ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA

SCUOLA DI SCIENZE

CORSO DI LAUREA TRIENNALE IN INGEGNERIA E SCIENZE INFORMATICHE

SVILUPPO DI UN FRONT-END DI SIMULAZIONE PER APPLICAZIONI AGGREGATE NEL FRAMEWORK SCAFI

Tesi in

Programmazione ad oggetti

Relatore:
prof. Viroli Mirko

Presentata da:
Aguzzi Gianluca

Correlatore:
dott. Casadei Roberto

Sessione II
Anno Accademico 2017/2018

Alle mie radici che mi hanno reso
l'albero robusto che sono oggi.

Indice

Introduzione

Nel corso degli ultimi anni i sistemi informatici hanno avuto uno sviluppo incrementale, non solo incrementando la potenza computazionale, ma anche riducendo drasticamente le dimensioni delle unità di calcolo. Questo ha portato alla nascita di nuovi dispositivi indossabili (e portabili) come smartphone e smartwatch divenuti, per la maggior parte della popolazione mondiale, beni di prima necessità. La miniaturizzazione delle macchine computazionali, inoltre, ha portato alla creazione di sensori intelligenti che possono essere installati nei posti più disparati. Quando quest'enorme insieme di dispositivi si connette in una rete comune come Internet, allora si inizia a parlare di *Internet of Thing* (**IoT**). In poco tempo si è passati da quello che è stato il paradigma di computazione standard per decine di anni, cioè il desktop computing (la computazione avviene in un singolo dispotivo), a *pervasive computing* (la computazione non è allocata in un solo dispositivo ma viene eseguito da un insieme di dispositivi che riescono a portare a termine un risultato comune). I linguaggi e i paradigmi di programmazione standard non sono riusciti a garantire delle tecniche di programmazione atte a gestire tali sistemi, perciò negli anni si è provato ad approcciarsi al problema in diversi modi. Le tecniche di Aggregate Programming (basate sul concetto di *field calculus*), ad esempio, aggiungono un livello d'astrazione tale da nascondere quelli che sono i dettagli legati alla rete (tipologia di connessione, topologia..) descrivendo l'algoritmo da eseguire su un insieme di dispositivi in termini del loro comportamento aggregato. Uno dei framework che permettono di utilizzare il paradigma di Aggregate Programming è `ScaFi`, sviluppato dal prof. Mirko Viroli e dal dott. Roberto Casadei. `ScaFi` mette a disposizione un **API** per descrivere applicazioni aggregate attraverso la sintassi del *field calculus* e una piattaforma distribuita, Akka based, su cui poter sviluppare applicazioni aggregate.

Il comportamento di un programma aggregato, vuoi per la nuova tipologia di pensiero, vuoi per la sua complessità, non è sempre facile da definire e perciò, per verificare il comportamento di questi sistemi, è nata l'esigenza di poter simulare sistemi reali dove avviare programmi aggregati. In questo

modo si ha un risultato visivo della simulazione e si verifica la correttezza della logica del programma. Data questa esigenza, l'obiettivo di questa tesi di laurea è quello di sviluppare un front-end di simulazioni aggregate descritte in `ScaFi` per permettere di testare e visualizzare scenari di utilizzo di tale framework. Per front-end non si intende solamente un semplice sistema che renderizzi dei dati prodotti dalle simulazioni aggregate ma, che inoltre aggiunga l'interazione con il sistema aggregato, la configurazione e l'avvio di simulazione. Il sistema in questione non deve funzionare per una classe predefinita di programmi aggregati, ma deve descrivere una serie di concetti generali che permettano di visualizzare il risultato a fronte di diverse configurazioni di un generico programma. Una nota importante dell'applicativo finale è quella di garantire l'avvio del sistema in più modalità (via interfaccia grafica, via console), oltre quella via codice.

L'iter di produzione del software in questione segue un modello di tipo *evolutivo* e fa uso di un prototipo che si è andato a delineare ed evolvere nel tempo. Quest'approccio, risultato vincente per questo problema, ha reso possibile raffinare quelli che erano i requisiti.

La tesi è articolata in sei capitoli: nel primo vengono introdotti gli studi pregressi realizzati durante il periodo di tirocinio svolto presso l'Università di Bologna, utili a comprendere l'insieme di scelte effettuate nelle altre fasi dello studio. Nel secondo capitolo viene descritta la fase di analisi del modello da realizzare, completa di elenco dei requisiti che il front-end deve rispettare, in questa fase in modo particolare, c'è stato uno scambio di informazioni costante con il prof. Mirko Viroli e il dott. Roberto Casadei nei panni dei 'committenti' nel progetto. Nel terzo capitolo si descrive la fase di progettazione architetturale, fornendo informazioni su come il software realizza quanto richiesto in fase di analisi in termini di architettura. Si descrive, quindi, la struttura generale del front-end e si spiega il perché delle varie scelte presenti in questa fase. Nel quarto capitolo vengono descritte, invece, quelle che sono le scelte di design dettagliato, in particolare si descrive com'è possibile interfacciare il front-end con il framework `ScaFi` e come avviare e descrivere una simulazione aggregata. Nel quinto capitolo vengono mostrate le scelte implementative per la realizzazione del software e vengono mostrati alcuni screenshot con i risultati ottenuti e, infine, nel sesto si offre una panoramica di quello che è il risultato finale mostrando la struttura dell'interfaccia grafica corredato di analisi delle performance estratte da alcune delle possibili simulazioni eseguibili sul front-end.

Al termine della tesi si è creato un front-end funzionante con diverse tipologie di simulazioni aggregate. L'architettura del sistema è resa flessibile dall'inserimento delle varie funzionalità e mettendo a disposizione un **API** che descriva un generico sistema aggregato e le varie azioni applicabili su

di esso in base alle logiche descritte all'interno del programma aggregato. Nelle conclusioni viene mostrato come il sistema non alteri quelle che sono le perfomance correnti del simulatore *ScaFi* permettendo così di utilizzarlo come un sistema di testing robusto e facendo in modo di valutare i programmi aggregati in diverse condizioni. I risultati sono esplicitati nelle conclusioni finali di questa tesi.

Capitolo 1

Background

L'analisi e la progettazione del front-end per la visualizzazione di simulazioni aggregate in `ScaFi` realizzato in questa tesi sono avvenuti a seguito dello studio di altre tecnologie e piattaforme.

Molte delle scelte prese durante lo sviluppo del progetto, sono state influenzate sia dalle tecnologie utilizzate, sia dalla struttura del simulatore preesistente, il tutto per arricchire il lavoro finale superando gli attuali limiti. Data la natura del sistema applicativo è necessario, innanzitutto, introdurre i concetti principali di *aggregate computing* e poi effettuare una panoramica generale sul linguaggio di programmazione utilizzato per lo sviluppo del progetto, ossia Scala. Nelle ultime due sezioni si analizzerà cos'è `ScaFi`, come ha reso possibile l'esecuzione di simulazioni aggregate e qual è lo stato corrente del simulatore.

Le varie sezioni non saranno esaustive, ma verrà dettagliato solo quanto necessario a comprendere il progetto di questa tesi.

1.1 Aggregate computing

La crescita esponenziale di dispositivi informatici indossabili, portatili ed embedded (smartphone, smartwatch, ...) ha avuto un grande impatto sull'intera popolazione. Questo insieme di entità connesse (figura 1.1) ha dato luogo a ciò che è definito *Internet of Things* (**IoT**), un modo di vedere gli oggetti comuni totalmente rivoluzionario e che ha portato alla necessità di programmare e far interagire questo insieme di entità. Inizialmente, sono stati usati vecchi paradigmi di programmazione (*single device view point*) risultati, però, inadeguati alla gestione di device embedded pervasivi a causa del loro crescente numero e del loro forte legame con concetti spaziali[1].

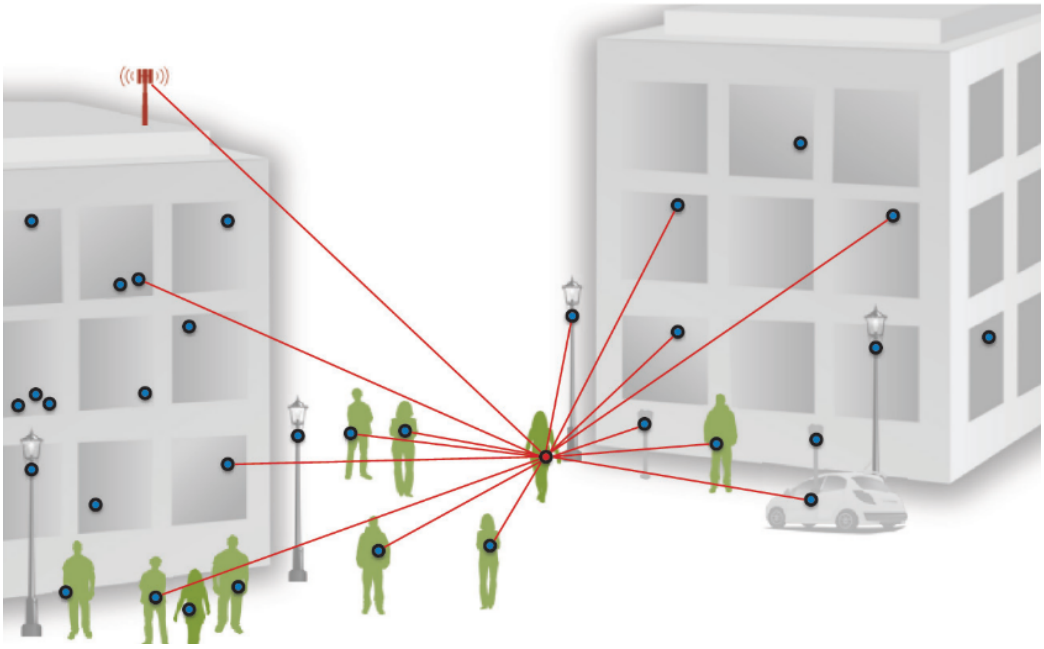


Figura 1.1: un possibile scenario di una situazione urbana dove si crea una rete tra persone, mezzi e 'cose' [1]

Negli anni sono nati tanti possibili approcci alternativi, uno di questi è *Aggregate Computing*: approccio *large scale* per ingegnerizzare sistemi distribuiti e con il quale è possibile definire un generico comportamento collettivo del sistema in modo semplice, componibile e ad alto livello. L'idea di base è quella di passare dalla visione del singolo device ad una aggregata, dove il sistema viene visto come una singola macchina computazionale.[1] [3].

Con questo tipo di paradigma, si possono definire delle funzionalità che permettono di creare applicazioni in diversi scenari come la gestione di situazioni critiche in eventi pubblici affollati (rilevamento di zone affollate, calcolo di percorsi alternativi, ...).

1.1.1 Modello computazionale

Un sistema aggregato consiste in un certo numero di *computational device*, ognuno dei quali esegue un *round* in tempo asincrono durante tutta la computazione. Un *export* è il valore dello stato dell'ultima computazione eseguita e durante l'esecuzione verrà propagato in modo ripetitivo ai vicini.

Ogni *round* è definito da un insieme di step: [15][3]

[1] **creazione dello stato del contesto di esecuzione**: include l'ultimo valore calcolato, i recenti *export* ricevuti dal vicinato e una foto dei

sensori locali installati sul dispositivo;

[2] **esecuzione locale del *programma aggregato***: basato sullo stato del contesto, genera un nuovo *export*;

[3] **propagazione dell'*export* a tutto il vicinato**;

[4] **attivazione degli attuatori**: eseguito usando come input il risultato prodotto dalla computazione.

Un generico comportamento globale del sistema può essere visto come una *computational dust* composta da azioni atomiche (*round*)[15].
Questo modello può essere descritto attraverso i *computational field* e il *field calculus*.

1.1.2 Radici e costrutti di base

Un *computational field* (figura 1.2) cerca di generalizzare il concetto fisico di campo magnetico: è una struttura dati distribuita che associa ad ogni device, localizzato in un punto dello spazio-tempo, un valore; tipi di campi computazionali in un contesto IoT sono, ad esempio, una collezione di temperature prodotte da un sensore, l'insieme di dati prodotto dall'accelerometro di uno smartphone, ecc.. *Field calculus* descrive l'insieme di primitive per rendere possibile la manipolazione di un campo computazionale in maniera globale ed espressiva e, in particolare, è un *modello teorico* usato da aggregate programming (come lo è il *lambda calculus* per un linguaggio funzionale). In aggregate computing vengono descritti dei costrutti base che sono:

- **funzioni**, $b(s_1; ...; s_n)$ applica la funzione b agli argomenti $s_1, ..., s_n$;
- **dinamiche**, $rep(x < -v)\{s^1; ...; s^n\}$ definisce lo stato della variabile x inizializzata con il valore v, che verrà poi aggiornata in modo periodico in base alla valutazione di $s_1, ..., s_n$;
- **interazioni**, $nbr(s)$ raccoglie una mappa per ciascuno dei vicini con il loro ultimo valore di s;

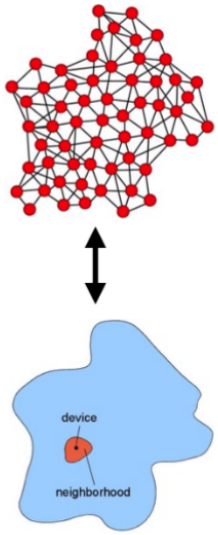


Figura 1.2: vista fisica secondo la logica dei campi computazionali di una rete di dispositivi [16]

– **restrizioni**, $if(cond)\{s_1;...;s_n\}$ else $\{s_1^{'};...;s_n^{'}\}$
divide la rete in due regioni: se cond è vera, vengono valutati $s_1, ..., s_n$, altrimenti lo saranno $s_1^{'}, ..., s_n^{'}$.

Questi costrutti base supportano portabilità, indipendenza dalla struttura utilizzata e possibilità di interagire con servizi non aggregati.[1][14][15]

1.2 Scala

Scala (**S**calable **L**anguage) nome dovuto alla capacità di questo linguaggio di crescere in base al dominio applicativo dell'utente, può essere usato sia per scrivere semplici script, sia per creare vere e proprie applicazioni complesse. Nasce con l'obiettivo di integrare le funzionalità e le caratteristiche di due paradigmi di programmazione, ossia quello ad oggetti (garantisce di avvicinare il dominio del problema a quello della soluzione) e quello funzionale (permette di creare codice flessibile e riusabile), creando un nuovo paradigma di programmazione (utilizzato in molti altri linguaggi come *Java*, *C#*, *Kotlin*)[9].

Perché mettere insieme due paradigmi che sembrano così distanti e differenti fra loro? L'obiettivo, nel caso specifico di Scala, è quello di prendere i vantaggi di entrambi i linguaggi promuovendo, ad esempio, l'utilizzo di strutture immutabili (supporta comunque l'uso di strutture mutabili) e dando la possibilità di gestire la programmazione dichiarativa (permette di definire linguaggi di tipo *Domain Specific Languange* (**DSL**)).

Il codice Scala, compilato direttamente per la *Java Virtual Machine* (**JVM**) porta con sé tutti i benefici dell'avere un'applicazione scritta in Java (multi piattaforma, virtual machine solida, ...).

Un aspetto cruciale di Scala (quasi un manifesto del linguaggio, come lo è l'immutabilità delle strutture dati) è la concisione del codice che guadagna di leggibilità e perde di verbosità. Di seguito vengono proposti due esempi di classi, una scritta in Java e una scritta in Scala, che mettono in evidenza tale caratteristica:

Java

```java
public class Person {
  private final String name;
  private final String surname;

  public Person(final String name, final
      String surname) {
    this.name = name;
    this.surname = surname;
  }

  public String getName(){
    return this.name;
  }

  public String getSurname() {
    return this.surname;
  }
}
```

Scala

```scala
class Person(val name: String, val surname :
    String)
```

1.2.1 Caratteristiche principali

Dopo una breve introduzione sul linguaggio Scala, si descrivono le sue principali caratteristiche:

- **è un linguaggio ad oggetti puro**, nella completa accezione di *"Everything is an object"*. Non esiste distinzione tra dati primitivi e oggetti complessi: secondo la logica di Scala tutti i concetti sono trattati come oggetti, in modo trasparente all'utilizzatore. Anche un'operazione semplice, quale potrebbe essere la somma di due numeri, in Scala si traduce in una mera chiamata a metodo descritta nella classe `Int`: quando si scrive 1 + 1, il compilatore modificherà questo codice in 1.+(1);

- **first class function**, ossia è possibile passare funzioni come argomenti di altre funzioni o chiamate a metodo, rendendo Scala un linguaggio funzionale a tutti gli effetti. Una funzione può essere descritta nel seguente modo:

  ```scala
  def inc(number : Int) : Int = number + 1}
  ```

  ma si possono usare versioni più concise come ad esempio:

  ```scala
  val inc = (i : Int) => i + 1
  ```

  e, utilizzando le funzioni come oggetti di prima classe, possiamo descrivere ad esempio:

```
def mapNumber(value : Int, f : Int => Int) : Int = f(value)
```

– **linguaggio tipizzato staticamente**, infatti Scala utilizza questo sistema di tipizzazione perché permette di intercettare gli errori prima di eseguire il codice a differenza di linguaggi tipizzati dinamicamente come *Javascript e Python*. Scala riesce comunque ad evitare, tramite un complesso sistema di inferenza, di creare codice verboso quando non necessario, lasciando all'utente l'impressione di scrivere in un linguaggio non tipizzato staticamente.

  Facendo un confronto con Java, supponiamo di dover istanziare un oggetto di tipo `Person`, in Java dovremo scrivere:

```
final Person myBrother = new Person("Cristiano","Aguzzi");
```

  in Scala invece potremo direttamente scrivere

```
val myBrother = Person("Cristiano","Aguzzi")
```

  non abbiamo quindi l'obbligo di specificare che la variabile `myBrother` sia di tipo `Person`.

1.2.2 Tecniche di programmazione di interesse

Scala offre un vasto insieme di costrutti e tecniche di programmazione, tra cui le più interessanti, approfondite in *Programming in Scala*[10], sono elencate di seguito:

– **rich interface**: i `trait` possono essere visti come interfacce di Java 'arricchite', che permettono cioè di descrivere l'insieme di operazioni che ci aspettiamo da un determinato oggetto e, inoltre, permettono di inserire attribuiti ed implementazioni. Una classe concreta potrà 'mixare' più `trait` simulando il supporto all'ereditarietà multipla (attraverso l'uso della tecnica di *Trait Linearization*). Il mix dei `trait` si può usare in modo simile a quanto descritto nel *pattern Decorator* spiegato in *Design Patterns: Elements of Reusable Object-Oriented Software*[5]. Un piccolo esempio può essere:

```
abstract class Pizza {
  def ingredients : String
}
class Margherita extends Pizza{
  override def ingredients = "pomodoro mozzarella "
}
trait Salsiccia extends Pizza {
  abstract override def ingredients : String = super.ingredients + "salsiccia "
```

```
}
trait Funghi extends Pizza {
  abstract override def ingredients : String = super.ingredients + "funghi "
}

val pizzaScelta = new Margherita with Salsiccia with Funghi
```

– **implicit conversion**: le conversioni implicite sono sicuramente uno degli strumenti più potenti messi a disposizione dal linguaggio che permette di convertire un tipo di dato in un altro senza scrivere direttamente la chiamata a metodo. Può essere usato per diversi scopi: per usare il pattern *Pimp my library*, per permettere la scrittura di un **DSL** e anche per implementare il pattern *Adapter* [5] agilmente. Le conversioni implicite sono utilizzate ampiamente nelle librerie standard (permettono ad esempio di convertire un numero `Int` in uno `Double`, ...). Nonostante la sua potenza, è uno strumento da utilizzare con parsimonia perché, qualora venissero descritte più conversioni implicite, non sarebbe sempre banale definire, in fase di debug, quale sarà utilizzata;

– **pattern matching**: è un meccanismo per controllare se un determinato valore segue un pattern. Può essere visto come una naturale evoluzione del costrutto `switch case` ma molto più generico. In Scala è possibile, insieme alle `case class`, usare questo costrutto per verificare se un oggetto generico segue uno dei pattern descritti. Il pattern matching è utilizzato in supporto delle espressioni regolari:

```
val date = raw"((\d{2})-(\d{2})-(\d{4}))".r
"27-04-1996" match {
  case date(day,month,year) => "what a beautiful day"
  case _ => "uh?"
}
```

– **default parameter e named argument**: sono due tecniche molto spesso utilizzate insieme. Con gli argomenti di default è possibile specificare uno o più valori standard. Utilizzando i named argument è possibile, invece, invertire l'ordine degli argomenti specificandone il nome nella signature. La sezione di codice di seguito mostra questa funzionalità:

```
//default argument
case class Point(x : Double = 0, y : Double = 0)
/* se sono presenti dei valori di default posso non passare nessun argomento */
val zero = Point()
//posso speficare anche un solo valore
val randomXPoint = Point(x = math.rand)
//posso invertire l'ordine degli argomenti
```

```
val randomPoint = Point(y = math.rand, x = math.rand)
```

questa tecnica può essere utilizzata per implementare il pattern *Builder* [5] in modo semplice;

– **abstract type**: come i tipi generici, gli abstract type sono dei tipi astratti che vengono definiti in fase di implementazione e permettono di realizzare il *polimorfismo a famiglie* in maniera più agile. Un esempio di come si definisce un tipo generico è descritto nel codice che segue:

```
trait Buffer {
  type T
  def element : T
}
```

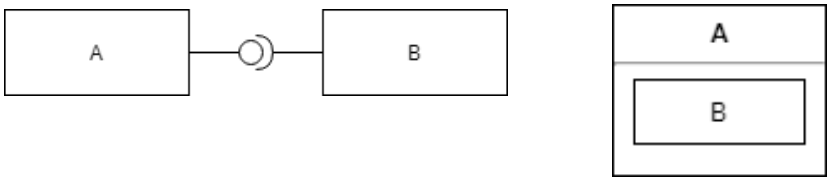
– **self-type**: nel libro *Programming in Scala* [10] viene descritto come: *"A self type of a trait is the assumed type of this, the receiver, to be used within the trait. Any concrete class that mixes in the trait must ensure that its type conforms to the trait's self type"*. Bisogna distinguere quella che è l'ereditarietà dei `trait` dai *self-type*: con l'ereditarietà stiamo rappresentando una relazione di tipo *is-a* mentre, con i self-type, diciamo che un `trait` ha bisogno di un altro `trait` e, di conseguenza, non esiste gerarchia di tipo.

In questa breve sezione altri concetti, utilizzati ampiamente e approfonditi durante l'intero lavoro di tesi, come i *bound* dei tipi generici, l'intera libreria per le strutture dati, l'accesso uniforme, le funzioni parzialmente definite, la gerarchia delle classi in Scala, companion object, by name argument ..., non sono stati menzionati, ma, per ulteriori approfondimenti al riguardo si rimanda a [10].

1.2.3 Convezioni UML usate

Nella fase di progettazione, l'utilizzo del meta linguaggio *Unified Modeling Language* (**UML**) è stato fondamentale per descrivere la struttura dell'applicazione.

