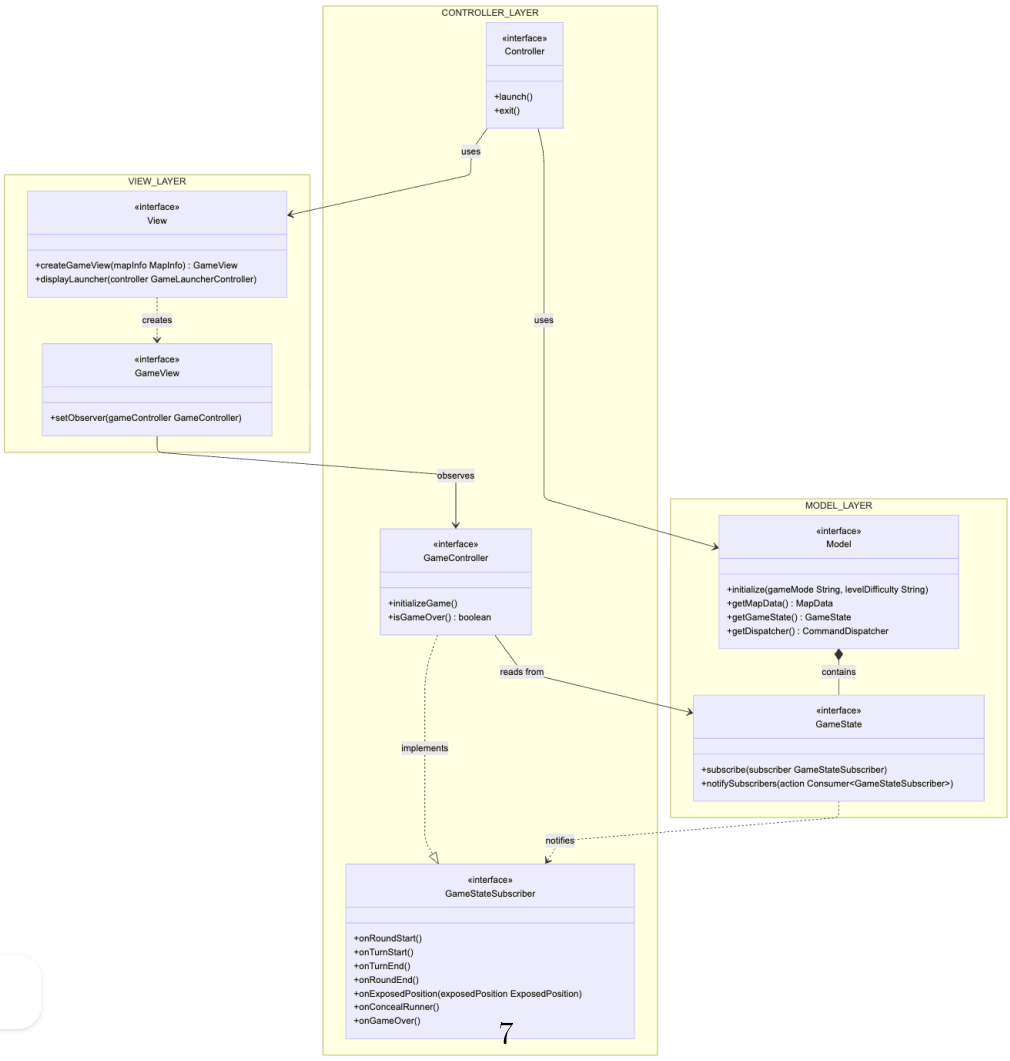

Figura 2.1: Architettura MVC

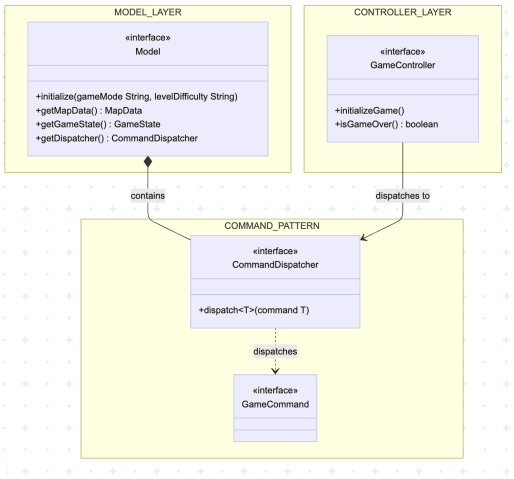

Figura 2.2: Architettura Dispatcher

L'architettura di Scotland Yard segue il pattern architetturale MVC. Il punto di ingresso dell'applicazione è la classe *ScotlandYard*, che contiene il metodo main. Si istanziano i tre componenti principali dell'architettura MVC, passando al *Controller* i riferimenti del *Model* e della *View*. Dopodiché viene delegata l'esecuzione al *Controller*.

Il *Controller* fa da intermediario tra *Model* e *View*.

Le classi appartenenti al *Model* si occupano esclusivamente di rappresentare e gestire lo stato del gioco, e non presentano alcun riferimento né al *Controller* né alla *View*.

Durante l'esecuzione dell'applicazione, il *Controller* principale delega ad altri controller la gestione di specifiche task. (Sezione 2.2)

Oltre ai tre layer principali di MVC (Model, View, Controller), sono stati introdotti due ulteriori layer, in cui è suddiviso il *Model*:

- Service : si occupa di rispondere ai comandi *GameCommand* chiamati dal giocatore umano e dal giocatore controllato dal computer.

- Repository : si occupa di gestire la persistenza dei dati.Sezione 2.2

In particolare, nel layer *Model* è presente un dispatcher *CommandDispatcher* che ha il compito di eseguire il giusto handler a seconda del comando *GameCommand* ricevuto in ingresso. I vari handler sono memorizzati all'interno del layer Service nel *Model*. Dunque, durante lo svolgimento di una partita di gioco, quando il giocatore esegue un *GameCommand*, viene chiamato dal controller il metodo dispatch() del *Dispatcher* che esegue il relativo handler registrato.

2.2 Design dettagliato

2.2.1 Francesco Barzanti

Caricamento della mappa da file JSON

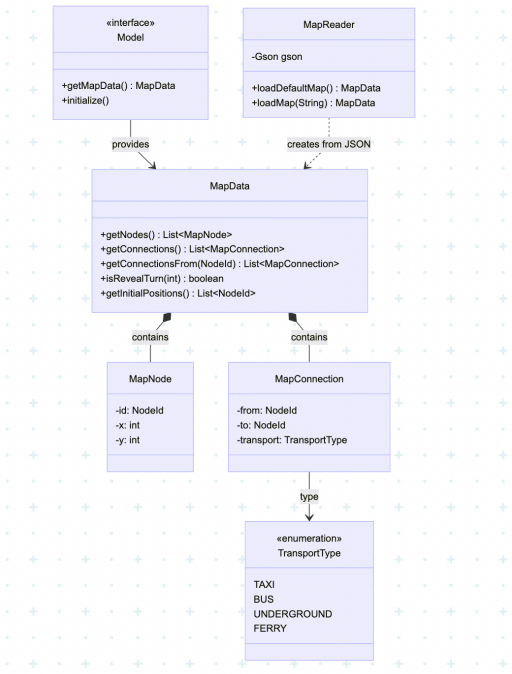

Figura 2.3: Pattern Factory Method, caricamento della mappa da JSON

Problema L'applicazione richiede il caricamento di una mappa complessa da file JSON contenente nodi e connessioni organizzate per tipo di trasporto, oltre a turni di rivelazione e posizioni iniziali. La struttura JSON è annidata e non omogenea: i nodi sono array di oggetti mentre le connessioni sono raggruppate per tipo di trasporto, ciascuno contenente array di coppie di ID. Era necessario separare la logica di caricamento dalla rappresentazione dei dati e garantire estensibilità per supportare mappe diverse o altri formati.

