Tafl Games

Relazione per il progetto
del corso di
Programmazione ad Oggetti
A.A. 2022/23

Alin Stefan Bordeianu
Elena Boschetti
Andrea Piermattei
Margherita Raponi

9 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.

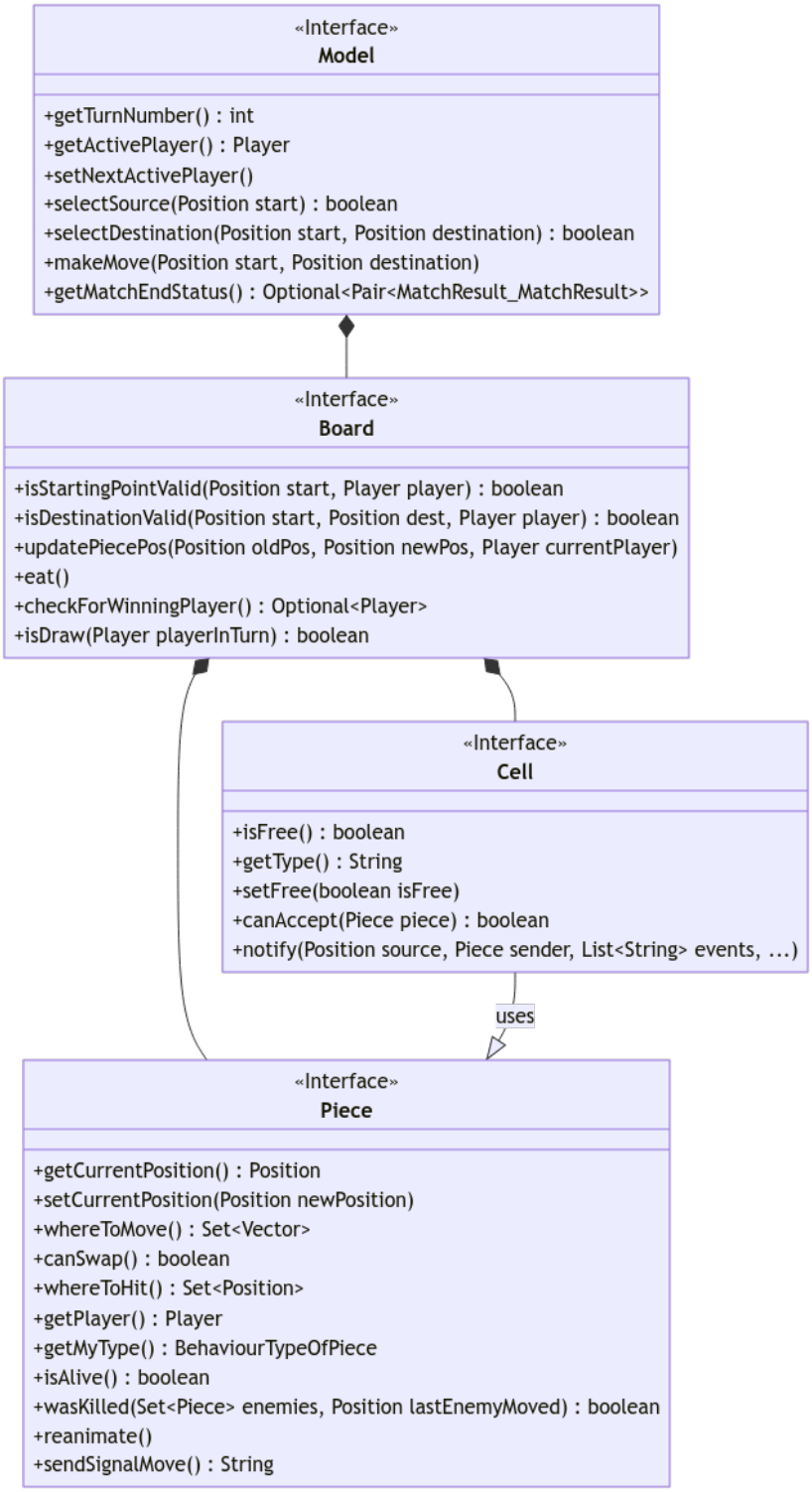

Figura 1.1: Schema UML del modello del dominio.

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, ricevendo direttamente l'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 rapporto tra Controller e Model si limita all'inizializzazione dello stato del Model (il setup della partita) e alle richieste del Controller al Model per ottenere informazioni sul suo stato interno.

Un passaggio del turno o l'annullamento di un mossa durante la partita implicano un cambiamento nello stato interno del Model (come minimo, la mossa di una pedina); tali cambiamenti devono essere riflessi dalla View, per cui il Controller segnala alla View la necessità di un'aggiornamento. A questo punto, la View si occupa di richiedere le informazioni necessarie al Controller, il quale le chiede a sua volta al Model. Il Controller gestisce, inoltre, 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.

Per quanto riguarda la View, essa è composta da più scene che si alternano tra loro, ognuna delle quali è rappresentata dall'entità `Scene`; in pratica, la View è un container di `Scene`. La comunicazione della Scene verso il Con-

troller (ad esempio, per la richiesta dei dati necessari all'aggiornamento della view della board) e verso la View (ad esempio, per il cambio di scena) avviene mediante delle entità intermedie, indicate con il termine "scene controllers"; ogni scena ha un proprio scene controller, che si occupa dell'invio delle richieste specifiche di quella scena. Questa scelta è stata fatta per dividere le diverse responsabilità legate alla scena: la `Scene` si occupa di costruire la parte strettamente grafica, mentre le interazioni con la View e il Controller legate alle funzionalità della scena sono delegate allo `Scene Controller`.

