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 3

2 Design **6**

2.1 Architettura 6

2.2 Design dettagliato 7

3 Sviluppo **14**

3.1 Testing automatizzato 14

3.2 Metodologia di lavoro 15

3.3 Note di sviluppo 16

4 Commenti finali **17**

4.1 Autovalutazione e lavori futuri 17

A Guida utente **18**

B Esercitazioni di laboratorio **20**

B.0.1 Bedeschi Federica 20

Capitolo 1
Analisi

1.1 Requisiti

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

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 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
- Se il gioco scelto è Minefield, l'utente potrà vedere una griglia quadrata di celle, inizialmente vuote (ovvero coperte). A questo punto:
    - L'utente potrà scegliere da quale cella partire e cercare di trovare le celle senza mina, scoprendole, e guardando i numeri che appaiono e che segnano quante mine sono vicine a quella cella
    - Durante il gioco l'utente può mettere in qualsiasi momento delle flag, ad indicazione del fatto che quella cella potrebbe contenere una mina; in questo modo la cella verrà disabilitata fino a quando l'utente non toglierà la flag
    - Il gioco sarà concluso solo quando l'utente riuscirà a scoprire tutte le celle che non contengono mine
- Se il gioco scelto è Flood it, all'avvio si potrà vedere una griglia quadrata di caselle colorate. A questo punto:
    - L'utente potrà selezionare una qualsiasi casella.
    - Quando una casella viene selezionata, il colore della casella in alto a sinistra cambia, diventando dello stesso colore della casella appena selezionata.

    - Se vicino alla casella in alto a sinistra ci sono delle caselle dello stesso colore, queste si “uniranno” a quella in alto a sinistra comportandosi d’ora in avanti come un'unica cella.
    - Ogni volta che viene selezionata una cella un contatore di mosse aumenta.
    - Quando tutta la tabella avrà lo stesso colore l’utente avrà vinto.
    - Se l’utente non riesce a completare il gioco entro un numero prestabilito di mosse avrà perso.
- 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

1.2 Analisi e modello del dominio

Il dominio dell’applicazione si compone di tre parti differenti e distaccate, che corrispondono al dominio di ogni gioco proposto.

Gioco Numerical bond:

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 ogni direzione. Ogni blocco dovrà essere collegato a un certo numero di blocchi. La griglia li racchiude, occupandosi di sapere quali blocchi sono presenti e in quale posizione, ne saprà determinare la possibilità di collegamento ed effettuarlo, e saprà riconoscere il proprio completamento.

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

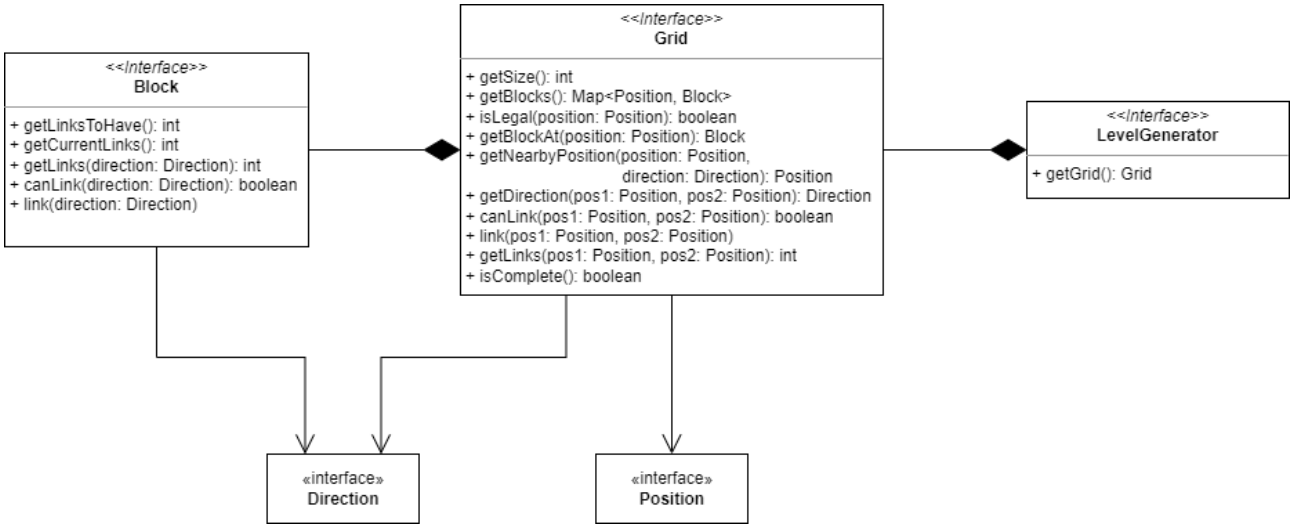


Figura 1.2.1: schema UML del dominio del gioco Numerical bond.