Per mostrare alcuni concetti avanzati introdotti in Scala, sono stati utilizzati dei formalismi non standard visto che, per il momento, non sono supportati in **UML**. Nella figura 1.3 vengono mostrate le notazioni introdotte.



(a) (b)

Figura 1.3: in (a) viene mostrata la notazione per graficare i *self-type*: *B* dipende da *A*; in (b) viene mostrato, invece, la notazione per descrivere gli *abstract type*: *A* ha un tipo generico *B*

1.2.4 Perché Scala?

Dopo questa panoramica la domanda sorge spontanea e, secondo Odersky, Spoon e Venners[10], i motivi principali sono: la compatibilità di Scala grazie all'utilizzo della **JVM**, la sinteticità del codice finale prodotto, l'uso di un linguaggio ad alto livello e la tipizzazione statica che permette un refactoring sicuro e rende più leggibile la documentazione (i concetti sono ben argomentati nel primo capitolo di *Programming in Scala*[10]). In particolar modo, il fatto di poter creare un **DSL** direttamente in Scala, ha ridotto il gap che si forma tra il model-domain e il problem-domain, permettendo così di rendere il *model driven development* più facile da seguire. Inoltre i costrutti che permettono di utilizzare molti dei design pattern descritti in *Design Patterns: Elements of Reusable Object-Oriented Software*, sono presenti nativamente in Scala e questo permette al programmatore di utilizzarli in modo naturale e intuitivo. Portando come esempio il pattern *Singleton*[10], in Java bisognava individuare innanzitutto la situazione adatta per applicarlo e, una volta trovata, bisognava porre particolare attenzione alla sua implementazione (problemi di concorrenza, problemi di inizializzazione,...); in Scala, invece, esiste una parola chiave (`object`) che ingloba esattamente lo stesso concetto descritto dal pattern. Allo stesso modo, possono essere usati pattern (*Simple Factory, Strategy, Adapter...*) senza necessità di implementazioni ad hoc.

Ovviamente Scala non si pone come panacea (data anche la sua complessità) ma è sicuramente un linguaggio di programmazione moderno potente e che, se usato adeguatamente, offre allo sviluppatore un vasto numero di *"strumenti"* per costruire i propri programmi.

1.3 Aggregate programming in Scala: `ScaFi`

La sezione 1.1 ha introdotto quelli che sono i concetti principali per comprendere cosa sia *aggregate computing*. Non si è parlato di alcun framework

Figura 1.4: due istanze di piattaforme aggregate in *ScaFi* [14]

che supportasse tale modello computazionale e che quindi garantisse l'esecuzione di programmi aggregati. Il prof. Mirko Viroli, in collaborazione con il dott. Roberto Casadei, ha costruito il framework `ScaFi` (link al repository: `https://github.com/scafi/scafi`) nato proprio per soddisfare questa esigenza.

Il progetto è suddiviso in due parti: [1][3]

- **aggregate programming support**: fornisce una sintassi specifica e una semantica atta a definire i costrutti base del *field calculus* (grazie ad un **DSL** scritto in Scala). Un programma aggregato può così essere espresso e combinato con il codice tradizionale;

Codice 1.1: field calcus descritto nella parte core di *ScaFi*

```
trait Constructs {
 def rep[A](init: A)(fun: (A) => A): A
 def nbr[A](expr: => A): A
 def foldhood[A](init: => A)(acc: (A,A)=>A)(expr: => A): A
 def branch[A](cond: => Boolean)(th: => A)(el: => A): A
 def aggregate[A](f: => A): A
 def sense[A](name: LSNS): A
 def nbrvar[A](name: NSNS): A
}
```

- **aggregate platform support**: permette di configurare ed avviare un sistema aggregato. Possibili istanze di piattaforma sono mostrate in figura 1.4.

I costruttori di base del *field calculus* in `ScaFi` possono essere spiegati attraverso due punti di vista complementari tra di loro:

- **punto di vista locale**: corrisponde alla logica operativa con una vista *device centric*, in cui le computazioni aggregate sono considera-

te nel contesto di un singolo device. Un *valore* è il risultato della computazione di un nodo in un tempo determinato;

– **punto di vista globale**: corrisponde alla logica naturale e si riferisce sia al livello di interpretazione del programma come un campo computazionale, sia alla sua manipolazione. Un *valore* è una foto dell'intero sistema del campo computazionale in un determinato tempo.

A questo punto si può descrivere quello che è un *programma aggregato* in `ScaFi` [3]: un insieme di definizioni di funzioni, dichiarazioni di variabili ed espressioni, tutte capaci di fondere i costrutti base di *ScaFi* con librerie Scala già esistenti. Esempi di programmi aggregati sono mostrati nella porzione di codice sottostante:

Codice 1.2: Esempi di semplici programmi aggregati in *ScaFi*

```
//il sistema valuta un valore costante:
"Hello World"
//leggere un valore di un sensore:
sens[Double]("temperatura")
//conta il numbero di round per ogni device
rep(0) {x => x+1}
//conta il numero di vicini di ogni nodo
foolhood(0)((x,y) => x + y) {nbr(1)}
```

1.3.1 Package `core`, aspetti avanzati

Nel package `core` di Scafi sono stati definiti gli aspetti principali per implementare il **DSL** e per permettere di eseguire un programma aggregato usando una *virtual machine*. La struttura complessiva è mostrata in figura 1.5). Il design della struttura risulta essere tanto complesso quanto riusabile ed estendibile: i concetti generali che servono per implementare il framework sono descritti nella classe `Core` che segue il *polimorfismo a famiglie*. Nel trait `Language` vengono descritti i costrutti principali del *field calculus* (mostrati nel codice 1.1). Gli altri concetti presenti nel diagramma in figura 1.4 vanno ad arricchire ed aggiungere funzionalità ai concetti descritti in `Core`. Nella classe `Semantics`, ad esempio, si struttura il concetto di *round* descritto nella sezione 1.1.1; in `Engine` si aggiungono le varie implementazioni per i concetti di *export e context*. In particolare *export* assume una struttura ad albero dove:

– nella radice si trova il valore calcolato nel *round*;

– nelle foglie sono presenti le varie chiamate ai costrutti di base usati per calcolare il valore finale.

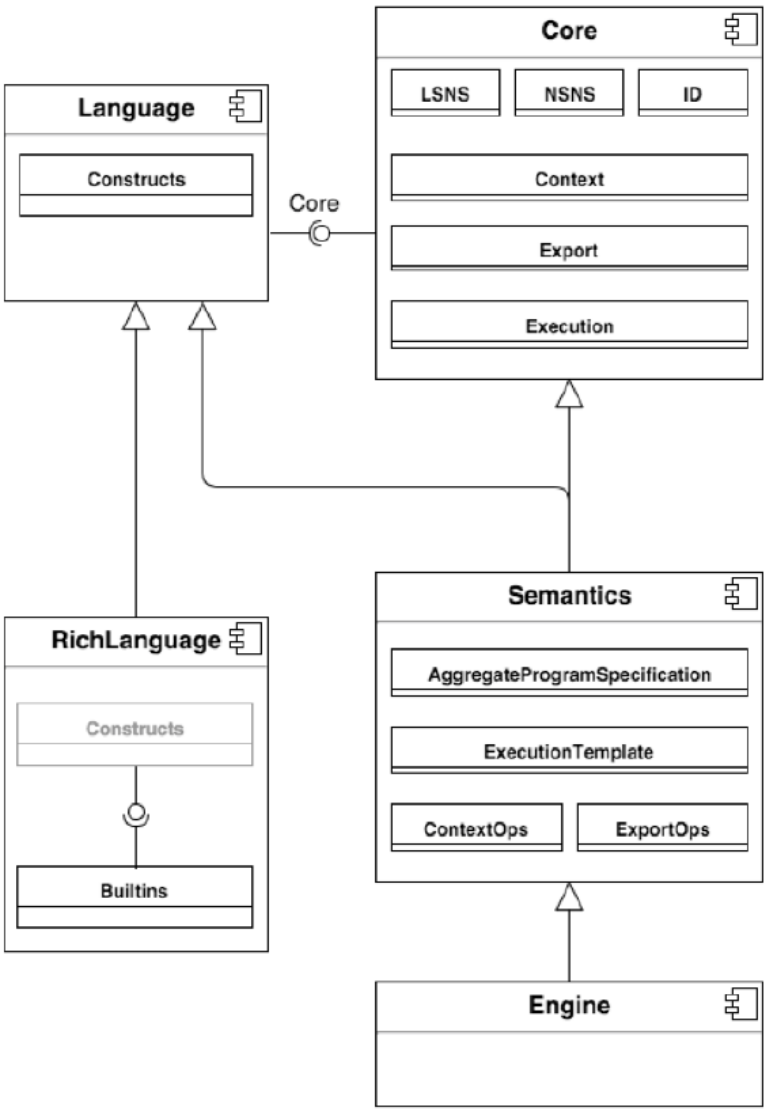


Figura 1.5: struttura del package `core` di *ScaFi* [2]

Un concetto esterno al *core* ma comunque rilevante, è quello delle *incarnations* in cui si fa uso massiccio del *Cake pattern*. Un'*incarnation* in `ScaFi` può essere vista come l'implementazione delle astrazioni di *aggregate computing*: include un linguaggio, una virtual machine e una piattaforma che astrae i concetti di spazio e tempo necessari ad un sistema aggregato.

Lo scopo di questa tesi non è stato quello di studiare dettagliatamente i concetti descritti in `core`, ma la loro comprensione ha portato all'utilizzo di tecniche avanzate quali:

– **polimorfismo a famiglie**: è una funzionalità del linguaggio che permette di esprimere relazioni multi-oggetto, garantendo sia la flessibilità, sia il riuso del codice e sia la *type safety* delle *famiglie* [4] (una famiglia è un gruppo di classi ricorsive e mutabili [6]). Questo concetto permette di descrivere un elemento generico di alto livello che verrà definito incrementalmente. Un esempio in Scala può essere:

```
trait Animal[ID] {
  val id : ID
  val legs : Int
}
case class Chicken[ID] (id : ID) extends Animal[ID]{
  override val legs = 2
}
trait Farm { //elemento radice generale
  type ANIMAL <: Animal[ID]
  type ID
  def animals : Set[ANIMAL]
  def animal(id : ID) : Option[ANIMAL]
  def add(animal : ANIMAL) : Unit
  def totalLegs : Int = animals.map(_.legs).sum
}
//raffina il concetto di Farm
class ChickenFarm extends Farm{
  private var chickens = Set.empty[ANIMAL]
  type ID = Int
  type ANIMAL = Chicken[ID]
  override def animals = chickens
  def animal(id : ID) = chickens.find(_.id == id)
  def add(animal : ANIMAL) = chickens += animal
}
val myChickenFactory = new ChickenFarm
//posso aggiungere elementi accettati nella fattoria
myChickenFactory.add(Chicken(10))
//ovviamente non compila! la classe accetta solamente istanze
//con id intero
myChickenFactory.add(Chicken("1.1.2"))
```

Nel capitolo di progettazione si introdurranno dei concetti che fanno uso del polimorfismo a famiglie e, visto che questa fase non deve dipendere dal linguaggio scelto, si è dovuta descrivere una modalità per identificare tale tecnica attraverso **UML**. In figura 1.6 viene mostrato un diagramma delle classi che descrive tale concetto.

– **Cake pattern**: è nato per risolvere il problema di dipendenza di due componenti e in Scala viene implementato attraverso l'uso dei *self types*: si definiscono dei `trait` in termini delle loro dipendenze con altri `trait`. Di seguito un semplice esempio:

```
trait Shell
trait Engine
trait Car {
```

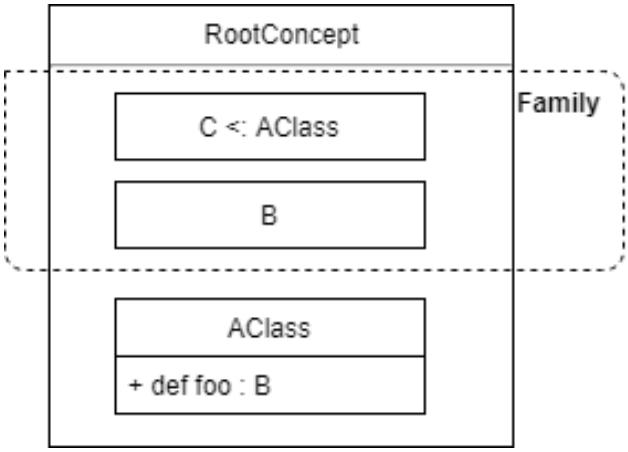


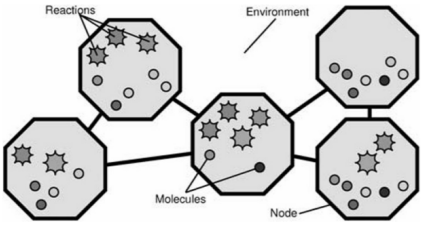
Figura 1.6: descrizione del polimorfismo a famiglie tramite UML : si descrive un concetto in termini di un insieme di tipi astratti che sono in qualche modo correlati.

```
  self: Car.Dependency
}
object Car {
  type Dependency = Shell with Engine
}

trait VolkswagenEngine
trait VolkswagenShell
trait SeatShell

class VolkswagenUp extends Car with VolkswagenEngine with VolkswagenShell
class SeatMii extends Car with VoklswagenEngine with SeatShell
//non compila, car deve essere mixata con un shell ed un engine
class FiatPanda extends Car
```

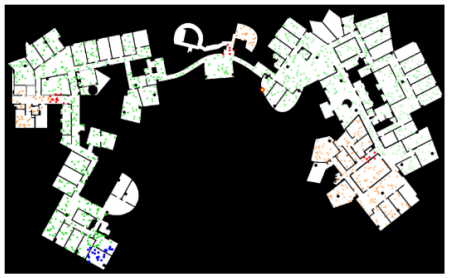
1.4 Simulazione

Una volta creato il framework di `ScaFi` è nata la necessità di eseguire dei test su un insieme di casi prova per valutare le varie possibili applicazioni aggregate descritte con il linguaggio. Per evitare di effettuare il deployment su una struttura reale si è pensato di effettuare una *simulazione* di un ambiente aggregato per poi eseguire e testare le applicazioni costruite.

Inizialmente, per raggiungere lo scopo, è stato usato *Alchemist*, un simulatore *event-driven* sviluppato dal dott. Danilo Pianini e realizzato appositamente per le simulazioni di sistemi pervasivi. Alchemist descrive un meta modello ispirato dalla chimica e, per eseguire le simulazioni, ha bisogno di un'*incarnazione* (è un *mapping* tra il meta modello di Alchemist e concetti di interesse dell'utente)[15]. Questo progetto ha permesso a Scafi, creando



(a) meta modello in Alchemist [12]



(b) simulazione indoor con ambiente complesso [15]

(c) simulazione outdoor importando dati da open street map [15]

un'*incarnazione* del meta modello di *Alchemist*, di utilizzare una piattaforma solida e performante per le applicazioni aggregate. La struttura complessiva del progetto del dot. Danilo Pianini è decisamente complessa: per approfondimenti si rimanda al sito `https://alchemistsimulator.github.io/`: e ai paper [15] [12]. Di seguito vengono mostrate delle figure sul meta modello e un insieme di simulazioni eseguite su Alchemist. Con il passare del tempo, `ScaFi` ha iniziato ad essere utilizzato per scopi didattici e, di conseguenza, si è sentita l'esigenza di avere un simulatore che permettesse di sfruttare la piattaforma in maniera intuitiva. A questo scopo è stato creato, direttamente in `ScaFi`, un simulatore con logiche ad hoc per il dominio in questione.

Il simulatore è stato dotato di un'interfaccia grafica per permettere agli utenti di interagire e valutare i propri programmi aggregati. Quest'ultima parte, insieme all'interfaccia grafica sviluppata, sono stati oggetto di studio e approfondimento essenziali per la creazione del front-end in cui si è cercato di analizzare quali fossero i problemi al fine di attuare decisioni il più possibile corrette.

1.4.1 Simulatore implementato in `ScaFi`, concetti principali

Il simulatore si suddivide in due parti principali:

– **parte logica**: è il simulatore vero e proprio, implementa un'incarnazione spaziale di `ScaFi` con una rete di nodi.

  Al simulatore, per eseguire un *round*, verrà passata un'istanza di una funzione `CONTEXT => EXPORT` la quale definisce il comportamento del programma aggregato. Inoltre, il simulatore effettua lo *"scheduling"* dei round, coordina la comunicazione dei device e ne gestisce lo stato;

– **parte grafica**: in questo livello di astrazione viene utilizzato il pattern *Model View Controller* (**MVC**) per mostrare le entità definite nella parte logica. Vi sono una serie di concetti per effettuare il bridge tra il modello *"grafico"* e quello *"logico"*, in particolare questa parte del progetto è stata trattata nella tesi di laurea di Chiara Varini, «Sviluppo di un simulatore per la piattaforma Scafi»[13].

Separando logicamente i due modelli è stato quindi possibile usare il modello grafico (sfruttato per mostrare informazioni sull'interfaccia grafica) come vista di quello logico.

La parte logica del progetto espone l'incarnazione `SpaceAwareSimulator` che permette di:

– ricevere l'insieme di nodi posizionati nello spazio del simulatore;

– ricevere l'insieme di vicini di un dato nodo;

– cambiare/ visualizzare lo stato di un sensore;

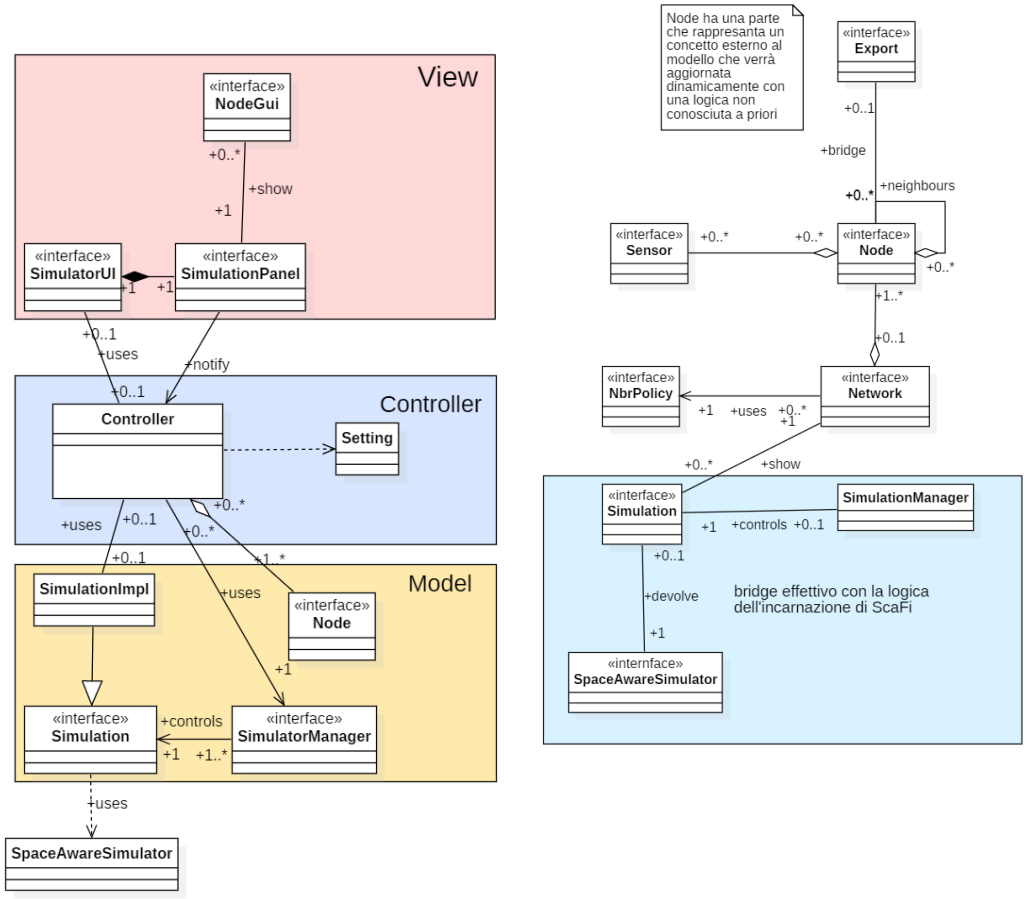
– eseguire un *round* e valutare l'export prodotto da tale computazione.

La struttura generale creatasi è riassunta nel diagramma **UML** successivo: 1.8a.

1.4.2 Modello grafico e Pattern Bridge

Il modello rappresentato in figura 1.8b descrive una realtà composta da nodi interconnessi tra di loro in una rete che si occuperà della gestione dei nodi stessi e della verifica del vicinato di ogni nodo (la classe Policy ci dirà se due nodi si possono definire come vicini o meno). Ogni nodo può avere: uno o più sensori, un id univoco, una posizione e una parte (export) che, rappresentando concetti esterni a questo contesto, non possiamo conoscere a priori.

Ogni sensore è composto da un nome e dal valore che può assumere durante l'esecuzione della simulazione. Un sensore può descrivere sia una realtà



(a) architettura riassuntiva del simulatore (b) la figura mostra il modello "grafico" del simulatore

Figura 1.8: model e architettura del modulo grafico

fisica (ad esempio un sensore di temperatura), sia un generico valore al quale verrà dato un significato solamente in un secondo momento.

La classe `SpaceAwareSimulator` internamente non ha un suo *clock* di esecuzione del programma aggregato (non fa altro che mantenere le strutture e le informazioni necessarie a rendere possibile l'esecuzione di un singolo *round*) e quindi è stata costruita la classe `Simulation` come concetto ponte tra il modello logico e quello grafico: permette di definire una certa cadenza temporale al termine della quale si scatenerà il *tick* che utilizza `SpaceAwareNetwork` per eseguire un *round* del programma aggregato e per aggiornare il modello grafico salvando il valore corrente dell'*export*. Lo scenario di esecuzione viene spiegato dal diagramma in figura 1.9

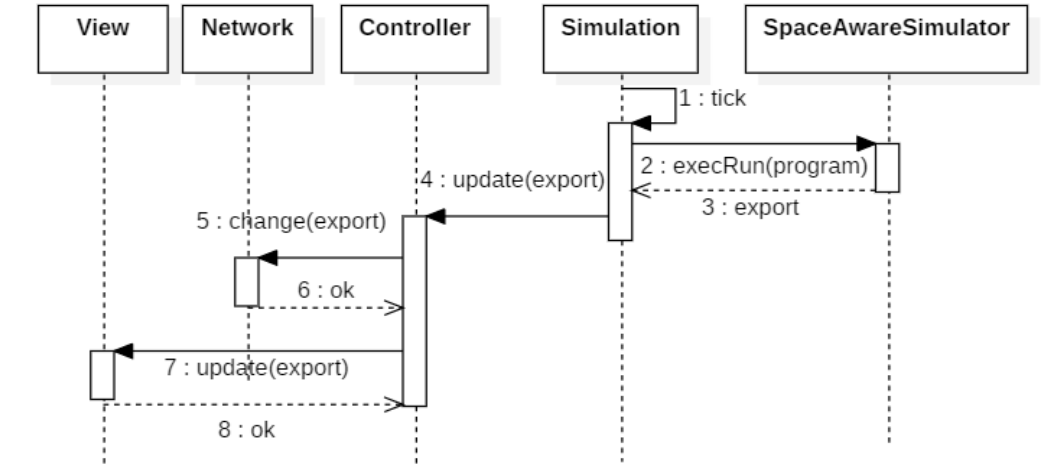


Figura 1.9: l'immagine mostra l'interazione tra i vari componenti del simulatore per eseguire un round

Capitolo 2

Requisiti e Analisi

In questo capitolo verrà descritta la fase d'analisi effettuata per la progettazione del sistema. I vari requisiti sono stati fortemente influenzati dallo studio pregresso del vecchio modulo grafico e delle varie analisi susseguitesi nel percorso di tirocinio.