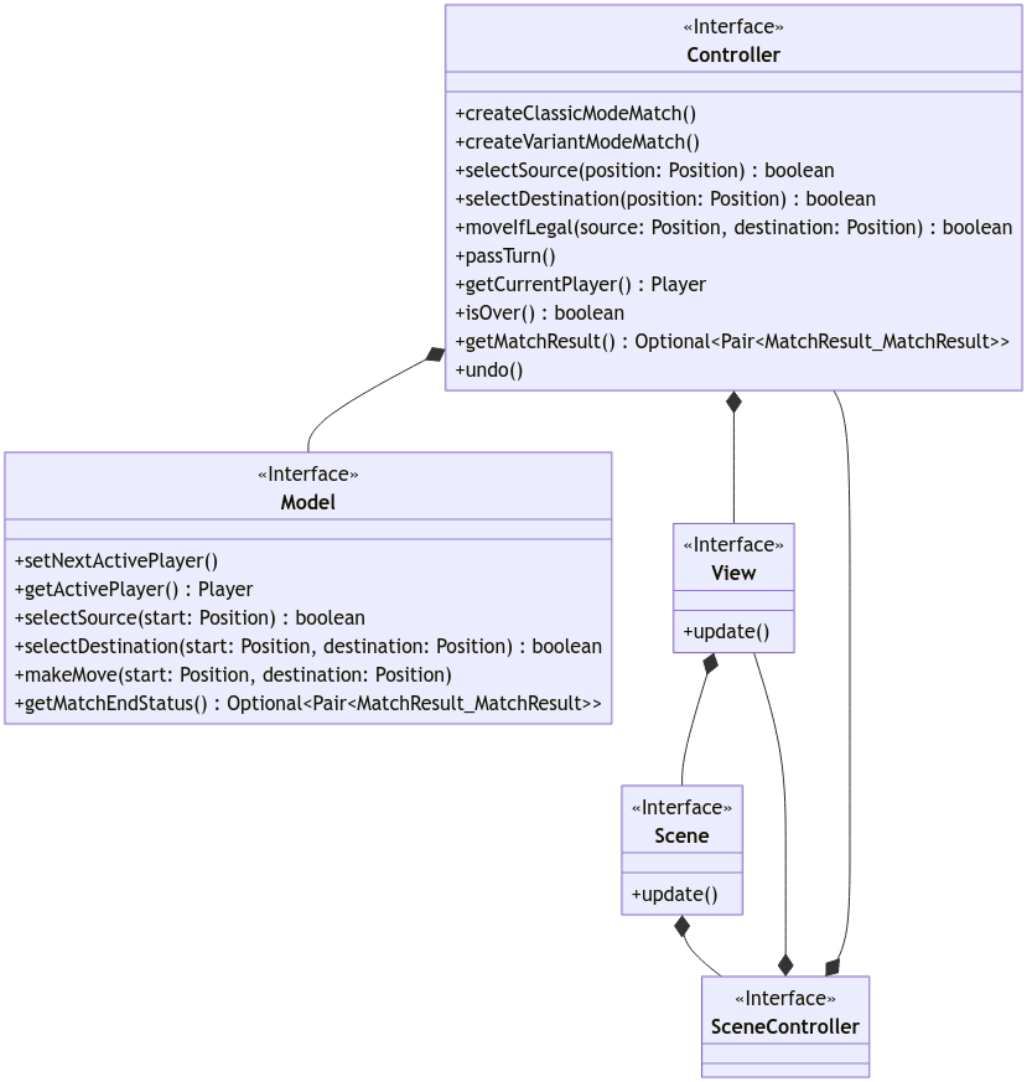

Figura 2.1: Schema UML dell'architettura dell'applicazione.

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 si deve poi assegnare correttamente il *proprio* stato. Infatti, essendo presenti molteplici entità concettualmente simili (ad esempio, varie `Pieces` e varie `Cells`), 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).

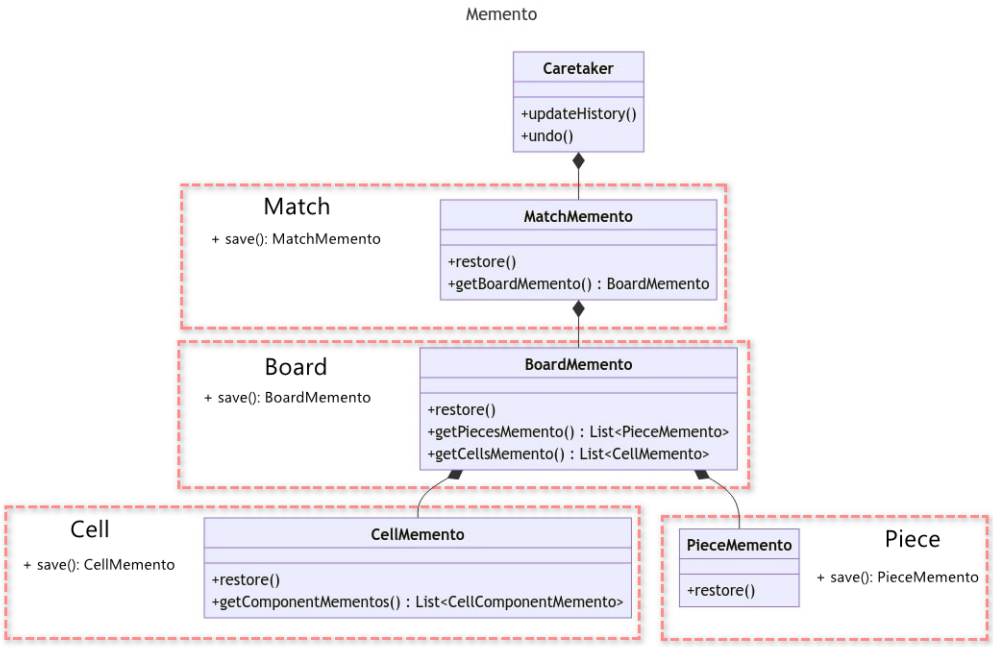

Figura 2.2: Pattern Memento tramite Inner Classes. In figura si vede anche che ogni entità del Model ha un metodo pubblico `save()`, mentre il metodo `restore()` fa parte delle Inner Classes, che corrispondono agli snapshot delle entità. I rettangoli rossi rappresentano le Outer Classes. Per quanto concerne 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 2.2. 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** (si veda la 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 in questo caso. 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.

