**Game Hub**

Bedeschi Federica, Delja Alesja,

Furegato Silvia

A.A. 2021/2022

Indice

1 Analisi 2

1.1 Requisiti . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.2 Analisi e modello del dominio . . . . . . . . . . . . . . . . . . 4

2 Design 7

2.1 Architettura . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.2 Design dettagliato . . . . . . . . . . . . . . . . . . . . . . . . . 9

3 Sviluppo 19

3.1 Testing automatizzato . . . . . . . . . . . . . . . . . . . . . . 19

3.2 Metodologia di lavoro . . . . . . . . . . . . . . . . . . . . . . . 20

3.3 Note di sviluppo . . . . . . . . . . . . . . . . . . . . . . . . . 21

4 Commenti finali 25

4.1 Autovalutazione e lavori futuri . . . . . . . . . . . . . . . . . . 25

4.2 Difficoltà incontrate e commenti per i docenti . . . . . . . . . 25

A Guida utente 27

B Esercitazioni di laboratorio 28

B.0.1 Bedeschi Federica . . . . . . . . . . . . . . . . . . . . . 28

B.0.2 Delja Alesja . . . . . . . . . . . . . . . . . . . 28

B.0.3 Furegato Silvia . . . . . . . . . . . . . . . . . 28

**Capitolo 1**

**Analisi**

* 1. Requisiti

L’applicazione GameHub si occupa di raccogliere diversi giochi. I giochi proposti sono giocabili in singolo e sono: Flood it, Minefield e Numerical bond.

Requisiti funzionali:

* L’utente potrà decidere a quale gioco giocare tramite il menu principale
* Una volta scelto il gioco, l’utente potrà impostare certi parametri per ottenere varianti diverse dello stesso gioco
* Se il gioco scelto è Flood it, …
* Se il gioco scelto è Minefield, …
* Se il gioco scelto è Numerical bond, l’utente potrà vedere una griglia quadrata di blocchi, ognuno con un certo numero all’interno. A questo punto:
  + L’utente potrà collegare tra loro qualunque coppia di blocchi adiacenti (orizzontalmente o verticalmente), fino a un massimo di due collegamenti per coppia
  + Il numero all’interno del blocco rappresenta il numero di collegamenti che il blocco deve ancora avere, di conseguenza esso si aggiornerà ad ogni collegamento che interessa il blocco stesso
  + Se l’utente, arrivato al massimo di due collegamenti tra due blocchi, si renderà conto che non sono necessari, potrà rimuoverli ripartendo di nuovo dai blocchi scollegati
  + Sarà inoltre concesso di fare più collegamenti del necessario, in tal caso il numero del blocco sarà negativo, a segnalare il fatto che ci sono dei collegamenti di troppo
  + L’utente vincerà la partita nel momento in cui tutti i blocchi della griglia segneranno 0
* Da ogni gioco sarà possibile aprire un menu di pausa, che consentirà all’utente di incominciare una diversa partita dello stesso gioco, di riprendere la partita stessa, oppure di tornare al menu principale interrompendo la partita
* Al termine di ogni gioco verrà visualizzata una schermata che indicherà se il gioco è stato vinto o meno, e permetterà di cominciare una nuova partita oppure di tornare al menu principale per scegliere un altro gioco

Requisiti non funzionali:

* Fluidità e intuitività
  1. Analisi e modello del dominio

Descrizione generica dello schema UML (generico, interfacce, model) del proprio gioco.

Il gioco Numerical bond si comporrà di una griglia quadrata di blocchi. Essi possono essere collegati tra loro, con un massimo di due collegamenti per lato. Ogni blocco dovrà essere collegato a un certo numero di blocchi. La griglia li racchiude, occupandosi di sapere quali blocchi sono presenti e dove, ne saprà determinare la possibilità di collegamento e potrà riconoscere il suo completamento.

Sarà inoltre necessario un generatore di livelli, che dovrà creare una griglia sempre risolvibile. Probabilmente questa sarà una delle difficoltà principali.

