TaflGames

Relazione per il progetto
del corso di
Object-Oriented Programming
A.A. 2022/23

Alin Stefan Bordeianu
Elena Boschetti
Andrea Piermattei
Margherita Raponi

7 aprile 2023

Indice

Capitolo 1

Analisi

Lo scopo del progetto è la realizzazione di un'applicazione incentrata su un gioco da tavolo denominato "Hnefatafl", appartenente alla famiglia dei Tafl Games, un insieme di giochi di origine nordica e celtica.

1.1 Requisiti

Requisiti funzionali

- Sono previste **due modalità di gioco**: una corrisponde alla versione originale, mentre l'altra è una variante di nostra invenzione che presenta degli elementi aggiuntivi.

- Entrambe le modalità di gioco si svolgono su una griglia quadrata di dimensione 11x11 e prevedono due giocatori, i quali dispongono di una squadra di pedine ciascuno. Un giocatore ha il ruolo di **difensore** e il suo compito è proteggere il re, che fa parte della sua squadra, fino a portarlo ad una delle uscite che si trovano agli angoli della griglia. L'altro giocatore, invece, ha il ruolo di **attaccante** e il suo scopo è catturare il re circondandolo con le proprie pedine.

- Entrambe le modalità prevedono la presenza di **pedine classiche** e di un **re**. La squadra del difensore è composta da un re e da 12 pedine classiche, inizialmente posizionate nella parte centrale della griglia, mentre la squadra dell'attaccante è composta da 24 pedine classiche distribuite sui quattro bordi della griglia. Le pedine si possono muovere orizzontalmente o verticalmente e non è possibile oltrepassare o sovrapporre altre pedine. Una pedina classica viene mangiata quando vengono posizionate due pedine nemiche in due celle adiacenti ad essa

e opposte (ad esempio, a sinistra e a destra della pedina). Il re, invece, viene mangiato quando è circondato su tutte e quattro le celle adiacenti da quattro pedine dell'attaccante; tale condizione stabilisce la vittoria dell'attaccante.

- Entrambe la modalità prevedono tre tipi di celle.
  - Le **celle classiche** non hanno alcun effetto o comportamento distintivo.
  - Il **trono** è la cella in cui è posizionato inizialmente il re. Ha la particolarità di avere una propria hitbox, comportandosi come una pedina se si cerca di fare una mangiata nelle celle adiacenti ad esso.
  - Le **uscite** sono celle presenti ad ognuno dei quattro angoli della griglia; se una di esse è raggiunta dal re, allora il difensore vince. Chiaramente, non è possibile posizionare sulle uscite nessuna pedina che non sia il re.
- La variante del gioco prevede quattro ulteriori tipi di pedine e due ulteriori tipi di celle.
  - Gli **slider** sono delle celle speciali che causano lo spostamento di una pedina alla cella più lontana possibile nella direzione indicata dalla freccia sulla cella.
  - Le **tombe** vengono posizionate in corrispondenza delle celle in cui viene mangiata una pedina.
  - La **regina** è una pedina speciale che ha il potere di riportare in gioco una pedina alleata, posizionandosi in una cella adiacente ad una tomba in corrispondenza della quale è stata mangiata tale pedina.
  - L'**arciere** ha l'abilità di partecipare ad una mangiata anche a distanza, minacciando le pedine che si trovano ad una distanza massima di tre celle sulla stessa riga o colonna in cui l'arciere viene posizionato.
  - Lo **scudo** ha il potere di sopravvivere al primo tentativo di mangiata che subisce.
  - Lo **swapper** può scambiare la propria posizione con una pedina avversaria, escluso il re.

Sia all'attaccante che al difensore sono assegnati una regina, due arcieri, due scudi e uno swapper. Al difensore è assegnato un re come nella modalità classica. Tutte le altre pedine rimanenti (18 per l'attaccante e 6 per il difensore) sono pedine classiche.

- Al momento della scelta della modalità di gioco, deve essere possibile visualizzare il regolamento di entrambe le modalità.
- Durante il suo turno, ogni giocatore deve avere la possibilità di annullare la propria mossa ed effettuarne un'altra prima di passare il turno all'altro giocatore.
- Al termine della partita, i giocatori devono avere la possibilità registrare il proprio risultato, immettendo anche il proprio nome; i risultati registrati dovranno poter essere visualizzati su una leaderboard.

Requisiti non funzionali

- L'applicazione deve essere in grado di gestire correttamente le risorse legate alla configurazione di gioco, alla parte grafica e al salvataggio e caricamento della leaderboard.
- Deve essere garantita all'utente un'interazione fluida con l'applicazione.

1.2 Analisi e modello del dominio

Il modello del dominio dell'applicazione corrisponde al concetto di partita, la quale incapsula tutto ciò che concerne la logica del gioco. L'entità centrale nel contesto della partita è la griglia di gioco (`Board`), che si occupa delle interazioni fra tutti gli altri elmenti del dominio (`Piece` e `Cell`).

La partita (rappresentata dal `Model`) si occupa della gestione dei turni di gioco e comunica con la `Board` per il controllo della validità delle mosse e il loro compimento e per ottenere il risultato della partita qualora vi siano le condizioni per la sua fine.

