**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.

Il gioco Minefield consiste in una griglia di blocchi che non sono visibili all’inizio del gioco. Al player viene chiesta all’inizio di gioco oltre a quanto vuole grande la griglia anche il numero di mine. Queste mine vengono piazziate nella griglia in modo random e la loro posizione verrà memorizzata in un array. In ogni cella si può vedere se la cella è vuota ha la mina o se essa è ha la flag e si comporterà in modo diverso a seconda del tipo.

**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.

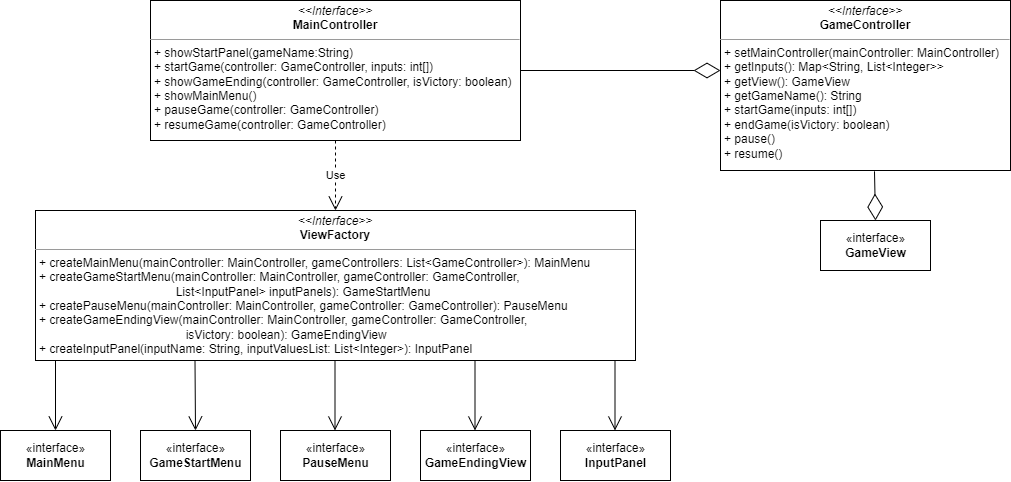


Figura 2.1: Schema UML architetturale di GameHub. Il MainController gestisce i GameController, e le sue implementazioni la view usando la ViewFactory.

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.

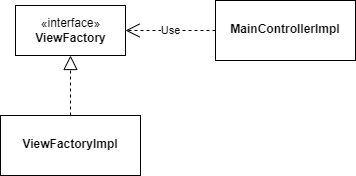
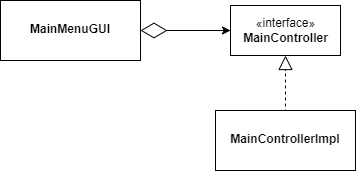


Figura 2.2.2: applicazione del pattern Strategy. Lo stesso vale per tutte le altre parti della view.

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).

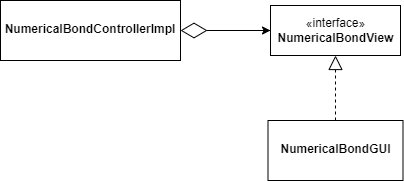
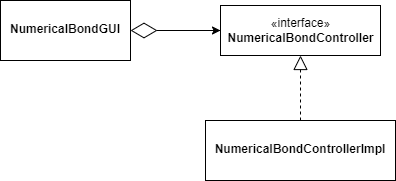


Figura 2.2.4: applicazione del pattern Strategy per NumericalBondControllerImpl.

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.

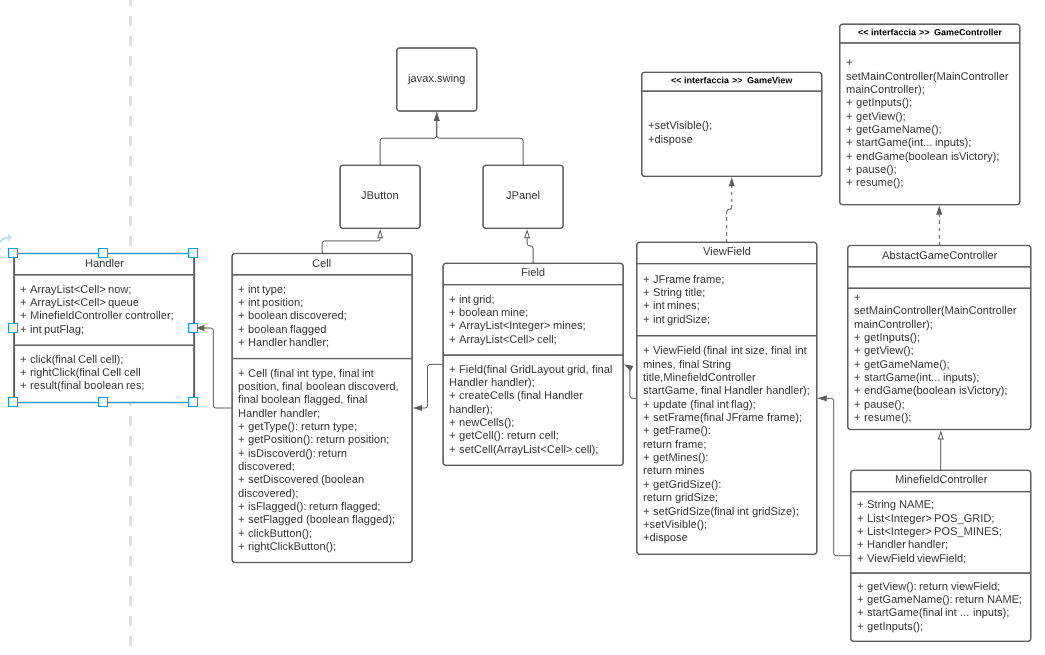
Alesja Delja:

In questa sezione ci si concentrerà su due parti:

* Implementazione del EndGame
* Implementazione del gioco Minefield

**Implementazione del gioco:**

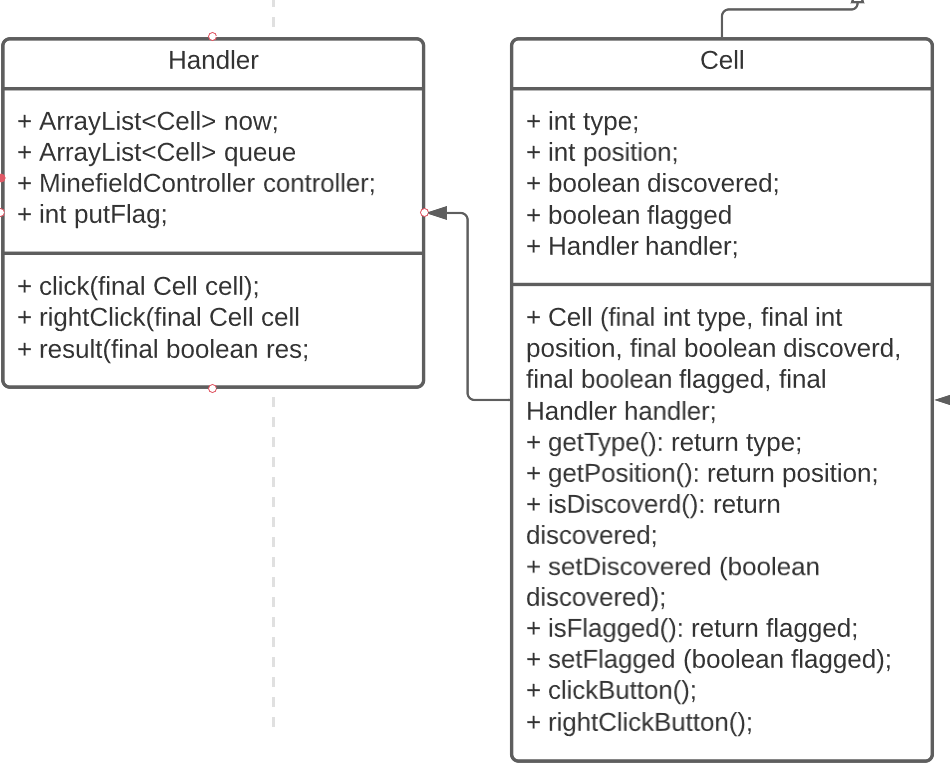
Problema: Inizialmente non capivo come dividere i vari le varie parti della creazione del campo di gioco in modo da rendere anche il meno ripetitivo il codice.



Soluzione: Ho optato per una classe viewField che creava il JFrame necessario a mettere i due pannelli necessari: uno per quello della pausa e uno per mettere la grid. Ho poi creato una classe Cell che estende il JButton per crea i diversi pulsanti e che cerca di capire se la cella ha una mina, è flaggata oppure è una cella vuota. Come ultima cosa ho creato la classe Field che estende JPanel e che crea tutta la griglia dei bottoni memorizzando la loro posizione e la posizione delle mine che venivano create.

**Implementazione del Handler**

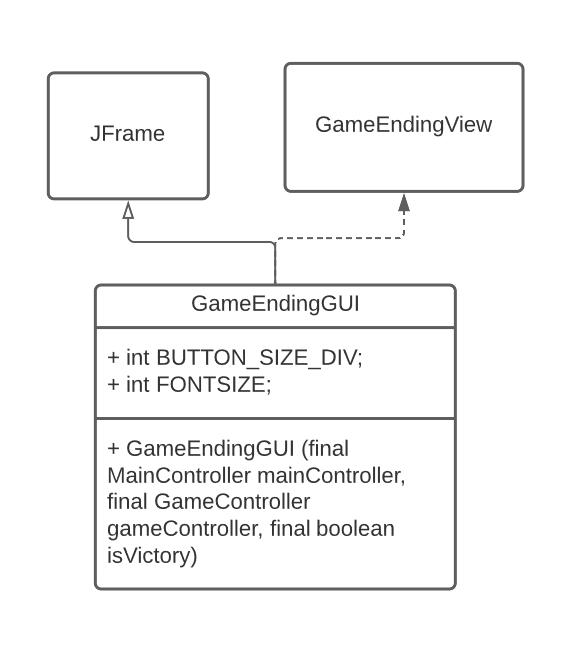
Problema: Volevo trovare un modo veloce e rapido per rendere la ricerca delle mine il più veloce possibile.



