Alma Mater Studiorum - Università di Bologna
Campus di Cesena

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

REVISIONE E REFACTORING DELL'INTERFACCIA UTENTE DEL SIMULATORE ALCHEMIST

Relazione finale in
PROGRAMMAZIONE AD OGGETTI

Relatore
Prof. MIRKO VIROLI
Correlatore
Prof. DANILO PIANINI

Presentata da
ELISA CASADIO

Seconda Sessione di Laurea
Anno Accademico 2015-2016

Indice

Sommario

Lo scopo di questa tesi è la revisione e il refactoring dell'interfaccia grafica del simulatore Alchemist, per offrire una migliore esperienza d'uso all'utente. L'interfaccia grafica permette la configurazione completa di una simulazione ed il suo avvio in modalità normale o batch. Per il suo sviluppo, è stata utilizzata la libreria JavaFX, che ha permesso di separare l'aspetto del programma dalla specifica comportamentale, e supporta l'Hi-DPI, che consente al simulatore di essere scalabile anche su schermi ad alta risoluzione. L'ambiente ottenuto è intuitivo, perché offre un'esperienza d'uso simile ad un IDE di sviluppo; ed è anche consistente rispetto allo stato del file system. Nel testo sono illustrati tutti i requisiti che il progetto deve soddisfare e, inoltre, sono descritte, in modo approfondito, la progettazione e l'implementazione dell'interfaccia realizzata. Il risultato ottenuto è un ambiente funzionante, che soddisfa buona parte delle specifiche richieste e che potrà essere ampliato in futuro con nuove funzionalità.

Introduzione

Lo scopo dell'elaborato di tesi è la revisione e il refactoring della già esistente interfaccia grafica del simulatore Alchemist.

L'interfaccia grafica o *graphical user interface* (GUI) [13] è l'insieme dei componenti grafici attraverso cui l'utente può interagire con il computer. Gli oggetti grafici rendono l'interfaccia di più facile utilizzo rispetto a quanto avveniva con la CLI (*command line interface*) o interfaccia a riga di comando. Una buona interfaccia grafica deve esaltare le associazioni visive, avere una grafica accattivante ed essere dotata di semplicità d'uso.

La prima interfaccia grafica viene presentata alla fine degli anni '60 da Doug Engelbart, conosciuto per essere l'inventore del mouse [17]. All'epoca l'interfaccia grafica consisteva in una interazione uomo-macchina attraverso comandi e istruzioni di codice.

L'interfaccia grafica come la conosciamo oggi è nata agli inizi degli anni '80 con il personal computer Xerox Alto e il Macintosh, la cui qualità grafica e le prestazioni sono cresciute col passare degli anni.

Attualmente l'interfaccia grafica di Alchemist disponibile risulta essere non molto user-friendly, in quanto è difficile per l'utente capire immediatamente come utilizzarla per configurare una propria simulazione. Questa difficoltà di utilizzo sta portando il simulatore ad avere problemi nell'espansione in ambito scientifico, della ricerca e della simulazione, perché si tende sempre a preferire strumenti che anche un utente non esperto possa utilizzare con facilità [8].

L'interfaccia deve permettere la configurazione completa di una simulazione ed il suo avvio in modalità normale o batch.

Viene introdotto il concetto di "progetto", che non era presente all'interno del simulatore, per offrire all'utente una esperienza d'uso simile a quella di un *integrated development environment* (IDE). Ciò porta a prevedere la possibilità di creare un nuovo progetto di simulazione dove andare a realizzare e configurare i propri file, di aprire un progetto già precedentemente creato e di salvare la configurazione di simulazione attuale.

L'obiettivo specifico del progetto è quindi quello di realizzare una nuova interfaccia grafica per il simulatore di Alchemist, dove sia possibile configurare e avviare la propria simulazione in modo semplice. Si vuole dare la possibilità di avere tutti gli strumenti necessari per realizzare e configurare la propria simulazione a portata di mano in un unico contenitore. Questo porta a realizzare un'interfaccia multi-pannello, dove nella stessa finestra compaiono aree specializzate nello svolgere uno specifico compito.

