ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura

Corso di Laurea in Ingegneria e Scienze Informatiche

Sviluppo di interfacce grafiche moderne in Kotlin e JavaFX: evoluzione della UI del simulatore Alchemist

Tesi di laurea in
Programmazione ad Oggetti

Relatore
Prof. Danilo Pianini

Candidato
Vuksa Mihajlovic

Correlatore
Prof. Mirko Viroli

Seconda Sessione di Laurea
Anno Accademico 2019-2020

Sommario

L'obiettivo di questa tesi consiste nell'analizzare il design delle interfacce moderne per software desktop, osservando le metodologie adottabili per raggiungere buoni livelli di *user experience* e di *usability* per quanto riguarda la funzionalità dell'interfaccia. Si vogliono considerare le miglior pratiche di sviluppo sia in un contesto generale sia nel contesto particolare delle UI grafiche, con particolare attenzione all'approccio proposto dai linguaggi di programmazione moderni, che si avvicina sempre più al misto funzionale-OOP anzichè rimanere rigidamente object-oriented.

Il software su cui si è incentrato il lavoro svolto è il simulatore Alchemist. Il contributo apportato consiste in funzionalità che permettono all'utente della UI di interagire con la simulazione attraverso mouse e tastiera. Si vuole anche dare importanza al linguaggio di programmazione utilizzato durante lo sviluppo, Kotlin, indicandone i punti di forza che lo rendono un linguaggio più espressivo e meno macchinoso rispetto al linguaggio usato precedentemente per contribuire allo sviluppo del simulatore, Java.

Indice

Introduzione

La computazione pervasiva è un ambito crescente dell'informatica che consiste nel considerare l'ambiente di esecuzione come un insieme di unità di elaborazione che soddisfano le seguenti caratteristiche:

- Si interfacciano con il mondo reale attraverso diversi tipi di sensori anzichè con utenti finali, diventando così parte indistinguibile dell'ambiente stesso;
- Interagiscono con le altre unità di elaborazione in modo costante ed implicito.

Un sistema di questo tipo differisce dal modo tradizionale in cui vengono utilizzati gli elaboratori, che invece comporta che ad un terminale corrisponda un solo utente. In un sistema pervasivo, decine o centinaia di dispositivi fanno parte di una rete che ha come input un ambiente (di cui possono anche far parte persone umane) osservato tramite sensori e ne cambia lo stato o fornisce un qualche altro tipo di output. In un sistema tradizionale, un individuo svolge un task ben definito con un obiettivo a sè stante su un unico elaboratore terminale: escluse le particolarità del task stesso, l'utente finale è l'unica sorgente di input, e questi avvengono in modo esplicito. La natura stessa di un ambiente pervasivo pone problemi di modellazione e sviluppo aggiuntivi rispetto a quelli tradizionali [3].

Nello sviluppo di un ambiente pervasivo, uno strumento che permetta di simulare il modello del sistema è di fondamentale importanza. Questa necessità ha spinto la nascita del simulatore Alchemist [6]. Oltre che per simulare ambienti pervasivi, il meta-modello (ovvero l'insieme delle entità di base di Alchemist e il loro funzionamento, attraverso il quale viene definita una simulazione) è abbastanza flessibile da permettere ad uno sviluppatore di estendere Alchemist in modo tale che si possano realizzare simulazioni anche in contesti diversi da quello pervasivo. Una implementazione di questo genere si dice *incarnazione*. Anche se inizialmente scritto inizialmente in Java, i contributi recenti sono stati aggiunti in Kotlin. Annunciato nel 2011 dalla società di sviluppo software cecoslovacca JetBrains, Kotlin è un linguaggio moderno ispirato da diversi linguaggi di programmazione nati nell'arco degli ultimi 2 decenni come Groovy, C#, Scala e Clojure che permette al programmatore di usare un approccio misto all'object-oriented e al funzionale. Grazie al fatto che Kotlin compila in Java ByteCode (oltre a Javascript e Kotlin

nativo), può essere eseguito sulla Java Virtual Machine. Questa proprietà fornisce a Kotlin due caratteristiche che sono state fondamentali per la crescita della sua popolarità.

- Permette ad un linguaggio di nascita relativamente recente di appoggiarsi alla vasta collezione di librerie esistenti;
- La completa interoperabilità con Java fa sì che il linguaggio sia facilmente adottabile in progetti sviluppati in Java.

La sintassi concisa e le features moderne lo hanno reso un linguaggio molto popolare: adottato come linguaggio ufficiale per lo sviluppo Android assieme a Java (per applicazioni, come Kotlin) e C/C++ (per librerie native attraverso l'NDK[1]), nei sondaggi annuali svolti da StackOverflow Kotlin è stato il secondo linguaggio più amato dagli sviluppatori nel 2018[2] ed il quarto nel 2019[3].

Il contributo descritto in questa tesi è stato apportato successivamente alla progettazione dell'interfaccia utente Alchemist JavaFX, nata poichè l'interfaccia precedente sviluppata in Swing era considerata complicata da usare per un utente inesperto [5]. Lo sviluppo si è incentrato sull'arricchire le funzionalità dell'interfaccia JavaFX. Le features principali implementate sono l'interazione dell'utente tramite mouse e tastiera con il mondo della simulazione e il rendering di un environment di simulazione geospaziale.

Per poter ridefinire gli *shortcut* da tastiera usati dall'utente per richiamare le funzionalità di interazione è stata anche sviluppata una UI in Kotlin, appoggiandosi alla libreria per interfacce utente *TornadoFX*. L'environment geospaziale è stato implementato attraverso la libreria *LeafletMap*, che permette di inserire una componente JavaFX contenente una mappa geografica renderizzata in un engine *JavaScript*.

Struttura della tesi. Nel capitolo 1 di questa tesi si considerano le motivazioni per cui è stato creato Alchemist e si esamina la necessità di apportare contributi alla sua interfaccia. Nel capitolo 2 si cita il contributo svolto, con una parte iniziale di analisi dello stato dell'interfaccia, una parte di progetto dove viene spiegata l'architettura dei contributi apportati e una parte finale nella quale vengono messi in risalto alcune particolarità dello sviluppo. Nel capitolo 3 si esamina il lavoro svolto e si suggeriscono eventuali migliorazioni.

[1] `https://developer.android.com/ndk/guides`
[2] `https://insights.stackoverflow.com/survey/2018/`
[3] `https://insights.stackoverflow.com/survey/2019/`

Capitolo 1

Background

1.1 Linguaggi di programmazione moderni

Nella storia dei linguaggi di programmazione si può osservare una certa tendenza nel cercare di aumentare l'espressività del codice sorgente attraverso l'integrazione di design pattern [4], ovvero soluzioni a problemi di programmazione ricorrenti, nella sintassi del linguaggio stesso. Si osservano diversi esempi di questo fenomeno:

- Rispetto ad assembly, i primi linguaggi di terza generazione astraggono il codice macchina con costrutti che permettono di aggregare e manipolare i dati in modo più naturale per il programmatore (ad esempio variabili, funzioni, struct, ...);
- Rispetto ai linguaggi imperativi, i linguaggi di programmazione ad oggetti permettono di organizzare e riusare codice attraverso incapsulamento, ereditarietà e polimorfismo.

La seconda osservazione che si vuole fare sull'evoluzione dei linguaggi è legata agli obiettivi dei linguaggi stessi. Come esempio, il linguaggio C nasce all'inizio degli anni '70 nei laboratori della Bell, a causa della necessità di un linguaggio per sviluppare software per sistemi operativi, e per questo motivo una delle features iniziali del linguaggio è la presenza dei puntatori [1] che permette un certo grado di efficienza fondamentale in questo contesto.

Si può dedurre che le features di un linguaggio sono direttamente legate alle necessità e alle tendenze di sviluppo della comunità di programmazione del periodo, e agli obiettivi per i quali il linguaggio esiste. Questa caratteristica è valida anche in un contesto moderno: i linguaggi di programmazione nati nell'ultimo periodo spesso cercano di integrare design pattern usati frequentemente al livello

[1] `https://www.bell-labs.com/usr/dmr/www/chist.html`

del linguaggio stesso e forniscono modi per risolvere problemi di sviluppo moderni. Seguono alcuni esempi che si possono trovare nei linguaggi moderni:

- Scala[2] è un linguaggio eseguibile sulla JVM che sceglie di unire la programmazione ad oggetti allo stile funzionale con features come monadi e pattern matching, adotta il modello ad attori[3] per risolvere problemi di parallelismo ed incorpora il pattern decorator (spesso usato precedentemente in Java) attraverso il mixin (feature simile ad ereditarietà multipla);

- Kotlin semplifica il *constructor overloading* di Java, implementa il delegate pattern tramite la keyword `by`[4], permette di creare DSL[5] e considera le funzioni *First-class*;

- Dart[6], linguaggio nato principalmente per sviluppare applicazioni web, mobile e desktop attraverso il framework Flutter[7], implementa le chiamate funzionali di Java (solitamente viste in classi che rispecchiano il pattern Builder) attraverso la *Cascade notation* e include il mixin come Scala. Una feature particolare della DVM (Dart Virtual Machine) chiamata *Hot reload*[8] è la possibilità di iniettare il codice in un processo Dart in esecuzione per visualizzare immediatamente i cambiamenti, nata a seguito della necessità di prototipare frequentemente lo sviluppo;

- Swift[9], linguaggio per sviluppare in ambienti Apple ispirato parzialmente a linguaggi della famiglia del C, introduce features moderne come pattern matching e protocol (simile ai mixin di Scala o Dart) garantendo allo stesso tempo compatibilità ed interoperabilità con codebase vecchie scritte in Objective C.

Spesso, queste "innovazioni" nei linguaggi di programmazione sono in realtà modelli e paradigmi già ben studiati. Ad esempio, il modello di parallelismo ad attori esiste dalla fine degli anni 70 [1], mentre i linguaggi funzionali nascono già a partire dagli anni 60, con Lisp nel '58, e successivamente negli anni 80 e 90 con Erlang nell'86 e Haskell nel '90. Nel contesto di Alchemist, Kotlin prende il posto di Java poichè l'obiettivo del linguaggio è quello di correggerne i difetti, attraverso features ispirate alla programmazione funzionale, garantendo comunque l'interoperabilità attraverso la JVM.