Soluzione È stato adottato il *pattern Factory Method* (Figura 2.3): la classe `MapReader` espone metodi di istanza (`loadDefaultMap()`, `loadMap(String)`) che fungono da factory, creando istanze di `MapData` a partire dal file JSON. Per la deserializzazione è stata scelta la libreria Gson che gestisce il parsing JSON, attraverso classi `JsonObject` e `JsonArray` per il parsing delle strutture nidificate. Il caricamento attraversa tre fasi: parsing JSON in struttura

intermedia, trasformazione delle connessioni da formato compatto a lista di oggetti, costruzione dell'istanza immutabile di `MapData`. Questo design permette di estendere il sistema aggiungendo nuovi factory method senza modificare il codice client.

Render Mappa di gioco

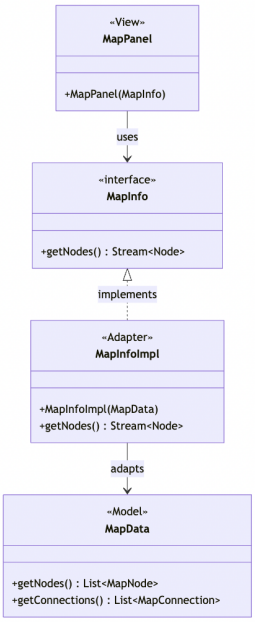

Figura 2.4: Pattern Adapter per il disaccoppiamento tra Model e View

Problema Il componente `MapPanel` deve renderizzare la mappa ma non può dipendere direttamente dalle classi del Model per rispettare MVC. Inoltre, necessita di informazioni aggregate non presenti nel Model: per colorare i nodi deve conoscere i tipi di trasporto disponibili da ciascun nodo, informazione distribuita tra le connessioni. Aggiungere questa informazione a `MapNode` inquinerebbe il Model con dati derivati utili solo alla presentazione.

Soluzione È stato applicato il *pattern Adapter* (Figura 2.4) combinato con DTO: `MapInfoImpl` adatta `MapData` (Model) all'interfaccia `MapInfo` (DTO) richiesta dalla View. L'adapter calcola i trasporti disponibili per ogni nodo filtrando le connessioni rilevanti e crea stream di DTO (`Node`) implementati come Java records immutabili. Quando la View chiama `getNodes()`, l'adapter genera al volo i DTO arricchiti con l'informazione aggregata, liberando

la View dalla logica di calcolo e garantendo separazione completa tra layer (zero dipendenze dirette Model-View).

Gestione del turno in modalità Mr. X

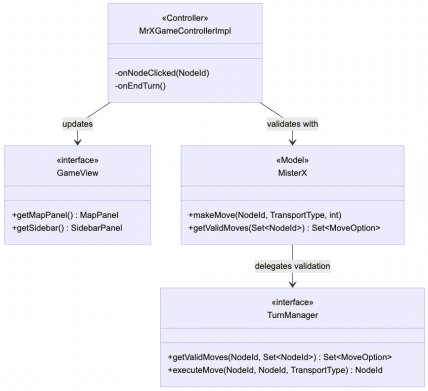

Figura 2.5: Architettura MVC per la gestione del turno di Mr. X

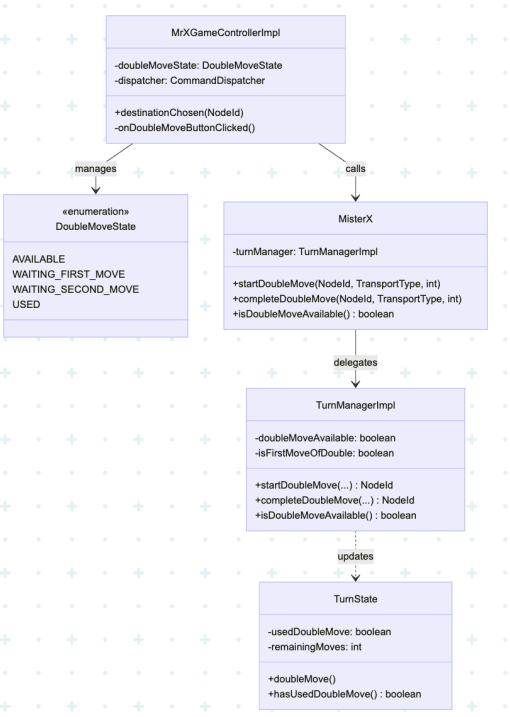

Figura 2.6: Gestione Double Move

Problema Nella modalità Mr. X, il giocatore controlla il personaggio tramite interfaccia grafica che richiede gestione coordinata di input utente (click nodi, selezione trasporto), validazione mosse (destinazione raggiungibile, ticket disponibili, posizioni non occupate), feedback visivo (nodi verdi, aggiornamento sidebar). La sfida è coordinare questi aspetti mantenendo separazione netta tra logica di presentazione (View), controllo (Controller), e dominio (Model).

Soluzione L'architettura adotta il *pattern MVC* (Figura 2.5) con `MrXGameControllerImpl` come orchestratore centrale. Il Controller registra listener su `MapPanel` e `SidebarPanel`, valida le azioni interrogando `TurnManager.getValidMoves()` (che filtra connessioni raggiungibili escludendo posizioni occupate), e aggiorna la View. La validazione è delegata interamente al Model: `TurnManager` fornisce i metodi di validazione mosse (`getValidMoves()`, `isValidMove()`), separando la logica di validazione dalla logica del player, `MisterX` lo utilizza senza conoscerne i dettagli, permettendo di variare le regole senza modificare il giocatore.

Problema Mister X ha a sua disposizione un biglietto Doppia Mossa, che se usato permette di muoversi di 2 mosse all'interno dello stesso round. L'esigenza è quella di implementare un flusso che sia chiaro e funzionale per il giocatore.(Figura 2.6)

Soluzione Una volta cliccato il Bottone "Doppia Mossa" nella sidebar, inizia il flusso gestito con *state machine* locale attraverso l'enum `DoubleMoveState` (AVAILABLE → WAITING_FIRST_MOVE → WAITING_SECOND_MOVE → COMPLETED → USED), dove ogni transizione delega operazioni a `MisterX.startDoubleMov` e `completeDoubleMove()` ed aggiorna la view. La notifica al Model avviene tramite sistema di dispatching comandi[1].

Tracciamento e persistenza della partita più lunga

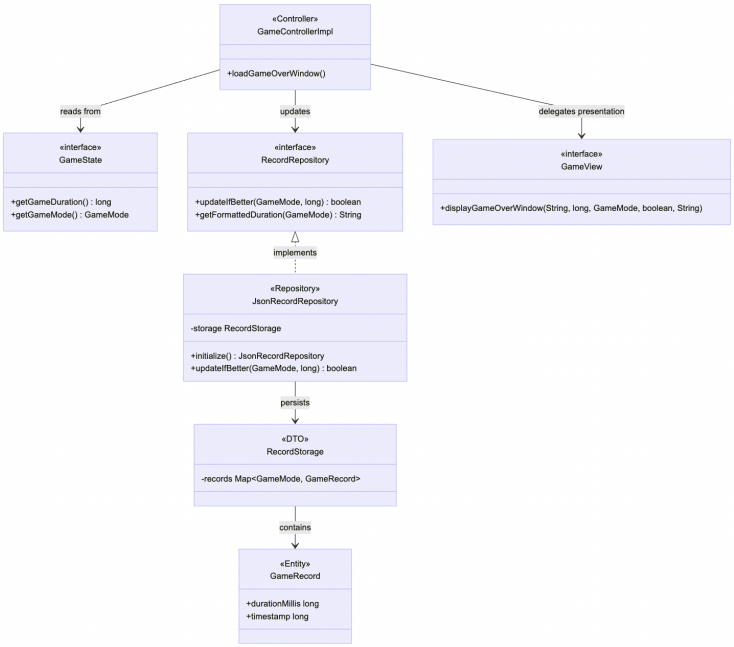

Figura 2.7: Pattern Repository per la persistenza del record

Problema Il gioco deve tracciare e persistere la durata della partita più lunga per ciascuna modalità (Detective e Mr. X), mostrando a fine partita se è stato stabilito un nuovo record. Era necessario mantenere record separati per

[1] Realizzato da Raffaello Fraboni

modalità con consistenza anche in caso di errori I/O, e gestire efficentemente il primo avvio quando non esiste il file di persistenza.