La trattazione verterà sulle motivazioni per cui è stato sviluppato un front-end, su quali specifiche dovrà avere e su che cosa dovrà fare.

Le varie sezioni andranno a delineare in modo dettagliato i requisiti del sistema (non solo del front-end, ma anche della corrente parte logica) per poi mostrare il caso d'uso del simulatore e la definizione di quello che sarà il modello del dominio.

2.1 Requisiti

In questa sezione si andranno a delineare i vari requisiti suddivisi in base alla loro categoria rendendo chiare le funzionalità del front-end.

2.1.1 Di business

Dopo lo studio del simulatore di `ScaFi` sono state rilevate una serie di problematiche sia di design che di implementazione che rendevano il software difficile da manutenere e non estendibile. Il simulatore soffriva inoltre di gravi problemi di performance dovuti principalmente alla gestione di aggiornamento dei nodi in seguito alla computazione di un round. In particolare, non erano ben distinte le responsabilità tra i due modelli e, soprattutto in quello grafico, vi erano dei controlli spaziali che dovevano essere effettuati solamente nella parte logica. Dopo un'attenta valutazione si è scelto di soppiantare quello che era il vecchio modulo grafico, sostituendolo con un

front-end che rendesse possibile utilizzare il simulatore per `ScaFi` non solo per simulazioni prettamente dimostrative.

2.1.2 Funzionali

Di seguito verranno mostrati l'insieme dei requisiti funzionali richiesti in fase di creazione del progetto:

- **possibilità di configurare l'applicazione all'avvio**: il front-end deve permettere di creare la complessa rete della simulazione seguendo una data logica, impostare i possibili parametri importanti per la simulazione e infine eseguirla;
- **configurazione non dipendente dalla tecnica d'avvio**: il front-end potrà supportare diverse modalità d'avvio, ad esempio via console o da linea di comando. I parametri che verranno usati per avviare la simulazione devono essere semanticamente gli stessi in tutte le possibili tecniche supportate;
- **configurare la produzione (rendering) complessa di output**: configurare la visione del sistema deve essere possibile senza modificare l'intera interfaccia grafica, il sistema dovrà permettere configurabilità sia per la visualizzazione (come viene mostrato un nodo? Un sensore?) che per il rendering (visualizza il vicinato di ogni nodo oppure no?);
- **visualizzazione di informazioni specifiche sulla simulazione e sull'ambiente simulato**: il front-end dovrà preoccuparsi di trovare un modo per raccogliere queste informazioni e per individuare come e dove mostrarle (ad esempio le informazioni possono essere salvate come log su un file oppure visualizzate via interfaccia grafica). Un'informazione interessante da visualizzare è, ad esempio, l'*export* prodotto da un singolo nodo ad un certo punto della computazione;
- **gestione tramite UI del ciclo di vita delle simulazioni (pausa, stop, restart**: la simulazione dovrà essere controllata in qualche modo via interfaccia grafica, per permettere all'utente in maniera intuitiva di modificarne i parametri (come ad esempio la velocità di esecuzione);
- **configurare interazioni complesse durante l'esecuzione**: il front-end dovrà permettere all'utente di interagire con il sistema simulazione per, ad esempio, spostare i nodi in un altra posizione, cambiare lo stato di un sensore...;

- **gestione di modalità batch**, il simulatore dovrà garantire di eseguire delle simulazioni in background senza interfaccia grafica per osservare un risultato finale;
- **gestire il movimento di nodi, con modelli intercambiabili** anche questo sarà un requisito prettamente della parte logica, ma il front-end dovrà visualizzare questo cambiamento osservando lo stato del modello logico. Per modello logico non si intende la mera gestione del movimento, ma anche scelte più complesse del tipo: cosa deve succedere quando due nodi collidono? Cosa deve succedere quando un nodo colpisce un ostacolo?;
- **gestire simulazioni su ambienti "realistici" (indoor, mappe)** l'intero simulatore dovrà attuare logiche in base all'ambiente simulato, il front-end dovrà permettere la visualizzazione degli ambienti scelti;
- **interfacce grafiche intercambiabili**: l'architettura finale dovrà permettere di cambiare interfacce grafiche (ad esempio passaggio da un'interfaccia 2D ad una 3D) senza modificare l'intero sistema.

2.1.3 Non funzionali

Per permettere di creare un sistema performante e utilizzabile con simulazioni complesse, sono state delineate le seguenti specifiche del sistema:

- **scalabilità del sistema**: l'intero modulo del simulatore dovrà essere scalabile sul numero dei nodi: il tempo d'esecuzione di un round e la creazione di uno stato del contesto non devono dipendere da quanto è densa la rete;
- **performance di simulazioni indipendenti dall'utilizzo dell'interfaccia grafica**: la visualizzazione del mondo simulato, con le varie informazioni associate, non deve portare un degrado significativo di performance d'esecuzione per round, in particolare si è deciso che in termini di tempo dell'esecuzione del singolo round, l'interfaccia grafica può introdurre un ritardo massimo del 30% rispetto alla simulazione standard;
- **tempo di inizializzazione del mondo lineare**: l'inizializzazione del mondo deve avere una complessità lineare (o sub lineare) per garantire tempi ragionevoli d'avvio e d'utilizzo del sistema.

- **movimento dei nodi all'interno dello spazio**: il tempo di movimento dei nodi dovrà essere tale da garantire simulazioni real time. Per svolgere tale compito, il movimento dovrà avere una complessità sublineare;

- **accesso al vicinato di un nodo**: per eseguire ogni round si dovrà calcolare il vicinato associato ad ogni nodo, perciò quest'operazione è di vitale importanza. L'obiettivo ideale è quello di permettere tale operazione in tempo costante ammortizzato;

- **garanzia di un certo *Frame Per Second***: le simulazioni dovranno essere renderizzate ad un certo numero di **FPS** predefinito e, per tutta la durata della simulazione, ci si aspetterà che tale valore rimanga costante.

2.1.4 Tecnologici

Nella fase di analisi dei requisiti sono state scelte anche due delle tecnologie usate nella fase di implementazione:

- Scala: per l'implementazione dell'intero progetto è stato scelto il linguaggio Scala per i motivi descritti nel capitolo 1.2 e anche per mantenere una certa coerenza con quello che era il resto del progetto già scritto in tale linguaggio.

- `ScalaFX`: come libreria grafica si è scelto `ScalaFX`: è un **DSL** che utilizza internamente la libreria `JavaFX` e permette la creazione di scene in maniera più veloce. Si è scelto di migrare verso questa libreria perché garantisce la piena interoperabilità con `JavaFX` e, allo stesso tempo, rende la scrittura dell'interfaccia grafica più semplice ed intuitiva. Per approfondimento su rimanda al sito `http://www.scalafx.org/` dove sono presenti sia l'intera *Application Programming Interface* (**API**), sia esempi applicativi e sia la documentazione della libreria. La riprova di quanto appena detto è evidenziata dal codice proposto di seguito, in cui vengono mostrati due esempi della stessa interfaccia grafica scritta sia in `ScalaFX` che in `JavaFX`.

  L'iter che ha portato alla scelta di questa libreria ha attraversato anche il problema delle performance e, a tale scopo, è stato di particolare importanza verificare quanti nodi il sistema potesse renderizzare con un frame rate pari a o superiore a 10 **FPS**.

Dopo vari test e in particolar dopo la lettura di `https://wiki.openjdk.java.net/display/OpenJFX/Performance+Tips+and+Tricks` si è concluso che tale libreria riesca a gestire decine di migliaia di nodi con il frame rate desiderato.

Java

```java
public class HelloWorld extends Application {
  private static final String HELLO = "Hello
      World";
  public static void main(String[] args) {
    launch(args);
  }

@Override
  public void start(Stage primaryStage) {
    primaryStage.setTitle(HELLO);
    final Label hello = new Label(HELLO);
    StackPane root = new StackPane();
    root.getChildren().add(hello);
    primaryStage.setScene(new Scene(root));
    primaryStage.show();
  }
}
```

Scala

```scala
object HelloWorld extends JFXApp {
  val Hello = "Hello World"
  stage = new JFXApp.PrimaryStage {
    title.value = Hello
    scene = new Scene {
      content = new StackPane {
        children = new Label(Hello)
      }
    }
  }
}
```

2.1.5 Di implementazione

Durante la stesura dei requisiti, ne sono stati evidenziati alcuni di implementazione di seguito elencati:

- **possibilità di aggiungere funzionalità in modo dinamico senza alterare il sistema**: il front-end, in futuro, potrebbe crescere ed aggiungere nuove funzionalità. Queste aggiunte, però, non dovranno alterare la struttura esistente: il sistema dovrà permettere l'aggiunta di tali funzionalità in modo modulare;
- **Visualizzazione disaccoppiata dalla logica del simulatore**: il sistema dovrà disaccoppiare la computazione del simulatore dalla visualizzazione (il vecchio modulo aggiornava l'interfaccia grafica ad ogni round computato);
- **possibilità di aggiungere pannelli informativi**: il front-end dovrà garantire la possibilità di aggiungere in modo semplice pannelli che mostrino informazioni circa la simulazione.

2.2 Casi d'uso

Seguendo ciò che è stato descritto nella fase di analisi dei requisiti, il sistema grafico dovrà permettere una fase di configurazione ed avvio in cui l'utente potrà scegliere in modo programmatica, via console o via interfaccia grafica, i parametri utili alla simulazione per poi eseguirla e un fase di interazione con il sistema creato dove l'utente, una volta avviata l'applicazione, potrà interagire con il sistema, sia da interfaccia grafica che da console.

Nelle sottosezioni seguenti vengono ampliati i concetti appena descritti.

2.2.1 Configurazione ed avvio

L'utente potrà modificare una serie di parametri che gli permetteranno di creare un'ambiente di simulazione ad hoc in base alle esigenze stessa della simulazione, i parametri importati sono:

- **programma aggregato da eseguire**: il sistema simulato potrà eseguire un programma aggregato passato come parametro di configurazione;
- **visualizzazione delle informazioni associate ad ogni nodo**: ogni simulazione avrà un'esigenza specifica di visualizzazione riguardante le informazioni associate ad ogni singolo nodo. Alcune volte siamo interessanti a renderizzare un sensore come una figura geometrica o altra volte invece vorremo mostrare solamente il valore contenuto da esso;
- **elaborazione del risultato di ogni round**: l'*export* prodotto ad ogni round può essere trattato in modalità diverse, alcune volte può descrivere un'azione (come delta movimento, accensione di un led,..) altre volte produce un dato da mostrare semplicemente in output senza la necessità di essere elaborato;
- **impostazione del mondo** ogni simulazione avrà delle esigenze specifiche riguardanti la struttura del mondo, cioè vorremo impostare i device associati ad un nodo, i boundary del mondo e i vari ostacoli (se presenti nella simulazione). Ad ognuno di queste informazioni vorremo associare una determinata strategia di output;
- **tempo d'aggiornamento dell'interfaccia grafica**: l'utente potrà scegliere ogni quanto l'interfaccia grafica verrà aggiornata a fronte delle modifiche attuate;

- **configurazione di log**: si potrà andare a modificare, per ogni simulazione, quella che è la configurazione di log. In particolare si potrà descrive quali sono le informazioni interessati da visualizzare e in alcuni contesti anche come visualizzarle;

- **impostazioni di rendering**: alcuni parametri potranno garantire la modifica di alcune impostazione di rendering. Si potrà, ad esempio, scegliere se visualizzare la rete di vicinato oppure no, se posizionare un'immagine di backgroud, ...

Visto il gran numero di parametri si è scelto di dare all'utente esperto la possibilità di modificarli in modo programmatico in modo dettagliato, mentre per l'interfaccia grafica e l'avvia da console si è pensato di mostrare un'interfaccia semplificata sull'insieme di possibili parametri permettendo di avviare le simulazioni in modo più user-friendly visto che si presume che non tutti gli utenti vorranno modificare dei parametri troppo specifici per i loro scopi.

2.2.2 Interazione con il front-end

Esisteranno diversi tipologie di interazione con il sistema:

- **azioni sullo stato del mondo** l'utente attraverso le diverse modalità di interazione potrà provocare direttamente dell'azioni sul modello del simulatore (come ad esempio cambiare lo stato di un sensore, muovere un nodo in un'altra posizione,...)

- **azioni sullo stato della simulazione**: l'utente potrà modificare il flusso della simulazione modificando la velocità d'esecuzione del programma aggregato, metterla in pausa, riavviarla e resettarla;

- **azioni per visualizzare informazioni specifiche del sistema** si potranno visualizzare alcune informazione sul sistema (round al secondo, export prodotto da un nodo specifico,...) attraverso dell'interazione predefinite.

Il diagramma in figura 2.1 mostra in modo grafico le specifiche appena elencate.

2.3 Analisi del modello

La strutturazione del modello del front-end realizzato in questa tesi, si articola a partire dall'idea, su cui si basa il precedente software, di realizzare una

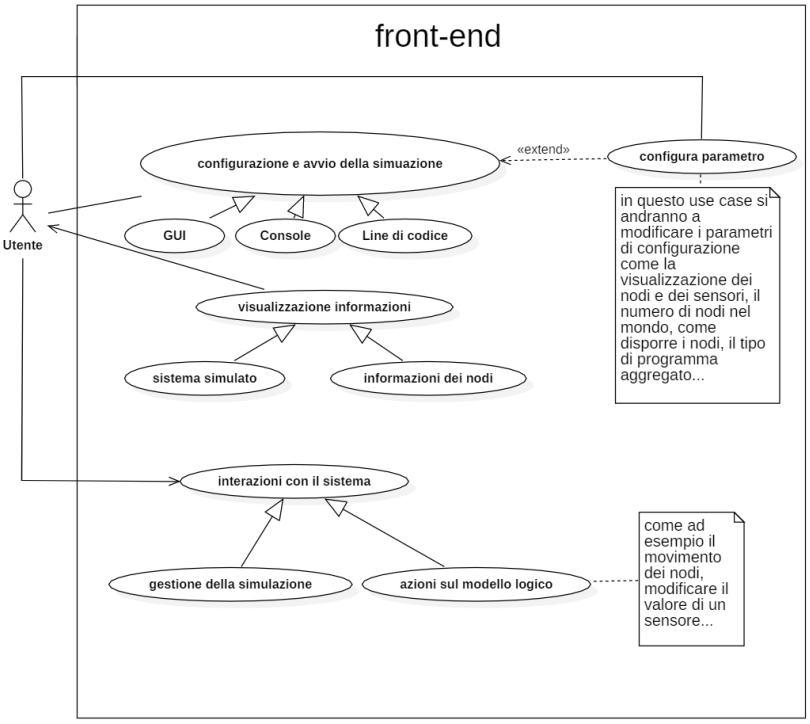


Figura 2.1: descrizione del sistema attraverso il diagramma use-case UML

vista dei concetti graficabili del modello logico. Rispetto al vecchio progetto vi sono alcune modifiche: sono stati tolti quelli che erano concetti puramente logici e ne sono stati aggiunti altri che realizzassero le specifiche descritte nell'analisi dei requisiti (come ad esempio il supporto a simulazioni indoor). La parte principale del modello è stata astratta come mostra l'immagine in figura 2.2.

Il concetto `World` è l'entry point del modello: contiene una rappresentazione dei nodi del modello logico e mantiene una rappresentazione dei confini del mondo attraverso `Bound`. Ogni nodo è identificato univocamente da un *id*, è localizzato nel mondo in una posizione ed è caratterizzato da una `Shape`. Vi è un concetto di vicinanza in cui si stabilisce che un nodo può avere zero o più vicini in base alle informazioni spaziali. Un `Device` è un'entità collegata ad un nodo, identificata da un nome (univoco all'interno del nodo) e rappresenta tutto ciò che l'interfaccia grafica dovrà visualizzare. Le varie informazioni sulla posizione, sul vicinato e sui sensori dovranno essere consone a ciò che viene descritto nel modello logico. Il concetto che fa da ponte tra i due modelli è `ScafiBridge`, il quale al suo interno gestirà la simulazione descritta da un programma aggregato e manterrà le due rappresentazioni sincronizzate. Le modifiche effettuate a run time sul modello logico verranno processate in

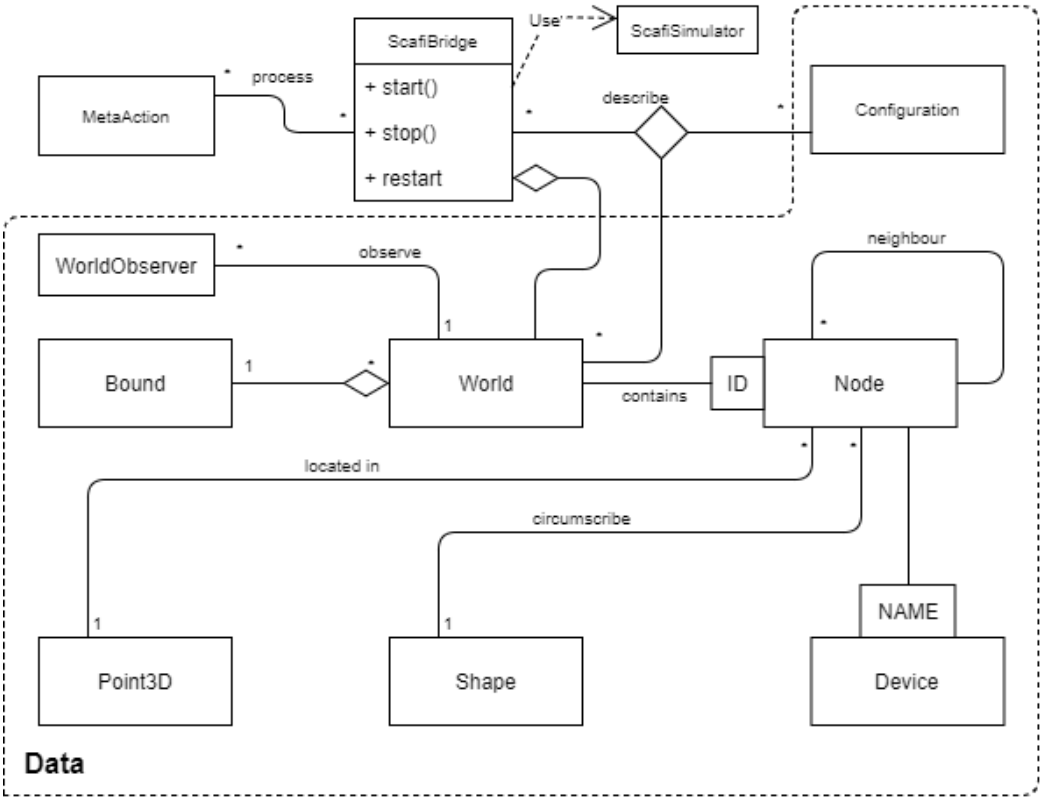


Figura 2.2: Modello del modulo front-end

base a delle istanze di `Action` che descrivono le azioni che il simulatore dovrà eseguire. `Configuration` ha il compito di configurare entrambi i modelli e descrivere i parametri principali per eseguire il front-end. Infine, il concetto di `WorldObserver` servirà per ricevere le modifiche effettuate su `World` per poi elaborarle o visualizzarle. E' importante notare la suddivisione del modello: vi è una parte in cui vengono descritti i dati, cioè lo stato dell'intero mondo; il resto descrive la logica del sistema, la quale modificherà il mondo e la sua rappresentazione.

Capitolo 3

Progettazione architetturale

In questa fase sono state prese le principali scelte architetturali del sistema per far in modo di soddisfare i requisiti definiti in fase di analisi. In particolare, in questa fase si cercherà di rispondere alla domanda: *"Come deve essere strutturato il sistema per realizzare quanto richiesto in analisi?"*

3.1 Architettura

La progettazione orientata agli oggetti portata avanti in questo lavoro di tesi, si sposa perfettamente con la scelta di strutturare il software mediante l'utilizzo di uno dei più noti pattern architetturali: Model-View-Controller (MVC). I pattern architetturali esprimono schemi di base per impostare l'organizzazione strutturale di un sistema software e MVC è uno dei più utilizzati.

Questo pattern, seguendo quanto scritto in Design Patterns [5], è caratterizzato da tre tipi di oggetti: il Model è l'oggetto dell'applicazione, la View è la sua rappresentazione grafica e il Controller definisce il modo in cui l'interfaccia grafica dovrà reagire a fronte di determinati input dell'utente. La potenza di questo pattern risiede nel riuscire a disaccoppiare la View dal Model ed in questo caso questa specifica è di vitale importanza in quanto le modifiche sul Model dovranno essere mostrate in modo asincrono alla View.

Il Controller è stato analizzato con particolare attenzione in quanto rischia molto spesso di essere trattato come una `"God" class` ossia una classe che controlla tanti oggetti nel sistema e che quindi dovrebbe essere aggiornata ad ogni loro modifica. Per non rischiare di cadere in questo problema si è suddivisa la logica del Controller in più parti ognuna con un compito ben preciso ed isolato dagli altri. La struttura generale è mostrata nella figura 3.1

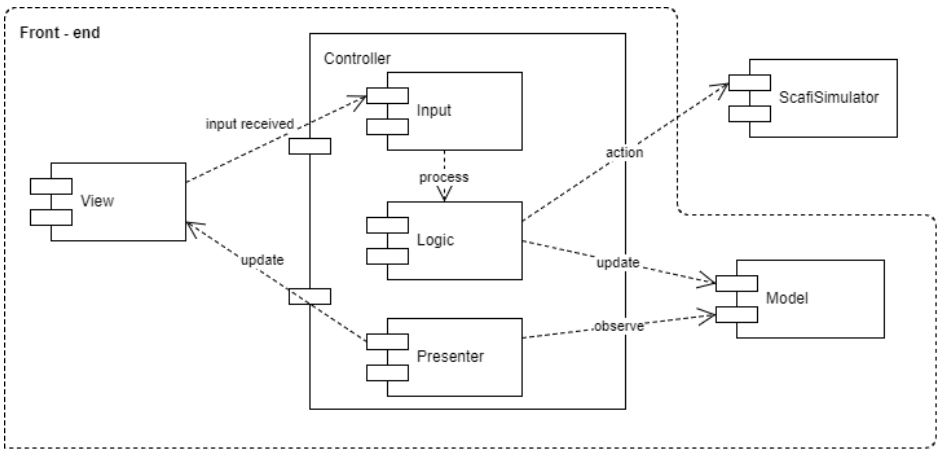


Figura 3.1: architettura software del front-end grafico.