Per esprimere al meglio i requisiti è quindi stata necessaria una prima fase di analisi su ciò che si doveva ottenere, seguita da una fase di progettazione dove sono state definite le strutture dati da utilizzare e infine la fase di implementazione dell'interfaccia grafica stessa.

La tesi è suddivisa in diverse parti, che ripercorrono le fasi di studio, analisi, progettazione e implementazione che hanno portato allo sviluppo dell'interfaccia grafica.

È quindi suddivisa nel seguente modo:

- Il primo capitolo descrive il simulatore di Alchemist, concentrandosi sulla sua architettura, sulle incarnazioni disponibili e sulla realizzazione delle simulazioni.
- Nel secondo capitolo viene presentato JavaFX introducendo la sua struttura e la sua utilità per lo sviluppo di interfacce grafiche.
- Il terzo capitolo ha lo scopo di descrivere e analizzare i requisiti del progetto presentato in questa tesi. Questi requisiti definiscono il comportamento e le qualità che il sistema deve avere.
- Nel quarto capitolo sono mostrate le fasi di progettazione dell'interfaccia grafica. Inoltre viene esposto come si intende realizzare il progetto, soffermandosi sugli aspetti più importanti.
- Il quinto capitolo si sofferma sugli strumenti utilizzati per realizzare il codice dell'interfaccia.
- Il sesto capitolo mostra come si è realizzata l'interfaccia, indicando anche come sono stati realizzati nello specifico alcuni componenti.
- L'ultimo capitolo conclude il lavoro ed espone i possibili ampliamenti futuri.

Capitolo 1

Il simulatore Alchemist

Una simulazione è un fenomeno che permette di catturare gli elementi presenti nella realtà per riprodurli in un ambiente controllato, dove possono essere eseguiti per fare predizioni sul sistema.

Alchemist [15] è un simulatore scritto in Java e sviluppato come progetto interno dell'Università di Bologna. Esso permette di simulare una vasta gamma di sistemi fra cui sistemi multi-agenti [19], sistemi chimici [15], ...

Il suo codice sorgente è disponibile al pubblico, in quanto è *open source*, su un repository di GitHub. Ciò permette a chiunque sia interessato di sviluppare una sua estensione o di risolvere eventuali bug.

1.1 Architettura

L'architettura di Alchemist è modulare, dove ogni modulo ha un preciso compito e si può avvalere anche di altri moduli per il suo funzionamento. I due moduli principali sono l'*Application Programming Interface* (API) e il motore di simulazione.

Alchemist lavora sulle seguenti entità, mostrate in Figura 1.1:

- *Molecola* - è il nome simbolico di un dato;
- *Concentrazione* - è il valore associato alla molecola;
- *Azione* - modella un cambiamento nell'ambiente;
- *Condizione* - è una funzione dello stato della simulazione che restituisce un booleano e un numero;
- *Reazione* - è un insieme di condizioni, azioni e distribuzioni temporali. Viene eseguita quando la condizione è vera e la sua velocità può essere, o meno, influenzata dal valore restituito dalla condizione;

