Elaborato di Progetto - Paradigmi di Programmazione e sviluppo

Relazione “Sudoku”

Lorenzo Pacini, Silvia Zandoli, Alberto Antonelli

15 Settembre 2020

Sommario

Il gruppo si è posto l'obiettivo di creare un progetto in scala che realizza il Sudoku, il classico gioco di logica. Con un'interfaccia semplice e user-friendly per l'utente, il gioco può essere accessibile facilmente.

Indice

Capitolo 1

Processo di sviluppo

1.1 Metodologia di sviluppo

Durante tutto il processo di sviluppo è stata adottata la tipologia Agile.

Il metodo Agile tenta di ridurre il rischio di fallimento sviluppando il software in finestre di tempo limitate chiamate iterazioni che, in genere, durano qualche settimana. Ogni iterazione è un piccolo progetto a sé stante e deve contenere tutto ciò che è necessario per rilasciare un piccolo incremento nelle funzionalità del software: pianificazione (*planning*), analisi dei requisiti, progettazione, implementazione, test e documentazione.

Più in dettaglio è stato seguito un approccio Scrum-like. Il ruolo di Product Owner è stato ricoperto da Lorenzo Pacini, mentre il ruolo di Scrum Master, incaricato di garantire il raggiungimento degli obiettivi fissati nello sprint, è stato ricoperto da Silvia Zandoli.

In fase di inizio del progetto e durante, è stato delineato il Product Backlog, nella quale sono contenute le principali macro attività da realizzare. Ad ognuna di esse è associata una stima in termini di durate di lavoro necessario. Ogni macro attività è stata poi suddivisa in piccoli task assegnati ai membri del gruppo.

Il lavoro è stato suddiviso in una serie di sprint settimanali contenenti i task . Ogni task è stato eseguito da un membro oppure anche come lavoro congiunto di due membri.

Al termine di ogni sprint è stata eseguita una sprint review nella quale tutti i componenti hanno condiviso il proprio lavoro. In particolare, si è discusso di eventuali problematiche che si sono riscontrate e possibili miglioramenti da apportare al progetto. Di seguito si sono concordati i nuovi task da assegnare per lo sprint successivo. Naturalmente, ad ogni task è associato un valore che stima il tempo di lavoro.

Oltre all'approccio Agile, è stata adottata un'ulteriore metodologia di sviluppo, la Test Driven Development.

Il Test Driven Development consiste nella scrittura dei test prima dell'implementazione vera e propria della funzionalità. Lo sviluppo è guidato (driven) dai test e non viceversa. Questo prevede prima di scrivere test e poi di farli passare. Dunque la scrittura del codice è orientata al superamento di test appositamente predisposti. Si verifica così molto semplicemente la correttezza del codice e la sua qualità.

Per garantire anche una migliore gestione dei task, si è creato un gruppo chat su Microsoft Teams. All'interno di esso, ogni componente, organizza meglio i suoi compiti e valuta lo stato di avanzamento generale. E' stato anche molto utile per organizzare le riunioni, pre fissandone la data e i temi da trattare. Non sempre è facile lavorare in un team composto da più persone, e Microsoft Teams può aiutare in tal senso, quantomeno a programmare le giornate di incontro.

1.2 Strumenti utilizzati

I principali strumenti utilizzati durante tutto il processo di sviluppo sono stati:

- Git, il lavoro di gruppo ha bisogno di un sistema di controllo di versione distribuito.
- GitHub, come servizio di hosting nel quale creare i repository.
- Travis CI, come servizio di continuous integration, abbinabile ai repository GitHub. La Continuous Integration viene apprezzata soprattutto nello sviluppo agile di software. L'obiettivo di questo moderno metodo è quello di suddividere il lavoro in porzioni più piccole per rendere il processo stesso di sviluppo più efficiente e poter reagire con maggiore flessibilità alle modifiche. GitHub informa Travis CI su ogni modifica attuata sul repository e garantisce il costante aggiornamento del progetto.
- build sbt, che è il sistema di build nativo per Scala. Esso è indispensabile per la gestione delle dipendenze delle librerie esterne, gestione del codice, delle dipendenze e delle risorse che devono essere costruite, testate e / o compilate, esecuzione complessiva dei test e generazione della relativa documentazione di progetto.

Capitolo 2

Requisiti

Il progetto Sudoku è nato con l'idea di sviluppare una versione in Scala del popolare gioco di logica. Esso è un gioco solitario, da risolvere con il ragionamento. Oltre all'implementazione del classico algoritmo di forza bruta sono state implementate altre tecniche e strategie di risoluzione. E' prevista la costruzione di una soluzione che sia facilmente accessibile da piattaforme differenti.

Il gioco si avvia con una iniziale interfaccia, dalla quale si può scegliere un file di Sudoku da completare, in base al livello di difficoltà prescelto.

Scegliendo un file si apre così la matrice di gioco. Una matrice di Sudoku consiste in una tabella di 81 caselle (o celle) suddivise su 9 righe, 9 colonne e 9 quadri di 3x3 caselle contigue ciascuno (detti anche blocchi o regioni).

L'obiettivo dei giocatori è quello di riempire le caselle vuote con valori da 1 a 9, rispettando la regola base (unica regola) che *"ogni riga, colonna e quadro contenga tutte le cifre da 1 a 9, senza ripetizioni"*.

La modalità di questo gioco è quindi esclusivamente single-player: l'utente gioca autonomamente; il suo obiettivo è quello di completare il sudoku e ottenere un punteggio alto nel minor tempo possibile.

2.1 Requisiti Utente

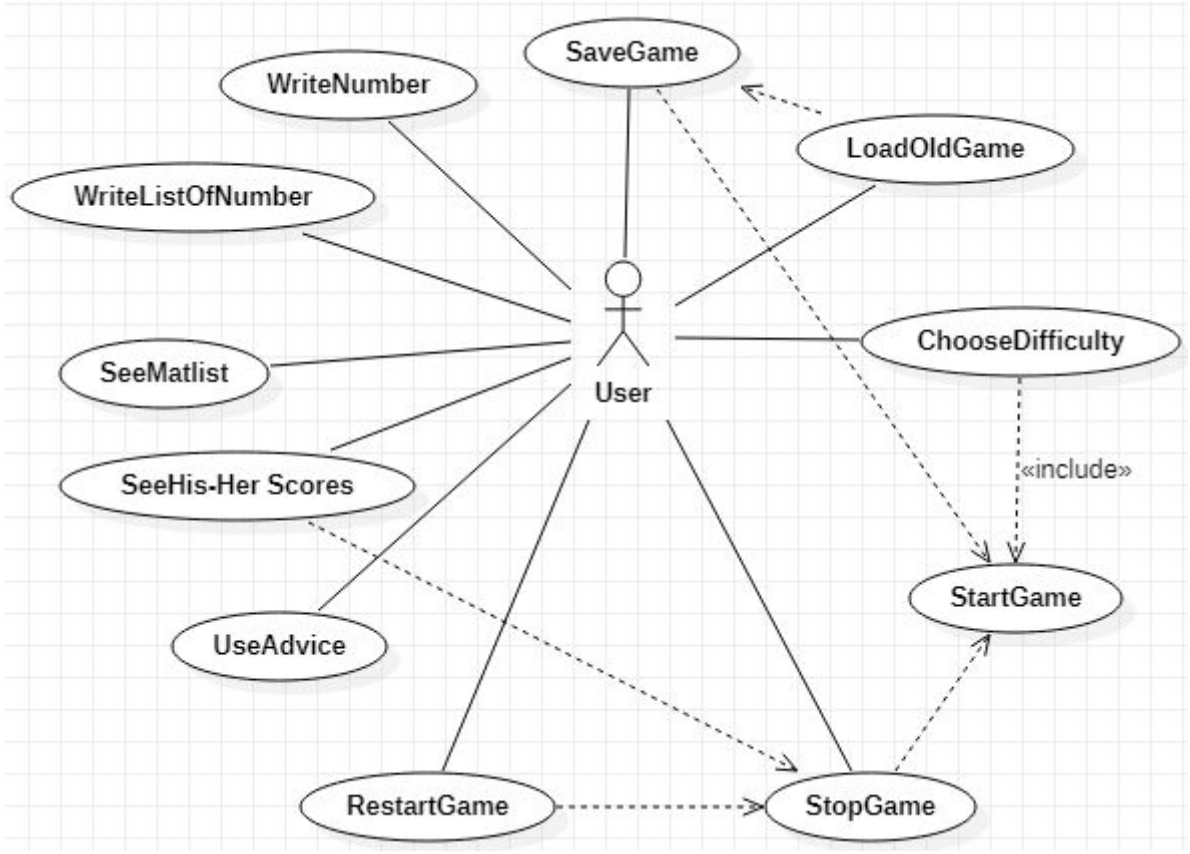
Come in tutti i giochi, l'utente è la parte essenziale da considerare per la buona riuscita del progetto.

Per quanto riguarda l'inizializzazione del gioco l'utente deve poter aprire, tramite una piccola interfaccia, un Sudoku su cui giocare, scegliendolo in base al livello di difficoltà.

Una volta aperto un Sudoku e iniziata la partita l'utente deve:

1. Poter scrivere su ciascuna cella vuota il numero che ritiene giusto.
2. Poter scrivere, nel caso non sia sicuro, un insieme di numeri su una casella lasciata libera.
3. Poter avvalersi di suggerimenti durante la partita e poter visualizzare l'insieme dei numeri possibili in una determinata casella.
4. Poter salvare il gioco e riprenderlo in un secondo momento.
5. Avere a disposizione un'interfaccia user-friendly per giocare e vedere il suo punteggio e il timer di gioco.
6. Poter stoppare una partita per visualizzare il timer e lo score e poi poterla avviarla di nuovo.