Gioco Minefield:

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 piazzate 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, se ha la mina o se essa ha la flag, e si comporterà in modo diverso a seconda del tipo.

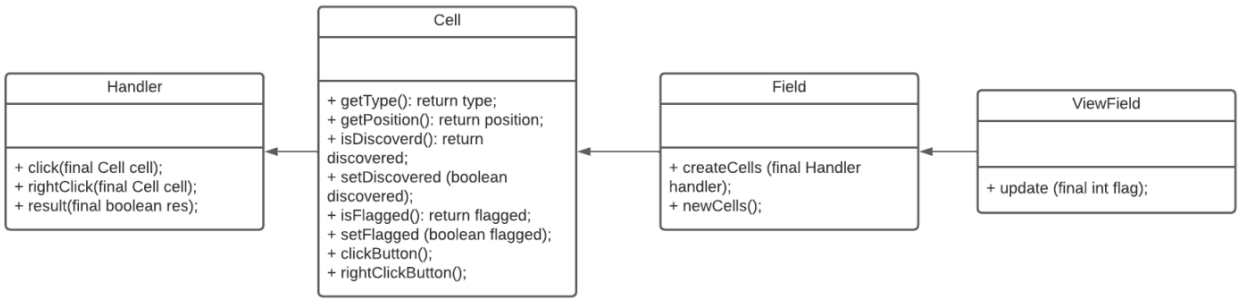


Figura 1.2.2: schema UML del dominio del gioco Minefield.

Gioco Flood it:

Il gioco Flood it sarà composto da una griglia quadrata di caselle. Prima di iniziare a giocare, all'utente verranno chiesti la dimensione della tabella ed il numero dei colori con cui si giocherà e la partita sarà generata di conseguenza.

Quando l'utente fa la sua mossa, le celle devono essere in grado di capire se le caselle intorno hanno lo stesso colore e di conseguenza "unificarsi".

Sarà necessario un calcolatore di mosse, per generare il numero di mosse massime concesse all'utente.

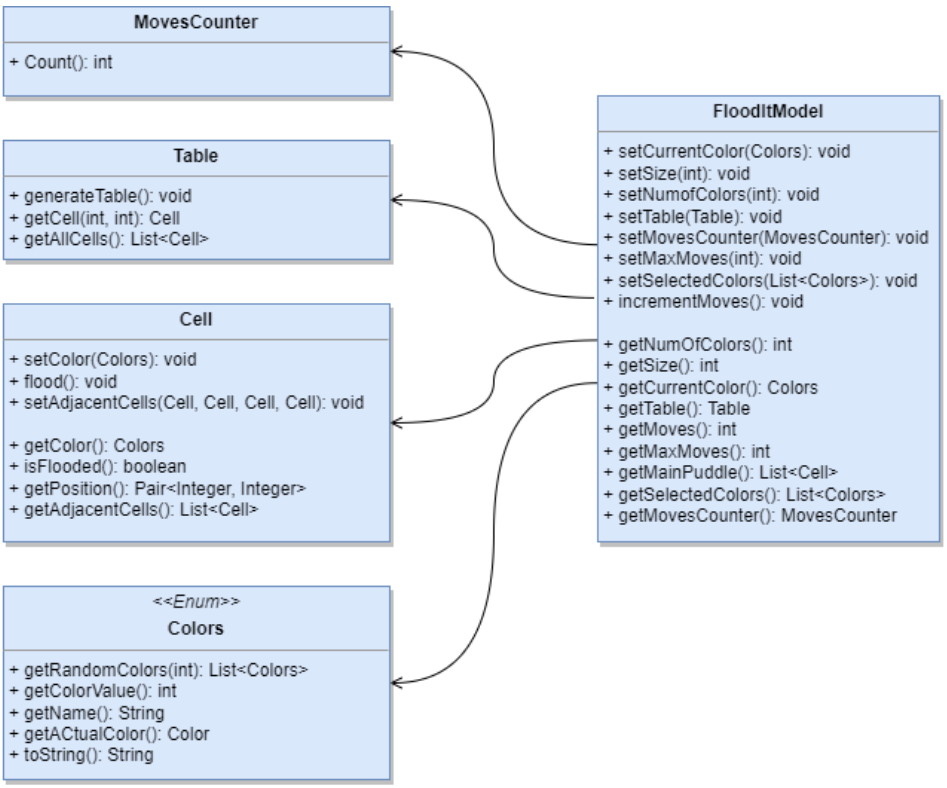


Figura 1.2.3: schema UML del dominio del gioco Flood it.

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. Quest'ultima, 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.

