

CA-SIM  
“Cellular Atomata Simulator”

Chiasserini Raul, Drudi Lorenzo, Sanzani Filippo, Zama Simone

23 aprile 2022

# Indice

<b>1</b>	<b>Analisi</b>	<b>3</b>
1.1	Requisiti . . . . .	3
1.1.1	Bryan's Brain . . . . .	4
1.1.2	CoDi . . . . .	5
1.1.3	Conway's Game of Life . . . . .	7
1.1.4	Langton's Ant . . . . .	8
1.1.5	Predators and Preys . . . . .	9
1.1.6	Rule 110 . . . . .	10
1.1.7	Requisiti funzionali opzionali . . . . .	11
1.1.8	Requisiti non funzionali . . . . .	11
1.2	Analisi e modello del dominio . . . . .	12
<b>2</b>	<b>Design</b>	<b>13</b>
2.1	Architettura . . . . .	13
2.2	Design dettagliato . . . . .	14
2.2.1	Lorenzo Drudi . . . . .	14
2.2.2	Filippo Sanzani . . . . .	21
2.2.3	Simone Zama . . . . .	27
2.2.4	Raul Chiasserini . . . . .	32
<b>3</b>	<b>Sviluppo</b>	<b>37</b>
3.1	Testing automatizzato . . . . .	37
3.1.1	Lorenzo Drudi . . . . .	37
3.1.2	Filippo Sanzani . . . . .	37
3.1.3	Simone Zama . . . . .	37
3.1.4	Raul Chiasserini . . . . .	38
3.2	Metodologia di lavoro . . . . .	38
3.2.1	Lorenzo Drudi . . . . .	38
3.2.2	Filippo Sanzani . . . . .	39
3.2.3	Simone Zama . . . . .	39
3.2.4	Raul Chiasserini . . . . .	40

3.3	Note di sviluppo . . . . .	40
3.3.1	Lorenzo Drudi . . . . .	40
3.3.2	Filippo Sanzani . . . . .	42
3.3.3	Simone Zama . . . . .	43
3.3.4	Raul Chiasserini . . . . .	43
<b>4</b>	<b>Commenti finali</b>	<b>44</b>
4.1	Autovalutazione e lavori futuri . . . . .	44
4.1.1	Lorenzo Drudi . . . . .	44
4.1.2	Filippo Sanzani . . . . .	44
4.1.3	Simone Zama . . . . .	45
4.1.4	Raul Chiasserini . . . . .	45
<b>A</b>	<b>Guida utente</b>	<b>47</b>
A.1	Le view . . . . .	47
A.1.1	Configurazione degli automi . . . . .	47
A.1.2	CoDi . . . . .	48
<b>B</b>	<b>Esercitazioni di laboratorio</b>	<b>49</b>
B.0.1	Lorenzo Drudi . . . . .	49
B.0.2	Filippo Sanzani . . . . .	49
B.0.3	Simone Zama . . . . .	50

# Capitolo 1

## Analisi

### 1.1 Requisiti

Il software mira alla costruzione di un simulatore di automi cellulari di nome CA-SIM (Cellular Automata Simulator). Un automa cellulare è un modello matematico che consiste in una griglia costituita da celle che rappresentano delle entità che evolvono nel tempo. Ciascuna di queste celle può assumere un insieme finito di stati, ogni unità di tempo questa griglia verrà aggiornata seguendo delle regole definite dalla tipologia di automa simulato. Questi modelli matematici trovano numerosi utilizzi, come ad esempio per rappresentare il comportamento fisico dei gas perfetti, l'evoluzione di una popolazione oppure per esprimere determinati pattern.

#### Requisiti funzionali obbligatori

- Il suddetto software dovrà dare la possibilità all'utente di visualizzare la simulazione di un automa alla volta scelto tra i seguenti:
  - Brian's Brain;
  - CoDi;
  - Conway's Game of Life;
  - Langton's Ant;
  - Predators and Preys;
  - Rule 110.
- CA-SIM dovrà permettere la simulazione automatica dell'automa scelto. Data quindi una configurazione iniziale verrà avviata la simulazione che seguirà le regole senza necessitare di input dall'utente.

- Il software dovrà dare la possibilità anche di una esecuzione manuale in cui l'utente decide quando passare alla successiva iterazione.
- La griglia iniziale dovrà essere generata da CA-SIM.
- CA-SIM dovrà offrire inoltre statistiche sulla simulazione come numero di iterazioni o di individui/tipologie di celle attualmente presenti.

Di seguito una breve spiegazione degli automi presentati sopra.

### 1.1.1 Bryan's Brain

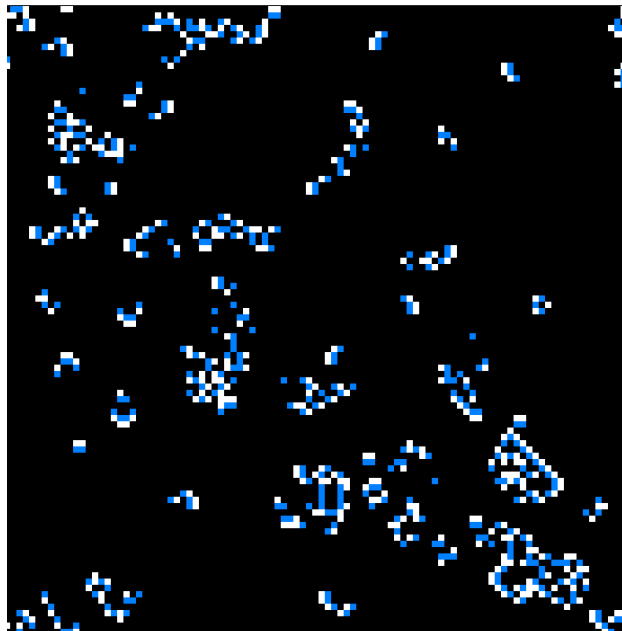


Figura 1.1: Un frame di Bryan's Brain.

Bryan's Brain è un automa cellulare bidimensionale, le sue celle possono assumere **tre** stati:

- Alive (bianco in Figura 1.1);
- Dying (azzurro in Figura 1.1);
- Dead (nero in Figura 1.1).

Le regole che governano la simulazione di questo automa cellulare sono le seguenti:

- Una cella *ALIVE* passa sempre nello stato *DYING*;
- Una cella *DYING* passa sempre nello stato *DEAD*;
- Una cella *DEAD* se ha esattamente 2 vicini *ALIVE* passa allo stato *ALIVE*, altrimenti rimane *DEAD*.

### 1.1.2 CoDi

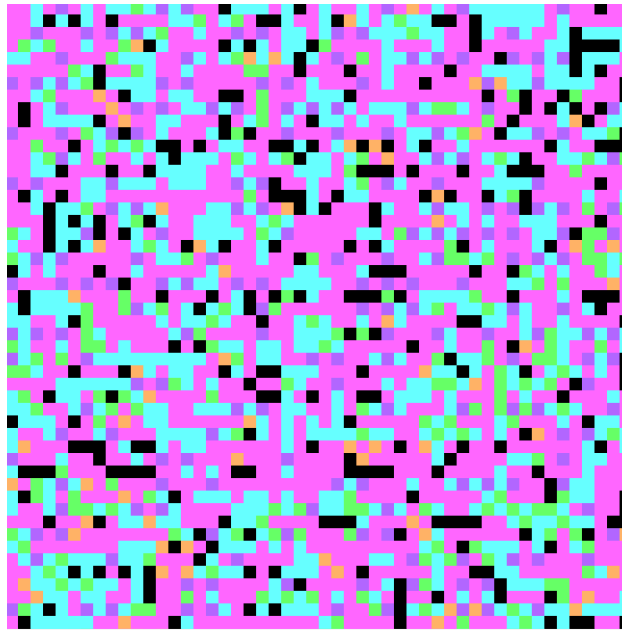


Figura 1.2: Un frame di CoDi.

Il nome sta per Collect and Distribute e si riferisce alla propagazione dei segnali nelle Spiking Neural Networks che cercano di emulare il funzionamento dei neuroni “naturalisti”. In questo automa ci sono vari tipi di celle che rappresentano le parti di un neurone e interagiscono tra di loro secondo regole che imitano il passaggio di segnali nei neuroni. La sua particolarità è che, a differenza della maggior parte degli automi, non si basa su una griglia 2D ma opera su uno spazio tridimensionale.

I tipi di celle sono:

- BLANK (nero in fig. 1.2): cella vuota;
- NEURON (viola in fig. 1.2): è il nucleo del neurone, da qui partono gli impulsi (o segnali);

- AXON (fucsia in fig. 1.2): sono i prolungamenti del corpo del neurone attraverso i quali viaggiano i segnali verso l'esterno. Ricevono gli impulsi dal nucleo e li trasmettono verso tutte le direzioni esclusa quella di ricezione (in fig. 1.2 notiamo gli assoni con segnale dal colore arancione);
- DENDRITE (azzurro in fig. 1.2): trasportano verso il nucleo i segnali provenienti dagli altri neuroni. In particolare essi ricevono segnali da tutte le direzioni ma possono propagare solamente in una direzione (in fig. 1.2 notiamo i dendriti con segnale dal colore verde).

Nel modello di neurone descritto da CoDi non esistono le sinapsi ed il passaggio di impulsi avviene direttamente tra DENDRITE ed AXON di neuroni limitrofi.  
 Inoltre ogni cella possiede:

- Un GATE: esso assume un significato diverso a seconda dello stato:
  - Neuroni: indica la direzione dell'assone;
  - Assoni: indica la direzione da cui ricevono il segnale;
  - Dendriti: indica la direzione verso la quale possono propagare il segnale.
- Un CROMOSOMA: descrive come assoni e dendriti possono inviare segnali (in particolare indica verso quali direzioni possono farlo);
- Un ACTIVATION COUNTER: valore che indica se la cella è attiva (emette un impulso) oppure no. Nei neuroni corrisponde al potenziale di membrana.

La sua esecuzione è suddivisa in due fasi, una prima fase chiamata "*Growth Phase*" in cui la rete neurale cresce sopra ai cromosomi contenuti in ognuna delle celle della griglia che compongono l'automa ed una successiva fase di signaling chiamata appunto "*Signaling Phase*", in quest'ultima fase i segnali partono dal corpo del neurone verso i loro assoni per poi passare ai dendriti a loro connessi. L'automa passa alla seconda fase quando non è più possibile far crescere la rete.

GrowthUpdateRule:

- BLANK CELL:
  - Con probabilità del 5% diventa NEURON;

- se arriva in input esattamente un segnale da un assone diventa AXON;
  - se arriva in input esattamente un segnale da un dendrite diventa DENDRITE;
  - altrimenti resta vuota.
- NEURON: invia un segnale "di dendrite" in tutte le direzioni tranne che verso il suo gate e verso l'opposto al gate verso le quali manda un segnale "di assone";
  - AXON: invia un segnale verso tutti i vicini a cui è possibile farlo;
  - DENDRITE: invia un segnale verso tutti i vicini a cui è possibile farlo.

SignalingUpdateRule:

- NEURON: ad ogni step incrementa il proprio potenziale di uno, se giunge al valore soglia allora trasmette l'impulso verso i propri assoni;
- AXON: trasmette i segnali in tutte le direzioni;
- DENDRITE: trasmette i segnali verso il proprio gate.

### 1.1.3 Conway's Game of Life

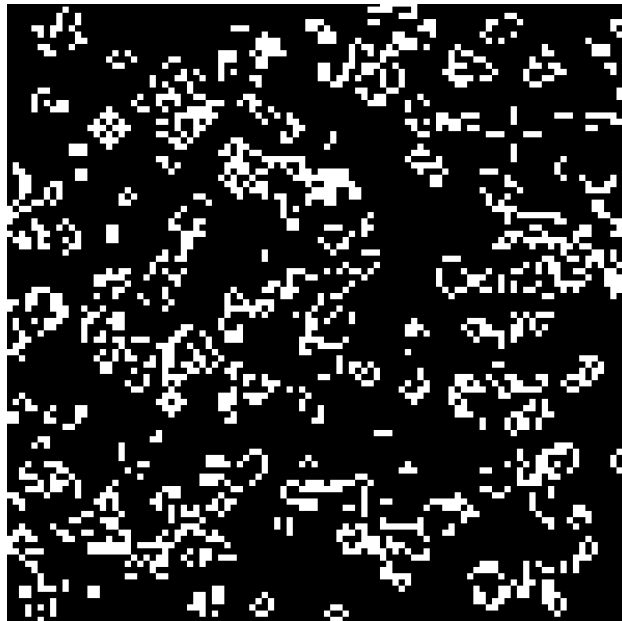


Figura 1.3: Un frame di Game of Life.