**Capitolo 2**

**Design**

2.1 Architettura

L’architettura di GameHub segue il pattern architetturale MVC (Model – View – Controller).

Il controller dell’applicazione è il MainController, che gestisce i menu (la view) e il cambio di contesto tra i vari GameController. Il MainController è di fatto un **Observer** dei GameController, dai quali riceve comunicazioni circa il loro stato (i.e. gioco in pausa, gioco finito); in base alle informazioni ricevute gestisce la view. Essa, invece, notifica al MainController le scelte dell’utente, il quale agisce di conseguenza notificando i GameController o gestendo la view stessa.

L’architettura è progettata in modo tale da permettere l’implementazione di un gioco a partire dal GameController, rendendolo responsabile del controllo del flusso di gioco.

In ogni gioco, presente o futuro, si è però liberi di organizzare la relativa architettura come meglio si crede, fermo restando che la scelta consigliata è il pattern MVC, utilizzando GameController come controller del gioco, GameView come view, e il proprio model. Infatti, l’interfaccia GameController è progettata per essere utilizzata per interfacciare la GameView con il model, senza alcun riferimento esplicito a determinate implementazioni degli stessi.

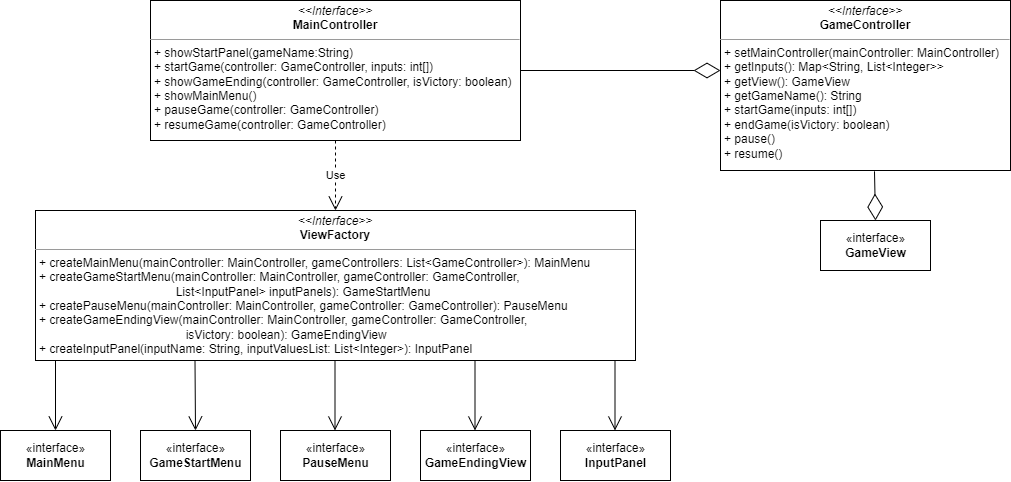


Figura 2.1: Schema UML architetturale di GameHub. Il MainController gestisce i GameController, e le sue implementazioni la view usando la ViewFactory.

I vantaggi nell’adottare questa architettura sono i seguenti:

* Facilità nel modificare o estendere le interazioni tra i GameController e la view dei menu, grazie al fatto che i GameController interagiscono con essa solo attraverso il MainController
* In caso di sostituzione in blocco dell’implementazione della view dei menu, nessuna modifica andrà fatta al MainController, né tanto meno ai GameController. Diverso è il caso della GameView, la cui gestione dell’implementazione rientra nel pattern architetturale scelto nel singolo gioco

2.2 Design dettagliato

Federica Bedeschi

In questa sezione ci si concentrerà su due parti:

* Interazioni tra il MainController, la view principale e i GameController, e implementazioni dei controller
* Interazioni e implementazioni nel gioco Numerical bond

**Indipendenza tra MainController e view principale:**

Problema: L’implementazione del MainController deve essere indipendente dall’implementazione della view principale, e viceversa. Ovvero, la sostituzione totale delle implementazioni delle due parti non deve andare ad impattare il codice dell’altra.