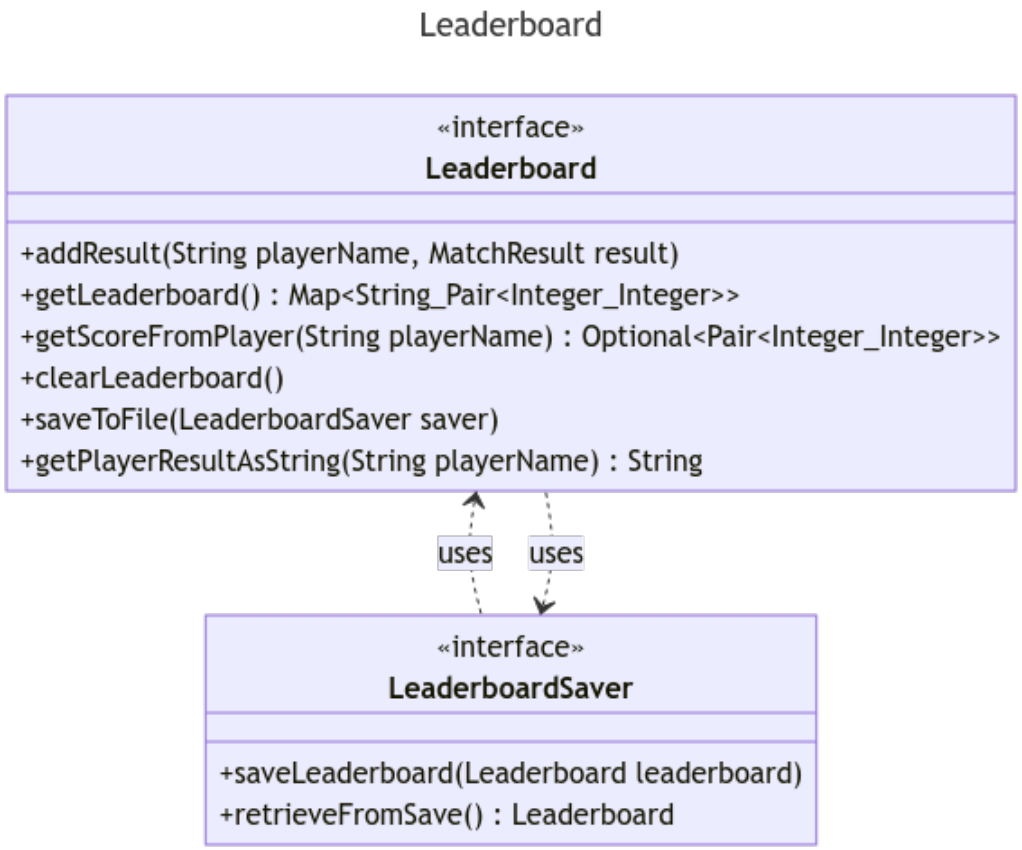

Figura 2.3: Diagramma UML che mostra i metodi delle interfacce Leaderboard e LeaderboardSaver.

La logica del salvataggio è stata inserita nel `LeaderboardSaver` (figura 2.3), mentre la gestione della leaderboard intesa come possibilità di aggiungere risultati e mostrare una classifica ordinata è affidata alla classe `Leaderboard` (figura 2.3). 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

La Tomba è un'entità particolare della modalità variante che viene piazzata sulla griglia di gioco in corrispondenza delle coordinate a cui è morta una

qualsiasi pedina. Durante la fase di analisi la `Tomb` è sempre stata considerata come una `Cell` a tutti gli effetti. Tuttavia quando si è giunti al punto di dover scrivere il codice che creasse le Tombe una volta morte le pedine, sono emersi alcuni problemi legati a questa visione della Tomba:

- la Tomba non deve rimpiazzare gli eventuali effetti delle celle su cui viene generata;
- se una Tomba è vuota, deve poter essere rimossa tranquillamente senza il rischio di lasciare un buco nella griglia, e preferibilmente senza che entità esterne debbano memorizzare che tipo di cella era collocata a tali coordinate in precedenza;
- lo stato della Tomba deve poter essere salvato agevolmente per permettere al giocatore di annullare una mossa in maniera coerente con le altre entità che producono dei memento, ma anche lo stato della cella precedente all'inserimento della Tomba deve essere mantenuto, il che implica che la cella precedente non dovrebbe essere eliminata.