Soluzione È stato applicato il *pattern Repository* (Figura 2.7): l'interfaccia `RecordRepository` espone operazioni di alto livello (`updateIfBetter()`, `getFormattedDuration()`), mentre `JsonRecordRepository` implementa la persistenza su file JSON con Gson. Il tracking temporale avviene in `GameState` che registra `gameStartTime` all'inizio e calcola `gameDuration` alla fine. `GameController` interroga `GameState` e chiama `repository.updateIfBetter()` che esegue atomicamente leggi-confronta-scrivi, ritornando se è un nuovo record. L'implementazione utilizza un *Factory Method* `initialize()` che verifica esistenza di directory e file, creando la struttura iniziale al primo avvio per evitare errori. La persistenza usa `Map<GameMode, GameRecord>` serializzata, con `GameMode` come chiave.

2.2.2 Irene Borri

Gestione dei personaggi

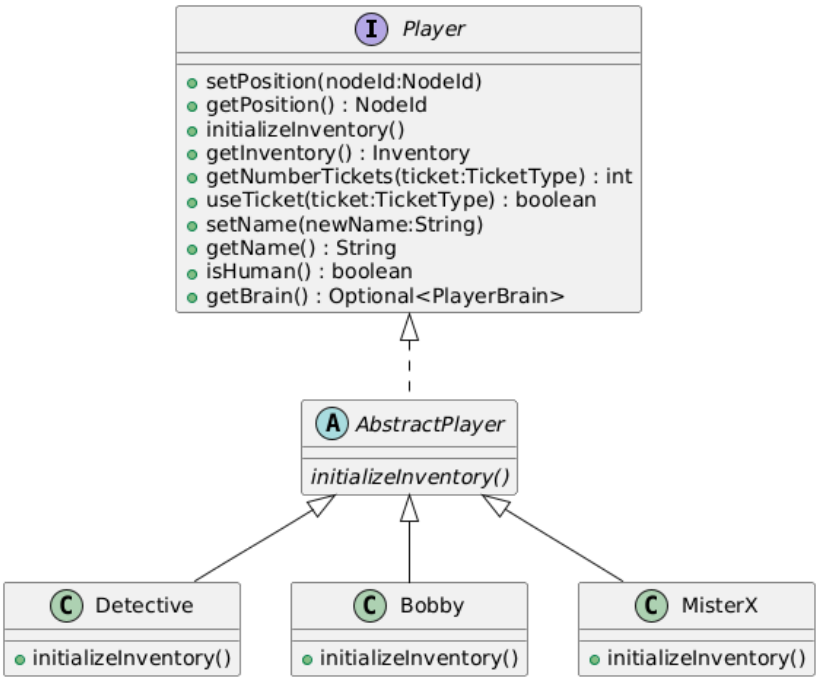

Figura 2.8: Interfacce e classi per i personaggi

Problema Scotland Yard prevede tre tipi differenti di personaggi, che possono svolgere le stesse azioni, come l'utilizzo di un biglietto per un mezzo di trasporto, ma che si differenziano proprio per il ruolo che svolgono all'interno del gioco, da cui dipende l'inventario posseduto. Questi ruoli sono : Detective, Bobby (ovvero l'aiutante del detective), Mister X (cioè il fuggitivo).

Soluzione Si adotta un'interfaccia *Player* che viene implementata dalla classe astratta *AbstractPlayerImpl*. Le classi *Detective*, *Bobby*, *Mister X* estendono la classe astratta e si differenziano per il costruttore in cui viene assegnato un diverso nome a ciascun player e per l'implementazione del metodo astratto *initializeInventory()*. L'uso della classe astratta *AbstractPlayerImpl* consente il riutilizzo di comportamenti comuni a tutti i giocatori e permette eventuali aggiornamenti o modifiche future.

Gestione dell'inventario

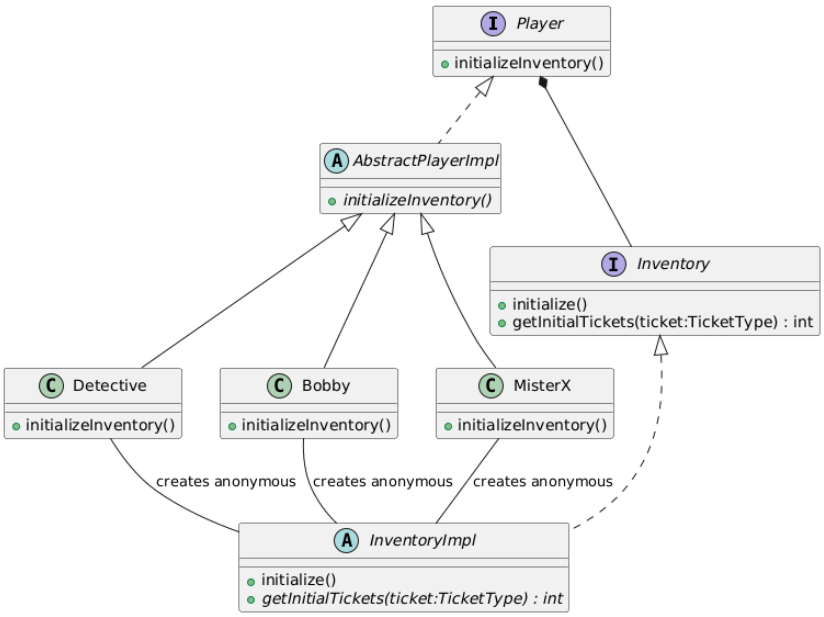

Figura 2.9: Gestione inventario con pattern Template Method

Problema L'applicazione deve gestire correttamente l'inventario dei personaggi. Ciascun giocatore possiede un inventario, che varia a seconda del ruolo che ha, ovvero a seconda di quale personaggio è : *Detective*, *Bobby*,

Mister X. Si è potuto osservare che, però, i differenti inventari condividono gran parte dei comportamenti.

Soluzione I vari inventari differiscono soltanto per il numero di biglietti di ciascun *TicketType* che vengono memorizzati. Dunque, viene adottato il *pattern comportamentale Template Method* : il metodo template è *initialize()*, che chiama per ciascun *TicketType* esistente il metodo astratto *getInitialTickets(TicketType)*, che restituisce un intero rappresentante il numero di biglietti iniziali del *TicketType* passato come argomento. In particolare, viene utilizzata un'interfaccia *Inventory*, che viene implementata dalla classe astratta *InventoryImpl*. Le classi concrete vengono istanziate come classi anonime all'interno del metodo astratto *initializeInventory()* delle classi specifiche dei giocatori : *Detective*, *Bobby*, *Bobby*. Grazie all'utilizzo di questo pattern è possibile massimizzare il riuso per eventuali agiornamenti futuri dell'applicazione (per esempio : si potrà aggiungere un nuovo tipo di *Player* con il relativo inventario, sfruttando il metodo template descritto qui).