[2] `https://www.scala-lang.org/`
[3] `https://doc.akka.io/docs/akka/current/typed/guide/actors-intro.html`
[4] `https://kotlinlang.org/docs/reference/delegation.html`
[5] `https://kotlinlang.org/docs/reference/type-safe-builders.html`
[6] `https://dart.dev/`
[7] `https://flutter.dev/`
[8] `https://flutter.dev/docs/development/tools/hot-reload`
[9] `https://developer.apple.com/swift/`

1.2 Sistemi pervasivi e loro simulazione

Si può osservare un parallelismo tra le complessità di un sistema pervasivo e quelle di diversi modelli osservabili in natura nel concetto di organizzazione spontanea di un sistema, ovvero la proprietà di un sistema organizzarsi in un ordine che ne permette il funzionamento ad un certo scopo senza la presenza di una entità organizzatrice [7]. Una conseguenza a questo ragionamento è che la realizzazione di un sistema pervasivo si può spesso semplificare imitando certe caratteristiche ritrovabili nei modelli fisici, chimici o di simile fattispecie. Allo stesso modo, data la naturale complessità di un modello che possiede questa proprietà, durante la progettazione di un sistema pervasivo è fondamentale uno strumento di simulazione per studiare al meglio le caratteristiche dell'ambiente e denotare la complessità del sistema stesso al fine di realizzare un design più robusto e gestire i casi particolari che nascono dall'interazione delle unità di computazione indipendenti tra loro [6].

1.3 Il simulatore Alchemist

La motivazione della nascita di Alchemist è un'estensione dei due concetti descritti nella sezione precedente: è particolarmente efficace progettare i sistemi pervasivi partendo da un modello naturale di riferimento, e la simulazione di un modello naturale è necessaria affinchè la sua progettazione abbia un esito positivo. Alchemist si ispira al modello chimico, rappresentato in fig. 1.1, per ereditarne varie proprietà che lo rendono una ottima soluzione per progettare sistemi pervasivi [8]. Le entità del modello del simulatore sono dunque le seguenti:

- Una *molecola* è il nome di un certo dato;
- Una *concentrazione* è il valore di un dato, associata quindi ad una molecola;
- Un *nodo* è un contenitore di molecole e zero o più reazioni;
- Un *ambiente* è l'astrazione di uno spazio che contiene i nodi;
- Un *vicinato* è una entità composta da un nodo (detto centro) ed un insieme di nodi (detti i vicini);
- Una *regola di linking* è una funzione che associa ad ogni nodo un vicinato;
- Una *reazione* è un evento propagato al verificarsi di zero o più condizioni;
- Una *condizione* è una funzione che associa l'environment ad una coppia di valore booleano e numero;

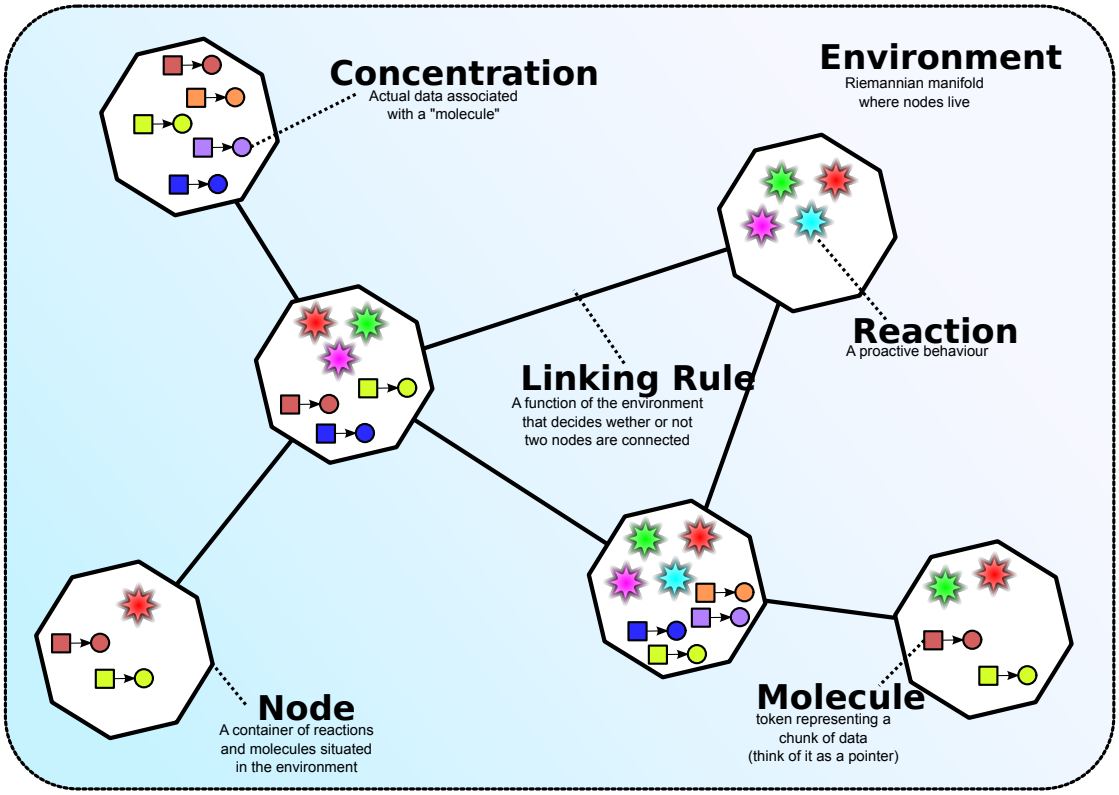

Figura 1.1: Una visualizzazione delle entità di Alchemist, ispirate al modello chimico. Dalla wiki ufficiale del progetto Alchemist.

- Una *azione* è un cambiamento nell'ambiente.

Il modello di base così descritto compone le fondamenta di ogni modello di simulazione realizzato in Alchemist. Definendo un dominio di simulazione tramite le entità sopra riportate si crea una implementazione del simulatore, detta *incarnazione*. Il livello di astrazione delle entità fondamentali di Alchemist fa sì che esso sia facilmente estendibile. Ne segue che un utente finale, se sufficientemente dotato di capacità di sviluppo software, può creare incarnazioni ad-hoc per le sue necessità. Queste incarnazioni possono poi essere eseguite, osservate e studiate tramite gli strumenti di output del simulatore quali interfaccia grafica (le entità del simulatore vengono rappresentate attraverso *effetti* visuali), logger, eccetera. Grazie al design pattern *Model View Controller*, il simulatore non è legato a nessun tipo di interfaccia utente ed è possibile sviluppare una interfaccia ed applicarla ad una simulazione senza modificare alcuna parte del progetto relativa al simulatore stesso.

1.3.1 Vecchia interfaccia Swing

Swing è un framework usato per sviluppare interfacce utente grafiche in Java. Fa parte della Java SE dal release 1.2 dell'anno 1998. L'interfaccia grafica classica di Alchemist è implementata attraverso Swing. Pur essendo completamente funzionante, presenta alcuni problemi di design. Si tratta di una interfaccia molto senplice che non fornisce informazioni su come interagire con la simulazione: ad esempio non vi è alcuna indicazione su come avviare la simulazione stessa, queste informazioni vengono invece delegate alle pagine di documentazione. Ulteriormente, non è possibile ridefinire i tasti della tastiera che corrispondono alle azioni della interfaccia. Infinte, l'apparenza dell'interfaccia stessa non è personalizzata per il simulatore Alchemist, ma piuttosto usa le configurazioni di base del framework Swing. Oltre che a non conferire ad Alchemist nessun tipo di identità dal punto di vista dell'aspetto grafico, si tratta di un'estetica datata, che detrae dalla *user experience* dell'utente finale [5]. Questi motivi hanno incitato la creazione di una interfaccia grafica nuova.

1.3.2 Nuova interfaccia JavaFX

JavaFX è una libreria di sviluppo per UI in Java, rilasciata inizialmente nel 2008. Fino alla versione 11 del *JDK* (Java Development Kit) viene distribuita nel JDK stesso, e come standalone dal *JDK 11* in poi. Ad oggi, è il supporto ufficiale Java per realizzare interfacce grafiche che riceve update più frequenti, mirati a continuare lo sviluppo della libreria e soddisfare le richieste di mercato. Nella versione iniziale 1.0, lo schema delle interfacce viene definito attraverso *JavaFX Script*, un linguaggio dichiarativo che compila in Java ByteCode, creato ad-hoc per realizzare interfacce. Con la versione 2.0 del 2011, JavaFX Script non viene più supportato e viene sostituito da *FXML*, un linguaggio di markup basato su XML. Questa caratteristica pone una netta distinzione tra JavaFX e Swing, poichè attraverso FXML è possibile creare schemi di interfacce utente senza coinvolgere codice sorgente Java, rafforzando ed incoraggiando l'utilizzo della famiglia dei design pattern simil-*MVC*. Lo stile di sviluppo classico Swing è invece composto unicamente da codice Java.

Lo sviluppo di una interfaccia per Alchemist realizzata in JavaFX inizia nel 2017 [5]. L'estetica di riferimento è quella del Material Design[10] di Google. La descrizione del layout delle componenti viene delegata a file FXML, mentre attraverso il codice Java vengono implementate le logiche di controllo delle componenti. Viene anche creato un nuovo sistema di gestione degli effetti visuali che descrivono lo stato della simulazione: l'utente può ridefinire certe caratteristiche di questi

[10] `https://material.io/design`

effetti attraverso file di configurazione *JSON*[11] e caricarli mentre una simulazione è ancora in esecuzione, cambiando così l'apparenza delle entità di Alchemist senza arrestarlo, o in alternativa specificare un file JSON al momento dell'avvio di Alchemist. È anche possibile per uno sviluppatore creare effetti completamente nuovi facendo uso di classi specifiche fornite da JavaFX. Gli effetti di base implementati sono semplici punti colorati per rappresentare i nodi e linee per rappresentare le connessioni tra i nodi. In fig. 1.2 si può osservare uno screenshot di questa interfaccia.

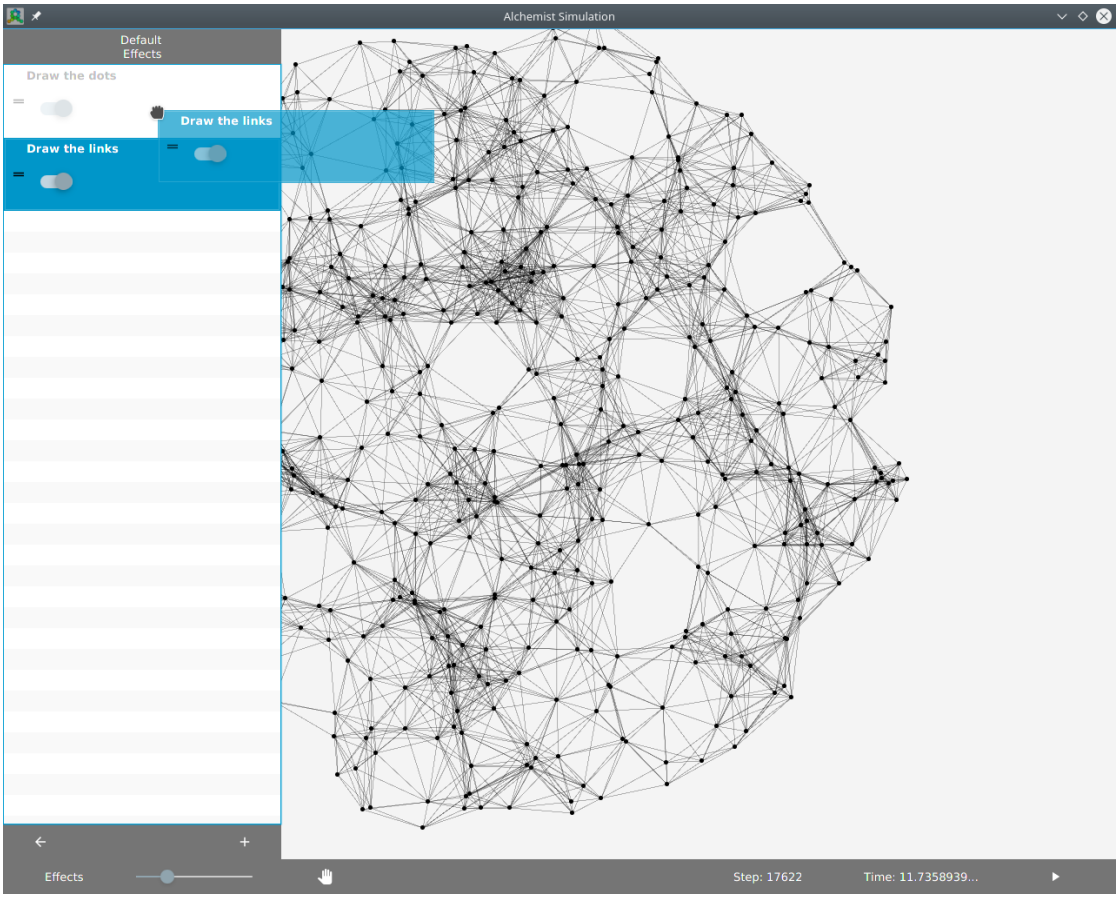

Figura 1.2: L'interfaccia Alchemist realizzata tramite JavaFX. Nella parte sinistra della finestra è presente una lista degli effetti attualmente caricati dal simulatore.

[11] `https://www.json.org/json-en.html`

1.3.3 Transizione da Java a Kotlin

A partire dal 2018, per i contributi futuri di Alchemist si è scelto di fornire la possibilità di usare Kotlin oltre che Java. Questa scelta che non ha necessitato alcun tipo di cambiamento per quanto riguarda il codice sorgente già esistente, dato che l'interoperabilità tra Java e Kotlin è garantita e completa grazie al funzionamento della *JVM*. Ulteriormente, alcune classi Java sono state convertite a Kotlin.

Tutti i contributi descritti in questo documento sono stati implementati in Kotlin. Il linguaggio punta ad essere facilmente adottabile da parte di un programmatore che già conosca Java, cercando di correggere alcuni punti deboli del linguaggio classico e fornendo strumenti aggiuntivi per ridurre la verbosità dello stile di sviluppo moderno di Java. Nei paragrafi a seguire vengono descritte solo alcune di queste *features* di Kotlin.

Sintassi

La sintassi di Kotlin è più concisa di Java; ne risulta che blocchi di codice Kotlin sono più compatti e più facilmente leggibili di codice equivalente Java. Inoltre, alcuni principi di buona programmazione sono incoraggiati dalla sintassi stessa. Seguono alcuni esempi.

- Keywords più corte, e simboli rimossi. Ad esempio, la keyword `new` non è usata e la terminazione di una riga non è indicata attraverso il carattere `;` ma viene invece omessa (rimane possibile specificarla se necessario per certi casi);

- Non è necessario scrivere il corpo di una funzione se ritorna un valore immediatamente. È anche possibile omettere il tipo di ritorno. Le seguenti due funzioni sono identiche

```
    fun powerOfTwo(n: Int) = n * n

    fun powerOfTwoWithBody(n: Int): Int {
        return n * n
    }
```

- La funzionalità di smart cast limita le operazioni necessarie per verificare il tipo di un valore

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x è di tipo String in questo scope
    }
}
```

- Il concetto *design for inheritance, otherwise prohibit it* è rafforzato dal fatto che non è possibile eseguire l'override di una classe o funzione a meno che non sia esplicitata la keyword `open`;

- L'immutabilità è incoraggiata, dato che per dichiarare un campo è necessario usare esplicitamente la keyword `var` per le variabili, oppure `val` per le costanti. In Java invece, non è necessario specificare che un campo è variabile, perchè lo è di default. Ulteriormente, le collezioni standard di kotlin sono immutabili a differenza di Java (con la possibilità di creare comunque collezioni mutabili), e la libreria standard fornisce funzioni ispirate al paradigma funzionale per manipolarle.

Null Safety

In Java, ogni variabile di tipo non primitivo può essere null e quindi potenzialmente scatenare *NullPointerException* a runtime. Questo comporta la necessità di prestare una particolare attenzione nel controllare i valori delle variabili. Con la versione 5 di Java vengono introdotte le annotazioni (puramente metadata), e diventa possibile specificare la nullabilità di una variabile attraverso annotazioni come `@Nullable` e `@NotNull`.

```
1 // il valore di foo non sarà mai null
2 final @NotNull Int foo;
3
4 // il valore di bar può essere null
5 final @Nullable Int bar;
```

L'uso di queste annotazioni risulta però molto verboso, e rimane comunque il compito tedioso di dover indicare che una variabile non può essere nulla anche se questo comportamento è il caso più comune [2]. Una variabile in Kotlin è invece *nullable* solo se esplicitamente indicato attraverso il simbolo `?`.

```
1 // il valore di foo non sarà mai null
2 val foo: Int
3
4 // il valore di bar può essere null
5 val bar: Int?
```

Nel contesto dell'interoperabilità, un valore di una classe Java che non specifica l'annotazione `@NotNull` viene considerato come nullable in Kotlin.

Lambda, Funzioni First-Class/Higher-Order e Function Types

Java non presenta la possibilità di assegnare ad un campo una funzione, restando fedele al concetto *Everything is an object*. Invece esistono i tipi *SAM* (Single Abstract Method), ovvero interfacce con un unico metodo *abstract* e opzionalmente un numero di metodi *default*, anche chiamate *functional interfaces*. Un semplice esempio di queste interfacce è il seguente.

```
@FunctionalInterface
interface ToStringFunction<S> {
    String apply(final S input);
}
```

Con Java 8 viene introdotta la funzionalità delle *lambda*, ovvero la possibilità di creare una classe anonima che implementa una SAM.

```
// una semplice lambda di 1 riga
final ToStringFunction<Integer, String> foo = n -> "the number is " + n;
foo.apply(5); // "the number is 5"

// una lambda con logica più complessa suddivisa su più righe
final ToStringFunction<String, String> bar = str -> {
    if (str.length() > 5) {
        return str.toUpperCase();
    } else {
        return str.toLowerCase();
    }
};
bar.apply("Airplane"); // "AIRPLANE"
bar.apply("Bike"); // "bike"

// un esempio analogo, senza l'utilizzo delle lambda
final ToStringFunction<Double, String> baz =
        new ToStringFunction<Double, String>() {
    @Override
    public String apply(final Double input) {
        return String.valueOf(input * -1.0);
    }
};
baz.apply(1.20); // "-1.20"
```

Lo stile idiomatico di Kotlin non fa uso delle SAM, sebbene lo permetta per garantire l'interoperabilità con Java. Invece, in Kotlin le funzioni possiedono alcune proprietà in più rispetto a Java:

- *Function Types*. Kotlin supporta la definizione di tipi di funzioni. Non è necessario fare uso di una interfaccia come `ToStringFunction` per descrivere una funzione che accetta un tipo generico e ritorna una stringa;

- *First-Class.* Le funzioni sono gestibili come ogni altra istanza di classe o primitiva, e sono quindi assegnabili a variabili o valori;
- *Higher-Order.* Le funzioni possono essere Higher-Order, ovvero possono accettare come parametri altre funzioni, oppure ritornare funzioni.

```
// Questa costante contiene una funzione che accetta una
// String e ritorna una String
val allCaps: (String) -> String = { it.toUpperCase() }
allCaps("This Will All Caps") // "THIS WILL BE ALL CAPS"

// Questa costante contiene una funzione che accetta un
// Int e ritorna un Double
val piTimes: (Int) -> Double = { Math.PI * it }
piTimes(2) // Tau

// Questa costante contiene una funzione che accetta una
// funzione da Int a Double ed un valore Int, e ritorna una String
val applyCalculation: ((Int) -> Double, Int) -> String = { function, value ->
    "The result is ${function(value)}."
}
applyCalculation(piTimes, 4) // "The result is Pi * 4."
```

Queste proprietà spesso permettono al codice Kotlin di essere molto meno verboso di Java, aumentando l'espressività del linguaggio e riducendo il numero di righe di codice necessarie.

Trailing Lambda

Se una funzione accetta una lambda function come ultimo parametro, è possibile omettere le parentesi tonde di chiamata a funzione. Questa feature, anche se puramente zucchero sintattico, è fondamentale per permettere agli sviluppatori di realizzare DSL semplici e poco verbosi.

Extension Functions

In Kotlin è possibile definire extension functions, ovvero funzioni che si possono chiamare su una certa classe sebbene la funzione non sia definita nella classe stessa. Anche in questo caso, i tipi generici conservano la loro piena funzionalità.

```
fun <T: Number> T.timesPi(): String = "${this.toDouble() * Math.PI}"

2.0.timesPi() // "Tau"
-1.timesPi() // "-Pi"
```

Una famiglia di extension functions incluse nella libreria standard di Kotlin

è quella delle *Scope Functions*[12], attraverso le quali è possibile eseguire blocchi di codice contestualizzati in base alla funzione che viene eseguita. Un esempio dell'uso della scope function `let`:

```
Person("Alice", 20, "Amsterdam").let {
    println(it)
    it.moveTo("London")
    it.incrementAge()
    println(it)
}
```

La funzione `let` permette di eseguire chiamate sul valore su cui viene invocata, e ritorna il valore dell'ultima espressione della lambda (in questo esempio, `Unit`). Grazie alla feature delle trailing lambda, l'uso di funzioni di questo genere risulta molto naturale. Senza la chiamata a `let` è invece necessario dichiarare una costante, che quindi rimane nello scope nel quale viene creata:

```
val alice = Person("Alice", 20, "Amsterdam")
println(alice)
alice.moveTo("London")
alice.incrementAge()
println(alice)
```

Properties

Mentre in Java è buona norma dichiarare i campi di una classe privati e creare proprietà pubbliche per accedervi (ovvero funzioni di *get* e *set*), Kotlin non separa i due concetti. É possibile definire il getter/setter (e la visibilità) di un campo attraverso il campo stesso:

```
class SomeClass {
    private val someInt = 0
    private var descriptor = "the value is"
    var someString: String
            get() = "$descriptor $someInt"
        set(value) {
            descriptor = value
            someInt++
        }
}

val foo = SomeClass()
println(foo.someString) // the value is 0
foo.someString = "the int is"
println(foo.someString) // the int is 1
```

[12] https://kotlinlang.org/docs/reference/scope-functions.html

Nel contesto dell'interoperabilità con Java, è possibile richiamare i getter/setter della proprietà di una classe Java (e alcune funzioni come `List.size()`) come se fossero campi.

```
val myList = java.util.ArrayList<Int>()
println(myList.size) // 0
```

Iterazione di collezioni

L'approccio di Kotlin ad iterare collezioni è diverso da Java: il modo idiomatico è nello stile funzionale attraverso funzioni come `forEach`, `map`, `associate` e `filter`.

```
val foo = listOf(
    3.427 to "E",
    -4.82 to "2",
    -1.6543 to "K",
    2.232 to "7",
    -5.326 to "2",
    -4.122 to "0"
)