I macro concetti rappresentati in figura 3.1 sono:

- `Input`: ha il compito di elaborare l'input dell'utente prodotto dal componente `View` e di renderlo comprensibile al componente `Logic`. Questa operazione viene esplicitata dalla dipendenza *process*;
- `Logic`: ha il compito di effettuare le azioni sul modello logico (*action*)e di aggiornare il modello grafico (la fase di *update* è critica: in questo punto si dovrà prestare attenzione a mantenere la correttezza semantica dei due modelli). Così facendo il componente logico può scegliere *quando* aggiornare il modello grafico garantendo disaccoppiamento con il reale aggiornamento avvenuto. In più dovrà permettere l'esecuzione dei *round* per eseguire la simulazione;
- `Presenter`: ha il compito di aggiornare la `View` (*update*)in base alla modifiche effettuate sul modello grafico (visualizzate attraverso *observer*) e potrà avere delle logiche di render (ad esempio scegliere se mostrare il vicinato o meno,...).

`ScafiSimulator` rappresenta un concetto logicamente esterno alla struttura del front-end, esso mantiene la rappresentazione logica del modello, ha il compito di eseguire il programma aggregato e di attuare le azioni prodotte dal front-end grafico.

La struttura rispetta inoltre altre specifiche degne di nota come:

- Non è presente una dipendenza tra `ScafiSimulation e Model`: questo permette di disaccoppiare i due mondi, non rendendo la visualizzazione strettamente collegata al modello logico. Così facendo è possibi-

le garantire gli stessi risultati di visualizzazione del sistema simulato indipendentemente dalla sviluppo del sistema logico;

- **assenza di dipendenze circolari tra i macro componenti del sistema**;
- **compiti ben definiti associati ad ogni componente**.

3.2 Model

Il concetto `World`, come descritto nella fase di analisi, dovrà essere l'*entry point* del modello e quindi, nella fase di progettazione si è posta particolare attenzione sul ruolo che gioca. Dato che `World` dovrà permettere di cambiare lo stato dei nodi che contiene in modo *safe*, si è pensato di strutturarlo in:

- **una sezione esterna immutabile**: `World` esporrà una serie di metodi che restituiranno una visione di nodi e device immutabile;
- **una sezione interna (non visibile all'utilizzatore) mutabile** : rimane la necessità di modificare lo stato interno del sistema, perciò si è creata una visione mutabile dei nodi e device che permetta di aggiornarne lo stato.

In questo modo l'utilizzatore potrà modificare lo stato del mondo attraverso la sua interfaccia così da evitare problemi di concorrenza e possibili errori accidentali dovuti all'uso errato dei concetti di nodo o di device.

In figura 3.2 viene mostrata la struttura principale del modello.

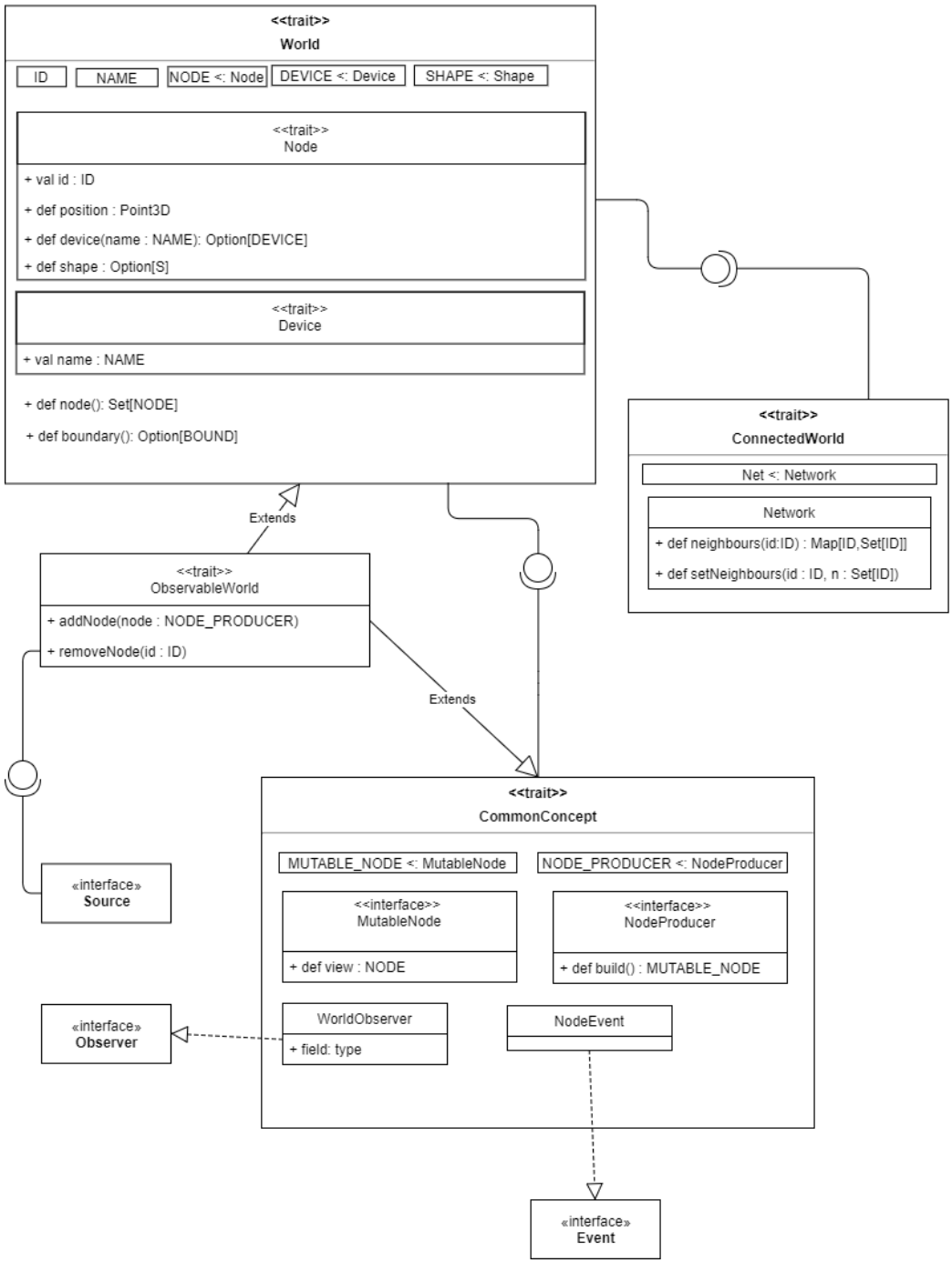


Figura 3.2: strutturazione dei concetti principali del modello.

Rispetto alla descrizione del modello effettuata in analisi, si è delineato che `World` sarà una classe generica nei tipi `ID`, `NAME`, `DEVICE`, `NODE`, `SHAPE` che sono stati introdotti per implementare il *polimorfismo a famiglie*. Il concetto generale `World`, che non permette altre azioni se non quella di visualizzare l'insieme di nodi posizionati al suo interno, viene poi ulteriormente definito attraverso alcuni concetti:

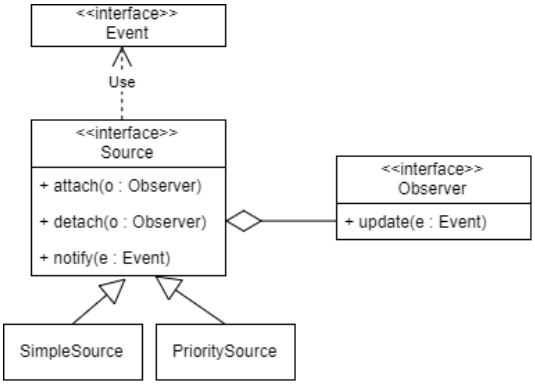


Figura 3.3: struttura pattern Observer. Vi sono due implementazioni di *`Source`* una con priorità (gli observer vengono notificati in base all'ordine di inserimento) e una senza.

- `ConnectedWorld`: descrive un mondo dove i nodi sono collegati da una rete modificabile;
- `CommonDefinition`: vengono descritte una serie di informazioni utili a effettuare l'implementazione del concetto astratto di `World` introducendo il tipo `MutableNode` (interpretazione interna di `Node` che può cambiare stato), e il tipo `NodeProducer` (permette la configurazione e la creazione di uno o più nodi).

`ObservableWorld` è un mondo osservabile che permette ad un `Observer` (seguendo il pattern descritto in [5], strutturato come in figura 3.3) di essere notificato degli eventi di modifica dello stato del mondo: ogni qualvolta lo stato del mondo si modifica, verrà generato un evento di tipo `NodeEvent` poi processato dai vari `WorldObserver` in base ai tipi di eventi che ascoltano. In più, a differenza di `World`, si possono inserire e rimuovere nodi.

L'idea è quella per cui l'utilizzatore può decidere come costruire il proprio `World` scegliendo quali trait usare e, per realizzarla, sono stati aggiunti altri concetti di base che permettessero il movimento dei nodi e la modifica dei device di un nodo. Infine, viene descritto il concetto di `Sensor` concludendo così la struttura base del modello (figura 3.4.

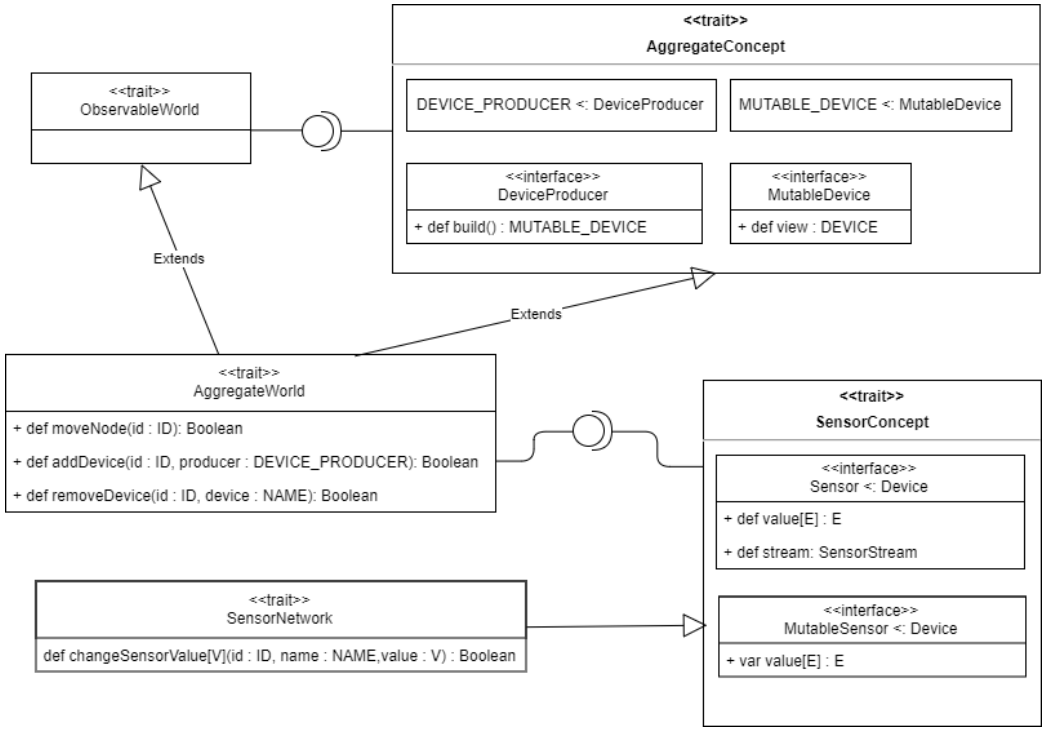
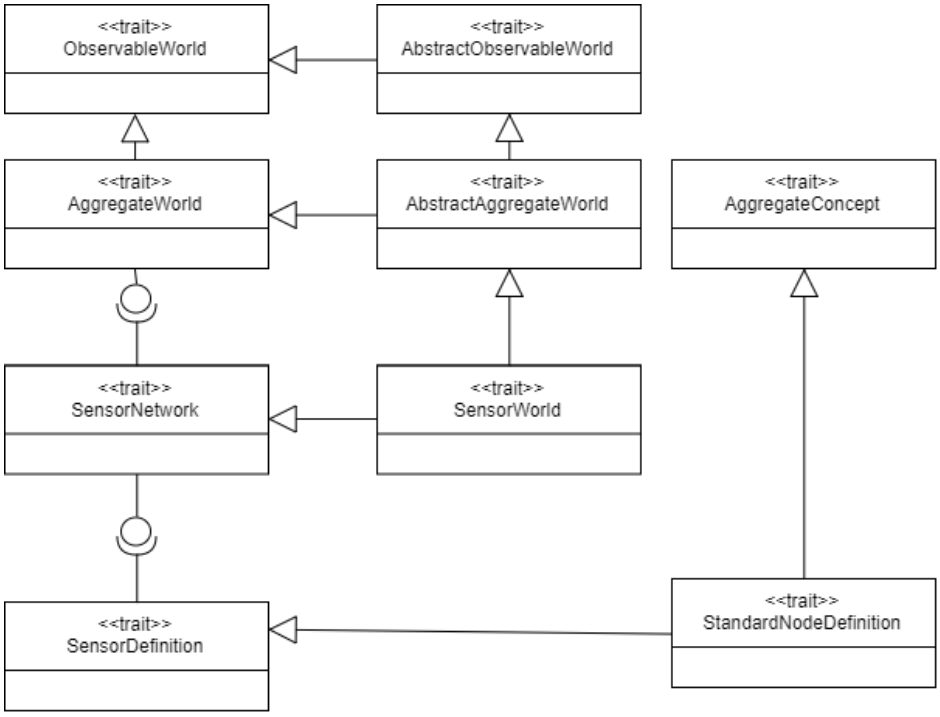


Figura 3.4: sono stati introdotti i trait *`MutableDevicev e DeviceProducer`* per permettere di descrivere device che possono cambiare di stato durante l'esecuzione e infine, in *`AggregateWorld`*, vengono descritte le operazioni di base che possono essere effettuate su un mondo mutabile: movimento, aggiunta e rimozione di un device. Per modellare il concetto di un sensore vengono aggiunti *`SensorConcept`* (delinea l'interfaccia di un sensore) e *`SensorNetwork`* (permette la modifica dello stato di un sensore). Viene introdotto il concetto di *`SensorStream`* che serve per differenziare quelli che sono i sensori utilizzati solo per scopi grafici (di output) e quelli che servono per la simulazione (di input).

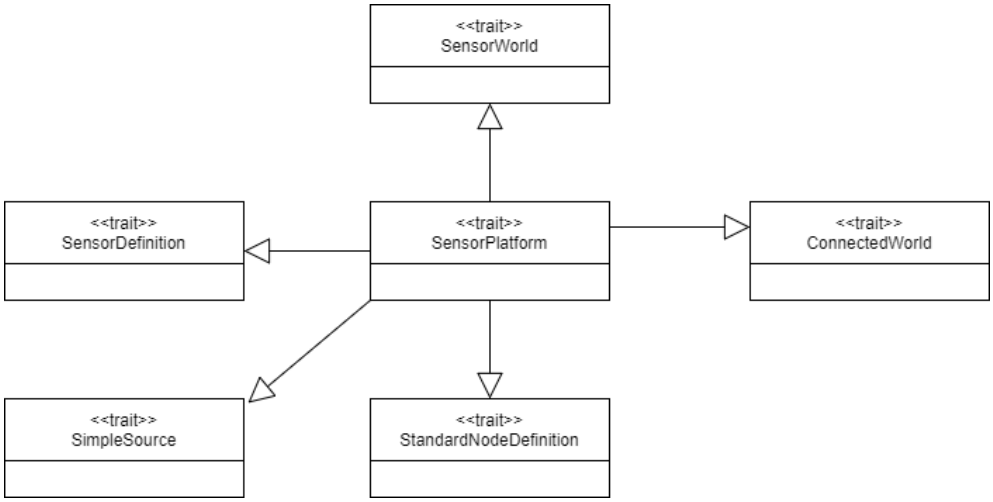
Per agevolare l'implementazione della struttura principale vengono definiti gli scheletri delle entità generali così da creare un'intera piattaforma in modo più semplice (nulla vieta all'utilizzatore della libreria di creare strutture dati ad hoc, ma è stata fatta questa scelta per semplificare l'implementazione di quella che sarà la vista sul mondo di `ScaFi`). L'insieme di tali concetti è mostrato in 3.5.

3.3 Scafi Simulator

Nella parte strutturata nella tesi [2] sono state effettuate delle piccole aggiunte per soddisfare sia i requisiti di performance che quelli funzionali. Per



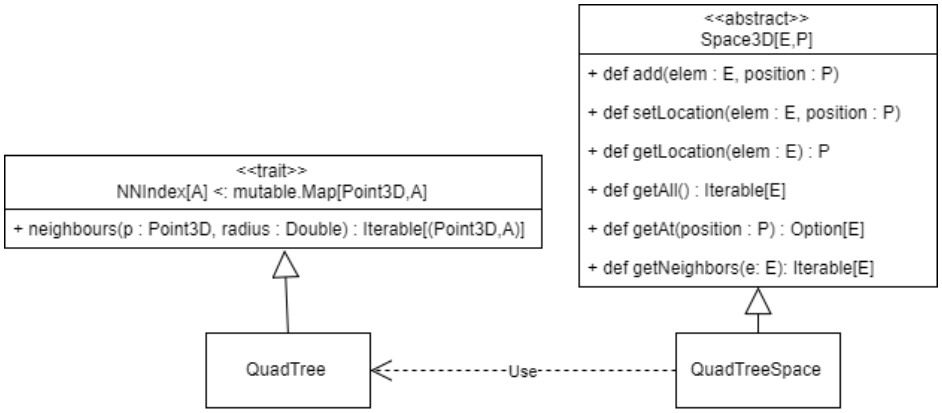
(a) descrive i concetti astratti per definire l'implementazione di un *World*.



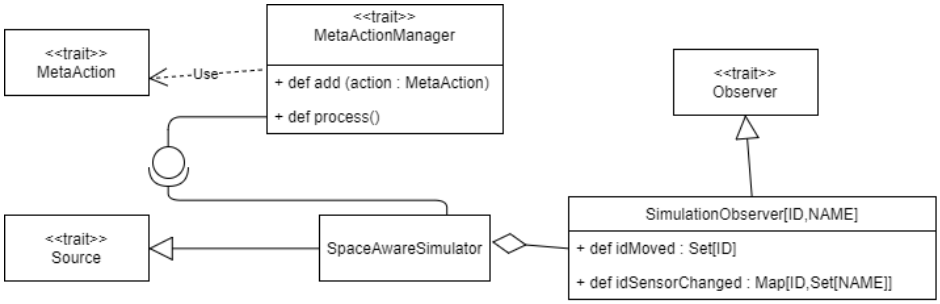
(b) piattaforma standard che descrive quali sono le dipendenze di*SensorPlatform*

Figura 3.5: descrizione della struttura creatasi nel modello del front-end.

superare gli attuali problemi di performance per il calcolo del vicinato è stato creato un indice spaziale ad hoc che si basa sulla struttura dati `QuadTree` scelto in seguito alla lettura del capitolo *Spatial Partitioning* in [8] (figura



(a) descrizione dello spazio ottimizzato.



(b) strutturazione del concetto *`MetaAction`*.

3.6a).

Inoltre è stata attuata una modifica funzionale al simulatore che permette di effettuare azioni sul suo livello logico (e non in base a delle decisione prese dal programma aggregato) introducendo il concetto di `MetaAction`. Per elaborare queste azioni, `MetaActionManager` metterà in coda tutte quelle ricevute che, attraverso il metodo `process()`, verranno valutate. Infine viene aggiunto un Observer al corrente simulatore per permettere di visualizzare le modifiche effettuate a questo livello anche dall'esterno (figura 3.6b).

3.4 Controller

Come introdotto nel capitolo 3.1 il controller è stato suddiviso in tre macro concetti. Si è costruita una struttura simile all'architettura *Game Loop* [8] per garantire disaccoppiamento tra la gestione dell'input, la logica della simulazione e l'effettiva visualizzazione delle modifiche effettuate sul mondo. È da ricordare il ruolo che riveste `World` nel front-end: deve essere una *vista*

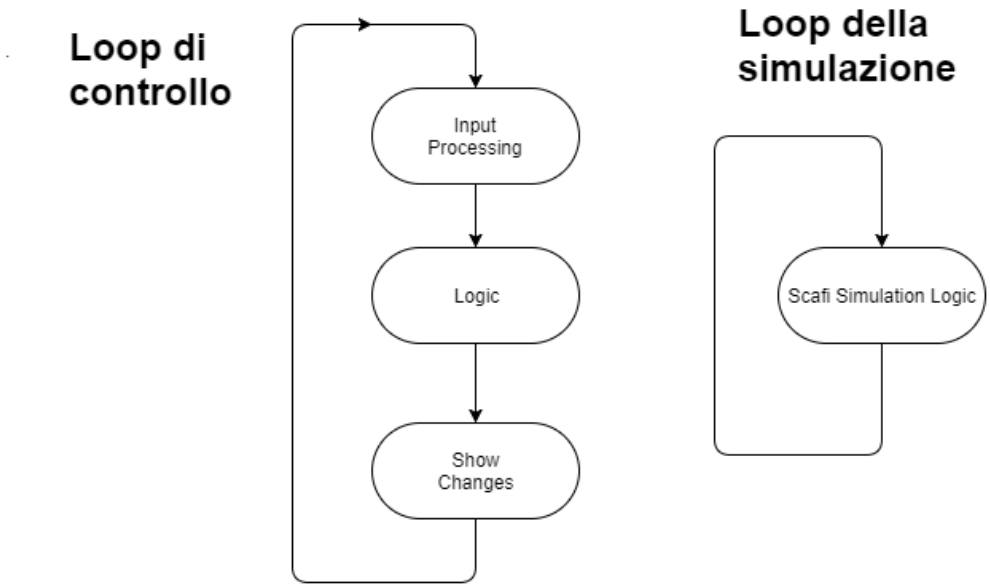
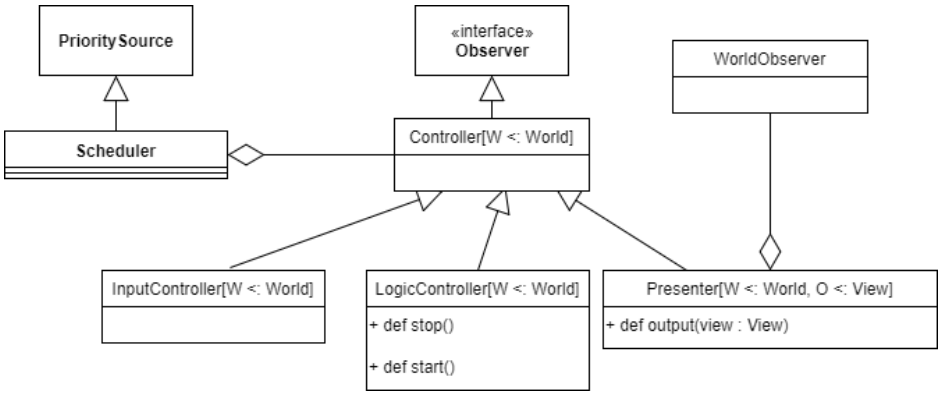


Figura 3.7: durante l'esecuzione dell'applicazione esisteranno due flussi di controllo distinti, uno che si preoccupa di intercettare l'input e visualizzare le modifiche sul modello e l'altro che eseguirà la simulazione aggregata con due tempistiche differenti

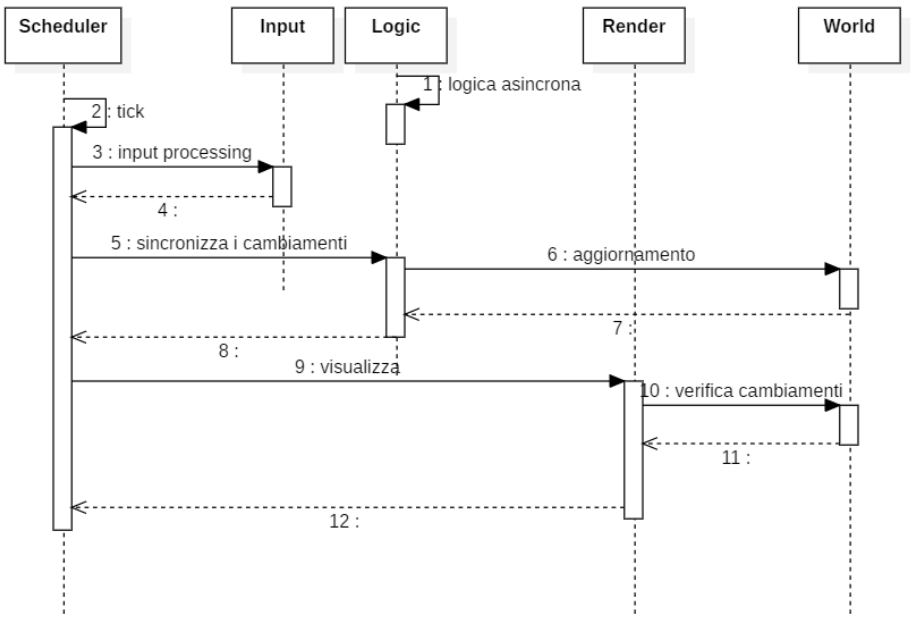
del modello logico, in particolare non verrà mai modificato direttamente da azioni di input prodotte dall'utente, ma tali azioni saranno in qualche modo interpretate e eseguite sul modello logico. A questo punto anche il modello del front-end verrà aggiornato. Questa scelta è stata determinante per evitare dipendenze complicate e aggiornamenti tra due moduli e, così facendo, nella parte logica del controller si dovrà solamente aggiornare `World` in base alle modifiche prodotte. La figura 3.7 rappresenta l'idea che ha guidato la progettazione di questo modulo.

3.4.1 Strutturazione del loop di controllo

Per fare in modo che ci fosse una certa temporizzazione di aggiornamento è stato aggiunto il concetto di `Scheduler` che ha il compito di notificare in modo sequenziale i controller del sistema (figura 3.8).



(a)



(b)

Figura 3.8: in (a) viene mostrata la struttura del controller; in (b) una possibile interazione tra le varie entità del front-end

3.4.2 Gestione dell'input e dell'output

In questo progetto si è scelto di gestire l'input utilizzando il pattern Command [5], entità che incapsula quella che è la logica di un comando e che quindi non dipende dall'input ricevuto: un comando catturerà una delle possibili interazioni con l'utente che poi verrà processata dall'InputController (in particolare i comandi che modificano lo stato del mondo logico creeranno in realtà una `MetaAction` gestita a tempo debito da `SpaceAwareSimulator`). In figura 3.9 viene mostrata la struttura implementata nel front-end. `Presenter`, posto come ultimo elemento nella logica di controllo, "presenterà" alla view le modifiche rese disponibili dai vari `WorldObserver`.

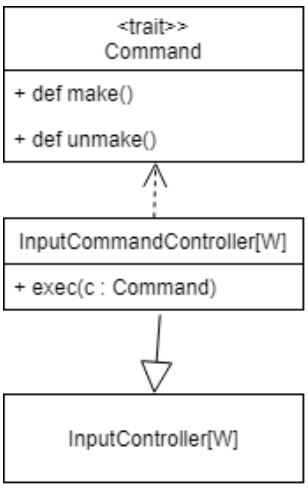


Figura 3.9: struttura *InputController*

3.4.3 Logiche di simulazione

La sezione `Logic` del front-end è stata strutturata per permettere il *Bridge* verso il simulatore: si è creata una struttura (figura 3.10) che descrivesse una logica di simulazione generale attraverso la creazione dei concetti di `SimulationContract` (è un *contratto* verso la simulazione esterna, permette di inizializzarla e farla ripartire), `SimulationPrototype` (descrive il prototipo della simulazione) e `ExternalSimulation` (definisce il tipo della simulazione esterna).

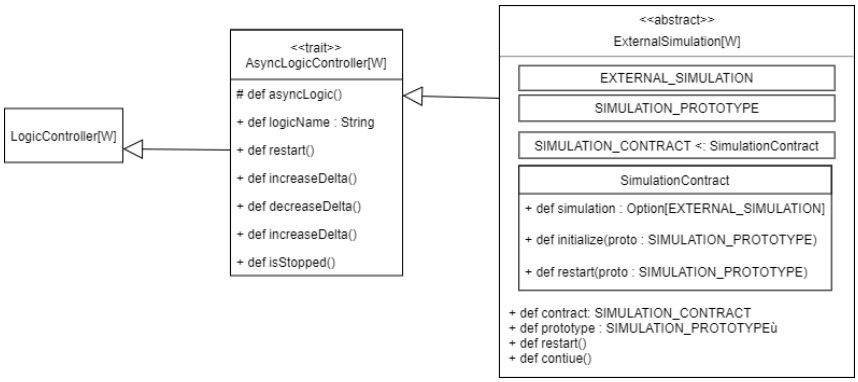


Figura 3.10: struttura logica che permetterà di effettuare il bridge verso *SpaceAwareSimulator*

3.5 View

La struttura della view è indipendente dalla libreria grafica che si andrà ad utilizzare in quanto sono stati descritti i concetti principali che la struttura grafica deve possedere. Quindi, per cambiare libreria grafica, si andranno a modificare solamente le implementazioni di questi concetti generali. Tale struttura è mostrata in figura 3.11.

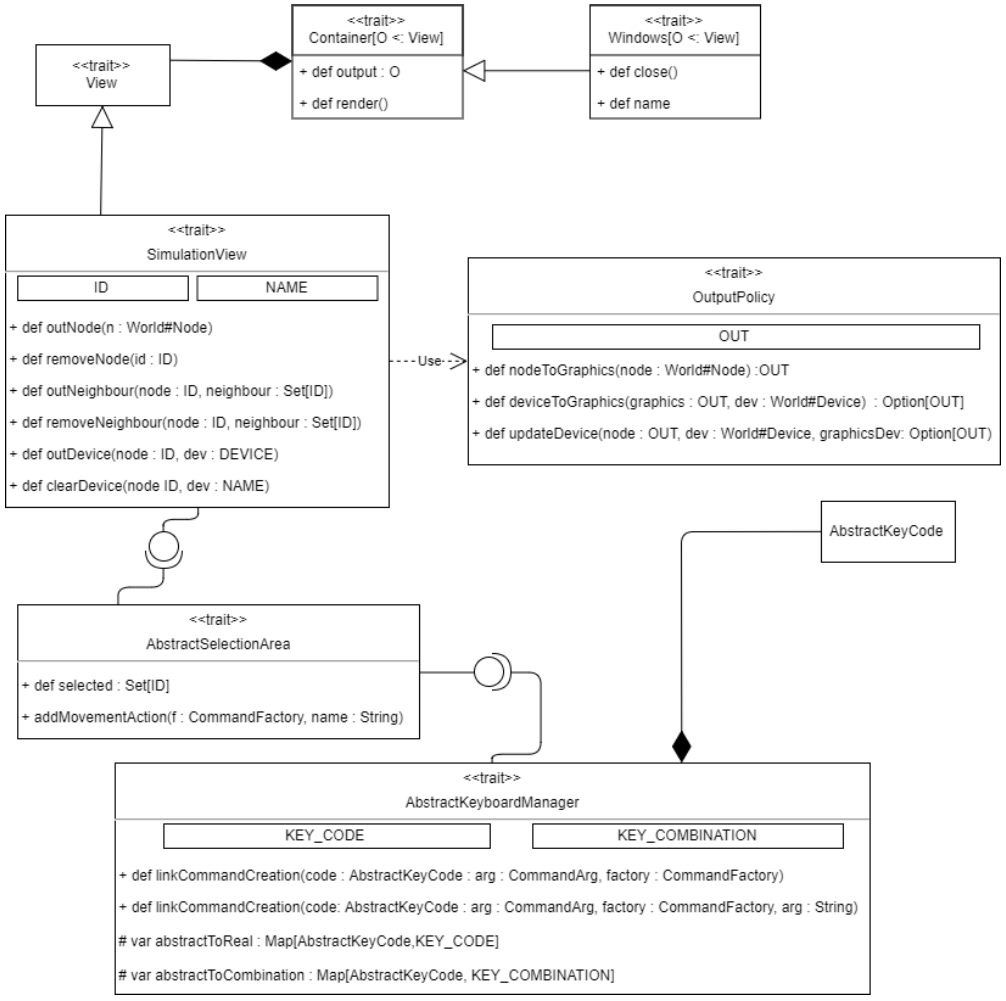


Figura 3.11: struttura generale dell'interfaccia grafica, in cui *View* è il concetto radice e *Container e Window* sono dei contenitori di una possibile *View*. *SimulationView* è l'interfaccia base per la schermata di visualizzazione ed utilizza *OutputPolicy* per mostrare gli elementi. *AbstractSelectionArea e AbstractKeyboardManager* sono due concetti utilizzati per l'interazione con l'utente, in particolare il gestore della tastiera descrive dei tipi generici, *KEY_CODE* e *KEY_COMBINATION*, che dipendono dalla libreria scelta.

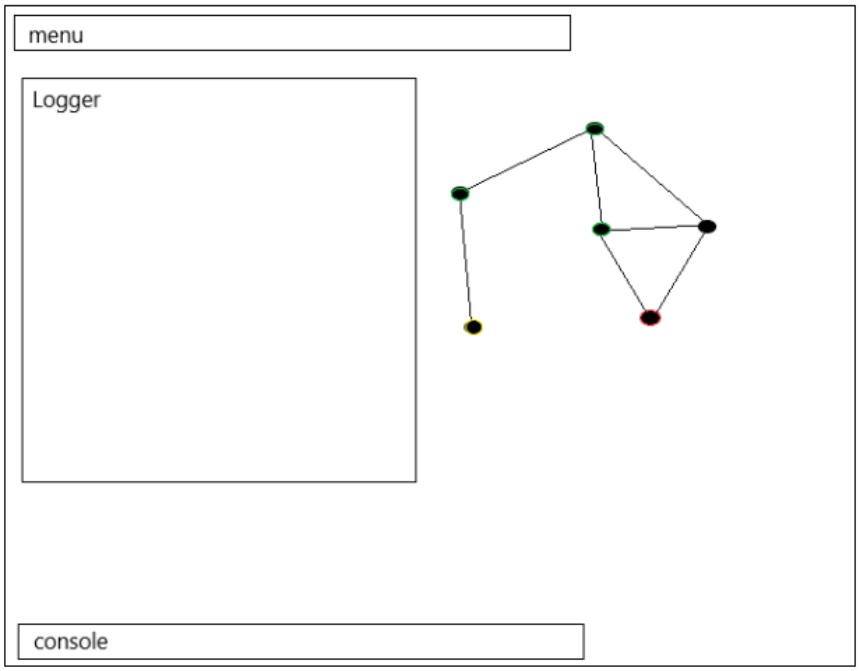


Figura 3.12: sketch dell'interfaccia grafica

La strutturazione della view realizzata in questo progetto di tesi seguirà quest'architettura aggiungendo alcuni concetti che permettono di usare delle gesture per spostarsi e ingrandire il mondo simulato. La schermata del front-end si suddividerà in quattro macro parti:

- **menu bar**: consiste in un menù tramite cui l'utente può chiudere l'applicazione o visualizzare un aiuto per l'utilizzo del software;
- **area di log**: pannello utilizzato per mostrare il log effettuato durante la simulazione;
- **console**: l'utente potrà interagire con il sistema anche utilizzando una console che verrà mostrata in basso nella finestra;
- **ambiente di simulazione**: in questa sezione, verranno mostrati i nodi e i device in base alla politica di output scelta. Si potrà, inoltre, interagire con il mondo attraverso la tastiera e il mouse.

Il *mockup* dell'interfaccia grafica è mostrato in figura 3.12.

Il diagramma della classi mostrato in figura 3.13 descrive la struttura ideata per sfruttare in questo contesto la libreria `ScalaFX`.

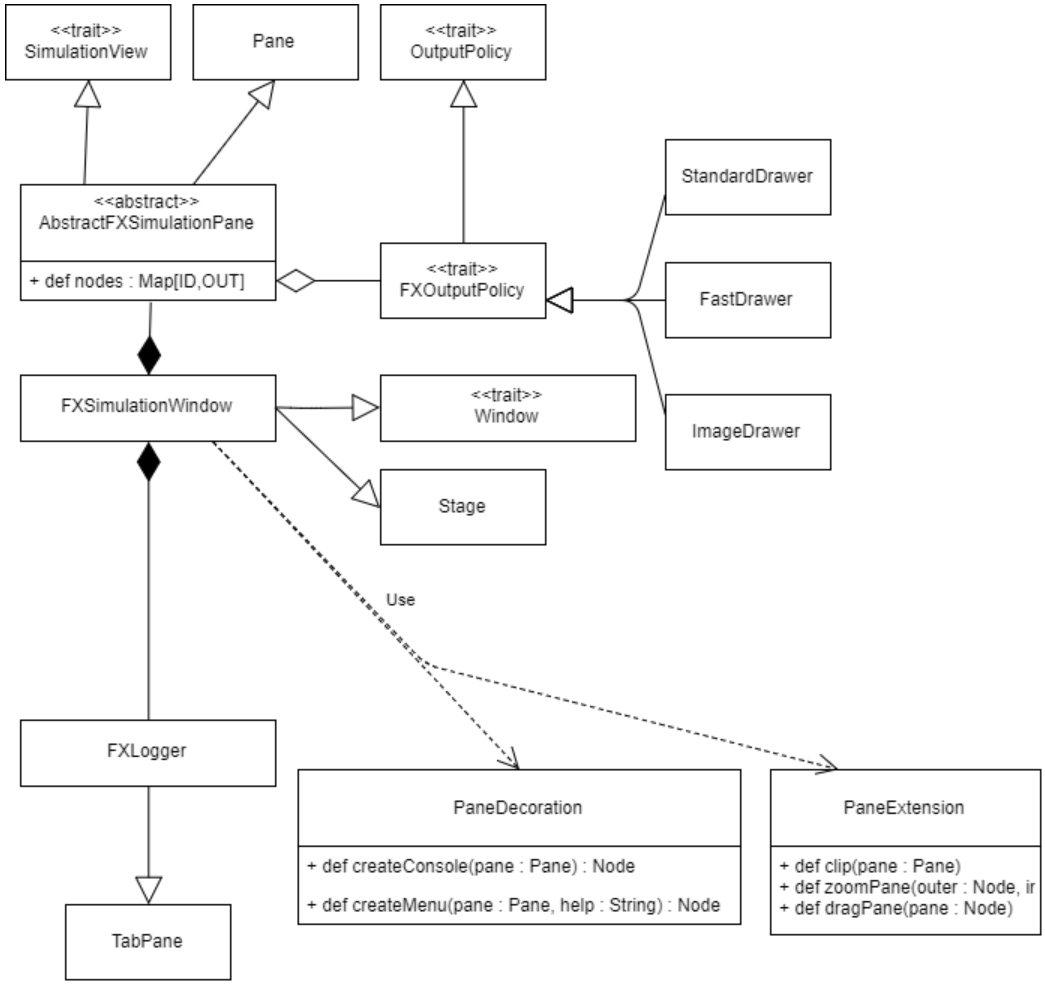


Figura 3.13: struttura riassuntiva dell'interfaccia grafica usando *ScalaFX*. Rispetto a quella generale, vengono aggiunti i concetti di *PaneDecoration* (usato per creare console e menù) e *PaneExtension* (usato per creare pannelli in cui sono abilitate le gesture di drag e zoom)

Capitolo 4

Progettazione di dettaglio

In questo capitolo verranno descritte le ulteriori scelte prese per implementare tutti quei requisiti funzionali che non dipendevano direttamente dalla struttura architetturale del software. Nell'ultima sezione, infine, si descriverà in che modo l'intera struttura creata si possa interfacciare con i concetti descritti nel modello logico di `ScaFi`.

4.1 Configurazione e avvio di una simulazione aggregata

In questa sezione verranno presentate tutte le scelte effettuate per permettere sia una configurazione conforme in tutte le possibili modalità e sia l'avvio di una simulazione aggregata.

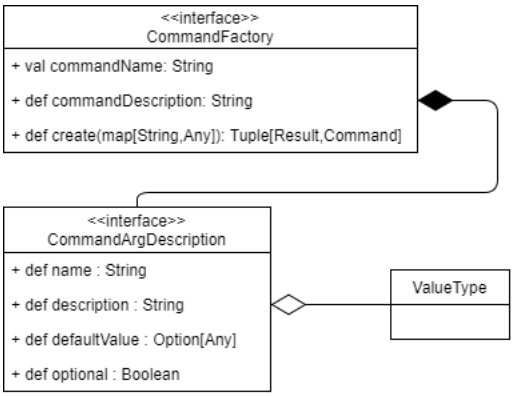


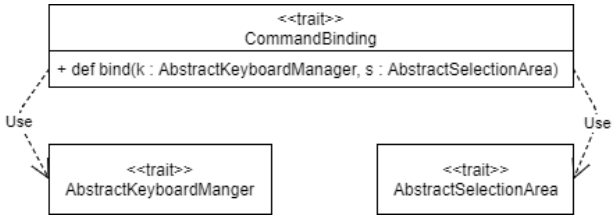
Figura 4.1: strutturazione del concetto *CommandFactory*

In primo luogo, è opportuno dare la definizione di `CommandFactory` (figura 4.1): entità con il compito di creare una famiglia di comandi aventi delle caratteristiche simili tra loro. Esso definisce, inoltre, il nome, la descrizione e gli argomenti che accetta il comando che andrà a creare. Le command factory, quindi, permettono di effettuare una descrizione generica del comando che non dipende direttamente da come esso verrà creato (se via interfaccia grafica o via tastiera).

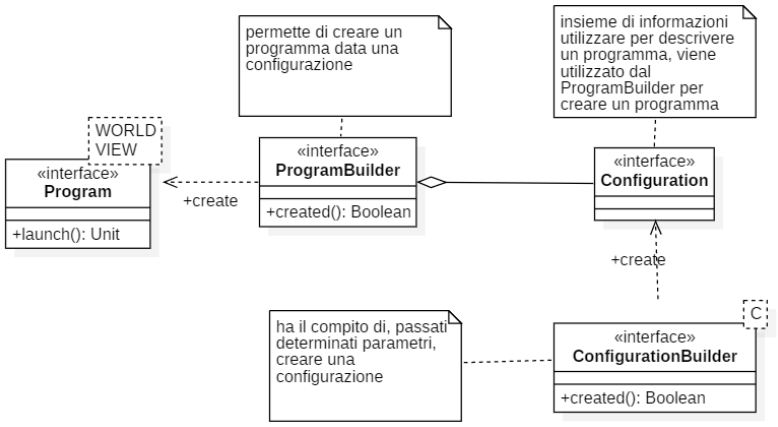
Per permettere la creazione e la configurazione dell'applicazione, la struttura è stata suddivisa in più concetti. L'interfaccia `Program` contiene la descrizione di tutto ciò di cui necessita un programma per essere avviato e permette di avviare l'applicazione collegando tra loro gli ambienti che ha a disposizione. Un ambiente è un'entità tramite cui si descrivono una serie di concetti connessi tra loro come quello dell'applicazione stessa e quello della view (dove vengono descritti sia il gestore della tastiera, sia il gestore di selezione dei nodi). In più vi è un'ultima entità chiamata, `CommandBinding`, che effettua il bind tra i codici della tastiera e gli argomenti per creare i comandi (vedi `CommandFactory`). In `Configuration`, classe che può essere creata via builder o caricata da file, ecc. vengono definiti una serie di concetti che permettono di descrivere il programma attraverso una configurazione. Essa può essere passata ad un `ProgramBuilder` che costruirà un Program in base alla configurazione passata. `Configuration`, in sostanza, verrà utilizzata per impostare gli ambienti utili a creare l'istanza di `Program` (in figura 4.2 viene analizzata la struttura creatasi).

4.1.1 Meta configurazione

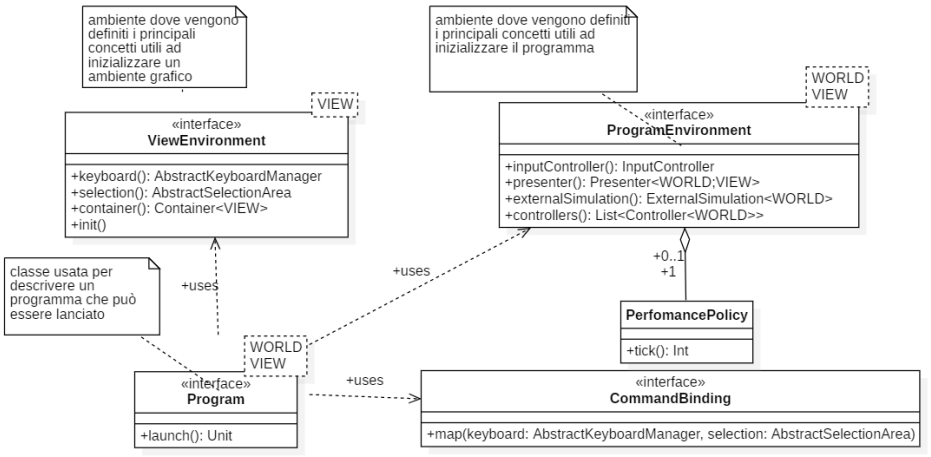
La creazione e la configurazione delle applicazioni sono realizzabili non solo per via programmatica, ma anche tramite interfaccia grafica e tutto ciò grazie all'utilizzo di un linguaggio che può essere in qualche modo interpretato in base alla tecnologia di input. Per fare ciò si è riutilizzato il concetto di `CommandFactory`: una serie di comandi descritti attraverso delle `CommandFactory` andranno a creare una `Configuration` per poi avviare l'applicativo. Inoltre, per permettere di strutturare più linguaggi, sono stati aggiunti i concetti di `Parser` e `Virtual Machine`. Un `Parser` è un'entità che ha il compito di verificare, dato un valore di ingresso, se è possibile creare un comando e in tal caso restituirlo. Tale Parser, per semplificare il proprio lavoro, può utilizzare un insieme di factory. Infine, una virtual machine è l'ambiente in cui i comandi verranno eseguiti e, in questo contesto, ne sono stati individuati due: uno a 'configuration-time' e uno a 'simulation-run-time'. La differenza sta nel fatto che quello a configuration-time non modifica il mondo, ma ha il compito di configurarlo e inizializzarlo. Quello a 'simulation-run-time', invece, una volta creato un comando, lo manda in esecuzione all'interno di `InputController` (la struttura finale è in figura 5.1a).



