Wazzle

Relazione del progetto

Programmazione ad oggetti

Alessio Barbanti, Lucia Castellucci, Luca Samorè

25 aprile 2022

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il gruppo si pone come obiettivo quello di realizzare una combinazione di due giochi fortemente ispirati a Ruzzle[1] e Wordle[2], due popolari puzzle game. Lo scopo è quello di realizzare un videogioco, che abbiamo intitolato Wazzle, che abbia come focus principale Ruzzle, mentre Wordle verrà utilizzato come minigioco, che, se vinto, fornisce vantaggi nel gioco principale.

1.1.1 Requisiti funzionali

- Il menù principale dovrà permettere all'utente di iniziare una partita ad uno dei due videogiochi, eventualmente visionando preventivamente un tutorial.
- Ruzzle dovrà mettere a disposizione un tabellone di gioco pieno di lettere, che nasconde al suo interno diverse parole. L'utente potrà combinare tra loro le lettere vicine e cercare di individuare le parole nascoste nella griglia di gioco.
- All'utente che indovina una parola su Ruzzle dovrà essere assegnato un punteggio che dovrà variare in base alla difficoltà delle lettere che la compongono.
- La partita di Ruzzle dovrà terminare una volta scaduto il tempo o una volta trovate tutte le parole.

[1] `https://it.wikipedia.org/wiki/Ruzzle`
[2] `https://it.wikipedia.org/wiki/Wordle`

- Al termine di ogni partita, Ruzzle dovrà dare informazioni all'utente in merito all'esito, ad esempio indicando quante parole avrà indovinato rispetto a quelle presenti in totale.
- Ruzzle dovrà permettere all'utente di utilizzare i bonus di gioco (sempre che ne siano stati cumulati).
- Wazzle dovrà mettere a disposizione il minigioco Wordle per permettere all'utente di accumulare bonus spendibili su Ruzzle.
- Wordle dovrà mettere a disposizione dell'utente 6 tentativi per indovinare una parola di 5 lettere.
- Al termine di ogni tentativo Wordle dovrà fornire all'utente informazioni su quanto l'utente si è avvicinato alla soluzione, colorando le lettere del tentativo appena sottomesso.
- La partita di Wordle dovrà terminare una volta terminati i tentativi possibili o una volta indovinata la parola nascosta.
- Al termine di ogni partita, Wordle dovrà dare informazioni all'utente sul risultato della stessa, ossia dovrà riportare la soluzione, quanti tentativi ha impiegato per trovarla (o meno), e dovrà indicare se e quale bonus è stato ottenuto.

1.1.2 Requisiti non funzionali

- L'applicazione dovrà gestire la memoria persistente tramite salvataggio su file.
- L'interfaccia grafica dell'applicazione dovrà essere responsive e dovrà adattarsi alla risoluzione dello schermo in uso.
- Dovrà essere disposta una grandezza iniziale della finestra che dovrà risultare confortevole ma ridimensionabile a piacere.

1.2 Analisi e modello del dominio

Wazzle dovrà essere composto da due giochi: Ruzzle e Wordle, ai quali faremo riferimento rispettivamente anche come gioco principale (o maingame) e minigioco (o minigame).
Il primo gioco dovrà fare uso di quattro elementi fondamentali. Prima di tutto dovrà mettere a disposizione una griglia di gioco sulla quale potrà giocare l'utente. La selezione delle lettere che compariranno nella griglia dovrà essere fatta casualmente, per scongiurare, il più possibile, l'eventualità che l'utente giochi più di una volta con la stessa griglia di gioco.
Tuttavia dovrà anche essere generata intelligentemente, per evitare che l'utente si cimenti nella ricerca di parole in una griglia priva o eccessivamente scarna di esse. Inoltre, Ruzzle dovrà far uso anche di un timer, che dovrà scandire i tempi di gioco, e di un punteggio, che dovrà dipendere dalle parole che sono state individuate dall'utente.
Infine, all'interno del gioco principale dovrà essere prevista una serie di bonus, che dovranno essere accumulabili in caso di vittoria al minigioco Wordle. Questa dovrà essere l'unica dipendenza tra i due videogiochi, affinché possano mantenere la loro indipendenza nonostante siano combinati in un unico gioco.
Wordle, dunque, dovrà essere gestito come un minigioco ausiliare rispetto a Ruzzle. Gli elementi di gioco di Wordle saranno invece due: la parola che l'utente dovrà indovinare e i tentativi che dovrà avere a disposizione per indovinarla. Ogni tentativo sottomesso dovrà fornire un riscontro in merito alla correttezza (o meno) della parola inserita.
La partita si dovrà concludere con esito negativo, nel caso in cui la parola non dovesse essere indovinata entro il numero di tentativi prefissati, o con esito positivo nel caso contrario. Quest'ultima casistica dovrà conferire all'utente un vantaggio nel gioco principale, misurato in termini di Bonus.
Infine, dovrà essere predisposta una gestione per parole utilizzate in entrambi i giochi, ossia quelle contenute nella griglia del gioco principale, e quelle da indovinare nel minigioco.

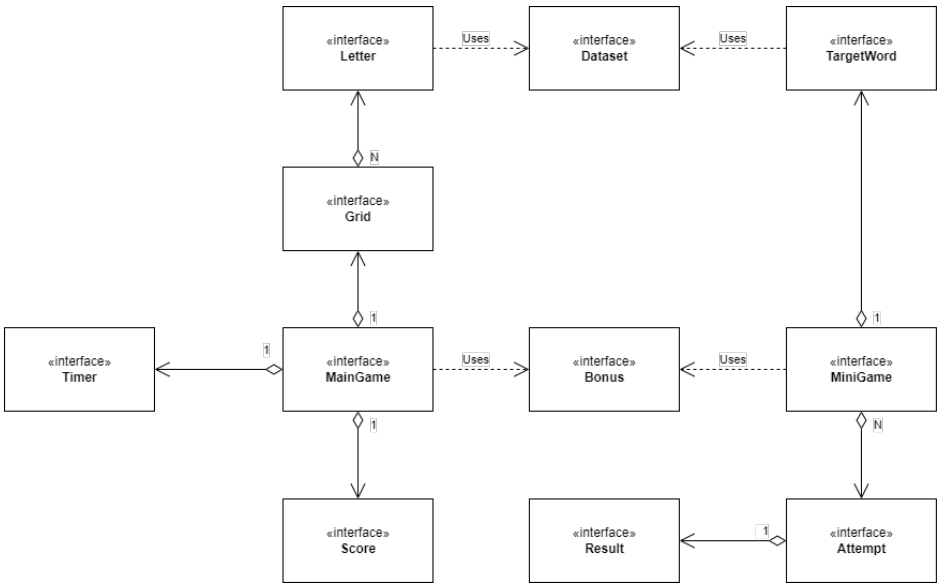

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro.

Capitolo 2

Design

2.1 Architettura

Il progetto segue il pattern architetturale MVC, con l'obiettivo di creare una separazione netta tra la logica dei giochi e la loro presentazione.

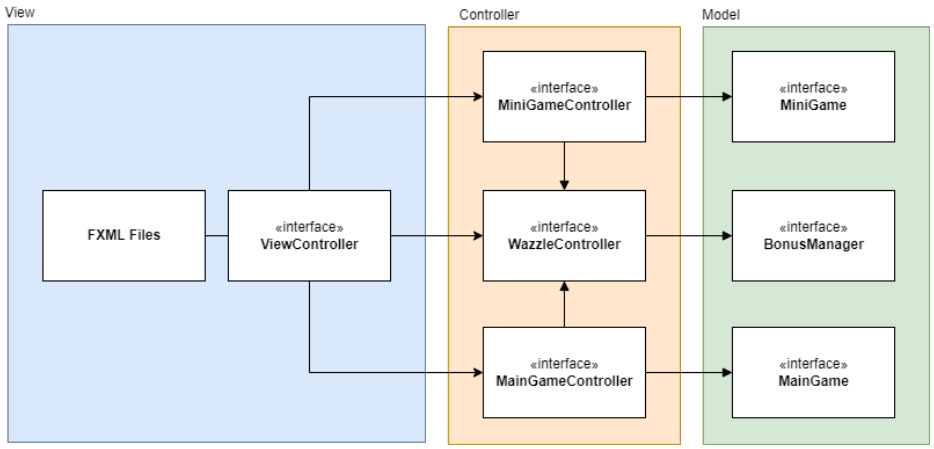

Figura 2.1: Schema UML dell'architettura generale dell'applicazione e del pattern MVC.

Le interfacce MainGame e MiniGame racchiudono tutti gli elementi dei due giochi. La gestione dei bonus è invece concentrata nell'interfaccia BonusManager, infatti, i bonus sono l'unico elemento di congiunzione del Model tra il MainGame ed il Minigame.
Il medesimo ragionamento è stato fatto per i controller dei due giochi che sono indipendenti tra di loro e fanno uso di WazzleController solo per accedere ai bonus e ad altre componenti comuni, come ad esempio la gestione dei file.
Infine, per quanto riguarda la gestione delle view, si è deciso di scomporle

ulteriormente in file *.fxml* associati al proprio controller di view. In questo modo la comunicazione tra controller di applicazione e view diventa una comunicazione unidirezionale da controller di view verso controller di applicazione, ne consegue che i controller di applicazione sono completamente indipendenti dalla view sottostante.
Dunque, nello specifico:

- Il Model incapsula la logica dei giochi, costituendo quindi il dominio dell'applicazione.
- I Controller di applicazione interagiscono direttamente con gli elementi del model, ed espongono ai controller di view le informazioni strettamente necessarie alle view.
- La componente View si suddivide a sua volta in Controller e View. I controller di view interagiscono direttamente con i Controller di applicazione e raccolgono gli elementi che devono essere presentati, mentre le view, ossia i file *.fxml*, si occupano di presentarli in modo efficace.