Soluzione: L’implementazione del MainController (MainControllerImpl) utilizza una ViewFactory, da cui prende la view (i menu di gioco), a prescindere dalla loro implementazione. La ViewFactory, come il nome suggerisce, utilizza il pattern **Abstract Factory**, definendo l’interfaccia per la creazione della view, ed utilizzando per essa interfacce della stessa. È quindi l’implementazione ViewFactoryImpl a specificare quali implementazioni della view creare. La view invece è indipendente dall’implementazione del MainController grazie al pattern **Strategy**, che permette all’implementazione del MainController di essere completamente sostituita senza provocare modifiche nella view, la quale ne conosce solo l’interfaccia.

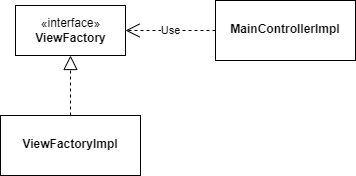
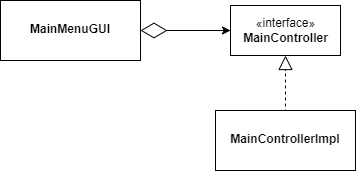


Figura 2.2.2: applicazione del pattern Strategy. Lo stesso vale per tutte le altre parti della view.

Figura 2.2.1: applicazione del pattern Abstract Factory.

**Indipendenza tra MainController e GameController ed estendibilità a più GameController:**

Problemi: L’implementazione del MainController deve essere indipendente dalle implementazioni dei GameController, e viceversa. Deve essere inoltre possibile estendere l’applicazione aggiungendo più giochi, ovvero aggiungendo più GameController.

Soluzione: Questi due problemi sono risolti con un semplice uso di interfacce, ovvero le interfacce e le implementazioni di MainController e GameController contengono solo riferimenti alle rispettive interfacce, GameController in MainController e MainController in GameController. In questo modo, grazie anche all’utilizzo di strutture per mantenere più GameController e al costruttore del MainControllerImpl, è possibile definire un numero variabile di giochi semplicemente passando ad esso le implementazioni dei costruttori dei GameController.

**Riuso di codice per i GameController:**

Problema: Le implementazioni dei GameController hanno molte parti comuni, in particolare quelle che interagiscono con il MainController, di conseguenza è necessario un riuso del codice.

Soluzione: Utilizzo di una classe astratta AbstractGameController che implementa l’interfaccia GameController e cattura le parti comuni alle implementazioni degli stessi. In particolare, è risultato comodo l’utilizzo del pattern **Template Method**. Esso è rappresentato dal metodo getInputs(), il quale deve costruire una Map<String, List<Integer>>, di conseguenza esso si occupa della sua creazione e della return, lasciando al metodo astratto e protetto addInputs() il compito di aggiungere input; ogni implementazione di GameController implementerà quindi quest’ultimo metodo in base alle proprie esigenze.

Immagine che contiene tavolo

Descrizione generata automaticamente

Figura 2.2.3: applicazione del pattern Template Method. Lo stesso vale per tutti gli altri GameController

**Indipendenza tra NumericalBondController e NumericalBondView:**

Problema: L’implementazione del NumericalBondController deve essere indipendente da quella della NumericalBondView, e viceversa. Ovvero, la sostituzione totale delle implementazioni delle due parti non deve andare ad impattare il codice dell’altra.

Soluzione: Utilizzo del pattern **Strategy** per entrambe le implementazioni (NumericalBondControllerImpl e NumericalBondGUI).

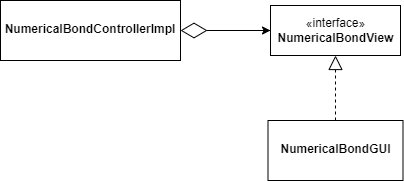
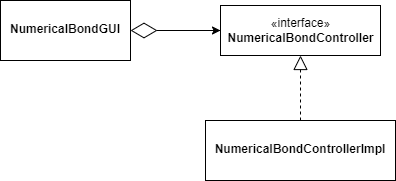


Figura 2.2.4: applicazione del pattern Strategy per NumericalBondControllerImpl.