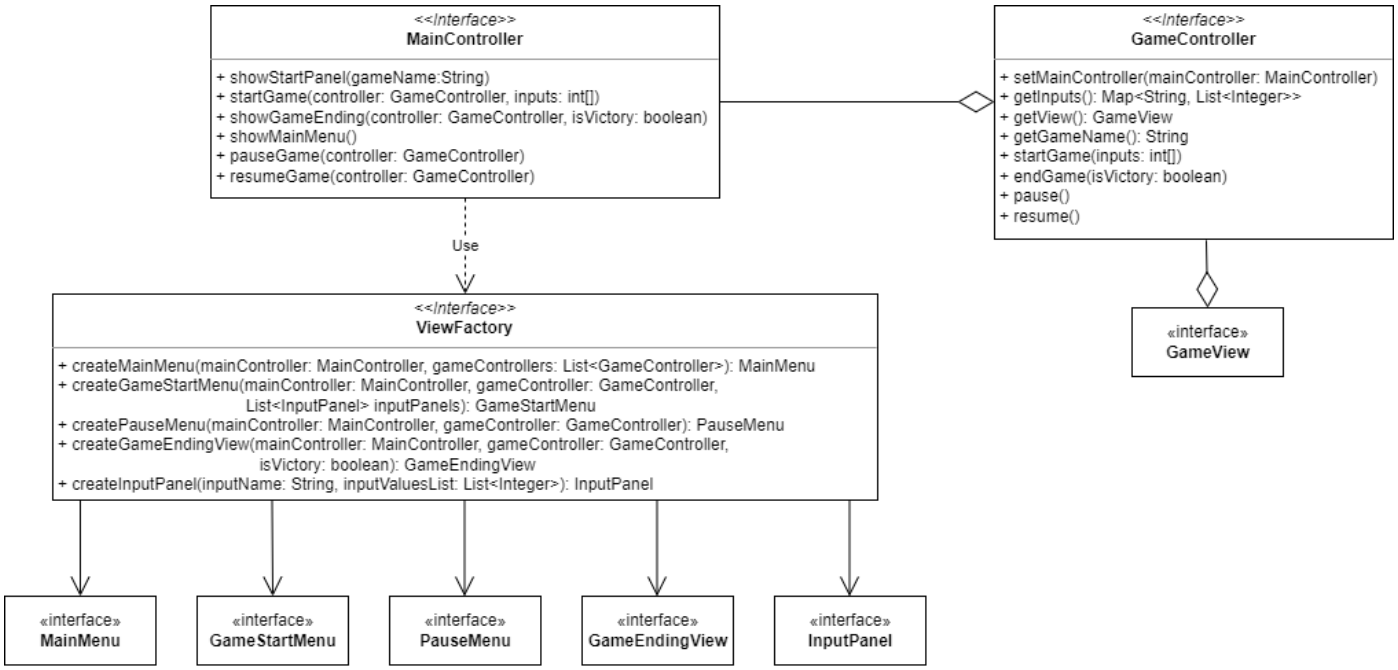


Figura 2.1: Schema UML architetturale di GameHub. Il MainController gestisce i GameController, e le sue implementazioni la view usando la ViewFactory.

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

Bedeschi Federica

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

L'architettura del gioco **Numerical bond** segue il **pattern architetturale MVC**. Il controller del gioco è il NumericalBondController, la view è rappresentata dalla NumericalBondView e il model è rappresentato dalla Grid.