Gestione del menù principale

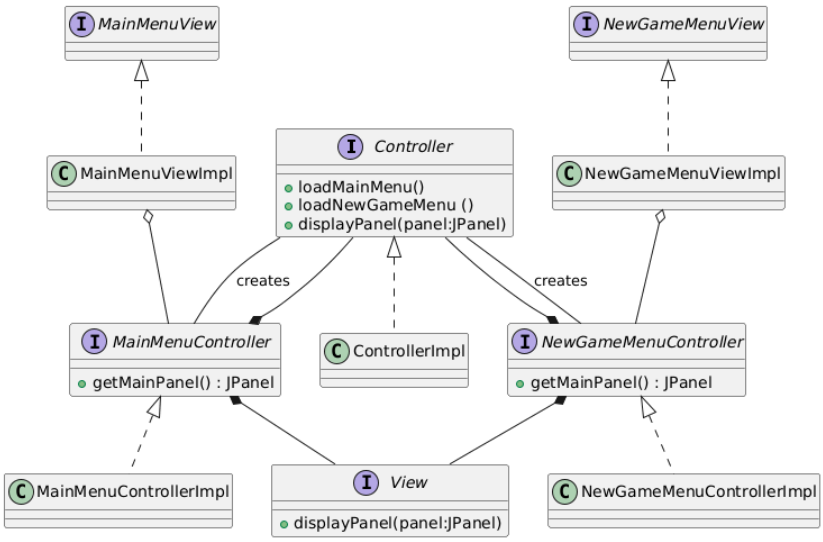

Figura 2.10: Gestione dei menù con pattern Observer

Problema L'applicazione deve permettere all'utente di selezionare ciò che desidera fare : iniziare una nuova partita di gioco oppure visualizzare le statistiche relative alle partite già giocate. Dunque, risulta necessaria la gestione di un menù.

Soluzione Si utilizzano due menù : uno che permette di scegliere se iniziare una nuova parita, visualizzare le statistiche o uscire dall'applicazione; e uno che permette di impostare una nuova parita, facendo selezionare all'utente modalità di gioco e difficoltà di gioco. In particolare, i menù vengono gestiti nei soli layer di Controller e View : sono presenti le interfacce *MainMenuController* e *NewGameMenuController*, entrambe con la loro implementazione (*MainMenuControllerImpl* e *NewGameControllerImpl*); e sono presenti le interfacce *MainMenuView* e *NewGameView*, con le relative implementazioni (*MainMenuViewImpl* e *NewGameControllerImpl*). L'interazione tra i due layer avviene tramite il *pattern Observer* : il controller è l'observer, mentre la view è l'observable; questo vale sia per il *MainMenu* che per il *NewGameMenu*.

Gestione del turno in modalità Detective

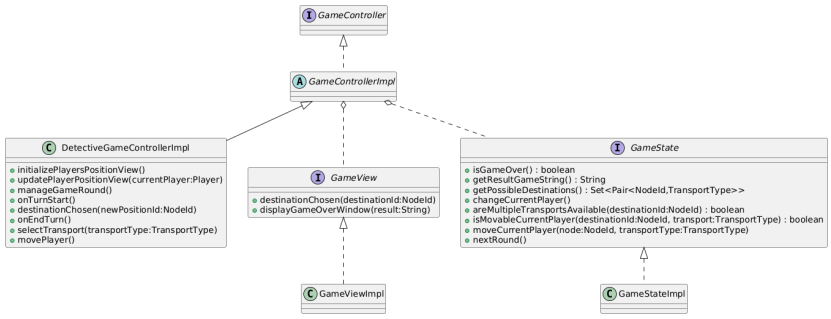

Figura 2.11: Interfacce e classi coinvolte nella gestione del turno nella modalità Detective

Problema Nella modalità Detective, l'utente controlla sia la pedina del suo giocatore, ovvero il detective, ma anche gli eventuali bobby (se presenti, a seconda della difficoltà scelta). L'applicazione deve gestire il flusso di gioco nei tre layer principali (Model, View, Controller), mantenendo indipendente la View.

Soluzione La gestione dei turni in questa modalità è controllata dal *DetectiveGameControllerImpl* che estende la classe astratta *GameControllerImpl*, che definisce i comportamenti comuni tra i due tipi di *GameController*[2]. Il *DetectiveGameControllerImpl* si occupa di gestire il flusso di gioco, chiamando opportunamente metodi della *GameViewImpl* o del *GameStateImpl*,

[2]Interfaccia e classe astratta sono state realizzate in stretta collaborazione con Francesco Barzanti

entrambi suoi campi. Quando si verifica un cambiamento nella *GameViewImpl*, come la pressione del bottone "Fine Turno" o la scelta del nodo su cui spostarsi, il *DetectiveGameControllerImpl* chiama metodi specifici del *GameStateImpl* per effettuare eventuali modifiche su di esso, per poi proseguire con la gestione del turno. L'esecuzione vera e propria dei *GameCommand*, come la mossa del giocatore, avvengono tramite il *CommandDispatcher*.[3]

2.2.3 Raffaello Fraboni

Command system

Problema Ogni giocatore deve poter interagire con il motore di gioco: il giocatore umano interagisce con la View, scatenando una reazione nel Controller che a sua volta dovrà interagire con il motore di gioco, il giocatore computer dovrà riutilizzare le stesse interfacce per effettuare le stesse interazioni dell'utente.

Si vuole progettare un'interfaccia unica per uniformare i punti di ingresso al motore di gioco che permetta di codificare tutte le interazioni possibili dell'utente.

Inoltre si vuole rendere il sistema predisposto per i test automatici permettendo di simulare le azioni utente tramite liste di operazioni elencate nel codice di test o serializzate su file per permettere di riprodurre casi errore incontrati da testers e utenti dell'applicazione.

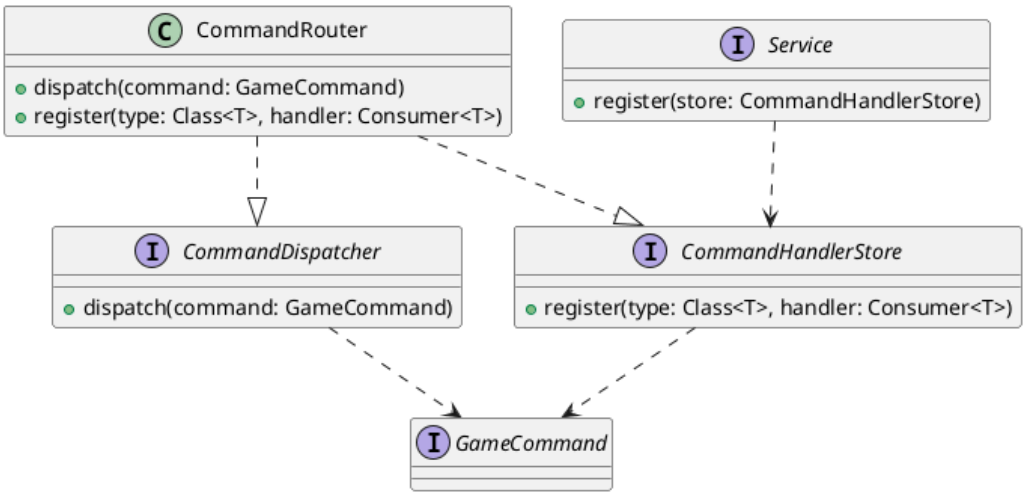

Figura 2.12: Rappresentazione UML del Command Dispatcher pattern per le azioni di gioco

[3]Realizzato da Raffaello Fraboni

Soluzione La soluzione scelta dei problemi elencati è stata di rappresentare le interazioni utente come oggetti del dominio stesso: ogni azione di gioco viene codificata come un *GameCommand* basato sul Command pattern.

