DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

- Corso di Laurea Triennale in Ingegneria Informatica -

Relazione di progetto per il corso di Programmazione Avanzata Java e C

H1V3

di

Nicol Stocchetti

matr. 734131

- Anno Accademico 2021/2022 -

Indice

Il Gioco

Questo progetto vuole essere una trasposizione digitale del famoso gioco da tavolo a due giocatori “**Hive**”, creato da **John Yianni**.

Regole

Hive è un gioco di posizionamento, non dissimile per struttura al gioco degli scacchi, in cui due giocatori (bianchi contro neri) si affrontano con l’obiettivo di neutralizzare l’**ape regina** dell’avversario.
Ogni giocatore ha a disposizione 11 pedine esagonali, ciascuna delle quali raffigura un insetto che può spostarsi lungo l’alveare in modi diversi a seconda della specie.

La partita ha inizio di fronte a un tavolo vuoto, dove a turni alterni ciascun giocatore può eseguire una delle seguenti mosse:

- **Posizionare un insetto** del proprio colore nell’alveare (finché ne restano a disposizione);
- **Muovere un insetto** già posizionato (se in grado di farlo).

L’**alveare** è costituito dall’insieme delle pedine posizionate sul tavolo di gioco.
Se si desidera mettere in gioco un insetto, rendendolo così parte dell’alveare, lo si può fare agganciandolo ad altre pedine già in gioco, con la condizione che esso debba essere posizionato unicamente a contatto con altri insetti del proprio colore, assicurandosi che nessuno dei suoi lati tocchi un insetto del colore avversario.

L’unica **eccezione** a questa regola ha luogo nel turno d’apertura, in cui, per poter creare l’alveare, il secondo giocatore dovrà posizionare un proprio insetto direttamente a contatto con quello precedentemente piazzato dall’avversario.

Una volta che un pezzo viene messo in gioco, **non può più essere rimosso dall’alveare**.

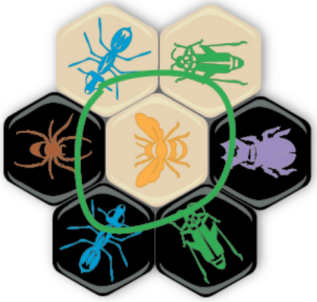
Per entrambi i giocatori vale la regola che l'ape regina deve entrare obbligatoriamente in gioco **entro il quarto turno** e finché essa non entra in gioco gli altri insetti non si possono muovere.

Per l'intera durata della partita l'alveare deve sempre costituire un **blocco unico** e continuo, può presentare aperture o cavità al proprio interno ma non può mai dividersi in due (o più) pezzi separati, nemmeno per eseguire un movimento: qualunque insetto che, spostandosi, spezzerebbe l'alveare (anche solo per un instante nell'arco del suo movimento) è da considerarsi impossibilitato a muoversi.

In generale le creature possono muoversi lungo l'alveare solo in movimento **scivolatorio**: se un pezzo è circondato a tal punto che, fisicamente, non può più scivolare fuori dalla sua posizione, allora non può essere spostato. Allo stesso modo, nessun pezzo può muoversi in uno spazio in cui non possa fisicamente scivolare.

Nel caso in cui un giocatore non possa né muovere, né aggiungere nuovi pezzi in campo, **salterà il proprio turno** e il gioco procederà in questo modo per un numero indefinito di turni, finché egli non sarà nuovamente in grado di compiere una mossa.

Vince chi riesce a intrappolare per primo l'ape regina avversaria, circondandola su tutti i lati. Nel caso in cui l'ultima mossa determini l'accerchiamento di entrambe le api regine contemporaneamente, la partita è patta.

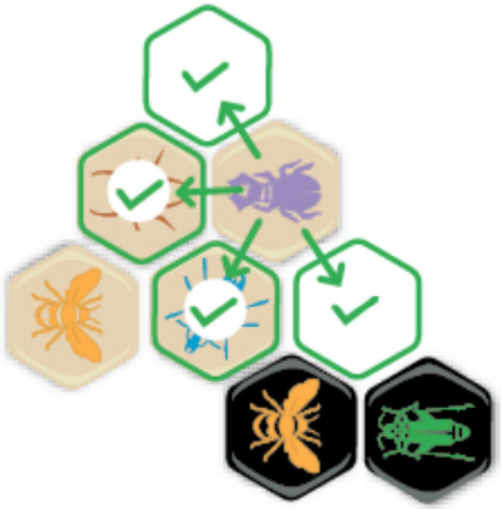


Tipi di pedine

Nel gioco base sono presenti 5 diversi tipi di insetti, ognuno con la propria strategia di movimento:

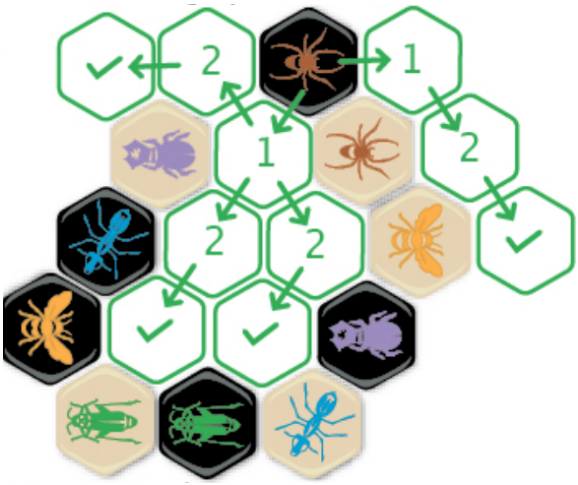
- **Ape Regina** (1 unità): è il pezzo più importante (e più debole) del gioco, paragonabile al **re degli scacchi**. La partita è vinta se l'ape regina avversaria è completamente circondata da altre pedine, indipendentemente dal loro colore. Si muove strisciando di un passo alla volta lungo l'alveare.

- **Scarabeo** (2 unità): similmente alla regina muove di un solo passo alla volta, ma può anche salire sopra altri pezzi dell'alveare. Posizionandosi sopra una qualsiasi altra pedina **ne impedisce il movimento** e contemporaneamente fa in modo che quello spazio venga considerato come se fosse del **colore della propria squadra**. Quando viene posizionato per la prima volta, lo scarabeo deve seguire le stesse regole di tutti gli altri pezzi: non può essere piazzato direttamente sopra un'altra pedina dell'alveare, anche se può esservi spostato più avanti.



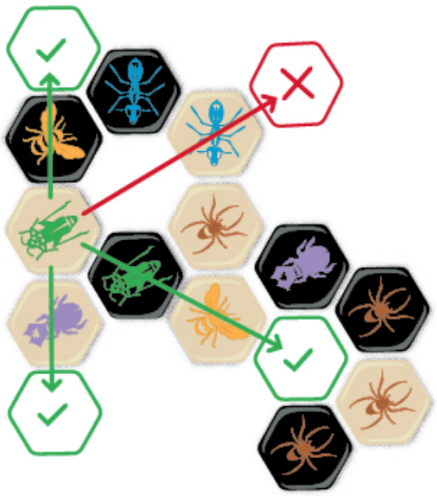
- **Ragno** (2 unità): muove esattamente di tre passi (né di più, né di meno) lungo l'alveare. Deve spostarsi lungo un percorso continuo e non può tornare sui suoi

passi nell'arco di un movimento.



- **Formica soldato** (3 unità): può muoversi ad una qualunque distanza lungo l'alveare, essendo così in grado di posizionarsi in qualunque punto che sia possibile raggiungere strisciando.

- **Cavalletta** (3 unità): a differenza degli altri insetti, non si muove strisciando lungo il bordo esterno dell'alveare, bensì **salta** dallo spazio su cui si trova oltre un numero qualunque di pezzi (almeno uno) in linea retta e arriva nel primo spazio vuoto che incontra lungo una **fila ininterrotta di pezzi contigui**. Questo le conferisce la capacità di posizionarsi in uno spazio vuoto circondato da altri pezzi.



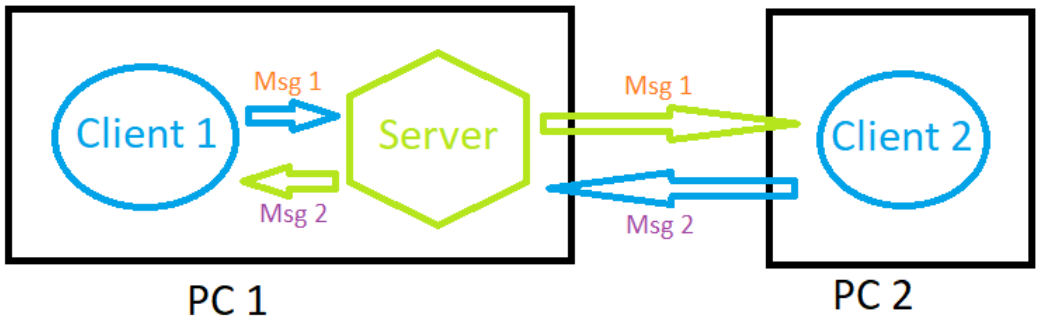
Requisiti

Di seguito verrà specificato in che modo il progetto **soddisfa tutti i requisiti** richiesti.