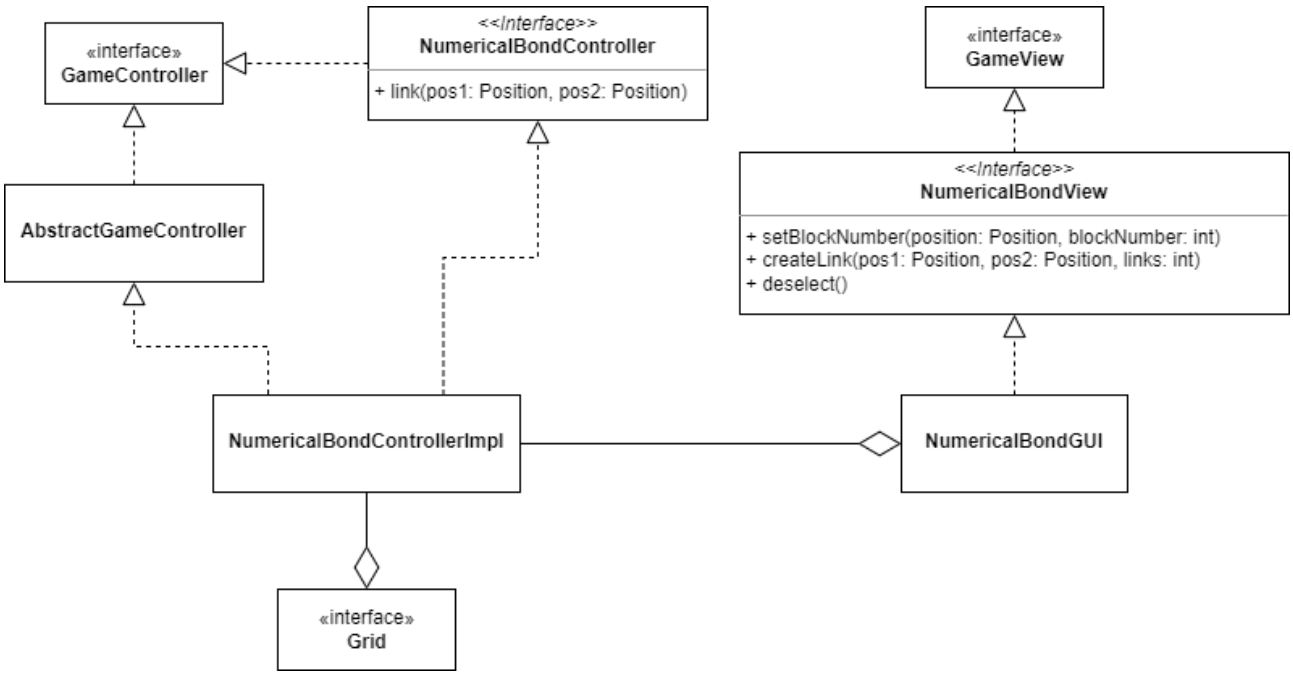


Figura 2.2.1: schema UML architetturale del gioco Numerical bond.

Nota: l'interfaccia Position qui non rappresentata è usata come classe di utilità. È di fatto una Pair<Integer, Integer> e viene utilizzata come "tipo di dato", analogamente a List o Map. Il concetto di posizione è ovviamente analogo sia nel model che nella view, di conseguenza l'interfaccia Position è comune alle due, ma non necessariamente l'implementazione.

Nella seguente parte sono elencati alcuni problemi affrontati e le relative soluzioni adottate.

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.

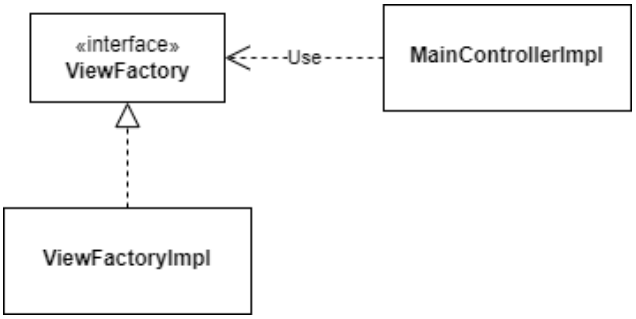


Figura 2.2.2: applicazione del pattern Abstract Factory.

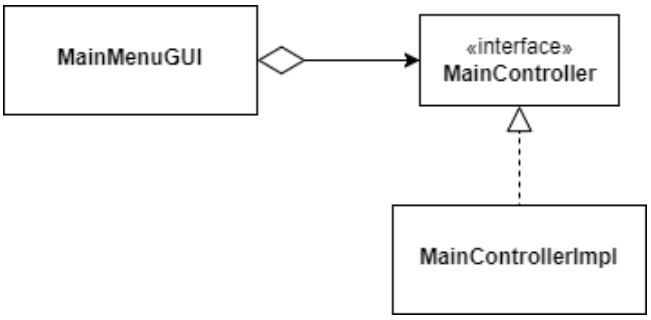


Figura 2.2.3: applicazione del pattern Strategy. Lo stesso vale per tutte le altre parti della view.

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 istanze delle diverse implementazioni 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.

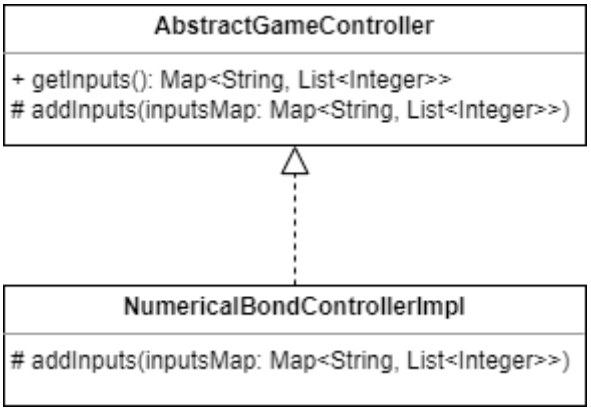


Figura 2.2.4: applicazione del pattern Template Method.
Lo stesso vale per tutti gli altri GameController

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).