Rispetto al Command pattern introdotto in [GHJV94] ho scelto di separare il comando dalla sua esecuzione, in questo modo i comandi non sono strettamente accoppiati alla logica applicativa, ciò permette di aggiungere più implementazioni interscambiabili di handler dello stesso comando. Il risultato è di fatti la variazione del pattern chiamata *Command Dispatcher pattern* discussa in [DF01].

I comandi contengono solo informazioni volte allo svolgimento delle azioni e non ripetono dati già contenuti nello stato applicativo. Ad esempio il comando per finire il turno non contiene l'identificativo del giocatore siccome lo stato di gioco contiene già quest'ultimo.

Il ruolo di *Command Dispatcher* è rivestito da *CommandRouter* che implementa sia un'interfaccia per effettuare il *dispatch* dei comandi, sia una seconda interfaccia per registrare un *Command Handler* per ogni comando. Si è scelto di separare l'implementazione in due interfacce per permettere di esporre solo il metodo *dispatch* al Controller infatti la registrazione degli handler può avvenire solo tramite *CommandHandlerStore* (Figura 2.12) che non viene esposto pubblicamente dal Model.

I *Command Handler* sono stati organizzati in classi *Service* e ad ogni comando ricevuto interagiscono con il GameState per applicare le necessarie transizioni di stato. In questo modo è possibile usare il *CommandDispatcher* come unico punto d'accesso per interagire con lo stato di gioco sia dai Controller che dal giocatore computer permettendo di uniformare le logiche (Figura 2.13).

Il motore di gioco dovrà essere deterministico tale che presa la lista di comandi di una partita, se eseguita nuovamente da uno stato pulito si raggiunga lo stesso stato applicativo iniziale, in altre parole i cambiamenti di stato indotti da un comando devono dipendere solo dai comandi eseguiti precedentemente. Rendere il motore di gioco deterministico ha il vantaggio di semplificare considerevolmente la riproduzione di casi d'errore, ad esempio sarebbe possibile configurare un sistema di test basato su fuzzing che restituisca la catena di eventi che ha portato allo stato d'errore.

Questo sistema permette di aggiungere funzionalità di salvataggio, caricamento e annullamento mosse, infatti sarebbe possibile salvare le partite come elenco di comandi eseguiti, ripristinare lo stato di gioco eseguendo nuovamente gli stessi comandi nello stesso ordine e semplifica notevolmente lo sviluppo dell'annullamento di mosse. Quest'ultimo può essere sviluppato ingenuamente salvando lo stato applicativo ad ogni comando, ma può essere migliorato

per salvare solo la differenza tra stati oppure includendo le operazioni inverse necessarie per ripristinare lo stato precedente.

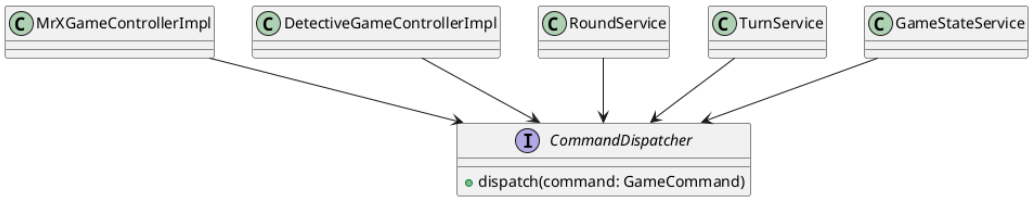

Figura 2.13: Rappresentazione UML dell'utilizzo del CommandDispatcher come unico punto d'ingresso per interagire con il motore di gioco

Giocatore Mister X computer

Problema Il giocatore computer deve poter interagire allo stesso modo dell'utente con il motore di gioco.

Inoltre è necessario progettare un algoritmo per scegliere la mosse utilizzate da Mister X che si adatta a seconda della difficultà scelta: il computer deve cercare di fare mosse peggiori a difficoltà più bassa.

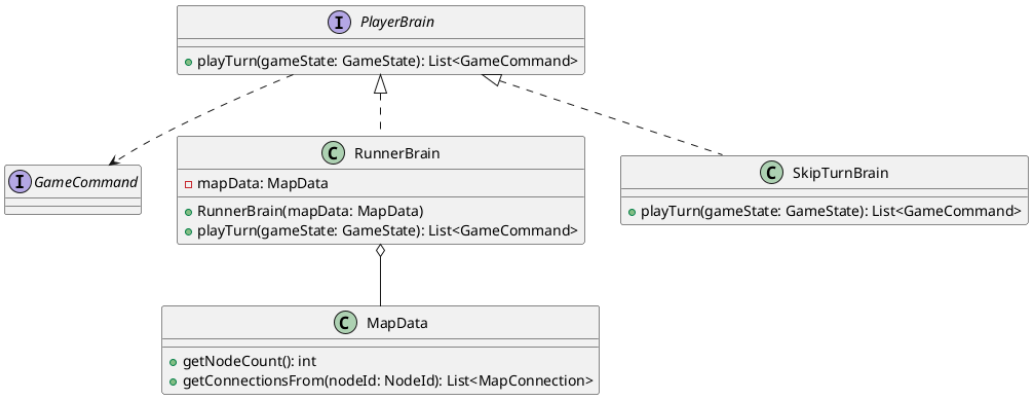

Figura 2.14: Rappresentazione UML dell'utilizzo del Strategy pattern per personalizzare il comportamento dei giocatori computer

Soluzione Viene associato ad ogni giocatore un riferimento ad un *PlayerBrain*, ogni giocatore computer ha un'implementazione di questa interfaccia assegnata e all'inizio dei suoi turni viene delegata l'esecuzione del turno a questa implementazione

Ogni *PlayerBrain* è composto da un solo metodo che restituisce la lista di operazioni effettuate sotto forma di comandi come descritto in 2.2.3.

Il *PlayerBrain* è la strategia del *Strategy* pattern che permette di cambiare l'implementazione del giocatore computer tra un giocatore e l'altro (Figura 2.14). L'implementazione chiamata *SkipTurnBrain* è un implementazione di sviluppo che si limita a passare il turno, invece l'implementazione che si occupa del ruolo di Mister X è chiamata *RunnerBrain* e ha due principali funzionalità:

Quando la difficultà del gioco è impostata a facile il computer si limita a scegliere casualmente una mossa valida da effettuare durante il turno.

Invece a difficoltà più alte, viene calcolato il minimo numero di turni tra ogni nodo e il ricercatore più vicino, dopodiché il computer sceglie casualmente il nodo destinazione tra i più distanti e a seconda della difficoltà sceglie mosse più o meno lontane dai cercatori.

Il calcolo della distanza è calcolato usando un algoritmo di *Multi-Source BFS* dove siccome non è necessario visitare un nodo più di una volta può essere implementato come un semplice *Breadth-First Search* dove i nodi da visitare iniziano da tutte le sorgenti, in questo caso i cercatori. L'algoritmo di partenza è stato preso da [TKC+14, Listing 1] per poi essere riadattato a questo caso d'uso.

L'algoritmo utilizzato per la scelta della mossa di Mister X può essere ulteriormente migliorato assegnando un costo agli "edge" basato sul biglietto richiesto, favorendo nodi lontani da un maggioranza di cercatori (i.e. a parità di distanza prioritizzare i nodi più lontani da altri detective) e tenendo in considerazione i biglietti disponibili del detective nel calcolo della distanza dei nodi.