Internamente, la `Board` contiene la collezione di celle e la collezione di pedine. Ogni cella è modellata dall'interfaccia `Cell` e ogni pedina è modellata dall'interfaccia `Piece`; tali interfacce definiscono il modo in cui la `Board` può interrogare e modificare lo stato delle pedine per applicare tutte le meccaniche di gioco, cioè spostamenti, mangiate ed effetti speciali.

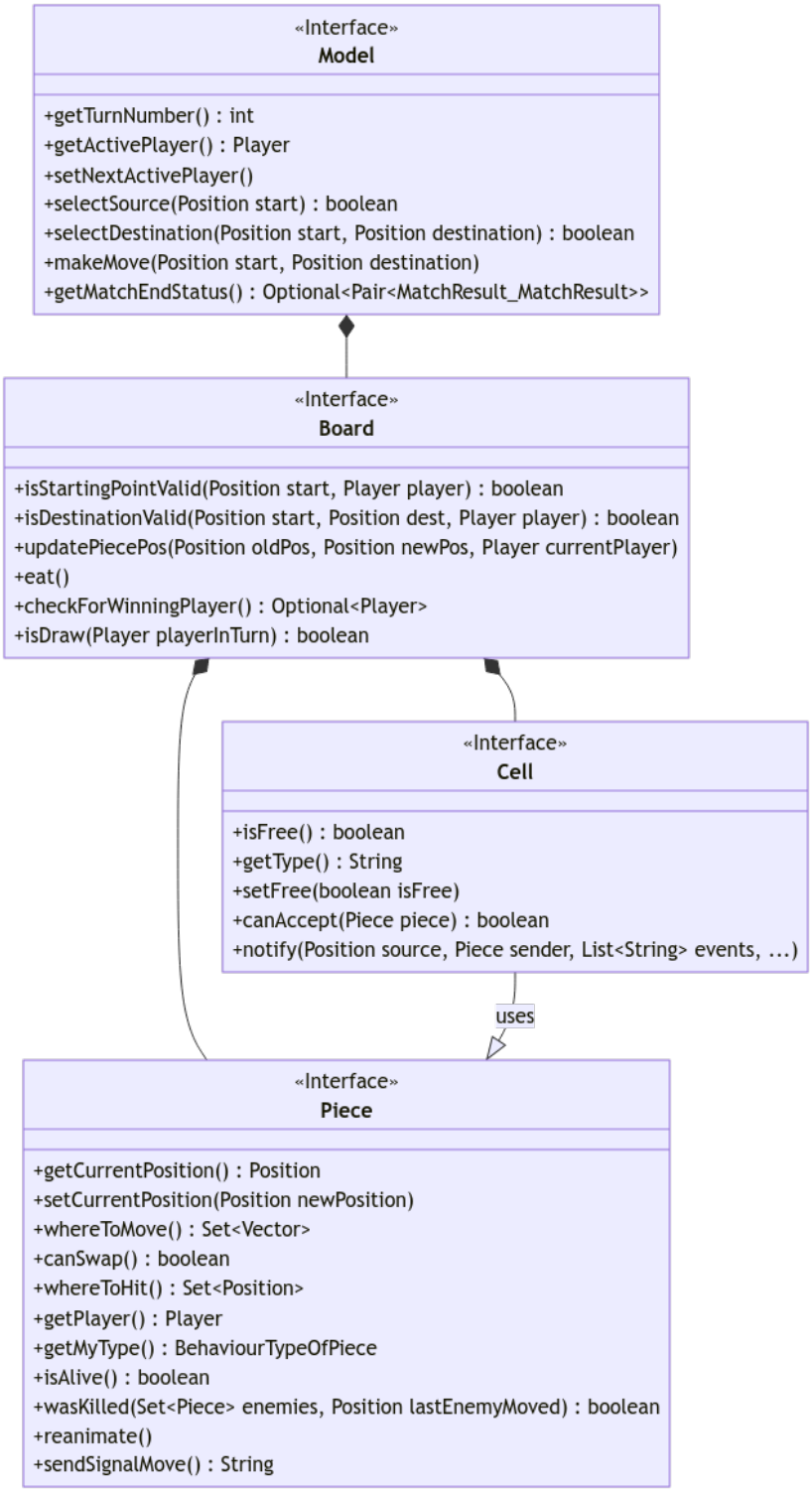

Figura 1.1: Schema UML del modello del dominio.

Capitolo 2

Design

2.1 Architettura

Come pattern architetturale, è stato scelto MVC in quanto permette di separare la logica dell'applicazione da ciò che viene mostrato all'utente, portando ad un'organizzazione interna dell'applicazione più pulita e più facilmente gestibile.

L'interfaccia `Model` racchiude la logica del dominio di gioco, isolandola da tutti gli altri componenti dell'architettura. Idealmente, il gioco potrebbe funzionare anche senza alcuna componente grafica, direttamente ricevendo input da linea di comando.

Il `Controller` si occupa del coordinamento fra Model e View, permettendo ai cambiamenti di stato della partita di essere visualizzati dall'utente. Il Controller gestisce anche alcune risorse non inerenti alla parte grafica dell'applicazione, ad esempio file di configurazione per il setup della partita e file per il salvataggio e il caricamento della leaderboard.