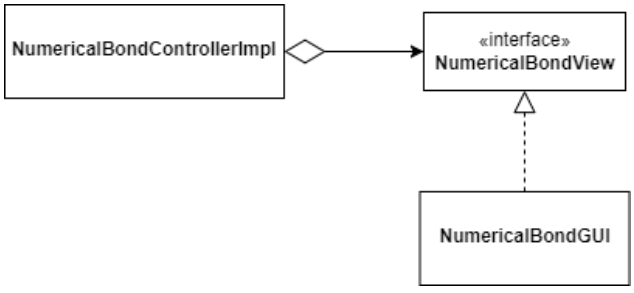


Figura 2.2.5: applicazione del pattern Strategy per NumericalBondControllerImpl.

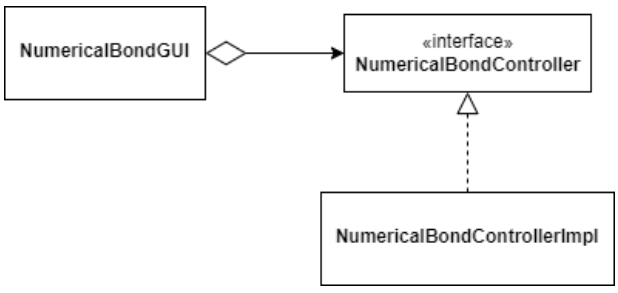


Figura 2.2.6: applicazione del pattern Strategy per NumericalBondGUI.

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.

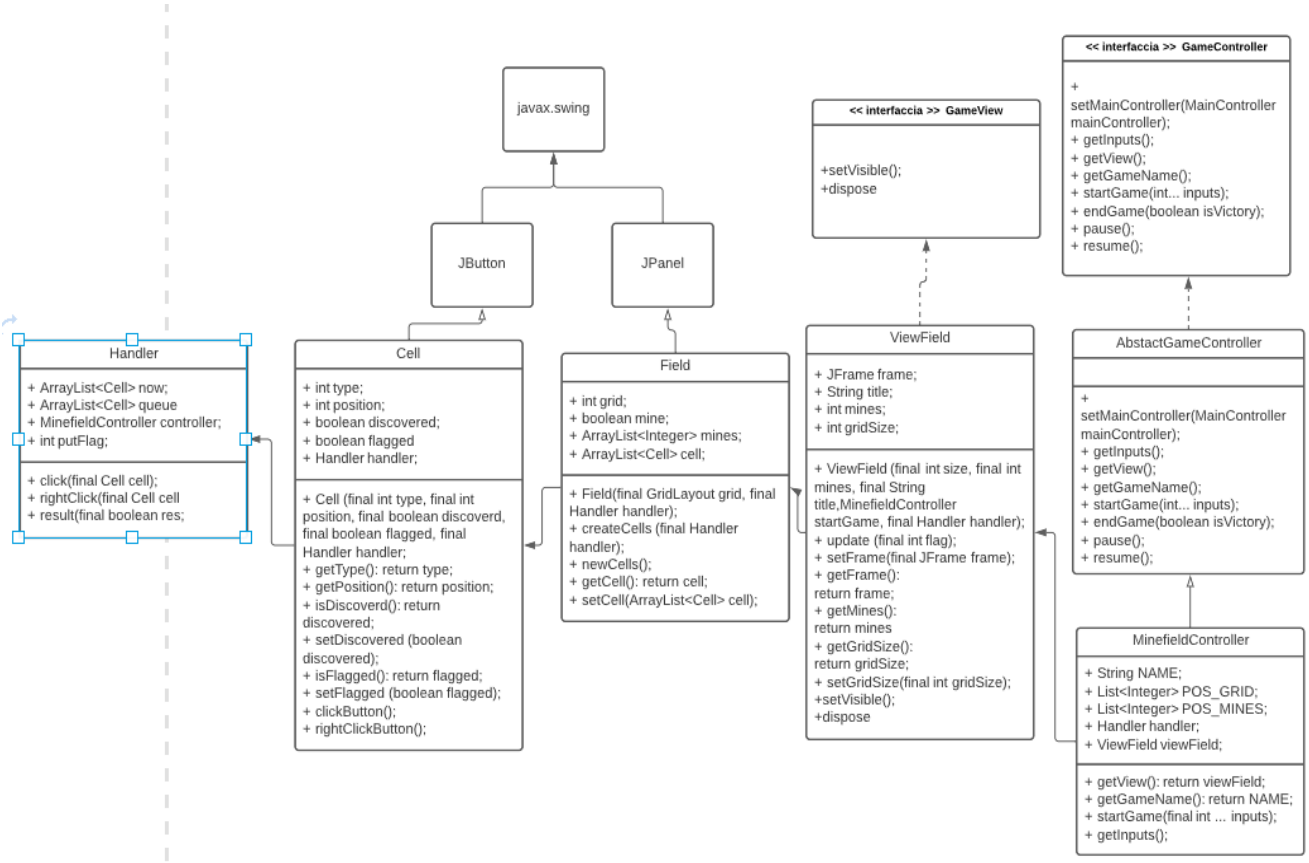
Delja Alesja

In questa sezione ci si concentrerà su due parti:

- Implementazione della GameEndingGUI
- Implementazione del gioco Minefield

Implementazione del gioco:

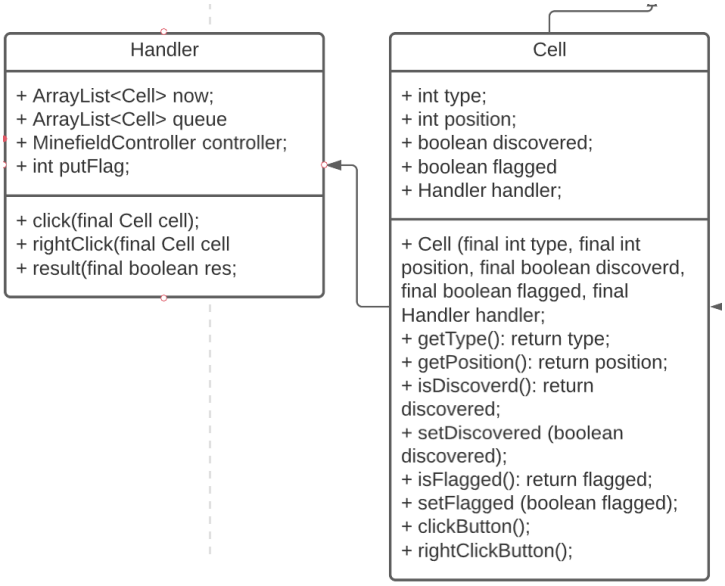
Problema: Inizialmente non capivo come dividere le varie parti della creazione del campo di gioco in modo da rendere anche meno ripetitivo il codice.



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 creare 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

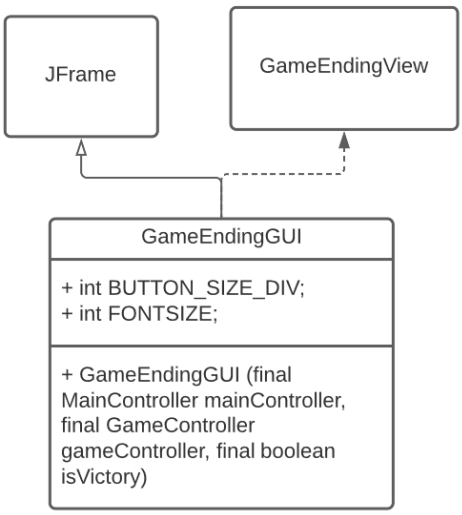
Problema: Volevo trovare un modo veloce e rapido per rendere la ricerca delle mine il più veloce possibile.



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 chiamato dal metodo rightClickButton della stessa classe.

Implementazione GameEnding:

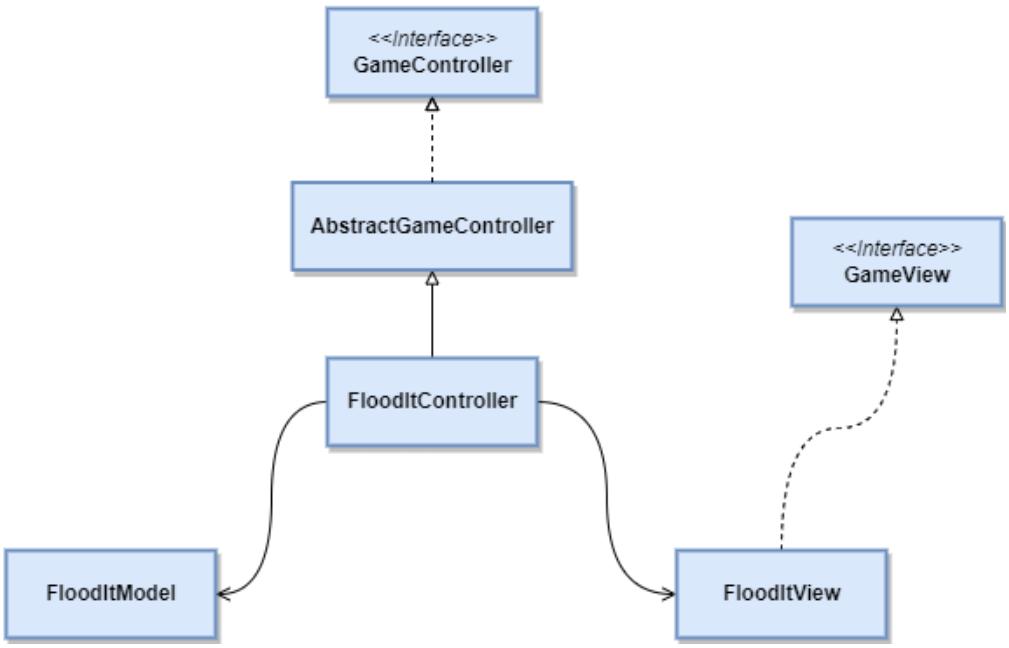
Problema: creare una classe che capiva quando un gioco finiva e far uscire il risultato della partita e la possibilità di creare un nuovo gioco o andare al menu principale.