2.1.1 Figura Casi D'uso



2.2 Requisiti Funzionali

Analizzando il caso di studio, il progetto può essere suddiviso nelle seguenti parti:

- Gioco (modalità single-player)
- Interfaccia di gioco principale
- Sistema di punteggio (timer e score)
- Interfaccia di caricamento e scelta della difficoltà

2.2.1 Gioco

Per quanto riguarda il gioco, i requisiti derivano in gran parte delle regole della versione originale del gioco stesso:

- La matrice del Sudoku viene inizializzata attraverso il caricamento di uno schema da completare che contiene le caselle con i numeri già inseriti e le caselle vuote contrassegnate da degli zeri.
- Il Sudoku è formato da una matrice 2D formata da 81 caselle. Alcune caselle sono già riempite con un numero e non sono più riscrivibili, mentre nelle restanti possono essere scritti esclusivamente caratteri numerici.
- All'interno di ogni casella vuota si può inserire un numero singolo o una lista di numeri
- l'inserimento del numero o della lista di numeri in una casella vuota avviene in base alla strategia del giocatore
- il giocatore, in un certo lasso di tempo, individua il numero che vuole inserire in una determinata casella, sulla base dell'adiacenza dei caratteri presenti nella board di gioco, osservando la corrispondente riga, colonna e blocco.
- Il punteggio (score) dell'utente è dato dal numero delle caselle completate correttamente.
- Ogni partita è caratterizzata da:
    - tempo
    - score
    - possibilità di incrementare il proprio score in seguito all'inserimento di un numero corretto.
- La partita termina quando l'utente completa il Sudoku. La partita può anche essere salvata e ripresa in un secondo momento.
- Una volta completato il Sudoku si può visualizzare il quantitativo di tempo impiegato a finirlo e se ne potrà iniziare un altro.
- Per visualizzare lo score fatto in un lasso di tempo, la partita può venire stoppata.

2.2.2 Interfaccia di gioco principale

Il compito principale dell'interfaccia utente è quella di visualizzare la matrice di gioco.

Iniziata la partita, l'utente può editare e inserire dei numeri in ogni casella libera.

Dopo che l'utente ha selezionato una casella libera, con un dialog mostra le opzioni di modalità di gioco (inserimento numero, inserimento lista di numeri, visione dei numeri possibili e opzione suggerimento).

Inoltre, con apposite dialog controlla, attraverso un controllo automatico sull'inserimento dei numeri, se il numero inserito è un carattere numerico, se è corretto o se appartiene alla lista dei numeri possibili, evidenziando così eventuali errori. Perciò l'utente viene guidato durante il gioco.

A lato visualizza anche la lista di possibili numeri, il timer e lo score, e le opzioni per salvare il gioco o avviare la risoluzione automatica.

2.2.3 Sistema di Punteggio

Il sistema del punteggio si basa sull'attribuzione di un incremento di valore ogni qualvolta si inserisca un numero corretto nel Sudoku. Esso nel caso si voglia compararlo con altre risoluzioni dello stesso Sudoku deve essere correlato al tempo impiegato nel risolverlo.

Il punteggio in caso di sospensione della partita sarà salvato per una successiva risoluzione.

2.2.4 Interfaccia di caricamento e scelta della difficoltà

Prima dell'avvio della partita all'utente, attraverso apposita interfaccia iniziale, è concesso scegliere un file di gioco all'interno di una cartella. La cartella contiene diversi schemi di Sudoku da completare, suddivisi in sottocartelle in base al loro livello di difficoltà (facile, intermedio, difficile). All'utente viene anche concesso di poter ricaricare un Sudoku salvato in precedenza.

2.3 Requisiti non Funzionali

Questo gioco è esclusivamente in modalità single-player. I requisiti non funzionali richiesti sono:

- Performance ovvero il gioco deve scorrere con una certa fluidità e con una tempestiva disponibilità di elaborazione. L'interfaccia grafica dovrà essere affidabile e per quanto possibile reattiva, con un buon compromesso tra qualità e prestazione.
- Usabilità ovvero il gioco deve essere user-friendly e di facile utilizzo per l'utente.

- Reattività ovvero l'utente deve ottenere risposte dal sistema in tempi brevi rispetto ogni sua azione.

2.4 Requisiti Implementativi

I requisiti implementativi utilizzati durante lo sviluppo del progetto sono:

- Scala, come linguaggio di programmazione. *Scala* è un linguaggio che si rivolge ai bisogni principali dello sviluppatore moderno. È un linguaggio per la jvm staticamente tipato, a paradigma misto, con una sintassi concisa, elegante e flessibile, un sistema di tipi sofisticato e idiomi che promuovono la scalabilità dai piccoli programmi interpretati fino ad applicazioni sofisticate di grandi dimensioni. Attraverso questo linguaggio è quindi possibile sfruttare al massimo le caratteristiche dei paradigmi object-oriented e funzionale.
- Javax, il toolkit grafico di Java, per la realizzazione dell'interfaccia grafica
- Prolog, come linguaggio di programmazione che adotta la programmazione logica. In questo progetto viene usato solo relativamente alla parte di realizzazione di un algoritmo brute-force del Sudoku con questo linguaggio
- Jar, ovvero un file che contiene tutto il necessario per avviare facilmente il gioco.

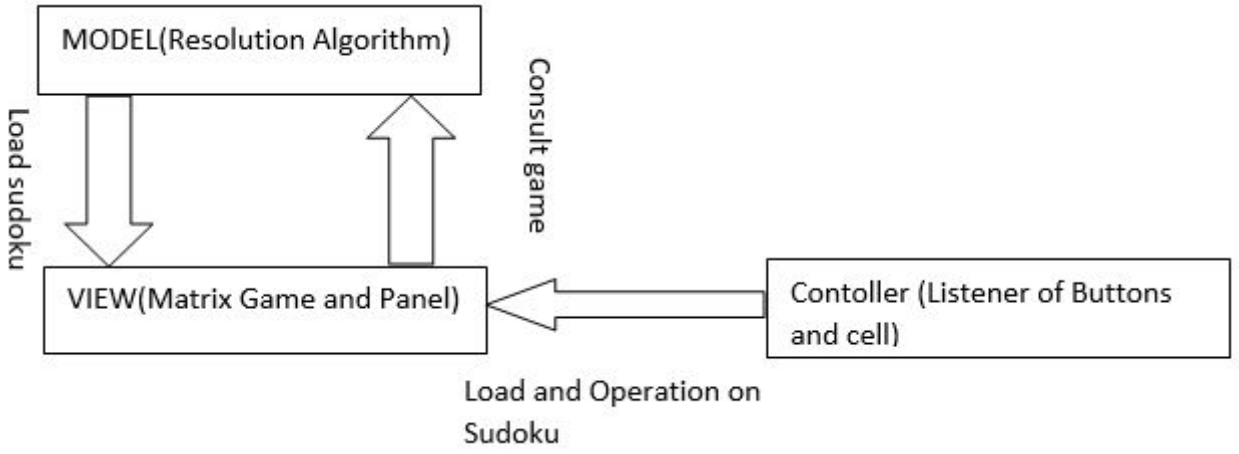
Capitolo 3

Design Architetturale

Nella definizione ad alto livello dell'architettura si è sfruttato il modello MVC in grado di sfruttare appieno il paradigma a oggetti. Sulla base di questo approccio, l'intera logica del sistema è modellata sfruttando il classico paradigma object oriented.

3.1 Componenti del sistema

Una visione dell'architettura ad alto livello è presente nella figura sottostante. In questa sono infatti visibili i macro componenti della progettazione dell'applicativo.



L'architettura proposta presenta lo schema MVC, dove la parte del Model è costituita dagli algoritmi di risoluzione del sudoku, la parte della View è costituita dalla grafica e la parte di Controller sono i listener ed i menù di caricamento per capire le azioni che l'utente andrà a fare.

3.1.1 Game Model

Il componente essenziale dell'architettura è certamente il Game Model. Come precedentemente descritto, al suo interno è presente tutta la logica di base dell'applicativo.

In particolare, ricopre un ruolo assolutamente centrale la classe Full Exploration. Tale classe è il vero motore di risoluzione, perché l'algoritmo di risoluzione del puzzle, (solve), è di forza bruta ed è basato su ricorsione con backtracking per ciascuna casella non occupata da numeri.

Tra gli algoritmi di risoluzione, oltre al classico algoritmo di forza bruta, sono implementati altri algoritmi e strategie di astuzia. Si veda il Capitolo 4 per una spiegazione più dettagliata.

Tutte le strategie di risoluzione vengono richiamate, fintanto che il Sudoku non viene risolto, da un “Engine”.

Prima di richiamare tali strategie di risoluzione viene riempita la MatList. La MatList è la matrice dei possibili numeri per ogni casella.

Inoltre, ancora prima, viene caricato un puzzle di Sudoku, su cui poi verranno eseguite tutte le tecniche di risoluzione.

3.1.2 Parte View/Controller

La view è stata realizzata con una matrice di JTextArea con captazione delle intenzioni dell’utente, ovvero:

1. Scrivere il numero sulla cella
2. Scrivere una serie di numeri su una cella
3. Visualizzare l’insieme dei possibili numeri su una cella
4. Avvalersi del suggerimento

A fianco a questa matrice è stato realizzato un pannello di comodo per permettere all’utente di salvare il gioco corrente, mettere in pausa il gioco, visualizzare tempo e punteggio di gioco.

La parte del controller è stata realizzata con Listener per captare le intenzioni dell’utente, ovvero quelle sopra citate per la parte di gioco e per poter scegliere il caricamento del gioco (gioco nuovo o gioco precedentemente salvato) mediante menù a tendina sull’interfaccia di caricamento.

3.2 Tecnologie

Le tecnologie usate sono state:

1. Java-Awt
    a. La Abstract Window Toolkit (AWT) è la libreria Java contenente le classi e le interfacce fondamentali per il rendering grafico e la gestione di eventi e listeners. Queste classi consentono di realizzare interfacce utente complesse e di definire l'interazione attraverso la specifica di elementi e di gestori degli stessi.
2. JavaX-Swing
    a. Swing è un framework per Java orientato allo sviluppo di interfacce grafiche. Parte delle classi del framework Swing sono implementazioni di widget (oggetti grafici) come caselle di testo, pulsanti, pannelli e tabelle.Swing è una libreria indipendente dalla piattaforma, sia in termini di linguaggio, sia in termini della sua implementazione. E’ estendibile, orientata ai componenti, e realizza bene un framework GUI MVC.
3. TuProlog

a. TuProlog è una delle più famose versioni JVM-based dell'engine Prolog. Permette di sfruttare il paradigma di progettazione logica, che si presta bene alla risoluzione di problemi di ricerca all'interno di un particolare spazio. Nel progetto viene utilizzato per la risoluzione di un algoritmo di cui se ne parlerà in seguito, (con anche test per verificarne la correttezza).

Capitolo 4

Design di Dettaglio

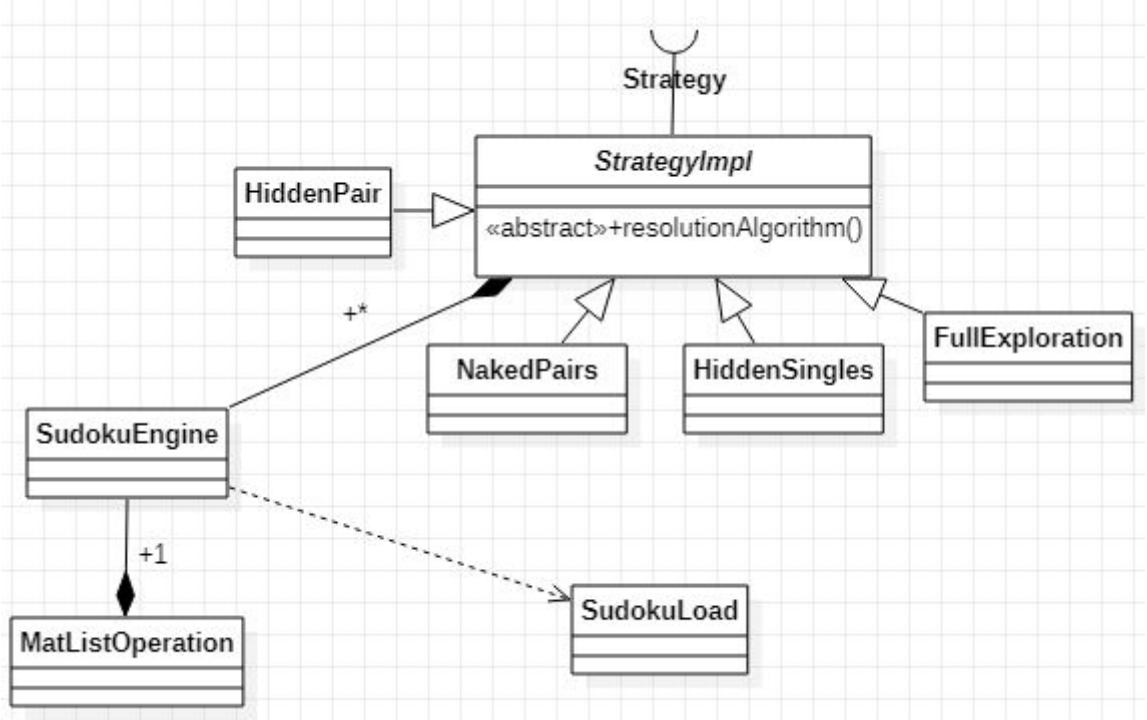
In questo capitolo sono illustrate le scelte effettuate a livello di design di dettaglio. L'intera applicazione viene suddivisa in moduli e per ognuno di essi è fornita una spiegazione dettagliata rispetto le scelte effettuate ed i pattern utilizzati.

4.1 Organizzazione del codice

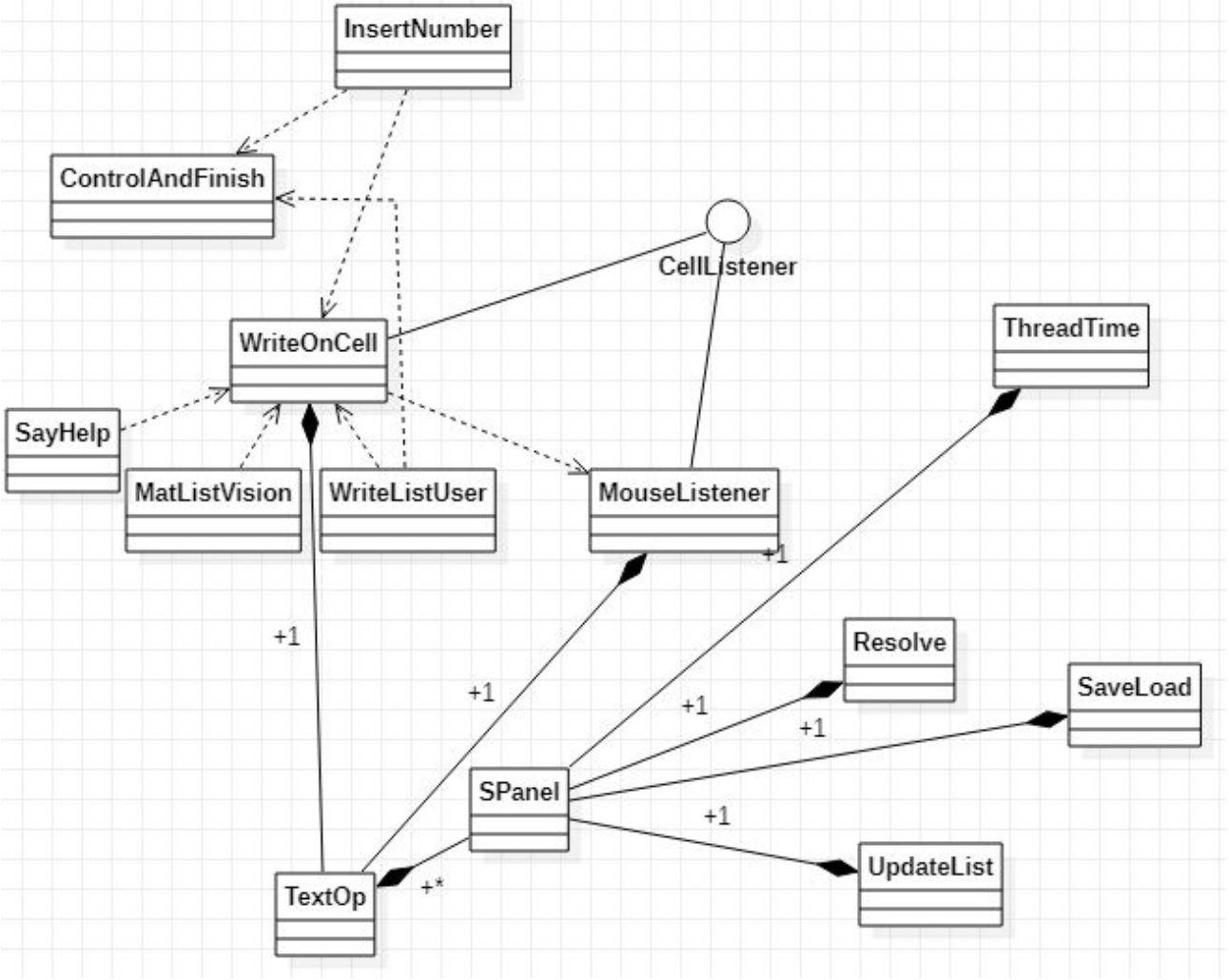
Il codice è strutturato nei seguenti package:

- grafic, che contiene tutta la parte di View e Controller
- resolutionAlgorithm
- strategies
- sudoku
- util
- prolog, presente nella directory principale del progetto
- test

4.1.1. Figura in dettaglio parte Model



4.1.2. Figura in dettaglio parte View e Controller



4.2 Pattern ricorrenti e aspetti di design avanzato

Durante l'intero processo di sviluppo e integrazione sono stati inclusi diversi pattern e aspetti di design avanzato.

Un pattern importante che è stato utilizzato nella classe Model è il pattern Strategy relativamente al metodo resolutionMethod, che in sudokuEngine viene istanziato con i metodi più appropriati delle varie strategie.

Un'altro pattern utilizzato nella parte grafica è stato il Template Method.

Questo pattern è stato usato in corrispondenza delle interfacce:

1. MyMouseListener (Estende l'interfaccia MouseListener per catturare l'evento di rilascio del mouse, al termine del quale l'utente può scegliere l'operazione da fare)
2. WriteOnCell (Estende l'interfaccia KeyListener per catturare l'evento di rilascio di un carattere dalla tastiera, a questo punto possono partire i controlli su ciò che l'utente ha scritto)

3.

con l'interfaccia CellListener, dal momento che entrambe le precedenti interfacce estendono l'interfaccia principale e implementano il metodo ActionListener.

```
trait CellListener {
  val row: Int
  val col: Int

  protected def actionCell(t: TextOpNumber): Int
}
```

(Template method)

Tra altri aspetti di dettaglio avanzati che sono stati utilizzati ricordiamo gli impliciti, l'ereditarietà multipla, la ricorsione in coda, le closure, le funzioni parametrizzate con tipo generico e singleton, il meccanismo del for - yield per computazioni ritardate.

4.3 Grafic

La grafica è stata realizzata mediante i seguenti package:

1. grafic/panels
2. grafic/panels/funAux
3. grafic/util

In particolare del package grafic/panels è molto importante citare le seguenti classi:

A. TextOpNumber (Estende da una JTextArea per manipolare le operazioni di aggiunta, rimozione e visione dei numeri mancanti sulle celle lasciate libere)
B. SPanel (Estende un JPanel e contiene tutti i pulsanti utili all'utente per iniziare/stoppare, chiamare la risoluzione e salvare lo stato del gioco)

Altre classi/oggetti degni di nota del package grafic/panels/funAux sono:

A. ThreadTime (Estende un Thread e permette di vedere lo scorrere del tempo su un JTextField)
B. SaveLoad (Modulo per permettere all'utente di salvare lo stato corrente del gioco:
    1. Matrice con caselle libere/occupate;
    2. Punteggio;
    3. Tempo)

Nel package grafic/util sono molto importanti i seguenti moduli:

I. AssociateListener (Permette di associare i listener per l'operazione da fare e per l'inserimento del numero una volta stabilita l'operazione da fare)
II. CreateMatrix (Crea la matrice di gioco, istanziando le celle e definendo i bordi di confine rossi (quelli che delimitano i sottoquadrati)
III. FileWork (Permette di cancellare e creare il file di salvataggio dello stato di gioco)

4.4 ResolutionAlgorithm

Il package resolutionAlgorithm contiene le varie strategie di Sudoku implementate.

Viene implementato il classico algoritmo di forza bruta del Sudoku. Per risolvere il puzzle del Sudoku si utilizza un algoritmo di backtracking (una visita in profondità di ricerca).

Vengono implementate anche tecniche più intelligenti.

L'algoritmo Hidden Pair si basa sulla presenza delle "coppie nascoste" o "hidden pairs" all'interno del Sudoku. Una coppia nascosta si verifica quando una coppia di numeri appare esattamente in due celle in una riga, colonna o blocco, ma questi due numeri non sono gli unici nelle loro celle. Una volta rivelata la coppia di numeri candidati tutti gli altri numeri presenti nelle due celle vengono cancellati, lasciando spazio solo alla coppia di "hidden pairs". Questa tecnica è molto utilizzata, e anche se non è in grado di arrivare da sola alla soluzione finale ne velocizza comunque il processo.

L'algoritmo Hidden Singles o "delle zone proibite" è il metodo più utilizzato quando si tenta di risolvere un sudoku, nonostante non sia in grado da solo di portare alla risoluzione ma solo velocizzarne il processo. Questa tecnica è basata sulla posizione di un dato numero nei diversi riquadri. È possibile infatti che, esaminando un gruppo di tre riquadri adiacenti, un dato numero compaia già in due. Nel terzo, quindi, sarà più semplice determinarne la posizione.

L'algoritmo Naked Pairs si basa sulla presenza di "coppie nude" o "naked pairs" all'interno del sudoku. Una coppia di questo tipo si verifica quando due numeri sono gli unici candidati in due caselle della stessa riga, colonna o blocco. In questo caso si possono escludere i due numeri dalle altre caselle di quella riga, colonna o blocco. In poche parole una colonna ha due caselle con la stessa coppia di candidati che però appaiono in altre sue caselle. Anche questa tecnica non è in grado di risolvere da sola tutto il Sudoku, ma comunque ne velocizza il processo.

4.5 Strategy

Questo package contiene l'interfaccia Strategy che permette di verificare quanto tempo è stato impiegato da una particolare strategia del sudoku. In questo progetto ricordiamo sono state implementate 3 strategie di intelligenza risolutiva (HiddenPair, NakedPair, HiddenSingles) e la strategia di forza bruta.

Come si vede dalla figura, si ricorda che qui è stato implementato il pattern Strategy.

```
/*Strategy 2 made by Lorenzo Pacini*/
val strat2 = new StrategyImpl {override def resolutionMethod(): Unit = totalHiddenSingles()}
```

4.6 Sudoku

Questo package contiene l'Engine che richiama tutti gli algoritmi di risoluzione e in più contiene anche ciò che permette di gestire la matrice MatList. Essa è la matrice dei possibili numeri, con le funzioni per inizializzarla, aggiornarla e inserimento dei numeri possibili in ogni casella.

Contiene anche una classe di utilità che gestisce la lettura di un file .txt , di farne il parsing e di caricarlo su un puzzle.

4.7 Util

Questo package contiene delle funzioni di utilità che verranno utilizzate altrove; come per esempio:

1. funzione che restituisce il puzzle di gioco
2. funzione che confronta 2 matrici con tipo generico
3. funzione che verifica se in una lista è presente un item (con uso della funzione foldLeft)
4. funzione che stampa il contenuto di una lista di una cella, avvalendosi di una funzione ricorsiva in coda e con passaggio della primitiva print (pattern higher-order)

4.8 Prolog

Il package Prolog si trova nella directory principale del progetto e contiene un algoritmo di risoluzione scritto con linguaggio Prolog.

Esso è basato sulla forza bruta.

Inizialmente è stato preso spunto da un repository per quanto riguarda lo schema principale.

Successivamente tutta la funzione validate è stata modificata ed è stata implementata basandosi sul concetto di ricorsione sulla lista secondo programmazione logica.

Il metodo allDifferent è stato implementato in questo modo:

Il caso base, (assioma) è basato sul fatto che un numero confrontato con una lista vuota è per forza diverso dalla lista vuota.

Per il passo di ricorsione, (frase del programma), ci si è basati sul fatto che si poteva proseguire sul resto della lista soltanto se il numero considerato è diverso dal numero in testa alla lista.

Più in dettaglio, allDifferent controlla che il primo elemento sia diverso dal resto all'interno di una lista.

E' stato poi implementato un altro metodo ricorsivo, different2 che richiama allDifferent su ogni possibile sottolista, scorrendo la lista iniziale.

```
% Determine if each square in either a row, column, or block are
% different

    validate(A, B, C, D, E, F, G, H, I) :- num(A), num(B), num(C), num(D), num(E), num(F), num(G), num(H), num(I),different2([A, B, C, D, E, F, G, H, I]).

        different2([]).
        different2([H|T]) :- allDifferent(H, T), different2(T).


allDifferent(A, []).
allDifferent(A, [B|T]) :- A\=B, allDifferent(A, T).
```

4.9 Test

In questo package sono riportati tutti i testi automatici svolti. La stesura di questi test è avvenuta prima di quella del software dell'applicativo. Lo sviluppo del software applicativo è stato orientato all'obiettivo di passare i test automatici precedentemente predisposti. Sono stati svolti dei test alle principali funzionalità del software e alle parti più critiche. E' stato svolto un test per ogni algoritmo di risoluzione, un test per la parte di grafica e un test generale.

Da notare che è stato fatto anche un test per l'algoritmo in Prolog. Attraverso l'engine del prolog, la teoria del programma logico è stata incorporata dentro il programma Scala.

Qui in basso alcuni esempi di test:

```
class testFull extends FunSuite {
  test( testName = "TestSudoku11") {
     ...
    assert(confrontPuzzle(solver.returnPuzzle(), getPuzzle))
  }
```

(TestFullExploration)

```
class testHiddenSingles extends FunSuite {

  test( testName = "Sudoku01") {
    ...
    assert(sudokuInput(6)(2) == sudokuSolved(6)(2))
  }
```

(HiddenSingles)

```
class testComplete extends FlatSpec with BeforeAndAfter {
  ...
 after {
   ...
 }
```

(Test con beforeAndAfter)

```
class sudokuPrologTest1 extends FunSuite {
    ...
   assert(solution == "sudoku(1,9,6,4,7,8,3,2,5," +
     "3,8,5,2,6,1,7,4,9," +
      ...
```

(Test per prolog)

Capitolo 5

Implementazione

All'interno di questo capitolo vengono analizzate e descritte in breve le parti a cui ognuno dei componenti del gruppo ha contribuito. Se necessario, alcune di esse verranno viste più in dettaglio.

5.1 Lorenzo Pacini

Mi sono occupato principalmente di implementare:

1. Algoritmo di risoluzione con forza bruta del sudoku in collaborazione con la studentessa Zandoli, (è stato inizialmente preso da un sorgente l'algoritmo per risolvere il sudoku di brute force e successivamente ne è stata migliorata l'implementazione rendendolo più scala-like con uso di funzioni quali filter, forEach e forall)
2. Algoritmo HiddenSingles che trova il numero su una cella libera basandosi su numeri uguali ad esso che sbarrano le altre celle libere a lui adiacenti rispettivamente in: riga, colonna e sottoquadrato.
   Questo algoritmo ha 2 modalità di funzionamento: su una specifica cella oppure su tutta la matrice, molto importante è stato l'uso della funzione flatMap per raggruppare le matlist delle celle di un sottoquadrato.
3. Interfaccia Strategy (precedentemente citata nel capitolo 4.5)
4. Sudoku Engine, che permette di vedere il metodo di funzionamento della combinazione degli algoritmi di intelligenza con l'algoritmo di brute force
5. MatListOperation, modulo implementato con Zandoli per fare operazioni sia sulle liste della matrice, sia sulla matrice di gioco.
   In particolare sono stati implementati i metodi per trovare la lista con elementi minimi, se trovata e contenente solo un numero la si svuotava e sostituiva la cella libera corrispondente con il numero rimasto nella matrice di gioco.
   In particolare è da citare l'uso del costrutto for-yield su updateList e minList.
6. Ho implementato il modulo SudokuLoad per caricare un file testuale, leggerne le righe e parsare ciascun carattere letto in un numero, (notare la chiusura della funzione closurePuzzle nella funzione parsePuzzle).
7. Object Sudoku per creare matrice di gioco e pannello per la grafica (notare il companion - object)
8. Modulo AssociateListener che contiene funzionalità per associare i listener alle celle e scrivere su celle libere.
9. Modulo CreateMatrix, precedentemente descritto nel capitolo 4.3
10. Modulo FileWork precedentemente descritto in 4.3 (notare l'uso del meccanismo implicit)
11. Package Object grafic che contiene funzioni utili di set e get (notare l'uso del meccanismo degli implicit per consentire di usare la funzione con il tipo di dato più appropriato)
12. TextOpNumber (vedere 4.3)
13. SPanel insieme a Antonelli (vedere 4.3)
14. Object AuxFunPanel per funzioni di inizio e stop del gioco

15. Object MakePanelGrafic
16. Object ThreadTime (vedere 4.3)
17. Object SaveLoad con Zandoli (vedere 4.3)
18. Interfaccia WriteOnCell con la relativa implementazione (vedere 4.2)
19. Interfaccia MyMouseListener con implementazione (vedere 4.2)
20. Interfaccia CellListener vedere 4.2
21. Object WriteListUser per scrivere e visualizzare i numeri scritti su una matlist di una cella
22. Object InsertNumber per scrivere il numero nelle celle libere della matrice, da notare il pattern matching in writeNumber.
23. Object UpdateListUser per l'aggiornamento delle liste e la loro visione.
24. Implementazione assieme a Zandoli del motore di risoluzione con forza bruta del sudoku in Prolog (è stato preso inizialmente un codice sorgente per l'implementazione del sudoku, ho fatto un primo passo di miglioramento della funzione validate e poi l'ultimo passo è stato fatto dalla studentessa Zandoli)

5.1.2

Qui di seguito alcune delle principali tecniche di buona programmazione:

```
@tailrec
final def computeOnList[T](f: T => Unit, list: List[T]): Unit = list match {
  case h :: t => f(h); computeOnList(f, t)
  case _ =>
}
```

(Ricorsione in coda)

```
for {i <- 0 until dimSudoku; j <- 0 until dimSudoku
    if matList(i)(j) != Nil && matList(i)(j).length < minLength
} yield {
  minLength = matList(i)(j).length
  ijMin = (i, j)
}
```

(For - yield)

```
def searchInSquare(row: Int, col: Int, num: Int): Boolean = {
  val ci = row / 3
  val cj = col / 3
  val squareCells = matList.grouped(3).toList(ci).flatMap { x => x.grouped(3).toList(cj) }

  squareCells.filterNot(list => list == Nil).count(list => list.contains(num)) == 1
}
```

(Flat - map)

```
def closurePuzzle(ch: Char): Unit = {
  puzzle(row)(col) = ch.asDigit
  col+=1
}
```

(Closure)

```
def graficSet [T: setGet](elem : T): Unit = {implicitly[setGet[T]].set(elem)}
def graficGet [T: setGet] : T = {implicitly[setGet[T]].get}

object setGet {
  implicit object stringSetGet extends setGet[String] {
    override def set(str: String): Unit = {write = str}
    override def get: String = write
  }
```

(Implicit)

```
/**
 * Made By Pacini (Alert Dialog made by Zandoli, n match made by Pacini)
 */
sealed trait MyMouseListener extends CellListener with MouseListener {
```

(Ereditarietà multipla)

5.2 Silvia Zandoli

La studentessa si è occupata principalmente di:

- Algoritmo di risoluzione Hidden Pairs. Si è occupata interamente di questo algoritmo, in seguito vengono fornite alcune spiegazioni un po' più dettagliate.
- Algoritmo di risoluzione FullExploration con Lorenzo Pacini: in particolare si è occupata dell'implementazione del metodo validate che controlla che un numero sia presente solo una volta per ogni riga, colonna o blocco. E' stato usato un approccio molto conciso e scala-like con l'uso di collezioni.
- Object MatListOperation con Lorenzo Pacini: essa serve per inizializzare, riempire o aggiornare la matrice (MatList) che contiene i possibili numeri per ogni casella. In particolare si è occupata dell'implementazione del metodo initList e del metodo possible. Quest'ultimo usa molto le collezioni, in particolare i set, e inserisce in ogni cella della MatList i numeri possibili. In particolare vengono esclusi i numeri per ogni riga, per ogni colonna e ogni blocco e con l'unione dei numeri esclusi per riga, colonna e blocco si ottiene il set di numeri da escludere. Facendo la differenza su questo set si ottengono i numeri da inserire nella corrispondente casella di MatList.
- Implementazione del controllo automatico (verifica del gioco) in cui vengono visualizzati gli errori commessi. Viene evidenziato se il numero inserito nella casella è corretto oppure no, vengono evidenziati eventuali errori durante l'inserimento delle liste di numeri e si controlla che vengano inseriti caratteri numerici. Tali controlli sono stati inseriti in InsertNumber, WriteOnCell, ControlAndFinish.

- preparare i vari schemi Sudoku e classificarli in base al livello di difficoltà (facile, intermedio, difficile).
- gestione del salvataggio e successivo caricamento di un Sudoku (Object SaveLoad). In particolare si è occupata del salvataggio del timer e dello score in appositi file e dell'implementazione del metodo read che permette di ripristinare i vecchi valori di tempo e di punteggio quando si carica un Sudoku precedentemente salvato. Per la lettura del tempo e del punteggio è stato usata la "Using", un'utility di Scala per la gestione di risorse. In questo caso "Using" gestisce l'apertura della risorsa (il File), l'operazione di lettura e la sua chiusura. Il contenuto letto verrà salvato su una Stringa per poi essere estrapolato.
- implementazione dell'interfaccia iniziale (object FileChooserMain) che permette il caricamento dei file del Sudoku (parte grafica e parte Action Listeners)
- object FileChooser che permette di navigare il file system e di selezionare uno o più file. In seguito vengono fornite spiegazioni un po' più dettagliate di questi due punti.
- gestione fine gioco (in ControlAndFinish) e pannello di scelta modalità che compare ogni volta che si clicca su una casella (in MyMouseListener)
- Costruzione dei relativi test rispetto tutto ciò che è stato direttamente sviluppato.
- In collaborazione con Lorenzo Pacini si è occupata dell'implementazione dell'algoritmo di forza bruta del Sudoku in **Prolog**, con relativo Engine e Test. Inoltre è riuscita ad arrivare all'implementazione finale e funzionante del metodo validate con i metodi AllDifferent e different2.

In linea si massima, nell'implementazione di tutti questi elementi, si è cercato di adottare un approccio funzionale alla risoluzione del problema sfruttando molti costrutti avanzati messi a disposizione da Scala. Si è fatto un largo uso di tecniche quali gli impliciti e le collezioni, cercando di utilizzare un linguaggio che sia il più conciso possibile e scala-like.

5.1.1 Hidden Pair

In questa implementazione è stata sfruttata al massimo tutta la concisione che il linguaggio Scala può offrire. Sono state utilizzate in larga scala le collezioni (List, Set, Map, etc) che hanno ottimizzato notevolmente il codice, e non è stato inserito nemmeno un ciclo for. E' stato inserito solo un for-yield.
Nella figura sotto è riportato solo un pezzo di codice, si tratta del metodo check che trova le coppie Hidden Pair e in quali celle si trovano. Da notare la potenza del linguaggio usato:

```
def check(ml: List[(List[Int], Int)]): List[PossiblePair] = {
  //possiblePairs contains all the possibles pairs with their coordinates
  val possiblePairs = ml.map(couples).toSet.subsets(2).toList.map(e => {
    //the first
    val val1 = e.head //(List[Set[Int]],Int)
    //the second
    val val2 = e.tail.head
    //intersection are the pairs which are present in both
    val intersection = val1._1.toSet.intersect(val2._1.toSet)
    PossiblePair(val1._2, val2._2, intersection)
  }).filter(_.intersection.nonEmpty) //those which have intersection nonEmpty could be the possible HiddenPairs
  val flattenPairs = possiblePairs.flatMap(p => {
    p.intersection.map(pair => {
      PossiblePair(p.cell1, p.cell2, Set(pair))
    })
  })
  //it finds the hidden pairs
  val hiddenPairs = flattenPairs.filter(p => {
    val cell1 = p.cell1
    val cell2 = p.cell2
    //on matList I remove elems of cell1 and cell2 and I leave other elements
    val exclusion = ml.filter(_._2 != cell1).filter(_._2 != cell2).flatMap(e => e._1.toSet).toSet
    //we have to compare inside intersection elem per elem. We get the first
    //If exclusion contains the first pair
    val c1 = exclusion.contains(p.intersection.head.head)
    //we have to compare inside intersection elem per elem. We get the second. tail contains more element so we get the head
    //If exclusion contains the second pair
    val c2 = exclusion.contains(p.intersection.head.tail.head)
    //if both the possible pairs aren't in exclusion they are an HIDDEN PAIR
    !c1 && !c2
  })
  hiddenPairs
}
```

Tale metodo sarà poi richiamato per ogni riga, poi per ogni colonna e poi per ogni blocco per trovare le corrispondenti coppie nascoste e in quali celle si trovano. Infine si elimineranno i numeri che nelle due corrispondenti celle non fanno parte della coppia candidata.

Sarà poi data una spiegazione orale più dettagliata di questo codice.

E' stata sicuramente l'implementazione più complessa che ho affrontato. E' stato difficile sia a livello algoritmico che a livello di scrittura del codice.

```
val rows = (0 until dimSudoku).toList.map(checkRow)
//we go to get the indexes
rows.zipWithIndex.foreach(t => {
  val row = t._2
  t._1.foreach(p => {
    val newValues = p.intersection.head.toList
    //in the two squares it returns the newValues tha
    //we overwrite the squares
    matList(row)(p.cell1) = newValues
    matList(row)(p.cell2) = newValues
  })
})
println("Hidden Pairs in rows:" + rows)
```

(Ricerca Hidden Pair per righe e modifica celle)

5.1.1 Interfaccia iniziale di caricamento (FileChooserMain) e FileChooser

Ha implementato l'interfaccia di caricamento iniziale (FileChooserMain).

Per la creazione dell'interfaccia di caricamento è stato usato JavaX. E' stata implementata la grafica con gli Action Listeners. Qui è stato inserito il pattern Adapter .

Il fine dell'Adapter è di fornire una soluzione astratta al problema dell'interoperabilità tra interfacce differenti.
Il problema si presenta ogni qual volta nel progetto di un *software* si debbano utilizzare sistemi di supporto (come per esempio librerie) la cui interfaccia non è perfettamente compatibile con quanto richiesto da applicazioni già esistenti.
In Scala questo pattern viene realizzato con conversioni implicite. Esse sono utili in queste occasioni: servono a ridurre il numero di conversioni esplicite che sono di solito necessarie per poter trasformare un tipo in un altro.
Qui viene adattata una funzione ad un ActionListener.
Tale implicito si trova nel companion object MyMenuHelpers, come si vede in foto.

```
object MyMenuHelpers {
  def createMenu(menu: JMenuItem*): JMenu = {
    val menuItem = new JMenu("File")
    menu.foreach(menuItem.add(_))
    menuItem
  }

  // Use of an implicit conversion to shorten the code
  implicit def functionActionListener(f: ActionEvent => Unit) =
    new ActionListener {
      def actionPerformed(event: ActionEvent) = f(event)
    }
}
```

Si tratta di un costrutto molto potente, che consente in questo caso di scrivere codice più conciso

```
openEasy.addActionListener((_: ActionEvent) => (load = false, initAndUpload(mainFrame, "input/easy"), cont()))
```

La funzione def createMenu, sempre creata in tale object, permette di aggiungere sul JMenu tutti i vari item con una sola riga di codice.

```
val menu = createMenu(openEasy, openMedium, openHard, loadFile, aboutMenu, exitMenu)
```

L'object FileChooser, invece, implementa un metodo che si avvale della classe JFileChooser che fornisce un supporto nativo per selezionare file o directory dal file system. Qui si è fatto largo uso di meccanismi quali gli impliciti.
Come si vede nella foto sottostante all'interno di Helpers sono definite due classi implicite. Le classi implicite consentono di aggiungere metodi personalizzati a tipi esistenti, senza dover modificare il loro codice, arricchendo così i tipi senza dover controllare il codice. Rendono così possibile aggiungere nuovi metodi a classi precedentemente definite.

```
object Helpers {
  implicit class MyStringHelper(str: String) {
    def filechooser(): JFileChooser = {
      val jfc = new JFileChooser(str)
      jfc.setMultiSelectionEnabled(false)

      jfc.setAcceptAllFileFilterUsed(false)
      val filter = new FileNameExtensionFilter( description = "Files txt",  extensions = "txt")
      jfc.addChoosableFileFilter(filter)
      jfc
    }
  }
  implicit class MyFileChooserHelper(jfc: JFileChooser) {
    def open(frame: JFrame): Int = jfc.showOpenDialog(frame)
    //def multipleFiles() = jfc.getSelectedFiles()
    def selectedFile() = jfc.getSelectedFile()
  }
}
```

5.3 Alberto Antonelli

- Naked Pairs: implementazione dell'algoritmo di risoluzione del sudoku tramite la strategia delle "coppie nude", spiegato in precedenza.

  Essendo un procedimento iterativo e di ricerca si è optato per l'utilizzo di vari cicli for che rendono il procedimento di ricerca agevole. Anche perché poi è uno degli algoritmi di risoluzione meno usati, anche se fondamentale per la risoluzione finale. In tutti i casi di ricerca, per riga, colonna o blocco la ricerca ha l'obiettivo di trovare due coppie di valori in caselle differenti, per poi naturalmente eliminarle in caselle ad esse correlate. Si è usato un flag per usare lo stesso procedimento per la ricerca sia per riga che per colonna.

  La difficoltà di questo algoritmo più che per l'implementazione, è stata difficoltosa nel ragionamento di come si applicava all'algoritmo.

  Naturalmente sono stati sviluppati i relativi test per assicurarsi che funzioni per il meglio.

- Funzione di refresh della lista degli elementi che l'utente ha inserito all'interno della casella come proprio elenco di possibili elementi. Consiste nella chiamata di un thread che agisce al verificarsi di un evento, in questo caso il "cliccare" su un pulsante. In questo momento il thread per ogni cella non vuota elimina gli elementi superflui.

- Visualizzazione nell'apposito "rettangolo blu" dei possibili elementi che possono ancora capitare nella casella selezionata.

- Suggerimento per l'utente, funzione che aiuta l'utente nel caso non riesca a capire quale elemento è contenuto in una determinata casella. Questa procedura utilizza

la funzione FullExploration che permette di avere la suluzione del sudoku a portata di mano.

- Funzione di risoluzione immediata del sudoku, utilizzando qualsiasi strategia di risoluzione che si è implementata.

Capitolo 6

Retrospettiva

Nel complesso, lo sviluppo del processo si è svolto senza particolari problematiche.

L'utilizzo del modello Test Driven Development (TDD) si è dimostrato una

soluzione vincente ed in grado di creare codice robusto.

Esso è stato combinato con tool automatici quali sbt e ha reso il tutto pratico ed efficiente.

Più di una volta, la presenza di test ha permesso di individuare tempestivamente bug sorti da piccoli cambiamenti nel codice altrimenti difficilmente catturabili.

Anche l'approccio Agile Scrum-based si è rilevato una buona scelta.

In conclusione, possiamo affermare che tutti i requisiti del progetto sono stati completati.

Purtroppo non si è riusciti ad inserire alcune delle feature opzionali che erano state inserite, per motivi legati ad esigenze personali.

6.1 Retrospettiva di Lorenzo Pacini

Dal mio punto di vista, (essendo il Product Owner), mi sento di dire che, sebbene sia il professore a dare il giudizio finale, è stato svolto un ottimo lavoro, poiché come precedentemente citato sono stati usati molti dei buoni pattern di programmazione e progettazione, diversi fra questi anche di aspetto avanzato.

I file che ho prodotto non superano mai le 100 righe di codice, i miei metodi non superano mai le 20 righe di codice e mediamente i file sono lunghi 50 - 60 righe, sintomo dal mio punto di vista di buona programmazione, (assenza di God Class).

Va altresì detto che ciascun modulo/classe è stato pensato per svolgere esattamente una funzione (Single responsibility principle).

Le pecche nel progetto sono, a mio avviso:

1. Il calcolo del punteggio considerato valido anche se usa il suggerimento (ho considerato questo un dettaglio)
2. Il Thread del tempo non sempre riesce a fermarsi

Le problematiche (tuttavia risolte), nel progetto sono state relativamente all'uso dello servizio di integrazione continua Travis CI utilizzato per creare e testare progetti software ospitati su GitHub e Bitbucket.

In particolare la build lanciava un'eccezione relativa al set della variabile DISPLAY, che ho poi risolto aggiungendo le seguenti righe di codice:

services:

- xvfb

Oltre a quell'operazione fatta, per far passare la build di travis è stato commentato il codice del test "A solved game" should "have all true in mask" nella classe testComplete, (che precedentemente andava in locale).

6.2 Retrospettiva di Silvia Zandoli

Dal mio punto di vista è stato realizzato un progetto buono e sono soddisfatta.

Come Scrum Master sono stata sempre disponibile nei confronti del team.

Lavorare in gruppo a mio parere non è sempre semplice, ha i suoi pregi, ma anche i suoi difetti.

Una pecca a mio avviso è stata la struttura iniziale della matrice del puzzle, un Array[Array[Int]] . A volte ha reso un po' più complesse certe parti di scrittura del codice. Personalmente avrei considerato il puzzle come una lista di celle, con ogni cella definita con le sue coordinate e un set di valori candidati.

Grazie a questo corso ho studiato questo linguaggio, che, seppur sia complesso, sta diventando sempre più importante conoscerlo. In molti corsi universitari non c'è un insegnamento simile, per cui posso ritenermi fortunata.

6.3 Retrospettiva di Alberto Antonelli

Dal mio punto di vista è stato svolto un ottimo lavoro, nonostante qualche difficoltà iniziale con l'implementazione delle strategie di risoluzione. Mi ritengo soddisfatto del progetto svolto, anche se con un maggior tempo disponibile si potrebbero risolvere alcune problematiche del sistema. Rispetto all'intero progetto sono solamente piccolezze.

Parlando del lavoro di gruppo, mi ritengo soddisfatto e ringrazio i colleghi per l'aiuto che mi hanno dato in certi momenti. Lavorare in gruppo porta spesso a delle difficoltà, soprattutto se non ci conosce o non ci si può vedere ma possiamo ritenerci abbastanza soddisfatti del lavoro svolto.

Sviluppi futuri

Tra gli sviluppi futuri che potrebbero essere intrapresi di questo progetto c'è sicuramente l'implementazione di un modulo che permette di generare i Sudoku in maniera casuale con diversi livelli di difficoltà. Il progetto attuale invece carica i Sudoku in base al livello di difficoltà e anche il timer e lo score nel caso del caricamento di un Sudoku salvato precedentemente.

In futuro lo stesso applicativo potrebbe generare casualmente i Sudoku, anche di difficoltà diverse, senza che questi vengano caricati.

Un'altro sviluppo futuro può essere quello di migliorare il modulo del Prolog e integrarlo maggiormente nel progetto.

Attualmente è stato implementato un modulo prolog per il completamento di un sudoku di qualsiasi difficoltà. Ma poichè il tempo di esecuzione di questo algoritmo di elevata complessità è risultato non ottimale, si è evitato di metterlo come algoritmo risolutore del sudoku in modo completo. In futuro si potrebbe pensare di migliorare in termini di efficienza tale algoritmo.

Appendice A

Guida Utente

Ora verrà mostrata una breve guida per capire al meglio come giocare con l'applicativo.

Dopo aver mandato l'esecuzione il programma, verrà mostrata una finestra per creare una nuova partita oppure caricarne una salvata in precedenza.

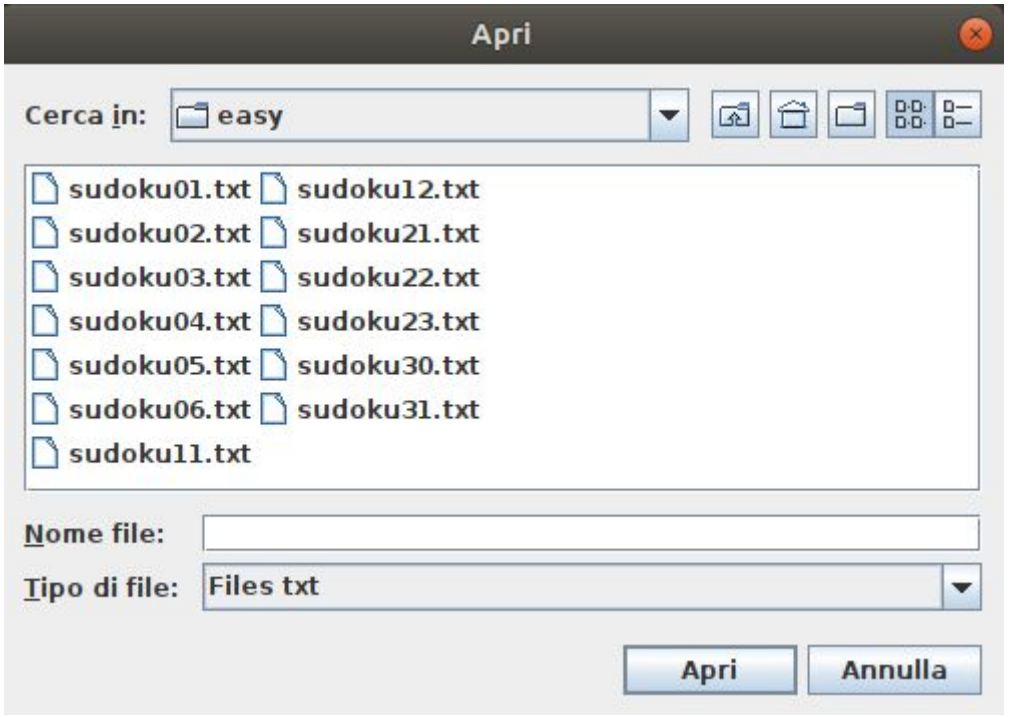
Cliccando su "File" verrà mostrato un menu con le seguenti opzioni:

- Open Easy
- Open Intermedium
- Open Hard
- Load Old
- About
- Exit

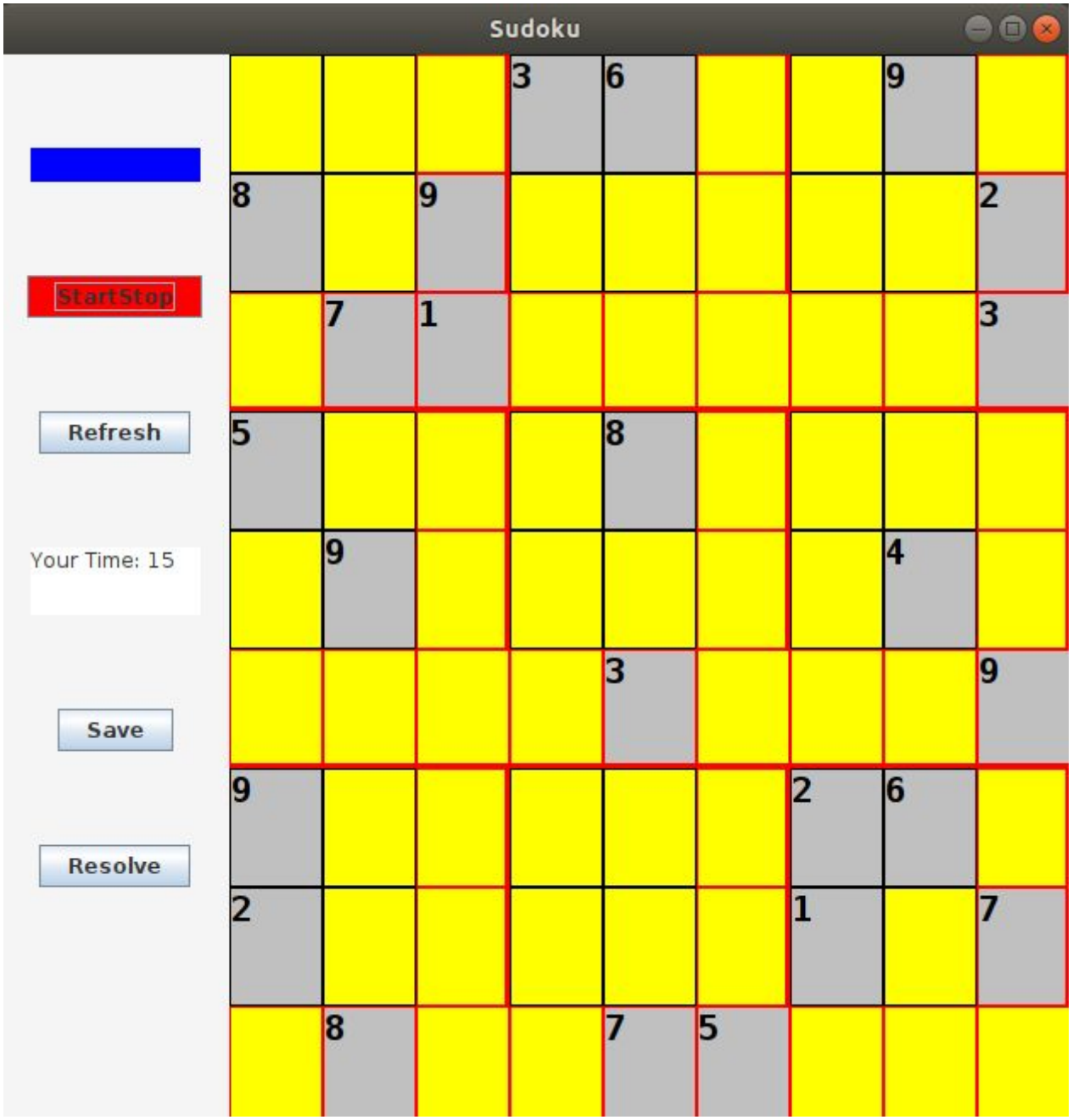
I primi tre permettono di creare una nuova partita e richiede fin da subito la scelta della difficoltà del puzzle che verrà mostrato.

Il pulsante Load Old permette invece di caricare un sudoku salvato in precedenza.

Una volta scelto come procedere, iniziando una nuova partita, l'applicativo mostrerà una finestra che permette di scegliere quale sudoku tra quelli indicati caricare.



La scelta di uno di questi porterà alla visualizzazione del sudoku, come si può vedere nell'immagine sottostante.



In questo caso è stato scelto il sudoku01 di difficoltà Easy.

Per prima cosa possiamo suddividere l'immagine precedente in due sezioni, la colonna laterale in cui sono presenti vari pulsanti e il quadrato giallo-grigio che contiene i numeri del sudoku.

La colonna è così composta dall'alto verso il basso:

- Rettangolo blu: in questo riquadro verranno mostrati tutti i possibili valori che possono finire in una determinata casella. Questa visualizzazione viene attivata quando si cliccherà sul pulsante "seeMatlist" spiegato successivamente.

- Pulsante “Start/Stop”: permette di fermare e mettere in pausa l’esecuzione del gioco (e del tempo), per una successiva ripresa.

- Pulsante “Refresh”: riguarda l’eliminazione dell’elenco di elementi inseriti dall’utente in ogni possibile riquadro, attraverso il pulsante “Insert list of numbers”. Verranno eliminati gli elementi che sono già presenti in altri riquadri della stessa riga, colonna o blocco 3x3 per i quali in precedenza era ancora possibile il verificarsi in quel riquadro. Il pulsante “Insert list of numbers” verrà spiegato successivamente.

- Rettangolo bianco: in questo viene mostrato il passare del tempo durante lo svolgimento del gioco. Una volta stoppato verrà mostrato il tempo impiegato e il punteggio ottenuto fino a quel momento.

- Pulsante “Save”: permette di salvare il sudoku aperto per una successiva ripresa.

- Pulsante “Resolve”: permette di risolvere e concludere il sudoku senza dover inserire alcun numero nella griglia. Può essere utilizzato quando non si vuole continuare a provare a risolvere il sudoku e si vuole vedere la soluzione.

Invece, per quanto riguarda la griglia, come si può vedere dall’immagine precedente le caselle sono di due differenti colori, grigie e gialle. Le prime sono caselle che non potranno essere modificate perché in esse sono indicati i valori definitivi e corretti per la risoluzione del sudoku. Le seconde invece sono le caselle a cui dovranno essere aggiunti valori per completare il sudoku. Cliccando su una casella verrà mostrata un’ulteriore finestra.

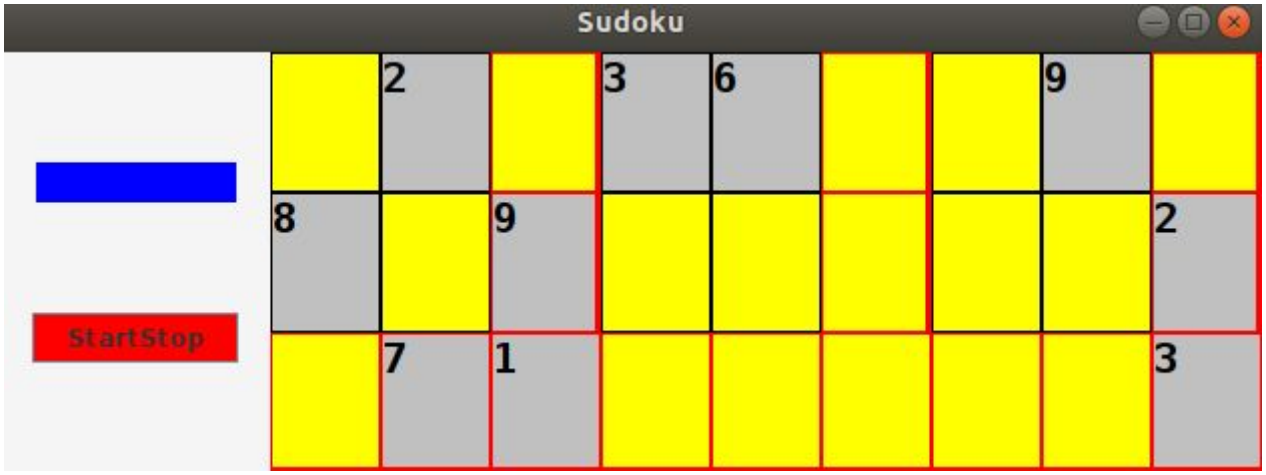
- Insert numbers: permette di inserire un elemento nel riquadro.

  Se il numero è sbagliato mostrerà una finestra di errore per poi permetterci di riprovare con un altro numero.

  In questo esempio abbiamo provato ad inserire il valore 5 nella casella.

Successivamente proviamo con il numero 2, ed è corretto.

Ora quella casella si colorerà di grigio, e il suo valore sarà definitivo perché coerente con la soluzione.



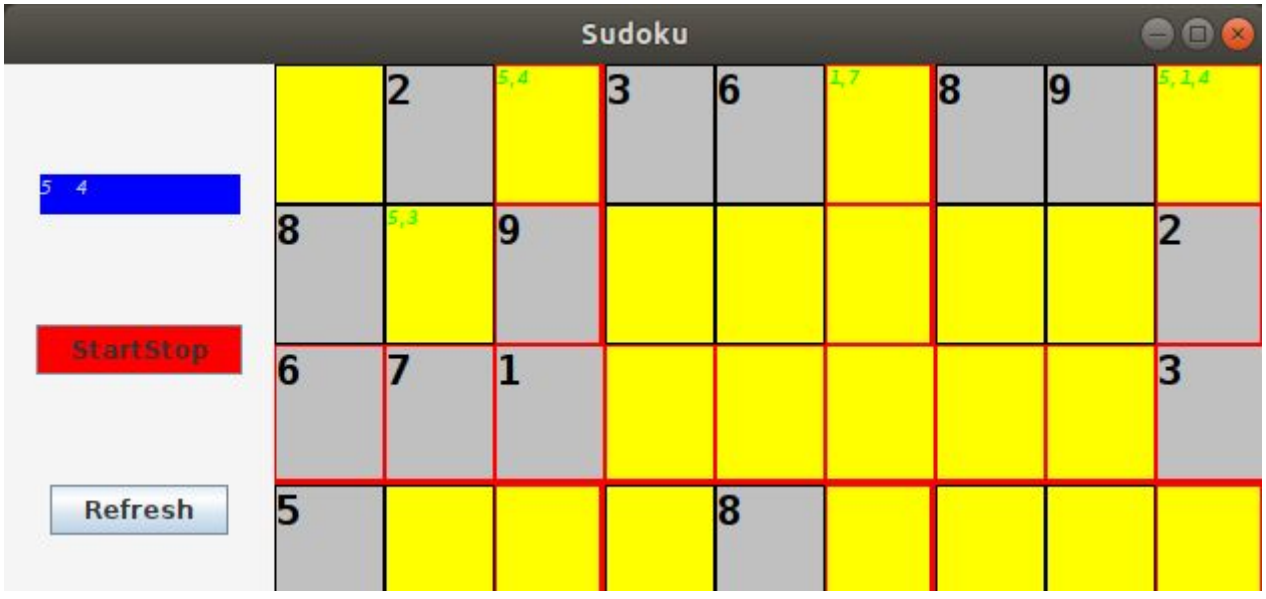
- Insert list of numbers: permette all'utente di inserire un elenco di valori all'interno della casella per facilitare la risoluzione.

  I valori devono essere scritti in sequenza senza alcun carattere di separazione o punteggiatura.

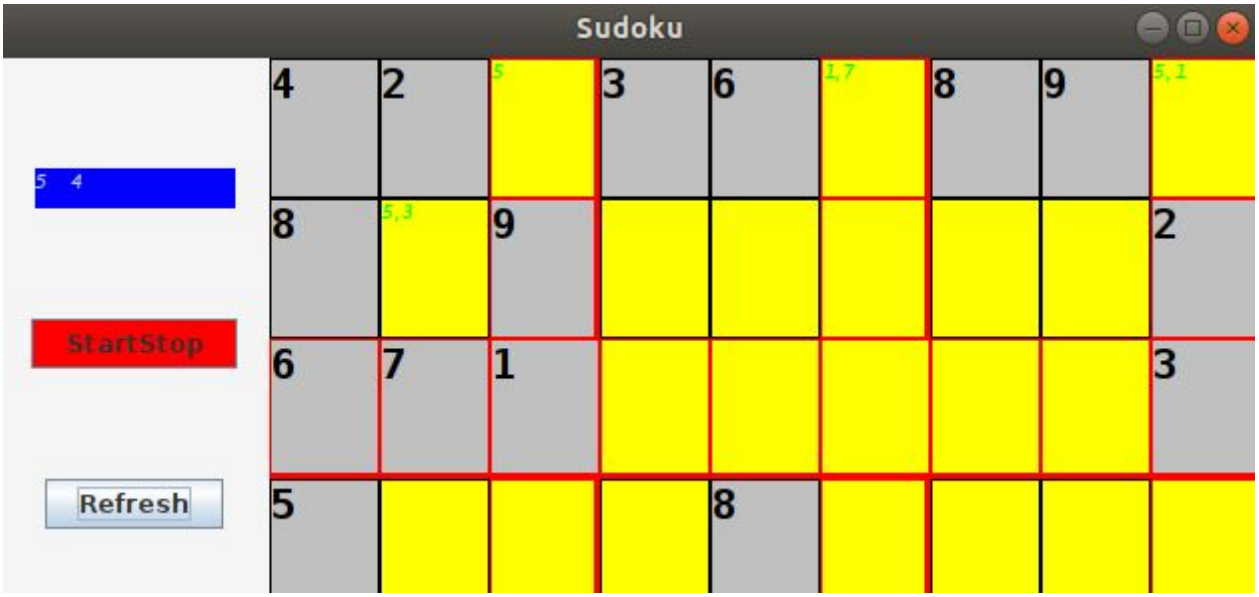
  L'utente potrebbe valutare in una determinata casella un elenco di valori che sono impossibilitati ad essere in altre caselle e per averli sempre presenti nel sudoku può usare questa funzione.

  Nell'esempio sottostante l'utente ha inserito vari valori nelle caselle, tutti ancora possibili altrimenti non verrebbero aggiunti se già presenti nella riga, colonna, blocco.

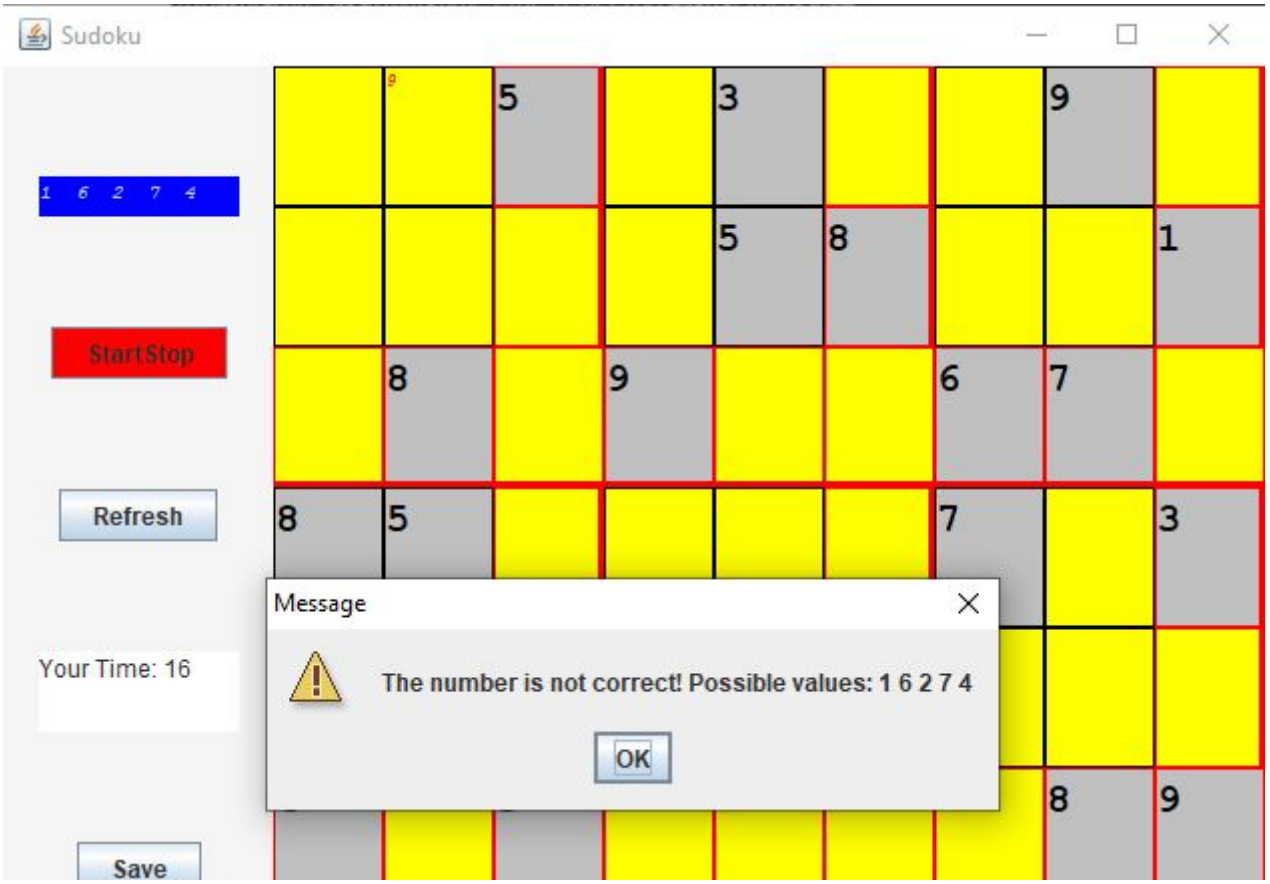
  Attraverso il pulsante refresh gli elementi inseriti come lista all'interno della casella verranno aggiornati rispetto ai successivi inserimenti in altre caselle.



Dopo l'inserimento del valore 4 in alto a sinistra e il refresh.



Nella figura sotto si è provato a inserire 9. Visto che tale numero non appartiene alla lista di numeri per quella casella, verrà segnalato errore.

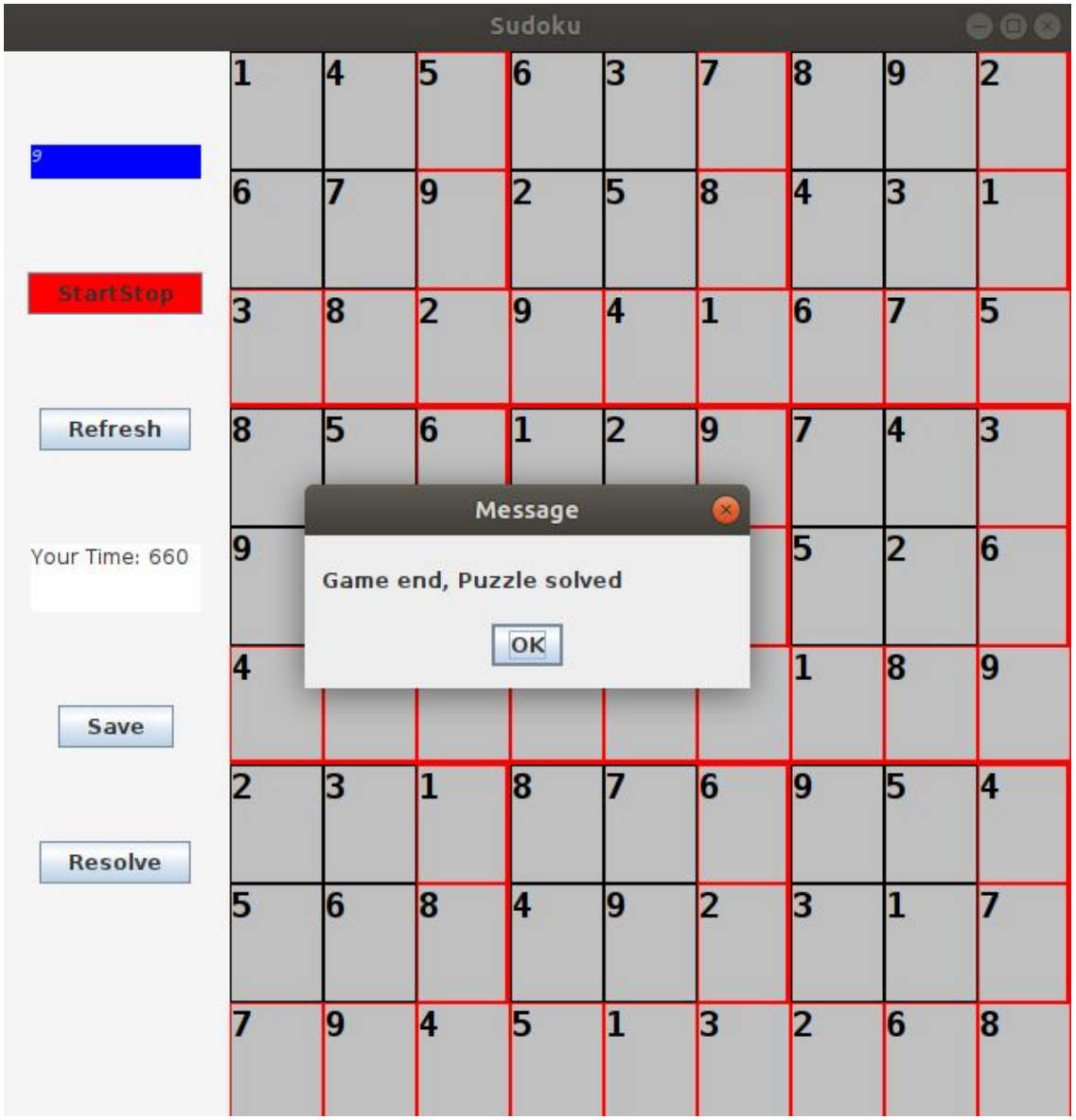


- seeMatlist: permette di vedere i possibili valori che possono capitare nella casella selezionata. Questi rispetto a quelli inseriti dall'utente conterranno sempre l'elemento corretto di quella determinata casella. La lista verrà visualizzata nel riquadro blu della colonna a sinistra come spiegato in precedenza.

- Help!: permette di inserire l'elemento corretto in quella casella, è l'applicativo che lo inserisce.

Una volta terminato il sudoku viene mostrata la schermata di fine sudoku.

L'applicativo successivamente chiede se si vuole iniziare una partita nuova.



Note aggiuntive

Il costo computazionale dell'algoritmo di backtracking nella forza bruta del sudoku nel caso peggiore è di circa $n^m <= n^{n*n} = 2^{n*n \log(n)}$,dove m sono le possibili scelte e n sono i numeri.

L'algoritmo di backtrack è quindi esponenziale ma il Sudoku n x n non è trattabile perchè ad oggi non esiste un algoritmo polinomiale.

È possibile tuttavia verificare la correttezza di una data soluzione in tempo polinomiale. Essa richiede circa m x n = $O(n^3)$ passi .

Bibliografia

https://www.sudokuwiki.org/sudoku.htm

(per quanto riguarda le tecniche di risoluzione utilizzate nel progetto)

http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/informaticaapplicata/all/pnp.pdf

(per quanto riguarda il costo computazionale)