La `View` si occupa del rendering grafico delle varie schermate dell'applicazione. A ciascuna schermata è associata una specifica `Scene` e ciascuna di esse è legata ad un proprio `SceneController` che gestisce il comportamento della scena e la comunicazione tra la `Scene` stessa, la `View` (la quale è, in pratica, un container di `Scene`) e il `Controller`.

2.2 Design dettagliato

2.2.1 Alin Stefan Bordeianu

Annullamento delle mosse

L'utente deve avere la possibilità di annullare la propria mossa, indipendentemente dal numero di eventi che questa ha causato, e lo stato della partita a seguito di un annullamento deve tornare precisamente alla situazione precedente. Mentre per la modalità classica questo compito è relativamente facile poiché a variare sulla `Board` sono solo la collocazione delle pedine ed il loro numero in seguito a delle mangiate, nella modalità variante intervengono più elementi oltre a questi, come ad esempio le **tombe** ed il loro contenuto, gli **sliders** ed il loro orientamento, movimenti articolati come quello degli **swappers** e così via.

In ottica di estensibilità e nel rispetto del *single-responsibility principle*, si rende necessario semplificare il più possibile il compito dell'unità responsabile dell'annullamento, evitando che questa sia a conoscenza dei dettagli interni di ciascuna delle entità da ripristinare ad uno stato precedente. Un altro importante problema è dato dal fatto che a ciascuna entità di cui si vuole salvare lo stato sia poi assegnato correttamente il *proprio* stato. Infatti, essendo presenti molteplici entità concettualmente simili (ad esempio, varie `Piece` e varie `Cell`), è importante organizzare gli elementi che comporranno lo stato salvato (uno **snapshot**) in maniera da riuscire ad attribuire poi a ciascuna entità il proprio stato precedente.

Infine, l'annullamento riguarda lo stato dell'intera partita, perciò tutte le entità principali (`Match`, `Board`, `Piece` e `Cell`) devono essere ripristinate in maniera sequenziale, facendo attenzione a non creare conflitti o paradossi (del tipo, ripristinare correttamente celle e pedine sulla griglia, ma non permettere più al giocatore di turno di riprovare con una nuova mossa perché il turno non viene ripristinato).

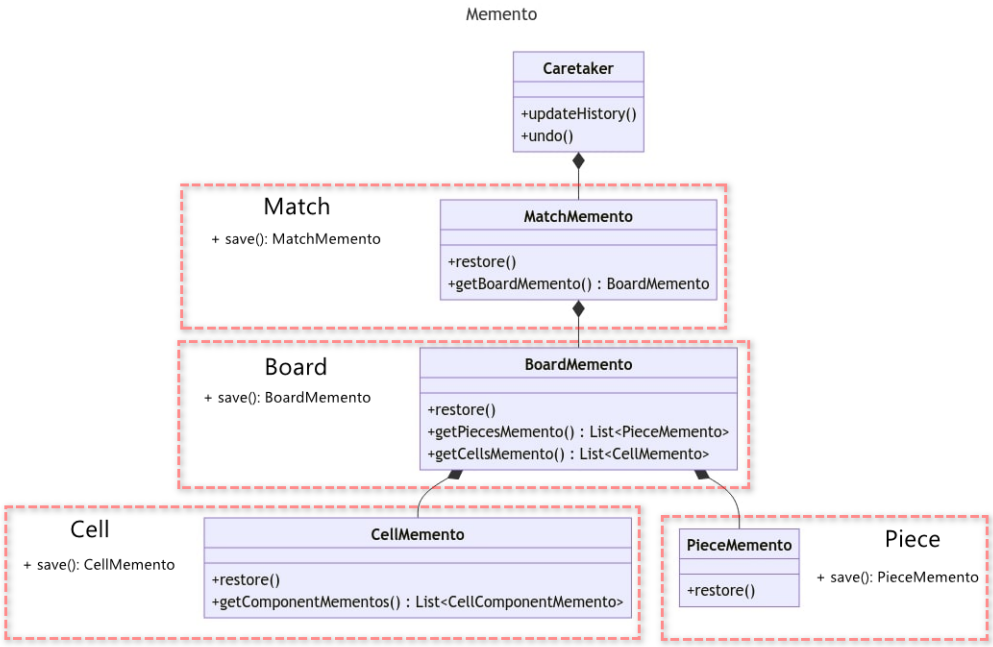

Figura 2.1: Pattern Memento tramite Inner Classes. In figura si vede anche che ogni entità del Model ha un metodo pubblico `save()`, mentre il metodo `restore()` fa parte delle Inner Classes, che corrispondono agli snapshot delle entità. I rettangoli rossi rappresentano le Outer Classes. Per quanto concerne i `CellComponentMemento`, ci saranno più dettagli in seguito.

Problema: salvare lo stato di molteplici entità in un contenitore centrale su cui sia semplice operare, in una maniera indipendente dal contenuto delle entità e in modo tale da poter associare a ciascuna entità il proprio stato salvato. Il processo deve essere il più automatico possibile, in quanto ripristinare lo stato su ciascuna singola entità "manualmente" è estremamente brigoso a livello di chiamate di codice e può causare errori facilmente.