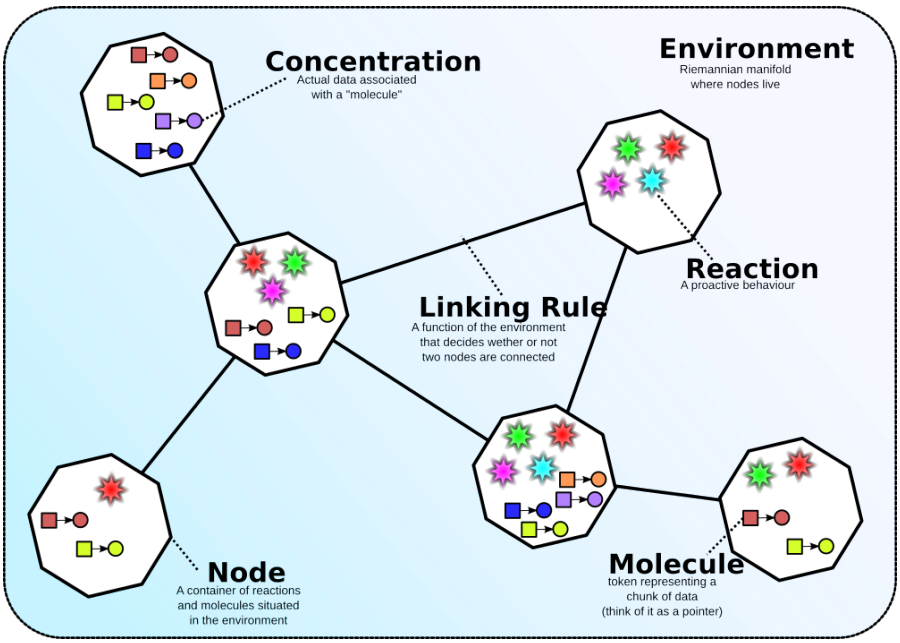


Figura 1.1: Questa rappresentazione grafica mostra un esempio delle principali astrazioni di Alchemist[1]. All'interno di un ambiente si trovano nodi, che sono collegati tra loro attraverso dei collegamenti. Ogni nodo è composto da reazioni e da molecole, ognuna delle quali ha associata una concentrazione.

- *Nodo* - è un contenitore di molecole e reazioni;
- *Vicinato* - è una entità composta da un nodo centrale e da una serie di nodi, suoi vicini;
- *Collegamento* - è una funzione che associa ogni nodo a un vicinato;
- *Ambiente* - è lo spazio e il contenitore dei nodi. Conosce la posizione dei nodi, la distanza tra un nodo e un altro e può offrire il supporto per lo spostamento dei nodi. Inoltre applica la regola di collegamento.

Tutte queste astrazioni sono generiche, infatti vengono mappate su concetti del dominio solamente in seguito all'implementazione di una incarnazione.

[1]https://alchemistsimulator.github.io/

1.2 Incarnazioni di Alchemist

Una incarnazione è una concretizzazione delle entità suddette, che fornisce gli strumenti per simulare specifiche classi di fenomeni. Essa include obbligatoriamente il concetto di concentrazione, ma può includere la definizione di ogni entità.

Attualmente, le incarnazioni disponibili per Alchemist sono:

- *Self-aware Pervasive Service Ecosystems* o SAPERE[2] [20]. Progetto che trae ispirazione dagli ecosistemi naturali, che lo rende adatto per modellare un ecosistema di calcolo distribuito. A differenza di altre proposte, SAPERE adotta le metafore naturali per governare i sistemi dinamici. Gli obiettivi principali sono di permettere a un ecosistema di avere la consapevolezza di sé, che solitamente è una caratteristica peculiare dei singoli componenti, di permettere l'autorganizzazione spaziale, l'autocomposizione e l'autogestione dei dati e dei servizi e, infine, di avere un'infrastruttura innovativa, leggera e modulare per la distribuzione dei servizi.

- *Protelis*[3] [16]. È un linguaggio funzionale basato sui campi computazionali, che fornisce una piattaforma per la programmazione aggregata. La sua accessibilità, portabilità e facilità di integrazione sono dovute all'interoperabilità con Java. Il linguaggio è particolarmente utile per sistemi di incontri a eventi di massa [16] e per la gestione di servizi di rete [16].

- *Biochemistry* [6][10]. Progetto che permette di simulare un ambiente multi-cellulare, modellando reazioni intracellulari, segnalazione intercellulare e la chemiotassi. L'incarnazione è stata usata per simulare la divisione cellulare [10].

1.3 Le simulazioni in Alchemist

In origine le simulazioni per Alchemist erano scritte in XML, un linguaggio di markup. Attualmente questo linguaggio non è più supportato ed è stato sostituito da YAML.