Inoltre un possibile miglioramento potrebbe essere l'utilizzo di meccanismi asincroni per effettuare le mosse dell computer ad esempio utilizzando un *Observable<GameCommand>* della liberia *RxJava* al posto di una semplice lista per poter aggiungere dei ritardi simulati tra pubblicazione di comandi e per aggiungere eventuali timeout di ricerca mosse in caso di sviluppo di meccanismi AI che migliorano la soluzione nel tempo.

L'algoritmo attuale si limita ad analizzare la posizione attuale e seleziona in maniera "greedy" tra le mosse che lo portano più lontano, un algoritmo più sofisticato potrebbe invece avvicinarsi ad eventuali detective per poi prendere un traghetto verso un nodo lontano da tutti i cercatori.

Tracciamento vittorie sconfitte

Problema Viene richiesto di tracciare il numero di partite vinte e perse del giocatore umano.

Il sistema di tracciamento deve persistere il dato anche dopo aver chiuso e riaperto il gioco. In caso di vittoria il contatore di vittorie deve essere incrementato altrimenti il contatore di sconfitte.

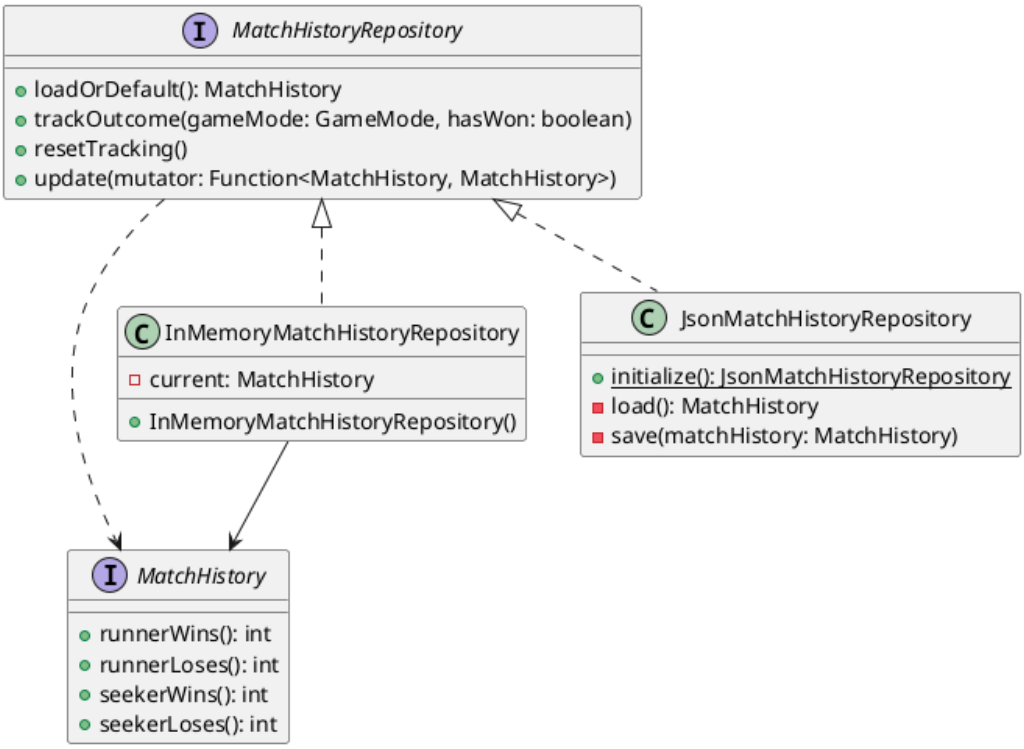

Figura 2.15: Rappresentazione UML del Repository e Strategy pattern per l'implementazione della MatchHistory

Soluzione Viene implementato un *JsonMatchHistoryRepository* per serializzare il numero di vittorie e sconfitte su file. Viene utilizzato il *Repository* pattern per incapsulare la logica di persistenza del dato e esponiamo solo i metodi necessari per incrementare le statistiche dopo aver vinto o perso una partita.

Siccome le statistiche non sono un aspetto critico del gioco, in caso di problemi di lettura o scrittura dei file, viene utilizzato il tracciamento in memoria al suo posto tramite *InMemoryMatchHistoryRepository* che implementa anch'esso l'interfaccia *MatchHistoryRepository*. In questo caso *MatchHistoryRepository* rappresenta la strategia dello *Strategy* pattern, mentre *InMemoryMatchHistoryRepository* e *JsonMatchHistoryRepository* sono le implementazioni (Figura 2.15).

Siccome le informazioni contenute nella pagina statistiche sono statiche, non è necessario implementare l'Observer pattern, invece possiamo utilizzare un approccio *Pull* dove all'apertura della pagina il Controller richiede le informazioni aggiornate al Model e le passa alla View per la visualizzazione senza la possibilità di aggiornamenti da parte del Model. Quando l'utente preme il tasto per azzerare le statistiche vengono caricate nuovamente dal Model.

Entrambe le implementazione sono state progettate per avere un unica fonte di verità. Nel caso dell'implementazione json ad ogni richiesta delle statistiche viene letto nuovamente il file e viene restituita una struttura immutabile *MatchHistory* con i contenuti del file. Nel caso dell'implementazione in memoria viene salvato un riferimento all'ultima versione delle statistiche ed ad ogni modifica viene sostituita.

Si è fatto uso di *higher-order functions* per codificare i cambiamenti di stato all'interno di *MatchHistoryImpl* per semplificare il codice che incrementa la statistica corrispondente al risultato della partita.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il progetto utilizza JUnit 5 e la libreria Mokito per eseguire gli unit test automatici.

InMemoryMatchHistoryRepositoryTest

Verifica il corretto funzionamento dell'aggiornamento delle statistiche riguardo le vincite e le sconfitte nelle varie modalità, testando usando un test parametrizzato tutti gli incrementi e anche le combinazioni.

Verifica che la lettura in memoria funzioni correttamente, anche dopo una modifica, e controlla che aggiornare le statistiche non modifichi oggetti MatchHistory precedentemente letti.

JsonMatchHistoryRepositoryTest

Test analogo al precedente ma ho deciso di tenerlo separato siccome questa versione può andare in errore dovuto a lettura e scrittura di file su disco. In aggiunta al test precedente ci sono controlli riguardo le `IOException`.

TurnStateImplTest

Verifica il corretto funzionamento del tracciamento delle mosse effettuate, del tracciamento delle posizioni assunte durante il turno, e verifica che non è possibile fare più mosse di quante disponibili.

CommandRouterTest

Verifica il corretto funzionamento del Command Dispatch pattern, questa implementazione supporta solo un handler per comando quindi verifica che si comporti correttamente in sostituzione di un handler con un altro. Verifica inoltre che il corretto callback venga chiamato quando viene fatta la dispatch di un comando.

PlayerTest

Verifica che un Player venga creato correttamente, sia che abbia *PlayerBrain* che no. Inoltre verifica che ciascun player inizializzi in maniera corretta il proprio inventario. Verifica inoltre il corretto funzionamento dei Player, ovvero : che venga settato corettamente il cambio nome, che venga cambiata correttamente la sua posizione, che venga utilizzato un biglietto in maniera giusta.

InventoryTest

Verifica che l'inventario venga creato correttamento per ogni tipo di giocatore, ovvero che contenga il giusto numero di biglietto per ciascun tipo. Verifica inoltre la correttezza del comportamento dell'inventario, cioè il controllo sulla presenza di un certo *TicketType* e il decremento di un certo tipo di biglietto al suo interno.

MisterXTest

Testing automatizzato della classe MisterX. Verifica la corretta creazione del player con posizione iniziale e nome, l'inizializzazione dell'inventario con ticket infiniti per TAXI, BUS e UNDERGROUND e ticket limitati per BLACK e DOUBLE_MOVE. Include test per il consumo dei ticket finiti fino all'esaurimento, il metodo setPosition(), e il corretto lancio di IllegalStateException quando si invocano metodi di movimento prima dell'inizializzazione con MapData.