Soluzione: si è adottato un sistema che riprende molti concetti del pattern Memento nella sua variante che sfrutta le Inner Classes del linguaggio Java, come visibile nella figura 2.1. Tali Inner Classes sono pubbliche e per loro natura hanno accesso ai campi, anche privati, della Outer Class che le contiene. La variazione rispetto al pattern tuttavia consiste nel fatto che la chiamata al metodo di ripristino `restore()` non avviene sull'Originator (ossia l'entità che ha generato lo stato salvato, chiamato da qui in poi **"me-**

mento"), ma sul memento stesso. Il vantaggio di questo approccio è che il memento, che altro non è che un'istanza di una Inner Class, ha sempre un riferimento alla Outer Class che lo ha generato. In questa maniera il Caretaker, cioè l'entità con lo scopo di mantenere lo storico dei memento, **non deve preoccuparsi di fare l'associazione fra ogni memento e l'entità che lo ha creato.**

Per quanto riguarda il ripristino delle entità nel corretto ordine, i vari memento sono stati composti fra di loro; in altre parole, a livello più basso ci sono i `CellMemento` e i `PieceMemento` che contengono, salvo eccezioni, soltanto informazioni sullo stato delle entità a cui si riferiscono; il `BoardMemento` contiene una collezione di `CellMemento` e `PieceMemento` oltre alle informazioni rilevanti sullo stato della `Board`, e infine il `MatchMemento` contiene un `BoardMemento`. Per salvare lo stato di una partita si crea un oggetto `CaretakerImpl`, il quale implementa l'interfaccia `Caretaker` e mantiene un riferimento ad una partita, e si chiama su di esso il metodo `updateHistory()`. Questo genera una catena di chiamate ai vari metodi pubblici `save()` delle entità che costituiscono il modello del dominio, e il risultato finale sarà un semplice `MatchMemento` che verrà conservato all'interno di uno stack (inteso come la struttura dati) nel CaretakerImpl. Per l'annullamento della mossa, sarà sufficiente chiamare il metodo `undo()` del Caretaker, che causerà la chiamata ai vari metodi `restore()` dei memento.

Questo sistema semplificato di salvataggio degli stati della partita può essere ulteriormente espanso, portando potenzialmente alla creazione di replay che siano costituiti dal semplice susseguirsi ordinato dei MatchMemento. Al momento l'applicazione consente di tornare indietro soltanto di una mossa, per due motivi:

- tornare indietro di più di una mossa significherebbe annullare anche le mosse dell'avversario, cosa che nel contesto di un gioco da tavolo non considero appropriata;
- il Caretaker potrebbe benissimo mantenere in memoria più di uno snapshot in linea teorica, ma esiste il rischio che tutti gli oggetti memento occupino molta RAM causando ritardi dell'applicazione. È stata presa in considerazione l'idea di salvare ciascun oggetto memento su file invece di tenerlo in memoria, ma per mancanza di tempo questa soluzione non è stata tentata.

Leaderboard

Si vuole tenere traccia dei risultati dei giocatori, costruendo una classifica che mostri il nome di ogni giocatore che abbia voluto registrare il risultato di una partita ed il risultato ottenuto. Tale classifica, anche denominata **leaderboard**, deve essere ordinata in base al numero di vittorie ottenute; nel caso in cui ci siano dei pareggi fra giocatori, l'ordinamento avviene considerando il numero di sconfitte (chi ne ha subite di meno avrà il posto in classifica più alto). I giocatori non avranno dei profili veri e propri, ma saranno identificati semplicemente da un nickname non vuoto che inseriranno al termine di ogni partita di cui vorranno registrare il risultato. La leaderboard deve anche poter essere svuotata. La leaderboard costituisce un dato persistente, che dunque andrà salvato su file.

Per effettuare il salvataggio su file è stata utilizzata la libreria esterna **SnakeYaml** (vedi documentazione qui), in quanto le associazioni fra giocatori e punteggi erano facilmente rappresentabili da una Map e la sintassi di Yaml consentiva un facile passaggio da e su file. Un problema riscontrato in questa fase è stato quello di trovarsi alle prese con il salvataggio di oggetti generici `Pair<X, Y>` che causavano difficoltà non indifferenti per via dei generici non reificati del linguaggio Java, ma alla fine ho optato per una soluzione più semplice che facesse uso di variabili facilmente gestibili come stringhe e interi, costruendo le Pair dinamicamente dopo la lettura di tali dati dal file.