Figura 2.2.5: applicazione del pattern Strategy per NumericalBondGUI.

**Indipendenza tra NumericalBondController e il model del gioco (rappresentato da Grid):**

Problema: L’implementazione del NumericalBondController deve essere indipendente da quella del model (Grid), e viceversa. Ovvero, la sostituzione totale delle implementazioni delle due parti non deve andare ad impattare il codice dell’altra.

Soluzione: La NumericalBondControllerImpl si riferisce al model solo grazie all’interfaccia Grid, ragion per cui non dipende dalla sua implementazione. Grid, invece, è totalmente all’oscuro dell’esistenza del NumericalBondController, men che meno delle sue implementazioni.

Alesja Delja

Silvia Furegato

**Capitolo 3**

**Sviluppo**

3.1 Testing automatizzato

Per il testing automatizzato è stato utilizzato JUnit (versione 5).

Per ogni gioco è presente un test differente, al fine di eseguire un testing più organizzato:

* FloodItTest:
* MinefieldTest:
* NumericalBondModelTest: si occupa di testare il model del gioco Numerical bond. Di conseguenza cerca di fare il testing delle funzionalità principali di Block (BlockImpl) e Grid (GridImpl). In particolare, si è rivolta attenzione a chiamate di metodi con parametri null o non accettabili dal dominio del parametro. I dettagli dei test sono visibili dalla Javadoc.

Per quanto la riguarda il testing della view, esso è stato effettuato manualmente, sia su Windows che su Linux.

3.2 Metodologia di lavoro

Il lavoro è stato suddiviso come segue:

* Federica Bedeschi: sviluppo del gioco Numerical bond (package main.games.numericalbond) e sviluppo dell’architettura di base dell’applicazione (packages main.gamehub.controller, main.gamehub.model, main.general). In particolare:
  + Interfacce sviluppate:
    - NumericalBondController, NumericalBondView, LevelGenerator, Grid, Block
    - MainController, GameController, GameView, ViewFactory, View (che viene estesa dalle seguenti interfacce, di fatto per ora vuote, MainMenu, GameStartMenu, PauseMenu, GameEndingView), InputPanel
  + Classi sviluppate:
    - NumericalBondControllerImpl, NumericalBondGUI, GamePanel, Link, LevelGeneratorImpl, GridImpl, BlockImpl, Position, NumericalBondModelTest
    - MainControllerImpl, AbstractGameController, InputPanelImpl (parte di logica), MainMenuGUI (parte di logica), Launcher
  + Enumerazioni sviluppate:
    - Direction
* Alesja Delja: sviluppo del gioco Minefield (package main.games.minefield) e sviluppo della schermata di fine gioco. In particolare:
  + Classi sviluppate:
    - MinefieldController, Field, Cell, Handler, ViewField, MinefieldTest
    - GameEndingGUI
* Silvia Furegato: sviluppo del gioco Flood it (package main.games.floodit), sviluppo dei menu di inizio gioco e di pausa e standardizzazione della grafica dell’applicazione. In particolare:
  + Interfacce sviluppate:
    - MovesCounter
  + Classi sviluppate:
    - FloodItController, FloodItView, GamePanel, FloodItModel, Table, Cell, MaxMovesCounter, Pair, FloodItTest
    - GameStartMenuGUI, PauseMenuGUI, InputPanelImpl (parte di grafica), MainMenuGUI (parte di grafica)
  + Enumerazioni sviluppate:
    - Colors
    - DashboardColor

Le parti di codice sviluppate separatamente sono state integrate tra loro grazie all’uso delle interfacce che le relative classi dovevano implementare. Non ci sono stati particolari problemi, in caso di modifiche alle interfacce comuni si sono effettuate sistemazioni in maniera efficace.

Il DVCS, Git, è stato utilizzato creando un repository principale di un componente del gruppo, del quale sono state fatte le fork per i restanti componenti. Dalle fork venivano effettuate pull requests per unificare il lavoro svolto col repository principale.

3.3 Note di sviluppo

Federica Bedeschi

Utilizzo di: