ALMA MATER STUDIORUM
Universitá degli studi di Bologna

SCUOLA DI SCIENZE - CAMPUS DI CESENA
Corso di Laurea in Ingegneria e Scienze Informatiche

SVILUPPO DI UN SIMULATORE PER LA PIATTAFORMA SCAFI

Elaborato in
PROGRAMMAZIONE AD OGGETTI

Presentata da:
Chiara Varini

Relatore:
Prof. Mirko Viroli

Correlatore:
Dott. Roberto Casadei

Seconda Sessione di Laurea
Anno accademico 2015-2016

Al mio nonno Nello e

alla mia fantastica famiglia,

grazie per avermi accompagnato fin qui

<u>tutti</u> insieme.

Indice

Introduzione

Nel corso degli ultimi anni la tecnologia ha assunto un ruolo sempre più rilevante nella vita delle persone, diventando ormai parte integrante delle loro interazioni col mondo: gli smartphone e i tablet, come primi esempi, sono tecnologie delle quali non si può più fare a meno e che all'occorrenza diventano vere e proprie finestre sul mondo. Come risultato dell'esponenziale avanzamento tecnologico, l'utilizzo di questi dispositivi è diventato ormai automatico ed immediato, quasi insostituibile per la maggior parte delle esigenze del nostro quotidiano. Il concetto di *Pervasive Computing* nasce proprio allo scopo di corrispondere a tali nuove necessità ed esigenze e, non limitandosi solamente allo sviluppo di personal computer o smartphone, possiede un respiro molto più ampio e che arriva sino ad inglobare gli oggetti utilizzati quotidianamente come orologi, macchine, indumenti ecc. Tali oggetti, dotati di componentistica HW e SW, possono quindi diventare veri e propri dispositivi "intelligenti", che adeguatamente settati, possono comunicare ed interagire tra di loro, creando una vera e propria rete di oggetti, la cosiddetta *Internet of Things*. Tuttavia, lo scopo del pervasive computing trascende la semplice interconnessione tra oggetti, e si traduce nell'obbiettivo più ambizioso di creare un ambiente intelligente in cui gli oggetti forniscono determinati servizi in ogni momento, contribuendo pertanto ad un netto miglioramento della qualità della vita. In questo ambiente le diverse tecnologie comunicano, collaborano e gestiscono informazioni adattandosi autonomamente così da svolgere il proprio compito nel contesto in cui si trovano. Questi sistemi possono essere di varia natura, dai più semplici, come il pilotaggio di luci a seguito del rilevamento di alcuni sensori in un ambiente, a sistemi più complessi che si appoggiano su un alto numero di device. Per quest'ultimi, non essendo abbastanza espressivi i classici paradigmi di programmazione, è nata la necessità di nuovi modelli che permettono di descriverli e progettarli ad un livello aggregato, in cui è possibile esprimere il comportamento del program-

ma in sé, senza perdersi nella descrizione dettagliata degli assignments di ogni singolo device. L'*Aggregate Programming* è un paradigma basato sul *fields calculus*, che permette di programmare un insieme di dispositivi in termini del loro comportamento aggregato. Un framework che implementa tale logica è *Scafi*, il quale offre una serie di servizi molto utili per la programmazione di applicazioni aggregate. Esso mette a disposizione una libreria Scala che realizza la semantica del field calculus in modo corretto e completo, una piattaforma distribuita Akka-based su cui sviluppare applicazioni, e l'esposizione di un'API generale e flessibile in grado di supportare diversi scenari. La progettazione di sistemi distribuiti aggregati ha aperto nuovi scenari per i programmatori del XXI secolo. Il programma che si ottiene implementando applicazioni di tale natura, non risulta puramente deterministico, ma il risultato finale può essere modificato da variabili diverse. Il comportamento di questi sistemi non è quindi definibile a priori, per cui risulta necessario effettuare del testing (sia automatizzato che non) in maniera fortemente accurata. Al fine di velocizzare tale procedimento è necessario utilizzare degli ambienti di sviluppo adatti, in cui poter testare le proprie applicazioni aggregate, verificandone il corretto funzionamento. Oltre a tutto ciò, è importante tenere in considerazione la necessità di un programma in grado di riprodurre graficamente le dinamiche del sistema aggregato testato, così da poter creare demo fruibili anche da parte di utenti non esperti. L'utilizzo di un simulatore grafico dedicato che permetta di testare agilmente e in modo completo il corretto comportamento delle applicazioni, appare quindi una necessità oltre che una vera e propria opportunità.

Il lavoro di tesi muove i suoi passi proprio dalle esigenze appena evidenziate e dunque lo scopo è quello di creare un simulatore per la piattaforma Scafi. Tuttavia, il simulatore è stato sviluppato in maniera completamente indipendente da Scafi. Tale strategia di sviluppo è stata adottata per rispondere all'eventuale esigenza futura di essere in grado di riprodurre anche il comportamento di applicazioni aggregate scritte con altri framework (ad esempio *Protelis*). La realizzazione del progetto si è sviluppata seguendo un processo iterativo ed incrementale, raffinando in maniera progressiva sia i requisiti che il codice. I requisiti di partenza sono incentrati sull'interfaccia grafica (rappresentazione e sovrapposizione dei nodi, gestione delle trasparenze, visualizzazione di immagini ecc..). L'implementazione della view si è rivelata la parte più dispendiosa a livello di tempo, in quanto è anche quella che ha richiesto di soddisfare il maggior numero di prerequisiti. Una volta che l'ambiente grafico è stato com-

pletamente strutturato, si è passato alla creazione di un modello apposito, tale da rispecchiare appieno le proprietà utilizzate per la creazione e la gestione di un'applicazione aggregata. Al fine di poter consentire la simulazione di diversi sistemi, all'interno del model sono state definite solamente le interfacce, che servono da API di collegamento con qualsiasi sistema rappresentabile. Ovviamente tale collegamento, che nel progetto prenderà il nome di *Bridge*, dovrà essere implementato appositamente per ogni specifica simulazione desiderata. Il risultato finale a cui si è puntato, è quello di rendere il più chiara e veloce possibile la rappresentazione di un qualsiasi sistema aggregato. Quindi, una volta definito il codice dell'applicazione distribuita utilizzando le API Scafi, sarà possibile importare il progetto sul simulatore e se ne potrà testare agilmente il corretto funzionamento. Allo stesso tempo si potrà anche ottenere una rappresentazione grafica intuitiva, semplice e pulita, in cui sono bene distinguibili tutti i diversi elementi che compongono il sistema.

Nel seguito della trattazione saranno riportate in maniera esauriente tutte le fasi di analisi, modellazione ed implementazione del simulatore suddivise secondo la seguente logica organizzativa:

- Introduzione alle tecnologie utilizzate:
  - Scala;
  - Interoperabilità tra Scala e Java;
  - Scafi.
- Presentazione del progetto:
  - Analisi dei requisiti;
  - Design;
  - Implementazione.
- Conclusione e risultati ottenuti:
  - Strumenti utilizzati;
  - Guida utente;
  - Sviluppi futuri.

Capitolo 1

Scala

Il particolare nome del linguaggio Scala ne esplicita apertamente l'aspetto caratterizzante, poiché deriva dall'unione dei termini "Scalable" e "Language". Essi, uniti insieme, esprimono il concetto di "Scalable Language" che, nomen omen, sottolinea la proprietà di scalabilità intrinseca di Scala. Per linguaggio scalabile si intende infatti la possibilità di adattarsi (scalare) alle esigenze dei programmatori, ossia poterlo utilizzare per semplici progetti oppure, mediante l'utilizzo di costrutti più sofisticati, per altri più grandi e complicati. Scala incorpora le proprietà del paradigma di OOP e, garantendo una adeguata strutturazione dell'intera applicazione, viene utilizzato anche per modellare grandi sistemi. Oltre ai paradigmi della programmazione ad oggetti, esso ingloba anche i principi della programmazione funzionale, permettendo di scrivere in modo semplice e conciso piccole parti di codice. Ad esempio, grazie all'utilizzo di feature funzionali, verificare se una determinata stringa contiene un carattere maiuscolo o no in Java richiederebbe l'utilizzare del costrutto *for*, così da scannerizzare ogni singolo carattere contenuto e verificarne l'eventuale matching, mentre in Scala è possibile inglobare questo procedimento all'interno di una particolare funzione che prende come argomento un predicato: _.isUpper. Queste ottimizzazioni garantiscono un'elevata velocità di scrittura del codice per cui esso risulta drasticamente ridotto, se confrontato con i tradizionali linguaggi ad oggetti. Nella Figura 1.1 sono riportati i codici Java e Scala dell'esempio appena descritto.

Sebbene il codice Scala riportato in figura possa rimandare alle *lambda* presenti in Java 8 (ossia particolari costrutti da utilizzare in accoppiamento con le interfacce funzionali, ad esempio ActionListener), la possibilità di utilizzare

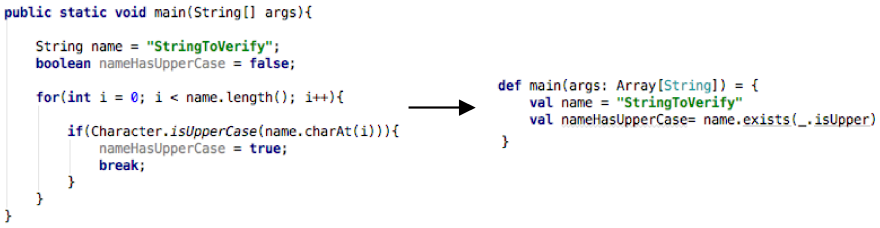


Figura 1.1: A sinistra codice Java e a destra codice Scala.

funzioni come valori e/o variabili risulta ancora una volta più veloce e semplice, eliminando il bisogno di dichiarare interfacce funzionali. L'unione di queste due caratteristiche, conferisce a Scala un'elevata flessibilità, permettendo di adattarlo ai più complessi ambiti applicativi e, allo stesso tempo, risultando efficace anche per piccole applicazioni. Tali caratteristiche contribuiscono a rendere Scala un linguaggio di programmazione adatto per un'ampia rosa di possibili utilizzi.

Dal punto di vista sintattico, si nota una forte assonanza con i linguaggi di scripting come ad esempio Python, poiché mirando alla minimizzazione del codice, puntano ad eliminare tutti i formalismi superflui come i punti e virgola e la dichiarazione dei tipi. Questo particolare aspetto di Scala aiuta il programmatore nella gestione della complessità.

La combinazione dei punti di forza di questi due stili di programmazione così diversi apre alla possibilità di creare nuovi modelli astratti e nuovi pattern di programmazione supportati nativamente da Scala. Nelle prossime sezioni saranno riportati gli aspetti fondamentali che costituiscono Scala.

1.1 Creare nuovi tipi di dato

Scala è stato progettato per adattarsi a qualsiasi programmatore che lo utilizzi, adeguandosi alle sue esigenze specifiche. É quindi possibile costruire nuovi tipi di dato e invocare su di essi operazioni aritmetiche come se fossero dati primitivi già presenti nel linguaggio. Un esempio basilare che illustra molto semplicemente questa funzionalità è la classe scala.BigInt. Ad un primo sguardo questa classe potrebbe sembrare una classe supportata nativamente come gli Integer. Infatti i valori di entrambi questi tipi utilizzano gli stessi *literals* e gli stessi operatori come * e -. Ma in verità, pur essendo parte di una libreria standard di scala, questo non è un tipo di dato nativo. Altri linguaggi come Li-

sp, Haskell e Python implementano nativamente molte strutture dati complesse come i BigInt acquisendo una maggior espressività bilanciata, tuttavia, da una perdita di flessibilità. L'approccio mantenuto da Scala permette una facile definizione di nuovi tipi e di conseguenza di nuove librerie utilizzabili fluidamente nel linguaggio. Come i BigInt sono stati standardizzati molti altri tipi dedicati alla manipolazione dei numeri decimali, complessi, razionali, range, polinomiali, etc. Un esempio di utilizzo di questi linguaggi è riportato di seguito, in cui si può notare come effettivamente un oggetto BigInt viene utilizzato come un semplice Integer.

```
def factorial(x: BigInt): BigInt =
          if (x == 0) 1 else x * factorial(x - 1)
```

Proprio come la definizione di nuovi tipi, in Scala, è anche possibile creare nuovi costrutti di controllo. Questo permette di ricorrere ancora una volta ad una sintassi più semplice ed espressiva.

Oltre che alla definizione di nuovi tipi e costrutti questo linguaggio presenta un ottimo supporto per la definizione di nuove librerie e per l'estensione di quelle già esistenti. Un esempio pratico è dato dal pattern *Pimp my Libary*. Questo infatti è concepito per estendere e aggiungere campi e metodi a tipi di dato già esistenti. Il pattern si sviluppa essenzialmente in due fasi:

- definire una classe che implementa il nuovo tipo di dato e i relativi "nuovi metodi";
- definire una conversione implicita tra il tipo originale e il nuovo.

Un esempio di applicazione di tale pattern è riportato di seguito.

```
class BlingString(string: String) {
    def bling = "*" + string + "*"
}

implicit def blingYoString(string: String) = new
    BlingString(string)
```

Si definisce una classe *BlingString* con il metodo *bling*, e poi viene utilizzato l'implicito per convertire un tipo java.lang.String nel nuovo tipo. É quindi possibile invocare il metodo bling da una qualsiasi stringa previo importazione dell'implicito. Il risultato ottenuto è:

```
scala> "Let's get blinged out!".bling
res0: java.lang.String = *Let's get blinged out!*
```

1.2 Linguaggio orientato agli oggetti

Per essere più precisi, Scala è un linguaggio puramente ad oggetti per cui ogni cosa, dai numeri alle funzioni, è un oggetto. Ad esempio, l'espressione '1 + 1' in Scala può essere riscritta come '1.+(1)'. Questo significa che non esistendo tipi di dati primitivi come accade in Java ma anche un numero o una lettera sono oggetti. Nell'esempio precedente ('1 + 1') la prima variabile '1' fa uso del metodo '+' definito della classe scala.Int. Questo linguaggio permette infatti di dichiarare i metodi di una classe utilizzando anche caratteri come i simboli aritmetici, i simboli di confronto e i doppi punti. Usare tali simboli permette di mantenere semplice e concisa la sintassi. Il metodo '+', ad esempio, può essere definito come metodo anche per una struttura dati o un nuovo tipo di dato numerico, inglobandolo nel linguaggio nativo. Un'ulteriore differenza con i classici linguaggi ad oggetti riguarda l'assenza della *dot notation* per invocare un metodo: quando un metodo richiede un solo parametro di ingresso, è possibile usare la notazione operatore, la quale prevede l'assenza del punto tra l'oggetto chiamante e il metodo invocato e l'assenza delle parentesi che circondano l'unico parametro di ingresso. Il principale vantaggio derivante dall'essere orientato agli oggetti è garantito proprio dai principi fondanti del paradigma OOP stesso, in quanto fornisce un supporto naturale alla modellazione software degli oggetti del mondo reale o del modello astratto da riprodurre. Ciò consente una più facile gestione e manutenzione di progetti di grandi dimensioni, favorendone la modularità e il riutilizzo di codice. Scala introduce funzionalità più avanzate rispetto a linguaggi come Java o C#, come ad esempio i *trait*. Essi assomigliano alle interfacce presenti in Java 8, in quanto entrambe possono avere campi e metodi già implementati. La differenza risiede nel fatto che se in Scala è consuetudine inserire campi e metodi in un trait, in Java è invece molto più frequente un utilizzo canonico delle interfacce con la definizione dei soli metodi astratti. Un oggetto può quindi implementare e mixare più trait, acquisendo la possibilità di suddividerne le sue diverse caratteristiche tra essi. Questo permette di aggiungere facilmente nuove funzionalità senza il bisogno di dover specificare una

superclasse, ma solamente aggiungendoli al trait.

1.3 Linguaggio funzionale

Come già accennato in precedenza, oltre alla programmazione ad oggetti, Scala incorpora anche le caratteristiche della programmazione funzionale, la quale trova fondamento su due idee. La prima è quella di considerare le funzioni come entità di prima classe, al contrario dei linguaggi funzionali più tradizionali. Questo significa che le funzioni possono essere trattate come dei valori, come accade per le variabili di tipo Int o String, per cui le funzioni di prima classe possono essere assegnate ad una variabile o essere usate come parametri di ingresso o di ritorno da un'altra funzione. Inoltre si possono definire funzioni innestate in altre funzioni senza il bisogno di nominarle, ma utilizzandole esattamente come una semplice variabile. Queste caratteristiche permettono di creare nuovi costrutti di controllo come un ciclo while e assegnarli ad una variabile, riutilizzandolo semplicemente invocando quest'ultima. La seconda idea è quella di pensare che una funzione debba mappare un valore di ingresso in un valore di uscita, o più semplicemente che una funzione debba comunicare con il mondo esterno solo attraverso i valori di input-output. Nella programmazione ad oggetti significherebbe utilizzare oggetti che espongono al loro esterno solo dati immutabili. Infatti, se un oggetto esponesse al mondo esterno un campo mutabile, allora una qualunque funzione invocata su di esso potrebbe usare proprio quel campo per calcolare il suo valore di uscita, provocando dei side effect imprevedibili. Scala definisce molti più tipi immutabili rispetto a Java: liste, mappe, set e tuple. Ci sono diversi motivi per cui l'utilizzo di oggetti immutabili può risultare preferibile in fase di programmazione. Ad esempio si possono usare le String di Java come chiavi di ricerca in un HashMap senza preoccuparsi che qualcuno possa apportare una modifica ai caratteri della chiave. Oppure, siccome un oggetto immutabile non può essere modificato, si può usare la concorrenza senza ricorrere a tecniche di sincronizzazione così da leggere simultaneamente i dati dell'oggetto, poiché non esisterebbero stati di inconsistenza. In generale quindi, se le singole parti di codice che costituiscono un'applicazione non dipendono da uno stato mutabile in comune, è possibile ragionare sulle singole parti senza dover tener conto di tutte le altre.

1.4 Tipizzazione

Un'altra particolarità importante del linguaggio riguarda la tipizzazione, che in Scala viene gestita in maniera molto avanzata. Una delle classificazioni possibili tra i vari linguaggi di programmazione, avviene proprio attraverso la tipizzazione utilizzata, ovvero il modo in cui un tipo viene assegnato ad una variabile. Esistono principalmente due tecniche differenti: la prima, usata dai più importanti linguaggi di programmazione, si chiama tipizzazione statica e prevede la definizione del tipo di ogni variabile a tempo di compilazione. Questa tecnica obbliga quasi sempre il programmatore a definire nella dichiarazione di una variabile anche il tipo che questa dovrà assumere. L'unica eccezione riguarda le situazioni in cui è possibile ricorrere alla *type inference* (ad esempio nella dichiarazione di Liste in Java), la quale permette di riconoscere il tipo di una variabile anche se non esplicitamente dichiarato.

La seconda, usata ad esempio nei linguaggi di scripting, si chiama tipizzazione dinamica ed è in grado di risalire al tipo di una variabile a run-time. La tipizzazione statica tende a essere più onerosa in termini di scrittura del codice rispetto a quella dinamica, ma permette di eliminare errori subdoli che danno al programmatore l'illusione che tutto funzioni. Infatti, se un errore di tipizzazione non viene rivelato direttamente dal compilatore, l'applicazione funziona fino al momento in cui una qualche sequenza di istruzioni non manda in errore il programma. In un caso come questo, non resta altro che cercare di correggere l'errore sperando che non ce ne siano altri.

Scala, pur essendo un linguaggio staticamente tipato, fa largo uso della type inference, consentendo quindi di alleggerire notevolmente il codice. I benefici più rilevanti derivanti da questo tipo di tipizzazione:

verifica delle proprietà: la tipizzazione statica non riesce a prevedere tutti i tipi d'errore che possono verificarsi a -time (come ad esempio la divisione per zero o la violazione dei limiti di un array) ma può ridurre i testing rispetto ad alcune proprietà di tipo, come il fatto di non sommare un booleano ad un numero o non accedere a campi privati al di fuori della singola classe;

refactoring sicuro: avvalendosi della proprietà precedente, è intuitivo capire come sia semplice verificare che, una volta modificato un parametro di una funzione, questo cambiamento sia riportato correttamente in tutto il

codice. Infatti basterebbe compilare e correggere solamente i punti in cui viene notificato un errore di tipo dal compilatore;

documentazione: per quanto riguarda la definizione di metodi e funzioni di interfacce e/o di componenti riutilizzabili, la tipizzazione statica risulta al quanto utile per la comprensione esplicita del "contratto" tra il componente e le classi che lo implementano/estendono.

Grazie all'unione della tipizzazione statica e dell'utilizzo di una sintassi alleggerita, Scala può essere usato come linguaggio di scripting e come ambiente di sviluppo affidabile per applicazioni più complesse.

Capitolo 2

Scala e Java

Una caratteristica fondamentale di Scala è la sua completa interoperabilità con il linguaggio Java. Pur esistendo anche una variante di Scala che funziona sulla piattaforma .NET, il supporto sulla JVM funziona meglio ed è molto più utilizzato. Infatti questi due linguaggi vengono utilizzati spesso insieme soprattutto al fine di implementare programmi e framework di grandi dimensioni. Il progetto sviluppato nell'ambito di questa tesi vuole esserne un esempio. L'interoperabilità si basa sul fatto che il codice Scala viene compilato in bytecode che può essere interpretato dalla Java Virtual Machine. Questo significa che Scala gode quasi delle stesse performance di Java (un esempio è proprio la sopracitata espressione '1 + 1', introdotta nel capitolo precedente). Infatti, tutte le volte in cui è possibile, i componenti Scala sono mappati direttamente sugli equivalenti Java. Nel caso dei tipi primitivi Java (int, double...) ciò consente di non dover rinunciare minimamente alle prestazioni. Esistono poi casi in cui il mapping tra Scala e Java non è applicabile direttamente: infatti in Scala sono presenti delle funzionalità avanzate che talvolta ampliano quelle fornite dalle classi standard in Java. In sostanza i livelli di interoperabilità da considerare sono due: Scala che incorpora codice Java o viceversa.

2.1 Mapping Java-Scala

Un codice Scala può operare liberamente su metodi, campi, classi ed interfacce Java e nessuna di queste funzionalità richiede una specifica sintassi o *glue-code*. In virtù di ciò la maggior parte di codice Scala dipende fortemente dalle librerie Java, in particolar modo da quelle standard (ad esempio il tipo della stringa *"abc"* in Scala è comunque java.lang.String). Non solamente Scala riutilizza i tipi di Java, ma addirittura li potenzia, come nel caso del tipo String a cui vengono aggiunti i metodi per trasformare una variabile di questo tipo in un numero (str.toInt) senza dover passare dal metodo statico della classe destinataria (Integer.parseInt(str)). Per gestire la possibilità di poter aggiungere metodi alle classi base, Scala utilizza una soluzione veramente generale e funzionale, sfruttando delle **conversioni implicite** di tipo. Tali conversioni implicite provvedono a cambiare automaticamente il tipo della variabile con quello corretto, nel caso in cui un certo tipo di variabile non corrisponda a quello standard solitamente dichiarato nelle librerie Java. E' bene rimarcare il fatto che il tipo corretto è ovviamente scritto in Scala. Nel caso precedente, quando il compilatore incontra l'istruzione str.toInt si accorge che il metodo utilizzato non appartiene alla classe definita nella libreria java.lang.String e pertanto ricerca la dichiarazione di una conversione implicita java.lang.String $\rightarrow$ scala.StringOps, dove è effettivamente definito questo metodo.

2.2 Mapping Scala-Java

Se utilizzare codice Java all'interno di Scala risulta alquanto facile, il contrario, seppur possibile, appare decisamente più complicato. Scala, infatti, presenta delle *feature* più avanzate e dunque non tutto il codice è direttamente mappabile sui costrutti Java, i quali quindi necessitano di qualche doveroso aggiustamento. Per queste feature la codifica in Java non si intende fissata a priori ma, al fine di attuarla, è possibile adottare diverse tecniche. Le feature più importanti non direttamente mappabili da Scala a Java sono:

- **singleton objects:** per ognuno di questi viene creata una classe Java con lo stesso nome e $ alla fine. Viene inoltre definito un campo MODULE$ che rappresenta l'unica istanza della classe creata a run time. Un esempio di conversione è riportato nella Figura 2.1 [9, Chapter 31], in cui è stato utilizzato il comando *javap* per visualizzare e informazioni relative alla classe Java App$ creata;
- **trait:** in generale se un trait dichiara solo metodi astratti, esso viene tradotto direttamente in un'interfaccia Java. Tuttavia nel caso di un utilizzo più avanzato del trait, il mapping risulta essere abbastanza oneroso.

```
object App {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```



```
$ javap App$
public final class App$ extends java.lang.Object
implements scala.ScalaObject{
    public static final App$ MODULE$;
    public static {};
    public App$();
    public void main(java.lang.String[]);
    public int $tag();
}
```

Figura 2.1: A sinistra singleton Scala e a destra classe Java.

2.3 Compilazione

Solitamente quando si desidera compilare un progetto contenente contemporaneamente codice Java e Scala, è sufficiente procedere dapprima alla compilazione della parte indipendente così da compilare la porzione rimanente di codice semplicemente inserendo i file .class già compilati nel classpath. Nel caso in cui vi sia una forte interdipendenza da ambo le porzioni di codice Java e Scala, per poter garantire il necessario supporto alle due diverse compilazioni, Scala consente di compilare inserendo direttamente i sorgenti Java al posto dei .class. Il compilatore Scala non procederà però alla compilazione diretta dei .java, ma potrà comunque parsarli controllandone il contenuto. Quindi come si evince da questo banale esempio, è sufficiente compilare dapprima la parte Scala, utilizzando anche i file sorgenti di Java, procedendo quindi di conseguenza con la parte Java utilizzando i file .class di Scala. Un esempio di compilazione è riportato nel seguente codice.

```
$ scalac -d bin MyProgram.scala MyProgram.java \
    MyProgramItem.java
$ javac -cp bin -d bin MyProgram.java MyProgramItem.java \
    MyProgramManagement.java
$ scalac -cp bin MyProgram
```

Capitolo 3

Scafi

Il framework Scafi [10] è un progetto open source[1] che permette la scrittura di programmi basati sul paradigma di programmazione aggregata, mettendo a disposizione un insieme di primitive Scala molto compatte, che definiscono quindi un vero e proprio linguaggio di programmazione specifico per aggregate programming. Scafi presenta molteplici utilizzi e fornisce sostanzialmente tre tipi di servizi:

- una libreria Scala che realizza la semantica del field calculus in modo corretto e completo;
- una piattaforma distribuita Akka-based su cui sviluppare applicazioni;
- un'API generale e flessibile in grado di supportare diversi scenari.

Questo framework è stato sviluppato dal prof. Mirko Viroli e successivamente esteso dal dott. Roberto Casadei nell'ambito della sua tesi, dalla quale sono state estrapolate informazioni ed immagini, utilizzate nel seguito della trattazione. Nel seguente capitolo verrà descritta la struttura di Scafi, introducendo in primo luogo la libreria *Core* per l'implementazione del field-calculus, passando successivamente alla descrizione della piattaforma ad attori per lo sviluppo di applicazioni aggregate e distribuite.

[1] il progetto è reperibile al seguente indirizzo `https://bitbucket.org/scafiteam/scafi`

3.1 Libreria Core

Per l'implementazione del DSL (Domain Specific Language) Scafi utilizza alcuni degli aspetti avanzati di Scala tra cui:

Type classes: dato un tipo se ne fornisce un'interfaccia astratta per definire più implementazioni specifiche;

Family polymorphism: creazione di famiglie di tipi ricorsivi le quali possono essere raffinate in modo incrementale;

"Pimp my library" pattern: basato sugli impliciti di Scala e utilizzato per estendere i tipi già esistenti con nuovi metodi e campi;

Cake pattern: si definisce un trait in termini delle sue dipendenze attraverso i self-types.

Nella Figura 3.1, viene riportata l'architettura di progetto del framework Scafi, mettendo in evidenza i componenti chiave della libreria.

Il componente **Core** definisce le astrazioni base e gli elementi architetturali che saranno poi raffinati dai componenti figli. Il componente **Language**, basato sulle astrazioni definite in Core, definisce i principali Constructs del DSL, il quale espone le primitive del fiedl calculus come metodi. Sopra queste primitive vi sono gli operatori Builtins che provvedono a rendere il linguaggio più espressivo (RichLanguage). Il componente **Semantics** estende la parte sintattica e strutturale di Language, raffina le astrazioni di Core e fornisce una semantica per i Constructs, venendo poi reso eseguibile dal componente **Engine**. L'implementazione di questi componenti base costituenti l'architettura è effettuata completamente in Scala. In definitiva, i componenti vengono rappresentati attraverso traits, classi, oggetti e così via, così da rendere possibile la creazione di famiglie di tipi ricorsivi, le quali possono essere raffinate in modo incrementale.

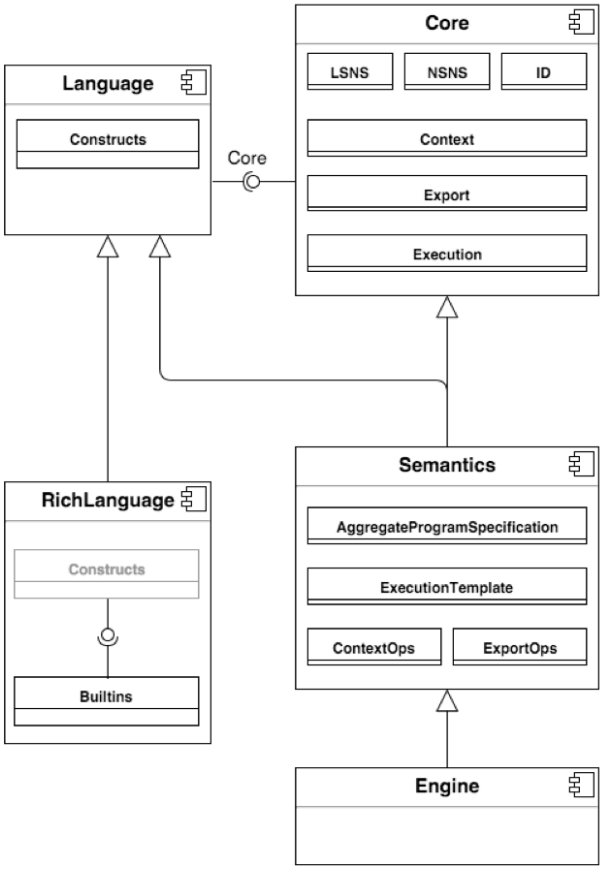


Figura 3.1: Design del DSL Scafi.

3.2 Piattaforma distribuita

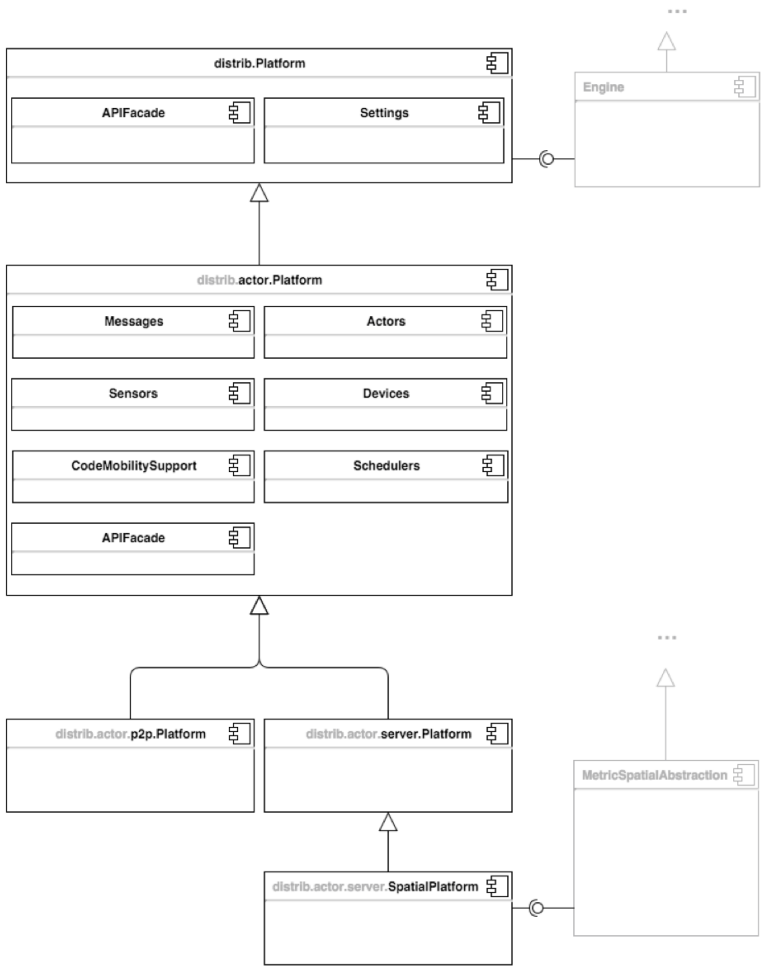
La piattaforma fornita da Scafi si basa sul modello ad attori e in particolare sull'implementazione fornita da Akka. Tale piattaforma si divide sostanzialmente in due tipi:

- decentralizzata: piattaforma peer-to-peer;
- centralizzata: piattaforma sever-based.

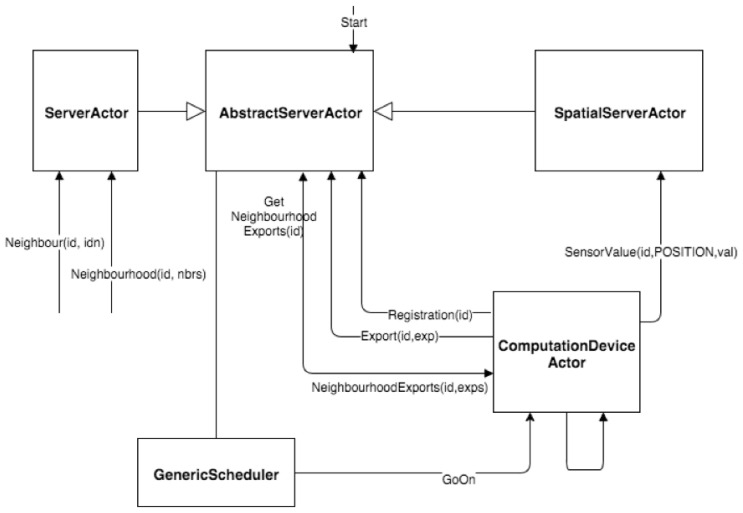
La piattaforma server-based viene specializzata nella *SpatialPlatform* che mantiene e gestisce le posizioni spaziali dei device, seguendo l'architettura del modello client/server. I singoli dispositivi rappresentano i diversi client rispetto ad un altro dispositivo centrale il quale mantiene tutte le informazioni delle

caratteristiche della rete ed è responsabile di propagare le informazioni rispetto agli stati dei diversi device. L'interazione fra i client e il server si sviluppa in due fasi: inizialmente ogni device deve registrarsi al server, dopodiché ogni volta che un device calcola un nuovo stato, denominato *Export*, lo deve comunicare al server. Per calcolare il nuovo stato ogni device richiede al server lo stato più recente dei suoi vicini con una certa frequenza.

Nella Figura 3.2a, viene riportata l'architettura dell'intera piattaforma messa a disposizione da Scafi, dove appare ben visibile la separazione tra i due tipi descritti in precedenza. Nella Figura 3.2b è invece riportata la struttura della piattaforma server-based, utilizzata nel prototipo del simulatore descritto nei capitoli successivi.



(a) Design architetturale della piattaforma Scafi.



(b) Design di dettaglio dei componenti della piattaforma server-based Scafi.

Figura 3.2

Capitolo 4

Simulatore

4.1 Panoramica del progetto

Lo studio preliminare per la progettazione del simulatore si è sviluppata in prima battuta durante il tirocinio svolto sempre presso l'università di Bologna sotto la supervisione del prof. Viroli. Si è iniziato con lo stabilire gli aspetti più rilevanti da rappresentare rispetto ad un'applicazione aggregata generica (i nodi e i collegamenti tra essi), aspetto successivamente descritto nel capitolo riguardante l'analisi dei requisiti, poi si è passati al definire come rappresentarli al meglio in modo che potessero essere chiari ed intuitivi.

Terminata la modellazione dell'interfaccia grafica si è passati alla definizione di alcune API il cui scopo è quello di permettere una facile integrazione tra il simulatore e un qualsiasi programma aggregato. Tale progetto non deve necessariamente essere implementato utilizzando Scafi, per questo si è proceduto definendole nel modo più generico possibile.

La parte finale del progetto è stata utilizzata per implementare un esempio concreto di collegamento tra il simulatore e un programma Scafi creato dal Dott. Roberto Casadei.

L'intero sistema è stato sviluppato basandosi sul pattern di progettazione MVC. Per la view, in cui si sono inseriti tutti i componenti grafici del simulatore, si è mantenuto un approccio di sviluppo di tipo bottom-up, in quanto si è deciso inizialmente di trovare un'adeguata rappresentazione per gli elementi principali, come i nodi, e in seguito si è pensato a come gestirli collettivamente in un unico ambiente. Al contrario, per quanto riguarda il model, in cui ritro-

viamo le API di collegamento, si è invece adottato un criterio di tipo top-down partendo dalla modellazione generale delle entità rilevanti al fine di rappresentare un'applicazione aggregata e, in un secondo momento, si sono affrontati gli opportuni raffinamenti.

Capitolo 5

Analisi

5.1 Requisiti

In questo capitolo sono presentati in maniera schematica ed esaustiva tutti i requisiti che sono emersi durante l'analisi preliminare del simulatore. Il progetto del simulatore in ambito di tesi nasce con lo scopo di poter rappresentare il comportamento di sistemi aggregati distribuiti e di permetterne l'interazione durante la loro esecuzione. Ad esso saranno demandati i seguenti compiti:

- rappresentare graficamente i componenti principali di un sistema caratterizzato dai seguenti elementi:
  - nodo, inteso come singolo device;
  - network, inteso come rete di collegamento tra i nodi;
  - immagine di sfondo.
- gestire le interazioni con la simulazione come:
  - configurazione iniziale;
  - avvio/interruzione;
  - esecuzione di n-passi.
- prevedere l'attuazione di azioni sui nodi, ossia:
  - cambiare il valore di un determinato sensore;
  - visualizzare una certa informazione riguardante il nodo;
- gestire il collegamento tra il simulatore e un'applicazione aggregata

5.2 Problema

Nella seguente sezione sono descritti i requisiti sopraelencati in maniera più esaustiva, ponendo particolare attenzione ai problemi derivanti da ciascuno.

Configurazione di visualizzazione: l'utente deve poter visualizzare in modo chiaro e ben definito le informazioni più rilevanti sulla simulazione. Tali informazioni possono essere rappresentate sotto forma di testo oppure graficamente attraverso dei colori associati ai diversi stati dei nodi. La dimensione dell'ambiente grafico non deve incidere sulla "porzione" di simulazione mostrata ma deve consentire di visualizzare tutto il sistema istante per istante. Infine deve essere possibile inserire un'immagine in background nel caso si voglia fare una demo con una determinata figura di sfondo, come ad esempio la piantina di un edificio o una mappa stradale. Ricapitolando, le funzionalità più significative che devono poter essere eseguite lato utente risultano pertanto:

- ridimensionamento della finestra → gestione evento *ComponentResized*;
- inserimento immagine di sfondo → inserimento componenti in un *LayerdPane*;
- visualizzazione dei collegamenti tra i nodi → ridefinizione del metodo *paintComponent()*;
- selezione di una porzione di nodi → ridefinizione del metodo *paintComponent()*;

Configurazione di visualizzazione dei nodi: è necessario poter rappresentare in modo intuitivo tutte le informazioni riguardanti i singoli nodi, oltre ovviamente alle necessarie informazioni sulla simulazione. Ogni nodo pertanto verrà "equipaggiato" di un pannello informativo in cui saranno mostrati i valori dei diversi sensori. Verrà inoltre predisposta, oltre al pannello, una funzionalità ad hoc che consenta di visualizzare un determinato valore per tutti i nodi (infatti il semplice utilizzo del pannello potrebbe rivelarsi scomodo). In maniera sintetica, le proprietà configurabili per la visualizzazione dei nodi risultano pertanto essere:

- cambio di etichetta dei nodi → utilizzo di una JLabel che può cambiare il testo mostrato oppure ometterlo;

- cambiamento nel colore dei nodi → utilizzo di un'icona;
- consistenza delle informazioni mostrate nel pannello → gestione di metodi di sincronizzazione tra modello e view;

Configurazione di una rete: per rete si intende un insieme di nodi collegati tra loro. Idealmente un'applicazione aggregata deve mantenere un corretto comportamento con qualsiasi configurazione di rete possibile, per cui si è pensato di lasciare all'utente la possibilità di agire di volta in volta sulla configurazione della rete su cui testare la propria applicazione. Della particolare rete di nodi sarà possibile configurare:

- il numero di nodi → se troppo elevato si rischia di incappare in un blocco del simulatore;
- il raggio di vicinato;

Configurazione della simulazione: per simulazione si intende il programma in esecuzione nella sua totalità. L'utente potrà in seguito definire delle proprietà generali del sistema, che saranno poi utilizzate per la creazione di quest'ultimo. Le proprietà personalizzabili sono:

- velocità della simulazione;
- programma aggregato de eseguire → mettendo a disposizione eventuali sistemi già predisposti nel simulatore;
- strategia d'esecuzione → predisponendo l'esecuzione di un sistema sia su piattaforma p2p-based che server-based.

Tali informazioni dovranno poi essere passate al modello il quale le utilizzerà per la creazione della simulazione.

Gestione simulazione: oltre alla configurazione iniziale, deve essere possibile gestire il sistema già in esecuzione e interagire con esso attraverso specifiche azioni, quali:

- start-stop-pausa;
- avanzamento di un numero di passi definito;
- avanzamento fino al verificarsi di un determinato evento;
- esecuzione di un'azione su un sottoinsieme di nodi selezionato → mantenere memorizzati i nodi all'interno dell'area di selezione;

- esecuzione di un'azione su tutti i nodi della rete.

Nella Figura 5.1 sono rappresentati i principali casi d'uso del simulatore.

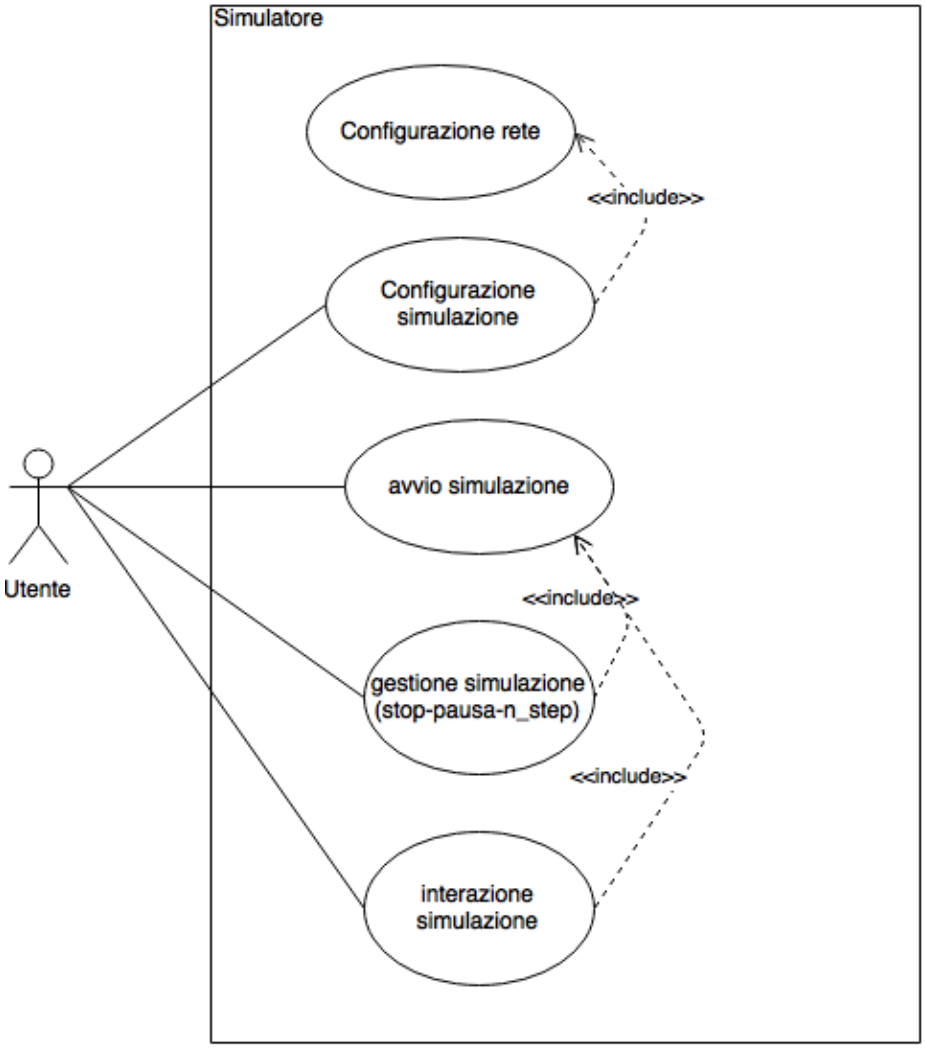


Figura 5.1: Diagramma UML dei casi d'uso del simulatore.

Capitolo 6

Design

Il sistema è stato progettato utilizzando il pattern di modellazione MVC, in cui la parte di view è rappresentata dall'aspetto grafico del simulatore e il model dalle API fornite per la gestione del collegamento con l'applicazione aggregata. La Figura 6.1 riassume lo schema generale del design del progetto sviluppato.

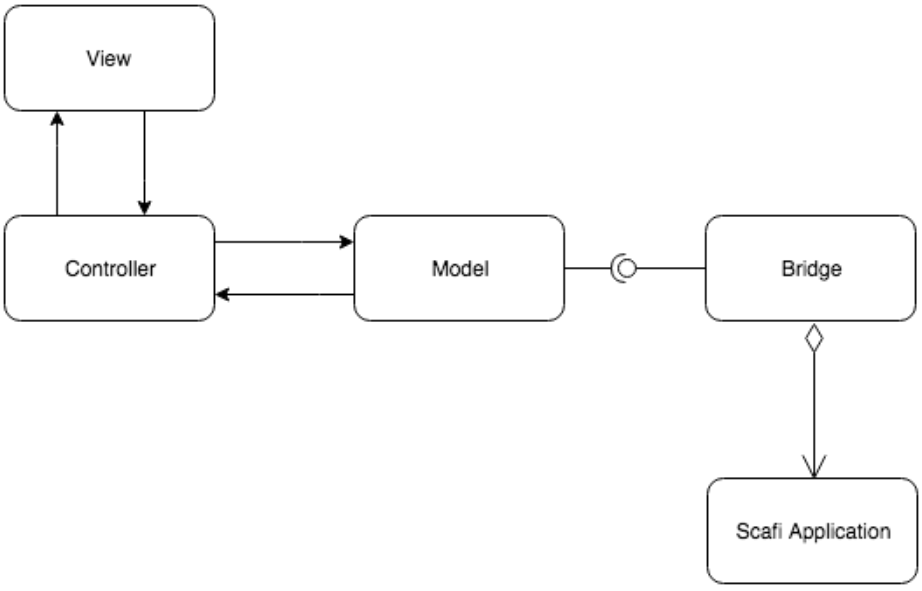


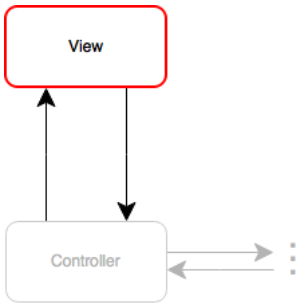
Figura 6.1: UML Design del sistema

Da questa immagine si evince facilmente come sia semplice sostituire l'applica-

zione aggregata da voler testare, infatti comporterebbe solo un aggiustamento nell'**Adapter**. Tutto questo è possibile grazie alla totale indipendenza del simulatore rispetto al programma aggregato. Ovviamente nel caso in cui si vogliano sviluppare più funzionalità grafiche rispetto a quelle presenti, sarà sufficiente provvedere il loro inserimento nella **View** utilizzando i metodi già predisposti dato che la GUI è completamente configurabile.

Nel seguente capitolo saranno presentati i design delle diverse parti.

6.1 Ambiente grafico



L'ambiente grafico è stato sviluppato interamente in Java, facendo largo uso dei concetti di modularità e incapsulamento: infatti i diversi requisiti della view sono stati incapsulati in classi dedicate e, in un secondo momento, sono stati raggruppati in altre, facenti funzione di contenitore.

6.1.1 Schermata iniziale

Nella Figura 6.2 viene mostrato il design dei principali componenti dell'interfaccia grafica che il simulatore offre all'avvio dell'applicazione. Si può notare che tale elemento si compone essenzialmente di tre classi:

- **MenuBarNorth:** incapsula le proprietà della parte superiore della schermata. Le diverse funzionalità sono inserite all'interno di alcuni menù denominati coerentemente con le azioni che mantengono. I diversi menù sono riportati di seguito, elencando le funzionalità mantenute:
  - **File:** creazione di una nuova simulazione, chiusura dell'applicazione;
  - **Simulation:** inserimento/rimozione immagine di sfondo, gestione della dinamica della simulazione (start, stop, pause, step);

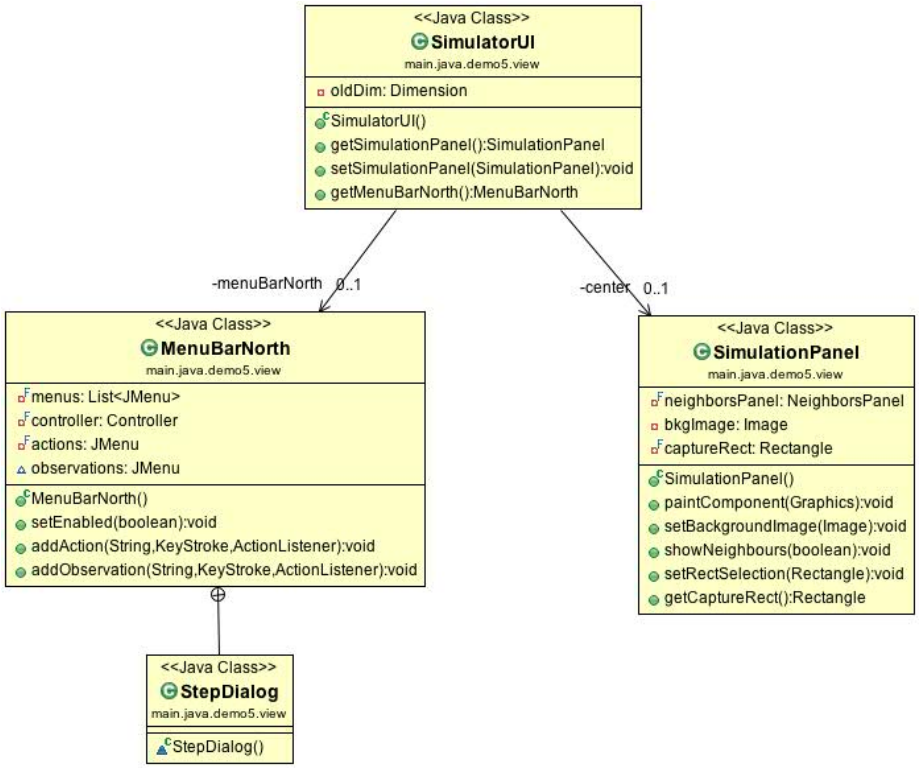


Figura 6.2: Diagramma UML della view relativa alla simulazione.

  - **Action:** visualizzazione/rimozione rete di vicinato, impostazione nodo (sorgente,ostacolo,normale), impostazione di un sensore ad un certo valore;
  - **Node:** visualizzazione id/export/niente, marcatura dei nodi che mantengono un sensore con un determinato valore.

  I menù File e Simulation sono preimpostati e non modificabili, mentre gli altri due (Action e Node) sono totalmente configurabili tramite i metodi *addAction(String, KeyStroke) e addObservation(String, KeyStroke)* forniti dalla classe MenuBarNorth.

- **SimulationPanel:** in esso viene effettivamente rappresentata l'esecuzione della simulazione e incapsula le proprietà della parte centrale. Questo componente estende la classe Java *JDesktopPane* così da poter aggiungere al proprio ContentPane tutti i *JInternalFrame* necessari al fine di rappresentare i nodi. Come si avrà modo di approfondire in seguito, ogni nodo è infatti rappresentato da un JInternFrame.

- **SimulatorUI:** ingloba i due elementi precedenti in un unico *JFrame* il quale gestisce anche gli eventi che riguardano l'aspetto generale del simulatore. Ad esempio esso controlla le politiche di ridimensionamento della finestra principale per cui, a fronte di un cambiamento di dimensione, provvede a mantenere la proporzionalità con la schermata iniziale riposizionando e ridimensionando opportunamente i componenti grafici.

6.1.2 Pannello della simulazione

Come già accennato in precedenza, il pannello dedicato alla rappresentazione dello svolgimento della simulazione è il componente *SimulationPanel*. Questo pannello deve assolvere a diversi compiti:

- rappresentazione dei nodi e gestione della loro eventuale sovrapposizione;
- rappresentazione della la rete di interconnessioni tra i nodi;
- rappresentazione di un'ipotetica immagine di sfondo.

La coesistenza di queste tre funzionalità ha determinato la suddivisone del SimulationPanel in diversi "piani", ognuno dei quali sarà visualizzato ad un certo "livello". Ovviamente i piani con livello inferiore saranno "coperti" da quelli di livello superiore. Per garantire la visualizzazione di tutti i piani, sono state gestite opportunamente le trasparenze di sfondo. In particolare il pannello della simulazione è stato suddiviso in tre piani distinti, come si evince dalla Figura 6.3.

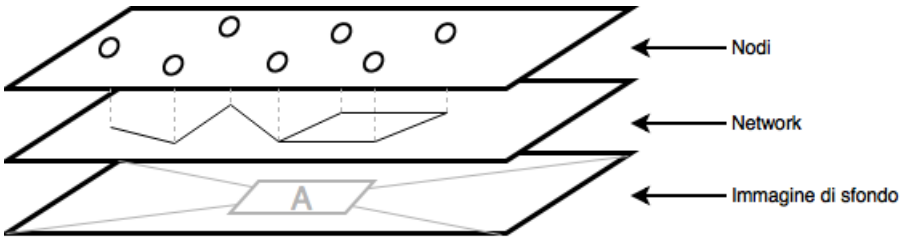


Figura 6.3: Suddivisone in piani del SimulationPanel.

Il pannello per la rappresentazione dell'immagine e quello per la rete di connessione tra i nodi sono entrambi gestiti internamente alla classe Simulation-

Panel: il primo viene definito tramite un campo privato interno e visualizzato grazie alla ridefinizione del metodo *paintComponent(Graphics)* mentre il secondo è stato modellato in una classe separata poiché, se visibile, viene ridisegnato ad ogni singolo spostamento dei nodi, così da mantenere un effetto uniforme del collegamento. Come punto d'incontro dei collegamenti, i quali non sono altro che delle linee rette, si è preso il punto centrale dei nodi e, siccome il pannello dove vengono rappresentati quest'ultimi si trova ad un livello superiore, le linee vengono coperte.

Un'ultima funzionalità messa a disposizione da questo componente consiste nella possibilità di selezionare un sottoinsieme di nodi. Questo aspetto è stato gestito ancora una volta agendo sul metodo paintComponent(Graphics) in cui è stato appositamente inserito una if-clause di controllo per la sua eventuale rappresentazione. Nella Figura 6.4 è riportato il diagramma UML che rappresenta i componenti appena descritti.

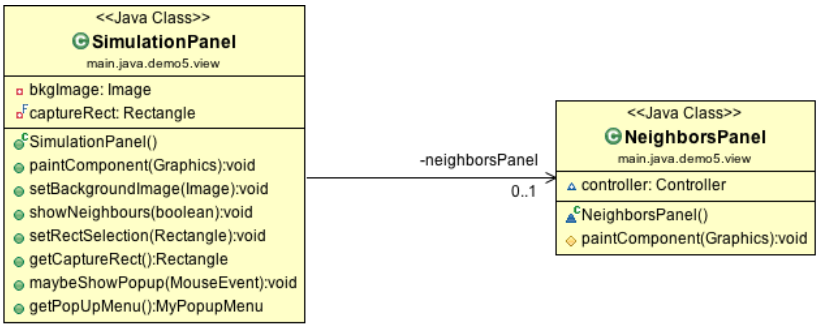


Figura 6.4: Schema generale della GUI.

6.1.3 Nodo

Il concetto di nodo in una applicazione distribuita rappresenta una singola unità computazionale del sistema, come ad esempio uno smartphone. Dal momento che un sistema di questo tipo prevede un numero molto alto di nodi, è necessario che questi siano modellati graficamente in modo semplice e chiaro. Gli aspetti che si sono voluti gestire rispetto i nodi sono i seguenti tre:

- rappresentazione iconografica (GuiNode);
- controllo dello spostamento (GuiNodeListener);
- visualizzazione di tutte le informazioni relative ai nodi(NodeInfoPanel).

Possiamo ritrovare tutti questi componenti nella Figura 6.5 che illustra il design generale dei nodi.

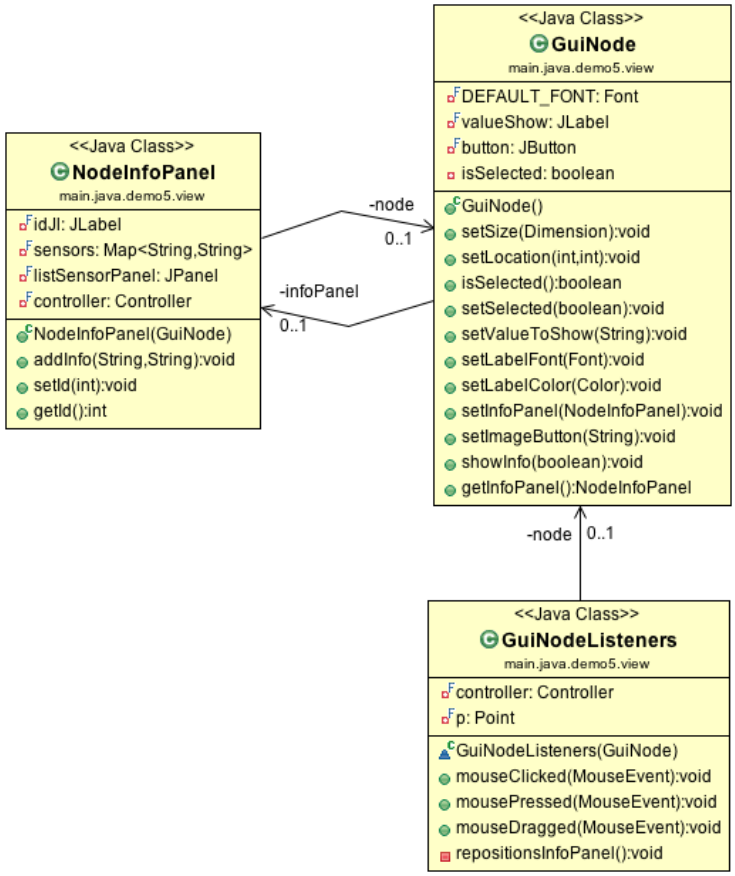


Figura 6.5: Diagramma UML di GuiNode.

GuiNode è la classe deputata alla gestione di tutti gli aspetti grafici del singolo nodo e pertanto mantiene tutti i metodi necessari per modificare il suo look and feel. In particolare esso permette di agire:

- sul ridimensionamento;
- sull'icona rappresentativa;
- sul valore visualizzabile.

GuiNodeListener è la classe dedicata alla gestione degli spostamenti di ogni nodo e del relativo pannello delle informazioni. Infatti, in seguito ad uno spostamento del nodo, è necessario che il pannello informativo ne segua gli stesso spostamenti, qualora sia visualizzato.

NodeInfoPanel è la classe che permette la visualizzazione di tutte le informazioni relative ad un nodo. Essendo che tali informazioni potrebbero eventualmente risultare diverse ad ogni simulazione oggetto di testing, il pannello può essere dinamicamente composto grazie al metodo *addInfo(String, String)*, dove il primo parametro inserito rappresenta il nome-chiave dell'informazione che si vuole mostrare, mentre il secondo parametro rappresenta l'effettivo valore da mostrare.

Queste tre parti (Barra dei menù, pannello della simulazione e rappresentazione dei nodi) vengono coordinate tutte all'interno del Controller del progetto. In particolare, all'atto della creazione dei nodi, il relativo inserimento nel pannello della simulazione viene gestito dal metodo start() del controller, in cui si avvia effettivamente una nuova simulazione. Nella Figura 6.6 è riportato il diagramma UML relativo a tutti i componenti grafici, inserendo anche la classe del controller definendo così la modalità con cui i GuiNode sono relazionati al SimulationPanel.

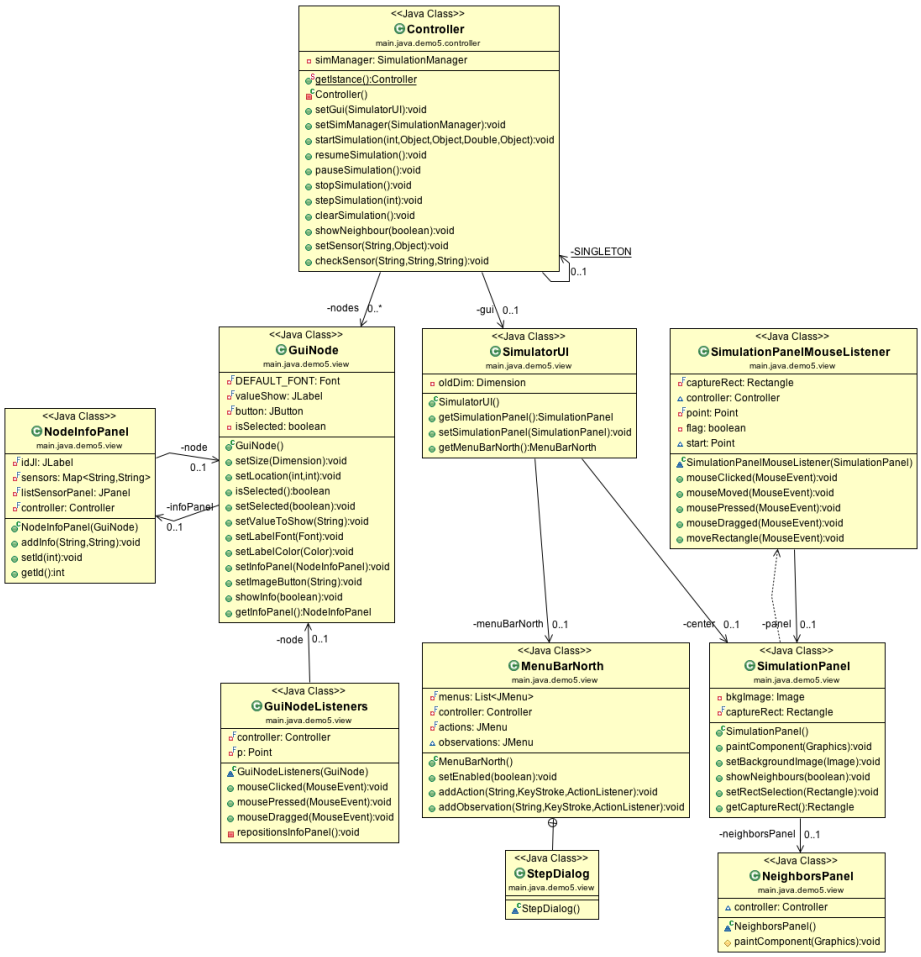
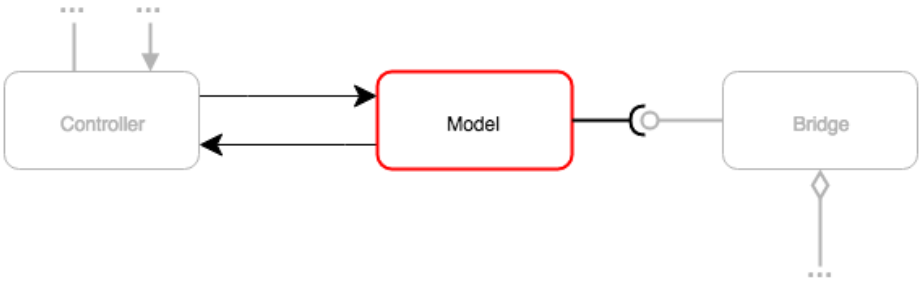


Figura 6.6: Diagramma UML completo della view del simulatore

6.2 Model



Il model è la parte del simulatore dedicata all'integrazione tra la parte grafica e l'applicazione aggregata, la quale provvede alla computazione vera e propria. Per questo motivo il model prevede l'esposizione di determinate interfacce, le quali verranno poi estese dalle classi concrete che effettivamente costruiscono il componente definito come *Bridge*. Il nome di questo componente è stato scelto in seguito all'utilizzo del pattern omonimo. Sebbene inizialmente si sia optato per un'implementazione completamente java based, data la natura "ibrida" del progetto si è ritenuto opportuno sostituire una interfaccia java della demo con un trait Scala. Ciò ha notevolmente semplificato la creazione dell'adapter per la demo stessa.

Come per la view anche questa parte è stata modellata in maniera totalmente indipendente dal resto del progetto. I vari aspetti di una simulazione che sono gestiti in maniera separata sono:

- gestione di un determinato insieme di nodi;
- gestione coerente e aggiornata della rete che gli unisce;
- gestione del programma da eseguire/in esecuzione;
- gestione strategia d'esecuzione.

Partendo dalla modellazione di queste caratteristiche, sono stati in seguito aggiunti anche altri aspetti che risultano particolarmente correlati alla definizione di applicazione aggregata e che saranno introdotti nel proseguo della trattazione.

L'entità principale delle API fornite dal modello è rappresentata dall'interfaccia **SimulationMananger**. Questa ha il compito di gestire tutte le interazioni con la simulazione configurata inizialmente dall'utente. Tale interfaccia dovrà provvedere ad avviare, terminare, mettere in pausa, eseguire un certo numero di computazioni e a gestire la velocità della simulazione. Tutti questi aspetti si sono poi tramutati in metodi esposti dall'interfaccia inserendo inoltre anche il metodo astratto *setSimulation(Simulation)* in modo da consentire l'esecuzione arbitraria di più simulazioni avviando una volta sola il simulatore.

Simulation incapsula tutte le proprietà definite nella fase di configurazione da parte dell'utente fatta eccezione per la rete di nodi. Dunque questa interfaccia mantiene l'istanza del programma da eseguire, la velocità della simulazione e la strategia d'esecuzione intesa come utilizzo della piattaforma p2p o server-based di Scafi.

Network è responsabile della coordinazione del set di nodi e delle interazioni tra essi, per cui anch'esso deve essere configurabile dall'utente. In particolare, si deve poter agire sul numero di nodi e sulle politiche di vicinato, intese come proprietà che i nodi devono soddisfare per ritenersi "vicini" (ad esempio opportuni vincoli sulla distanza tra di essi).

Node rappresenta un singolo nodo della rete che può effettuare delle generiche azioni tramite il metodo *doAction(Action)* e gestire determinati sensori, i quali possono essere sia letti che scritti, ovvero è possibile modificarne il valore.

Action e **Sensor** rappresentano rispettivamente le Azioni e i Sensori correlati ai nodi. Queste interfacce sono volutamente molto generiche poiché si prevede che per ogni specifica simulazione esse vadano ridefinite di volta in volta.

Come rappresentato nella Sezione 6.2 il controller e il model possiedono entrambi un riferimento l'un con l'altro. Sebbene un riferimento model→controller possa risultare scontato, il contrario si è reso necessario in quanto si sono registrate delle computazioni autonome di alcuni valori mostrati, per cui un riferimento al controller per la notifica dell'aggiornamento dei valori è doveroso.

Nella Figura 6.7 è riportato lo schema UML delle interfacce costituenti la parte model del simulatore.

Di tali interfacce verrà poi riportata una parte di implementazione nel capitolo dedicato.

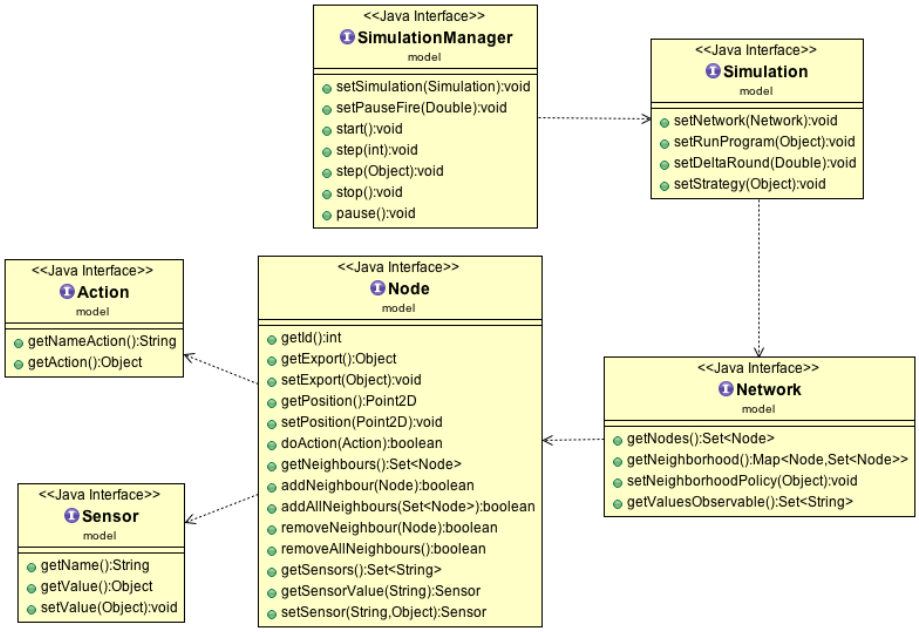


Figura 6.7: Schema generale del Model.

6.3 Bridge

Il collegamento tra il simulatore e Scafi si è basato sul pattern di progettazione *Bridge*. Tale pattern permette di separare l'interfaccia di una classe dalla sua implementazione. In tal modo si può usare l'ereditarietà per fare evolvere l'interfaccia o l'implementazione in modo separato. Siccome le classi che devono implementare le interfacce appena descritte devono comunque avere un riferimento alle classi presenti in Scafi per l'avvio della simulazione aggregata, questo pattern è risultato il più adatto da utilizzare. I componenti sviluppati al fine di modellare questa parte sono stati realizzati tutti in Java fatta eccezione per la classe **SimulationImpl** e della relativa interfaccia **Simulation** implementate in Scala. Per far integrare i due framework (Scafi e il simulatore) è stato necessario definire all'interno del simulatore stesso un oggetto apposito denominato **BasicSpatialIncarnation**. Quest'ultimo, estendendo la classe BasicAbstractSpatialSimulationIncarnation di Scafi, permette l'effettivo funzionamento del sistema. Infatti viene utilizzato all'interno della classe SimulationImpl al fine di definire tutto il contesto specifico per l'avvio della simulazione, ad esempio fornisce l'implementazione del network della simulazione attraverso la classe **SpaceAwareSimulator**. Tutte le classi implementate per la creazione del brid-

ge sono state inserite in una *directory-root* dedicata denominata *scala*. Il design della sezione appena descritta è riportata nella Figura 6.8.

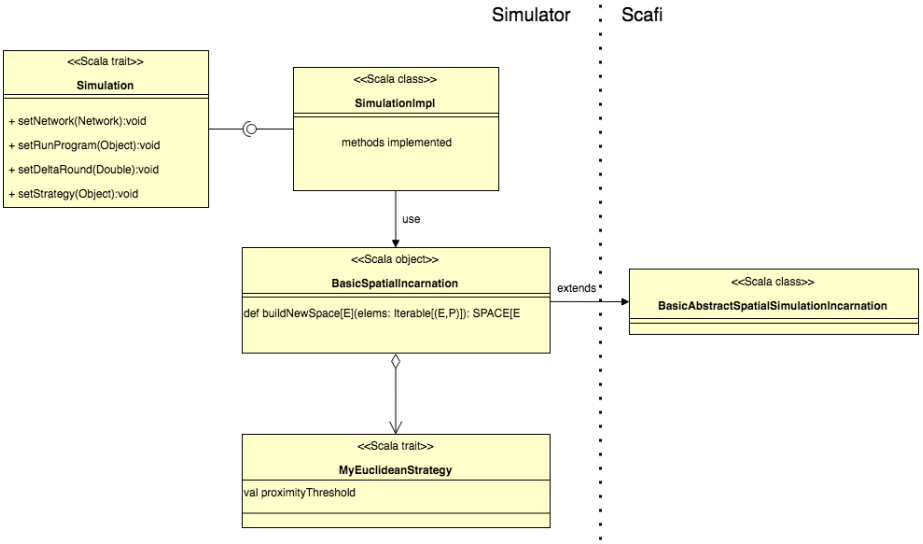


Figura 6.8: Schema design del Bridge.

Capitolo 7

Implementazione

Come già anticipato il simulatore è stato implementato utilizzando due diversi linguaggi di programmazione: Scala e Java. In questo capitolo vengono presentati in primo luogo i dettagli implementativi più rilevanti, soprattutto per quanto concerne la convivenza dei due linguaggi. In seguito viene invece presentata l'implementazione di una demo creata per testare l'efficacia del progetto.

7.1 Dettagli Implementativi

Per l'implementazione del simulatore si è utilizzato uno dei costrutti più avanzati di Java 8, le lambda, così da rendere più fluida le lettura di liste e set, inoltre si è fatto anche uso di alcuni aspetti avanzati della libreria java.swing per la realizzazione dell'ambiente grafico:

GridBagLayout: utilizzato nell'impostazione delle form dedicate all'interazione con l'utente. Questo Layout è uno dei più flessibili e complessi tra quelli messi a disposizione da Java. Infatti permette di disporre i componenti della view in una griglia consentendo di gestirla nel modo più appropriato. Le dimensioni di righe e colonne possono essere personalizzate rispettivamente in larghezza e altezza; un qualsiasi componente può inoltre occupare più celle sia in verticale che in orizzontale. Essenzialmente quindi il GridBagLayout inserisce un elemento in un rettangolo dimensionato a piacere all'interno di una griglia. Le proprietà per l'inserimento degli elementi nella griglia sono gestite da un oggetto denominato *GridBagConstraints*.

Il GridBagLayout è stato utilizzato nella definizione delle classi **ConfigurationPanel** (per la configurazione iniziale della simulazione) e **SensorOptionPane** (per la manipolazione dei sensori).

Nella Figura 7.1 è riportato l'aspetto finale del ConfigurationPanel di cui, a titolo di esempio, se ne riporta una porzione di codice.

Listing 7.1: Esempio di utilizzo del GridBagLayout.

```
1 JPanel p1 = new JPanel(new GridBagLayout());
2 GridBagConstraints gbc = new GridBagConstraints();
3 gbc.insets = new Insets(5, 0, 0, 10); //padding esterno
4 //creazione delle righe della form
5 insertRow("Number of nodes:", nodeNumberField, p1);
6 insertRow("Neighborhood policy:", neinghborsAreaField,
      p1);
7 insertRow("Pause fire:", deltaRoundField, p1);
```

Listing 7.2: Esempio di utilizzo del GridBagLayout.

```
1
2 //Inserimento di una riga nella griglia
3 private void insertRow(String name, JComponent comp,
      JPanel p){
4 //inserimento nome-chiave a sinistra
5     gbc.gridx = 0;  //colonna
6     gbc.gridy = y;  //riga
7     gbc.anchor = GridBagConstraints.LINE_END;
          //allineamento
8     p.add(new JLabel(name), gbc);
9 //inserimento componente a destra
10    gbc.gridx = 1;
11    gbc.gridy = y;
12    gbc.anchor = GridBagConstraints.LINE_START;
13    p.add(comp, gbc);
14    y++; //viene incrementato ogni inserimento di riga
15 }
```

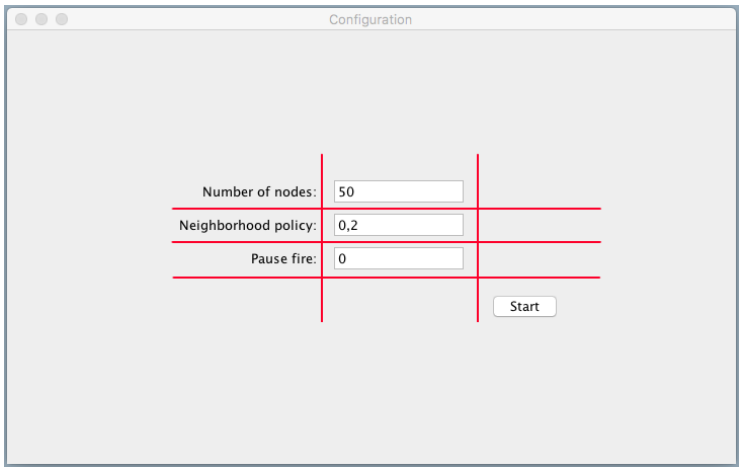


Figura 7.1: Pannello di configurazione iniziale.

JDesktopPane: è il contenitore predisposto per l'utilizzo di più *JInternalFrame* ed è utilizzato nella definizione del **SimulationPanel**. Entrambe le classi (JDesktopPane e JInternalFrame) discendono da JComponent e pertanto ne forniscono tutti i metodi. Estendendo anche *JLayeredPane*, un oggetto JDesktopPane supporta anche tutti i metodi per la gestione di frame multipli e per la loro sovrapposizione. Di seguito è riportato il metodo per la creazione di un **GuiNode** (JInternalFrame) e per il suo inserimento nel SimulationPanel (JDesktopPane). Per quanto riguarda l'inserimento del pannello riportante i collegamenti della rete, esso è stato aggiunto al SimulationPanel con livello 1 in quanto il LayeredPane prevede una priorità maggiore per i livelli con un numero inferiore.

Listing 7.3: Metodo start della classe Controller.

```
1 public void startSimulation(int nNodes, Object plngd,
2  Object runProgram, Double deltaRound, Object strategy)
      {
3         for(int i = 0; i<numNodes; i++){
4             GuiNode guiNode = new GuiNode();
5             //inserimento GuiNode nel JDesktopPane
6             gui.getSimulationPanel().add(guiNode, 0);
7             ...
8         }
9  }
```

Le lambda sono state utilizzate in maniera pervasiva su tutto il codice, principalmente per la definizione degli *ActionListener* dei diversi componenti e per la manipolazione di set e mappe. Nel seguito si riporta un esempio di utilizzo di questo costrutto dove è presente il metodo *paintComponent(Graphics)* del pannello dedicato alla rappresentazione della rete di interconnessione tra i nodi (NeighboursPanel). In tale metodo viene richiesto al controller un report di vicinanza di ciascun nodo rispetto a tutti gli altri. Il controller, sottoponendo la richiesta al model, restituisce una Map<Node, Set<Node> > che verrà poi manipolata opportunamente attraverso le lambda.

Listing 7.4: Metodo paintComponent(Graphics) della classe NeighboursPanel.

```
1  @Overrid
2  protected void paintComponent(Graphics g) {
3          super.paintComponent(g);
4      this.removeAll();
5      g.setColor(Color.black);
6      //call the neighborhood to the network object
7      controller.getNeighborhood().forEach((n,nghb)->{
8          nghb.forEach(ng->{
9              g.drawLine(middlePoint(n).x, middlePoint(n).y,
10                         middlePoint(ng).x,
                               middlePoint(ng).y);
11         });
12     });
13 }
```

7.2 Bridge

In merito all'unione tra il simulatore e l'applicazione Scafi si è provveduto ad implementare le interfacce predisposte dalle API definite nel model. Le classi concreate sono state implementate per lo più in Java (SimulationManagerImpl, NodeImpl...), l'unica implementata in Scala è SimulationImpl. Al fine di implementare al meglio le funzioni fornite da questa classe se ne sono create altre due distinte per incapsulare esclusivamente la logica dei run-program:

- GradientProgram, per la demo;

- ChannelProgram;

Un punto di notevole interesse riguarda l'implementazione della funzione *setRunProgram* contenuta nella classe SimulationImpl, riportata qui di seguito.

Listing 7.5: Funzione setRunProgram realizzata in Scala.

```
1  def setRunProgram(program: Any): Unit = {
2
3      //mappa i nodi con gli id
4      val mapperPos: java.util.function.Function[Node,
           Point2D] = new java.util.function.Function[Node,
           Point2D] {
5        override def apply(n: Node): Point2D = new Point2D
             (n.getPosition.getX, n.getPosition.getY);
6      }
7
8      //mappa i nodi con le posizioni
9      val mapperId : java.util.function.Function[Node, Int] =
           new java.util.function.Function[Node, Int] {
10       override def apply(n: Node): Int = n.getId
11     }
12
13     //creazione della mappa
14     val idList: List[Int] =
           List(this.network.getNodes.stream().map[Int](mapperId)
15                                   .iterator().asScala.toList:
                                         _*)
16     val posList: List[Point2D] =
           List(this.network.getNodes.stream().map[Point2D](mapperPos)
17                                   .iterator().asScala.toList:
                                         _*)
18
19     val devsToPos: Map[Int, Point2D] =
           idList.zip(posList).toMap //mappa id->posizione
20     net = new SpaceAwareSimulator(
21       space = new Basic3DSpace(devsToPos,
             proximityThreshold =
```

```
            this.network.getPolicy().asInstanceOf[Double]),
            //valore settato da gui
22        devs = devsToPos.map { case (d, p) => d -> new
              DevInfo(d, p,
23          lsns => if (lsns == "sensor" && d == 3) 1 else 0,
24          nsns => nbr => null)
25        }
26      )
27
28      val sensors: List[SensorEnum] =
            scala.List(SensorEnum.values: _*)
29
30      sensors.foreach(se => net.addSensor(se.getName,
            se.getValue)) //setto i sensori al sistema
31
32      val ap = new GradientProgram  //run program (Gradiente)
33      this.runProgram = () => net.exec(ap)
34    }
```

Si riporta infine un pezzo di codice esemplare per quanto riguarda l'interoperabilità tra Java e Scala. Infatti all'interno della classe **Controller** scritta in Java viene creata un'istanza della classe SimulationImpl scritta in Scala. Tutto questo semplicemente dichiarando l'import adeguato. Il metodo riportato nella Sezione 7.2 è lo stesso già proposto in precedenza dove si è discusso l'utilizzo del JDesktopPane, tuttavia se ne riporta l'intero codice e se ne evidenzia la riga in cui viene istanziata la classe Scala.

```
//metodi gestione simulazione
public void startSimulation(final int numNodes, Object policyNeighborhood, Object runProgram, Double deltaRound, Object strategy) {

    for(int i = 0; i<numNodes; i++){
        GuiNode guiNode = new GuiNode();
        Node node = new NodeImpl(i, new Point2D.Double( new Random().nextDouble(), new Random().nextDouble()));
        guiNode.setLocation(Utils.calculatedGuiNodePosition(node.getPosition()));
        this.nodes.put(node, guiNode);
        gui.getSimulationPanel().add(guiNode, 0);   //aggiungo il JInternalFrame con livello più alto
    }

    Simulation simulation = new SimulationImpl();   //creazione simulazione SCALA

    simulation.setNetwork(new NetworkImpl(this.nodes.keySet(), policyNeighborhood));
    simulation.setDeltaRound(deltaRound);
    simulation.setRunProgram(new GradientProgram());//runProgram);
    simulation.setStrategy(strategy);
    simManager.setSimulation(simulation);
    simManager.setPauseFire(deltaRound);    //pausa tra un round e un altro
    simManager.start(); //avvio della simulazione

    simManager.getSimulation().setSensor(SensorEnum.SOURCE.getName(), nodes.keySet(), Boolean.valueOf(false));
    simManager.getSimulation().setSensor(SensorEnum.OBSTACLE.getName(), nodes.keySet(), Boolean.valueOf(false));

    controllerUtility.addObservation();   //aggiungo le osservazioni
    controllerUtility.addAction();        //aggiungo le azioni
    controllerUtility.enableMenu(true);

    System.out.println("START");
}
```

Figura 7.2: Metodo startSimulation() della classe Controller.

7.3 Demo

Una volta terminata l'implementazione del simulatore si è proceduto alla creazione del bridge tra il simulatore e l'applicazione aggregata già predisposta dal dott.Casadei. Per procedere con l'interconnessione delle due entità, si è preliminarmente provveduto a:

- istanziare il simulatore all'interno del metodo start del Controller, utilizzando la configurazione di rete inserita tramite GUI;
- gestire il ciclo della simulazione (start-stop-pausa) all'interno della classe **SimulationManagerImpl**.

Ciò consente di poter inglobare all'interno dell'architettura della GUI la logica per l'esecuzione della simulazione Scafi.

Finita questa fase si è proceduto all'effettiva implementazione del bridge. Per semplicità, e al fine di separare bene le diverse parti del progetto, le classi create per l'inserimento dell'applicativo aggregato sono state inserite in una nuova Sources Root denominata *scala*. All'interno di questa nuova directory si è creato in primis un oggetto **BasicSpatialIncarnation** il quale, estendendo la classe astratta **BasicAbstractSpatialSimulationIncarnation** di Scafi, permette di eseguire una simulazione posizionando i nodi in un determinato

spazio virtualmente tridimensionale. Dopodiché è stata implementata la classe SimulationImpl, nella quale è definita tutta la logica di creazione della simulazione all'interno del metodo *setRunProgram(Any)*: l'argomento accettato da tale metodo è proprio il programma aggregato da eseguire. Questa strategia favorisce la gestione di un futuro caricamento dinamico della classe rappresentante il programma aggregato. Tuttavia nella demo sviluppata il caricamento avviene staticamente all'interno della classe SimulationImpl attraverso l'istruzione `val ap =new GradientProgram //run program (Gradiente).`

Un altro aspetto definito all'interno di questa classe riguarda il posizionamento dei nodi tramite la definizione della seguente funzione:

```
override def setPosition(n: Node): Unit =
    net.setPosition(n.getId,
new Point2D(n.getPosition.getX, n.getPosition.getY))
```

In seguito ci si è concentrati sulla gestione dei sensori: per ciascuno di essi si è provveduto al mapping sul simulatore attraverso il codice sottostante.

```
 val sensors: List[SensorEnum] =
     scala.List(SensorEnum.values: _*)
    sensors.foreach(se => net.addSensor(se.getName,
        se.getValue))
```

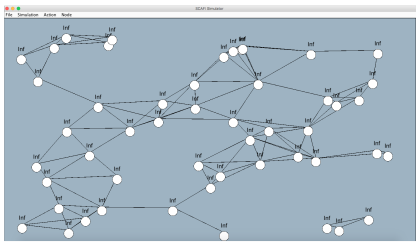
Infine si è implementato un metodo per consentire la modifica del valore dei sensori durante l'esecuzione della simulazione, rendendo manifesto l'utilizzo contestuale di Java e Scala.

Listing 7.6: Funzione per la modifica del valore dei sensori.

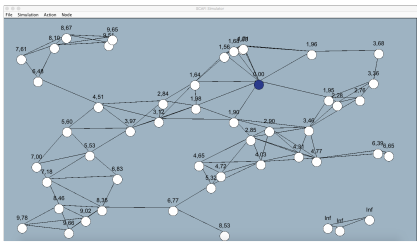
```
def setSensor(sensor: String, ids: util.Set[Node], value:
    AnyRef): Unit = {

    val mapperId : java.util.function.Function[Node, Int] =
        new java.util.function.Function[Node, Int] {
      override def apply(n: Node): Int = n.getId
    }
    val idSet: Set[Int] =
        Set[Int](ids.stream().map[Int](mapperId)
```

```
7       .iterator().asScala.toSeq: _*)
8       net.chgSensorValue(sensor, idSet, value)
9     }
```

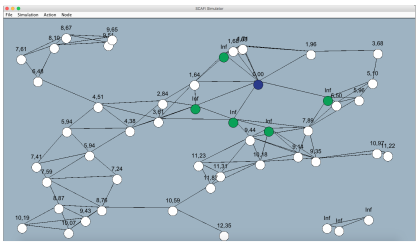
Nelle immagini seguenti sono riportati i vari step della simulazione del calcolo del gradiente su una rete formata da 50 nodi. I nodi blu rappresentano i nodi sorgente, mentre quelli verdi gli ostacoli. La stringa *"Inf"* sta a significare un valore infinito dell'*Export*.



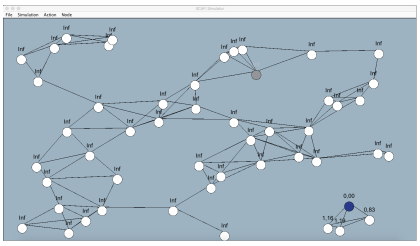
(a) Rete iniziale formata solo da nodi semplici.



(b) Creazione di un nodo sorgente → Export = 0.00.



(c) Creazione di più nodi ostacolo → Export = Inf.



(d) Spostamento del nodo sorgente.

Figura 7.3: Calcolo del gradiente.

Capitolo 8

Strumenti

In questa sezione vengono presentati i diversi tool utilizzati per lo sviluppo e il mantenimento del progetto. Verranno presentati in ordine gli strumenti per lo sviluppo del codice e per il suo *version control*. Per quanto concerne la creazione delle diverse icone utilizzate nel simulatore si è utilizzato *Adobe Photoshop CC*.

8.1 IntelliJ IDEA

Il progetto del simulatore è stato realizzato utilizzando l'IDE di sviluppo *IntelliJ IDEA*. Tale scelta è stata dettata dato dal fatto che questo ambiente di sviluppo supporta nativamente la creazioni di progetti Scala e Java. La differenza con *Eclipse*, un altro IDE di largo utilizzo nella pratica, consiste nel supporto nativo esclusivamente Java-oriented di quest'ultimo. Inoltre si è stato testato il plugin per l'utilizzo del linguaggio di programmazione Scala, trovandolo tuttavia meno intuitivo e fruibile rispetto ad IntelliJ IDEA. Sempre nell'ambiente Eclipse si è fatto uso del plugin *Object AID* per la creazione degli schemi UML presenti nell'elaborato.

8.2 SBT

Come strumento di continuos integration si è utilizzato *SBT*, un framework open source creato per progetti implementati in Scala e Java.

Questo strumento utilizza un basso numero di concetti così da riuscire a definire in maniera flessibile il file build.sbt in cui vengono dichiarate le dipendenze e le caratteristiche del progetto. Le principali funzionalità di SBT sono:

- il supporto nativo per la compilazione del codice Scala e l'integrazione con molti framework di test;
- inserimento di codice Scala utilizzando un DSL nel file build.sbt;
- gestione delle dipendenze utilizzando Ivy (il quale si appoggia sul repository Maven);
- gestione della continuos intergration;
- integrazione con l'interprete Scala per una rapida compilazione e debugging;
- supporto alla contemporanea sussistenza di progetti Scala e Java.

Quest'ultima caratteristica si è rivelata fondamentale per la realizzazione del progetto. Di seguito è riportata la dichiarazione del simulatore all'interno del file build.sbt.

```
1 // 'guiScafi' project definition
2 lazy val guiScafi = project.
3         dependsOn(core, simulator).
4     settings(commonSettings: _*).
5     settings(
6         name := "scafi-simulator-gui",
7         javacOptions ++= Seq("-source", "1.8", "-target",
              "1.8")
8     )
```

8.3 Git e Bitbucket

Git è stato utilizzato come DVCS (Distributed Version Control System) attribuendogli il compito di mantenere traccia dei cambiamenti riportati nel codice durante lo sviluppo. In questo modo è stato quindi possibile tener traccia di

tutte le versioni salvate (o committate) in modo tale da poterne richiamare una specifica in un secondo momento.

Sebbene si sarebbe anche potuto utilizzare *Mercurial*, la scelta è però ricaduta su Git essenzialmente per due ragioni:

- Git fornisce più funzionalità rispetto a Mercurial;
- dato il tracciamento di SCAFI con Git e in virtù del sistema di suddivisione in branch di quest'ultimo, è stato possibile sviluppare il progetto parallelamente a SCAFI.

Come repository web based per Git si è utilizzato *BitBucket* del quale ci si è serviti del servizio di pull request e della navigazione dello storico.

Capitolo 9

Guida utente

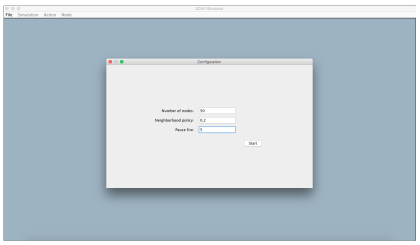
Questo simulatore è stato progettato partendo dal presupposto che gli utenti che ne usufruiranno abbiano comunque le competenze adeguate per impostare e gestire la simulazione di un'applicazione aggregata. Pertanto la guida utente che segue si soffermerà solamente su di un insieme ristretto di funzionalità. Come prima cosa deve poter essere necessario configurare una simulazione in termini delle sue caratteristiche, ovvero:

numero di nodi: totale dei nodi che compongono la rete;

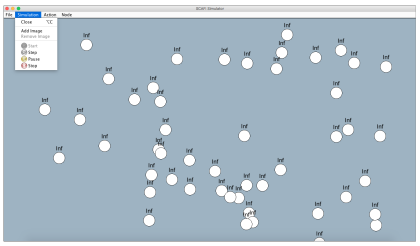
politiche di vicinato: descrive il raggio entro il quale due nodi vengono riconosciuti come vicini;

pause fire: determina la velocità con cui si svolge la simulazione e in particolare, descrive il tempo di pausa tra un round e l'altro della simulazione. Dunque si richiede di inserire un qualsiasi numero, prestando ovviamente attenzione al fatto che la velocità di computazione è proporzionale alla numerosità dei nodi presenti.

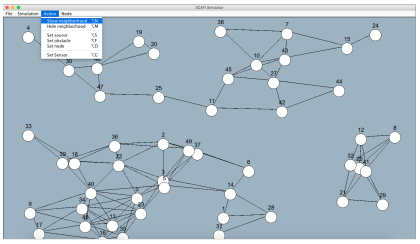
Tutti questi campi hanno come limite inferiore il valore 0 in quanto, per tutti e tre, un termine negativo non avrebbe alcun senso. Una volta configurata una qualsiasi simulazione, è possibile interagire con essa attraverso le varie azioni selezionabili dai menù **Simulation**, **Action** e **Node**. Nella Figura 9.1 sono riportati alcuni esempi di utilizzo del simulatore.



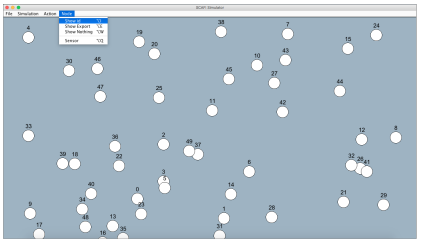
(a) Configurazione.



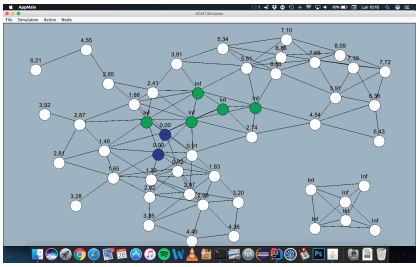
(b) Interazione simulazione.



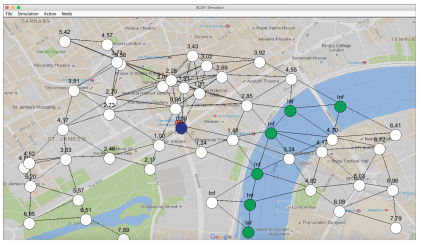
(c) Azioni.



(d) Visualizzazione ID dei nodi.



(e) Esempio simulazione.



(f) Immagine di sfondo.

Figura 9.1: Screenshot del simulatore.

Capitolo 10

Conclusioni

10.1 Commenti finali

Il progetto di un simulatore per la piattaforma Scafi è nato da una forte esigenza di testing e soprattutto rappresentazione grafica di un sistema distribuito aggregato. Lo studio del linguaggio di programmazione Scala, svolto preliminarmente all'inizio della tesi, è stato oggetto di vivo interesse e fonte di stimoli. Ci si è concentrati primariamente sulla comprensione della filosofia di base e dei concetti fondamentali di Scala, demandando gli approfondimenti volti al fine di delineare tutte le possibili sfaccettature che tali linguaggio può offrire ad un tempo futuro. Contestualmente a ciò è stata ampliata la conoscenza della programmazione funzionale, la quale era stata solamente accennata durante il corso di Programmazione ad oggetti. Non essendo stato posto alcun obbiettivo in particolare riguardo a Scala, si ritiene che la fase di studio sia stata produttiva, in quanto il fatto di possedere delle nozioni basilari sulla strutturazione del particolare linguaggio è stato di fondamentale importanza, se non indispensabile, per la creazione del bridge tra l'applicazione distribuita e il simulatore. Nella fase di analisi dei requisiti sono state estrapolate delle solide fondamenta sulle quali costruire l'applicazione: particolare interesse ha suscitato lo studio e soprattutto la comprensione del funzionamento un sistema distribuito aggregato. In seguito si sono approfonditi i passaggi riguardanti la creazione di una simulazione sulla piattaforma Scafi e su come ottenere le informazioni mantenute dai componenti attivi da mostrare a video. Lo sviluppo della view si è rivelato alquanto impegnativo: infatti, nei precedenti progetti svolti durante il corso di

studio, tale aspetto (la GUI) è quasi sempre stato sviluppato da altri componenti dei team di lavoro. Il lavoro di tesi è stata quindi un'ottima occasione per poter approfondire e ampliare le conoscenze di un altro importante aspetto del linguaggio Java, rivelandosi inoltre fonte di soddisfazione. La creazione del bridge ha permesso di delineare in maniera chiara e puntuale le modalità con cui Scala e Java cooperano e, grazie soprattutto al supporto del dott. Casadei, si è riusciti ad implementare le classi Scala in maniera anche appagante. In conclusione si ritiene che il lavoro svolto abbia centrato appieno tutti gli obbiettivi prefissati inizialmente e che essi siano stati sviluppati in modo corretto. Ciò consentirà quindi di partire da basi molto valide per il particolare sviluppo futuro presentato nel paragrafo successivo.

10.2 Sviluppi futuri

Il progetto di tesi non intende certo essere un simulatore completo ma è stato implementato così da facilitare il suo sviluppo futuro nel modo più semplice ed immediato possibile. Tale sviluppo futuro verterà sui seguenti aspetti da gestire e modellare opportunamente:

- creazione di una demo basata sul *ChannelProgram*;
- gestione più sofisticata delle icone dei nodi (ad esempio gestendo una correlazione tra il gradiente calcolato e il colore del nodo);
- scelta della strategia d'esecuzione tra piattaforma p2p e server-based;
- inserimento dinamico della classe in cui è definito il run program;
- eventuale disposizione dei nodi in una griglia predefinita.

Bibliografia

[1] Java api documentation - version 8. https://docs.oracle.com/javase/8/docs/api/.

[2] Stackoverflow. http://stackoverflow.com/questions/tagged/java.

[3] Scala api documentation - version 2.12.0. `http://www.scala-lang.org/api/current`, 2010 (accessed December 7, 2014).

[4] James Elliott Marc Loy David Wood Brian Cole, Robert Eckstein. *Java Swing*. O'Reilly, second edition, 2002.

[5] Roberto Casadei. *Aggregate Programming in Scala: a Core Library and Actor-Based Platform for Distributed Computational Fields*. PhD thesis, Scuola di Ingegneria e Archiettura, 2015.

[6] Simone Costanzi. *Integrazione di piattaforme d'esecuzione e simulazione in una Toolchain Scala per aggregate programming*. PhD thesis, Scuola di Ingegneria e Archiettura, 2015.

[7] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns Element of Reusable Object-Oriented Software*. Addisom-Wesley, 1995.

[8] Mirko Viroli Jacob Beal, Danilo Pianini. aggregate programming for the internet of things. *Qmags*, 2010.

[9] Bill Venners Martin Odersky, Lex Spoon. *Programming in Scala*. artima, ii edition, 2010.

[10] Mirko Viroli Roberto Casadei. Towards aggregate programming in scala. *Proceeding PMLDC '16 First Workshop on Programming Models and Languages for Distributed Computing*, 2016.

[11] Gianluca Tartaggia. *Il linguaggio di programmazione scala.* PhD thesis, Ingegneria Informatica Padova, 2012.