foo.associate { (double, string) -> round(double).toInt() to string.isInt() }
    .filter { (key, value) -> key < 0 && value }
    .keys
    .fold(10) { a, b -> a + b }
    .let(::println) // 1
```

Non esiste il classico statement `for (int i = 0; i < list.size(); i++)`: lo statement `for` di Kotlin (usato raramente poichè l'iterazione funzionale è più concisa) è pari al `foreach` di C# o al rispettivo Java, ovvero è possibile iterare solo su una classi che implementano l'interfaccia `Iterable`.

Capitolo 2

Contributo

Tutto il contributo descritto è stato svolto su una repository GitHub, fork[1] della repository ufficiale di Alchemist[2]. L'IDE utilizzato è IntelliJ IDEA[3] di JetBrains. Durante lo sviluppo autonomo, il tool Travis CI abbinato ai vari task di controllo di Gradle ha permesso di mantenere la qualità del codice richiesta dalle specifiche della repository ufficiale.

2.1 Analisi

2.1.1 Elementi di Alchemist usati dalla UI

In questa sezione sono illustrate le interfacce di Alchemist che vengono coinvolte durante lo sviluppo dell'interfaccia grafica.

OutputMonitor Descrive un modo astratto per visualizzare la simulazione. Non è necessariamente visuale: ogni strumento di output che voglia rappresentare una simulazione eredita questa interfaccia per osservarne lo stato ad ogni step di simulazione.

Node Descrive un nodo in un environment, ed è il contenitore della concentrazione.

Environment Il contenitore dei nodi, che ne detta le interazioni. Ogni informazione riguardante i nodi è ricavabile da questa interfaccia.

[1] `https://github.com/Vuksaa/Alchemist`
[2] `https://github.com/AlchemistSimulator/Alchemist`
[3] `https://www.jetbrains.com/idea/`

Position Rappresenta la posizione di un nodo. Ogni environment dichiara il suo tipo di position. Ad esempio, un environment che rappresenta uno spazio 2D potrà specificare una posizione con valori X e Y come coordinate per le due dimensioni dell'environment stesso. Ad ogni step della simulazione, ogni nodo è associato ad una ed una sola posizione.

Wormhole Entità che fa da ponte tra l'*Environment* di Alchemist ed una GUI. Le responsabilità principali di questa entità sono le seguenti.

- Fornisce funzioni per trasformare una posizione dell'Environment in una posizione della View e viceversa;
- Permette di modificare il *viewport* (la porzione di Environment visibile all'utente attraverso la GUI) specificando un centro nuovo del viewport oppure aggiustando il livello di zoom.

2.1.2 Architettura e stato iniziale dell'interfaccia JavaFX

Lo scheletro della UI JavaFX è descritto attraverso diversi file FXML, che specificano la struttura delle varie componenti visuali. Centrale al funzionamento della GUI è il sistema di effetti che rappresentano lo stato della simulazione. Un effetto è una classe che descrive come rappresentare una certa caretteristica dell'Environment di Alchemist in un preciso istante di tempo della simulazione. La natura astratta di questa entità permette allo sviluppatore di rappresentare visualmente nodi, link, reazioni o qualsiasi delle altre entità di Alchemist legate all'Environment; è anche possibile realizzare effetti complessi che prendono in considerazione diverse caratteristiche dell'Environment anzichè una sola entità. L'implementazione dell'*OutputMonitor* dell'interfaccia JavaFX gestisce questi effetti durante la simulazione, computando ad ogni step (come da fig. 2.1) una nuova *queue* di effetti da rappresentare a video.

Il viewport della GUI JavaFX consiste di un elemento JavaFX detto *Canvas*. Questa classe espone funzioni per permettere al programmatore di disegnare sull'elemento attraverso comandi primitivi che renderizzano sagome geometriche semplici (cerchi, rettangoli, ...). Gli effetti caricati nel simulatore sono descritti attraverso comandi di questo tipo.

Il Canvas che viene usato come viewport è contenuto nella classe astratta *AbstractFXDisplay*, implementazione dell'interfaccia *OutputMonitor* di Alchemist. Questa classe gestisce gli effetti caricati e si occupa di renderizzarli ad ogni step della simulazione.

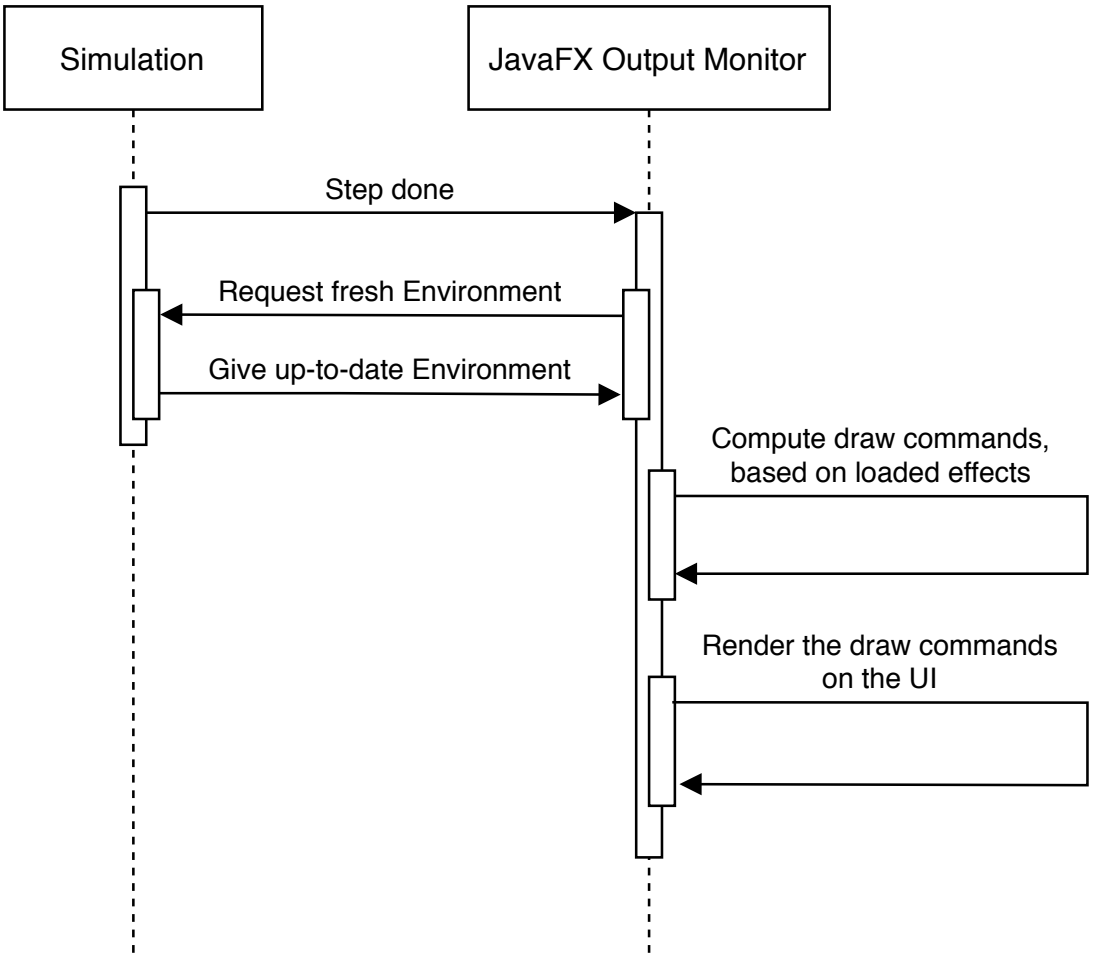

Figura 2.1: I macro-step eseguiti ad ogni step della simulazione per renderizzare gli effetti della simulazione.

2.1.3 Interazioni

Sebbene il sistema di effetti fornisse gli strumenti necessari per rappresentare le entità dell'environment, nel suo stato iniziale la UI non forniva un modo all'utente per navigare l'environment o modificarne lo stato, nonchè qualunque maniera di interagire con la simulazione ad eccezione dei comandi *pause* e *play*. Ai fini di questo documento si considera un'*interazione* una qualsiasi azione da parte dell'utente, svolta attraverso uno strumento di input come il mouse o la tastiera, che scateni un certo evento nell'environment della simulazione oppure che richieda un *feedback* visuale all'interno del viewport. Le interazioni individuate sono descritte a seguire.

Panning

Il *panning* è una operazione di traslazione del viewport. Si ricorda che ogni azione sul viewport viene effettuata tramite l'entità *Wormhole* di OutputMonitor. Ogni volta che il viewport viene aggiornato, è necessario ridisegnare gli effetti della simulazione per rendere l'interfaccia responsiva agli input dell'utente. Il modo scelto di implementare questa feature via mouse è attraverso il classico click-and-drag del puntatore. Per quanto riguarda l'input da tastiera, si è scelto di fare uso di 4 tasti che rappresentano i 4 punti cardinali per stabilire la direzione dello spostamento (esempio calzante sono i tasti delle frecce direzionali). Ad esempio, tenendo premuto il tasto legato alla direzione "nord", il viewport si sposterà verso l'alto. Combinando due direzioni adiacenti tra di loro (ad esempio, nord ed ovest), si ottiene un movimento obliquo dato dalla somma dei due movimenti di base (nord-ovest).

Zoom

Lo zoom è l'ingrandimento o la riduzione della porzione di environment visibile nel viewport. Per cambiare il livello di zoom, si è deciso di usare come strumento di input la rotellina del mouse.

Selezione di uno o più nodi

Questa interazione è necessaria per effettuare successivamente altre interazioni sui nodi stessi. Si è scelto di realizzare la selezione dei nodi attraverso il mouse, prendendo come punto di riferimento la classica selezione di elementi visuali nei sistemi operativi moderni. Come da fig. 2.2, cliccando e tenendo premuto un bottone del mouse l'utente potrà disegnare un rettangolo di selezione spostando il cursore; rilasciando il bottone, tutti i nodi che intersecano il rettangolo di selezione verranno aggiunti alla selezione. In alternativa, sarà possibile semplicemente cliccare all'interno del viewport per selezionare il nodo più vicino al cursore.

Si vuole anche dare la possibilità all'utente di modificare una selezione corrente aggiungendo elementi non presenti in essa o rimuovendo elementi selezionati (rappresentato in fig. 2.3). Queste modifiche devono avvenire nel simulatore nello stesso modo in cui si effettua una normale selezione.

Un modo per selezionare i nodi facendo uso esclusivo della tastiera non è stato progettato poichè risulterebbe troppo contorto per l'utente.

Eliminazione di nodi

L'eliminazione di un nodo risulta essere una azione molto semplice: quando l'utente fornisce il comando *elimina* (da mouse oppure da tastiera), i nodi attualmente

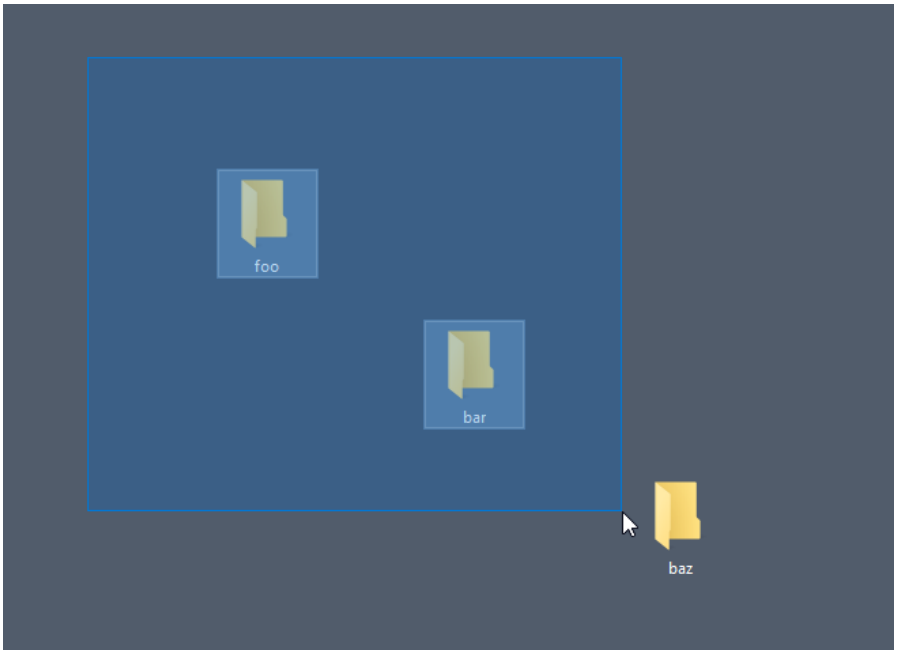

Figura 2.2: La selezione di diverse cartelle, in Windows 10. Le cartelle intersecate dal rettangolo di selezione verranno aggiunte alla selezione una volta rilasciato il tasto sinistro del mouse.

selezionati vengono eliminati dall'environment.

Spostamento di nodi

Lo spostamento è una interazione che richiede più attenzione dell'eliminazione, dato che l'utente deve anche specificare dove spostare i nodi selezionati. Per semplicità, si è deciso di gestire lo spostamento nel seguente modo:

1. L'utente seleziona alcuni nodi tramite il rettangolo di selezione oppure cliccando sui nodi;

2. Premendo un tasto apposito sulla tastiera, viene messo in coda un comando di spostamento che verrà eseguito al prossimo click del bottone sinistro del mouse;

3. L'utente clicca in un punto del viewport e viene azionato il comando di spostamento sul simulatore. Lo spostamento dei nodi è concluso, e il click del mouse torna alle sue funzionalità originali.

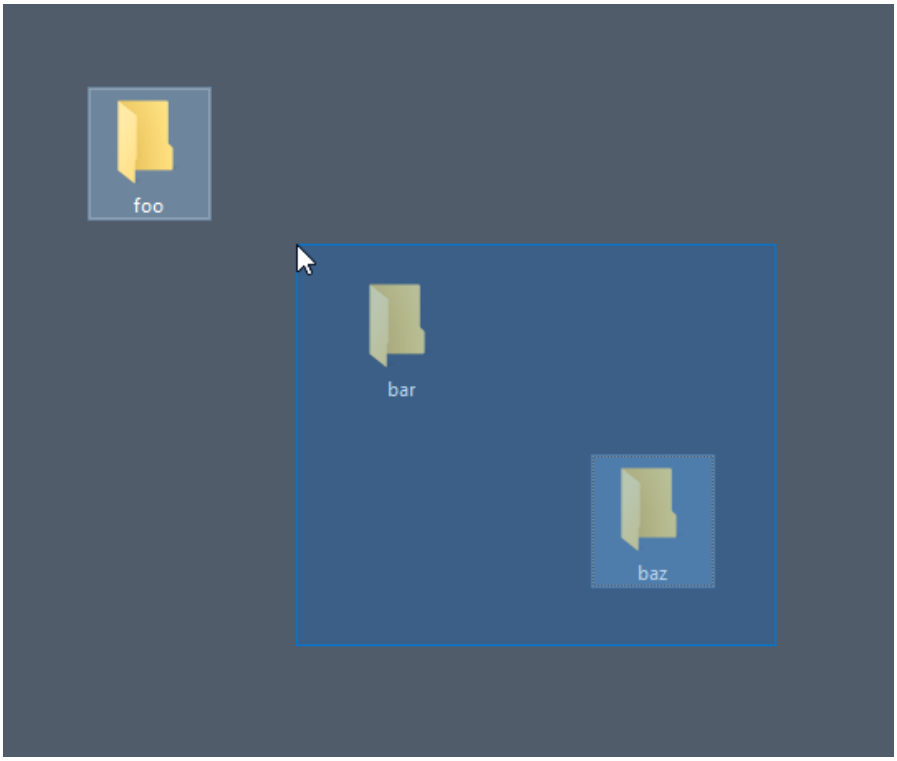

Figura 2.3: La modifica di una selezione. Le cartelle *foo* e *bar* sono attualmente selezionate. Rilasciando il tasto sinistro del mouse, la cartella *bar* verrà rimossa dalla selezione e la cartella *baz* verrà aggiunta. In Windows 10 si effettua tenendo premuto il tasto *CONTROL* e facendo una selezione.

2.1.4 Keybinds

Una feature necessaria e non presente nella interfaccia JavaFX o in quella vecchia Swing è la gestione degli *shortcuts* da tastiera. Sebbene l'utente interagisca con il simulatore attraverso la tastiera, non è attualmente possibile ridefinire i tasti associati alle interazioni.

2.1.5 Supporto per mappe

La UI deve fornire il supporto per le simulazioni di ambienti geospaziali. Il simulatore è già in grado di eseguire simulazioni geospaziali e rappresentarle in forma visuale attraverso la GUI sviluppata con Swing (fig. 2.4), appoggiandosi ai servizi di mapping di OpenStreetMap. Questa funzionalità è quindi necessaria affinchè la UI JavaFX sia al pari con la UI classica.

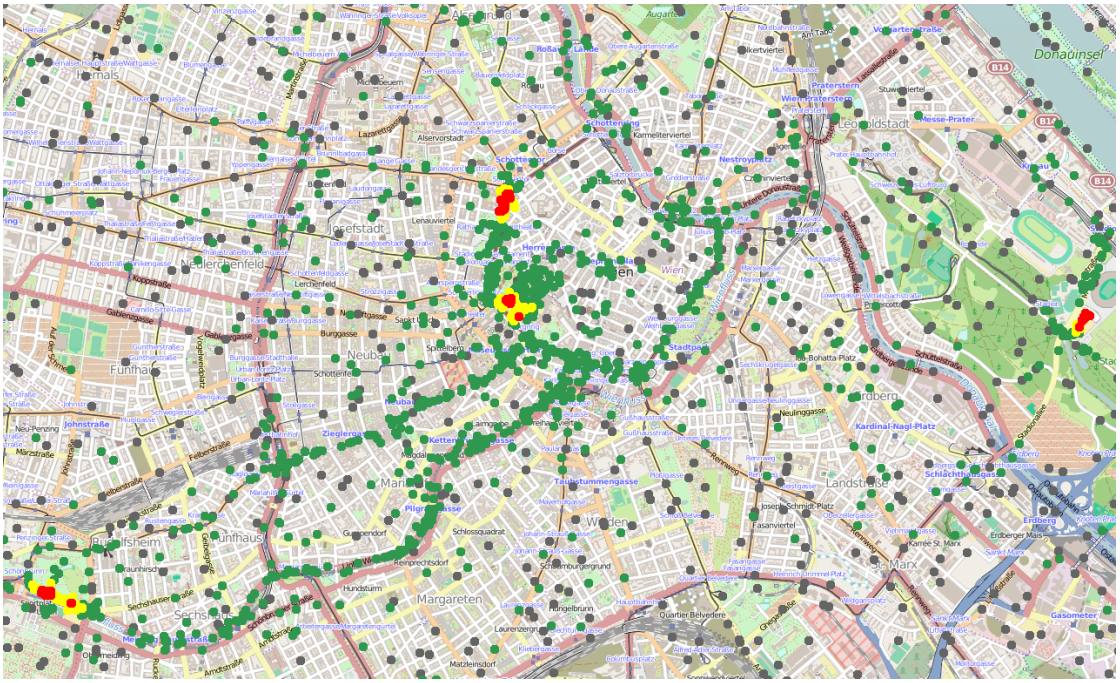

Figura 2.4: Un esempio di rendering di una simulazione in un environment geospaziale tramite la UI Swing. In questo esempio, i punti gialli e rossi indicano un addensamento di persone che impedisce il flusso stradale.

2.2 Progetto

2.2.1 Interazioni

Le interazioni vengono gestite da una classe denominata *InteractionManager*. La classe è realizzata in riferimento a JavaFX, dato che è strettamente legata all'implementazione JavaFX di *OutputMonitor*. Questa classe ha il compito di interpretare i comandi da mouse da e tastiera dell'utente ed applicare le interazioni. Si può notare che le interazioni descritte nel capitolo di analisi si possono suddividere in due categorie in base all'azione che ne risulta: una interazione può causare un effetto sulla simulazione oppure sulla view, ma non su entrambe. Ad esempio, nel caso dell'eliminazione di un nodo, l'effetto dell'interazione è esclusivamente quello di ordinare alla simulazione di eliminare il suddetto nodo. Al contrario, nel caso della selezione di un nodo, l'effetto dell'interazione è quello di denotare che il nodo è attualmente in uno stato di "selezionato", e che interazioni sulla simulazione andranno a coinvolgere il suddetto nodo; i nodi selezionati in un dato momento devono essere noti all'utente, quindi questa interazione deve renderizzare sulla view l'informazione.

Gestione delle interazioni con effetti sulla view

Per alleviare le complessità di queste interazioni ed estrarre le logiche necessarie che risultano essere molto specifiche all'interazione stessa, si è scelto di delegare i task relativi a questa categoria di interazioni a micro-classi che hanno come unico compito quello di risolvere un problema specifico relativo ad un certo tipo di interazione.

Panning Per gestire questa interazione si è creata una classe *DirectionalPanManager* (il termine "Digital" indica che gli input sono limitati ad un certo numero di direzioni, una per ogni punto cardinale) che si occupa di agire sulla wormhole a seconda delle direzioni che le vengono passate. In fig. 2.5 è possibile osservare la sequenza di azioni che caratterizzano i funzionamento di questa classe.

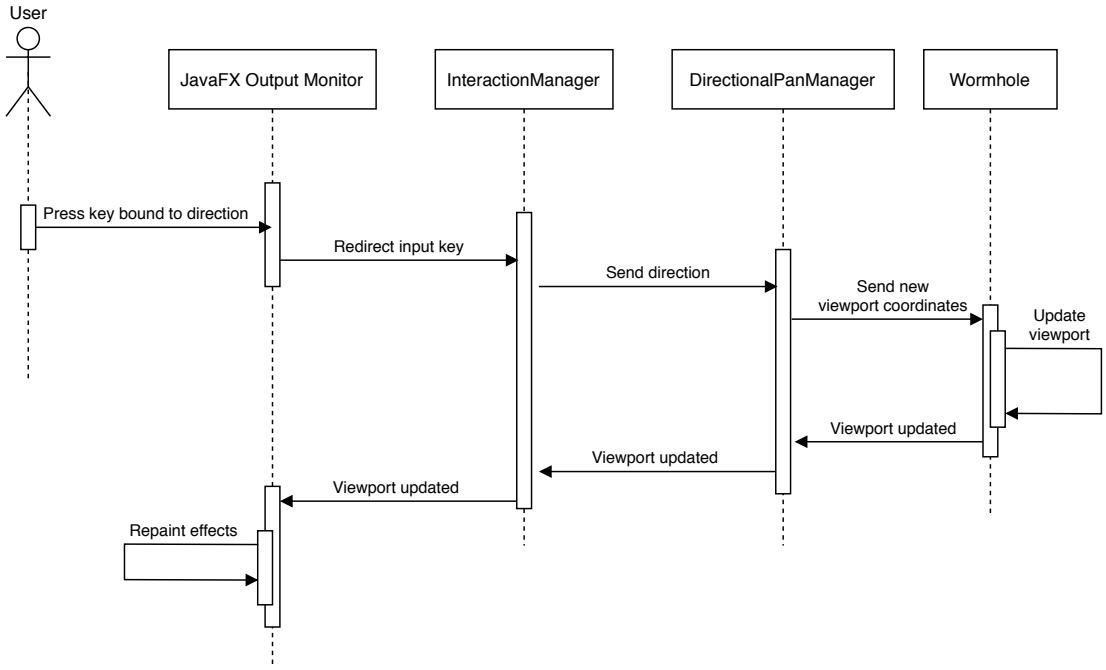

Figura 2.5: La traslazione del viewport tramite DirectionalPanManager.

Il panning svolto con il mouse è stato gestito attraverso un insieme di eventi già presenti in JavaFX. Nello specifico, JavaFX fornisce vari eventi legati alle azioni del mouse su una specifica componente visuale. Di seguito sono elencati gli eventi usati nell'implementazione del pan:

- Pressed: un bottone del mouse è stato cliccato. Indica l'inizio di una azione di pan;

- Dragged: un bottone del mouse è stato cliccato ed il cursore del mouse si è spostato. Indica che l'Environment visibile nel viewport si deve spostare in base allo spostamento del cursore;

- Released: un bottone del mouse è stato rilasciato. Indica la fine di una azione di pan;

- Exited: il cursore è uscito dal viewport. Indica la fine di una azione di pan.

Una classe di appoggio *AnalogPanHelper* si occupa di gestire gli stati del pan e di calcolare la nuova posizione del viewport. A prescindere dal modo in cui il pan sia stato eseguito, ad ogni spostamento del viewport deve corrispondere un *repaint* degli effetti.

Zoom Lo zoom consiste di una semplice azione sulla wormhole, che fa uso della classe *ZoomManager* già presente in Alchemist per calcolare il livello di zoom da effettuare.

Selezione La gestione della selezione risulta particolarmente intricata rispetto alle altre interazioni. Questa interazione si può separare in 3 parti, ciascuna delle quali richiede un modo diverso per rappresentare l'informazione all'utente:

- l'utente può selezionare i nodi, attraverso un rettangolo di selezione;

- i nodi intersecati dal rettangolo di selezione devono essere rappresentati in modo diverso rispetto agli altri nodi;

- i nodi selezionati in un certo momento devono essere distinti dagli altri nodi.

Poichè i nodi stessi si possono spostare durante l'esecuzione della simulazione, è necessario renderizzare le informazioni del secondo e del terzo punto ad ogni step di simulazione. Per questo motivo, OutputMonitor deve poter invocare il rendering di questi feedback, come in fig. 2.6.

Gestione delle interazioni con effetti sul simulatore

Le interazioni che agiscono sull'Environment risultano meno complicate da progettare, poichè non è necessario svolgere nuove operazioni sugli elementi visuali; è responsabilità di OutputMonitor, attraverso il sistema degli effetti, renderizzare i cambiamenti nell'environment. La classe Environment predispone i metodi necessari per eliminare e spostare i nodi.

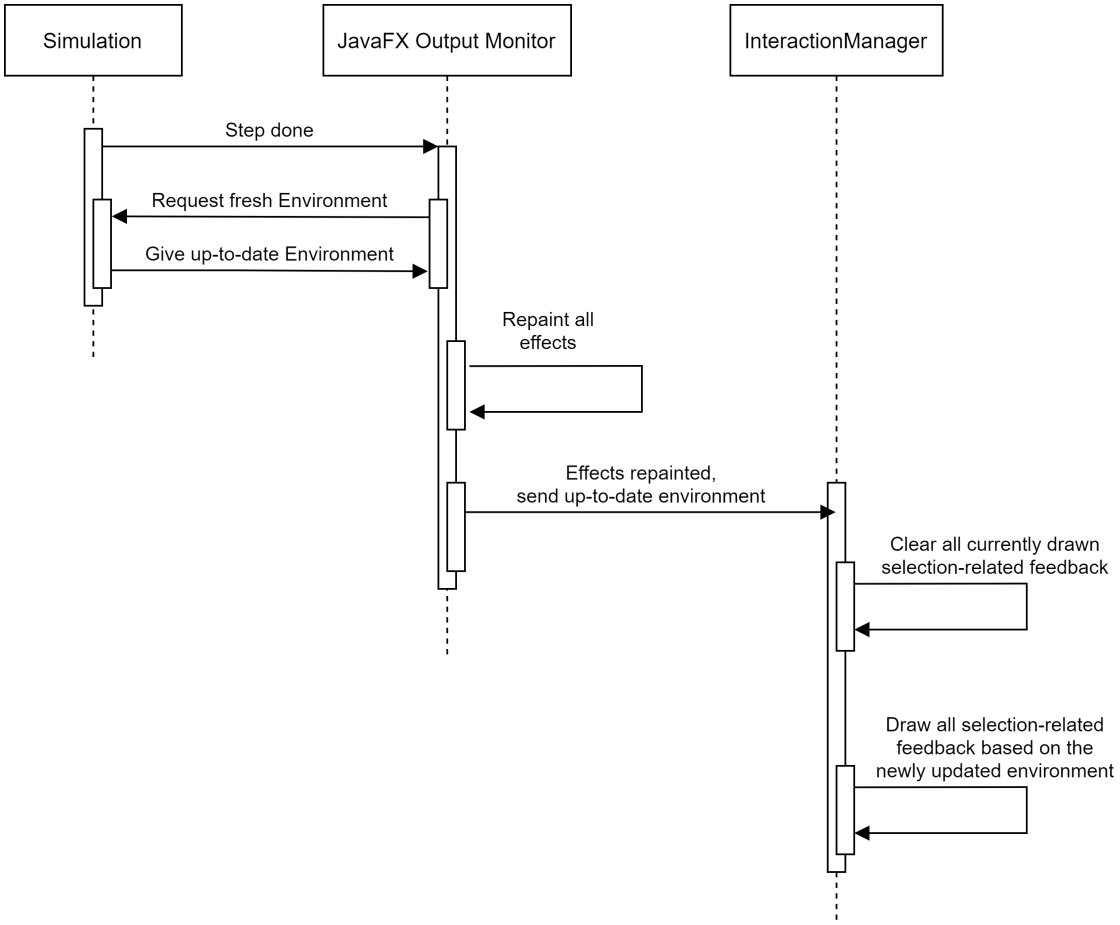

Figura 2.6: Il rendering periodico del feedback delle selezioni.

Rendering delle interazioni

Le interazioni vengono renderizzate facendo uso di un Canvas, come gli effetti. Per far sì che le interazioni siano visibili all'utente con più priorità degli effetti, il Canvas delle interazioni deve essere sovrapposto al Canvas degli effetti.

2.2.2 Input da mouse/tastiera

Mentre la classe *InteractionManager* descritta nella sezione precedente si occupa di interpretare gli input dell'utente, non gestisce la cattura degli input stessi. A questo scopo, si è scelto di creare un sistema di cattura di eventi. Non si fa uso della struttura degli eventi JavaFX poichè, sebbene sia un sistema molto completo che permette allo sviluppatore di gestire in maniera molto intricata le interazioni tra gli elementi visuali, risulta più semplice creare una soluzione più ristretta e

meno complicata che possa essere gestita secondo le necessità, rappresentata in fig. 2.7.

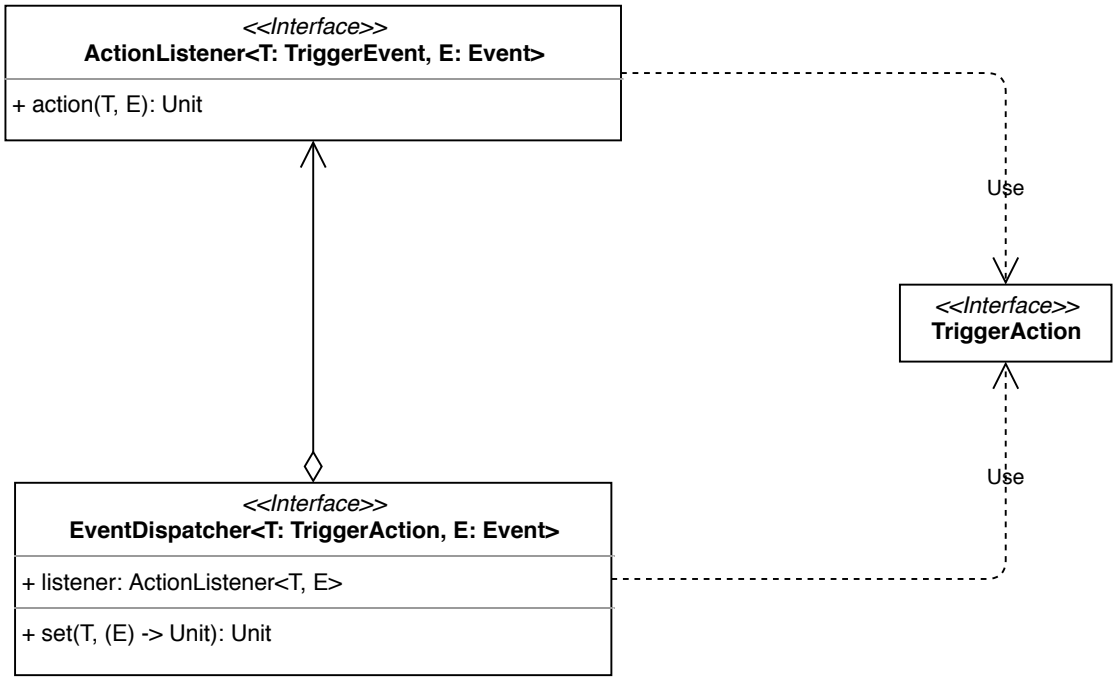

Figura 2.7: La struttura di interfacce realizzata per gestire gli eventi.

TriggerAction Interfaccia *token* che rappresenta una azione generica che possa scatenare un evento. Ad esempio, per l'implementazione degli eventi della tastiera, la pressione di un determinato tasto.

EventDispatcher Tramite la funzione *set*, permette di associare una certa *TriggerAction* ad una azione rappresentata come una funzione generica che accetta come parametro un *Event* di JavaFX. In questo modo, è possibile far uso di classi di eventi JavaFX quali MouseEvent e KeyEvent, che contengono le informazioni necessarie per interpretare i comandi dell'utente. Ogni *EventDispatcher* è associato ad un unico *ActionListener*.

ActionListener Permette di scatenare un evento tramite la funzione *action*. Passando una certa TriggerAction, l'istanza EventDispatcher associata all'ActionListener invoca una specifica azione precedentemente registrata tramite la funzione *set*.

2.2.3 Keybinds

Ad ogni azione che si può legare ad un tasto della tastiera viene associato un valore in un enumeratore denominato *ActionFromKey*, e le associazioni dei tasti sono rappresentate da una *Map<ActionFromKey, KeyCode>*. Questa mappa è contenuta in una classe denominata *Keybinds* che segue il pattern *Singleton*. La struttura di questa mappa rende facilmente integrabile questo sistema di shortcuts con codice già esistente che usi JavaFX, e quindi con il sistema di eventi descritto nella sezione precedente.

GUI L'interfaccia per ridefinire le scorciatoie (in fig. 2.8) consiste in una lista contenente le coppie di valori costituite dall'azione del simulatore e dal tasto da tastiera attualmente associato.

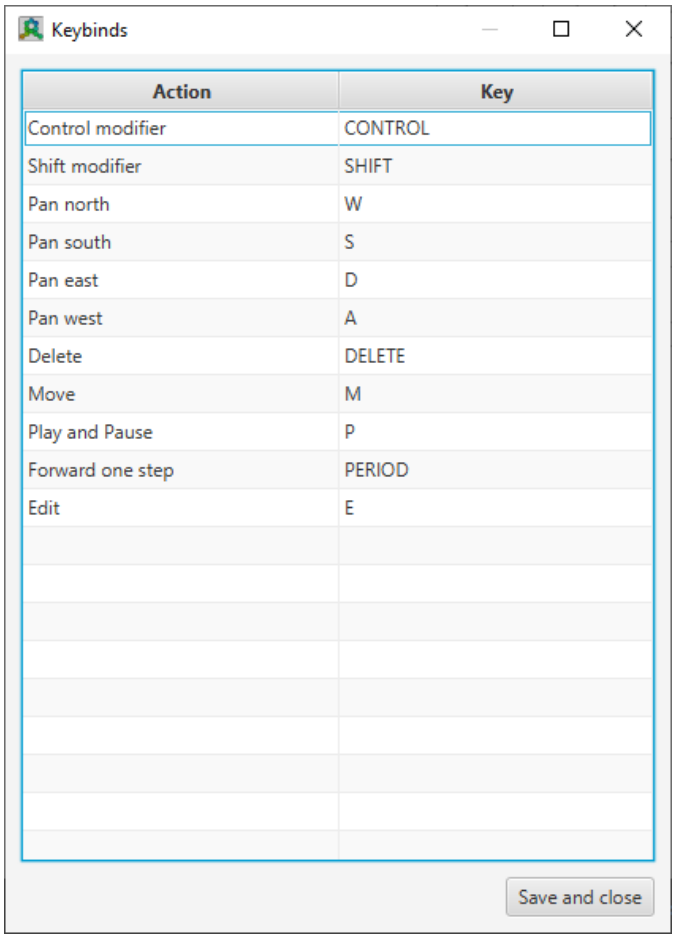

Figura 2.8: L'interfaccia per ridefinire i tasti.

Per riassociare una azione ad un tasto è sufficiente fare doppio click su un elemento. In questo modo si aprirà una finestra (in fig. 2.9) che cattura il prossimo tasto premuto, il quale verrà associato all'azione in questione. Questa UI è richiamabile dalla finestra principale di Alchemist attraverso il bottone *Keys*.

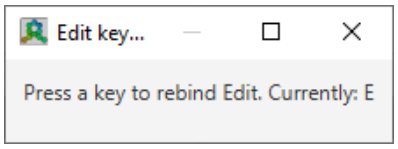

Figura 2.9: La finestra modale che permette di modificare una associazione tra azione e tasto.

La configurazione di tasti viene caricata e salvata in un file *.json* nella cartella utente dell'utente attuale. Dato che al primo avvio tale file può non esistere, nel *classpath* è presente un file di configurazione default che viene caricato se il file su disco non è disponibile.

2.2.4 OutputMonitor per environment Geospaziali

Questa feature viene implementata estendendo due interfacce di Alchemist:

- Una implementazione di *OutputMonitor* (nel caso specifico dell'interfaccia FXUI, si estende la classe astratta *AbstractFXDisplay*) usata per rappresentare posizioni di tipo geospaziale che renderizzi come sfondo una mappa;
- Una implementazione di *Wormhole2D* raffinata per posizioni di tipo geospaziale che permetta di convertire un punto geospaziale in un punto del viewport (e viceversa) e che permetta di fare operazioni di traslazione/zoom sulla mappa.

Al fine di realizzare queste implementazioni, è stato necessario modificare *AbstractFXDisplay*; questa classe dispone di un metodo *protected* che permette di disegnare sul canvas degli effetti di OutputMonitor attraverso la classe JavaFX *GraphicsContext*[4]. Le funzioni fornite da GraphicsContext sono però troppo primitive per realizzare uno sfondo complesso come una mappa geospaziale. Anzichè fare uso di GraphicsContext si fa uso della classe JavaFX *Group* [5], che permette di aggiungervi un numero di elementi JavaFX. Esponendo l'istanza di questa classe alle implementazioni, AbstractFXDisplay permette di gestire lo sfondo del viewport in maniera più generica.

[4] `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/canvas/GraphicsContext.html`

[5] `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Group.html`

2.3 Dettagli di sviluppo

2.3.1 Strumenti di controllo della qualità del codice

Alchemist fa uso del build system *Gradle*[6] per risolvere le dipendenze, generare la JavaDoc del codice e compilare il progetto. Una famiglia di task di fondamentale importanza è quella dei task di verificazione: sono stati implementati diversi plugin per permettere allo sviluppatore di seguire le linee guida della buona prassi nello stile del codice. Lo sviluppatore può eseguire questi controlli in qualsiasi momento lanciando i relativi task Gradle. I plugin di verificazione usati da Alchemist sono i seguenti:

- Checkstyle e Ktlint analizzano rispettivamente lo stile del codice Java e Kotlin e scatenano warning/errori se il codice non è conforme alle regole impostate;
- PMD e Detekt analizzano rispettivamente il codice Java e Kotlin per rilevare errori comuni di programmazione come variabili non usate, copy-paste di blocchi di codice, elevata complessità ed altro;
- Spotbugs analizza il ByteCode che viene compilato per rilevare problemi difficilmente rilevabili da un tool che analizza esclusivamente codice sorgente. Dato che opera sul ByteCode, Spotbugs riesce a controllare il codice sorgente di tutti i linguaggi della JVM.

Per automatizzare il processo di controllo del codice che viene caricato sulla repository GitHub, si fa uso del tool di Continuous Integration *TravisCI*[7]. Ad ogni *push* alla repository GitHub, TravisCI esegue un processo denominato *build* (secondo una configurazione pre-impostata nel progetto Alchemist) che verifica la compilazione del codice, la correttezza dei test JUnit (sui diversi sistemi operativi supportati ed indicati nella configurazione) ed esegue i plugin di analisi del codice. Nel caso uno di questi task non vada a buon fine, l'intero processo di build fallisce e lo sviluppatore ne viene notificato. Attraverso questo sistema è possibile garantire che le contribuzioni apportate raggiungano un certo livello di qualità del codice e di stile.

2.3.2 Uso di Optional in Kotlin

`Optional` è una classe del package `java.lang` introdotta in Java 8 che permette allo sviluppatore di rappresentare il valore di ritorno di un metodo nel caso in cui

[6] `https://gradle.org/`
[7] `https://travis-ci.org/`

l'assenza di risultato è un valore di ritorno valido[8], anzichè usare il valore `null` che introduce potenziali errori e risulta essere più verboso.

L'uso di `Optional` non è strettamente necessario in Kotlin. Questo perchè la *null safety* e le *extension functions* di Kotlin forniscono un modo conciso per gestire la nullabilità. Rimane necessario però usare gli optional ai fini dell'interoperabilità con codice Java, come spiegato di seguito.

In Kotlin

```
1 // definizione
2 fun getNullable(): String?
3
4 // chiamata
5 getNullable()?.let { println(it) }
```

L'operatore di *safe call* `?.` chiama la funzione solo se il valore su cui viene chiamato non è nullo, mentre le *scope functions* (in questo caso si usa `let`) permettono di usare il valore appena controllato, con la garanzia che non ha valore `null`.

```
1 // definizione
2 fun getOptional(): Optional<String>
3
4 // chiamata
5 provideNullable().ifPresent { println(it) }
```

Anche usando la classe `Optional`, il codice rimane sintetico e semplice da interpretare. Questa versione è meno idiomatica però, dato che fa uso di una classe

[8] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html`

Java anzichè appoggiarsi sul construtto naturale del linguaggio.

In Java

```java
// definizione
String getNullable()

// chiamata
final String potentiallyNull = getNullable();
if (potentiallyNull != null) {
    System.out.println(potentiallyNull);
}
```

Usare `null` come potenziale valore di ritorno in Java rende il codice notevolmente più verboso.

```java
// definizione
Optional<String> getOptional()

// chiamata
getOptional().ifPresent(foo -> System.out.println(foo));
```

L'uso di `Optional` riduce notevolmente le righe di codice e permette di non usare `null`, considerata buona pratica in generale in Java.

Conclusione

Sebbene il codice Kotlin non tragga alcun vantaggio da `Optional`, dato che Alchemist è un progetto multi-language (e in gran parte ancora in Java) si è scelto di usare `Optional` anzichè valori nullabili dove appropriato. Nello specifico, la classe `Keybinds` che permette di leggere gli shortcuts da tastiera attualmente assegnati espone una funzione che accetta come parametro una azione, rappresentata dalla classe `ActionFromKey`, e ritorna un `Optional<KeyCode>`, dato che non è garantito che ogni azione abbia un tasto associato.

2.3.3 Implementazione di keybinds tramite TornadoFX

La UI usata per assegnare gli shortcuts da tastiera è stata realizzata con TornadoFX[9], framework basato su JavaFX, scritto in Kotlin, che permette di realizzare interfacce grafiche in modo idiomatico.

[9] `https://tornadofx.io/`

DSL Domain Specific Language: letteralmente, un linguaggio di programmazione creato apposta per risolvere un problema specifico, legato quindi ad un certo dominio. Nel contesto di Kotlin, le proprietà del linguaggio riportate già precedentemente permettono allo sviluppatore di creare un DSL interamente Kotlin.

TornadoFX è un esempio di DSL.

Codice Kotlin TornadoFX Grazie al DLS di TornadoFX, il seguente blocco di codice genera la GUI per ridefinire gli shortcuts da tastiera descritta in sezione 2.2.3.

```
class ListKeybindsView : View() {

    val controller: KeybindController by inject()

    override val titleProperty: StringProperty
        get() = messages["title_keybinds_list"].toProperty()

    override val root = vbox(SPACING_SMALL) {
        tableview(controller.keybinds) {
            column(messages["column_action"], Keybind::actionProperty)
                .minWidth(ACTION_COLUMN_MIN_WIDTH)
            column(messages["column_key"], Keybind::keyProperty)
                .minWidth(KEY_COLUMN_MIN_WIDTH)
                .remainingWidth()
            setMinSize(TABLEVIEW_MIN_WIDTH, TABLEVIEW_MIN_HEIGHT)
            smartResize()
            bindSelected(controller.selected)
            vgrow = Priority.ALWAYS
            onDoubleClick {
                Scope().let {
                    setInScope(controller.selected, it)
                    find(EditKeybindView::class, it).openModal()
                }
            }
        }
        hbox(SPACING_SMALL) {
            region {
                hgrow = Priority.ALWAYS
            }
            button(messages["button_close"]) {
                action {
                    Keybinds.config = controller
                        .keybinds
                        .associate { it.action to it.key }
                    Keybinds.save()
                    close()
                }
            }
        }
        paddingAll = SPACING_SMALL
    }
}
```

2.3.4 Libreria LeafletMap per OutputMonitor geospaziale

Per realizzare l'OutputMonitor geospaziale si è fatto uso di LeafletMap[10], libreria inizialmente creata per essere usata nell'applicativo SportsTracker[11]. LeafletMap è un wrapper di Leaflet[12], libreria JavaScript OpenSource facilmente estendibile che permette di renderizzare mappe facilmente in un web browser o in qualsiasi ambiente che possa eseguire codice JavaScript. L'implementazione di LeafletMap consiste dunque nel caricare Leaflet in un elemento JavaFX attraverso la classe `WebEngine`[13].

Per realizzare l'implementazione di OutputMonitor e della Wormhole di Alchemist, LeafletMap è stata estesa e personalizzata con funzioni che permettano di manipolare la mappa e ricavarne le informazioni necessarie, come ad esempio convertire un punto sullo schermo in un punto descritto da latitudine e longitudine. Queste funzioni eseguono chiamate sull'oggetto JavaScript `Leaflet`, tramite l'istanza della classe `Engine` esposta da `LeafletMap`.

[10] `https://github.com/ssaring/sportstracker/tree/master/leafletmap`
[11] `https://github.com/ssaring/sportstracker`
[12] `https://leafletjs.com/`
[13] `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/web/WebEngine.html`

Capitolo 3

Conclusioni

3.1 Risultati

Grazie all'implementazione delle interazioni, l'interfaccia JavaFX risulta essere a buon punto da un punto di vista delle funzionalità. La realizzazione del sistema di supporto per gli shortcuts da tastiera comporta che un utente può personalizzare i tasti usati dal simulatore secondo le sue preferenze e visualizzarli all'interno del software stesso anzichè esclusivamente nelle pagine di documentazione di Alchemist. Attraverso l'OutputMonitor che è stato sviluppato, è possibile eseguire simulazioni di Environment geospaziali con una mappa geografica come sfondo. Inoltre, il refactoring realizzato sulla classe `AbstractFXDisplay` permette agli OutputMonitor che verranno sviluppati in futuro di gestire più liberamente lo sfondo del viewport.

3.2 Lavori futuri

Un potenziale miglioramento alla qualità dell'interfaccia è quello di aggiornare gli shortcuts da tastiera nel momento in cui vengono modificati; al momento è invece necessario riavviare l'applicazione per poter cambiare gli shortcut. Una interazione che si può considerare di migliorare è quella dello spostamento dei nodi: un modo più caratteristico delle interfacce per spostare elementi può essere ad esempio il classico drag-and-drop.

3.2.1 Performance di task periodici

Sebbene la UI sia sufficientemente performante durante l'esecuzione di una simulazione, è possibile ottimizare il modo in cui vengono renderizzati gli effetti:

- Ricevere dalla classe `Simulation` gli unici elementi che hanno cambiato stato durante lo step della simulazione, dimodochè si rielaborino solamente gli effetti di questi elementi, mentre per gli elementi inalterati si possono usare le computazioni dello step precedente;
- Il pan è una operazione di traslazione su tutti gli effetti, ma al momento vengono renderizzati nuovamente. Si potrebbe cercare una soluzione più efficiente dove l'intero canvas viene traslato, se possibile.

3.2.2 LeafletMap

L'uso di `LeafletMap` nell'implementazione dell'`OutputMonitor` geospaziale comporta un potenziale problema: ogni operazione di conversione delle posizioni è un'invocazione alla classe `WebEngine` di JavaFX, dato che `Leaflet` è una libreria JavaScript. Ogni chiamata a `WebEngine` deve essere però eseguita sul thread dell'interfaccia grafica di JavaFX. Nel caso fosse necessario convertire molte posizioni, la UI potrebbe subire notevoli rallentamenti. Nei test svolti, non si è comunque rilevato un rallentamento particolarmente problematico.

3.2.3 Ulteriori miglioramenti

Seguono alcuni possibili miglioramenti di varia natura:

- Le interazioni che comportano modifiche all'environment si possono migliorare rendendo noto a UI che una interazione è in corso.
- Lo slider che permette di modificare il numero di FPS della UI non è implementato.

Bibliografia

[1] Varol Akman. Review of actors: A model of concurrent computation in distributed systems. *AI Magazine*, 11(4):92, Dec. 1990.

[2] Patrice Chalin and Perry R. James. Non-null references by default in java: Alleviating the nullity annotation burden. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 227–247, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[3] Alois Ferscha. *Pervasive Computing*, pages 379–431. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[4] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer, 1993.

[5] Niccolò Maltoni. *Progettazione object-oriented di un'interfaccia grafica JavaFX per il simulatore Alchemist.* PhD thesis, Università di Bologna, 2017.

[6] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. *J. Simulation*, 7(3):202–215, 2013.

[7] Mirko Viroli and Matteo Casadei. Biochemical tuple spaces for self-organising coordination. In John Field and Vasco T. Vasconcelos, editors, *Coordination Models and Languages*, pages 143–162, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[8] Mirko Viroli, Matteo Casadei, Sara Montagna, and Franco Zambonelli. Spatial coordination of pervasive systems through chemical-inspired tuple spaces. In *Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2010, Budapest, Hungary, 27-28 September 2010, Workshops Proceedings*, pages 212–217. IEEE Computer Society, 2010.