Probabilmente l'automa cellulare più famoso, ideato negli anni '70 dal matematico John Horton Conway, permette la creazione di pattern che si ripetono con un determinato periodo  $N$  (ogni  $N$  iterazioni l'automa riproduce la stessa sequenza). È un automa Turing completo.

Game of life è un automa bidimensionale in cui le celle possono assumere due stati:

- *ALIVE* (bianco in Figura 1.3)
- *DEAD* (nero in Figura 1.3)

Le celle di questo automa cambiano stato secondo le seguenti regole:

- Ogni cella *ALIVE* con meno di 2 vicini *ALIVE* diventa *DEAD*, per sottopopolazione.
- Ogni cella *ALIVE* con 2 o 3 vicini *ALIVE* rimane *ALIVE*.
- Ogni cella *ALIVE* con più di 3 vicini *ALIVE* diventa *DEAD*, per sovrappopolazione.
- Ogni cella *DEAD* con 3 vicini *ALIVE* diventa *ALIVE*, per riproduzione.

#### 1.1.4 Langton's Ant



Figura 1.4: Un frame di Langton's Ant.

è una macchina di Turing a due dimensioni, la versione base di questo automa prevede la simulazione di una sola formica che si muove secondo determinate regole e colora le celle di bianco e nero, ma può essere esteso a più colori. Le regole utilizzate nell'implementazione di Langton's Ant all'interno di CA-SIM sono le seguenti:

- All'interno di una griglia bidimensionale formata da celle sono presenti una o più formiche;
- Ogni cella può assumere uno di due stati: ON (bianco) e OFF (nero);
- Ad ogni iterazione le formiche si girano in senso orario se risiedono su una cella con stato OFF, altrimenti si girano in senso antiorario, invertono lo stato della cella in cui risiedono e si spostano di una posizione nella direzione in cui sono rivolte.

### 1.1.5 Predators and Preys

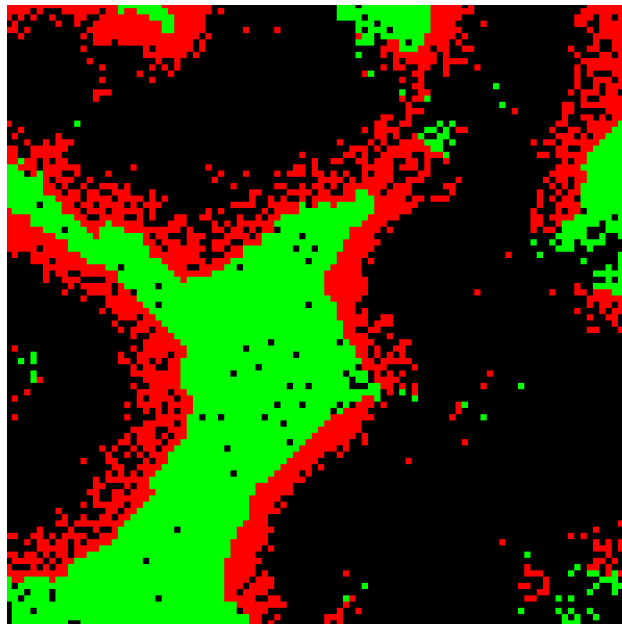


Figura 1.5: Un frame di Predators and Preys.

questo automa cellulare cerca di simulare l'evoluzione di una popolazione contenente prede e predatori in un ambiente chiuso. Si possono notare pattern interessanti come l'estinzione di una o entrambe le specie o un equilibrio

tra le due dove il numero di individui continua ad oscillare ma non raggiunge mai lo zero.

Nell'implementazione di CA-SIM le celle possono assumere uno di tre stati: preda (verde), predatore (rosso), morto (nero). Ad ogni iterazione le celle prede selezionano una cella random tra i propri vicini e se questa è morta si spostano su di essa. I predatori invece mangiano una preda vicina se presente, o si spostano su una cella morta se presente. Ogni cella preda e predatore ha un contatore di salute che, quando raggiunge il valore massimo, permette alla cella di riprodursi nel momento in cui si sposta, se ciò accade il contatore tornerà ad un valore predefinito. Il contatore aumenta ad ogni iterazione per le prede, mentre per i predatori diminuisce in ogni iterazione in cui non mangiano una preda e aumenta ogni volta che ne mangiano una. Se questo contatore raggiunge il suo valore minimo, la cella muore.

### 1.1.6 Rule 110

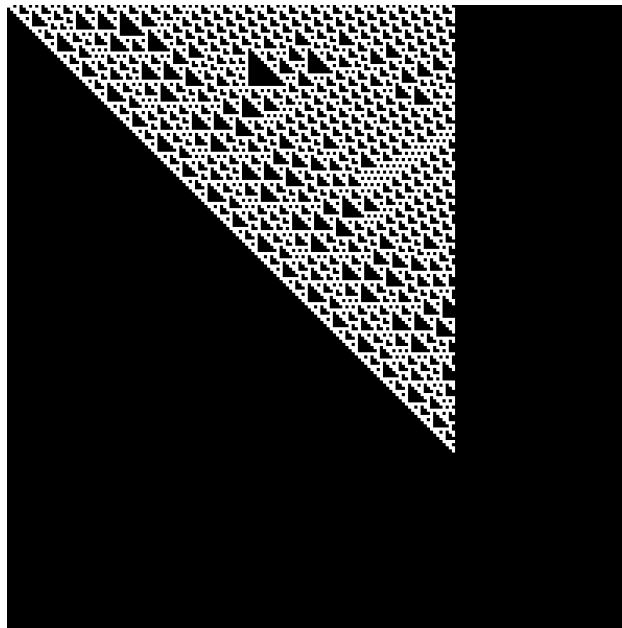


Figura 1.6: Un frame di Rule110.

Rule110 è un automa cellulare ad una dimensione che si espande all'infinito, ogni cella può assumere due valori e lo stato successivo è definito dai valori che la cella stessa e i suoi vicini assumono prima dell'aggiornamento. Ha la particolarità di essere Turing completo.

Le celle possono assumere due stati:

- *ALIVE* (1 nella tabella sotto, bianco in Figura 1.6)
- *DEAD* (0 nella tabella sotto, nero in Figura 1.6)

In base alla configurazione di una cella e delle sue due vicine si può definire lo stato della stessa cella al passo successivo in base ad uno schema:

000	001	010	011	100	101	110	111
0	1	1	1	0	1	1	0

### 1.1.7 Requisiti funzionali opzionali

- CA-SIM dovrà, oltre agli automi cellulari già citati, includere:
  - WireWorld: automa cellulare Turing completo creato nel 1987 molto utilizzato per simulare Transistors.
  - Empire: automa cellulare che permette la simulazione dell'evoluzione di una popolazione.
- CA-SIM dovrà dare all'utente la possibilità di definire uno stato iniziale tramite l'interfaccia grafica.
- CA-SIM dovrà permettere il salvataggio ed il caricamento degli stati iniziali da file.
- CA-SIM dovrà permettere di fare zoom su determinate parti della griglia durante la simulazione.

### 1.1.8 Requisiti non funzionali

- CA-SIM dovrà essere portabile e quindi sarà possibile utilizzarlo su macchine Windows, Mac OS e UNIX/Linux.
- CA-SIM dovrà offrire una visualizzazione fluida di tutti gli automi cellulari.

## 1.2 Analisi e modello del dominio

CA-SIM fornirà all'utente la possibilità di scegliere quale automa simulare tra le diverse opzioni presenti. Una simulazione di un automa cellulare (Automaton) consiste in una griglia formata da delle celle (Cell) che rappresentano le entità stesse della simulazione. Ogni cella è caratterizzata da uno stato (S) che può evolvere nel tempo. Queste griglie potranno essere generate in modo casuale o caricate da file, quest'ultima operazione verrà implementata se il monte ore lo consentirà. Ogni automa dovrà quindi avere la possibilità di inizializzare la propria griglia. Durante la simulazione l'automata sarà responsabile di aggiornare tutte le sue celle fornendo poi lo stato aggiornato della griglia. Per eseguire l'operazione di aggiornamento sarà necessario definire le regole secondo le quali le celle evolvono nel tempo (UpdateRule) e dal momento in cui queste regole vengono applicate ad ogni cella considerando anche lo stato delle celle vicine gli automi stessi dovranno poter calcolare quali sono i vicini di una cella.

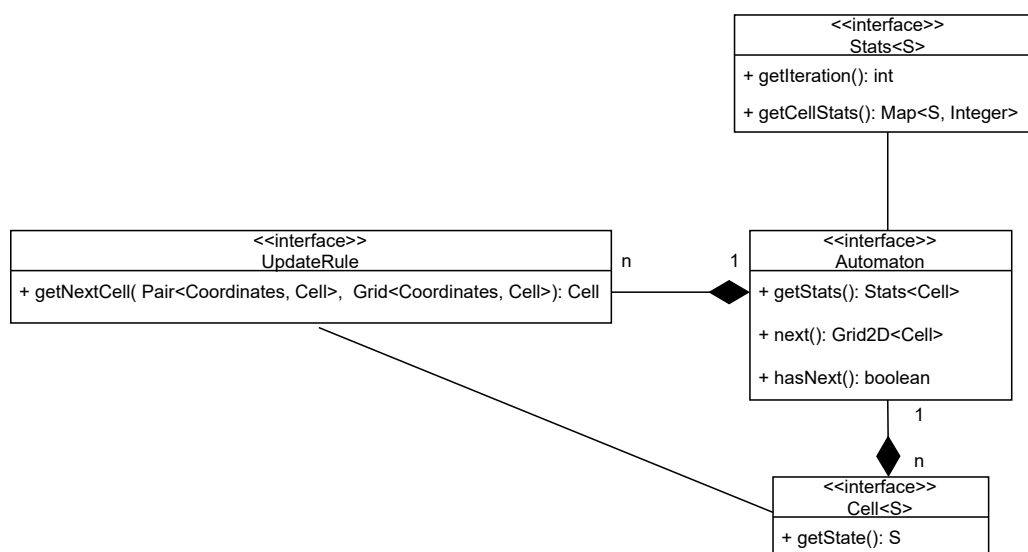


Figura 1.7: Rappresentazione del modello del dominio.

# Capitolo 2

## Design

### 2.1 Architettura

L'architettura di CA-SIM segue il pattern architetturale MVC. Abbiamo optato per questa architettura per poter separare in modo netto la logica di simulazione, dall'eventuale gestione di file per il salvataggio e dalla view. I tre macro-blocchi di MVC saranno definiti da un'astrazione di un'automa cellulare rappresentata dall'interfaccia *Automaton* e dalla view che sarà generica e permetterà la visualizzazione di tutti gli automi. Per far comunicare la view e il model ci si appoggerà all'interfaccia *AutomatonController*.

Inoltre la view è stata a sua volta scomposta, le interfacce grafiche più complesse verranno definite usando file *.fxml*, questi conterranno la struttura della view, ci sarà poi una classe *ViewController* che si occuperà di controllare la view definita in precedenza e interfacciarla con i controller di MVC.

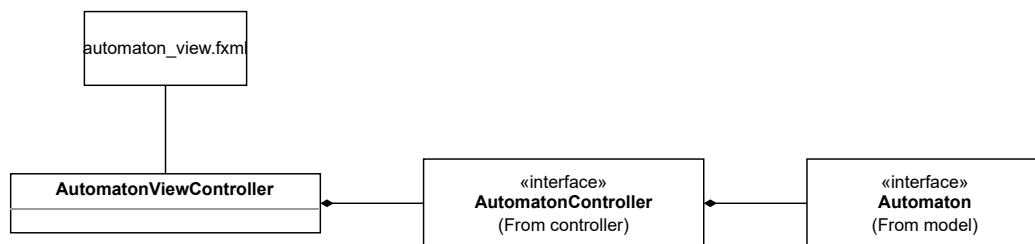


Figura 2.1: MVC con la suddivisione tra definizione e controller della view, nello specifico la view di simulazione.

## 2.2 Design dettagliato

### 2.2.1 Lorenzo Drudi

Nello sviluppo di CA-SIM mi sono occupato della creazione di una generica astrazione di un automa cellulare che potesse poi essere utilizzata facilmente per tutte le diverse implementazioni. Successivamente ho anche implementato l'automa CoDi.

#### Model Abstraction

Inizialmente l'obiettivo è stato quello di individuare tutti gli elementi in comune per poter costruire una struttura implementativa ottimale. Per vederne una rappresentazione vedi Figura 1.7.

**Cell** è l'entità rappresentante le celle che compongono la griglia dell'automa. Al proprio interno contiene lo stato corrente della cella "S". Lo stato di ogni automa è descritto da una enumerazione.

#### Rappresentazione dell'automa

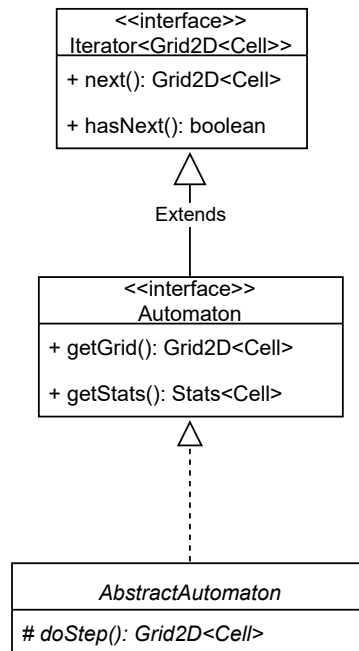


Figura 2.2: Rappresentazione di *Automaton*, l'interfaccia principale dell'astrazione.

**Automaton** è l'interfaccia che rappresenta l'automa stesso. Esso permette di ottenere lo stato corrente dell'automa (quindi della griglia che lo rappresenta), permette di controllare se è possibile effettuare un ulteriore step oltre che passare allo step successivo ed ottenere statistiche quali numero di iterazioni e numero di celle vive per ogni tipo.

**Problema** Per quanto riguarda le statistiche l'incremento dell'iterazione è un comportamento comune a tutti gli automi. Essi sono quindi tutti dipendenti da questo concetto.

**Soluzione** La soluzione scelta è stata quella di utilizzare il **Template Method Pattern** ed avere quindi una abstract class **AbstractAutomaton** che racchiude gli elementi comuni dei diversi automi. Il template method è **next** che chiama il metodo protetto e astratto **doStep**. Questa soluzione ha permesso di minimizzare la duplicazione di codice racchiudendo il concetto di iterazione nella astrazione e rendendo le implementazioni indipendenti da esso.

## Regole di aggiornamento

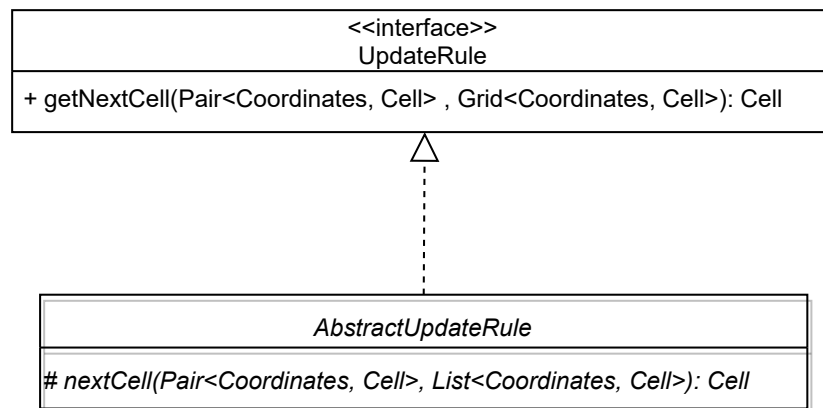


Figura 2.3: Rappresentazione di *UpdateRule*, l'interfaccia che descrive l'aggiornamento di una cella dell'automa.

**UpdateRule** rappresenta la regola che permette di ottenere il successivo stato di una cella. In particolare opera tramite il metodo **getNextCell** che prende in input la coppia Coordinate/Cella e la griglia e ritorna una nuova cella aggiornata rispetto a quella presa in input. Sottolineiamo che viene ritornata una nuova cella e non viene quindi modificata la cella presa in



input. Questo consente di aumentare l'*encapsulation* non dovendo inserire quindi nella cella metodi che consentono di variarne campi.

**Problema** I diversi automi durante la fase di update di una cella non necessitano dell'intera griglia ma solamente dei suoi vicini. Quest'ultimo concetto però non è generico, infatti gli automi utilizzano diverse definizioni di "vicini di una cella" e quindi diverse strategie per poterli ottenere partendo dalla griglia.

**Soluzione** Nella fase di design è stato scelto l'utilizzo del pattern **Strategy**. Nel problema in questione la strategia è l'algoritmo che data una cella e la griglia in cui si trova permette di ottenere la lista dei suoi vicini. Essa viene passata al costruttore di **AbstractUpdateRule**, classe astratta che implementa **UpdateRule**. Nel nostro specifico caso questo viene fatto grazie all'utilizzo della functional interface **BiFunction**.

**Problema** Data la strategia che permette di passare dalla griglia ai vicini la sua applicazione è la medesima per tutti gli automi (basterà applicare infatti l'algoritmo alla griglia). Oltre che un elemento in comune il passaggio dalla griglia ai vicini è anche complessità che si va ad aggiungere ad ogni automa.

**Soluzione** Visto il problema è stato scelto di nascondere il processo di calcolo dei vicini e lasciarlo interno all'astrazione. Per fare questo è stato utilizzato nuovamente il pattern **Template Method**. In questo caso il metodo template è **getNextCell** che chiama il metodo astratto e protetto **nextCell**. Come nel caso precedente anche in questa situazione questo ci permette di minimizzare la duplicazione di codice e ridurre poi la complessità interna alle implementazioni dei diversi automi.

**Stats** è un **Data Transfer Object** contenente le statistiche sull'automa. In particolare contiene il numero di step fatti ed il numero di celle vive per ogni stato possibile. È utilizzato per trasportare le statistiche riguardanti gli automi.

## CoDi

L'implementazione di *CoDi* sfrutta pensatamente l'astrazione presentata precedentemente (Figura 2.4 mostra i componenti principali dell'implementazione).

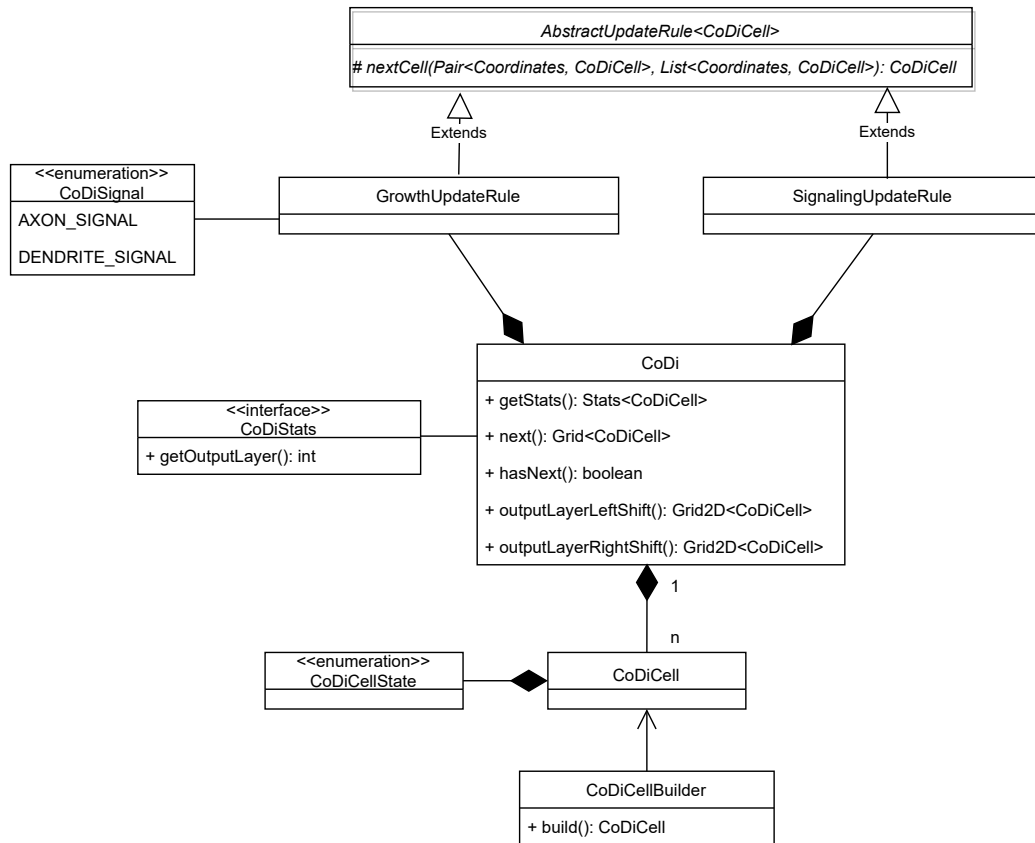


Figura 2.4: General view di *CoDi*.

## La classe CoDi

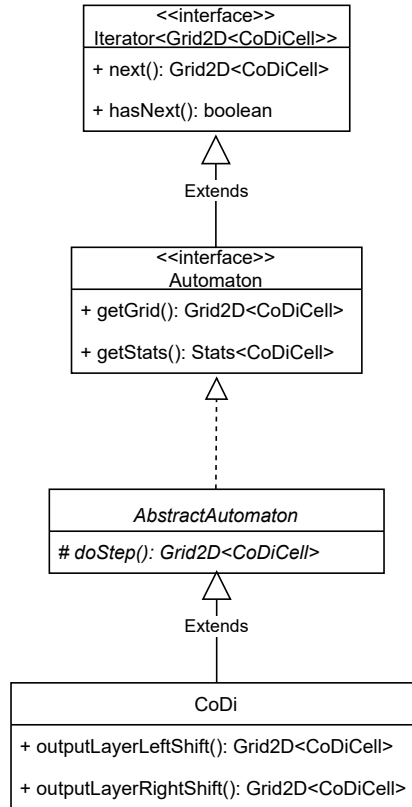


Figura 2.5: L'implementazione di **Automaton** per *CoDi*.

**CoDi** è la classe principale dell'automa. Essa estende **AbstractAutomaton** aggiungendo ulteriori funzionalità. L'automa infatti opera su una griglia 3D dando in output verso l'esterno solamente un layer (uno strato) di questa. *CoDi* tramite i due metodi *outputLayerLeftShift* e *outputLayerRightShift* permette di cambiare il layer che viene visualizzato dando la possibilità di eseguire uno shift di questo verso sinistra oppure verso destra.

## Le celle di CoDi

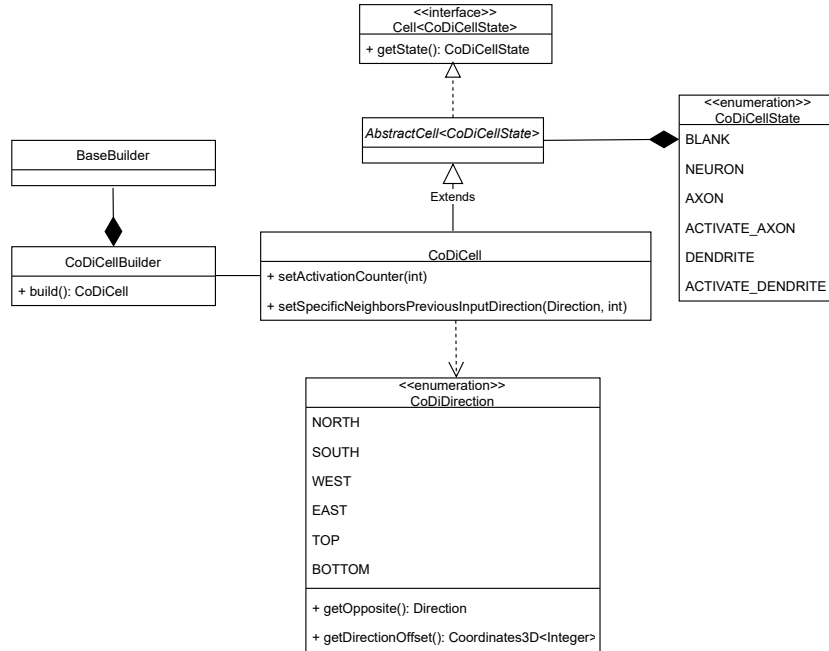


Figura 2.6: L'implementazione di **Cell** per *CoDi*.

**CoDiCell** è l'entità che rappresenta le celle che vanno a comporre la griglia dell'automa. Per la rappresentazione degli stati che possono assumere le celle è stata utilizzata l'enumerazione **CoDiCellState**. Oltre agli stati base contiene anche *ACTIVATE\_AXON* e *ACTIVATE\_DENDRITE* volti alla rappresentazione della cella quando al proprio interno contengono un segnale. (Per quanto riguarda il gate dei neuroni, visto che la visualizzazione grafica è in 2D, esso viene posto sullo stesso layer di visualizzazione del neurone.)

Per il concetto di direzione è stata utilizzata l'enumerazione **CoDiDirection**, che contiene tutte le direzioni necessarie alla regola di Von Neumann applicata ad uno spazio tridimensionale. Ad ogni direzione è associato un offset (da un punto di vista fisico non è altro che il versore di quella direzione), questo permette agilmente di ottenere il vicino di una cella attraverso la somma di coordinate.

**CellBuilder** è un'interfaccia che consente la creazione di **CoDiCell**.

**Problema** In molte situazioni, specialmente in *GrowthUpdateRule* e *SignalUpdateRule*, devono essere create spesso nuove *CoDiCell*. Questo perché

l'aggiornamento di una cella prevede proprio la sostituzione di essa con una nuova cella.

**Soluzione** Viste le numerose proprietà risulta molto comoda la possibilità di avere una costruzione step-by-step di una cella. Per risolvere questo problema si è scelto il pattern creazionale **builder** con una fluent interface per rendere più pulito il codice.

### Le regole di aggiornamento di CoDi

**GrowthUpdateRule** e **SignalingUpdateRule** sono le due regole di update applicate nelle due diverse fasi dell'automa. Essendo l'esecuzione divisa in due differenti parti ed essendo le regole da applicare indipendenti tra loro la decisione è stata quella di separarle totalmente. Per la descrizione dei segnali è stata utilizzata l'enumerazione **Signal** che permette di indicare segnali provenienti da assoni (AXON\_SIGNAL) e da dendriti (DENDRITE\_SIGNAL).

## 2.2.2 Filippo Sanzani

Nella realizzazione di CA-SIM mi sono principalmente occupato dell'astrazione della view, nello specifico i componenti e la view di simulazione e nell'individuazione delle view necessarie al funzionamento del software. Ho anche implementato l'automa cellulare bryan's brain.

### Astrazione della view

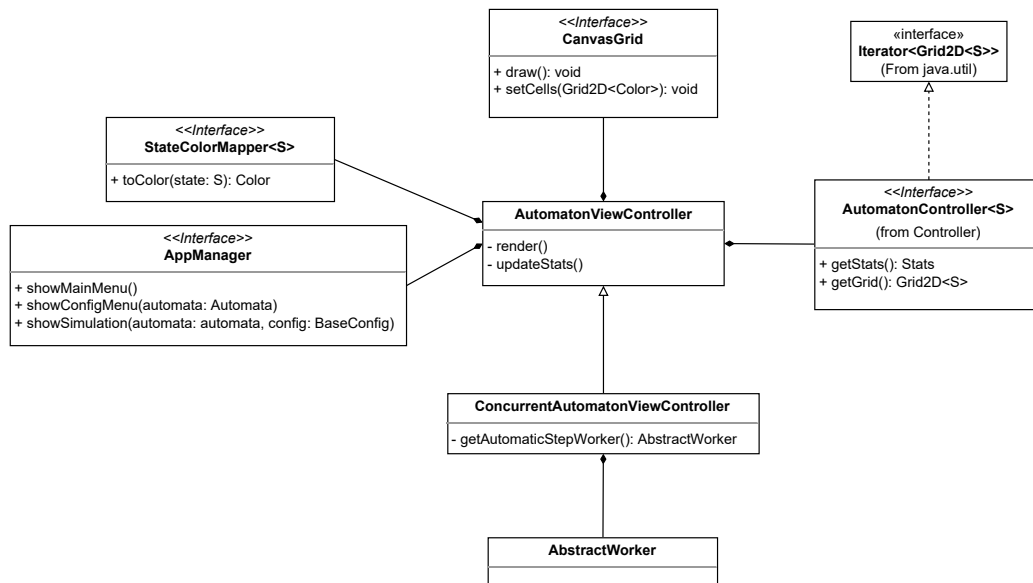


Figura 2.7: Rappresentazione delle principali classi che compongono la view di simulazione. "S" è la enum che rappresenta lo stato della cella dell'automa simulato.

Per la visualizzazione delle varie simulazioni e la gestione della view sono subito state individuate alcune componenti fondamentali e qualche classe di utility che ha permesso di semplificare l'astrazione e la simulazione di automi cellulari differenti.

In particolare, La classe **AppManager** mette a disposizione i metodi per *assemblare* i vari componenti di MVC per poi mostrare le corrispondenti view. Un'altra classe di utility usata è **BaseBuilder** che fornisce i metodi *registerCall*, *checkValue* e *checkNonNullValue* che rispettivamente permettono di:

- *registerCall*: usato per garantire che un metodo NON venga chiamato più volte in modo da evitare stati inconsistenti. Per fare questo viene ispezionato lo stack e se un metodo viene chiamato due volte

- *checkValue* e *checkNonNullValue*: aiutano la validazione degli input lanciando un'eccezione con un messaggio d'errore custom in caso di problemi.

## Gestione delle view

Durante la progettazione della view mi sono reso conto che sarebbe stato necessario avere una classe capace di gestire la possibilità di avere più schermate (per esempio i menu e le view di simulazione), da questo bisogno nasce la classe **PageContainer** che estende un *AnchorPane* fornendo i metodi per aggiungere, rimuovere e nascondere le view. Questi metodi vengono usati per gestire le view con una struttura dati simile ad uno stack, ogni volta che bisogna cambiare una view viene infatti aggiunta sopra alle altre che vengono poi nascoste, quando bisogna tornare indietro viene rimossa la view in cima allo stack e si mostra quella immediatamente sotto. **PageContainer** gestisce in maniera centralizzata il resizing del font in funzione alla dimensione della finestra, in questo modo si mantengono della dimensione giusta i componenti su schermi di risoluzione diversa.

## Controller della view di simulazione

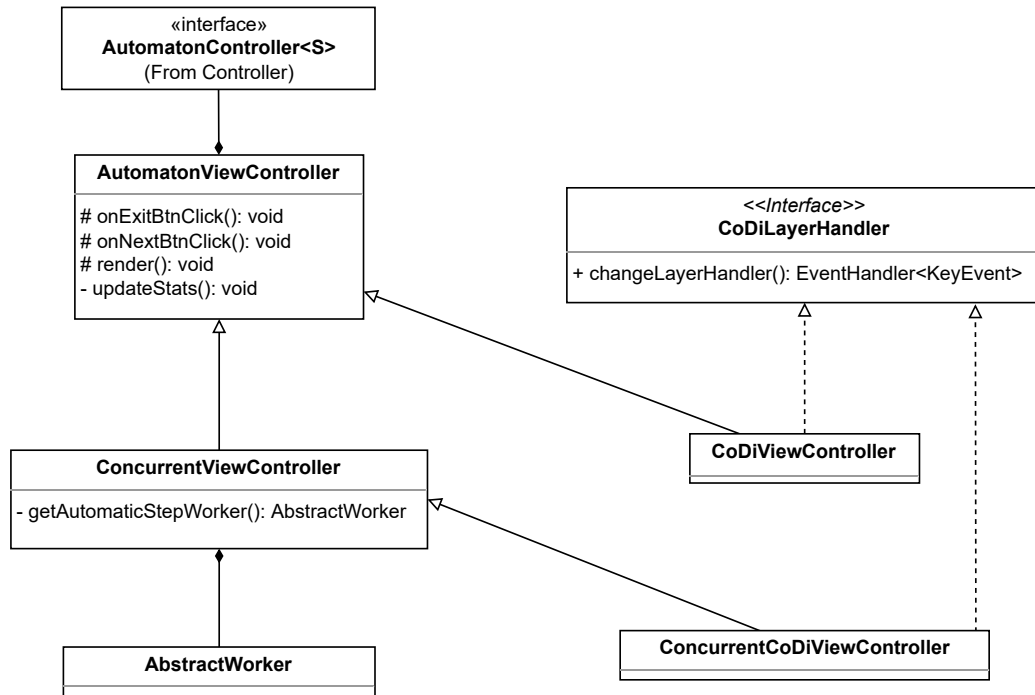


Figura 2.8: Le classi che fanno da controller della view, è stato necessario introdurre funzionalità aggiuntive per CoDi.

Per la gestione delle view FXML si è reso necessario implementare dei view controller, classi che contengono i binding con i componenti della ui di Java-FX e permettono la comunicazione con il controller che fa da ponte tra view e model. Le classi implementate per questo derivano tutte da **Automaton-ViewController** che definisce i metodi principali per gestire i click nella ui, renderizzare il nuovo stato dell'automa e mostrare le statistiche aggiornate.

La simulazione automatica ha richiesto di estendere *AutomatonViewController* in **ConcurrentViewController** che sfrutta un'altro thread per automatizzare le richieste di nuovi stati al controller per poi mostrarli a video. È stato necessario usare una classe custom per il thread di simulazione che permettesse di avviare, fermare e controllare lo stato del thread in modo semplice, funzionalità messe a disposizione da **AbstractWorker** che sfrutta il pattern **Template Method** per permettere di customizzare il comportamento da eseguire.

La simulazione di CoDi ha inoltre richiesto l'aggiunta di altre funzionalità per selezionare il layer da visualizzare, queste view sono state implementate



da Lorenzo Drudi estendendo *ConcurrentViewController* e *AutomatonViewController* e io ho estratto l'interfaccia **CoDiLayerHandler** che ne astrae il funzionamento comune.

Purtroppo per la gestione dei binding di JavaFX tramite i tag *@FXML* non sono riuscito ad usare il pattern decorator come avrei voluto, probabilmente con più tempo si sarebbe potuto trovare una soluzione che promuove il riuso del codice senza dover usare l'ereditarietà tra classi.

## Canvas Grid

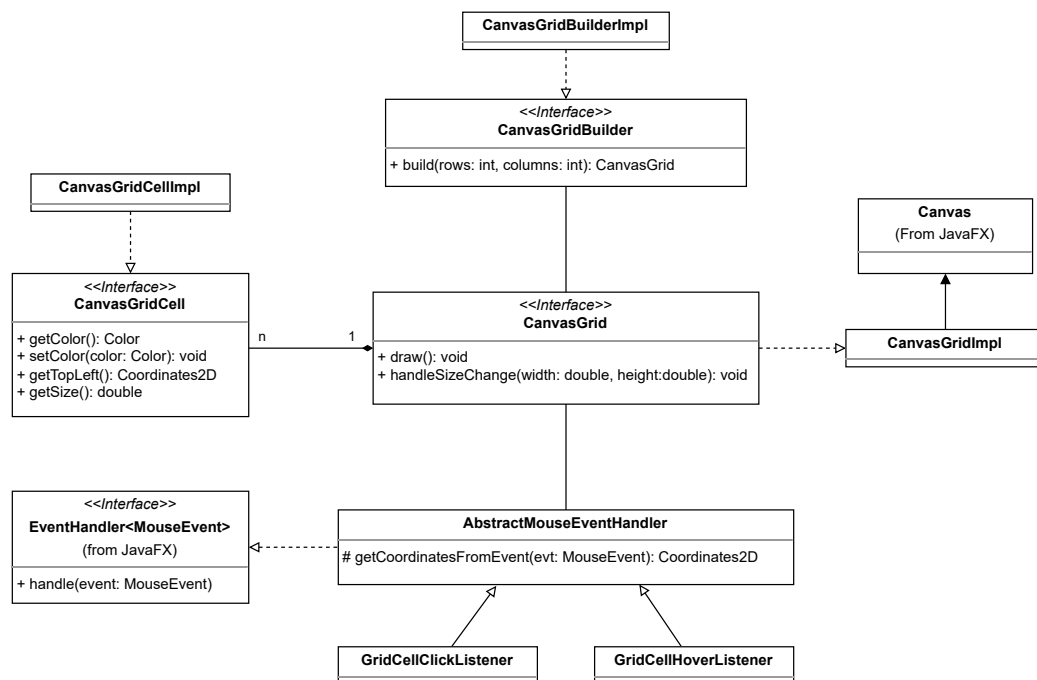


Figura 2.9: Il componente grafico Canvas Grid.

Viene ora presentato il componente grafico più elaborato, *CanvasGrid*, usato per disegnare la griglia di simulazione e progettato per poter interagire con l'utente via mouse.

*CanvasGrid* è formato da una griglia contenente le celle da disegnare, ogni cella è istanza di **CanvasGridCell** e contiene le informazioni necessarie alla visualizzazione che sono dimensioni, posizione e colore. Il componente è stato progettato per gestire gli input da mouse (tramite click e hover), per questo sono stati implementati degli event handler partendo dalla classe **AbstractMouseEventHandler** che fornisce le funzionalità per ottenere la cella selezionata e poi inoltra l'evento alla griglia che lo gestirà.

Per costruire *CanvasGrid* è necessario impostare vari valori, come dimensioni della griglia, delle celle, colore e dimensioni del separatore e alcune di queste configurazioni sono opzionali, per questo è stato creato **CanvasGrid-Builder** che guida e semplifica la creazione degli oggetti di tipo *CanvasGrid* fornendo una *fluent interface* e sfruttando il pattern **builder**.

## Alert Builder

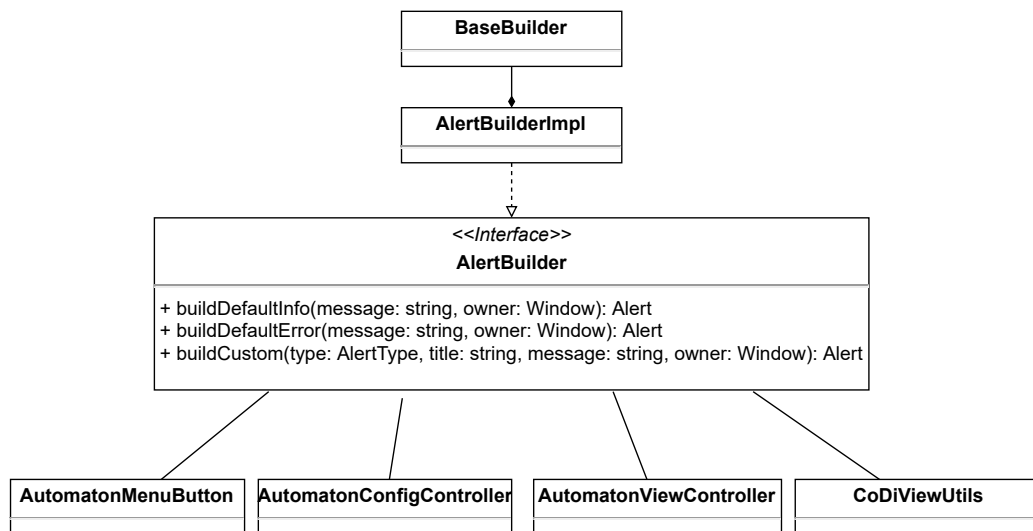


Figura 2.10: L'interfaccia **AlertBuilder**

**Problema** È necessario fornire una classe che guidi il programmatore alla creazione di *alert* di vario tipo (per esempio di errore o di info) facendo in modo che non vengano dimenticati eventuali campi ritenuti obbligatori e allo stesso tempo fornendo metodi che rendano semplice creare gli alert usati più di frequente.

**Soluzione** **AlertBuilder** è stata creata usando il pattern **builder** con una *fluent interface*, in questo modo si possono concatenare le chiamate al builder rendendo il codice più pulito e significativo, inoltre sono stati messi a disposizione due metodi **buildDefaultInfo** e **buildDefaultError** che permettono di velocizzare la creazione di alert di info o di errore.

## Bryan's Brain

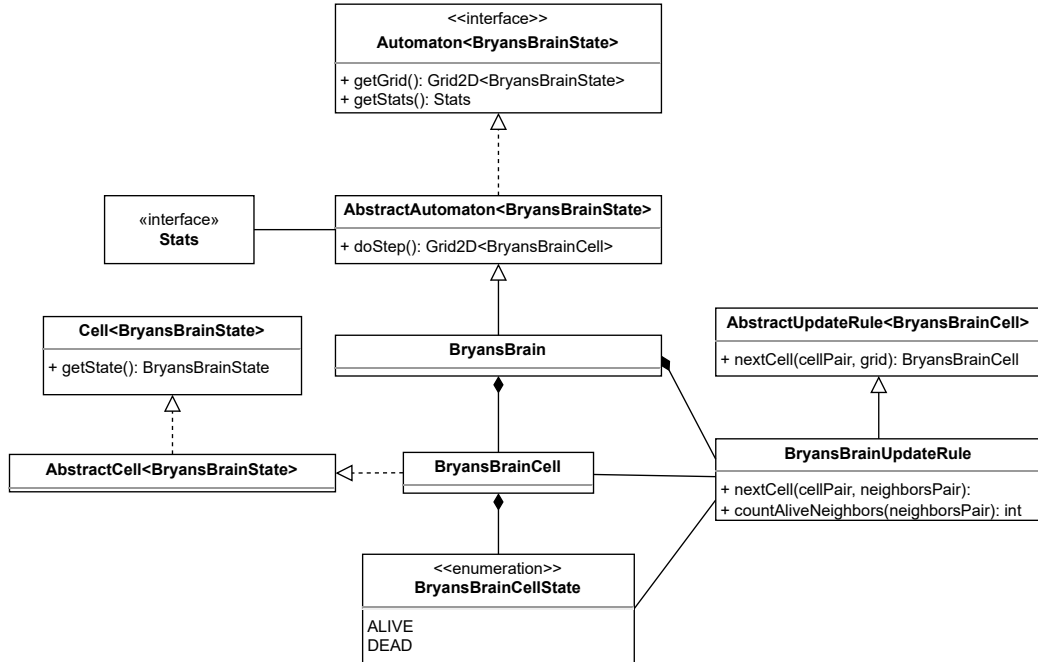


Figura 2.11: L'implementazione di **Automaton** per Bryan's Brain.

L'implementazione di *Bryan's Brain* sfrutta pensantemente l'astrazione presentata in Figura 1.7. Figura 2.11 mostra i componenti principali dell'implementazione.

### 2.2.3 Simone Zama

Durante lo sviluppo di CA-SIM mi sono occupato principalmente della gestione delle coordinate contenute nelle griglie degli automi e le relative utility. Ho poi sviluppato gli automi "Langton's Ant" e "Predators and Preys".

#### Coordinates

Per gestire le coordinate è stata utilizzata un'interfaccia priva di metodi chiamata *Coordinates*. Questa interfaccia è poi implementata dalle classi *Coordinates2D* e *Coordinates3D*. Queste ultime classi utilizzano, come tipi delle coordinate che rappresentano, dei generici che devono estendere la classe *Number*.

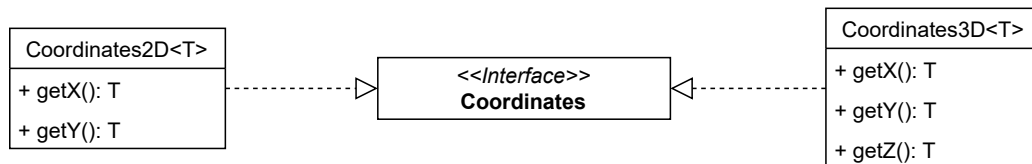


Figura 2.12: L'implementazione di Coordinates.

**Problema** Per la realizzazione di diverse componenti del progetto sono necessarie diverse operazioni per manipolare le coordinate.

**Soluzione** Viene creata una classe finale che comprende tutti i metodi statici di utility necessari denominata **CoordinatesUtil**.

#### Langton's Ant

È importante specificare che questa implementazione di Langon's Ant non sfrutta completamente l'astrazione presentata da *Lorenzo Drudi*, più precisamente non utilizza una estensione della classe *AbstractUpdateRule*. Questo perché le celle all'interno dell'automa non cambiano il proprio stato in base alle celle che hanno vicino, ma è l'entità formica che cambia lo stato della cella in cui si trova. Risulta quindi più semplice implementare la formica non come stato di una cella ma come entità separata, che si trova su una cella di cui cambia lo stato, in questo modo non dovremo calcolare il nuovo stato di ogni cella nella griglia ma solo di quello delle celle in cui si trova almeno una formica; riducendo drasticamente il numero di operazioni effettuate ad ogni iterazione.

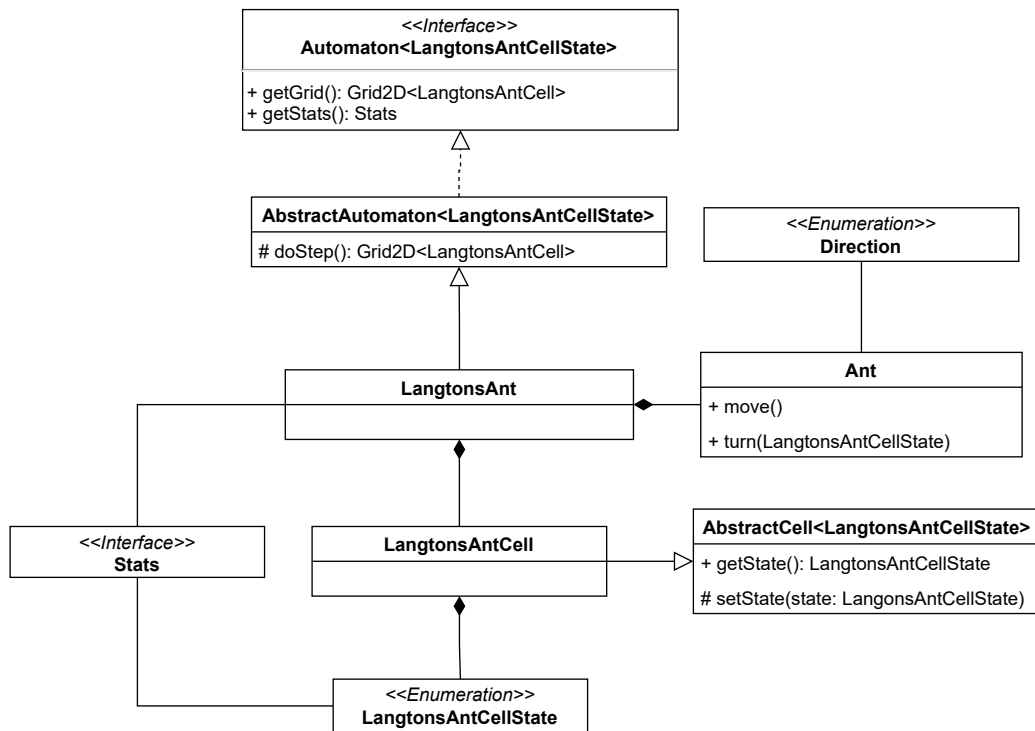


Figura 2.13: L'implementazione di Langton's Ant

**Ant** è la classe che rappresenta una formica, essa mantiene la propria posizione all'interno della griglia dell'automa e la propria direzione. Quest'ultima rappresenta le quattro direzioni cardinali verso cui una formica può muoversi ed è implementata attraverso l'enumerazione **Direction**. La formica è in grado di cambiare la direzione verso cui guarda in base allo stato della propria cella, e può cambiare la propria posizione in base alla direzione verso cui è diretta.

**LangtonsAntCell** è la classe che rappresenta le celle che compongono la griglia dell'automa. Estende *AbstractCell* e mantiene il proprio stato rappresentato attraverso l'enumerazione *LangtonsAntCellState* contenente i due stati esistenti all'interno dell'automa.

**LangtonsAnt** è la classe principale e rappresenta l'intero automa. Essa estende *AbstractAutomaton* e raccoglie al suo interno i precedenti elementi. Ad ogni iterazione esegue le regole di aggiornamento dell'automa per ogni formica presente.

## Predators and Preys

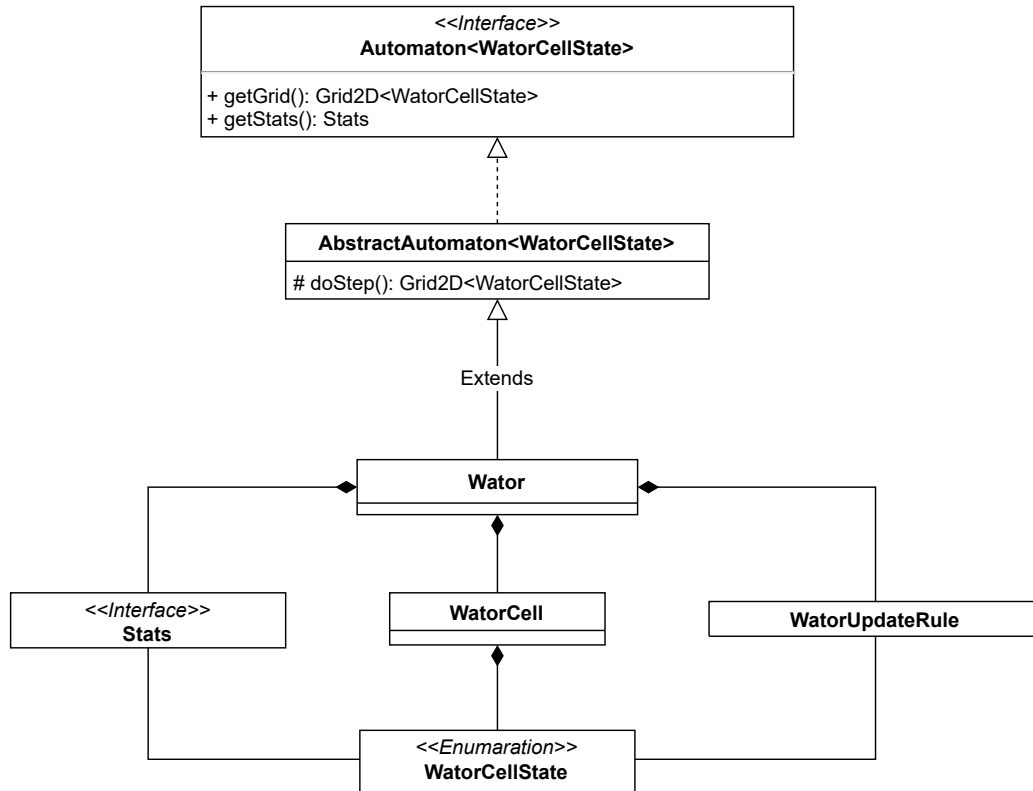


Figura 2.14: L'implementazione di Predators and Preys

**WaterCell** è la classe che rappresenta la cella dell'automa, estende *AbstractCell* e implementa le meccaniche del contatore della salute e della riproduzione. Dato che i predatori e le prede possono muoversi nelle celle vicine questa implementazione necessita di un modo per cambiare lo stato di una cella, viene quindi fornito un metodo che presa in input una cella ne copia lo stato.

**WaterUpdateRule** implementa le regole di aggiornamento dell'automa e estende *AbstractUpdateRule*. A differenza delle regole degli altri automi di CA-SIM, *WaterUpdateRule* agisce direttamente sullo stato dell'automa, non su una sua copia, e oltre a cambiare la cella sulla quale opera può alterare le celle vicine.

**Water** è la classe principale dell'automa, estende *AbstractAutomaton* e aggrega gli elementi dell'automa.

## Configurazione degli automi

Al momento della creazione degli automi sono necessari diversi parametri, di diversa quantità e tipo a seconda dell'automa che si vuole creare.

**Problema** Al momento della loro creazione i costruttori dei diversi automi necessitano di valori di configurazione, alcuni sono comuni per tutti gli automi altri sono presenti solo in alcuni di essi. È quindi necessario semplificare la creazione della configurazione di un automa.

**Soluzione** Per creare la configurazione di un'automa è stato scelto di utilizzare classi contenenti i valori di configurazione. La classe **BaseConfig** rappresenta la configurazione di un automa generico in cui sono presenti solo i valori di altezza e larghezza della griglia, la classe **WrappingConfig** estende *BaseConfig* e introduce il valore utilizzato per indicare il tipo della griglia. Per gli automi *CoDi* e *Langton's Ant* sono necessari ulteriori parametri, in questo caso vengono estese le classi *BaseConfig* e *WrappingConfig* nelle classi **CoDiConfig** e **LangtonsAntConfig** rispettivamente.

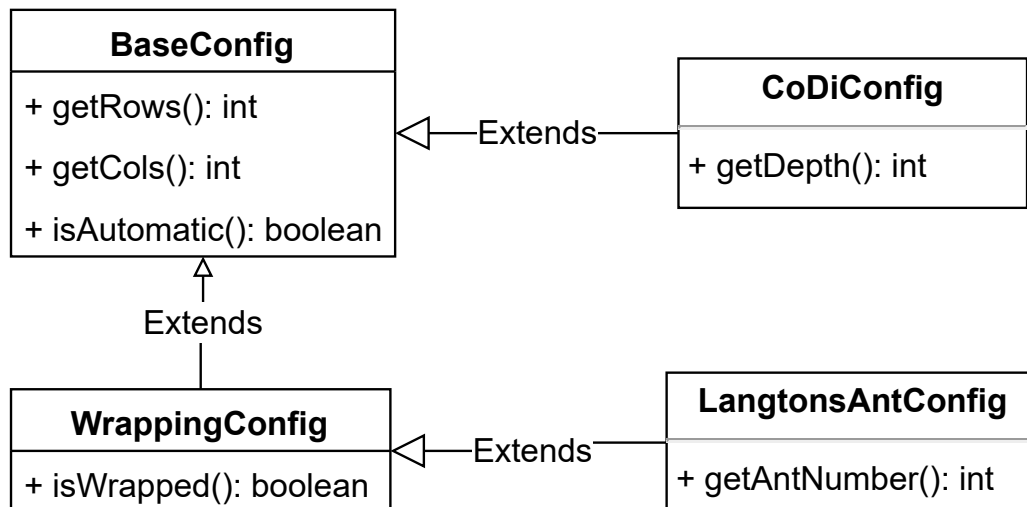


Figura 2.15: L'implementazione delle configurazioni degli automi.

## Costruzione degli automi

**Problema** Durante l'esecuzione di CA-SIM si vuole dare la possibilità di generare diversi automi con configurazioni diverse più volte. È quindi necessario fornire al programmatore un modo semplice di creare i diversi automi.

**Soluzione** È stata creata una **Factory** di automi che permette di creare ogni automa, data la sua configurazione.

**AutomatonFactory** è l'interfaccia che definisce i metodi con cui vengono costruiti nuovi automi, è presente un metodo per ogni automa, viene implementata da **AutomatonFactoryImpl**.

### Costruzione delle mappe stato-colore

**Problema** All'interno della view di CA-SIM è necessario associare ad ogni stato di ogni automa un colore con cui verrà rappresentato.

**Soluzione** Viene fornita una **Factory** tramite l'interfaccia **StateColorMapperFactory** che fornisce per ogni automa un metodo per ottenere il proprio **StateColorMapper**, ovvero l'interfaccia funzionale che associa ad ogni stato un colore. La classe **StateColorMapperFactoryImpl** implementa *StateColorMapperFactory*.

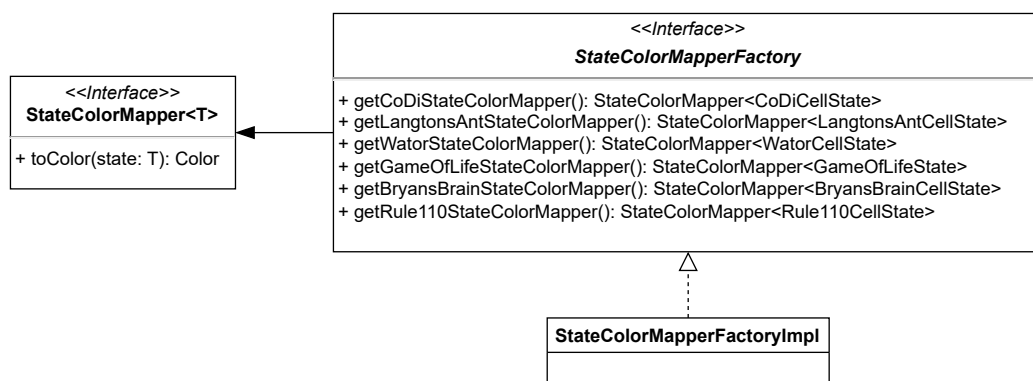


Figura 2.16: L'implementazione di **StateColorMapperFactory**



## 2.2.4 Raul Chiasserini

Nella realizzazione di CA-SIM mi sono occupato della gestione delle griglie e dei controller per il menu e per gli automi. Ho anche implementato gli automi cellulari Conway's Game Of Life e Rule110.

### Grid

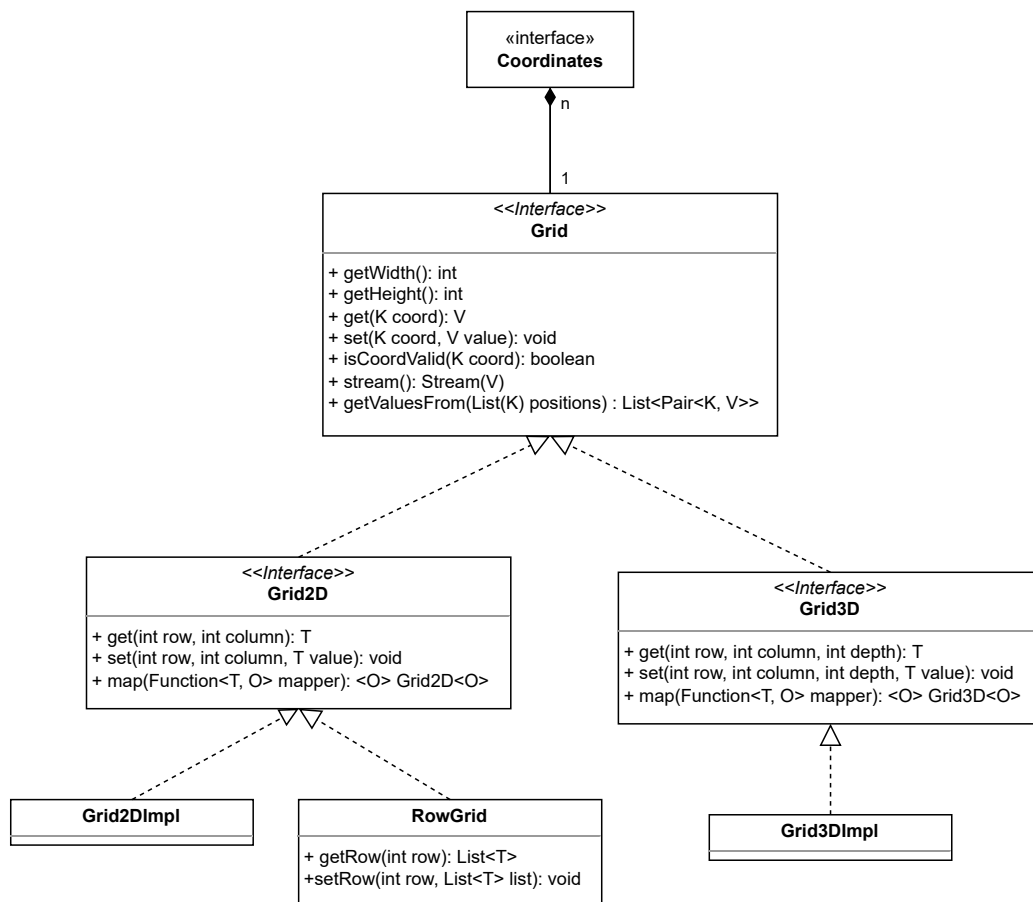


Figura 2.17: UML dello schema di **Grid**.

**Grid** è l'interfaccia principale di tutte le griglie, permette di definire una griglia in cui ogni cella è identificabile tramite una **Coordinates** e può anche contenere un valore iniziale. L'interfaccia contiene getter e setter di base e un metodo di controllo delle coordinate.

## Grid2D

**Grid2D** è un'interfaccia che estende *Grid* e che rappresenta una griglia bidimensionale. **Grid2DImpl** è la sua implementazione, è possibile inizializzare *Grid2DImpl* in modo semplice dando valori di default tramite un *Supplier*. *Grid2D* fornisce inoltre il metodo *map* che è molto utile per ottenere una griglia *trasformata* partendo da quella originale. Questo viene spesso sfruttato quando si vuole passare dalla griglia degli stati delle celle alla griglia dei colori da mostrare (in combinazione con *StateColorMapper*, Figura 2.16).

## Grid3D

**Grid3D** è un'interfaccia che estende *Grid* che rappresenta una griglia tridimensionale. È del tutto simile a *Grid2D*, differisce solo per la terza dimensione e quindi ogni cella di questa griglia avrà anche un parametro *depth*, per questo motivo non vengono usate *Coordinates2D* ma *Coordinates3D*. La sua implementazione **Grid3DImpl** ha la stesse funzionalità di *Grid2DImpl*.

## RowGrid

**RowGrid** è l'implementazione che estende *Grid2D* permettendo di fare get e set di intere righe. È stata implementata secondo il pattern **Decorator**. Si è rivelata utile per *Rule110*, Sezione 1.1.6, che è un automa che aggiorna una riga alla volta controllando solo lo stato di quella precedente.

## AutomatonController

**AutomatonController** è l'interfaccia del controller dell'automa, definisce un metodo *getGrid* per lo stato e uno *getStats* per avere le stats dell'automa. Come *Automaton* estende *iterator* permettendo di chiamare *next* per calcolare ed ottenere il nuovo stato dal model e sfrutta *hasNext* per controllare se la simulazione è terminata. **AutomatonControllerImpl** è la sua implementazione.

## Main Menu

**MenuController** è l'interfaccia usata per gestire la view del controller del menù principale di scelta dell'automa. Definisce un solo metodo *getMenuItems*. **MenuControllerImpl** è la sua implementazione. È importante sottolineare che la view del menu principale è l'unica generata tramite codice e non ha un file FXML associato, questo perché si è pensato che fosse più

semplice visto che la lista di automi da riprodurre è generata dinamicamente dal controller e non sono presenti ulteriori componenti. Inizialmente per ottenere gli automi si è pensato di usare un'annotazione custom ma ci si è poi resi conto che ciò avrebbe complicato inutilmente il codice, si è quindi optato per usare una enumerazione che contiene tutti gli automi.

## Conway's Game Of Life

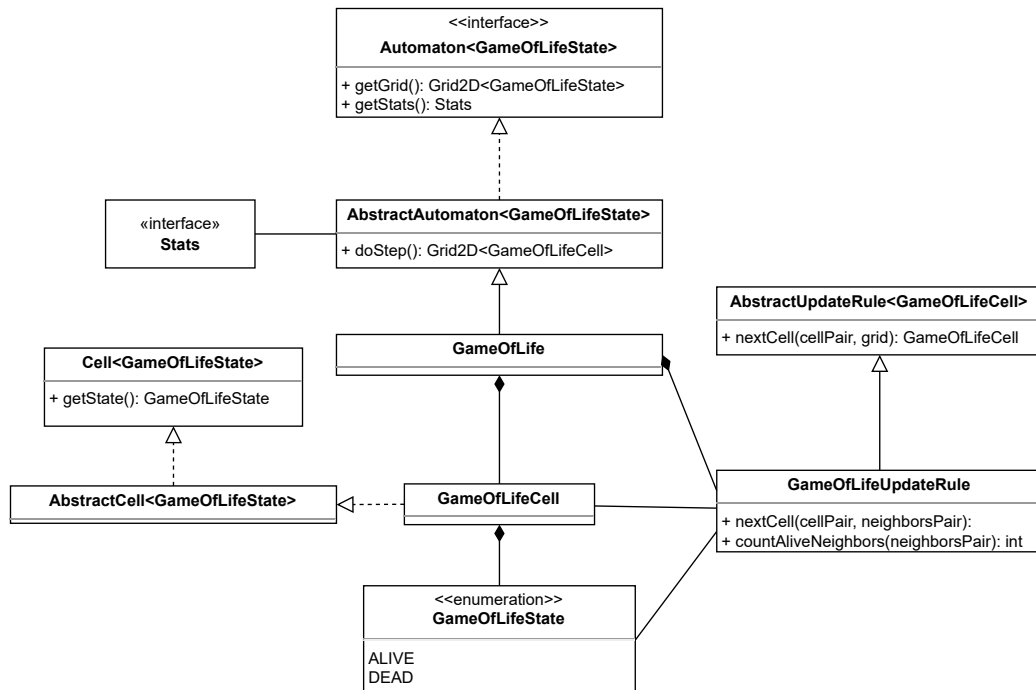


Figura 2.18: UML di **GameOfLife**.

L'implementazione di *Conway's Game of Life* sfrutta pensatamente l'astrazione presentata in Figura 1.7. Figura 2.18 mostra i componenti principali dell'implementazione.

## Rule110

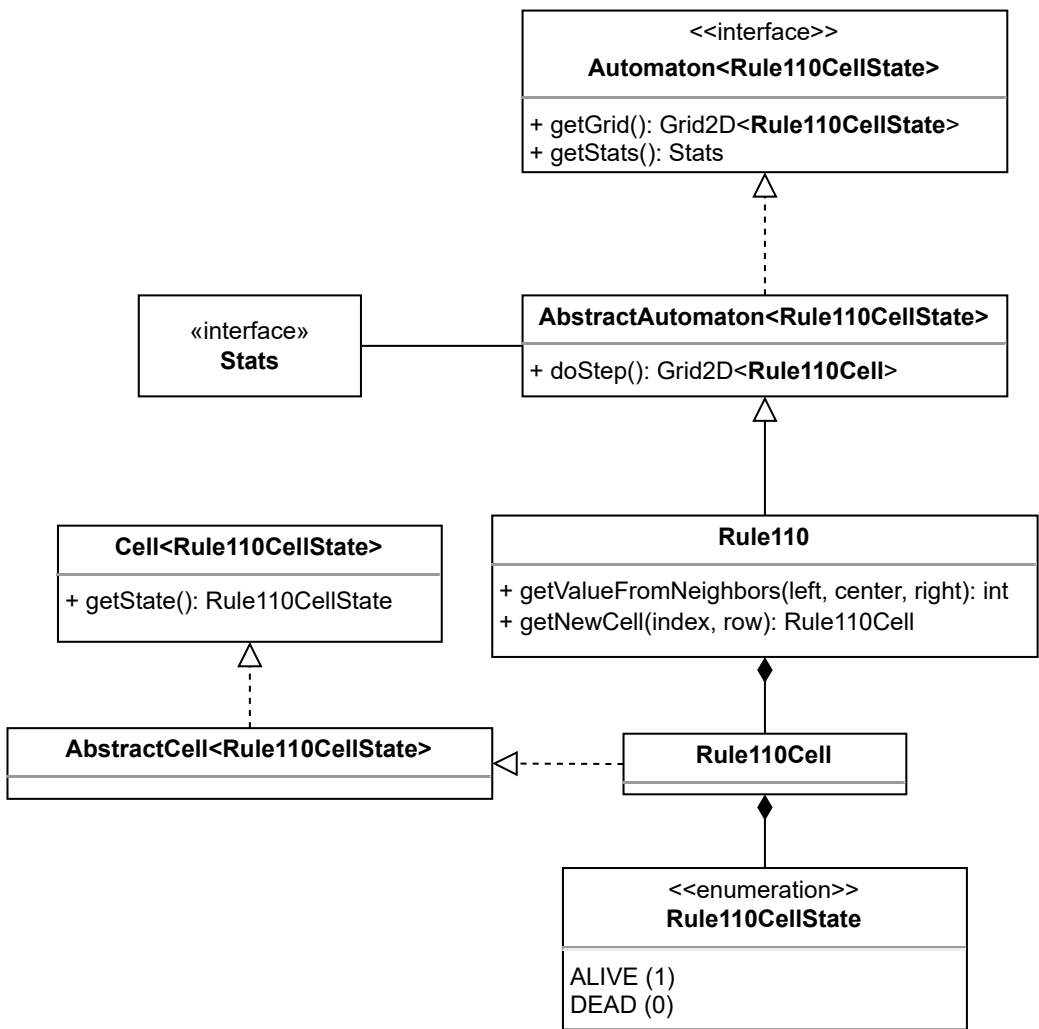


Figura 2.19: UML dello schema di **Grid**.

L'implementazione di Rule110 non segue completamente l'astrazione mostrata in Figura 1.7, mi sono infatti reso conto che l'uso di update rule avrebbe complicato inutilmente l'implementazione dell'automa, si sarebbe infatti dovuto trovare un modo per indicare la posizione dei in relazione alla cella da aggiornare in un determinato momento. Per ovviare a questo problema è stata sfruttata la classe *RowGrid* per operare sulle singole righe della griglia iterando su ogni cella e chiamando il metodo *getNewCell* per ottenere il valore della prossima cella in base alla sua posizione e ai suoi vicini nella riga prima. Un altro accorgimento usato per rendere il codice più pulito è

stato il seguente: assegnando il valore 1 alle celle *ALIVE* e 0 a quelle *DEAD* ogni terna rappresentata dalla cella da aggiornare e dai suoi due vicini può essere rappresentata con un intero in  $[0, 7]$ , si possono poi mappare i valori in questo range al corrispondente stato della prossima cella generata con una map.

# Capitolo 3

## Sviluppo

### 3.1 Testing automatizzato

All'interno di CA-SIM il testing automatizzato è stato utilizzato principalmente per testare classi di utility. Esso è stato realizzato tramite il framework JUnit 5.

#### 3.1.1 Lorenzo Drudi

- PageContainerTest
- NeighborsFunctionsTest
- CoDiCellTest
- CoDiCellBuilderTest
- CoDiCellSupplierTest
- CoDiCellFunctionTest

#### 3.1.2 Filippo Sanzani

- RangeTest
- ResultTest

#### 3.1.3 Simone Zama

- Coordinates2DTest

- Coordinates3DTest
- CoordinatesUtilTest
- AntTest
- WatorCellTest

### 3.1.4 Raul Chiasserini

- Grid2DTest
- Grid3DTest
- RowGridTest

## 3.2 Metodologia di lavoro

### 3.2.1 Lorenzo Drudi

- Realizzazione dell'astrazione sulla quale poi sono stati implementati tutti gli automi cellulari: **model.abstraction.\***;
- Implementazione di CoDi:
  - Model: **model.codi.\***;
  - Controller: **CoDiControllerImpl**;
  - View: **CoDiViewController** e **ConcurrentCoDiViewController**;
- Menù di configurazione degli automi: **AutomatonConfigController** e **AutomatonWrapConfigController**.
- **NeighborsFunctions**

Ho inoltre contribuito a:

- **Grid3D** migliorando le prestazioni dopo aver notato problemi di performance nella simulazione di CoDi;
- **StateColorMapperFactory**;
- **PageContainer**.

### 3.2.2 Filippo Sanzani

- Realizzazione del componente grafico *CanvasGrid*: **casim.ui.components.grid.\***;
- Realizzazione della view di simulazione **casim.ui.view.\*** (La parte di CoDi è stata progettata, realizzata ed implementata da Lorenzo Drudi, io ho solo estratto l'interfaccia comune);
- Astrazione del menu principale: **casim.ui.menu.\***;
- Classi di utility (dentro *casim.utils*):
  - Range: **casim.utils.range.\***;
  - **AbstractWorker**;
  - **BaseBuilder**;
- **AppManager**;
- **PageContainer**;
- **AlertBuilder**;
- Bryan's Brain: **casim.model.bryansbrain.\***.

Ho inoltre, in piccola parte contribuito a:

- **NeighborsFunctions.java**;
- **AutomatonFactory**.

### 3.2.3 Simone Zama

- Classi di utility:
  - Coordinates: **casim.utils.coordinate.\***;
  - Configs: **casim.utils.automaton.config.\***;
  - Direction: **casim.utils.Direction**;
- Implementazione di Langton's Ant **casim.model.langtonsant.\***
- Implementazione di Predators and Preys **casim.model.wator.\***
- **AutomatonFactory** **casim.utils.automaton.\***
- **StateColorMapperFactory** **casim.ui.utils.statecolormapper.\***

Ho inoltre contribuito a:

- **BaseBuilder**



### 3.2.4 Raul Chiasserini

- Intefacce e implementazioni delle griglie: **casim.utils.grid.\***;
- Implementazione di Conway's Game of life: **casim.model.gameoflife.\***
- Implementazione di Rule110: **casim.model.rule110.\***
- AutomatonController: **casim.controller.automaton.\***
- MenuController: **casim.controller.menu.\***
- Menu Principale (sfruttando l'astrazione di *Filippo Sanzani*)

## 3.3 Note di sviluppo

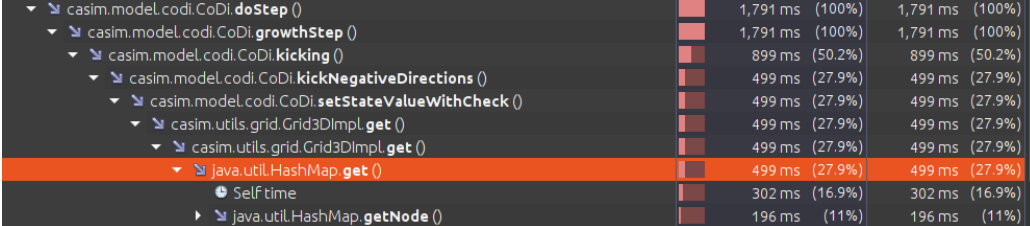
Come approccio di lavoro abbiamo scelto di partire con un attenta ed approfondita analisi del dominio per arrivare, tramite il confronto di idee, ad una descrizione del software tramite UML che potesse poi aiutarci durante lo sviluppo. Centrale è stato l'utilizzo di **GIT**. Abbiamo adoperato una metodologia basata su *Git Flow* con il branch "master" come branch principale ed il branch "develop" con le modifiche apportate in vista della successiva release. Da quest'ultimo branch sono stati creati poi diversi feature branch ogni qual volta un membro dovesse lavorare su una nuova feature a se stante. Ciò ha permesso lo sviluppo del software in maniera isolata e del tutto controllata. Talvolta, nelle situazioni più ostiche, al fine di trovare la soluzione ottimale, è stato necessario un confronto di gruppo.

### 3.3.1 Lorenzo Drudi

- **Optional** in tutte le situazioni in cui sono presenti campi che possono anche non contenere un valore, il loro utilizzo mi ha permesso di non usare *null* all'interno del codice;
- **Stream** utilizzati per la visitazione di strutture dati;
- **Lambda** ogni qual volta è stata utilizzata una functional interface le lambda sono risultate fondamentali per rendere il codice più chiaro e pulito;
- **JavaFx** framework utilizzato per lo sviluppo del menu di configurazione;

- **Apache commons** libreria utilizzata quando necessitavo di utilizzare delle Pair, la scelta è stata proprio quella di sfruttare l'implementazione che questa libreria offre;
- **Generici** all'interno di CA-SIM il loro utilizzo è stato fondamentale per progettare un'astrazione che si potesse adattare a tutti gli automi, sono stati fondamentali per massimizzare il riutilizzo di codice;
- **Wildcard** usate per rendere più flessibile la gestione dei generics nel model;
- **VisualVM** software utilizzato per eseguire profiling per l'automa CoDi. Basandosi su una griglia 3D rispetto agli altri automi risultava molto lento ed è stato necessario analizzare l'esecuzione per trovare i punti in cui l'automa era più lento.

Analizzando l'esecuzione di CoDi mi sono reso conto che questo era il punto che maggiormente rallentava l'esecuzione. Si può notare come la maggior parte del tempo sia speso facendo accessi ad una HashMap. Questo perché la griglia 3D era implementata tramite questa struttura dati. Confrontandomi anche con i miei colleghi abbiamo deciso di passare ad una griglia implementata attraverso liste concatenate con l'obiettivo di migliorarne le performance.



casim.model.codic.CoDi.doStep ()	1,791 ms (100%)	1,791 ms (100%)
casim.model.codic.CoDi.growthStep ()	1,791 ms (100%)	1,791 ms (100%)
casim.model.codic.CoDi.kicking ()	899 ms (50.2%)	899 ms (50.2%)
casim.model.codic.CoDi.kickNegativeDirections ()	499 ms (27.9%)	499 ms (27.9%)
casim.model.codic.CoDi.setStateValueWithCheck ()	499 ms (27.9%)	499 ms (27.9%)
casim.util.grid.Grid3DImpl.get ()	499 ms (27.9%)	499 ms (27.9%)
casim.util.grid.Grid3DImpl.get ()	499 ms (27.9%)	499 ms (27.9%)
java.util.HashMap.get ()	499 ms (27.9%)	499 ms (27.9%)
Self time	302 ms (16.9%)	302 ms (16.9%)
java.util.HashMap.getNode ()	196 ms (11%)	196 ms (11%)

Figura 3.1: Esecuzione di **CoDi** con **Grid3D** implementata tramite *HashMap*.

Tramite questo screenshot è possibile notare il marcato miglioramento di performance successivo alla nuova implementazione della griglia.



- **Wildcard** usate per rendere più potente e flessibile la gestione dei generics nelle astrazioni del model, della view e delle griglie;
- **VisualVM** è stato usato per fare profiling e del software, si è notato che l'implementazione di *Grid2D* con **Map** e si è deciso di reimplementarla con **List**, vedere sopra per screenshot di una situazione analoga con *Grid3D*;
- **Multithreading** è stato usato per permettere la simulazione automatica degli automi lato view-controller.

### 3.3.3 Simone Zama

- **Stream** utilizzati per la visita di strutture dati;
- **Lambda** utilizzate per snellire e semplificare il codice nell'utilizzo di functional interface, spesso usate insieme a gli stream;
- **Apache Commons** utilizzo di Pair e Triple nell'astrazione delle coordinate;
- **Generici bounded** utilizzati nell'astrazione delle coordinate.

### 3.3.4 Raul Chiasserini

- **Stream** utilizzati per la visita di strutture dati;
- **Lambda** utilizzate per avere un codice più chiaro e semplice nell'utilizzo di functional interface e insieme agli stream;
- **Apache Commons** libreria scelta per l'implementazione che offre di Pair;
- **Generici** sono stati fondamentali per progettare del codice che si potesse adattare e riutilizzare senza problemi a tutti gli automi;
- **Wildcard** sono state utilizzate per rendere più facile e flessibile la gestione dei generics nelle griglie.

## Codice preso da terze parti

- **Result**: [RES];
- **Empty**: [EMP].

# Capitolo 4

## Commenti finali

### 4.1 Autovalutazione e lavori futuri

#### 4.1.1 Lorenzo Drudi

Questo primo progetto è stato utile per incrementare le mie capacità di team-working oltre che per accrescere le skills riguardanti lo sviluppo di un progetto. Sono infatti riuscito a rafforzare le mie skills nell'utilizzo di git oltre che ad apprendere numerose good practice che permettono uno sviluppo migliore ed anche più efficiente. Soprattutto sono contento di essermi occupato della parte di astrazione di un automa cellulare, il suo sviluppo non è stato banale e mi ha permesso di accrescere le mie capacità di progettazione. Per quanto riguarda i punti deboli devo dire che più volte all'interno del progetto mi è capitato di prendere scelte che poi non si sono rivelate le migliori, spesso erano presenti dettagli a cui inizialmente non pensavo. Sicuramente questo è dovuto alla poca esperienza ed una visione quindi del problema non ampia quanto, in alcune situazioni, servirebbe. Sono certo che questo migliorerà anche grazie a progetti come quello portato a termine per OOP.

#### 4.1.2 Filippo Sanzani

Questo è stato per me il primo progetto di dimensioni medie che ho creato da zero, in passato mi è capitato di lavorare in azienda su software scritti da altri ma non ho mai avuto tempo di approfondire certe scelte di design, questo progetto mi ha permesso di avere modo di sbagliare ed di imparare durante il processo di ideazione, implementazione e testing del software dandomi un'idea più completa di una parte importante del ciclo di vita del software. Durante lo svolgimento del progetto ho imparato ad organizzare il mio lavoro in funzione a quello degli altri, sfruttando tool come git per esem-

pio e comunicando per fare in modo che nessuno fosse bloccato in attesa del lavoro di altri. Questo processo di apprendimento è avvenuto per *trial and error*, non è quindi sempre stato un qualcosa di lineare ma abbiamo incontrato difficoltà, per esempio ho sottostimato il lavoro necessario a completare determinate funzionalità, inoltre mi sono reso conto che siamo partiti con un design più complicato del necessario partendo con l'idea di applicare pattern al problema (cercando quindi di adattare i problemi al pattern e non al contrario). Questo ha portato a dover spendere tempo aggiuntivo a sistemare il design ma è stato importante per capire cosa non fare in futuro.

### 4.1.3 Simone Zama

Questa è stata la mia prima esperienza in cui ho partecipato alla progettazione di un software, e come tale mi ha permesso di imparare nella pratica le fasi della progettazione e dello sviluppo. Durante il progetto ho avuto la possibilità di approfondire l'utilizzo di git, e soprattutto ho potuto migliorare le mie abilità di programmatore java, nello specifico nell'utilizzo di stream e lambda all'interno del codice. Questo progetto mi ha inoltre permesso di imparare come lavorare all'interno di un team di sviluppo, organizzando il proprio lavoro in sincrono con gli altri membri del team per ottimizzare i propri risultati. Alcune delle scelte che ho effettuato durante lo sviluppo del progetto si sono rivelate non ottimali, il che mi ha fatto capire l'importanza di una corretta analisi e progettazione da effettuare precedentemente all'effettiva implementazione. Penso che il mio contributo al gruppo di lavoro sia stato utile al raggiungimento dell'obiettivo finale, in quanto ho partecipato alla progettazione e condiviso le mie opinioni sulle diverse implementazioni. Per quanto riguarda il lavoro svolto da me nello sviluppo del progetto penso di aver dato buoni risultati, soprattutto nelle parti sviluppate in team; ammetto però di non aver sfruttato pienamente i vantaggi della programmazione ad oggetti in alcune delle implementazioni che ho sviluppato personalmente e di avere commesso alcuni errori nell'utilizzo di git.

### 4.1.4 Raul Chiasserini

Questa è stata la prima esperienza con un progetto di queste dimensioni, questo mi ha aiutato molto nell'imparare ad utilizzare tool importanti per il lavoro in team come git e mi ha anche aiutato a comprendere meglio aspetti della programmazione java che non avevo esplorato abbastanza in precedenza come l'utilizzo delle lambda e degli stream. Lavorare in un progetto come questo mi ha anche aiutato a capire quanto sia fondamentale tutta la parte precedente alla scrittura del codice come l'organizzazione, la progettazione

e la divisione dei compiti. Per quanto riguarda la mia parte sono contento di aver lavorato sia nello sviluppo di automi sia nella parte riguardante le griglie, ho infatti avuto modo di fare una parte indipendente ed una parte in cui mi sono coordinato con gli altri membri del gruppo per implementare codice utile a tutti. Purtroppo per motivi lavorativi e personali non ho potuto partecipare al progetto quanto avrei potuto e voluto, soprattutto nella parte di progettazione, questo mi ha rallentato molto nel capire la struttura del progetto. Quest'ultimo fattore unito alla poca esperienza sono state le difficoltà maggiori.

# Appendice A

## Guida utente

### A.1 Le view

L'utilizzo del software dovrebbe essere abbastanza immediato, vengono di seguito descritte rapidamente le varie view presenti:

- Main menu: permette la scelta dell'automa da simulare;
- Configuration menu: permette di scegliere che impostazioni usare per la simulazione;
- Simulation view: permette di visualizzare la simulazione.

#### A.1.1 Configurazione degli automi

Nella view di configurazione vengono richiesti i seguenti parametri:

- Size: la dimensione della griglia (size x size);
- Modalità di simulazione: se *Manual* si avrà una simulazione manuale, l'utente dovrà quindi clickare next per ottenere il prossimo stato, *Automatic* prevede invece una simulazione automatica;
- Wrapping: Determina se usare una griglia che permette di propagare gli stati delle celle oltre i bordi della griglia rientrando dalla parte opposta, per esempio la riga "sopra" alla prima sarà l'ultima mentre quella a sinistra della prima colonna sarà l'ultima colonna e viceversa.

*Si consiglia di non usare Size troppo grandi, altrimenti l'applicativo potrebbe rallentare fortemente, si consigliano valori minori di 500 per tutti gli automi tranne CoDi, per il quale consigliamo valori minori di 60. Questi valori sono stati presi valutando le prestazioni sui nostri PC e potrebbero variare su altre macchine.*



### **A.1.2 CoDi**

CoDi ha una funzionalità particolare in fase di simulazione, essendo un automa che ha uno stato in tre dimensioni è possibile scegliere quale layer mostrare usando i tasti A (per tornare in dietro di un layer) e D (per avanzare di un layer).

# Appendice B

## Esercitazioni di laboratorio

### B.0.1 Lorenzo Drudi

- Laboratorio 05: <https://virtuale.unibo.it/mod/forum/discuss.php?d=87881#p135235>
- Laboratorio 06: <https://virtuale.unibo.it/mod/forum/discuss.php?d=87880#p135201>
- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p136361>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=89272#p137137>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p138404>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=91128#p139598>

### B.0.2 Filippo Sanzani

- Laboratorio 06: <https://virtuale.unibo.it/mod/forum/discuss.php?d=87880#p135136>
- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p136322>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=89272#p137099>

- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p138323>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=91128#p139490>

### **B.0.3 Simone Zama**

- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p136814>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p138361>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=91128#p140278>

# Bibliografia

[EMP] nwoolcan - empty.java.

[RES] nwoolcan - result.java.