Architettura Client-Server

Quando si vuole iniziare una partita uno dei due giocatori deve agire da host, avviando un server sulla propria macchina.
Sia la parte **client** che la parte **server** del gioco sono accessibili attraverso un'unica applicazione, che presenta nel menù principale la possibilità di scegliere se creare il proprio server e attendere l'arrivo di un altro giocatore, oppure collegarsi come client a una partita hostata da qualcun altro.
Se si decide di hostare una partita bisogna dichiarare la porta attraverso la quale offrire il servizio di gioco; dopo aver confermato la propria scelta l'applicazione avvierà due **thread**: il primo fungerà da **server** per tutte le persone che si collegheranno a questa partita, mentre il secondo si comporterà da **client** per il giocatore che fa da host e si collegherà in automatico al server.
A questo punto un secondo giocatore potrà collegarsi, usando esclusivamente la parte client della propria applicazione, al server messo a disposizione dal primo giocatore, dando inizio alla partita. Per fare ciò, bisogna dichiarare sia l'indirizzo IP che la porta del server con il quale si desidera instaurare la comunicazione in un apposito menù e, dopo aver confermato i dati, attendere l'avvio del collegamento.

Lo schema di funzionamento è quindi il seguente:



I client gestiscono localmente la visualizzazione grafica del gioco e la selezione delle mosse, mentre il server si occupa di inoltrare gli aggiornamenti ai client (es. nuova struttura dell'alveare, nuovi messaggi in chat, inizio dei turni).

Interfaccia grafica e MVC

L'interfaccia grafica è stata realizzata con **Java Swing**, purtroppo trattandosi di un framework datato non consente di impostare un'architettura **MVC** perfettamente in linea con il modello teorico, in quanto in alcuni casi non è possibile effettuare determinate operazioni grafiche senza accedere a parti di model direttamente dalla view (ad esempio quando è necessario effettuare operazioni di disegno che coinvolgono oggetti **Graphics** e **Graphics2D**) . Nonostante ciò, il programma è comunque strutturato in una parte grafica

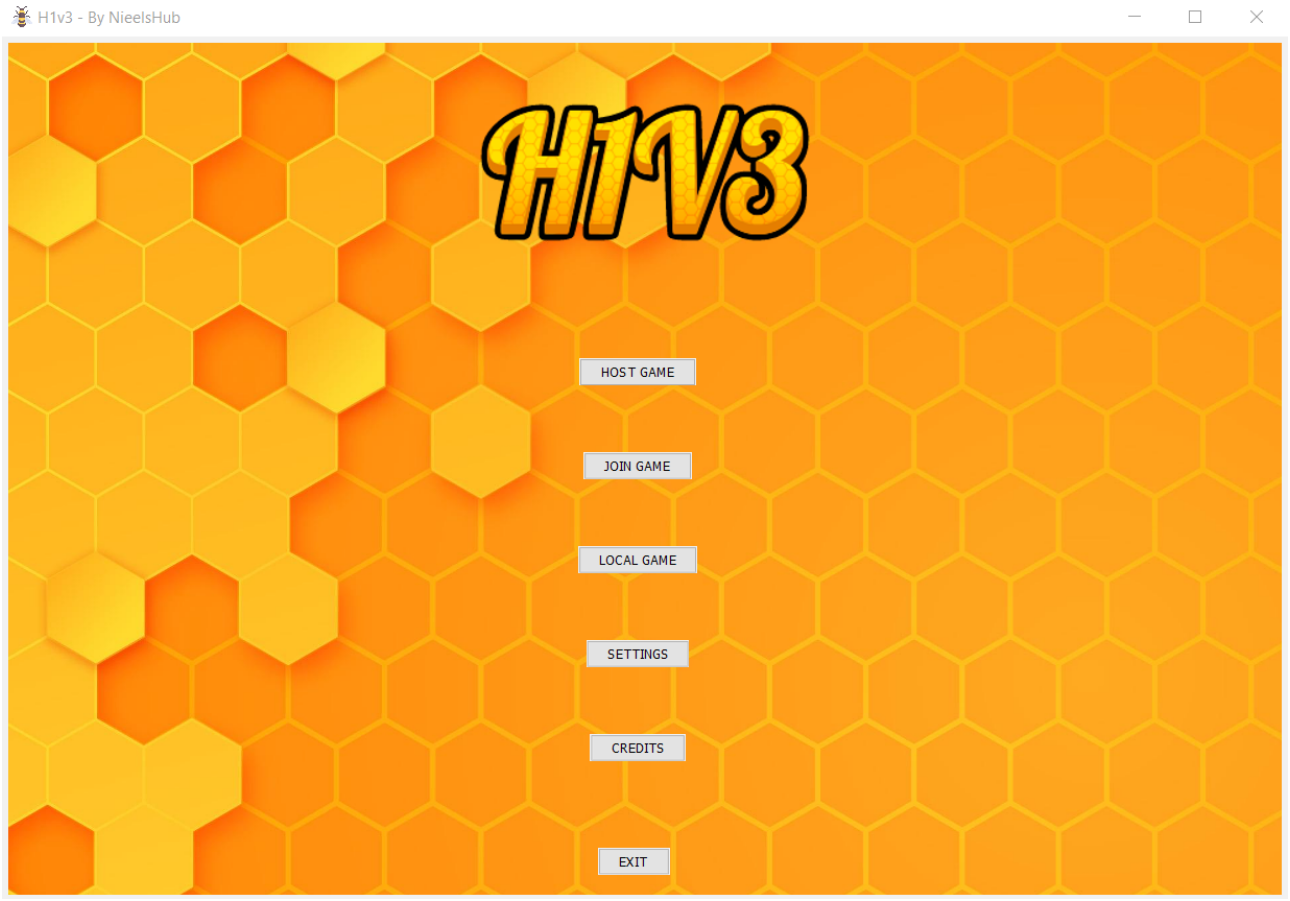
(UI) e un model (struttura di pedine e alveare, networking) distinte, la cui interazione è mediata da un controller (**HiveMain**) tramite l'uso di **eventi** e **listener**.
Sono presenti anche alcune classi di **utility**, come quella per la serializzazione dell'alveare o per la scrittura e lettura del file **XML** delle impostazioni di gioco.

Thread

Oltre a essere impiegati per la gestione parallela di server e client (e dei loro task), è stato utilizzato un **ExecutorService** anche per parallelizzare l'elaborazione di uno dei metodi più pesanti lato model: il **controllo di coesione** dell'alveare.
Esso deve essere effettuato ogni turno su ogni pezzo in gioco, per determinare se sia possibile spostarlo senza dividere l'alveare in più parti (mossa illegale).
Per controllare la coesione dell'alveare, che è stato modellizzato come un grafo in cui ogni pedina è rappresentata da un nodo e ogni collegamento tra pedine costituisce un arco, viene utilizzata una versione modificata dell'algoritmo **DFS**, che verifica se da un nodo qualunque di partenza è possibile raggiungere tutti gli altri. Data la natura dell'algoritmo, partendo da nodi diversi anche l'elaborazione del risultato avviene diversamente, per cui lanciare lo stesso task su più thread asincroni (il cui numero dipende dalla dimensione dell'alveare) partendo da nodi diversi consente di poter eseguire l'elaborazione più volte in parallelo e restituire il risultato dell'elaborazione più veloce (quella che, in base alla topologia dell'alveare, parte da un pezzo in posizione "avvantaggiata").

```java
private boolean checkHiveCohesionMultithread(int piecesInterval, ArrayList<Piece> excludedPieces) {
    ArrayList<Piece> taskStartingPieces = new ArrayList<Piece>();
    boolean result = false;
    //int numberOfTasks = placedPieces.size() / piecesInterval;

    for (int i = 0; i < placedPieces.size(); i += piecesInterval) {
        taskStartingPieces.add(placedPieces.get(i));
    }

    //System.out.println(taskStartingPieces);

    ExecutorService executor = Executors.newCachedThreadPool();

    ArrayList<Callable<Boolean>> tasks = new ArrayList<>();

    for (Piece p : taskStartingPieces) {
        tasks.add(() -> checkHiveCohesion(p, excludedPieces));
    }

    //System.out.println(tasks);

    try {
        result = executor.invokeAny(tasks);
        executor.shutdownNow();
        //System.out.println("Executor stopped " + System.currentTimeMillis());
        //System.out.printf("COHESION HAS BEEN FOUND TO BE : %b\n", result);

    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    } finally {
        //Close the executor service
        executor.shutdownNow();
    }

    //System.out.println("Method finished " + System.currentTimeMillis());
    return result;
}
```

Implementazione



H1V3 ricalca in tutto e per tutto le regole del gioco originale, suggerendo, di turno in turno, le mosse legali per ciascuna pedina, in base alla configurazione attuale del campo di gioco.
Oltre alla partita online, c'è la possibilità di giocare anche in **locale** utilizzando un unico dispositivo e alternandosi nella scelta delle mosse.