(a)



(b)



(c)

Figura 4.2: in (a) viene descritto come funziona *CommandBinding*; in (b) è mostrata la struttura per la creazione e la configurazione di *Program*; in (c) si mostrano le relazioni tra *Program* e gli *Environment*

4.1.2 Avvio

Sono state strutturate due possibili modalità d'avvio alternative alla linea di comando: via interfaccia grafica e via console. Entrambe utilizzano lo stesso insieme di `CommandFactory` per essere create e quindi verranno modificati gli stessi parametri di configurazione. Così facendo, data una nuova implementazione di `CommandFactory`, si potranno ampliare quelli che sono i parametri di configurazione in entrambe le modalità, senza modificare le implementazioni. In figura 4.3 viene mostrato il diagramma **UML** della struttura appena descritta e un piccolo *sketch* dell'interfaccia grafica.

4.2 Sistema di log

Il front-end garantisce la gestione di un sistema di log articolato in cui sarà possibile scegliere l'interpretazione da dare ai dati ricevuti. Per permettere tale rappresentazione si è suddiviso il log in *channel* in ciascuno dei quali si scrivono informazioni tra loro correlate (ad esempio esisterà un canale di errore dove verranno inviati tutti i log prodotti dagli errori del sistema). Ogni `Log` è caratterizzato da un nome, un valore e un canale. `LogManager` (sottotipo di `Source` e strutturato come descritto nel pattern Singleton [5]) permette di creare il `Log`, riceverlo e propagarlo ai vari `LogObserver`. Un Observer decide in che modo gestire il log (ad esempio `FileOutputObserver` salverà il log in un file con lo stesso nome del canale) decidendo inoltre quali canali accettare. Tra le varie implementazioni di `LogObserver` ne esiste una che produce un output grafico seguendo una strategia preimpostata. In figura 4.4 è presente la struttura completa.

4.3 Collegamento con i concetti di `ScaFi`

La struttura finora descritta è da contestualizzare nell'ambito di `ScaFi` e, in particolare, si ha bisogno di descrivere l'insieme di classi che permettono al front-end di interfacciarsi con il simulatore. `ScafiLikeWorld` è un'estensione di `SensorPlatform` con i tipi descritti da `SpaceAwereNetwork` (`ID` di tipo `Int`, `NAME` di tipi `String`...) mentre `ScafiBridge` è il concetto che fa da ponte tra il front-end e il simulatore. Il bridge, internamente, definisce una serie di concetti usati per inizializzare una simulazione aggregata (il nome del programma aggregato, il raggio del vicinato...) estendendo le funzionalità descritte in `ExternalSimulationController`. Visto che `ScafiBridge` è sottotipo di `AsyncLogic` si suddividerà in due parti, una sincrona e una asincrona. La parte asincrona effettuerà in modo ciclico le seguente azioni:

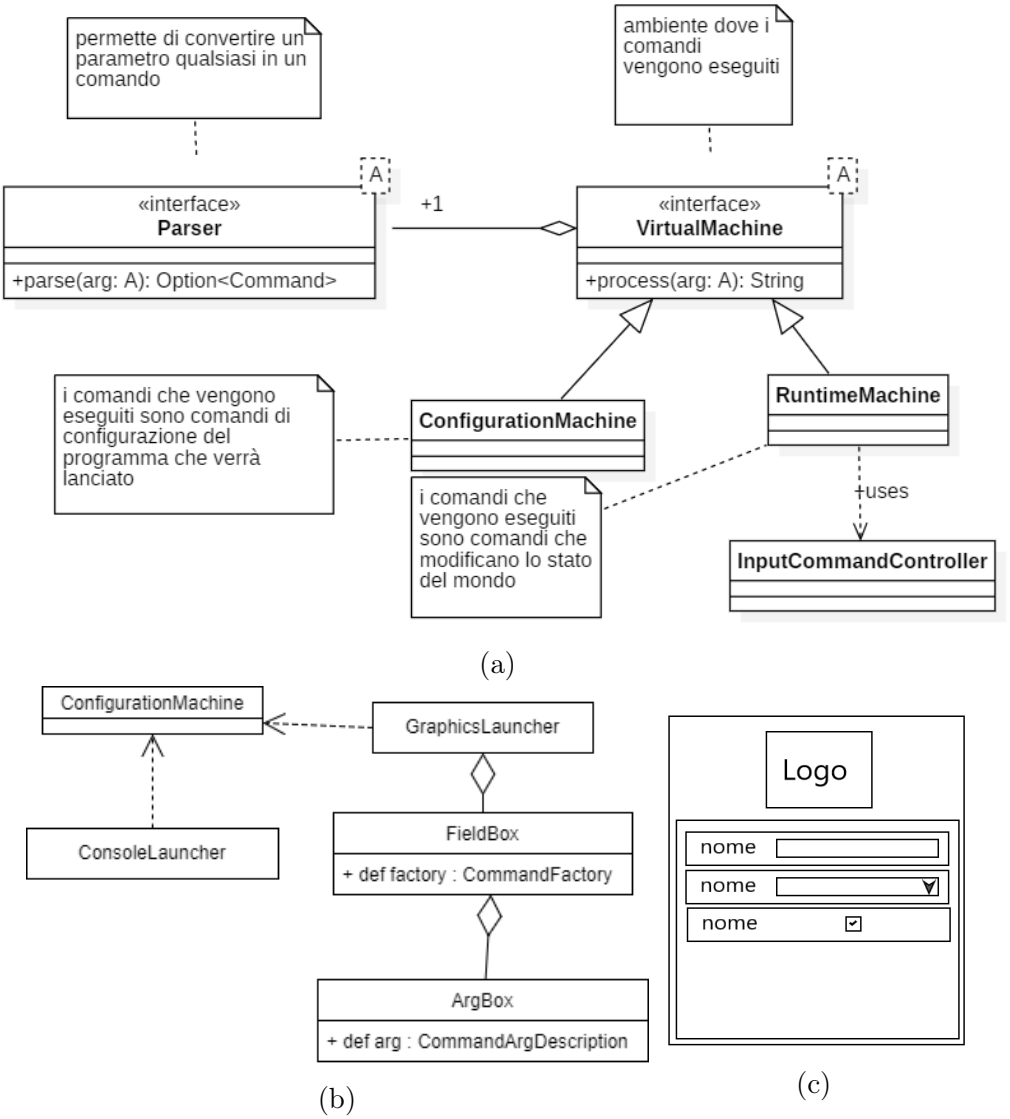


Figura 4.3: in (a) viene descritta la struttura per creare un linguaggio; in (b) sono mostrate le due possibili tecniche d'avvio dell'applicazione. In particolare, `FieldBox` descrive l'insieme di parametri da inserire per creare un comando di configurazione e `ArgBox` è utilizzato per visualizzare un singolo argomento; in (c) vi è un piccolo mockup dell'interfaccia d'avvio

- esegue il programma aggregato,
- aggiorna la mappa con tutti gli export da valutare,
- verifica se sono presenti nuove meta azioni da mettere in coda,

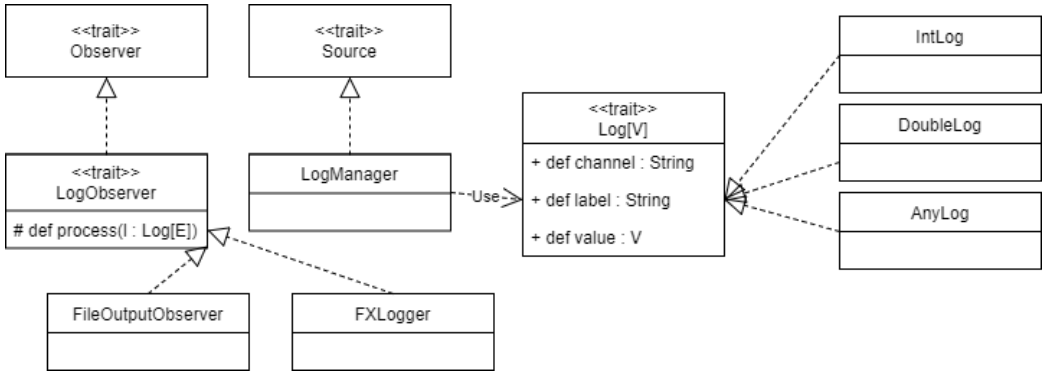


Figura 4.4: diagramma delle classi che descrive la gestione dei log nel front-end.

– esegue le meta azioni.

La parte sincrona dovrà osservare le modifiche effettuate sul modello logico e aggiornare il modello del front-end in modo tale da visualizzare i cambiamenti e i valori restituiti dal programma aggregato. In particolare dovrà:

– valutare tutti gli export prodotti fino a quel momento,

– verificare l'insieme di nodi che sono stati modificati nel modello logico,

– aggiornare la posizione di tali nodi nel modello del front-end.

Questa struttura attraverso le due logiche (quella applicativa e di aggiornamento) eseguite a velocità diverse permette il disaccoppiamento tra l'effettiva produzione di un cambiamento sul modello logico e la sua visualizzazione nell'interfaccia grafica. Al livello del front-end si è pensato di trattare l'*export* in modo simile a come vengono trattati i sensori, cioè verrà tradotto in un device associato ad un nodo. Per permettere di convertire un export in un device c'è stata la necessità di aggiungere `ExportValutation`: è una `Strategy` [5] che, preso un `Export`, lo converte in un valore da visualizzare nell'interfaccia grafica. Questo permette di mostrare l'export nella modalità preferita dall'utente. Per creare le `MetaAction` al livello del front-end si è introdotto un Builder, `MetaActionProducer`: descrive come, da un *export* o comunque un valore generico, si può creare una `MetaAction`.

Si è descritto l'insieme di parametri usati per la configurazione del sistema (riassunti in figura 4.1) che rappresentano:

– **come inizializzare il mondo**: permette di posizionare i nodi secondo una certa logica che può essere casuale o seguire un certo pattern (ad esempio a griglia);

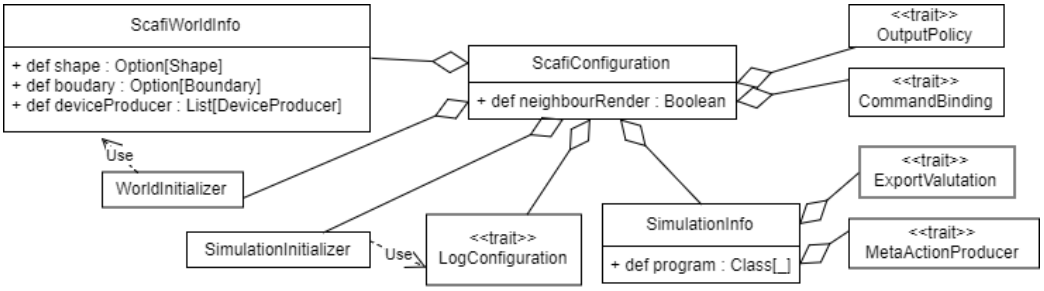
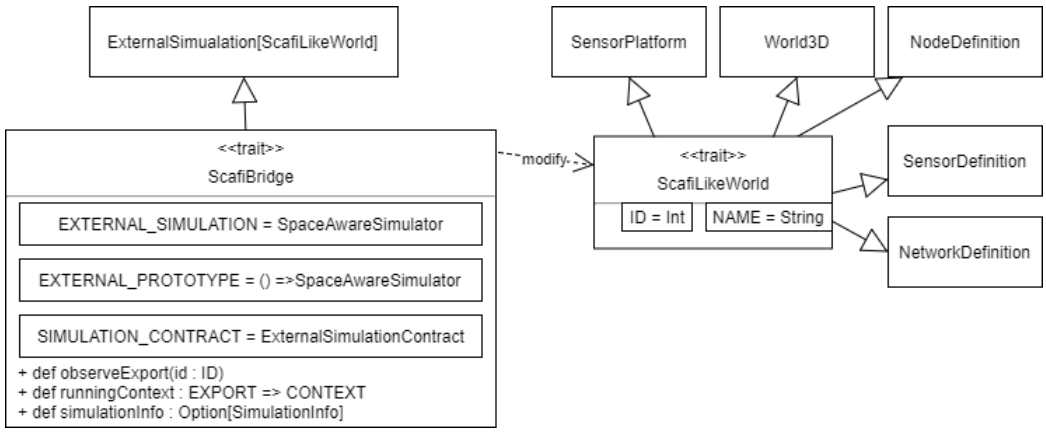


Figura 4.5: configurazione per avviare una simulazione in *ScaFi*

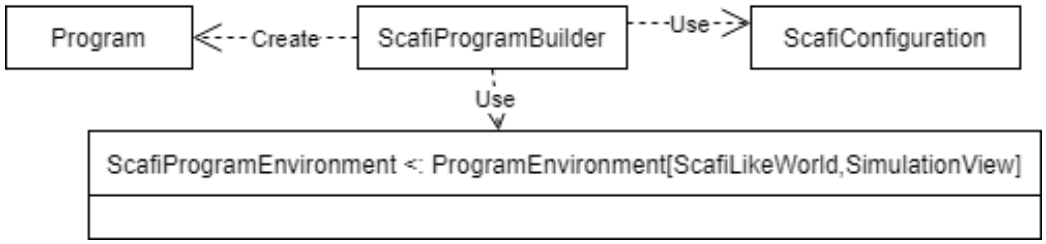
- **come inizializzare la simulazione**: si permette di descrivere i parametri utili in una simulazione aggregata (come ad esempio il raggio del vicinato, il nome del programma aggregato, valutazione dell'export, meta azioni utilizzabili...);
- **configurazione di log**: l'utente può scegliere il tipo di log da effettuare, quali canali mostrare,..
- **informazioni riguardanti il mondo**: possono essere modificati device di ogni nodo, la figura associata e il `Bound` del mondo;
- **rendering dei vicinato** c'è la possibilità di scegliere se effettuare il rendering dei vicini oppure no;
- **aggiornamento view** attraverso `PerformancePolicy` si potrà scegliere il tempo di aggiornamento della `View`;
- **collegamento tra pulsanti da tastiera e comandi da eseguire**: è possibile creare la propria configurazione ed associare ad ogni pulsante una logica diversa da quella standard.

Le varie `CommandFactory` usate per descrivere il linguaggio di configurazione e per effettuare le modifiche sul mondo sono state descritte in `ScafiParser`. Per permettere l'avvio dell'applicazione è stato descritto uno `ScafiProgramEnvironment` che indica quali sono i componenti principali che deve avere il front-end e che verrà utilizzato da `Program` per avviare la simulazione.

La figura 4.6 mostra la struttura creatasi.



(a)



(b)

Figura 4.6: in (a) vengono mostrate le dipendenze di *ScafiLikeWorld* e l'interfaccia di *ScafiBridge*; in (b) è graficata la creazione di un programma che esegua applicazioni aggregate in *ScaFi*.

Capitolo 5

Implementazione

In questo capitolo verranno catturati quelli che sono gli aspetti implementativi utili a comprendere meglio i vari dettagli descritti nella fase di progettazione.

5.1 Creazione del mondo

Il mondo descritto in fase di progettazione è stato concretizzato nella classe `World` nel seguente modo:

```
//describe a place where an immutable set of node are located
trait World {
  //abstract type declaration
  type ID
  type NAME
  type DEVICE <: Device
  type P = Point3D
  type S <: Shape
  type NODE <: Node
  //A boundary of the world (a world may has no boundary)
  def boundary : Option[Bound]
  //get all nodes on this world
  def nodes :Set[NODE]
  //return the node with ID specified
  def apply(id : ID) : Option[NODE]
  //return a set of node with the IDs specified
  def apply(nodes : Set[ID]) : Set[NODE]
  //Node describe an immutable object in a world
  trait Node {
    val id : ID
    def position : P
    def shape : Option[S]
    def devices: Set[DEVICE]
    def getDevice(name : NAME) : Option[DEVICE] = devices find {_.name == name}

    override def toString = s"Node id = $id, position = $position, shape = $shape, devices
        = $devices"
```

```
  }
  //a generic immutable device that could be attached on a node
  trait Device {
    val name : NAME
    override def toString: String = "device name = " + name
  }
}
```

il risultato ottenuto permette una lettura intuitiva di quello che rappresenta un mondo in questo contesto e che può essere arricchito come segue:

```
//a world with a network
trait ConnectedWorld {
  this : ConnectedWorld.Dependency =>
  //the type of network
  type NET <: Network
  trait Network {
    //the neighbours of a node
    def neighbours(n: ID): Set[ID]
    //the neighbour in the world
    def neighbours(): Map[ID,Set[ID]]
    //set a neighbours of a node
    def setNeighbours(node :ID,neighbour :Set[ID])
  }
  //return the current representation of the network
  def network : NET
}
object ConnectedWorld {
  type Dependency = World
}
```

attraverso i *self-type* si è potuto descrivere `ConnectedWorld` in termini di dipendenze con `World`. Così facendo, sarà possibile descrivere un mondo attraverso i suoi *componenti*, rendendo il concetto di world componibile. In particolare questo si vede in `SensorPlatform`, mostrata nel codice che segue, che è stata definita come l'insieme di componenti essenziali ad una piattaforma:

```
trait SensorPlatform extends SensorWorld
  with ConnectedWorld
  with SensorConcept
  with AbstractNodeDefinition
  with SimpleSource
  with BoundaryDefinition {
}
```

Come ci aspettiamo `ScafiLikeWorld` estenderà questo concetto descrivendo quali implementazioni usare:

```
/**
* scafi world definition
*/
trait ScafiLikeWorld extends SensorPlatform
  with World3D
```

```
  with SensorDefinition
  with StandardNodeDefinition
  with StandardNetwork {

  override type ID = Int
  override type NAME = String
}
```

visto che `ScafiLikeWorld` verrà utilizzato da più oggetti durante l'esecuzione e dato che è una singola istanza, si è pensato di utilizzare il pattern Singleton implementando `scafiWorld`:

```
//ScafiLikeWorld singleton concept
object scafiWorld extends ScafiLikeWorld {
  var boundary: Option[SpatialAbstraction.Bound] = None
}
```

5.2 Gestione della simulazione

Durante la fase di implementazione è nato un particolare interesse nel comprendere come gestire le meta-azioni e gli *export*. Come introdotto in fase di progettazione, sono stati descritte due entità, una per la gestione della meta azioni (`MetaAction`) e una per la gestione dell'export (`ExportValutation`).

`MetaAction`, attualmente, supporta un numero limitato di azioni mostrate nel codice che segue:

```
trait MetaAction
/**
* meta action used to move a node into another position
*/
case class NodeMovement(id : ID, point : P) extends MetaAction

/**
* meta action used to move node using a dt movement
*/
case class NodeDtMovement(id : ID, dt : (Double,Double)) extends MetaAction

/**
* meta action used to change the value of a sensor
*/
case class NodeChangeSensor(id : ID, sensor : LSNS, value : Any) extends MetaAction

/**
* meta action used to move a set of node
*/
case class MultiNodeMovement(movementMap : Map[ID,P]) extends MetaAction

/**
* a meta action used to change a set of node sensor value
*/
```

```
case class MultiNodeChangeSensor(ids : Set[ID], sensor : LSNS, value : Any) extends
    MetaAction
```

Ogni `MetaAction` descrive i dati necessari per la propria esecuzione, come poi effettivamente eseguirla sarà compito di `SpaceAwareSimulator` che, in base all'azione ricevuta, deciderà che cosa modificare nel suo ambiente. In questo modo, dato che rappresentano un dato immutabile, le stesse meta azioni possono essere riutilizzate. La logica associata ad ogni `MetaAction` è stata descritta nel metodo `computeAction` mostrato di seguito:

```
private def computeAction(meta : MetaAction) : Unit = meta match {
  case NodeMovement(id,point) => this.setPosition(id,point)
  case NodeDtMovement(id,dt) => val currentPosition = this.space.getLocation(id)
  this.setPosition(id,Point3D(currentPosition.x + dt._1, currentPosition.y + dt._2,
       currentPosition.z).asInstanceOf[P])
  case MultiNodeMovement(map) => map.foreach {x => this.setPosition(x._1,x._2)}
  case NodeChangeSensor(id, sensor,value) => this.chgSensorValue(sensor,Set(id),value)
  case _ =>
}
```

Al livello del front-end, `MetaActionProducer` si dovrà preoccupare di convertire un valore generico in una meta azione. Per attuare questa conversione, `MetaActionProducer` è stato strutturato come una funzione che possedesse tale capacità e il risultato ottenuto è presentato dal codice che segue:

```
//describe a meta action producer used to create a meta action
trait MetaActionProducer[O] {
  //convert any val into output value
  def valueParser : (Any) => (Option[O])
  //put the action used to parse any val
  def valueParser_= (function : (Any) => (Option[O]))

  def apply(id : ID, export : EXPORT) : MetaAction =
    if(valueParser(export.root()).isDefined) {
      this.apply(id,valueParser(export.root()).get)
    } else {
      MetaActionManager.EmptyAction
    }

  def apply(id : ID, argument : O) : MetaAction
}
```

È stato definito il concetto di `ExportValutation` il cui compito è quello di convertire il valore dell'export in un valore associato ad un sensore. Il pattern Strategy ha permesso di rappresentare questo concetto tramite una funzione:

```
type EXPORT_VALUTATION[A] = (EXPORT => A)
```

In `scafiSimulationExecutor`, implementazione di `ScafiBridge` e di tipo *single instance*, vengono descritte le due logiche principali del sistema. Al-

l'interno di `asyncLogicExecution`, che in linea generale verrà eseguito molte più volte del numero di aggiornamenti dell'interfaccia grafica, viene svolto il programma aggregato e vengono processate le meta azioni:

```
//take SpaceAwareSimulator
val net = contract.simulation.get
//exec scafi aggregate program
val result = net.exec(runningContext)
//add export produced into a map
exportProduced += result._1 -> result._2
val metaActions = this.simulationInfo.get.metaActions
//add meta action if export can parse to meta action value
metaActions.filter(x => x.valueParser(result._2.root()).isDefined).foreach(x =>
    net.add(x(result._1,result._2)))
//process meta action
net.process()
```

la logica sincrona, invece, quando verrà chiamata dallo scheduler controllerà quali sono stati i cambiamenti e modificherà il modello del front-end. Ad esempio, per visualizzare le modifiche dell'export prodotto, prima di tutto lo convertirà in un valore e poi lo inserirà all'interno di un sensore di tipo *output* (non collegato con il modello logico ma creato solamente per motivi di visualizzazione). L'export in generale può essere usato per due motivi:

- per essere valutato e mostrato sull'interfaccia grafica;
- per essere trasformato in una meta azione che modificherà lo stato del mondo logico.

È interessante notare che questi due diversi utilizzi non vanno in contrapposizione: in fase di debug, ad esempio, potremo sia essere interessati a visualizzare il valore dell'export sia ad attuare le meta azioni associati a tale output. Il codice che segue mostra la logica della valutazione dell'export:

```
/*for each export produced the bridge valutate export value and produced output
    associated*/
val exportValutations = simulationInfo.get.exportValutations
if(exportValutations.nonEmpty) {
  //take export modified durint the delta time
  var exportToUpdate = Map.empty[ID,EXPORT]
  exportToUpdate = exportProduced
  exportProduced = Map.empty
  for(export <- exportToUpdate) {
    for(i <- exportValutations.indices) {
      world.changeSensorValue(export._1,indexToName(i),
      exportValutations(i)(export._2))
    }
  }
}
```

Le modifiche effettuate sul modello logico saranno notificate ad un observer (`SimulationObserver`). In base agli id dei nodi modificati, si aggiornerà la

loro rappresentazione nel modello del front-end. Il codice che segue mostra come è stata implementata questa logica in `scafiSimulationExecutor`:

```
//used update the gui world network
var idsNetworkUpdate = Set.empty[Int]
//check the node move by simulation logic
simulationMoved foreach {id =>
  val p = contract.simulation.get.space.getLocation(id)
  /*verify if the position is realy changed (the moved can be produced by gui itself)*/
  world.moveNode(id,p)
  //the id to update in gui network
  idsNetworkUpdate ++= world.network.neighbours(id)
  idsNetworkUpdate ++= contract.simulation.get.neighbourhood(id)
  idsNetworkUpdate += id
}
//update the neighbourhood foreach node
idsNetworkUpdate foreach {x =>
    {world.network.setNeighbours(x,contract.simulation.get.neighbourhood(x))}}
```

5.3 Interfaccia grafica

La strutturazione dell'interfaccia grafica è stata la parte principale di questo progetto ed è stata agevolata dall'utilizzo della libreria `ScalaFX`. Inizialmente sono state strutturate le varie schermate senza introdurre alcun concetto di stile e in seguito, grazie al supporto del CSS, è stato possibile modificare lo stile dell'interfaccia ottenendo un risultato gradevole. Per poter avviare un `Program`, oltre ad utilizzare `ScafiProgramEnvironment`, si ha bisogno di definire anche quello che sarà l'ambiente grafico di esecuzione. In `ScalaFX` ciò si realizza come segue:

```
/**
* standard fx view environment for simulation view
*/
object ScalaFXEnvironment extends ViewEnvironment[SimulationView] {
  /*
   * define the window width and height, some information about
   * image logo and icon logo
  */
  lazy val standardConfiguration = WindowConfiguration.apply(800,600)
  var windowConfiguration : WindowConfiguration = standardConfiguration
  //standard value of fx application
  var drawer : FXOutputPolicy = StandardFXOutput
  //simulation pane
  private lazy val pane = new FXSimulationPane(drawer)
  //main container
  private lazy val cont = new FXSimulationWindow(pane,true,windowConfiguration)
  //keyboard manager
  override def keyboard: AbstractKeyboardManager = pane
  //selection area manager
  override def selection: Option[AbstractSelectionArea] = Some(pane)
  //view container
```

```
  override def container: Container[SimulationView] = cont
  //initialize fx environment
  override def init(): Unit = {
  initializeScalaFXPlatform()
    Platform.runLater {cont}
  }
}
```

5.3.1 Strategia di output

`FXSimulationPane` sfrutterà il concetto di `OutputStrategy` per visualizzare i nodi e i device. Le strategie principali descritte nel front-end sono:

- `StandardFXOutput`: mostra i nodi in base alla loro figura. Per quanto riguarda i sensori, quelli che hanno un valore booleano vengono mostrati come dei cerchi concentrici con colore diverso in base al nome, per gli altri, verrà visualizzata una label con la rappresentazione del valore sotto forma di stringa,
- `FastFXOuput`: simile a `StandardFXOuput` con la differenza che i sensori sono trattati in modo diverso: al posto di creare dei cerchi si modifica il colore della figura geometrica in base al nome del sensore. Questa strategia ottiene performance migliori di quelle della strategia standard,
- `ImageFXOuput`: mostra i nodi con un'immagine scelta prima di eseguire l'applicazione. I device vengono raffigurati come in `StandardFXOutput`,
- `GradientFXOuput`: mostra i nodi con la loro figura. Riconosce solamente i sensori con un valore numerico a cui viene associato poi un colore.

L'implementazione di `GradientFXOutput` è descritta nella sezione di codice successiva:

```
/**
  * a output strategy that associated a color to
  * number, allow to see graphically the result
  * of computation
*/
case object GradientFXOutput extends FXOutputPolicy {
  override type OUTPUT_NODE = javafx.scene.shape.Shape
  override def deviceToGraphicsNode(node: OUTPUT_NODE, dev: DEVICE): Option[OUTPUT_NODE] =
       None

  override def updateDevice(node: OUTPUT_NODE, dev: DEVICE, graphicsDevice:
       Option[OUTPUT_NODE]): Unit = {
    dev match {
      case SensorDevice(sens) => sens.value[Any] match {
```

(a) (b) (c)

Figura 5.1: in (a) viene mostrato l'output di *GradienteFXOutput*; in (b) quello di *ImageFXOutput*; in (c) quello di *StandardFXOutput*

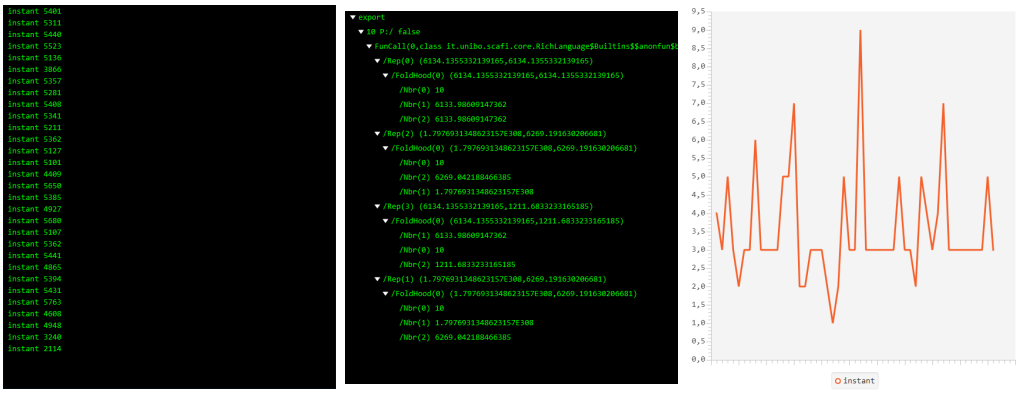
```
        case number : Number => node.fillProperty().setValue(numberToColor(number))
        case _ =>
      }
      case _ => None
    }
  }
  private implicit def numberToColor(v : Number) : Color = {
    val doubleValue : Double = v.doubleValue()
    Color.hsb(doubleValue,1,1)
  }
  override def nodeGraphicsNode (node: NODE): OUTPUT_NODE =
      modelShapeToFXShape.apply(node.shape,node.position)
}
```

Infine vengono mostrate le diverse strategie di visualizzazione nella figura 5.1.

5.3.2 Strategia di log

Il log si struttura in modo simile a quanto descritto con la strategia di output, cioè associa una strategia ad ogni tipo di log. Attualmente sono supportate tre strategie:

- **testuale**: si mostrano semplicemente il valore ed il nome associati al log,
- **tramite diagramma**: se il log è di tipo numerico mostra un grafico con l'andamento nel tempo di tale log,
- **ad albero**: se il dato del log si può interpretare come un albero, allora esso viene visualizzato secondo questa politica (ad esempio, serve per visualizzare l'export associato ad un nodo).



(a) (b) (c)

Figura 5.2: in (a) viene mostrato il log sotto forma testuale, in (b) sotto forma di albero e in (c) sotto forma di grafico

In genere, vengono usate delle strategie standard in base al log ricevuto (un log di tipo stringa ad esempio si associa un output testuale). Si può, però, specificare in modo esplicito come il logger dovrà trattare il log ricevuto nel seguente modo:

```
GraphicsLogger.addStrategy("simulation-round",LogType.textual)
```

di seguito sono presentati degli screenshot (figura 5.2) che mostrano le varie tecniche di log implementate nel front-end.

5.4 Avvio

L'applicazione potrà essere avviata e configurata in diverse modi. Quello standard è via linea di codice e utilizza `ScafiProgramBuilder` al quale devono essere passati come minimo gli argomenti obbligatori che sono:

- come inizializzare il mondo,
- quale programma eseguire,
- come inizializzare la simulazione.

Gli altri parametri (tipo di device associati ai nodi, binding dei comandi, ...) hanno dei valori di default che l'utente può modificare a suo piacimento. Il codice seguente mostra come avviare una simulazione con il programma aggregato `BasicDemo` e con 50 nodi posizionati nel mondo in cui il vicinato ha un raggio di 140 unità:

```
  ScafiProgramBuilder (
    Random(node = 50, width = 500, height = 500),
    SimulationInfo(program = classOf[BasicProgram]),
    RadiusSimulation(radius = 140)
  ).launch()
```

Utilizzando i concetti di `Parser` e di `CommandFactory` è stato possibile creare un configuratore di simulazioni via console:

```
object Console extends App {
  //creation of configuration machine
  val configurationMachine = new ConfigurationMachine(UnixConfiguration)
  //creation of runtime machine
  val runtimeMachine = new RuntimeMachine(UnixRuntime)
  println(international("welcome")(KeyFile.Configuration))
  val log = new ConsoleOutputObserver
  LogManager.attach(log)
  while(!scafiConfiguration.created){//configuration phase
    LogManager.notify(StringLog(Channel.CommandResult,
    Label.Empty,configurationMachine.process(readLine())))
  }
  LogManager.detach(log)
  while(true) { //runtime phase
    println(runtimeMachine.process(readLine()))
  }
}
```

La simulazione descritta con il codice precedente potrà essere lanciata via console con le seguenti istruzioni:

```
random-world 50 500 500
radius-simulation BasicDemo 140
launch
```

Infine è possibile avviare l'applicazione attraverso un configuratore grafico, creato a partire dalla classe `CommandFactory`. Usando `GraphicsLauncher` si potranno compilare i campi di una form mostrata in una semplice interfaccia grafica. Di seguito un'immagine (figura 5.3) esemplificativa dell'avvio della stessa simulazione descritta in precedenza.

5.4.1 Ricerca tramite reflection dei programmi aggregati

Per permettere di visualizzare tutti i programmi aggregati descritti via codice e senza inserire direttamente una stringa associata al nome del programma, si è pensato di *markare* i vari programmi aggregati con le *Java annotations*. Inoltre, dato che alcuni parametri sono difficili da configurare da console o via interfaccia grafica, le *Java annotations* vengono utilizzate per descrivere di cosa il programma aggregato necessiti per essere configurato. L'annotation `Demo` e stata creata nel seguente modo:

```
//mark a class as a scafi demo
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Demo {
  String description() default "";
  SimulationType simulationType() default SimulationType.STANDARD;
}
```

`SimulationType` descrive il tipo di simulazione che ci aspettiamo dal programma aggregato. Per adesso è possibile descrivere le seguenti simulazioni:

- `STANDARD`: crea una simulazione dove è presente un insieme di sensori di tipo acceso spento,
- `ON_OFF_INPUT_ANY_OUTPUT` crea una simulazione dove è presente un solo sensore di input ed è possibile visualizzare un solo output associato ad un *export*,
- `MOVEMENT` l'*export* prodotto viene trattato per essere convertito in una `MetaAction` di movimento.

Per far riconoscere una classe come programma aggregato basterà scrivere:

```
@Demo
class BasicProgram extends AggregateProgram
//movement program
@Demo(simulationType = SimulationType.MOVEMENT)
class BasicMovement extends AggregateProgram ...
//specific simulation
@Demo(simulationType = SimulationType.ON_OFF_INPUT_ANY_OUTPUT)
class AdHoc extends AggregateProgram ...
```

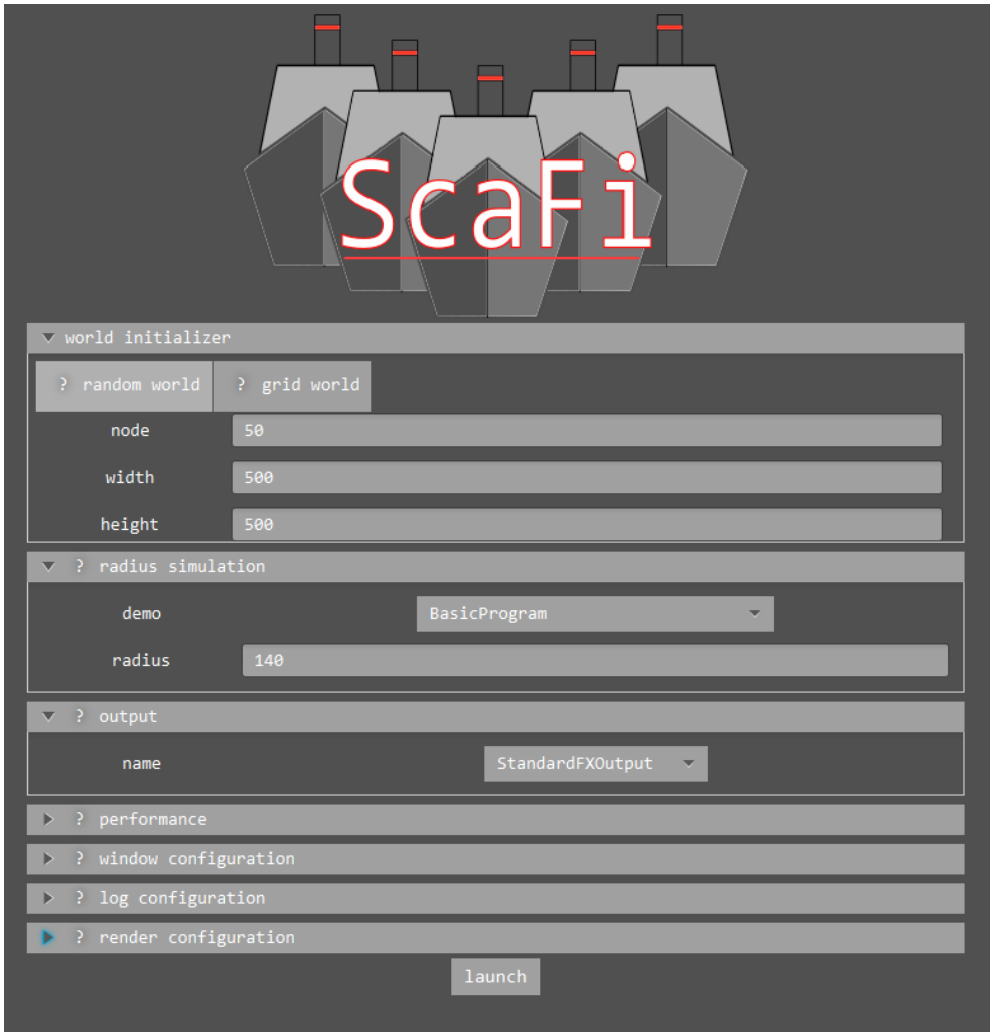


Figura 5.3: schermata di configurazione per una simulazione aggregata

Capitolo 6

Valutazione

Nella prima sezione del capitolo verranno mostrati i risultati grafici ottenuti dal front-end. Nella seconda sezione si mostreranno quattro programmi aggregati, tre statici (i nodi non si muovono durante la simulazione) e uno dinamico (i nodi sono in continuo movimento).

6.1 Risultati grafici ottenuti

La struttura della schermata principale del front-end è mostrata nella figura 6.1 che segue il mockup mostrato in 3.12.

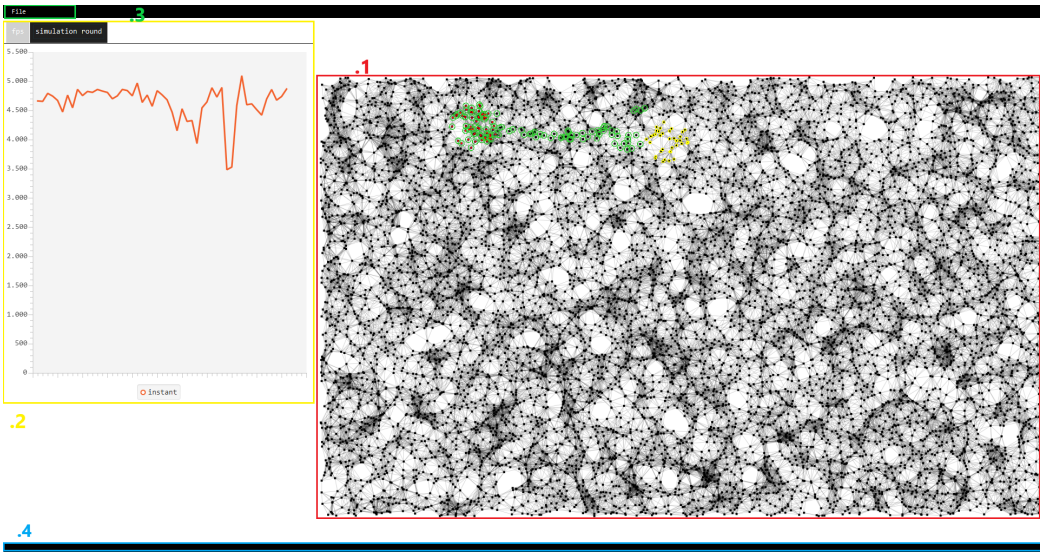


Figura 6.1: schermata principale del front-end

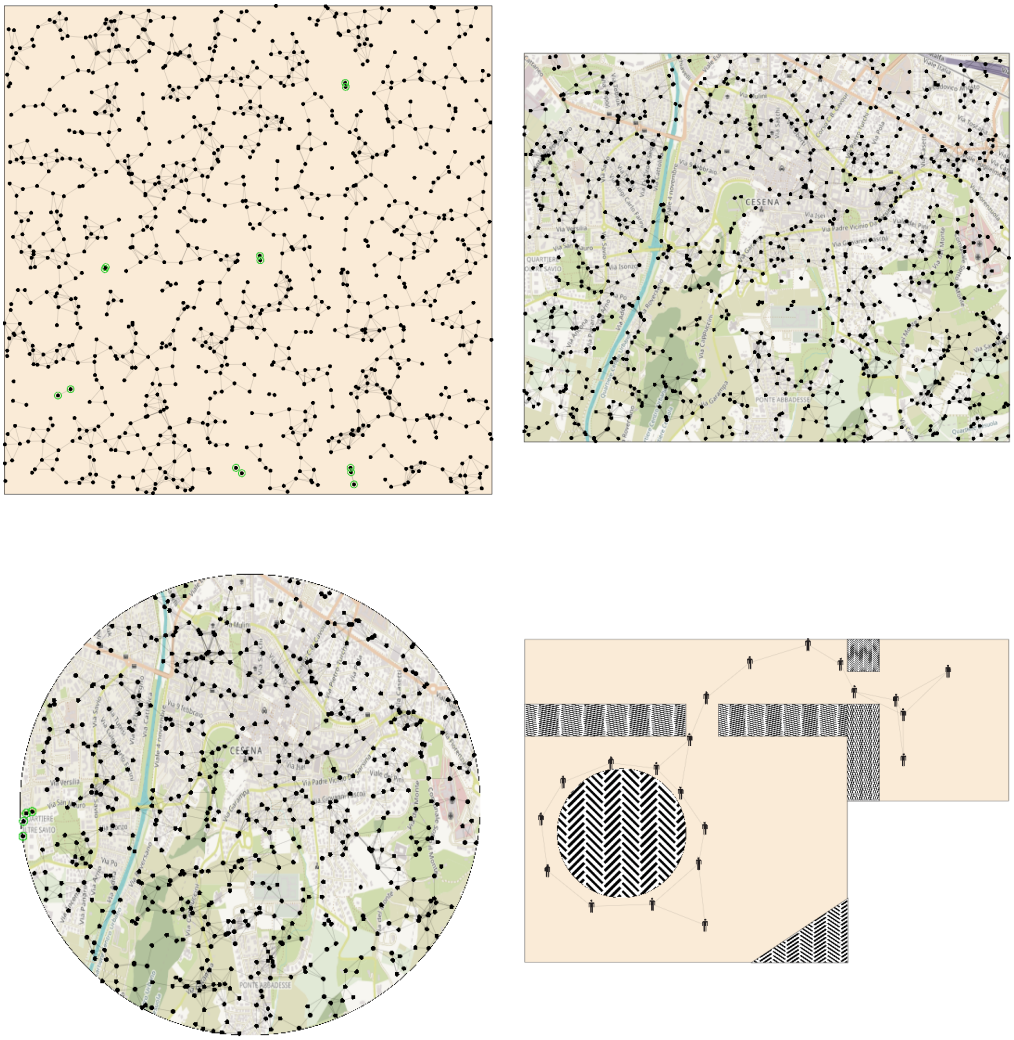


Figura 6.2: esempi di simulazioni eseguite con il front-end.

La struttura creata risulta flessibile per diverse tipologie di simulazione, anche ove si presentino esigenze grafiche particolari. Si mostrano una serie di screenshot (figura 6.2 di ciò che il front-end può produrre modificando i vari parametri di simulazione.

6.2 Esempi e analisi di simulazioni aggregate

Durante tutta la fase di sviluppo è stato importante valutare il lavoro eseguito attraverso delle demo di programmi aggregati descritte nel package `sims`. Questa fase, parallela allo sviluppo, è stata necessaria, inoltre, per testare le performance e le funzionalità del software.

Per verificare le performance si creerà un mondo formato da nodi disposti a griglia e non casualmente così da poter testare il programma nelle stesse condizioni. Si verificheranno i round per secondo e il tempo di esecuzione di ogni round, con un numero medio di vicini pari a 4 per le simulazioni statiche. Per la simulazione dinamica, invece, il numero di vicini medio risulta più approssimato in quanto cambia in ogni simulazione.

Per ogni simulazione verranno mostrati un output grafico e due diagrammi che descrivono:

[1] il tempo medio per eseguire un round al variare dei nodi,

[2] il numero di round eseguiti al secondo al variare dei nodi,

Ogni diagramma utilizzerà gli stessi colori per rappresentare:

- attraverso il blu, i dati raccolti con simulazioni in cui era presente un'interfaccia grafica;
- attraverso l'arancione, i dati raccolti con simulazioni prive di interfaccia grafica.

Purtroppo, non si avranno mai due condizioni identiche di simulazione e, nonostante si sia effettuata una media su diverse simulazioni, si presentano occasionalmente dei dati outlier. In generale, però, siamo interessati a vedere l'andamento generico delle perfomance, quindi avere picchi in eccesso che si normalizzano all'aumentare dei nodi risulta del tutto normale. Un'altra considerazione importante da fare è la seguente: visto che i nodi ai lati hanno un numero di connessioni minori rispetto a quelli centrali in un sistema a griglia, si noterà come con pochi nodi si hanno alte perfomance che si normalizzeranno all'aumentare degli stessi. Questo risultato si ha perchè, con pochi nodi, è più probabile che venga eseguito un round ai lati del world e, tale round, avrà un tempo di esecuzione minore rispetto a quelli centrali per via del minor numero di vicini.