2.2 Design dettagliato

2.2.1 Alessio Barbanti

Premessa

In fase di design ci si è accorti che la quantità lavoro non era bilanciata come ci si era aspettato al momento della proposta, motivo per cui lo sviluppo del Minigame è stato spostato interamente a me. Procedo quindi alla spiegazione in dettaglio dei punti salienti del design delle parti a me assegnate usando schemi UML contenenti solo i metodi rilevanti per la descrizione del problema.

Calcolo della frequenza di apparizione delle lettere all'interno del dataset

Per lo sviluppo della mia parte di model riguardante il MainGame ho inizialmente progettato la classe Dictionary, che si occupa di conservare il dataset di parole passato dal controller. Diverse operazioni all'interno dell'applicazione faranno riferimento a questa classe che, seppur semplice, è la base di quasi tutta l'applicazione.
Questo Dictionary successivamente viene usato come base per poter definire la frequenza di apparizione di ogni singola lettera all'interno della lista di

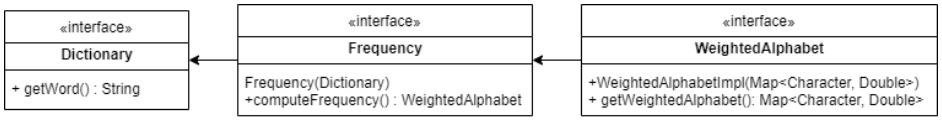

Figura 2.2: Schema UML del calcolo della frequenza di apparizione delle lettere all'interno del dataset.

parole usata come dataset.
Inizialmente era prevista un'ulteriore parte in cui si era pensato di dare un punteggio a tutte le lettere ma successivamente abbiamo scartato l'idea in quanto abbiamo deciso di dare un punteggio solamente alle lettere scelte e non a tutto l'alfabeto presente nel dataset di partenza.

Rimozione delle parole dal dataset non adatte al minigioco corrente

Problema: Le parole contenute dentro Dictionary potrebbero aver bisogno di una raffinazione in base ad alcuni standard, la classe che si occupa di filtrare le parole secondo un criterio deve essere facilmente estendibile nel caso in cui si voglia implementare un nuovo minigioco.

Soluzione Per poter rendere estendibile questa funzionalità ho adottato il pattern **Decorator**. Questo pattern si adatta bene alle mie esigenze, in quanto modifica il comportamento di un oggetto senza cambiarne l'interfaccia. Inoltre, mi dà modo di poter riutilizzare decorator già implementati in precedenza, nel caso sia necessaria una combinazione di più di questi.
Ho quindi iniziato a fare il design della classe FilteredDictionary ovvero l'Abstract Base Decorator che successivamente viene esteso da FiveWordDecorator che si occupa di filtrare tutte le parole di dimensione diversa da cinque. In questo modo, nel caso di una futura implementazione di altri minigiochi, (come ad esempio Bull and Cow o Jotto - alternative a Wordle valutate inizialmente dal gruppo), basterebbe creare un nuovo concrete decorator contenente una diversa funzionalità di filtraggio.

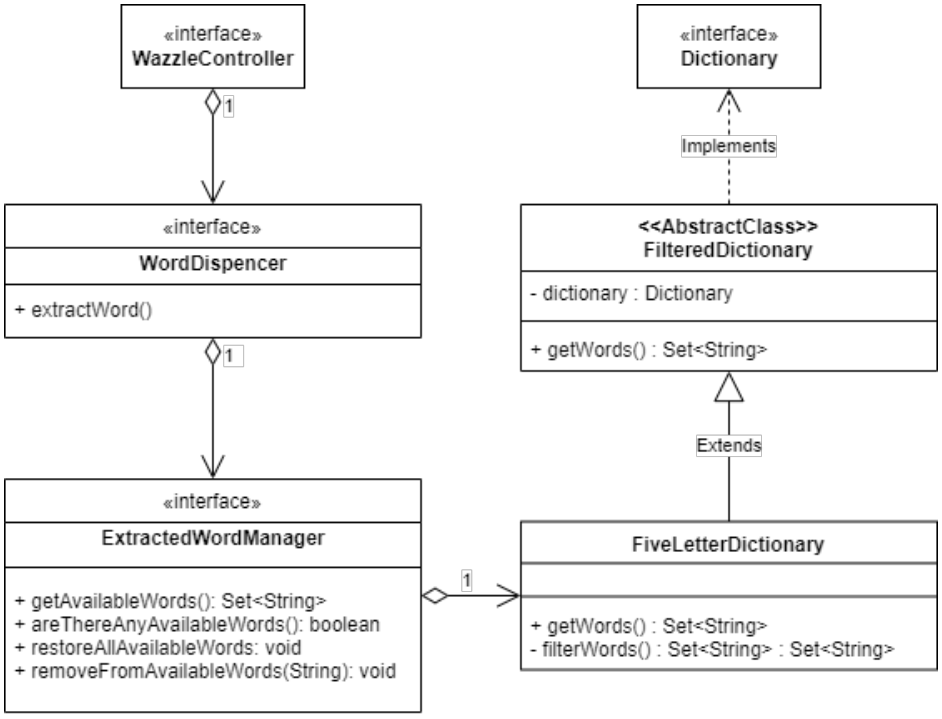

Figura 2.3: Schema UML delle entità che gestiscono la creazione delle lettere.

Salvataggio e ripresa di una partita interrotta

Problema: Dove tenere le informazioni rilevanti riguardanti il Minigame per poterne permettere il salvataggio e il ripristino ad uno stato precedente senza però violarne l'incapsulamento.

Soluzione: La prima soluzione a cui ho pensato è stato il pattern Memento, usato solitamente per questo tipo di operazioni. In tal caso, la classe Minigame avrebbe avuto il ruolo di Originator, ossia il componente che si occupa di salvare il suo stato interno in un oggetto di classe Memento e ripristinarsi al bisogno. Quest'ultimo ruolo sarebbe invece stato ricoperto dalla classe SavedMinigame, mentre il ruolo di Caretaker sarebbe stato ricoperto dal controller di Minigame, che al momento dell'inizializzazione del minigioco decide se avviare una nuova partita o caricarne una se è presente un file di salvataggio.
Ho però deciso di non adottare questo pattern poiché non si adattava esattamente ai miei bisogni, ma di ideare un design simile che differiva dal classico pattern Memento per alcuni aspetti. Il Caretaker non contiene al suo interno lo stack dei precedenti Memento ma all'occorrenza salva e carica da file il

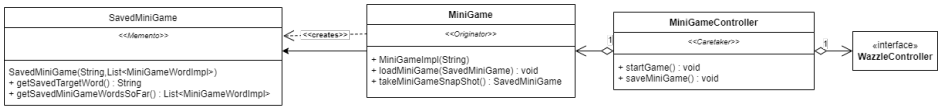

Figura 2.4: Schema UML delle entità che gestiscono il salvataggio e ripresa di una partita interrotta.

Memento passandolo e prendendolo dalla classe WazzleController sviluppato dalla mia collega Castellucci. Inoltre, il Memento, per una questione di salvataggio su file, non è una Nested Class di Minigame ma una classe esterna. Ho trovato però che questo design fosse comunque un buon compromesso per mantenere l'incapsulamento delle informazioni della classe Minigame, in quanto nessun'altra classe può modificare le informazioni al suo interno e la ripresa di uno stato precedente è possibile solo da Minigame stesso.

Gestione dell'esito delle parole

Problema: Dove tenere le informazioni di Minigame e la gestione dei tentativi senza violare il *Single Responsibility Principle*.

Soluzione: Il design iniziale prevedeva la computazione delle informazioni all'interno della classe Minigame. Ad una seconda osservazione ho notato che Minigame si sarebbe dovuto occupare di compiti relativamente scorrelati tra di loro, e quindi ho deciso di scorporare la logica di conservazione delle informazioni e la logica di computazione. La logica riguardante lo stato del Minigame è rimasta dentro l'omonima classe, mentre la logica di computazione è stata spostata nell'entità chiamata WordChecker. Quest'ultima si occupa di controllare se la parola immessa dall'utente è corretta e, nel caso in cui non lo fosse, di calcolarne le differenze.

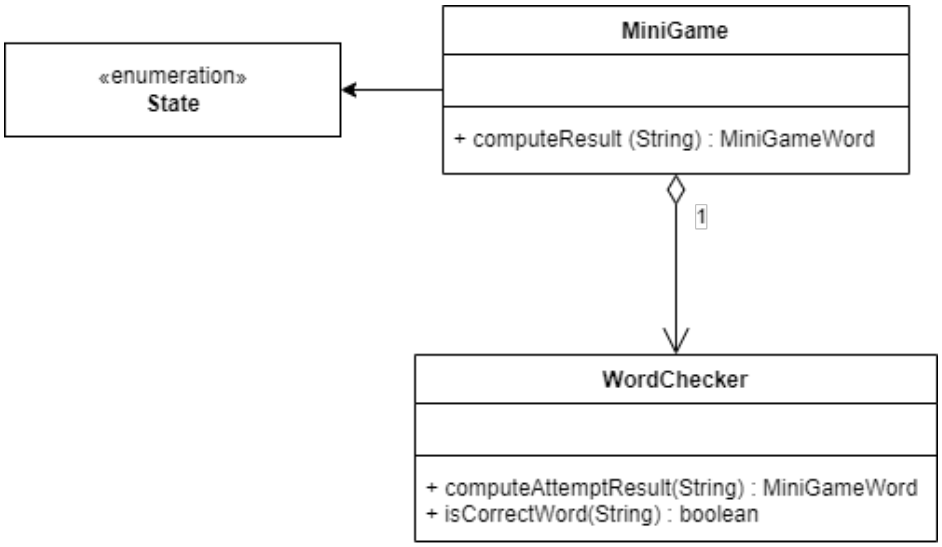

Figura 2.5: Schema UML delle entità che gestiscono il salvataggio e ripresa di una partita interrotta.

Gestione dei controller di applicazione

Per quanto riguarda il controller di Minigame da me progettato, oltre ciò che è già stato illustrato precedentemente, il suo scopo è quello di gestire il flusso di gioco ricevendo richieste dalla View, passarle al model per la loro computazione e interrogarlo per sapere lo stato corrente del gioco.
Nel caso in cui la partita si sia conclusa con una vittoria il MinigameController lo notifica al WazzleController, il quale aggiorna conseguentemente il numero dei bonus.
Il tipo di bonus e altre informazioni riguardanti il model sono, infine, raccolte e mostrate nella View delle statistiche.

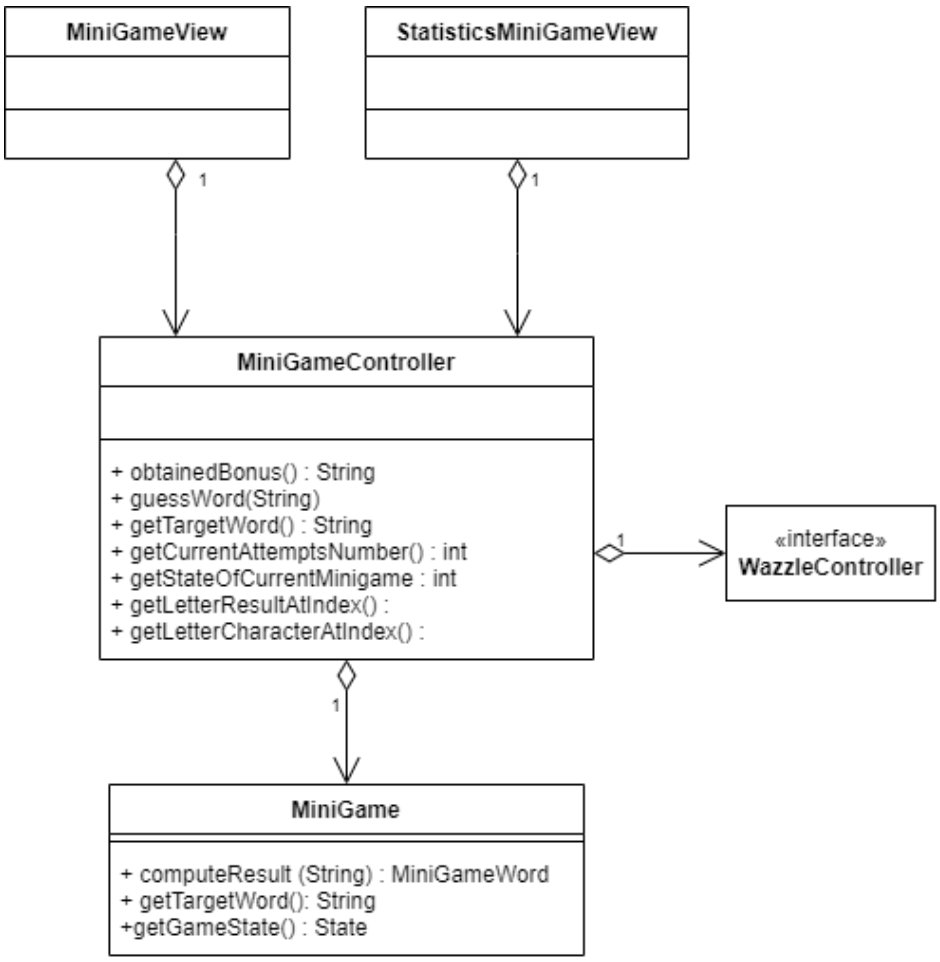

Figura 2.6: Schema UML della gestione dei controller di applicazione.

2.2.2 Lucia Castellucci

Premessa

Nello sviluppo di Wazzle mi sono occupata della porzione di model che crea, sotto ogni loro aspetto, le lettere che andranno a comporre la griglia di gioco. Sempre in merito al model mi sono occupata dell'intera gestione dei bonus. Infine, mi sono occupata dello sviluppo del controller di applicazione del gioco principale, dello storico delle partite del gioco principale e dei relativi aspetti grafici.

Creazione delle lettere

Il primo problema che ho affrontato è quello della creazione di una lettera di gioco, partendo da un WeightedAlphabet. Nello specifico, una lettera, all'interno del gioco principale, è contraddistinta da un carattere, da una posizione e da un punteggio. L'algoritmo di generazione delle lettere è stato pensato per essere indipendente dal dateset che viene fornito. Questa scelta è stata fatta nell'ottica di eventuali future implementazioni o modifiche, quali ad esempio un eventuale cambiamento del dataset, oppure l'aggiunta di una funzionalità che permetta di scegliere la lingua.

Problema: Individuare quali sono i caratteri più idonei ad essere attribuiti alle lettere presenti nel gioco. Per idoneità dei caratteri, si intende la loro attitudine a non dare adito ad un numero elevato di ri-generazioni delle lettere. In sintesi, il problema consiste nel fornire, al termine di tutte le computazioni, un insieme di lettere che possa essere facilmente adattato a diventare una griglia di gioco.

Soluzione: Creazione di due oggetti che abbiano come unico scopo (*Single Responsability Principle*) quello di classificare le lettere in range (AlphabetClassifier), e quello di scegliere i caratteri da inserire nelle lettere (LetterChooser).

Il primo oggetto, dunque, partendo da un WeightedAlphabet, esegue la classificazione di ogni carattere in Range. Tale classificazione permette ad un secondo oggetto LetterChooser di effettuare una scelta dei caratteri casuale, estraendo, per ogni Range, un numero di caratteri appartenenti ad esso in numero proporzionale rispetto al peso del Range di appartenenza.

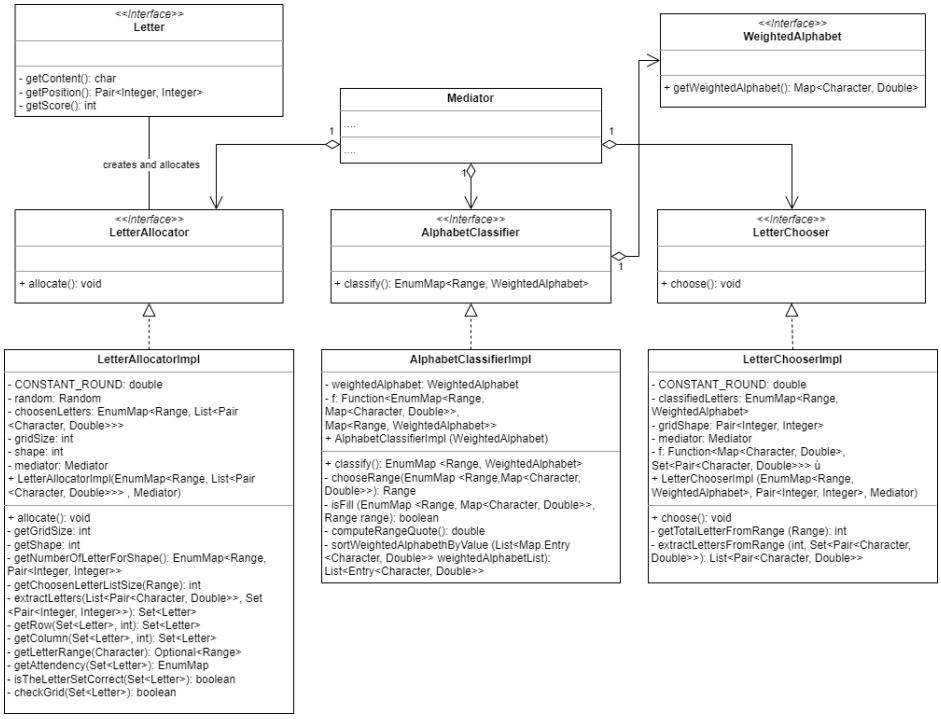

Figura 2.7: Schema UML delle entità che gestiscono la creazione delle lettere.

Il compito di generazione delle lettere presenti nella griglia viene poi proseguito dallo ScoreConverter, che determina il punteggio di ogni carattere (di cui parlerò di seguito in dettaglio), per essere concluso dal LetterAllocator, che determina, anch'esso in modo casuale e rispettando un determinato criterio di allocamento, la posizione di ogni carattere all'interno della griglia. LetterAllocator si occuperà dunque anche di generare le Letter, considerando che dispone di tutti i caratteri, dei relativi punteggi, e della posizione da lui determinata.

Inizialmente il design di questa porzione di dominio non prevedeva alcun design pattern, seppur la progettazione iniziale prevedeva una catena di dipendenze tra le mie classi. Questa problematica è stata poi discussa con il collega Samorè, che ha poi sviluppato un design che centralizzasse la creazione delle Letter e ne integrasse la porzione di dominio il cui sviluppo era a suo carico, facendo uso della classe Mediator.

Attribuzione del punteggio alle lettere

L'attribuzione del punteggio ad ogni carattere si pone come passaggio intermedio tra la scelta dei caratteri (eseguita dal LetterChooser) e l'allocazione di essi (eseguita dal LetterAllocator).

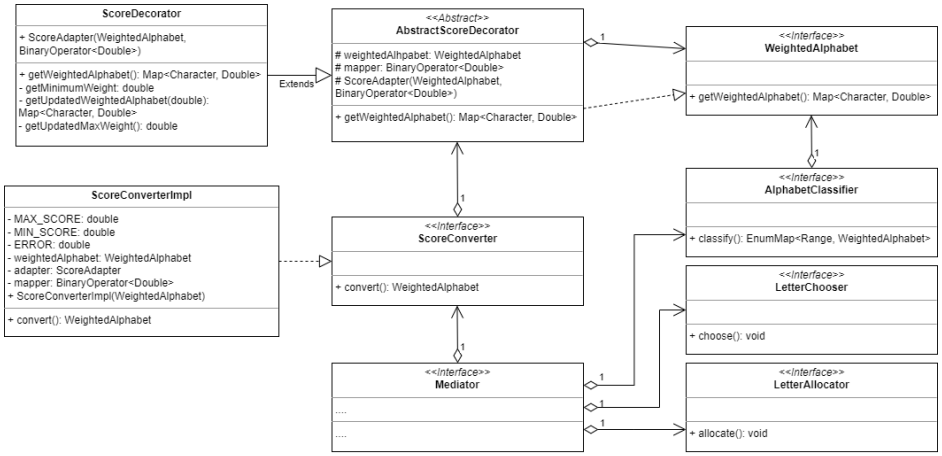

Figura 2.8: Schema UML delle entità che gestiscono la creazione delle lettere.

Problema: Il punteggio dipende strettamente dalla frequenza di un carattere all'interno del dataset di partenza (rappresentato dall'oggetto WeightedAlphabet). Questa analogia tra il concetto di punteggio delle lettere e quello di WeightedAlphabet, pongono un problema di duplicazione del codice, siccome il concetto di frequenza e quindi anche quello di WeightedAlphabet, è già esistente all'interno del dominio.

Soluzione: Onde evitare inutili duplicazioni di codice, e con l'intento di non violare il *Single Responsibility Principle*, ho progettato il punteggio facendo uso del design pattern **Decorator**. Nello specifico, il WeightedAlphabet funge da Component e viene implementato da AsbtractScoreDecorator, che funge da Decorator. Infine, il ruolo di Concrete Decorator è svolto dallo ScoreDecorator. Dunque, la computazione del punteggio viene ultimata e fornita al Mediator dallo ScoreConverter.

Creazione e gestione dei bonus

Mi sono occupata della parte di gestione dei bonus. I bonus sono stati pensati per essere l'unico collegamento tra il gioco principale e il minigioco. Questo permette, nell'ottica di future implementazioni o modifiche, di poter sostituire il minigioco che li dispensa senza dover modificare altre porzioni di Model.

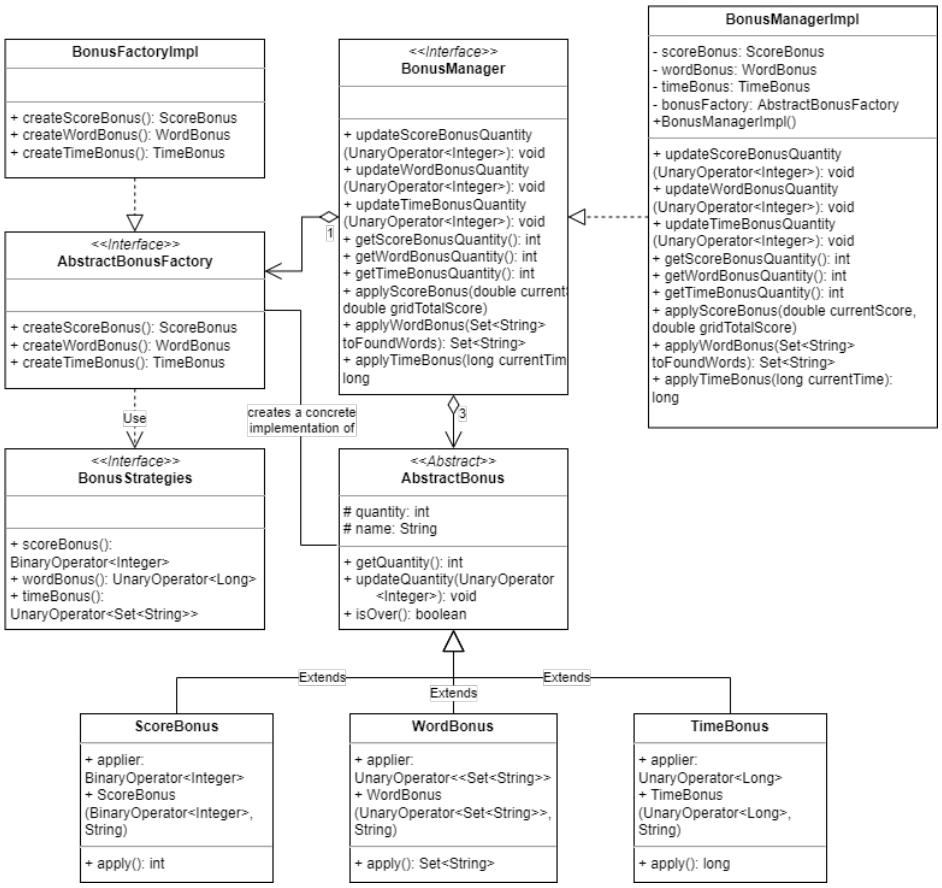

Figura 2.9: Schema UML delle entità che gestiscono la creazione e gestione dei bonus.

Problema: I bonus hanno diverse proprietà in comune, ossia possiedono tutti quantità e nome. Hanno invece strategie di applicazione diverse, nella forma e nel contenuto.

Soluzione: Ho creato una classe che astraesse le proprietà in comune dei bonus, e ho fatto in modo che tutti i bonus la estendessero.

Problema: I bonus appartengono alla stessa famiglia di oggetti, dunque la loro creazione comporterebbe la necessità di duplicare del codice.

Soluzione: Ho centralizzato la creazione dei Bonus servendomi del design pattern **Abstract Factory**.

Per ottimizzare ulteriormente la leggibilità del codice, ho creato un'interfaccia che racchiudesse tutte le strategie di applicazione dei bonus, chiamandola BonusStrategies. L'intera gestione dei bonus, compresa la creazione e l'aggiornamento di essi, ho deciso di affidarla al BonusManager, di modo che sia l'unico entry point per i Controller alle componenti Model dei bonus.

Gestione del timer di gioco

Per gestire il timer di gioco all'interno del main game sono state create due interfacce: una a livello di Controller, ossia GameTimer, che gestisce il TimerTask, e una a livello di Controller di View, che gestisce l'AnimationTimer, ossia GameTimerView.

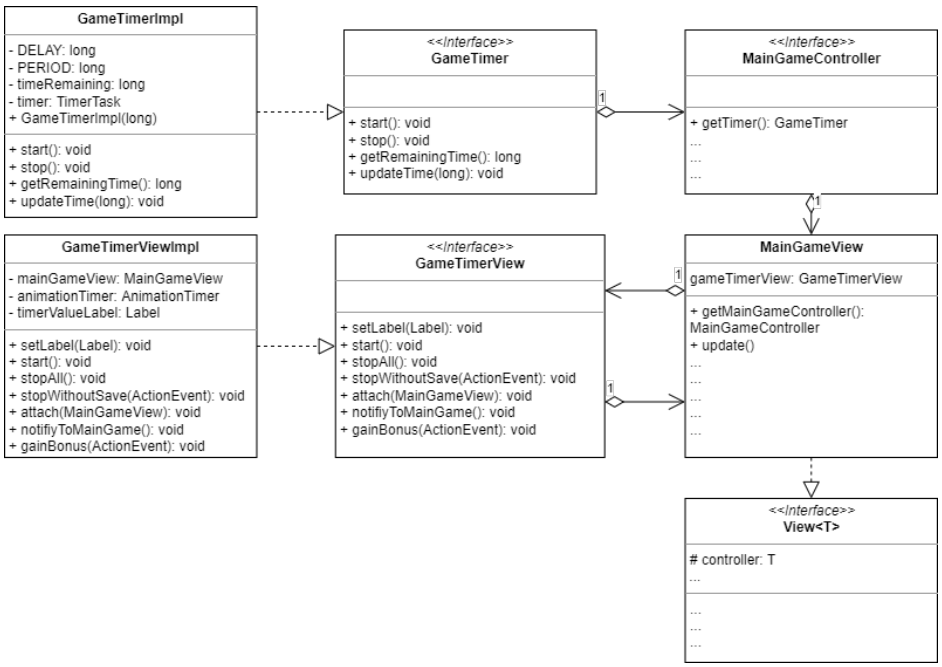

Figura 2.10: Schema UML delle entità che gestiscono il timer.

Problema: Per poter esternare la gestione del timer di gioco, rispetto al Controller della View del gioco principale, è necessario che vengano scambiate informazioni tra GameTimerView e MainGameView.

Soluzione: Ho applicato il design pattern **Observer**, affidando al MainGameView il ruolo di Observer, e a GameTimeView il ruolo di observable. In questo modo, non appena il timer termina, il controller viene informato, e la View viene aggiornata.

Controller principale di wazzle e dello storico partite

Per quanto riguarda le componenti Controller, mi sono occupata della gestione del Controller principale di Wazzle e di quello dello storico partite.

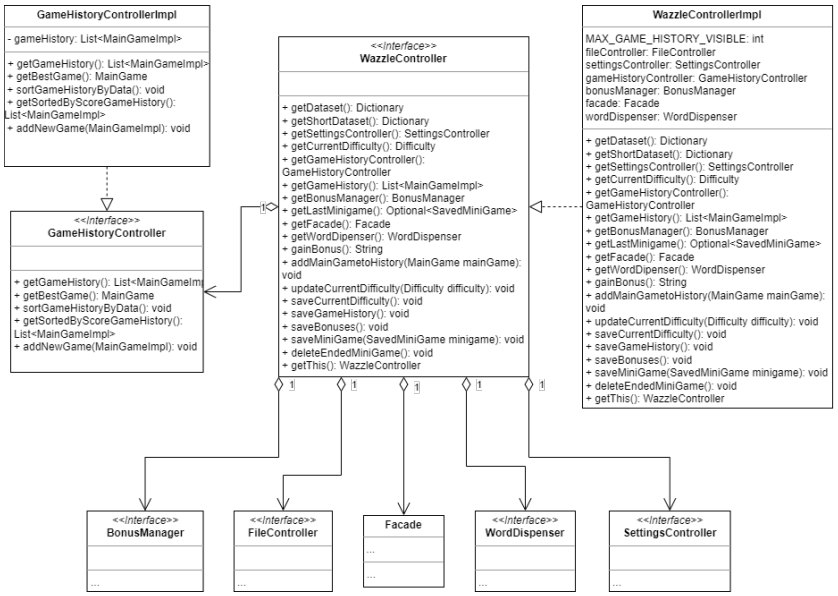

Figura 2.11: Schema UML del Controller principale di Wazzle e del Controller dello storico partite.

Problema: Il sistema presenta un numero elevato di Controller e di componenti del Model necessari alla gestione delle partite. Alcuni oggetti vengono utilizzati in più di una sotto-sezione del sistema, duplicando il codice, con la conseguente violazione del principio *Don't Repeat Yourself*.

Soluzione: Ho implementato un WazzleController che segue il design pattern **Facade**, in modo tale da centralizzare e snellire le comunicazioni tra Controller, garantendo una gestione delle operazioni sulle partite molto più semplice e lineare.

Il WazzleController contiene i seguenti elementi:

- AbstractGameFactory
- BonusManager
- FileController
- GameHistoryController
- SettingsController
- WordDispenser

Il WazzleController viene dunque utilizzato principalmente per:

- Iniziare le partite (sia del maingame che del minigame)
- Salvare le partite del maingame
- Elargire e salvare i bonus
- Aggiornare e salvare le impostazioni
- Aggiornare e salvare lo storico partite
- Tracciare le parole già fornite nel minigame come parola da indovinare, per evitare di ri-sottoporla nella medesima sessione di gioco.
- Riprendere una partita del minigame interrotta prima di essere stata conclusa

Mi sono occupata anche del Controller della GameHistory, che tiene traccia delle ultime partite effettuate nel MainGame, ed è anch'esso contenuto nel WazzleController.

Gestione grafica del menù principale e dello storico partite

Infine, ho realizzato due Controller di View e il relativo file *.fxml* estendendo un'interfaccia generica che ha realizzato il nostro collega Samorè.

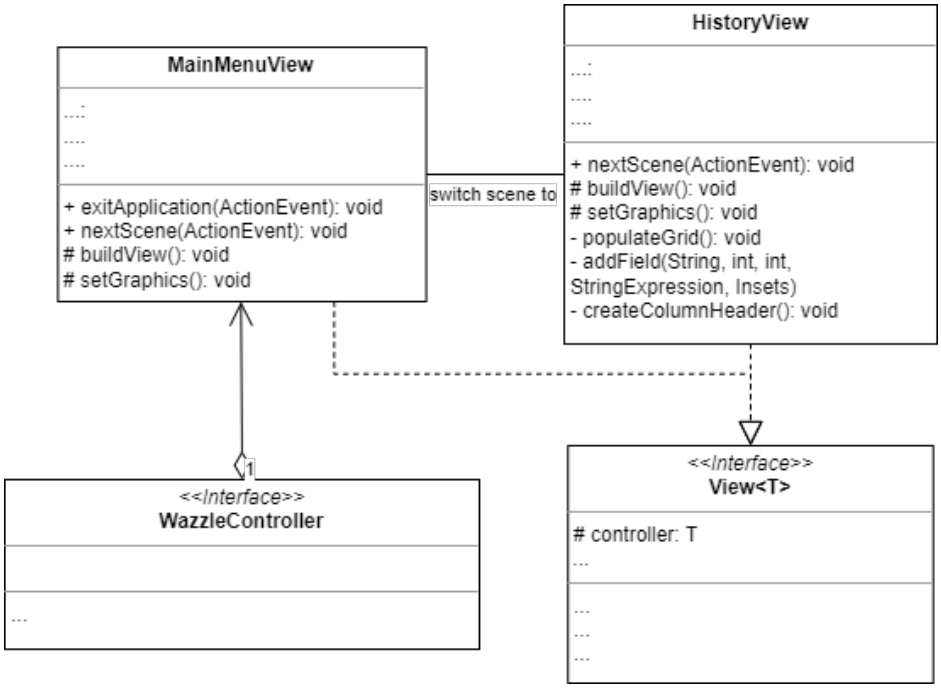

Figura 2.12: Schema UML delle entità che gestiscono le View del menù principale e dello storico partite.

2.2.3 Luca Samorè

Premessa

Nel corso dello sviluppo di Wazzle mi sono occupato della sottoparte di model che si concentra sulla generazione di una griglia che rispetti alcuni parametri. Mi sono inoltre dedicato alla creazione delle partite di Ruzzle, della loro gestione in termini di controller di applicazione e di view. Un altro aspetto fondamentale del sistema che ho sviluppato è la gestione del salvataggio dei dati, realizzando un controller in grado di gestire le operazioni di lettura e scrittura su file di testo e *.json*. Infine sono riuscito a creare una semplice astrazione per i controller di view.

Generazione delle griglie di gioco di Ruzzle

Una griglia di Ruzzle è caratterizzata da un processo di generazione piuttosto complesso, il cui punto centrale è la creazione degli oggetti Letter svolta da LetterAllocator. Una volta generate un certo numero di Letter, infatti, si ha la possibilità di formare una griglia di gioco, rappresentata come oggetto Grid. La creazione vera e propria di tale oggetto è affidata a GridGenerator, il quale deve valutare la qualità delle Letter selezionate in base ai parametri forniti sotto forma di oggetto Difficulty.

Problema: bisogna separare il processo di ottenimento degli oggetti Letter dalla creazione vera e propria di un oggetto Grid.

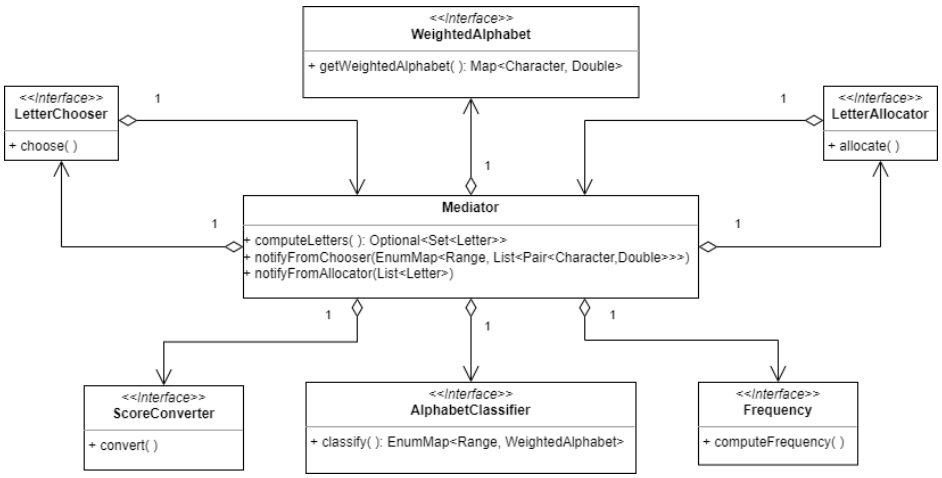

Figura 2.13: Schema UML della classe Mediator.

Soluzione: ho creato una classe Mediator il cui compito è incapsulare e centralizzare il processo di ottenimento delle Letter. La classe segue l'omonimo pattern ed è stata pensata per essere un'interfaccia (Mediator si comporta come un Facade) per GridGenerator. In questo modo, a GridGenerator occorre chiamare un metodo su un'istanza di Mediator da lui creata per ottenere le Letter da valutare.