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.

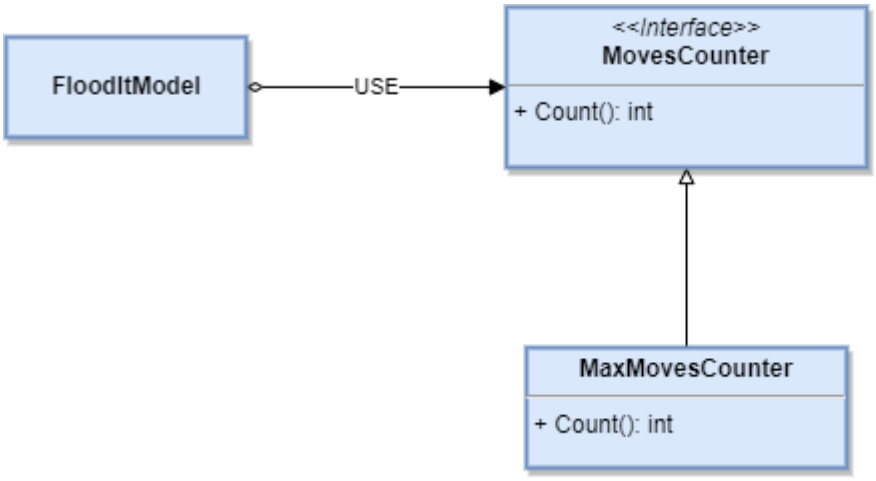
Furegato Silvia

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, altre classi della dashboard view. 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

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:

- NumericalBondModelTest: si occupa di effettuare il testing del 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.
- 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.
- FloodItTest: si occupa di effettuare il testing della corretta generazione della tabella di gioco ed il corretto funzionamento di alcune classi del model. In particolare:
    - 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.

Per quanto la riguarda il testing della view, esso è stato effettuato manualmente.

3.2 Metodologia di lavoro

Il lavoro è stato suddiviso come segue:

- Bedeschi Federica: 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, Position
        - 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, PositionImpl, NumericalBondModelTest
        - MainControllerImpl, AbstractGameController, InputPanelImpl (parte di logica), MainMenuGUI (parte di logica), Launcher
    - Enumerazioni sviluppate:
        - Direction
- Delja Alesja: 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
- Furegato Silvia: 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

Bedeschi Federica

Utilizzo di:

- **Lambda expressions:** presenza in varie parti del codice per definire ActionListener e Runnable, e in conseguenza agli Stream
- **Stream:** presenza in varie parti del codice, soprattutto in presenza di List o Map
- **Optional:** presenza in varie parti del codice, soprattutto nei tipi di ritorno di alcuni metodi.

Furegato Silvia

Utilizzo di:

- **Generici:** Ho utilizzato la classe Pair<X, Y> che ci è stata fornita a lezione per memorizzare la posizione delle caselle nella tabella di gioco.
- **Lambda Expression:** Utilizzate per snellire il codice.
- **Stream:** Ho utilizzato uno stream nella enum Colors per la “traduzione” di un java.awt.Color in un colore della enum.

Capitolo 4
Commenti finali

4.1 Autovalutazione e lavori futuri

Bedeschi Federica

Tutto sommato sono abbastanza soddisfatta del lavoro che ho svolto. Essendo la prima volta che progetto l'architettura di un software ritengo di aver fatto un discreto lavoro, utilizzando interfacce e qualche pattern, anche se sicuramente avrei potuto fare meglio, soprattutto nella parte della view. Il gioco Numerical Bond credo sia anch'esso progettato abbastanza bene, anche se le cose da sistemare non mancano neanche qui (model in particolare).