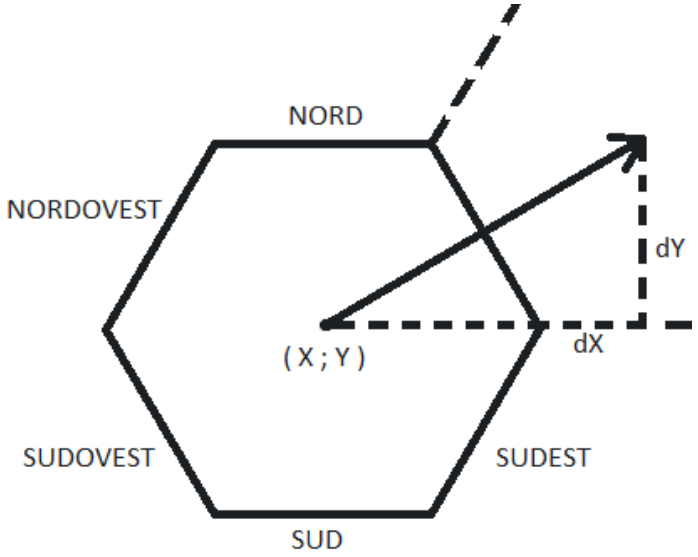
Sono state introdotte anche alcune funzionalità aggiuntive, come la **chat** online e la possibilità di variare il numero di pedine in gioco (tranne l'ape regina, che è sempre solo una) dal menù di **impostazioni**.

L'organizzazione della struttura base del gioco può essere descritta analizzando brevemente le principali classi che lo compongono.

Piece

Come accennato precedentemente, la struttura base dell'alveare è organizzata in un **grafo** i cui **nodi** sono gli insetti in gioco, collegati tra loro da diversi **archi** (al massimo uno per ogni faccia della pedina esagonale) che rappresentano i collegamenti tra i pezzi i cui lati si toccano.
Oltre a questo tipo di rappresentazione, ogni pedina contiene anche delle **coordinate cartesiane** che ne specificano la posizione sul tavolo di gioco, l'origine di tale sistema di riferimento si trova in corrispondenza del primo pezzo messo in gioco.

Questo doppio sistema consente da un lato di poter rappresentare graficamente l'alveare in maniera molto semplice, basandosi esclusivamente sulle coordinate cartesiane, e offre dall'altro la possibilità di eseguire, nel model, elaborazioni di una certa complessità, sfruttando la logica dei grafi (ad esempio per il calcolo della coesione dell'alveare o di come ciascun insetto può percorrerlo diversamente).



Le facce di una pedina vengono definite da un **Enum**, che specifica, per ogni lato, quali sono il suo precedente e successivo, nonché il dX e dY da aggiungere alle coordinate cartesiane del pezzo corrente per trovare la posizione del pezzo che si trova ed esso collegato tramite il lato (arco) in questione.
Ciascuna pedina memorizza i propri collegamenti in una **HashMap** che presenta come chiave un lato (uno dei sei possibili valori dell'**Enum**) e come valore un altro pezzo: nel complesso questo sistema garantisce l'irripetibilità dei lati e l'unicità dei possibili collegamenti lungo di essi.

Oltre alle posizioni sui lati, ciascun pezzo può anche avere una pedina che sta sopra di lui e una al di sotto (situazione che si può verificare al movimento degli scarabei).

Pur contenendo diversi metodi per la gestione delle pedine, ad esempio quello per collegare due pezzi tra loro o per controllare quali lati di una pedina risultino già occupati, **Piece** è una **classe astratta**, che rimanda l'implementazione delle logiche di movimento di ciascuna pedina alla dichiarazione di specifiche sottoclassi (una per ogni tipologia di insetto).

Piece contiene anche una **inner class Placement** che, specificato un pezzo e un suo lato, rappresenta la possibilità di poter posizionare delle pedine in quel punto.