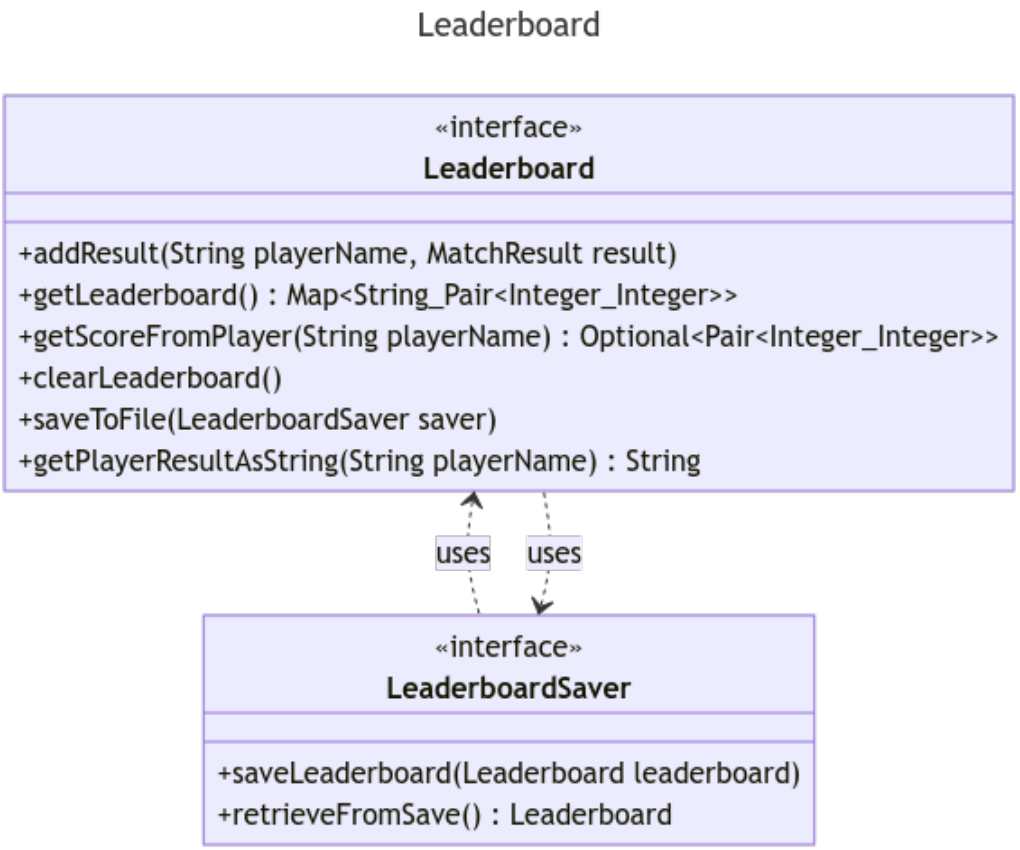

Figura 2.2: Diagramma UML che mostra i metodi delle interfacce Leaderboard e LeaderboardSaver.

La logica del salvataggio è stata inserita nel `LeaderboardSaver` (figura 2.2), mentre la gestione della leaderboard intesa come possibilità di aggiungere risultati e mostrare una classifica ordinata è affidata alla classe `Leaderboard` (figura 2.2). Il metodo `getPlayerResultAsString(String playerName)` è presente in vista di una potenziale funzionalità di filtraggio dei risultati della classifica, ma al momento per ragioni di tempo non è stato sfruttato nella creazione dell'interfaccia grafica. La `Scene` riservata alla visualizzazione della leaderboard è la `HighScoreScene`, pilotata dal `HighScoreController`. All'utente è consentito visualizzare e svuotare la leaderboard.

Tombe come CellComponents

2.2.2 Elena Boschetti

Match setup

La fase di setup della partita prevede la creazione della griglia di gioco e la disposizione delle pedine su di essa; in termini di codice, ciò corrisponde alla creazione di un oggetto di tipo `Board` e all'inizializzazione delle collezioni di celle e pedine che essa conserva internamente.

La costruzione delle due collezioni fa uso di due diversi builder: `CellsCollectionBuilder` per la collezione di celle e `PiecesCollectionBuilder` per la collezione di pedine. Ogni metodo delle interfacce dei builder permette di aggiungere alle collezioni un sottoinsiemi di pedine e celle di un determinato tipo. Da notare che i builder ricevono le coordinate a cui posizionare celle e pedine dall'esterno (in quanto passate come parametri ai diversi metodi), quindi i builder operano in maniera indipendente dal modo in cui la configurazione iniziale di celle e pedine viene fornita.

La configurazione iniziale dipende dalla modalità di gioco (classica o variante). Il caricamento della configurazione per ogni modalità viene avviato mediante l'interfaccia `SettingsLoader`. Dietro tale interfaccia, si possono definire diverse possibilità per quanto riguarda il reperimento della configurazione, creando per ciascuna di esse una classe che implementi l'interfaccia. L'implementazione di `SettingsLoader` da me definita prevede il caricamento della configurazione da un file XML (uno per ogni modalità di gioco), ma sarebbe possibile definire altre classi che la ottengano in maniera differente. È una possibilità che potrebbe essere utile anche nel caso in cui si verifichino problemi: ad esempio, se il caricamento da file XML non va a buon fine per qualche motivo, si può decidere di procedere per una via alternativa, creando un nuovo tipo di loader che cerchi di reperire la configurazione in altro modo.