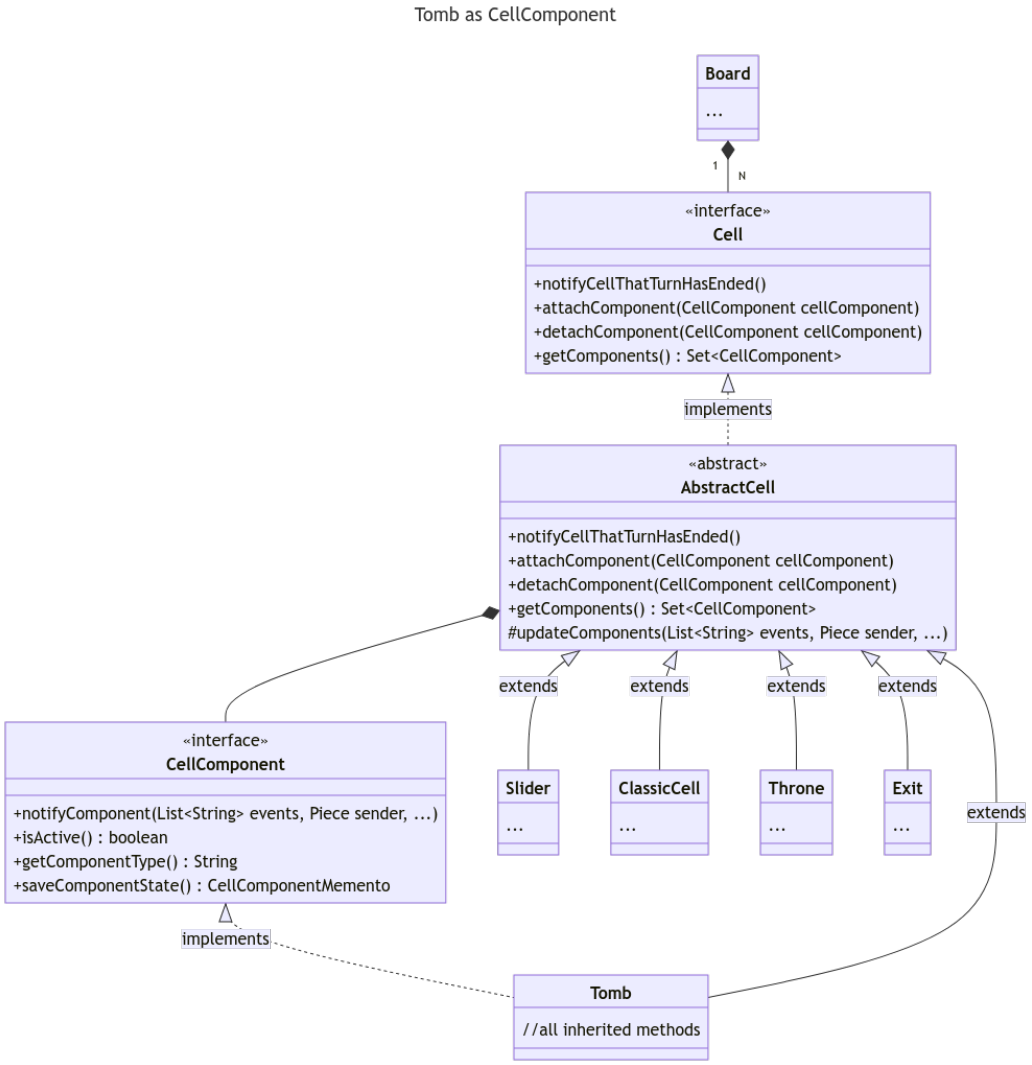

Figura 2.4: `Tomb` viene ora trattata come un CellComponent, che può essere attaccato o staccato a seconda di certi eventi che la `Board` si occuperà di rendere noti alla `Cell`.

Problema: realizzare la `Tomb` in una maniera che catturi meglio la sua natura di caratteristica opzionale di `Cell` già esistenti, senza rimpiazzare le `Cells` già presenti alle coordinate della morte di una pedina.

Soluzione: si è scelto di trattare la Tomba non più come una Cella a sé stante, ma come un componente da poter attaccare e staccare alle `Cells` normali (figura 2.4). Per farlo è stato applicato il **pattern Composite**. Il **base component** è costituito dall'interfaccia `Cell`, che è il genere di en-

tità di alto livello con cui opera la `Board`, la quale funge da **client**. I **leaf components**, ossia quelli che non hanno riferimento agli altri componenti e forniscono una implementazione base dei metodi del componente di base, sono tutte le celle ad eccezione della `Tomb`: `ClassicCell`, `Throne`, `Exit` e `Slider`. A fare da **composite** c'è la classe astratta `AbstractCell`, che implementa `Cell` e contiene la logica per attaccare, staccare e aggiornare i componenti. Infine la `Tomb` viene ora trattata come un `CellComponent`, e per piazzarne una su una qualsiasi altra casella è sufficiente che il client (la `Board`) attacchi un nuovo componente ad una `Cell` (questo avviene indirettamente tramite vari metodi che notificano eventi che potrebbero essere rilevanti o meno per le specifiche caselle). Per evitare duplicazioni di componenti, l'`AbstractCell` ha un Set di `CellComponents`. Ogni `CellComponent` ha anche un metodo `isActive()` mediante il quale può essere monitorato per capire quando è possibile staccarlo. Più in dettaglio, se la `Tomb` non contiene più alcun pezzo perché tutti sono stati resuscitati, allora la `AbstractCell` stacca in autonomia tale componente quando viene notificata dalla `Board` che il turno è finito. *Nota: sebbene in questa sottosezione vengano nominate molte classi, soltanto i metodi che riguardano il salvataggio dei CellMemento e i CellComponents sono mia responsabilità.*

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.

Per la costruzione della collezione di celle e della collezione di pedine, sono usati due diversi builder: `CellsCollectionBuilder` e `PiecesCollectionBuilder`. Ogni metodo delle interfacce dei builder permette di aggiungere alle collezioni un sottoinsieme 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.

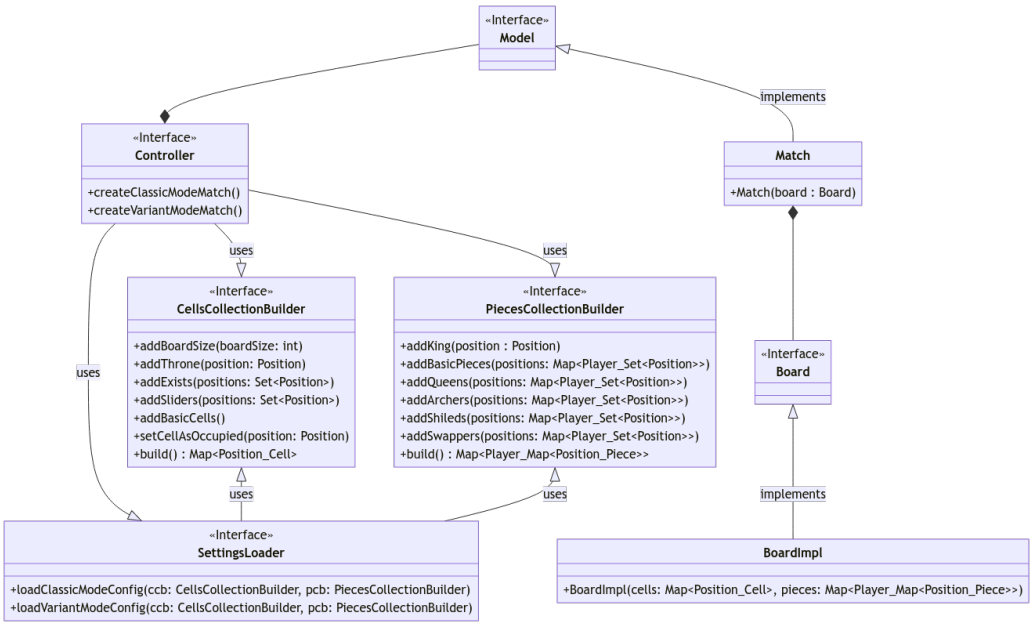

Figura 2.5: Schema UML che rappresenta l'interazione tra le entità coinvolte nel setup.