A questo punto, GridGenerator deve effettuare una serie di controlli affinché le Letter generate siano utilizzabili per formare un oggetto Grid. Un insieme di Letter viene definito valido per formare una griglia se il numero di parole che si possono formare con quelle Letter si trova all'interno di un range, definito dall'oggetto Difficulty (lowerBound e upperBound), il quale viene passato a GridGenerator al momento della sua istanziazione. Se il numero di parole formabili non cade all'interno del range, le Letter devono essere generate nuovamente.

Problema: GridGenerator viola il Single Responsibility Principle, poiché si occupa sia della valutazione delle Letter che della creazione di una Grid.

Soluzione: ho separato la validazione delle Letter in un'interfaccia a parte chiamata GridValidator. Il suo compito è quello di prendere le Letter generate e applicare una serie di filtri su di esse (forniti dall'interfaccia funzionale Filters). Se le Letter passano tutti i filtri allora possono essere utilizzate per creare una Grid, altrimenti devono essere ricalcolate.

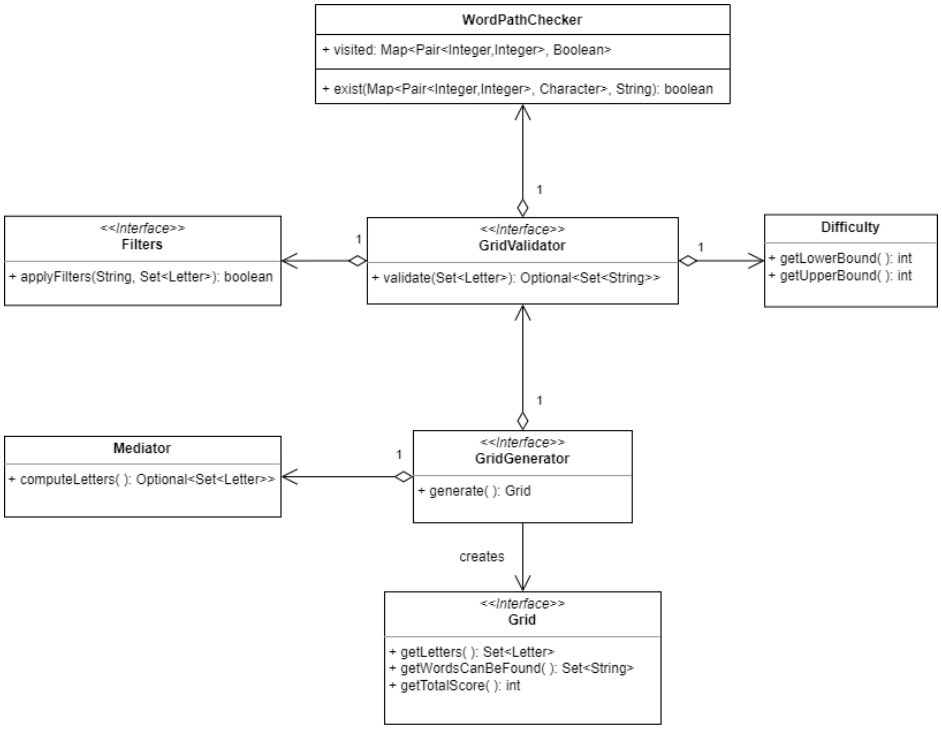

Figura 2.14: Schema UML delle entità che gestiscono la generazione delle griglie di gioco di Ruzzle.

Gestione dei file

Un aspetto cruciale del sistema è sicuramente la gestione delle operazione di I/O sul file system.

Problema: i dati di Wazzle devono poter essere salvati su file e deve essere possibile leggerli all'occorrenza. Inoltre, devono essere gestiti anche i file di sola lettura, come il dataset utilizzato per Ruzzle e Wordle.

Soluzione: questo carico di responsabilità è stato concentrato all'interno di un'interfaccia chiamata FileController, la quale mette a disposizione dei metodi per leggere e scrivere oggetti diversi. In particolare, si è deciso di adottare due tipologie di file: i file di testo, utilizzati per mantenere su disco il dataset delle parole per Ruzzle e Wordle, e i file *.json*, utilizzati per salvare lo stato degli oggetti relativi alle partite, ai bonus e ai settaggi di gioco.

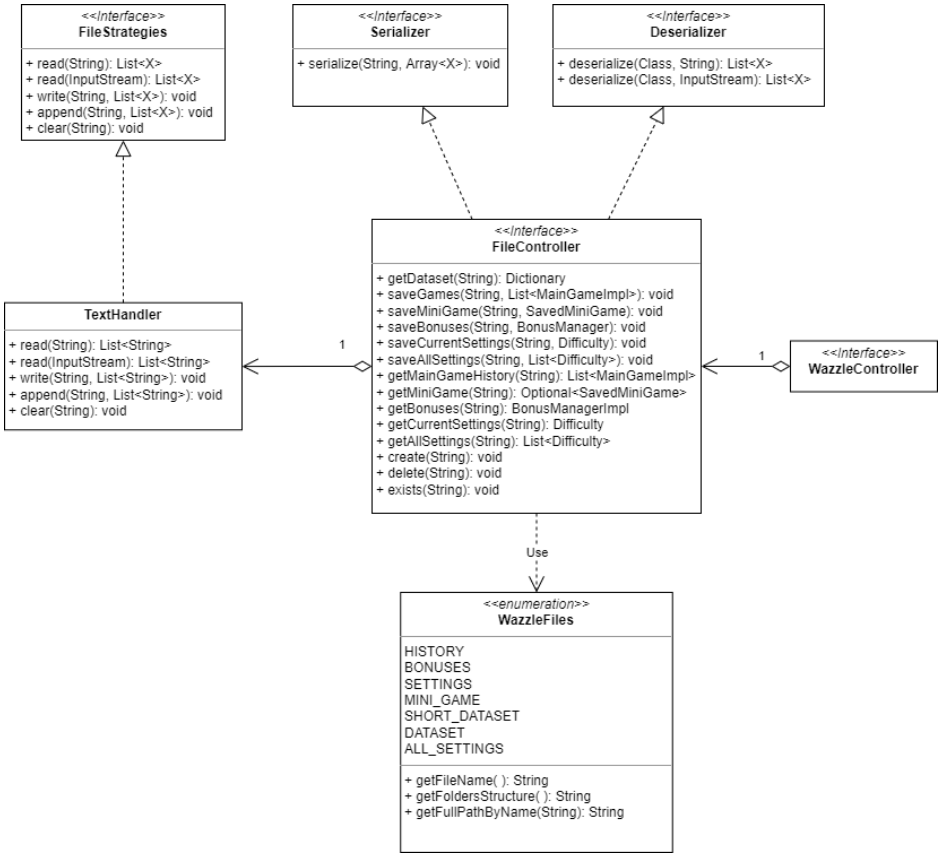

Figura 2.15: Schema UML delle entità che gestiscono i file.

Per rispondere a queste necessità ho deciso di creare le seguenti interfacce:

- *Serializer*: si occupa di serializzare un qualunque oggetto che gli viene passato su un file *.json*.
- *Deserializer*: restituisce una sequenza di oggetti letti da file *.json*
- *FileStrategies*: mette a disposizione dei metodi generici per leggere e scrivere su file di testo.

FileController è, inoltre, dotato di metodi per la creazione, eliminazione di file e controllo della loro esistenza. Ho infine creato una classe TextHandler che implementa FileStrategies e lavora su oggetti String. Questa classe ha il compito di gestire esclusivamente i file di testo. Ho infine separato i file di sola lettura da quelli accessibili sia in lettura che in scrittura. Siccome non è possibile scrivere file all'interno di un jar, questi ultimi vengono salvati sul file system della macchina nella home directory dell'utente al seguente

percorso: *wazzle/files*. Se non esiste, tale struttura di directory viene creata a runtime, all'avvio dell'applicazione.

Astrazione dei controller di view

Il livello di presentazione del sistema è stato scomposto in file .fxml, ognuno dei quali è associato a un proprio controller di view. Tali controller gestiscono il binding con i componenti di JavaFX e permettono di far comunicare i controller di applicazione con le view effettive. Questo ultimo aspetto è possibile poiché i controller di view contengono al loro interno un'istanza del controller di livello applicazione sul quale lavorano. Ciascun controller di view, inoltre, possiede dei metodi per impostare gli elementi grafici, per effettuare un cambio di view e per inizializzare la view stessa.

Problema: tutti i controller di view presentano degli elementi comuni.

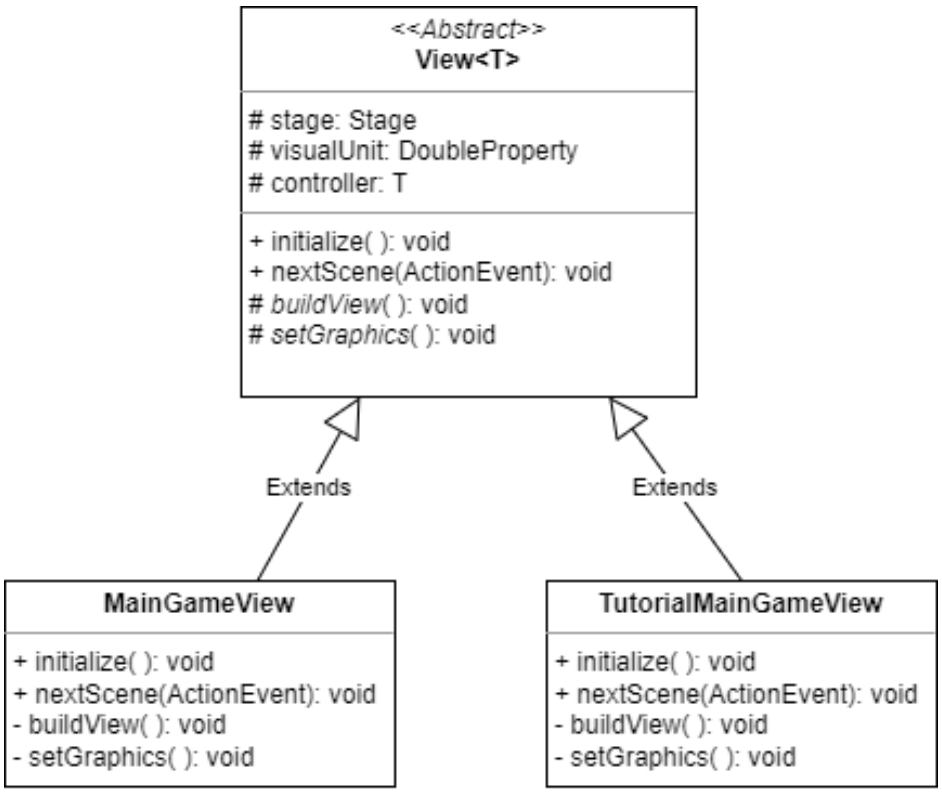

Figura 2.16: Schema UML delle View e dell'entità che le astrae.

Soluzione: ho estrapolato tutti gli elementi comuni in una classe astratta generica View, dove il type parameter rappresenta il tipo del controller di livello applicazione sul quale la view lavora.

Gestione delle partite di Ruzzle

La parte di sistema più sostanziosa che ho realizzato è sicuramente quella riguardante il ciclo di vita delle partite di Ruzzle, il quale comincia con la creazione della partita e finisce con il suo salvataggio appena questa si conclude.

Una partita di Ruzzle è rappresentata da un oggetto MainGame, il cui compito è quello di mantenere lo stato e di fornire all'esterno le informazioni su di esso.

Problema: in ottica di estendibilità, Wazzle deve poter supportare diversi puzzle game ed è quindi necessario un modo per gestire la creazione di questi oggetti.

Soluzione: ho deciso di concentrare la creazione degli oggetti relativi ai giochi in un'unica interfaccia chiamata AbstractGameFactory, implementando dunque il pattern creazionale **AbstractFactory**. Tale interfaccia contiene tanti factory method quanti sono i giochi supportati da Wazzle.

Il flusso della partita viene gestito da MainGameController, il quale si occupa di ricevere le richieste che gli arrivano dal controller di view e di interrogare l'oggetto MainGame per leggere informazioni sulla partita in corso, oppure per eseguire delle operazioni su di esso (come il consumo di un bonus o il controllo di un tentativo) e modificando di conseguenza la view. Una partita inizia quando viene chiamato il metodo start() sull'oggetto GameTimer situato dentro MainGameController e può terminare in due casi:

- l'utente ha trovato tutte le parole del dataset che si potevano formare nella griglia di gioco
- è scaduto il tempo di gioco.

In entrambe le situazioni MainGameController chiede a WazzleController di aggiungere la partita appena conclusa allo storico delle partite giocate e di salvare lo storico aggiornato su file *.json* (sfruttando il FileController sopra descritto).

Lo storico delle partite viene visualizzato in HistoryView. È importante ricordare che l'utente può modificare alcuni parametri di gioco come la difficoltà e la dimensione della griglia da SettingsView.

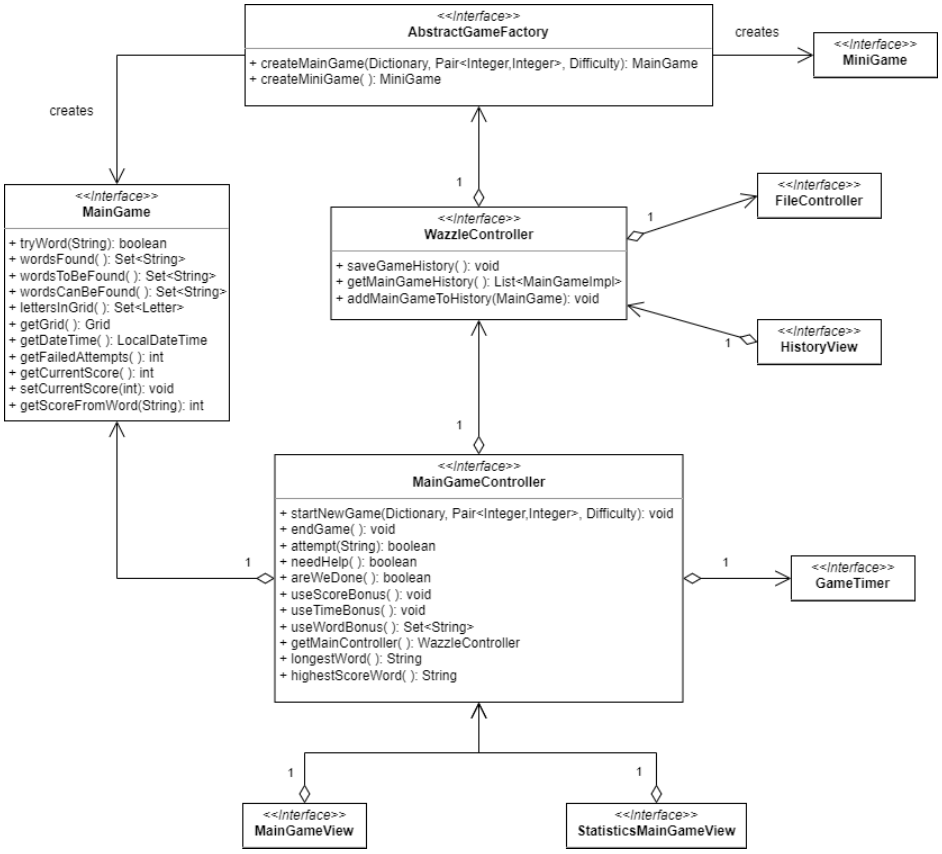

Figura 2.17: Schema UML delle entità che gestiscono il MainGame.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per verificare il corretto funzionamento delle classi di Model che implementano gli algoritmi più complessi si è pensato di implementare dei test automatizzati facendo uso del framework JUnit 4. Per le classi contenenti le porzioni di codice più critiche l'implementazione è stata eseguita seguendo un approccio Test Driven.
Nello specifico i test realizzati dai singoli componenti sono:

Alessio Barbanti

- TestExtractedWordManager
- TestFilteredDictionary
- TestFrequency
- TestMiniGame
- TestWordChecker

Lucia Castellucci

- TestAlphabetClassifier
- TestBonus
- TestLetterAllocator
- TestLetterChooser

- TestScoreConverter

Luca Samorè

- TestFileHandling e l'interfaccia TestFileUtils
- TestGridGenerator
- TestGridValidator

3.2 Metodologia di lavoro

La metodologia di sviluppo adottata dal team è stata quella di seguire un approccio Agile: Il ciclo di sviluppo del progetto è stato diviso in iterazioni nelle quali si è svolta prima una parte di design, poi una parte di test e infine una di sviluppo, spostandosi da l'una all'altra in caso di necessità. Il lavoro svolto singolarmente dai componenti è il seguente:

3.2.1 Alessio Barbanti

Gestione del flusso di gioco del Minigame:

- Package wazzle.controller.minigame.*

Gestione dei Decorator per future implementazioni:

- Package wazzle.controller.minigame.dictionary.*

Frequenza di apparizione delle lettere nel dataset:

- Package wazzle.model.common:
  - Dictionary
- Package wazzle.model.maingame.alphabet.*

Gestione dello stato del Minigioco e logica di salvataggio:

- Package wazzle.model.minigame.*

Gestione dei tentativi immessi dall'utente:

- Package wazzle.model.minigame.attempt.*

Estrazione delle parole adatte al minigioco corrente:

- Package wazzle.model.minigame.word.*

Controller di View:

- Package wazzle.view.controller:
  - MiniGameView
  - QWERTYKeyboard
  - SettingsVew
  - StatisticsMiniGameView
  - TutorialMiniGameView

Varie:

- Package wazzle.controller.maingame:
  - SettingsController

3.2.2 Lucia Castellucci

Menù principale e l'avvio dell'applicazione:

- Package wazzle:
  - App
  - Launcher
- Package wazzle.view.controller:
  - MainMenuView

Creazione delle lettere e attribuzione del punteggio:

- Package wazzle.model.maingame.letter.*
- Package wazzle.model.maingame.letter.score.*

Creazione e la gestione dei bonus:

- Package wazzle.model.common.bonnus.*
- Package wazzle.model.common:
  - BonusManager e BonusManagerImpl

Gestione del timer:

- Package wazzle.controller.maingame.timer.*
- Package wazzle.view.controller:
  - GameTimerView e GameTimerViewImpl

Controller di applicazione Wazzle e dello storico partite:

- Package wazzle.controller.common:
  - WazzleController e WazzleControllerImpl
- Package wazzle.controller.maingame:
  - GameHistoryController e GameHistoryControllerImpl

View del main menu e dello storico partite:

- Package wazzle.view.controller:
  - MainMenuView
  - HistoryView

3.2.3 Luca Samorè

Creazione e gestione di oggetti MainGame:

- Package wazzle.model.common:
  - AbstractGameFactory
  - GameFactoryImpl
  - Facade
- Package wazzle.model.maingame:
  - MainGame e MainGameImpl
  - Mediator

Generazione di oggetti Grid:

- Package wazzle.model.maingame.grid.*

Gestione dei file:

- Package wazzle.controller.common.files.*
- FileController e FileControllerImpl
- WazzleFiles

Controller di applicazione di MainGame:

- Package wazzle.controller.maingame:
  - MainGameController e MainGameControllerImpl

Interfacce e classi di utility per le View:

- Package wazzle.view.*

Astrazione per le View:

- Package wazzle.view.controller.View

View:

- Package wazzle.view.controller:
  - MainGameView
  - TutorialMainGameView (modificato)

3.3 Note di sviluppo

3.3.1 Alessio Barbanti

- **Optional**: Usato in particolar modo per la gestione della presenza o meno del file di salvataggio del minigame.
- **Stream**: calcolo della frequenza di lettere nel dataset per la parte di model del maingame, computazione dei risultati nel minigame.
- **Lambda**: principalmente all'interno degli stream per una maggiore puliza e comprensione del codice ed event handling nella parte di view di minigame.
- **JavaFx**: framework grafico per la realizzazione delle view di gioco.

3.3.2 Lucia Castellucci

- **Optional**: utilizzati in ogni occasione nella quale il valore di un campo o il risultato di una computazione avrebbe potuto essere nullo.
- **Stream**: utilizzati in continuazione per manipolare delle strutture dati.
- **Lambda**: utilizzate per migliorare la leggibilità del codice e velocizzarne la scrittura grazie alle functional interfaces utilizzate individualmente o come parametri negli stream.
- **JavaFx**: framework utilizzato per lo sviluppo delle view del menu iniziale e dello storico delle partite. Oltre alle funzioni di base della libreria sono stati utilizzati meccanismi più avanzati quali i biding per garantire un resize fluido della finestra.
- **Multithreading**: utilizzato per la gestione del timer di gioco del gioco principale.

3.3.3 Luca Samorè

- **Stream**: per effettuare operazioni su strutture dati.
- **Optional**: per eliminare l'utilizzo di valori null all'interno del codice.
- **Lambda**: utilizzate per migliorare la pulizia e la leggibilità del codice nelle situazioni in cui ci si aspetta di utilizzare un'interfaccia funzionale.
- **Generici**: utilizzati per la costruzione di classi e metodi riusabili, in particolare per quanto riguarda la gestione dei file, l'astrazione delle view e alcuni metodi di utility.
- **JavaFx**: framework grafico per la realizzazione delle view di gioco.
- **Gson**: libreria utilizzata per la serializzazione e deserializzazione di oggetti su file json.

Tutti i crediti per la realizzazione di
wazzle.model.maingame.grid.WordPathChecker vanno a jump1221. La soluzione è stata condivisa dall'autore al seguente link: `https://leetcode.com/problems/word-search/discuss/27811/My-Java-solution`

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Alessio Barbanti

Nonostante avessi già sviluppato un progetto di dimensioni avvicinabili a questo ho trovato la difficoltà di quest'ultimo di un ordine di grandezza superiore, probabilmente il problema è anche derivato dalla scelta condivisa dal gruppo di effettuare prima il design del gioco principale e successivamente quello del minigame, oltre che ad una iniziale errata stima del carico di lavoro personale riguardante la prima parte dello sviluppo del model che ha necessitato di tempi diversi da quelli preventivati.
Per diverso tempo (specialmente nella parte iniziale) mi sono trovato nella situazione di non poter lavorare a causa della mancanza di un design di gruppo condiviso da tutti i membri, e ciò mi ha portato ad un lavoro concentrato nelle ultime settimane e a parer mio sotto alcuni aspetti insoddisfacente.
Per quanto riguarda le difficoltà riscontrate in questo progetto, è stata in primis l'applicazione del patter MVC, in quanto ho avuto parecchie difficoltà a separare le tre parti e definire di preciso chi e cosa dovesse fare, in secundis l'abbandono di un membro del gruppo a progetto iniziato ha portato ad un rapida riassegnazione del lavoro che ha portato alla situazione di cui sopra.
Questo progetto mi ha insegnato però come affrontare un lavoro in team ad un livello di complessità superiore a quello a cui ero abituato, ottenendone un enorme beneficio, oltre che nelle conoscenze delle tecniche di programmazione, anche a livello personale migliorando il mio approccio nelle discussioni e la messa in discussione di soluzioni che ritenevo valide.
In futuro penso di conservare questo progetto nel mio repository privato di git dopo aver applicato qualche correzione ad alcune parti che ancora non mi soddisfano, ad esempio vorrei migliorare la stesura della javadoc, snellire di

molto le classi di View e magari implementare qualche funzionalità aggiuntiva.
Infine una mezione d'onore va anche al Covid che ha costretto tutti e tre i membri del gruppo ad una settimana abbondante di malattia e di quarantena rallentando il lavoro.

4.1.2 Lucia Castellucci

Prima di tutto, ci tengo molto a dire quanto l'impegno e la motivazione dei miei compagni di progetto sia stata fondamentale per raggiungere questo obiettivo.
Detto ciò, nonostante mi sia cimentata in diversi progetti in questi 3 anni di università, Wazzle è stato senza ombra di dubbio quello più faticoso, ma anche quello che ha permesso una crescita maggiore.
Col senno del poi, per quanto riguarda la progettazione, nonostante sia io che i miei compagni ci siamo impegnati moltissimo, avremmo voluto ottenere risultati migliori. Abbiamo progettato poi cestinato e ri-progettato diverse volte alcune porzioni di progetto perché la progettazione non ci convinceva al cento percento.
Spesso abbiamo ottenuto dei risultati che ci hanno convinto, ma altrettanto spesso siamo rimasti un po' delusi dal risultato. Temo che ciò sia dovuto al fatto che abbiamo misurato il risultato in termini di design pattern.
Nonostante ciò, penso che rispetto a quando ho iniziato a progettare e a scrivere il codice le mie capacità siano migliorate molto, grazie a tutte le volte nelle quali ho dovuto approfondire argomenti che conoscevo più superficialmente di quanto non credessi.
Infine, trovo che questo progetto più di tutto abbia migliorato le mie capacità di lavorare in gruppo, che trovo che inizialmente fossero assai scarse.
Complessivamente dunque reputo di aver fatto un buon lavoro, di essermi impegnata al massimo, ma di poter ancora migliorare molto le mie capacità, sia in termini di progettazione che in termini di codice.

4.1.3 Luca Samorè

Wazzle è stato per me il primo progetto di medie dimensioni che ho affrontato. Mi ha permesso da un lato di verificare le mie attuali competenze in termini di progettazione di sistemi software, e dall'altro di migliorare notevolmente le mie skill da programmatore, spingendomi a scrivere codice sempre più pulito ed elegante. È importante sottolineare anche l'impatto che ha avuto l'immensa professionalità e partecipazione dei miei colleghi: nonostante l'abbandono di uno dei membri siamo riusciti, grazie alla buona volontà e al

forte legame instauratosi all'interno del team, a portare a termine il progetto, superando una lunga serie di ostacoli sia interni che esterni all'ambiente di lavoro.
Affrontare un progetto del genere, inoltre, mi ha permesso di riflettere circa i miei punti deboli: non avendo esperienza di system design, ho notato che l'errore più frequente commesso durante lo sviluppo è stato quello di "ragionare per design pattern", con l'obiettivo di utilizzare il maggior numero di pattern possibili, massimizzando l'estendibilità e il riuso. Questa mentalità è sbagliata perché aumenta il rischio di forzare l'utilizzo di un pattern dove invece non andrebbe utilizzato. Spesso, infatti, sono state scartate soluzioni di design parecchio avanzate perché sarebbe stato troppo complicato implementarle.
D'altro canto, la soddisfazione più grande che mi ha lasciato questa esperienza è il risultato ottenuto dall'aver intrapreso un percorso agile: sin dall'inizio del progetto ho spinto i miei colleghi ad adottare questa metodologia di sviluppo e, sebbene non sia stata rispettata a pieno sotto alcuni punti di vista, siamo comunque riusciti a rispettare i suoi punti cardine. Se avessimo favorito il classico approccio waterfall non saremmo riusciti a consegnare il progetto per la deadline scelta.
Wazzle, nel suo denouement, ha contribuito alla mia crescita professionale, migliorando in modo significativo le mie skill di team working e organizzazione del lavoro.

Appendice A

Guida utente

Wazzle è dotato di un menù principale che contiene tre principali alternative:

- GIOCA ORA: Premendo su questo pulsante verrà avviato il gioco principale (Ruzzle). A destra del pulsante di avvio del gioco si trova un pulsante contenente un "?". Cliccando su di esso si verrà indirizzati ad una schermata contenente un tutorial inerente al gioco principale.
- OTTIENI BONUS: Premendo su questo pulsante verrà avviato il minigioco (Wordle). A destra del pulsante di avvio del gioco si trova un pulsante contenente un "?". Cliccando su di esso si verrà indirizzati ad una schermata contenente un tutorial inerente al minigioco.
- STORICO PARITE: Premendo su questo pulsante verranno mostrate le ultime 10 partite fatte al gioco principale. Una volta visionate è possibile tornare al menù principale.