Per costruire le collezioni di pedine e celle secondo la configurazione, il `SettingsLoader` riceve un `CellsCollectionBuilder` e un `PiecesCollectionBuilder` e interagisce con essi per creare e posizionare correttamente ogni tipologia di celle e pedine; ad esempio, il `SettingsLoader` ottiene le posizioni delle Exit Cells e successivamente chiama il metodo `addExits()` del `CellsCollectionBuilder`, passando tali posizioni. Una volta completata la costruzione delle due collezioni, esse vengono passate alla `Board` che viene istanziata, la quale viene a sua volta passata all'istanza di `Match` che viene creata.

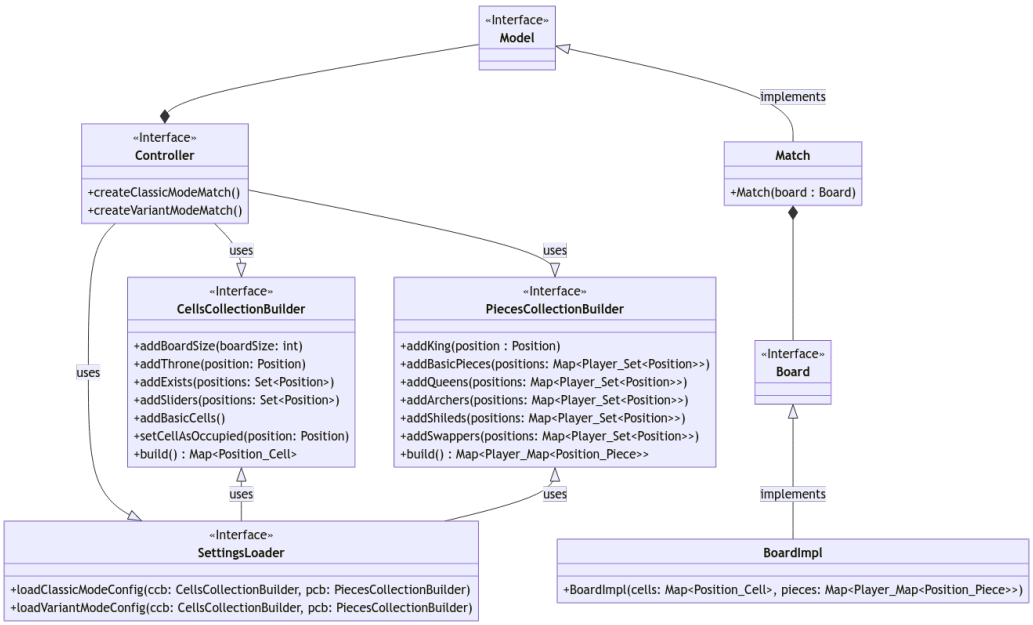

Figura 2.3: Schema UML che rappresenta l'interazione tra le entità coinvolte nel setup.

Scenes

La view dell'applicazione, realizzata utilizzando la Java Swing API, è composta da un serie di scene. L'implementazione della `View` prevede la creazione del container più esterno della view (un `JFrame`) e l'alternarsi delle diverse scene viene gestito impostando il frame con un opportuno layout (`CardLayout`) che consente di sostituire il container interno (un `JPanel`) che viene mostrato, il quale corrisponde alla scena.

Considerando la chiara differenza di ogni scena a livello grafico e le differenze di comportamento che ogni scena può avere (ad esempio, in risposta ad un input dell'utente), ho deciso di racchiudere il concetto di scena in un'entità a sè stante. L'interazione diretta con ogni scena avviene mediante l'interfaccia `Scene`, la quale consente di ottenere il container che costituisce la scena e di interagire con essa, ad esempio per il suo aggiornamento mediante il metodo `update()`. L'implementazione della `View` ha un riferimento alla `Scene` attualmente mostrata, il quale permette che un'interazione con la `View` abbia un effetto sulla scena corrente; ad esempio, se il controller dell'applicazione richiede l'aggiornamento della view mediante il metodo `update()` della `View`, in tale metodo viene richiamato il metodo `update()` della scena corrente.

Riflettendo sugli aspetti comuni che possono esserci tra le scene (ad esempio, la creazione del container e alcune sue impostazioni, oppure l'uso di altri parametri comuni, come il font) ho ritenuto che fosse utile definire una classe astratta (`AbstractScene`) che implementasse tali aspetti comuni tra le scene.

Le implementazioni concrete di ogni scena estendono `AbstractScene` e definiscono tutto ciò che è inerente alla specifica scena; in particolare, ogni classe costruisce la parte grafica della scena e definisce altri eventuali metodi utili all'interazione dalla `View` alla scena. Ad esempio, qualora una scena debba avere un aggiornamento a seguito di un input dell'utente, esso può essere innescato chiamando il metodo `update()`, come detto precedentemente.