Per eseguire i vari test è stato utilizzato un portatile Dell Inspiron 7570 con un processore Intel i5-8250u con otto gigabyte di ram.

6.2.1 Costrutto rep

Il programma aggregato più semplice, ma comunque degno di essere testato, è quello che segue:

```
@Demo
class BasicProgram extends AggregateProgram {
  override def main() = rep(0)(_ + 1) // the aggregate program to run
}
```

Questo programma su ogni nodo incrementerà di uno il valore precedentemente calcolato. Ogni nodo ha un valore di partenza pari a zero. Di seguito una serie di screenshot 6.3 che mostrano come il front-end gestisce questa semplice applicazione.

I diagrammi descrivono come è presente in questo caso un lieve degrado di performance quando è presente l'interfaccia grafica ed è interessante notare come questa differenza di prestazioni rimanga costante al variare dei nodi, fatto che denota come l'interfaccia grafica introduca un degrado costante (in questo caso) e non associato ai nodi.

La perdita di perfomance è dovuta al fatto che, ad ogni frame, viene aggiornata l'intera rete di nodi (dato che il numero di round è molto maggiore del numero dei nodi), ma visto che la percentuale media è circa il 10%, questo degrado risulta accettabile.

6.2.2 Gradiente

In questo caso il programma aggregato, attraverso il gradiente, mostra per ogni nodo, la sua distanza da un insieme di nodi segnati come sorgente. La simulazione mostra un colore associato alla distanza (figura 6.4).

I diagrammi mostrano come, con questo programma aggregato, le perfomance restano per lo più inalterate: con 20000 nodi, ad esempio, sono praticamente le stesse. Anche in questo caso la rete, ad ogni frame, viene completamente ridisegnata e il lieve degrado può essere associato a questo fattore.

6.2.3 Canale

Questo programma, date una sorgente e una destinazione, si occupa di creare un canale che eviti una zona con degli ostacoli. In output ci sarà un valore booleano che indica il percorso da effettuare per raggiungere la destinazione (figura 6.5).

In questo caso possiamo concludere direttamente che non vi è perdita di performance e ciò è dovuto al fatto che il numero di nodi che vengono modificati ad ogni frame è inferiore al numero globale di nodi renderizzati.

6.2.4 Flock

Questa simulazione esprime l'output come un delta spostamento. La logica descritta tende a raggruppare i nodi formando dei *flock* (raggruppamenti di nodi)(figura 6.6).

In questo caso, vi è una differenza di performance tra le due simulazioni costante: il movimento di un nodo nella schermata è decisamente più pesante che cambiare il colore di un nodo. Il dato positivo è che il degrado anche in questo caso è per lo più costante all'aumentare dei nodi. Il degrado massimo si verifica con 100 nodi dove vi è una differenza di circa il 25%, sempre inferiore alla percentuale definita in fase d'analisi e comunque, con 100 nodi, questa differenza non è del tutto rilevante.

3785	3721	3797	3606	3574	3756	3718	3677	3744	3754
3698	3768	3714	3728	3729	3578	3689	3633	3714	3701
3749	3809	3652	3786	3609	3692	3636	3694	3791	3627
3752	3744	3679	3648	3732	3689	3752	3819	3656	3653
3706	3723	3771	3609	3681	3682	3754	3737	3710	3770
3628	3648	3586	3659	3756	3700	3809	3810	3684	3661
3674	3611	3706	3691	3727	3639	3685	3554	3659	3711
3668	3803	3629	3689	3707	3697	3693	3702	3783	3746
3696	3760	3717	3782	3697	3681	3743	3691	3820	3648
3724	3755	3685	3779	3742	3711	3754	3722	3721	3664

(a)

7642	7564	7758	7401	7394	7582	7692	7486	7606	7628
7589	7585	7513	7481	7522	7438	7513	7473	7604	7577
7496	7570	7574	7632	7410	7545	7466	7532	7677	7560
7566	7609	7541	7403	7540	7526	7627	7733	7523	7469
7698	7597	7670	7518	7559	7526	7697	7679	7621	7600
7527	7496	7571	7362	7598	7505	7596	7660	7542	7504
7475	7411	7507	7510	7497	7574	7475	7461	7505	7515
7512	7618	7485	7576	7524	7413	7469	7479	7702	7571
7557	7523	7628	7714	7557	7459	7631	7510	7618	7422
7559	7677	7572	7621	7723	7509	7563	7609	7545	7501

(b)

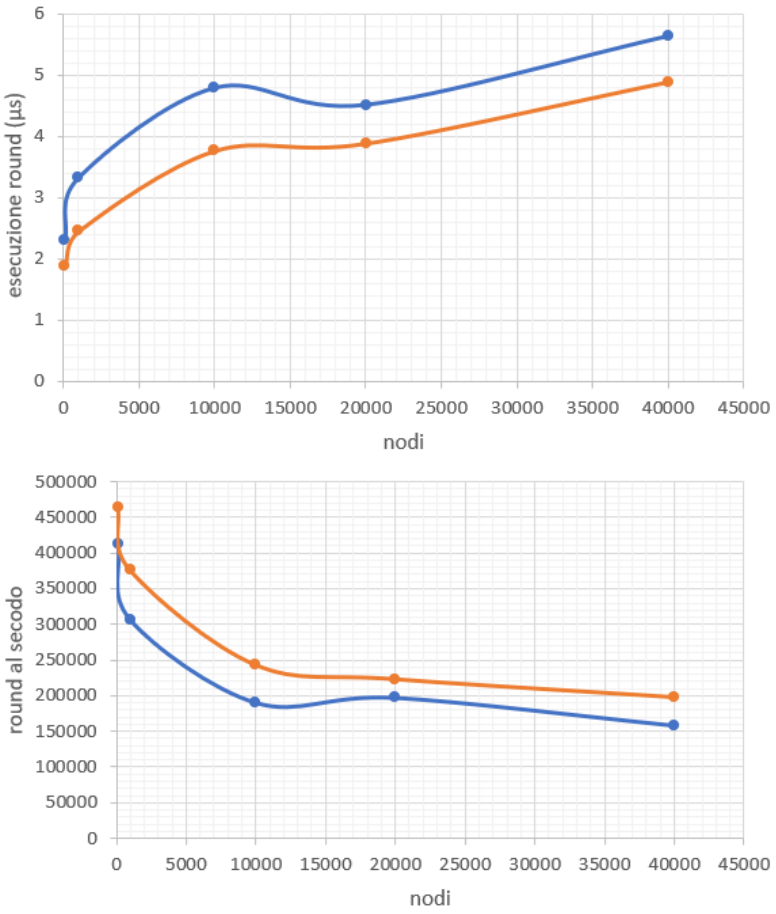
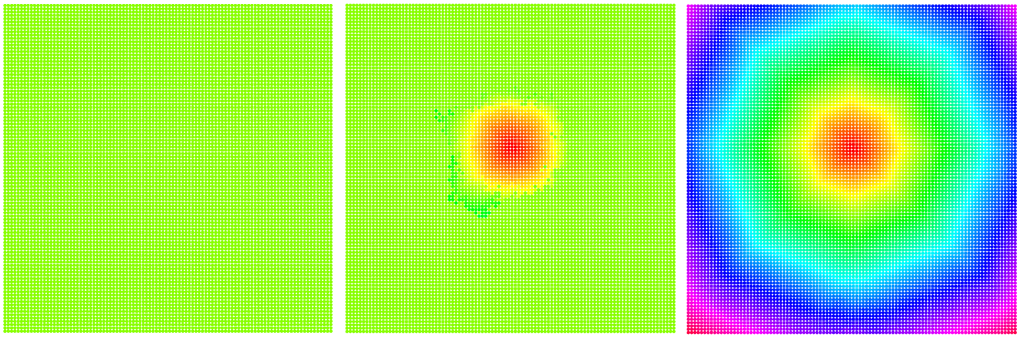


Figura 6.3: semplice output del programma aggregato che usa *rep*.



(a) (b) (c)

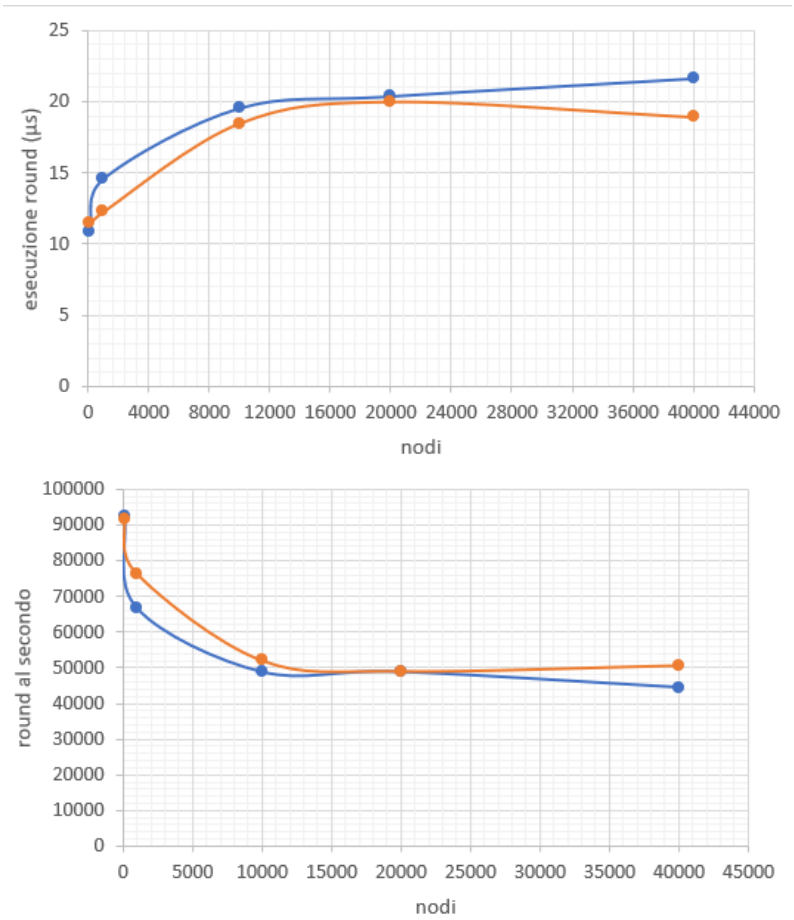
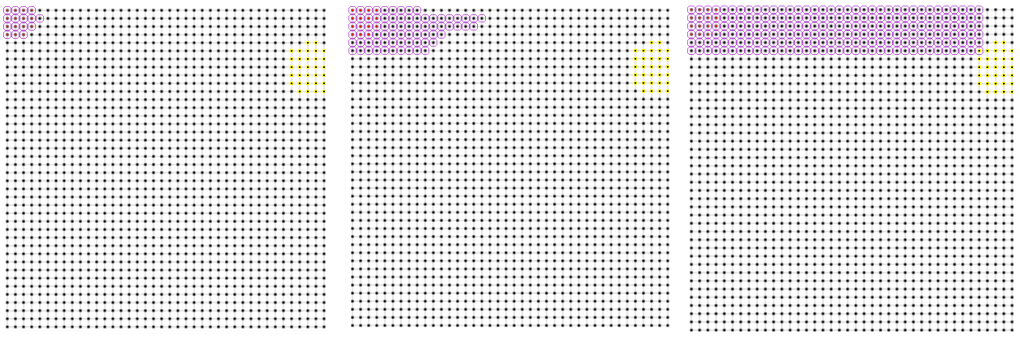
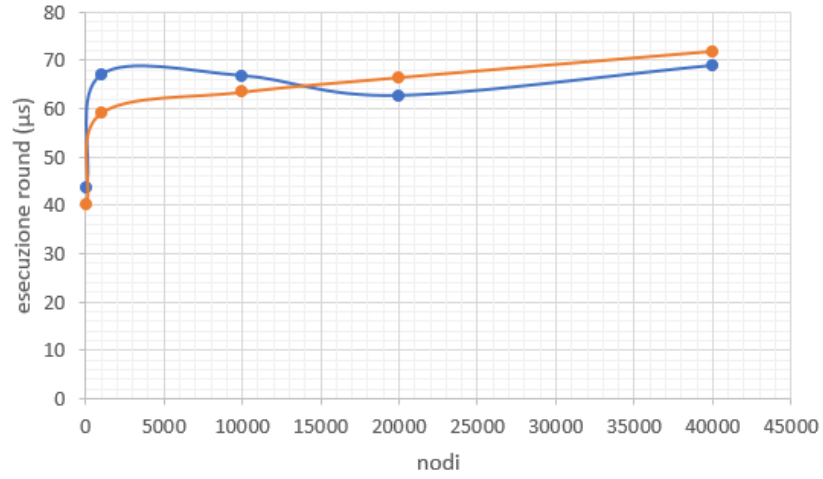


Figura 6.4: output che fa uso di *GradientFXOutput* per mostrare un programma aggregato con il gradiente.



(a) (b) (c)



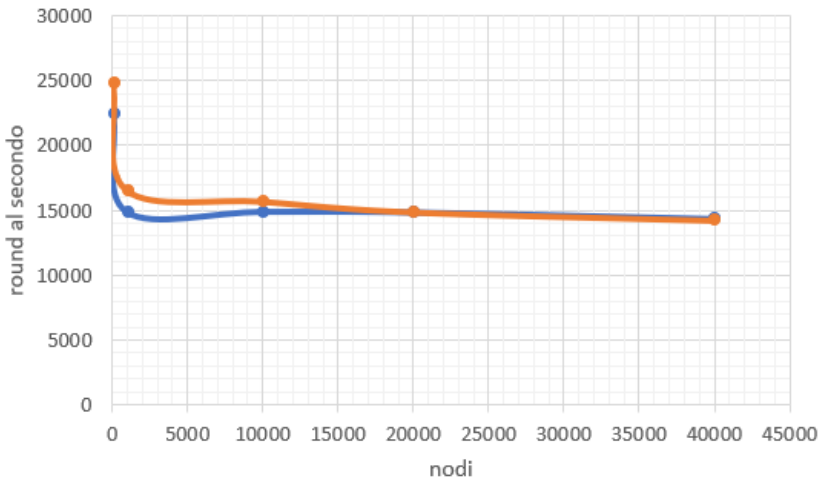
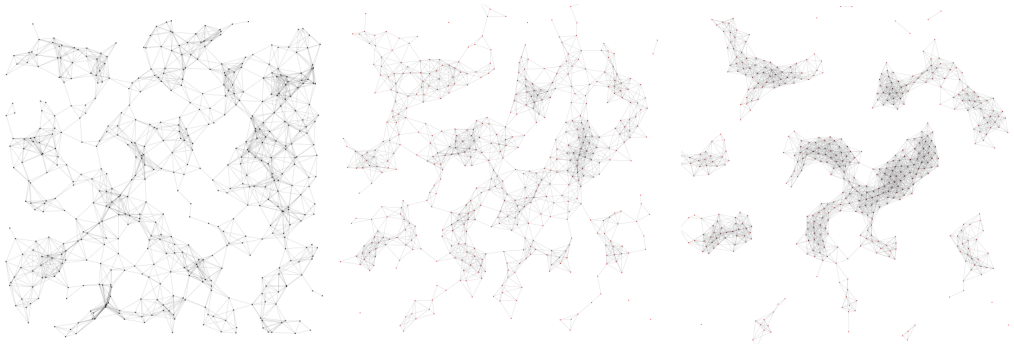
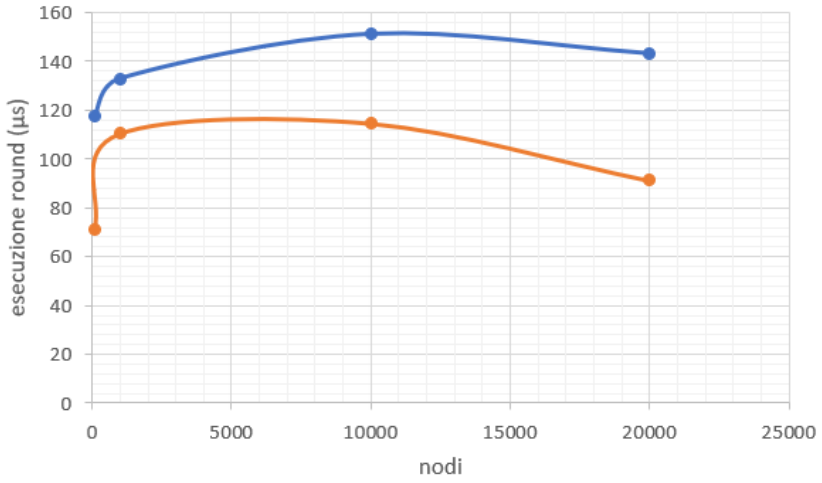


Figura 6.5: output del programma *channel* renderizzato con *StandardFXOutput*.



(a) (b) (c)



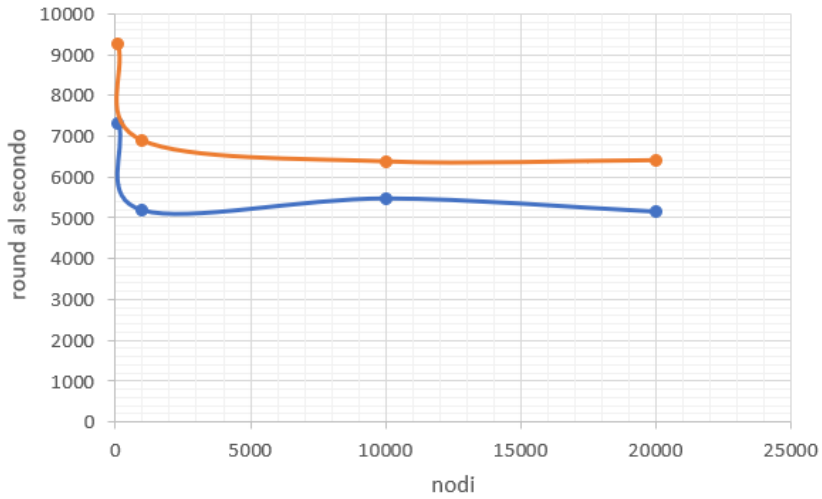


Figura 6.6: output che produce movimento a partire da un export attraverso *MetaActionProducer*.

Conclusioni

Il risultato finale del lavoro di questa tesi si è rivelato valido e soddisfa molti dei requisiti descritti in fase di analisi. Le performance ottenute dimostrano come il front-end riesca a mantenere per lo più inalterato il livello di prestazioni del simulatore in simulazioni statiche. In simulazioni con nodi in continuo movimento il front-end introduce un degrado di performance costante (che non dipende dai nodi nella scena). Risultati sperimentali mostrano come il front-end riesca a gestire, senza renderizzare il vicinato, decina di migliaia di nodi in simulazioni statiche e dinamiche. La renderizzazione del vicinato, invece, crea un degrado che dipende direttamente da quanto è densa la maglia di nodi.

Il sistema, per quanto soddisfi la maggior parte dei requisiti complessivi, non può considerarsi concluso, ma, d'altronde, l'obiettivo della tesi era quello di creare un sistema non necessariamente concluso ma piuttosto un sistema robusto, avendo la certezza che, nei lavori futuri, non ci si dovesse più preoccupare di dettagli specifici (come ad esempio performance generali, ...) e che l'implementazione dei restanti requisiti fosse possibile in modo agevole. Alcuni temi interessanti che possono essere ancora sviluppati per aumentare le funzionalità del sistema sono:

- sviluppo di un'interfaccia grafica in 3D,
- uso del parallelismo a livello del simulatore per aumentare le perfomance,
- creare al livello del simulatore modelli fisici più complessi del semplice delta movimento,
- aggiungere il supporto di nuovi modi per inizializzare la simulazione (importando i dati da OpenStreetMap, ...),
- aggiungere il supporto a **JSON** per descrivere simulazioni aggregate.

Come ultima nota personale al lavoro, non posso che ritenermi soddisfatto del sistema creato. Mi ha permesso, infatti, di scontrarmi contro una

realtà ben più complessa di quella alla quale mi ero abituato in questi anni di studi, aggiungendo difficoltà con cui non ero abituato a confrontarmi. Nonostante ciò, il lavoro è stato appagante e divertente, mi ha permesso inoltre, di conoscere nuovi linguaggi e paradigmi di programmazione che mi hanno fatto sicuramente crescere sul lato professionale. Mi piacerebbe continuare a lavorare a tale progetto e aggiungere le funzionalità che ancora mancano.

Bibliografia

[1] Jacob Beal, Danilo Pianini e Mirko Viroli. «Aggregate Programming for the Internet of Things». In: Computer 48.9 (2015), pp. 22–30. ISSN: 00189162.

[2] Roberto Casadei. «Aggregate Programming in Scala: a Core Library and Actor-Based Platform for Distributed Computational Fields». URL: `http://amslaurea.unibo.it/10341/`.

[3] Roberto Casadei e Mirko Viroli. «Towards Aggregate Programming in Scala». In: First Workshop on Programming Models and Languages for Distributed Computing on - PMLDC '16 (2016), pp. 1–7.

[4] Erik Ernst. «Family polymorphism». In: European Conference on Object-Oriented Programming. Springer. 2001, pp. 303–326.

[5] Erich Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994. ISBN: 0201633612.

[6] Atsushi Igarashi, Chieri Saito e Mirko Viroli. «Lightweight family polymorphism». In: Asian Symposium on Programming Languages and Systems. Springer. 2005, pp. 161–177.

[7] Roberto Casadei Mirko Viroli. ScaFi repository. `https://github.com/scafi/scafi`.

[8] Robert Nystrom. Game Programming Patterns. Genever Benning, 2014. ISBN: 0990582906.

[9] Martin Odersky e Tiark Rompf. «Unifying functional and object-oriented programming with Scala». In: Communications of the ACM 57.4 (2014), pp. 76–86.

[10] Martin Odersky, Lex Spoon e Bill Venners. Programming in Scala. Artima Press, 2016. ISBN: 0981531687.

[11] Danilo Pianini. Alchemist repository. `https://alchemistsimulator.github.io/`.

[12] Danilo Pianini, Sara Montagna e Mirko Viroli. «Chemical-oriented simulation of computational systems with Alchemist». In: Journal of Simulation 7.3 (2013), pp. 202–215.

[13] Chiara Varini. «Sviluppo di un simulatore per la piattaforma Scafi». URL: `http://amslaurea.unibo.it/12188/`.

[14] Mirko Viroli, Roberto Casadei e Danilo Pianini. «On execution platforms for large-scale aggregate computing». In: Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing Adjunct - UbiComp '16 (2016), pp. 1321–1326.

[15] Mirko Viroli, Roberto Casadei e Danilo Pianini. «Simulating Large-scale Aggregate MASs with Alchemist and Scala». In: 8 (2016), pp. 1495–1504.

[16] Mirko Viroli et al. «From Field-Based Coordination to Aggregate Computing». In: International Conference on Coordination Languages and Models. Springer. 2018, pp. 252–279.