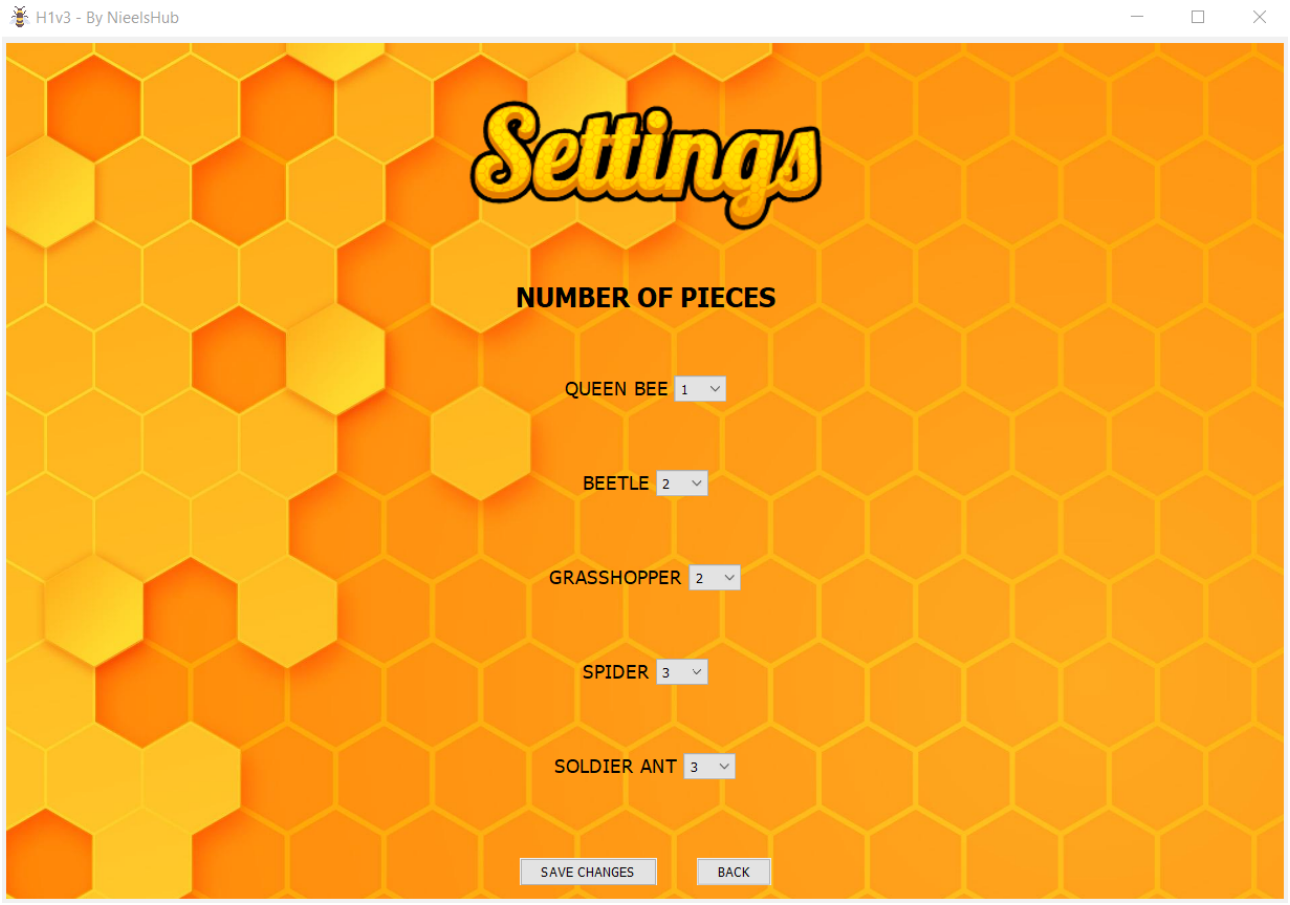
```java
//Static inner class
/**
 * Describes the possible positioning of a piece on a particular side of another piece.
 * @author Nicol Stocchetti
 *
 */
public static class Placement implements Serializable {

    private static final long serialVersionUID = 1L;

    private Piece neighbor;
    private Side positionOnNeighbor;

    /**
     * The constructor.
     * @param neighbor the piece to which the positioning is relative, Piece.
     * @param side the side of the neighbor on which to possibly position new pieces, Side.
     */
    public Placement(Piece neighbor, Side side) {
        this.neighbor = neighbor;
        this.positionOnNeighbor = side;
    }

    public Piece getNeighbor() {
        return neighbor;
    }
    public void setNeighbor(Piece neighbor) {
        this.neighbor = neighbor;
    }
    public Side getPositionOnNeighbor() {
        return positionOnNeighbor;
    }
    public void setPositionOnNeighbor(Side positionOnNeighbor) {
        this.positionOnNeighbor = positionOnNeighbor;
    }

    @Override
    public String toString() {
        return "(" + neighbor.getName() + " " + neighbor.getColor() + "-" + neighbor.getId() + " on " + positionOnNeighbor + ")";
    }

}
```