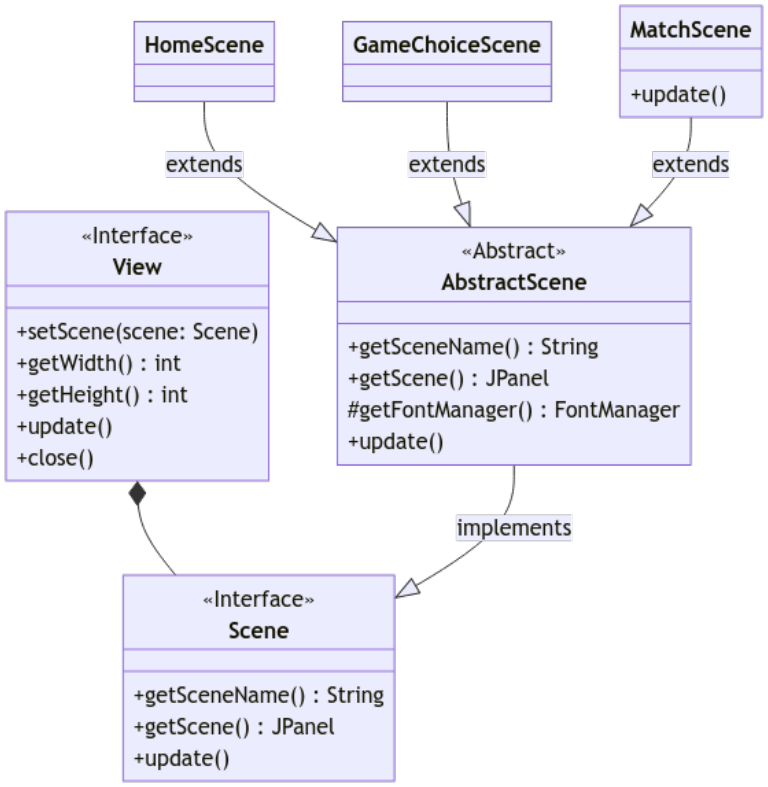

Figura 2.4: Schema UML che rappresenta il rapporto tra le entità che descrivono le scene della view. Sono mostrate solo le classi concrete che definiscono la schermata iniziale dell'applicazione (`HomeScene`), la schermata della scelta della modalità di gioco (`GameChoiceScene`) e la schermata della partita (`MatchScene`) a titolo esemplificativo.

In seguito a questa fase di design, mi sono occupata di implementare la scena della home del gioco (`HomeScene`), la scena della scelta della mo-

dalità di gioco (`GameChoiceScene`) e della scena che mostra il regolamento (`RulesScene`).

Scene Controllers

Nello sviluppo delle funzionalità delle scene, si è rivelato quasi sempre necessario che la scena interagisse con la `View` (che, come detto precedentemente, ha la funzione di wrapper della scena) o con il Controller dell'applicazione. Per questo motivo, ho deciso di introdurre delle entità, a cui mi riferisco con il termine "scene controllers", che regolano il comportamento della scena e che hanno il ruolo di mediatore tra `Scene`, `View` e `Controller`.

Tutte le scene hanno alcune funzionalità e necessità in comune, come il passaggio alla scena successiva o precedente e la necessità di conoscere le dimensioni della view per dimensionare i componenti interni; per questo motivo, ho deciso di raggruppare queste in un'unica interfaccia (`BasicSceneController`). Successivamente, ho notato che gli scene controllers condividevano anche parte della loro implementazione: ad esempio, contengono tutti un riferimento alla `View` e al `Controller` e ottengono dalla view le informazioni utili per il dimensionamento nello stesso modo; ho quindi deciso di realizzare una classe astratta (`AbstractSceneController`) che implementasse tali aspetti comuni.

Ogni scena ha poi le proprie specifiche funzionalità; ciò rende necessario che lo scene controller di ogni scena fornisca degli ulteriori metodi per poterle attuare, oltre a quelli stabiliti nel `BasicSceneController`. Ad esempio, Lo scene controller della home scene permette anche il passaggio alla scena che mostra gli high scores e permette di comunicare alla `View` la richiesta di chiusura dell'applicazione. Lo scene controller della scena di scelta del gioco, invece, permette la creazione della partita nella modalità scelta e permette di passare alla scena che mostra il regolamento di gioco. Ho quindi deciso che, per ogni scena, il rispettivo scene controller dovesse avere una propria interfaccia, che estendesse `BasicSceneController` (si veda la Figura 2.5 in relazione agli esempi appena fatti).

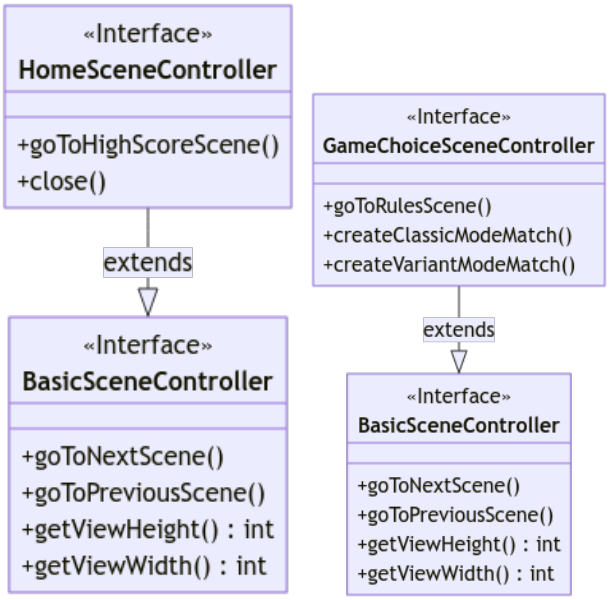

Figura 2.5: Due esempi di interfaccia di due scene controllers.