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 e l'alternarsi delle diverse scene viene ge-

stito impostando il frame con un opportuno layout che consente di sostituire il container interno che viene mostrato, il quale corrisponde alla scena.

Considerando la chiara differenza di ogni scena a livello grafico e di comportamento (ad esempio, in risposta ad un'azione dell'utente), ho deciso di racchiudere il concetto di scena in un'entità a sé stante. L'interazione diretta con ogni scena avviene mediante l'interfaccia `Scene`. 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, alcune impostazioni grafiche come lo sfondo o il font) ho ritenuto che fosse utile definire una classe astratta (`AbstractScene`) che si occupasse di implementare tali aspetti comuni, ad esempio creando una scena con alcuni parametri già impostati e di fornendone altri su richiesta.

Le implementazioni concrete di ogni scena estendono `AbstractScene` e definiscono ciò che è inerente alla specifica scena; in particolare, ogni classe costruisce la parte grafica della scena corrispondente e, se necessario, ridefinisce i metodi di interazione tra `View` alla scena, come il metodo `update()` menzionato precedentemente.

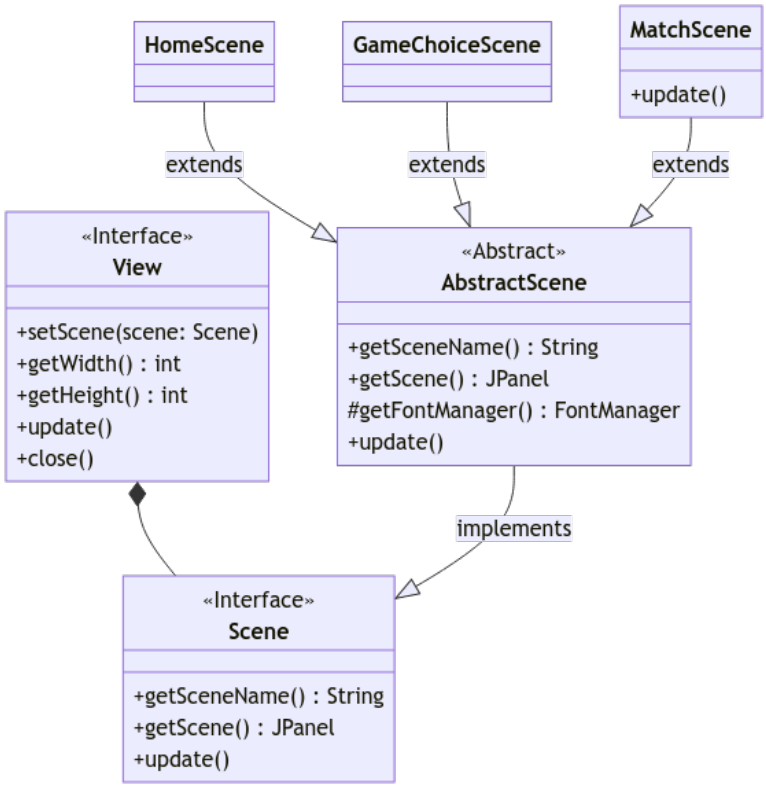

Figura 2.6: Schema UML che rappresenta il rapporto tra le entità che descrivono le scene della view. Sono mostrate solo le classi concrete che definiscono la schermata iniziale dell'applicazione (`HomeScene`), la schermata della scelta della modalità di gioco (`GameChoiceScene`) e la schermata della partita (`MatchScene`) a titolo esemplificativo.

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 par-

te 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 2.7 in relazione agli esempi appena fatti).

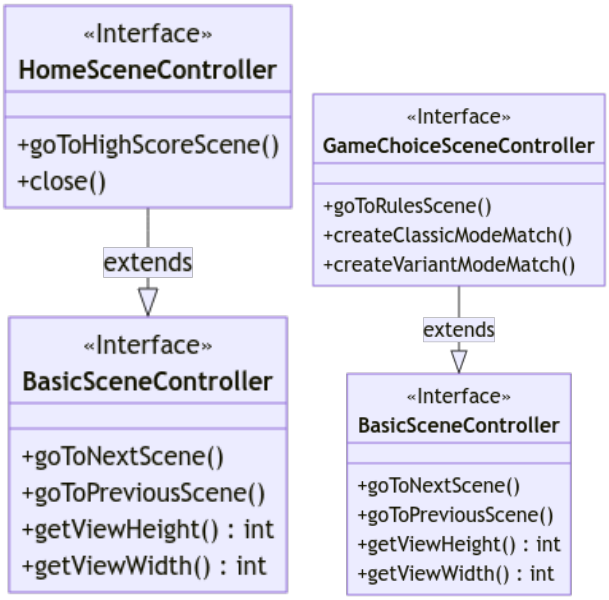

Figura 2.7: Due esempi di interfaccia di due scene controllers.

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.

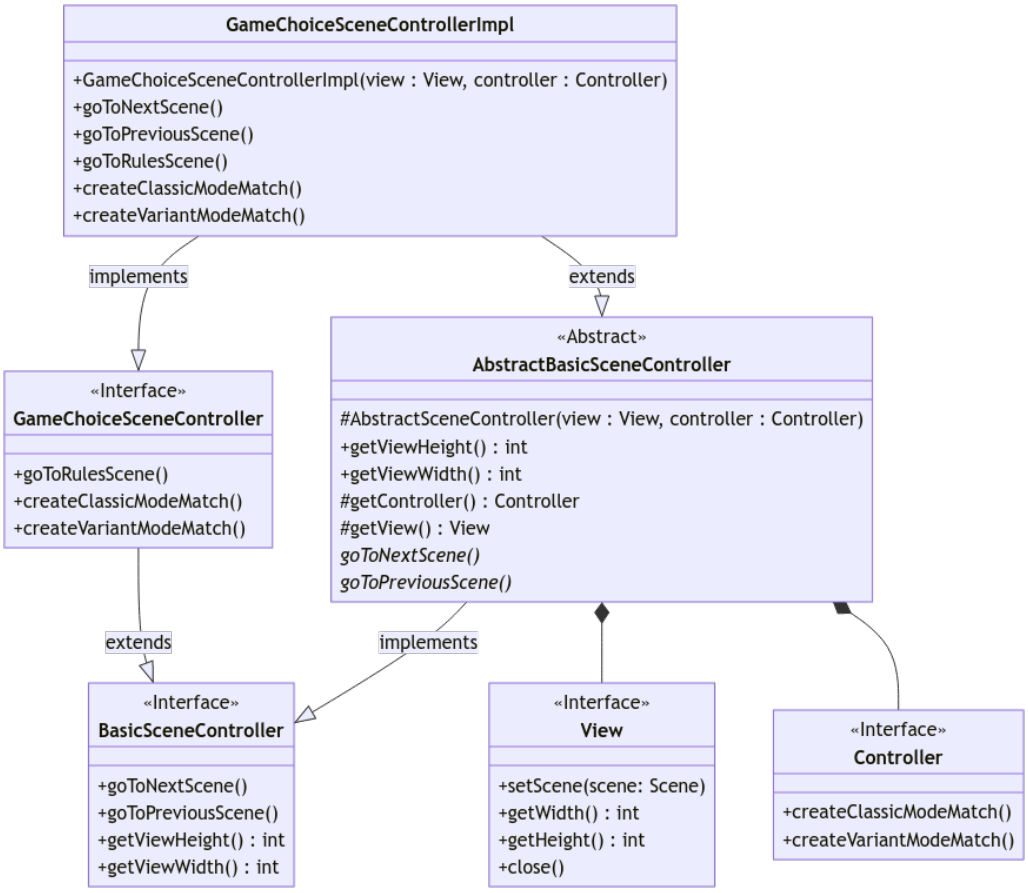

Figura 2.8: Schema UML che rappresenta il design di uno scene controller (in questo caso, lo scene controller della scena di scelta del gioco).

2.2.3 Andrea Piermattei

2.2.4 Margherita Raponi

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il testing automatizzato ha riguardato la parte logica dell'applicazione. Data la natura piuttosto articolata e "gerarchica" del dominio di gioco, il testing è stato portato avanti con un approccio bottom-up. Sono prima stati testati singolarmente i componenti più piccoli del dominio (celle e pedine), controllando che il loro stato interno fosse inizializzato correttamente. Essi sono successivamente stati testati sulla griglia di gioco, verificando che si ottenessero gli effetti desiderati dall'interazione tra le diverse pedine e celle. Chiaramente, è stata testata anche la corretta configurazione iniziale di celle e pedine secondo quanto stabilito dal regolamento delle due modalità di gioco. Infine, il tutto è stato testato nel contesto della partita, considerando quindi un ambiente di gioco completo e l'alternanza dei turni tra i giocatori. Per la scrittura e lo svolgimento dei test, è stato utilizzato il framework `JUnit 5`.

3.2 Metodologia di lavoro

La prima parte del progetto è consistita nella definizione delle entità principali del dominio di gioco e nella definizione delle loro interfacce. In questa fase, abbiamo lavorato insieme, cercando di far emergere dal confronto le scelte migliori che potessero portarci ad un buon design architetturale da cui partire.

Una volta stabilita l'architettura di base, ciascun membro del gruppo si è dedicato al design dettagliato della parte del Model di propria competenza e successivamente alla sua implementazione, lavorando ciascuno su un branch personale. Considerando le inevitabili dipendenze tra le diverse parti del

dominio (il Match utilizza la Board e la Board utilizza l'implementazione delle pedine e delle celle), si sono rilvelati necessari alcuni accorgimenti per poter comunque permettere lo sviluppo delle parti in parallelo. Ad esempio, fino a che l'implementazione delle pedine non è stata completa, durante lo sviluppo della Board sono state impiegate delle classi fittizie da utilizzare come "placeholder" delle implementazioni reali, in modo da poter comunque sviluppare buona parte delle funzionalità della Board e renderla parzialmente testabile. Discorso simile vale per il Match e per il setup della partita finché non è stata completata l'implementazione della Board, con la differenza che il testing del Match è stato necessariamente fatto solo dopo che la Board è stata completata, in quanto le dipendenze tra le funzionalità del Match da quelle della Board sono più estese e una classe fittizia non permetteva un adeguato testing. A seguito dell'integrazione delle sottoparti complete (ad esempio, l'integrazione della Board nel Match), in fase di testing sono talvolta emersi alcuni bug che non si erano verificati testando la sottoparte singolarmente in un contesto non altrettanto completo, e ciò ha richiesto alcuni aggiustamenti.

Dopo che l'implementazione di tutte le parti di Model è stata completata e testata, è iniziata la costruzione della View. Parallelamente con essa, è stata portata avanti anche la costruzione del Controller, mano a mano che si delineavano le interazioni necessarie a ciascuna parte della View. Si rende noto, inoltre, che lo Scene Controller non era un elemento previsto nell'architettura iniziale ed è stato intodotto proprio durante questa fase.

Quando anche le funzionalità base della View e del Controller sono state realizzate, ci si è curati di migliorare l'applicazione, collaudando altre funzionalità (ad esempio, l'annullamento delle mosse), risolvendo alcuni bug di lieve entità e apportando alcuni rifinimenti grafici.

Durante tutta la fase implementativa del progetto, è stato utilizzato Git come DVCS, secondo le modalità descritte a lezione: ogni volta che un componente del gruppo ha avuto necessità di lavorare su una particolare feature, è sempre stato creato un branch secondario su cui svolgere tutto il lavoro fino completamento della feature, dopo il quale veniva fatto il merge del feature branch nel branch `develop`.

3.2.1 Alin Stefan Bordeianu

In autonomia mi sono occupato di:

- creazione delle classi `Position`, `Vector` e `VectorImpl` (package taflgames.common.api e taflgames.common.code), che costituiscono il principale mezzo di comunicazione di tutti gli elementi dell'architettura;

- creazione del `Caretaker`, del sistema di registrazione degli stati della partita mediante il pattern Memento, scrittura delle InnerClass che implementano le varie interfacce Memento nel caso delle `Cells` e della `Board` (package taflgames.model.memento.api e taflgames.model.memento.code);
- creazione della `Leaderboard`, del `LeaderboardSaver` e della relativa `Scene`: `HighScoreScene` e il proprio `HighScoreController`;
- creazione della classe `Installer` che si occupa della creazione del file di salvataggio della leaderboard nella home dell'utente;
- la creazione di un `CellImageMapper` e di un `PieceImageMapper` che potessero associare a ciascuna pedina e cella uno stato (rispettivamente `CellState` e `PieceState`) utilizzabile poi nella View (package taflgames.controller.entitystate e taflgames.controller.mapper);
- un `FontManager` e un `Limiter` (un oggetto che limitasse il numero di caratteri inseribili in dei JTextFields utilizzati per la registrazione dei risultati degli utenti);
- creazione della scena `UserRegistrationScene` e relativo `UserRegistrationController`.
- creazione della scena `GameOverScene` e relativo `GameOverController`.

In collaborazione ho lavorato:

- con Elena Boschetti:
  - sulla pianificazione di strategie per risolvere alcuni problemi fondamentali dell'applicazione, in particolare le dipendenze che le entità Tomb, Queen e Slider creavano fra i vari elementi del dominio;
  - sulla realizzazione di un sistema indipendente dal sistema operativo per caricare il file leaderboard.yaml, in cui vengono salvati i risultati della Leaderboard;
  - sul Controller dell'applicazione;
  - sul MatchMemento, concordando sull'interfaccia da esporre al Caretaker;
- con Andrea Piermattei:
  - sulla realizzazione del PieceMemento, concordando sull'interfaccia da esporre al Caretaker;

  - solo a livello di pianificazione, sul come strutturare il MatchPanel;

- con Margherita Raponi:

  - sulla creazione della Tomba mediata dalla Board, e sul sistema che l'ha resa un CellComponent;
  - sul debugging della Board e dell'Eaten;
  - sulla condizione di patta;
  - sui test delle varie Celle e comportamenti della Board.

Elena Boschetti

Di seguito, sono elencate la parti di cui mi sono occupata in autonomia.

- Mi sono occupata dell'implementazione dell'interfaccia del Model dell'applicazione, nello specifico della classe `taflgames.model.Match`.
- Ho realizzato i builder per la costruzione delle collezioni di celle e pedine (package `taflgames.model.builders`).
- Mi sono occupata del caricamento da file XML della configurazione della griglia di gioco in base alla modalità scelta e dell'inizializzazione delle collezioni di celle e pedine coerentemente con essa (package `taflgames.controller.settingsloader`);
- Mi sono occupata del design della view, sia per quanto riguarda le scene (package `taflgames.view.scenes`), sia per quanto riguarda gli scene controllers (package `taflgames.view.scenecontrollers`), e ho realizzato le interfacce e le classi di base di entrambi (`Scene`, `AbstractScene`, `BasicSceneController`, `AbstractBasicSceneController`).
- Ho implementato le scene e gli scene controllers della schermata della home dell'applicazione (`HomeScene`, `HomeSceneController` e relativa implementazione), della schermata di scelta del gioco (`GameChoiceScene`, `GameChoiceSceneController` e relativa implementazione) e della schermata del regolamento di gioco (`RulesScene`, `RulesSceneController` e relativa implementazione). Per quanto concerne il regolamento, mi sono occupata anche della sua stesura su file HTML, in modo da avere una resa grafica migliore.

Ho inoltre collaborato:

- con Alin Stefan Bordeianu:
  - per la risoluzione di alcuni problemi legati al rapporto tra le entità del dominio, confrontandoci per valutare idee diverse al fine di progettare delle buone soluzioni in termini di design, che minimizzassero le dipendenze tra le entità;
  - per l'implementazione dell'interfaccia `MatchMemento` all'interno della classe `Match`;
  - per il design e l'implementazione del Controller dell'applicazione;
  - per alcuni dettagli implementativi delle classi atte al caricamento e al salvataggio della leaderboard, in particolare per quanto riguarda il caricamento del file.
- con Margherita Raponi:
  - per l'integrazione tra Match e Board;
  - per il debugging della `Board` a seguito di problemi emersi in fase di testing del `Match` e della `Board`.

3.3 Note di sviluppo

3.3.1 Alin Stefan Bordeianu

Uso di Lambda expressions

(costruttore del TombMementoImpl, metodo getLeaderboard in LeaderboardImpl)

Uso di Stream

(TestLeaderboard, tutti i memento con le varie collezioni, vari metodi restore e save)

Uso di Optional

(LeaderboardImpl in getScoreFromPlayer())

Uso della libreria esterna SnakeYaml

(LeaderboardSaverImpl)

Uso della libreria SLF4J

(LeaderboardSaverImpl)

Link a parti di codice esterne e link alle risorse utilizzate:

- Codice della classe Pair presa dal docente: `https://bitbucket.org/mviroli/oop2022-esami/src/master/a01a/e1/Pair.java`
- Font "Latin Runes v2.0": `https://www.fontget.com/font/latin-runes/`
- Immagine di una plancia di legno da mettere sotto alle labels della GUI: pxhere.com

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

4.1.1 Alin Stefan Bordeianu

Ho avuto l'occasione di confrontarmi con problemi non banali e ho allenato le mie capacità di trovarvi soluzioni, collaborando all'interno di un team. Grazie al tempo speso seguendo le lezioni e facendo prove per conto mio, sono stato capace di sfruttare meccanismi avanzati del linguaggio Java come gli Stream e le Lambda che mi hanno permesso di scrivere più agevolmente del codice che fosse di qualità e anche facilmente testabile.