Hive

Rappresenta l'alveare, visto come l'insieme dei pezzi in gioco più quelli ancora da giocare.

```java
/**
 * The Hive's constructor.
 * @param piecesSet a map containing what kind of pieces are to be generated for the game and their number, LinkedHashMap.
 */
public Hive(LinkedHashMap <Class<?>, Integer> piecesSet) {
    Class<?> pieceKind;
    int numberOfPieces;
    int i;

    for (Entry<Class<?>, Integer> entry : piecesSet.entrySet()) {
        pieceKind = entry.getKey();
        numberOfPieces = entry.getValue();

        try {
            Constructor<?> constructor =  pieceKind.getConstructor(PieceColor.class);//PieceColor.class returns the Class
            for(i = 0; i < numberOfPieces; i++ ) {
                whitesToBePlaced.add((Piece)constructor.newInstance(PieceColor.WHITE));
                blacksToBePlaced.add((Piece)constructor.newInstance(PieceColor.BLACK));
            }
        }
        catch(NoSuchMethodException e) {
            System.err.println("The specified constructor doesn't exist!");
            e.printStackTrace();
        }
        catch(Exception e) {
            System.err.println("Unknown error.");
            e.printStackTrace();
        }
    }
}
```

Il costruttore sfrutta dei meccanismi di **reflection** per generare i pezzi di partenza, ricevendo una **HashMap** che specifica quali classi di pedine mettere in gioco e in che numero.
Tale **HashMap** viene generata nel controller a partire da un file **XML** di configurazione, che è modificabile dal menù di impostazioni del gioco.



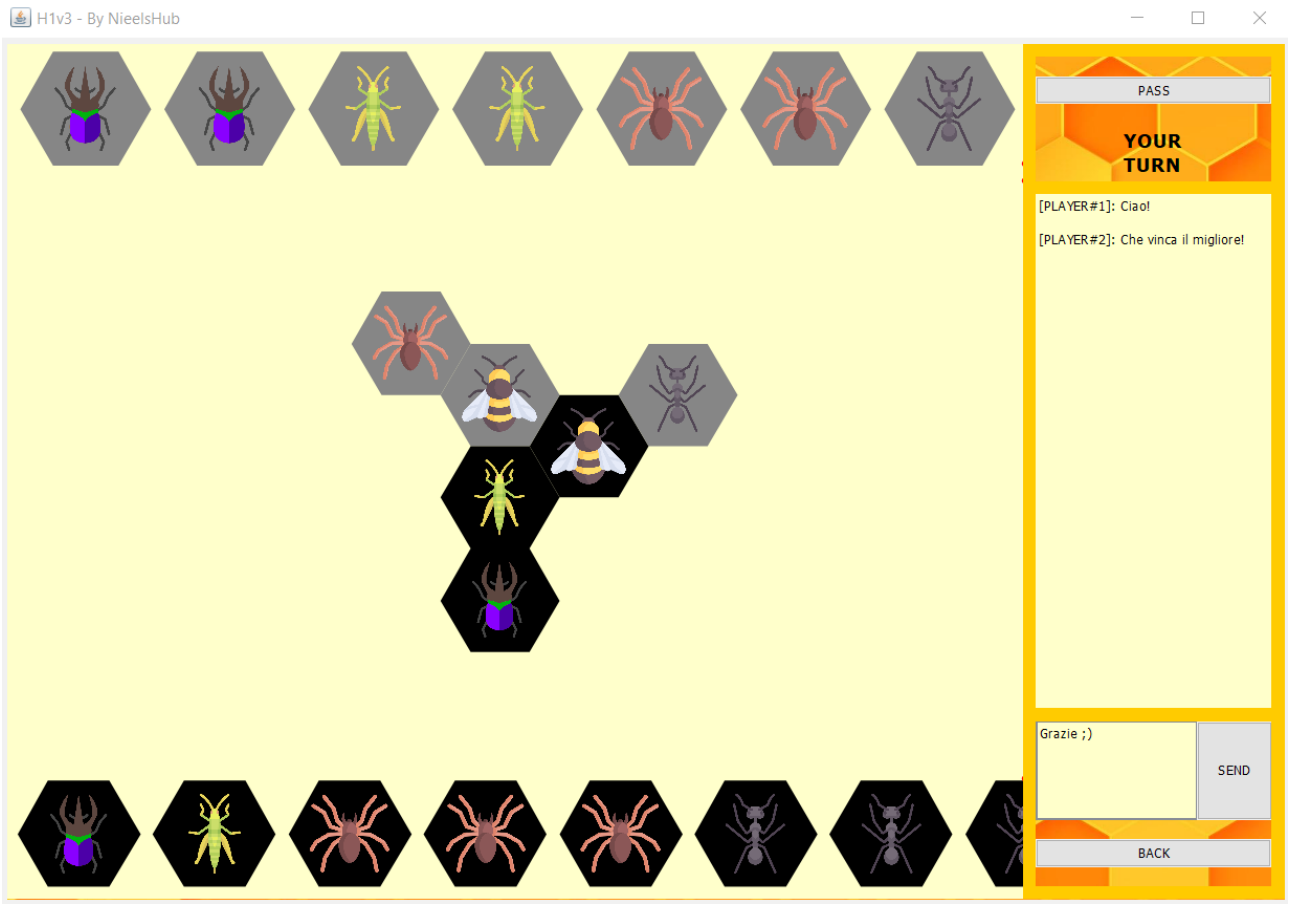
La classe **Hive** contiene tutti i metodi per l'interazione generica tra le pedine (controllo dei colori, posizionamento, spostamento, ecc.) che vengono usati nel controller per modificare il model a seguito della ricezione di un evento da parte della view.