L'implementazione di ogni scene controller deve quindi implementare la sua specifica interfaccia e estendere `AbstractSceneController`; di fatto, essa consiste solo nell'implementazione delle funzionalità specifiche della rispettiva scena.

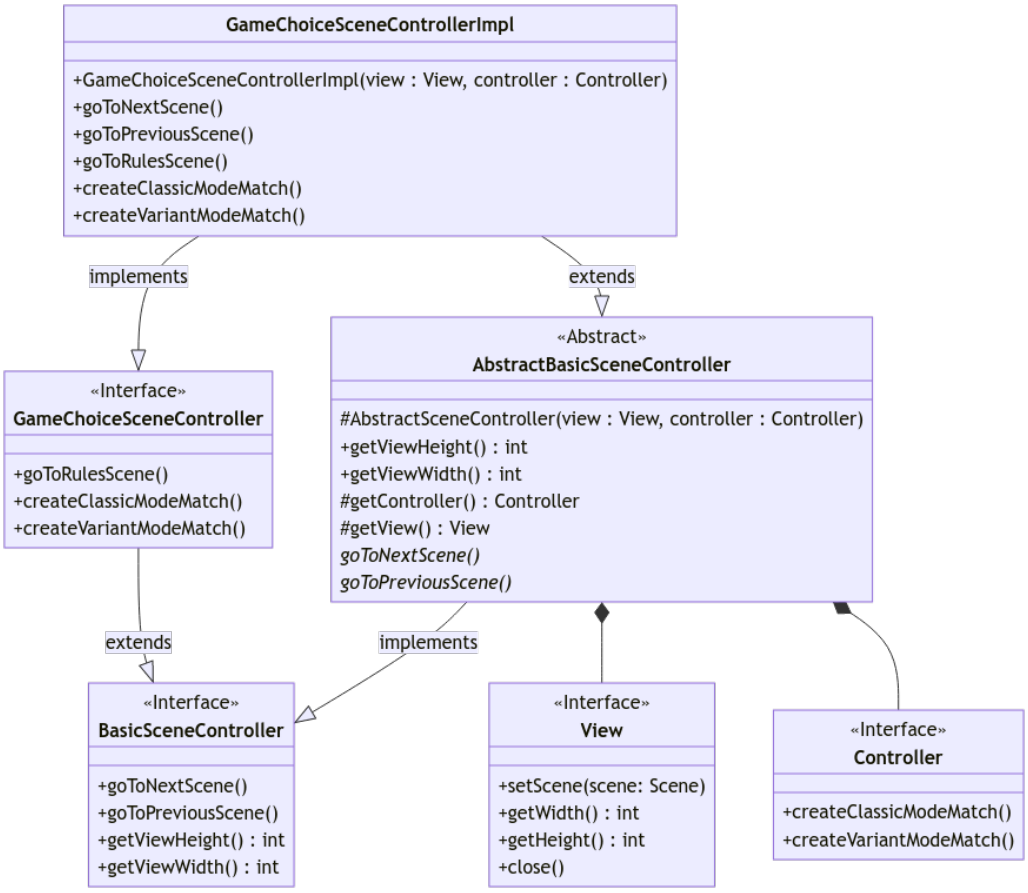

Figura 2.6: Schema UML che rappresenta il design di uno scene controller (in questo caso, lo scene controller della scena di scelta del gioco).

In seguito a questa fase di design, mi sono occupata di implementare gli scene controller delle scene di mia competenza.

2.2.3 Andrea Piermattei

2.2.4 Margherita Raponi

Capitolo 3

Sviluppo

3.1 Testing automatizzato

3.2 Metodologia di lavoro

3.3 Note di sviluppo

3.3.1 Alin Stefan Bordeianu

3.3.2 Elena Boschetti

Parsing di file XML utilizzando le funzionalità fornite dal package `javax.xml.parsers` del JDK

Permalink: [utilizzo in SettingsLoaderImpl.java]

Uso di `LoopingIterator` dalla libreria Apache Commons Collections

Permalink: [utilizzo in Match.java]

Utilizzo della libreria SLF4J

Utilizzata in più punti. Permalinks:

- utilizzo in ControllerImpl.java
- utilizzo in TestBoardSetup.java
- utilizzo in TestMatch.java

Uso di `Stream` e lambda expressions

Permalink: [utilizzo in TestBoardSetup.java]

Uso di reflection

Utilizzata in più punti nel test del setup della partita. Permalink: [un utilizzo in TestBoardSetup.java]

Scrittura di metodo generico

Permalink: [utilizzo in TestBoardSetup.java]

Uso di `Optional`

Permalink: [utilizzo in Match.java e/o TestMatch.java]

Uso di `javax.swing.text.html.HTMLEditorKit` per il rendering di file HTML

Permalink: [uso in RulesScene.java]

3.3.3 Andrea Piermattei

3.3.4 Margherita Raponi

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.2 Difficoltà incontrate e commenti per i docenti

Appendice A

Guida utente

Appendice B

Esercitazioni di laboratorio

B.0.1 alinstefan.bordeianu@studio.unibo.it

B.0.2 margherita.raponi@studio.unibo.it