Le varie implementazioni le ritengo buone, anche se avrei voluto cimentarmi un po' di più nell'uso di Stream e Lambda expressions, li ritengo uno strumento molto utile e potente, che può allo stesso tempo rendere il codice più chiaro e conciso; ma il tempo è sempre un fattore limitante, e per come ho organizzato le cose non sono riuscita ad averne abbastanza. Anche le classi del model di Numerical bond avrei potuto organizzarle meglio, dividendole in più classi più corte.

Delja Alesja

Avrei potuto organizzare meglio il tempo a mia disposizione e creare un progetto più pulito e chiaro usando molte cose che avrebbero facilitato il mio lavoro e che quindi mi avrebbero permesso di aver un prodotto migliore.

Furegato Silvia

Ritengo che nella fase iniziale della progettazione avremmo dovuto lavorare di più sulla parte comune e sulle varie interfacce prima di iniziare la realizzazione dei singoli giochi, infatti sono convinta che l'iniziale mancanza di interfacce abbia notevolmente rallentato lo sviluppo dei giochi: aumentando il carico di lavoro a causa degli adeguamenti che è stato necessario fare per il corretto funzionamento dei giochi e diminuendo la qualità del codice prodotto.

Nonostante ciò, mi ritengo abbastanza soddisfatta del risultato finale. Penso che comunque sia stata un'esperienza molto stimolante, che ci ha allenato al lavoro in team, all'utilizzo di git come gruppo e ci ha mostrato per intero il processo di realizzazione di un progetto.

Appendice A
Guida utente

L’utilizzo dell’applicazione è intuitivo, ma ne viene fornita lo stesso una guida qualora ci fossero dubbi.

Utilizzo dell’applicazione (parti generali):

All’avvio dell’applicazione si apre il **menu principale**, che permette all’utente di scegliere il gioco da giocare premendo sul relativo pulsante. A questo punto si apre un **menu di inizio gioco**, che permette di impostare dei parametri di gioco tramite dei menu a tendina, e di iniziare il gioco così impostato premendo il pulsante Play.

Una volta all’interno di un gioco, sarà sempre possibile aprire un menu di pausa, cliccando sul pulsante situato in alto. Nel **menu di pausa** sono presenti tre pulsanti: New game (permette di iniziare una nuova partita, si apre il menu di inizio del relativo gioco), Resume (permette di continuare la partita in corso) e Exit (permette di tornare al menu principale).

Terminato il gioco, viene mostrata una schermata che contiene l’**esito della partita** e due pulsanti: New game (analogamente a prima, permette di iniziare una nuova partita, aprendo il menu di inizio del relativo gioco) e Return to main menu (permette di tornare al menu principale).

Utilizzo del gioco Numerical bond:

Si compone di una griglia quadrata di pulsanti bianchi. E’ possibile cliccare su di essi. Alla pressione di un pulsante questo si scurisce, ad indicazione del fatto che è stato selezionato (è possibile deselezionarlo cliccandolo di nuovo). Alla pressione di un secondo pulsante, se questo è adiacente verticalmente o orizzontalmente al primo, i due pulsanti si collegano con una linea, altrimenti non succede niente. Quando il collegamento viene effettuato, i numeri contenuti nei pulsanti, che indicano il numero di collegamenti che quel blocco deve raggiungere, diminuiscono di uno. Con lo stesso meccanismo è possibile effettuare un secondo collegamento tra due blocchi già collegati. Il massimo di collegamenti tra due blocchi è due, ragion per cui selezionando di nuovo i due blocchi i due collegamenti si azzerano, permettendo al giocatore di cambiare le proprie scelte. I blocchi segnalano il loro completamento riportando il numero 0 e colorandosi di verde. Se un blocco già a 0 viene ulteriormente collegato, questo mostrerà un numero negativo e si colorerà di rosso, al fine di segnalare al giocatore gli eccessivi collegamenti. Una volta che tutti i pulsanti saranno verdi, il gioco sarà vinto.

Utilizzo del gioco Minefield:

L’utilizzo è intuitivo.

Unica nota sulle flag: per metterle e toglierle basta fare click destro con il mouse e il contatore in alto si aggiorna in automatico.

Utilizzo del gioco Flood it:

Lo scopo del gioco è quello di riempire una tabella di caselle colorate con uno stesso colore, in un numero massimo di mosse. Si parte dalla casella in alto a sinistra, ogni volta che viene selezionata una casella, il colore della casella in alto a sinistra cambia nel colore della casella selezionata; se vicino ad essa ci sono caselle con lo stesso colore, verranno “allagate” ed al click successivo si comporteranno come un'unica grande casella.

Appendice B
Esercitazioni di laboratorio

B.0.1 Bedeschi Federica

- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=88829
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=89272
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=90125
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=91128