MisterXMovesTest

Testing di integrazione delle mosse di MisterX sulla mappa di test. Verifica il cambiamento di posizione durante makeMove() e l'immutabilità dei ticket infiniti, il decremento dei ticket BLACK quando si usa il trasporto FERRY.

Per le double move verifica il consumo del ticket DOUBLE_MOVE, il cambiamento di posizione nella prima e seconda mossa, e il corretto lancio di IllegalStateException quando i ticket DOUBLE_MOVE sono esauriti.

GameRecordTest

Testing automatizzato dell'entità GameRecord. Verifica la creazione di record validi con duration e timestamp, il metodo isValid() che restituisce false per duration pari a zero, il lancio di Exception per duration negative e per timestamp nulli. Include test per il costruttore di default che crea record invalidi, l'uguaglianza tra record basata su duration e timestamp.

JsonRecordRepositoryTest

Testing automatizzato del repository JSON per la persistenza dei record su file. Verifica il pattern Factory Method. Include test che aggiorna solo se la nuova duration è maggiore, rifiuta duration invalide, e mantiene record separati per GameMode.DETECTIVE e GameMode.MISTER_X.

RecordStorageTest

Testing automatizzato del contenitore RecordStorage. Verifica il costruttore di default che inizializza con record invalidi per entrambe le modalità il lancio di NullPointerException con mappa nulla. Include test getRecords() che implementa defensive copy restituendo una mappa immutabile, e la corretta gestione di aggiornamenti multipli dello stesso GameMode.

MapConnectionTest

Testing automatizzato della classe MapConnection. Verifica la creazione con costruttore a 3 parametri che produce ID vuoto, e costruttore a 4 parametri che include l'ID opzionale.

MapDataTest

Testing automatizzato della struttura dati MapData. Verifica la validazione NullPointerException su tutti i parametri del costruttore e i getter. Include test per getNodeById(), getConnectionsFrom() con e senza filtro TransportType, getNeighbors(), isRevealTurn() per identificare i turni di rivelazione, info() che restituisce MapInfo.

MapNodeTest

Testing automatizzato della classe MapNode. Verifica la creazione con NodeId e coordinate (x, y), l'uguaglianza basata esclusivamente sull'ID, la disuguaglianza tra nodi con ID diversi anche se hanno stesse coordinate. Include test per la gestione di coordinate negative, zero e molto grandi.

MapReaderTest

Testing automatizzato del pattern Factory Method per il caricamento mappe JSON. Verifica loadDefaultMap() che carica la mappa standard con nodi e connessioni, loadMap() che parsa correttamente nome, conteggio nodi, nodi e coordinate, connessioni. Include test per parsing dei revealTurns e initialPositions convertite a NodeId, lancio di Exception per path nullo, per file non esistenti e JSON malformato, verifica della bidirezionalità delle connessioni, immutabilità delle liste restituite.

TransportTypeTest

Testing automatizzato dell'enum TransportType. Verifica l'esistenza di tutte le costanti (TAXI, BUS, UNDERGROUND, FERRY), values() che restituisce array di 4 elementi.

3.2 Note di sviluppo

3.2.1 Francesco Barzanti

Utilizzo della libreria GSON

Utilizzata in vari punti. Un esempio è `https://github.com/novelhawk/OOP25-scot-yard/blob/35666eeae519e6570df8ab801bcff1c5fb1d11bd/src/main/java/it/unibo/scotyard/model/map/MapReader.java#L27C2-L30C6`

Utilizzo di `Stream`

Utilizzati in vari punti. Un esempio: `https://github.com/novelhawk/OOP25-scot-yard/blob/35666eeae519e6570df8ab801bcff1c5fb1d11bd/src/main/java/it/unibo/scotyard/commons/dtos/map/MapInfoImpl.java#L43C4-L52C6`

Utilizzo di lambda expressions

Utilizzate in vari punti. Permalink: `https://github.com/novelhawk/OOP25-scot-yard/blob/35666eeae519e6570df8ab801bcff1c5fb1d11bd/src/main/java/it/unibo/scotyard/view/ViewImpl.java#L115C1-L123C6`

Utilizzo di `Optional`

Permalink: `https://github.com/novelhawk/OOP25-scot-yard/blob/35666eeae519e6570df8ab801bcff1c5fb1d11bd/src/main/java/it/unibo/scotyard/model/map/MapData.java#L101C5-L103C6`

Codice riutilizzato da altri progetti

Per la gestione della selezione della risoluzione di gioco, è stata riutilizzata e adattata la logica e la struttura implementata nel progetto **OOP23-TD** (Tower Defense).

Nello specifico, sono stati adattati i seguenti componenti dal progetto originale:

- `Size` e `SizeImpl`: interfaccia e implementazione per la rappresentazione delle dimensioni dello schermo
- `GameLauncherView` e `GameLauncherViewImpl`: interfaccia e implementazione della vista del launcher di gioco
- `GameLauncherController` e `GameLauncherControllerImpl`: interfaccia e implementazione del controller per il launcher
- `Window` e `WindowImpl` (parzialmente): interfaccia e implementazione della finestra principale, con modifiche per adattarle alle specifiche esigenze del progetto Scotland Yard

Il codice originale è disponibile al seguente repository:
`https://github.com/giacomoarienti/OOP23-TD/tree/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8`
Gli adattamenti effettuati hanno riguardato principalmente l'integrazione con l'architettura MVC specifica di Scotland Yard e la personalizzazione dell'interfaccia grafica secondo i requisiti del gioco.

3.2.2 Raffaello Fraboni

Funzioni di ordine superiore

Permalink: https://github.com/novelhawk/OOP25-scot-yard/blob/f6cfc2a3b6f586847aac48eed78be1d158e0c084/src/main/java/it/unibo/scotyard/model/game/matchhistory/MatchHistoryImpl.java#L18-L73

Utilizzo di reflection e metodi generici

Permalink: https://github.com/novelhawk/OOP25-scot-yard/blob/f6cfc2a3b6f586847aac48eed78be1d158e0c084/src/main/java/it/unibo/scotyard/model/router/CommandRouter.java#L17-L37

Utilizzo di `Iterator`

Permalink: https://github.com/novelhawk/OOP25-scot-yard/blob/f6cfc2a3b6f586847aac48eed78be1d158e0c084/src/main/java/it/unibo/scotyard/model/service/GameStateService.java#L44-L54

Utilizzo di `Stream`

Permalink: https://github.com/novelhawk/OOP25-scot-yard/blob/f6cfc2a3b6f586847aac48eed78be1d158e0c084/src/main/java/it/unibo/scotyard/model/game/GameStateImpl.java#L381-L394

Utilizzo della libreria `Mokito`

Permalink: https://github.com/novelhawk/OOP25-scot-yard/blob/f6cfc2a3b6f586847aac48eed78be1d158e0c084/src/test/java/it/unibo/scotyard/model/router/CommandRouterTest.java#L49-L53

Utilizzo della libreria `Gson`

Permalink: https://github.com/novelhawk/OOP25-scot-yard/blob/f6cfc2a3b6f586847aac48eed78be1d158e0c084/src/main/java/it/unibo/scotyard/model/game/matchhistory/JsonMatchHistoryRepository.java#L55-L75