HexField

Estensione di un **JComponent**, è la classe più importante della view, che fornisce un ambiente in grado di visualizzare i pezzi esagonali del gioco, la loro selezione e le loro possibili posizioni.
Consente anche di interagire con i pezzi tramite il mouse: quando vengono effettuati una selezione o uno spostamento validi, **HexField** lo comunica al controller attraverso degli **eventi**.
Non viene utilizzata direttamente all'interno del programma, ma è estesa da due classi figlie, **GameField** e **ToBePlacedField**, che vanno a costituire rispettivamente il campo di gioco e le due finestre delle pedine ancora da posizionare. Questa scelta è giustificata dal fatto che le due classi figlie hanno moltissimi metodi in comune, ma delle leggere differenze nella visualizzazione e gestione dei pezzi:

- **GameField** deve mostrare solo i pezzi già posizionati nell'hive e permettere di muoverli, è possibile muovere la visuale del campo di gioco in tutte le direzioni.

- **ToBePlacedField** deve mostrare solo i pezzi ancora da posizionare di uno specifico colore e permettere di selezionarli per poi essere eventualmente posizionati nell'alveare, li mostra affiancati e permette di scorrere la visuale solo in orizzontale, lungo la fila.



Complessivamente, il progetto è costituito da più di trenta classi, la cui analisi nel dettaglio rimando alla **javadoc** che accompagna i sorgenti (visionabili al link: https://github.com/NieelsHub/H1v3).

Ulteriori Sviluppi

Esistono attualmente tre espansioni del gioco base, che aggiungono tre nuovi tipi di insetti: la coccinella, la zanzara e l'onisco. Dato che l'alveare è già predisposto al caricamento dinamico delle pedine di gioco, sarebbe interessante includere la possibilità di aggiungere, tramite **reflection**, delle nuove classi che implementino il comportamento delle espansioni, o addirittura offrire la possibilità a ogni utente di creare i propri insetti custom.
A questo scopo è necessario cercare a **runtime** tutte le classi che ereditano da **Piece**, questa operazione può essere effettuata utilizzando la libreria **Reflections** (https://github.com/ronmamo/reflections), oppure **Classgraph** (https://github.com/classgraph/classgraph).

Un altro sviluppo interessante potrebbe essere quello di creare una versione del programma in cui sia possibile unirsi a una partita come spettatore, tramite un client che visualizzi la partita ma sia in grado di interagire solamente con la chat (il progetto è attualmente già predisposto in alcuni punti per inserire questa funzionalità). In questo ambito sarebbe utile anche inserire la possibilità di selezionare, dal menù opzioni, il nome utente che si vuole visualizzare in chat.