YAML [2] è un linguaggio usato per la serializzazione dei dati, che grazie al suo formato permette all'utente di leggerlo e scriverlo in modo semplice. È un linguaggio che funziona come linguaggio di markup, molto simile a XML ed è

[2]http://www.sapere-project.eu/
[3]http://protelis.github.io/

superset di JSON. In un file YAML, i valori accettati sono numeri, stringhe, data, ora e booleani. All'interno del codice possono essere definite variabili nel formato chiave-valore, sequenze (quelle che in Java sono chiamate array o liste), istruzioni JSON e ancore, che permettono di riferire un contenuto all'interno del documento, invece di duplicarlo. Un esempio di codice YAML è mostrato nel Codice 1.1.

Alchemist richiede un documento YAML strutturato secondo un preciso formato. In particolare conterrà:

- `incarnation` - è una chiave obbligatoria, che si aspetta una stringa, grazie alla quale restituisce la classe corretta. In questo momento sono accettati `sapere`, `protelis` e `biochemistry`, in quanto sono le uniche incarnazioni attive.

- `variables` - permette di elencare i valori delle variabili di simulazione. Ogni variabile indicata deve avere una corrispondente classe Java che implementa l'interfaccia Variable.

- `environment` - permette di caricare l'implementazione dell'ambiente, che se non specificato è uno spazio bidimensionale continuo.

- `network-model` - permette di caricare l'implementazione dei collegamenti da usare nella simulazione.

- `displacements` - elenca le posizioni dei nodi all'inizio della simulazione, il loro contenuto e il loro programma in termini di reazioni.

Per una descrizione completa della struttura del documento YAML si rimanda al tutorial del simulatore[4].

Ogni simulazione può caricare classi personalizzate, che devono essere presenti nel classpath per poter essere utilizzate.

[4]https://alchemistsimulator.github.io/pages/tutorial/simulations/

Codice 1.1: Esempio di file YAML utilizzato per una simulazione all'interno di Alchemist, dove viene creata una griglia di nodi dal punto (-5, -5) al punto (5, 5) del piano cartesiano.

```yaml
incarnation: protelis

network-model:
  type: EuclideanDistance
  parameters: [0.5]

displacements:
  - in:
    # Regular grid of nodes
    type: Grid
    # Starting from (-5, -5), ending in (5, 5), with nodes every (0.25, 0.25) and no randomness
    parameters: [-5, -5, 5, 5, 0.25, 0.25, 0, 0]
```

Capitolo 2

Introduzione a JavaFX

JavaFX [3] è una libreria per la costruzione di interfacce grafiche, che si caratterizza per la progettazione, creazione, testing e distribuzione di applicazioni che operano in modo coerente su tutte le piattaforme. Consente la personalizzazione dell'aspetto delle applicazioni realizzate attraverso strumenti di visual layout oppure con l'uso del *Cascading Style Sheets* (CSS). Un'altra funzionalità molto importante, è il supporto Hi-DPI, disponibile per Mac OS dalla versione 8 e per Linux e Windows dalla versione 9, che permette alle interfacce di adattarsi a qualsiasi risoluzione di schermo senza che la loro qualità venga compromessa.

JavaFX permette di costruire le interfacce utente in modo che la logica rimanga separata dal codice, e per fare ciò si avvale dell'uso di FXML, un linguaggio di markup dichiarativo basato su XML. Inoltre permette di incorporare pagine web all'interno delle applicazioni e di importare effetti grafici realizzati con Swing.

2.1 Architettura

La struttura interna di JavaFX è a strati (Figura 2.1), perciò tutti i componenti che si trovano nel livello più alto possono sfruttare le funzionalità dei componenti situati al di sotto di loro.

Al primo livello è situato *Scene Graph*, il quale al suo interno ha una struttura gerarchica ad albero formata da nodi, ognuno dei quali rappresenta un elemento visuale dell'interfaccia utente. Si trovano anche le *JavaFX APIs*, ossia un insieme di procedure che permettono flessibilità nella costruzione di applicazioni rich client, in quanto forniscono un insieme completo di Java API pubbliche.