Utilizzo di test parametrizzati `JUnit5`

Permalink: https://github.com/novelhawk/OOP25-scot-yard/blob/f6cfc2a3b6f586847aac48eed78be1d158e0c084/src/test/java/it/unibo/s

cotyard/model/game/matchhistory/InMemoryMatchHistoryRepository
Test.java#L64-L90

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Francesco Barzanti

Riflettendo sul lavoro svolto, riconosco che molte delle parti che ho realizzato potevano essere progettate meglio. Ho modificato più volte il codice riguardante la gestione del turno di Mr. X, segno che la progettazione iniziale era carente. Questi ripensamenti continui sono stati fonte di rallentamento, giustificati in parte dal fatto che il coordinamento con Irene, che ha sviluppato il turno del Detective, è avvenuto principalmente quando entrambi avevamo già completato le rispettive implementazioni. Con il senno di poi, avremmo dovuto dedicare più tempo all'analisi condivisa del problema prima di procedere con il codice, approccio che si è rivelato inefficiente. Un altro aspetto che abbiamo sottovalutato è l'importanza dei test: concentrandoci principalmente sull'implementazione delle funzionalità, abbiamo trascurato la scrittura sistematica di test, il che ha reso più difficile identificare tempestivamente problemi di integrazione tra componenti. Nonostante queste considerazioni critiche, ritengo comunque di aver contribuito in modo significativo al progetto e di aver imparato lezioni preziose che mi torneranno utili in futuro. In particolare, ho compreso l'importanza della comunicazione costante durante lo sviluppo e della flessibilità nell'accettare che esistano molteplici soluzioni valide allo stesso problema. Sono grato ai miei colleghi per la collaborazione dimostrata, insieme siamo riusciti a completare un prodotto funzionante nonostante le difficoltà incontrate.

Irene Borri

Per me questo progetto è stato uno dei progetti più complessi realizzati durante il mio percorso universitario. Inizialmente, ammetto di essere stata intimorita dall'idea di doverlo affrontare. Ma, ora che è concluso, posso affermare che, nonostante le difficoltà incontrate durante il suo sviluppo, sono piuttosto soddisfatta del lavoro che è stato svolto dal nostro gruppo. Tuttavia, riconosco che ci sono aspetti del progetto che potevano essere realizzati in maniera migliore. Uno di questi è la gestione del turno del giocatore nella modalità Detective : avevo realizzato questa in parallelo con Francesco (che ha lavorato alla stessa task ma in modalità Mister X), ma il coordinamento con lui è avvenuto solo verso la fine, cioè quando entrambi avevevamo quasi concluso le implementazioni di queste task. Un altro aspetto che avrei potuto implementare meglio è la gestione dell'IA nella modalità Mister X . Essendo ormai abbastanza vicini alla deadline e essendomi concentrata prima sulle altre task, ho lasciato per ultima la gestione dell'IA dei detective, che, per effettuare una mossa, si muove in maniera casuale o pseudo-casuale, senza utilizzare un vero e proprio algoritmo efficiente sui cammini minimi. Però, ritengo che l'aspetto che ha creato più problemi per il gruppo durante lo sviluppo dell'applicazione sia la mancata analisi approfondita e progettazione (a livello di design) delle entità del dominio : abbiamo cominciato a scrivere codice subito dopo aver fatto l'analisi del problema, senza prendere in considerazione dettagli di design che poi si sono rivelati importanti. Nonostante questi aspetti negativi, ci sono stati anche aspetti positivi. In particolare, sono grata per la disponibilità e collaborazione che è stata presente all'interno del gruppo, perché, nonostante la difficoltà affronatate, credo che abbiamo realizzato un software che svolge il suo lavoro e il cui svilippo, personalmente, mi ha insegnato delle importanti lezioni per il futuro.

4.1.1 Raffaello Fraboni

Il progetto è stato sicuramente tra i più impegnativi del corso, sicuramente ci sono state delle importanti lacune in fase di analisi, le prime riunioni le abbiamo passate cercando di definire il flusso del dato, l'architettura da usare e la struttura del codice ma secondo me ci abbiamo dedicato troppo poco tempo e ci ha portato a fare delle decisioni sbagliate e una separazione MVC non completa, in particolare ci siamo trovati dipendenze circolari tra View e Controller e logica applicativa nel livello Controller. Siamo riusciti a ridurre questo problema con varie modifiche e aggiustamenti ma non completamente.

La mia conoscenza del pattern MVC in Java, in particolare sulla gestione dello stato, non era sufficiente quando abbiamo iniziato i lavori, grazie a co-

noscenze pregresse ero in grado di riconoscere errori architetturali ma non ero in grado di guidare il team verso decisioni corrette senza prima sperimentare.

Inoltre parte del mio lavoro era incentrato sulla creazione della mappa di gioco, che ha richiesto parecchio tempo quindi non sono riuscito a seguire il codice nella prima fase, probabilmente la più critica. Questa prima parte è stata più complessa del previsto infatti ho dovuto trovare un immagine da usare come mappa da gioco, associare la posizione in pixel di ogni nodo (ho scritto un programma per facilitare ma ho comunque dovuto eseguire diverse rielaborazioni manuali dell'immagine per avere un risultato accurato), associare manualmente ad ogni posizione l'id del nodo (ho usato un modello OCR inizialmente ma ho dovuto sistemare manualmente gli errori), e inoltre segnare tutti i collegamenti per ogni rispettivo mezzo di trasporto (anche qui ho scritto un programma per assistire, usando DFS e manualmente scrivendo ogni collegamento), sfortunatamente questa prima parte, nonostante abbia richiesto diverse ore non può essere valutata.

Nel gruppo ho cercato di essere più supportivo possibile, ho cercato di guidare nell'utilizzo di git e ho cercato quanto più di risolvere i problemi di architettura incontrati senza andare a toccare le parti assegnate agli altri componenti del gruppo (salvo quando necessario per la mia parte). Ho segnalato tutti i problemi incontrati nelle parti degli altri membri e ho evidenziato possibili lacune architetturali senza mai impormi sugli altri membri.

Abbiamo fatto molta fatica a gestire le tempistiche infatti abbiamo dovuto sistemare quanto più gli ultimi giorni, nonostante ciò penso che siamo riusciti a chiudere diversi punti aperti e a sviluppare un prodotto funzionante, mi ritengo soddisfatto del lavoro svolto e della nostra abilità di sistemare i punti aperti in così poco tempo.

Appendice A

Guida utente

Nuova partita

Figura A.1: Schermata di creazione nuova partita

In creazione di una nuova partità (Figura A.1) è possibile scegliere la modalità di gioco, tra Mister X e Detective, a seconda della modalità di gioco sarà possibile giocare Mister X oppure i Detective e Bobbies.

La difficoltà va a cambiare il livello del computer che gioca l'avversario e il numero di Bobbies.

Statistiche

Figura A.2: Schermata delle statistiche

Nella schermata delle statistiche (Figura A.2) è possibile vedere le statistiche attuali oppure effettuare il reset che cancella i dati memorizzati.

Schermata di gioco

Figura A.3: Schermata di gioco

Nella schermata di gioco è possibile selezionare i nodi verdi per scegliere il nodo destinazione.

Se raggiungibile con più di un mezzo di trasporto allora viene chiesto all'utente quale si vuole utilizzare.

Mister X ha inoltre un bottone per utilizzare il suo biglietto doppia mossa che gli permette di effetuare due mosse nello stesso turno.

Dopo aver premuto il bottone Doppia Mossa, sarà possibile selezionare il primo nodo da visitare, confermare la prima mossa premendo il bottone apposito (che compare al posto di quello della doppia mossa) e poi è possibile ripetere lo stesso procedimento per la seconda mossa.

Premere il bottone fine turno passa il turno al giocatore successivo.

Bibliografia

[DF01] Benoit Dupire and Eduardo B. Fernández. The command dispatcher pattern. 2001.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[TKC+14] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. The more the merrier: Efficient multi-source graph traversal. *Proc. VLDB Endow.*, 8(4):449–460, 2014.