Penso di aver fatto un buon lavoro anche per quello che concerne le scritture dei test delle parti di mia competenza, che fortunatamente non hanno dato troppi problemi, ma anche sfruttando gli strumenti che mi venivano forniti dall'IDE in fase di debugging e di merge conflicts. Una delle principali vittorie che ho ottenuto è stata prendere confidenza con Git, strumento che avrei voluto conoscere fin dal primo giorno di università.

Sono soddisfatto di essere riuscito ad andare oltre agli elementi visti a lezione, apprendendo qualcosa in più sulla libreria SnakeYaml e sui file di configurazione.

Uno dei rimpianti che ho è non aver avuto molto tempo di ripassare sul codice scritto per effettuare operazioni di refactoring; ho scoperto alcune parti dove invece piccole modifiche avrebbero ripulito un po' il design (banalmente, per fare un esempio, le varie interfacce del Memento avevano tutte in comune un metodo void, restore(), che poteva stare bene in un'interfaccia a sé da cui tutte le altre estendessero).

L'autocritica maggiore che posso farmi è che ho la sensazione di non aver contribuito molto alla scrittura di parti davvero sostanziali dell'applicazione, per quanto il Memento sia stato divertente. Forse quest'ultima considerazio-

ne è data dal fatto che il mio obiettivo è arrivare a lavorare nell'ambito dello sviluppo dei videogiochi e non sento di aver toccato con mano le meccaniche fondamentali dello Hnefatafl. Se trovassi del tempo in futuro, mi piacerebbe molto estendere l'applicazione, magari introducendo la possibilità di registrare dei replay grazie al Memento, inserire suoni e nuove modalità di gioco, pedine o celle, e magari spaziare anche su più aspetti di grafica, Java Swing permettendo.

Elena Boschetti

Questo progetto è stata un'esperienza davvero interessante e mi ha insegnato molto. Ora ho un'idea più chiara di come viene condotto lo sviluppo di un prodotto software e sono più consapevole dei problemi che possono emergere in corso d'opera e delle buone pratiche per previnirli ed affrontarli. Non sono infatti mancate le difficoltà, sia di carattere tecnico che organizzativo, ma ho sempre cercato di farvi fronte in modo pratico e utilizzando tutte le risorse che avevo a disposizione.

Le fasi che mi hanno messo più alla prova sono state le fasi di analisi e di design, in merito alle quali non avevo esperienze pregresse nell'ambito di un progetto di dimensioni medio-grandi come questo. Considerando ciò, sono complessivamente soddisfatta del design finale delle parti di mia competenza, anche se è sicuramente possibile fare dei miglioramenti. Questo vale particolarmente per la progettazione che ho svolto per parte di view dell'applicazione: tutte le scelte prese sono state ragionate, ma mi chiedo come si sarebbe potuto avere un design meno articolato e come si potrebbe rendere più semplice l'interazione tra Controller e View. Sono invece più soddisfatta per quanto riguarda il design le parti di Model e Controller che ho realizzato, per le quali ritengo di essere giunta a delle buone soluzioni.

Sono inoltre contenta di aver acquisito maggiore dimestichezza con la programmazione ad oggetti e con gli strumenti introdotti durante il corso, quali Git e Gradle, e di aver deciso di sperimentare con altre tecnologie, come ho fatto per la configurazione della board su file XML e per la scrittura del regolamento in HTML.

L'esperienza del lavoro di gruppo è stata anch'essa molto formativa: la divisione e la sincronizzazione del lavoro tra più persone non sono aspetti banali come potrebbe sembrare e la collaborazione è fondamentale per il raggiungimento di un buon risultato finale, motivo per cui mi sono sempre resa disponibile in caso di problemi. Ritengo che il mio contributo all'interno del gruppo sia stato positivo.

Penso che in futuro mi dedicherò nuovamente a questo progetto con molto piacere: dopo tutto l'impegno messo e il lavoro svolto, sarebbe gratifican-

te migliorare il software e renderlo più completo, aggiungendo funzionalità, migliorando la parte grafica e facendo del refactoring.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Alin Stefan Bordeianu

Il corso è il più impegnativo che abbia affrontato in tutta la triennale. Nonostante apprezzi le sfide e le occasioni di espandere i miei orizzonti, devo ammettere che questo corso non lascia molto spazio per altro, in quanto il progetto richiede un grande sforzo per essere finito in tempo e spesso e volentieri si sacrificano ore di lezione (e di sonno) per la sua realizzazione. Un suggerimento potrebbe essere quello di consentire agli studenti di fare la proposta di progetto e almeno l'analisi già durante il corso stesso. Vorrei ribadire anche il fatto che si potrebbero approfondire ancora di più gli aspetti legati al DVCS, magari dedicando una lezione di laboratorio a un mini progettino su cui lavorare a coppie, giusto per aiutare gli studenti a prendere confidenza con i vari pull, push, commit, merge, ecc. quando ad utilizzare lo strumento si è in più di uno.

Aggiungo anche una considerazione sul laboratorio di C#: purtroppo non è stato dedicata sufficiente attenzione a quella parte del corso (ad esempio i test che dovevamo usare per verificare il nostro codice erano errati e malfunzionanti), e per queste ragioni il primo impatto con tale linguaggio non è stato molto gradevole.

Appendice A

Guida utente

Si rende presente il fatto che nelle varie partite bisogna cliccare il tasto "Pass turn" dopo ogni mossa per permettere al gioco di avanzare.

Appendice B

Esercitazioni di laboratorio

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

- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p169729
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=115548#p171164
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=118995#p175327
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=119938#p176526

B.0.2 andrea.piermattei@studio.unibo.it

B.0.3 margherita.raponi@studio.unibo.it