Soluzione: Ho creato la classe Handler che con il metodo click controlla se la cella premuta ha una flag o meno e se vicino a lei nelle celle adiacenti ci siano delle mine vicine facendo salire il dangerCount ogni volta che ne trova una. Queste celle inoltre vengono memorizzate nell’ArrayList queue. Inoltre, ogni qualvolta venga premuto una mina o finisca il gioco viene chiamato il metodo result che fa apparire l’interfaccia dell’EndGame con la vittoria o la perdita del giocatore. Il metodo rightClick invece permette di inserire le flag dal player aggiornando anche la view con il numero di Flag presenti nella griglia. Click viene chiamato così da ClickButton nella classe Cell e rightClick viene chiamto dal metodo rightClickButton della stessa classe.

Implementazione GameEnding:

Problema: creare una classe che capiva quando un gioco finiva e far uscire il risultato della partita e la possibilità di creare un nuovo gioco o andare al menu principale.



Soluzione: Ho implementato una classe che ha come costruttore 3 diversi input: il primo serve per passare al menu principale quando viene premuto il bottone “Return to main menu”, il secondo serve per fare capire alla classe quale gioco è appena finito per così sapere quale gioco deve essere aperto quando si vuole rigiocarlo premendo “New Game”. L’ultimo input serve a capire se il giocatore ha vinto o meno il gioco.

Silvia Furegato:

**Flood It**

Per la realizzazione del gioco ho adottato il pattern MVC, per facilitare una futura modifica di funzionalità o visualizzazione. Inoltre controller e view si adeguano alle interfacce GameController e GameView che consentono alla dashboard di interfacciarsi con il gioco.



**Problema: Calcolo delle mosse massime.**

Si vuole dare la possibilità di cambiare l’algoritmo che calcola le mosse massime. Per questo motivo ho deciso di adottare il pattern Strategy, lasciando così aperta la possibilità, in futuro, di cambiare il calcolatore di mosse massime con uno più sofisticato oppure quella di lasciare all’utente la scelta del calcolatore di mosse da utilizzare.



**Problema: Colori personalizzati.**

Per la colorazione delle caselle volevo utilizzare una palette di colori personalizzata. Pertanto ho creato una enum con i colori scelti, in questo modo è possibile apportare facilmente modifiche alla palette di gioco ed in più presenta metodi riusabili anche al di fuori dello sviluppo di Flood It.

**Dashboard**

Mi sono occupata della realizzazione di parte della dashboard view di Game Hub, ho realizzato le classi PauseMenuGUI e GameStartMenuGUI ed ho adeguato, solamente nella parte grafica, MainMenuGUI e GameEndingGUI. Anche in questo caso, per mantenere una maggior coesione e facilità di modifica nello stile della view, ho utilizzato una enum per definire i colori della dashboard.

**Capitolo 3**

**Sviluppo**

3.1 Testing automatizzato

I diversi test sono:

* MinefieldTest: la classe consiste nel controllare che la gridsize sia impostata nel modo giusto creando subito dopo una ViewField 2\*2 con 2 mine vedendo se anche se passo valori più bassi di quelle delle insert esso funziona. L’altro metodo testa se l’array delle mine viene correttamente riempito controllando con un array casuale se viene riempito o meno.
* Per quanto riguarda il gioco Flood It è stata testata tramite JUnit la corretta generazione della tabella ed il corretto funzionamento di alcune classi del model.
* TestEnumColors: Controlla che il metodo getRandomColors() lanci un’eccezione nel caso gli venga passato in input un numero di colori troppo alto o troppo basso.
* TestCellFlooding: Controlla che il metodo flood() della casella funzioni correttamente.
* TestMaxMovesGeneration: Testa che il MaxMovesCounter conteggi correttamente le mosse massime.
* TestTable: Controlla che la generazione della tabella sia avvenuta correttamente. Prende come casella di controllo la prima in alto a sinistra e controlla che quelle adiacenti siano state collegate correttamente. Dove non è prevista una casella troverà null.
* TestModelResetting: Dopo aver parzialmente popolato il model, controlla che il metodo clear() resetti correttamente il model, azzerando ogni suo campo.

In aggiunta ai test JUnit, il gioco è stato testato anche manualmente.