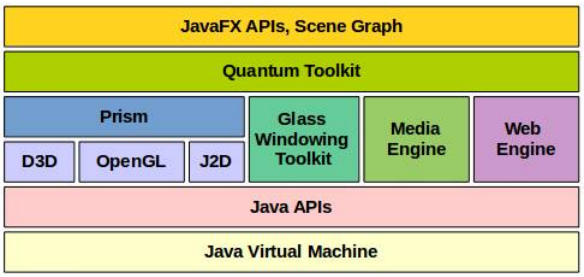


Figura 2.1: L'architettura di JavaFX [11], formata da cinque strati, dove il componente di livello superiore può attingere alle funzionalità dei componenti di livello a lui inferiori.

Al secondo e terzo livello si trova il sistema grafico, formato da *Quantum Toolkit* e *Prism*, che supporta la grafica 2D e 3D (permette di usare un renderer software, quando un sistema non può supportare l'accelerazione hardware).

Al terzo livello sono presenti *Glass Windowing Toolkit*, che gestisce le code di eventi; *Media Engine*, che offre le funzionalità per supportare file multimediali audio e video; e *Web Engine*, che supporta i linguaggi web.

2.2 Layout

La JavaFX SDK fornisce dei pannelli che supportano diversi stili di layout, utili per realizzare interfacce grafiche per le applicazioni. In realtà il layout delle applicazioni JavaFX si può suddividere in tre principali sezioni, come mostrato in Figura 2.2, le quali verranno descritte qui di seguito partendo dalla più esterna fino ad arrivare a quella più interna:

- *Stage* è il guscio esterno dell'applicazione e contiene tutto il programma;
- *Scene* è contenuto nello stage ed è il luogo dove le interfacce cambiano;
- *Pane* è il contenuto nella scena ed è il luogo dove vengono posizionati i componenti. Può avere una struttura gerarchica con un pannello radice e altri pannelli figli.

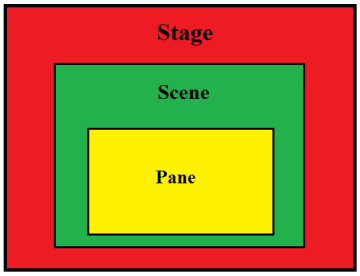
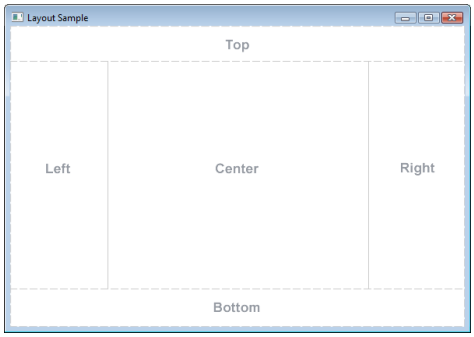


Figura 2.2: La struttura del layout in JavaFX è composta da tre sezioni, ognuna delle quali è inserita all'interno di quella di livello superiore.

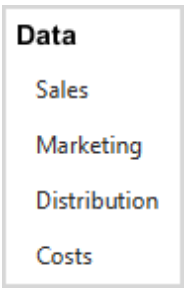
2.2.1 Pannelli

I pannelli messi a disposizione dell'utente sono:

- *BorderPane* (Figura 2.3a) - dispone i componenti in cinque regioni: in alto (*top*), a sinistra (*left*), a destra (*right*), in basso (*bottom*) e al centro (*center*). Solitamente questo tipo di layout è usato per creare quelle interfacce che prevedono una toolbar in cima, una barra di stato in basso, un pannello di navigazione sulla sinistra, informazioni aggiuntive sulla destra e un'area di lavoro al centro della finestra.
- *VBox* (Figura 2.3b) - organizza i componenti su una sola colonna in verticale, disponendoli uno sotto l'altro.
- *HBox* (Figura 2.3c) - organizza i componenti su una sola riga in orizzontale, affiancandoli l'uno all'altro.
- *StackPane* (Figura 2.3d) - colloca i componenti su un unico stack, sovrapponendo ogni nuovo componente a quello precedente.
- *GridPane* (Figura 2.3e) - crea una griglia flessibile di righe e colonne in cui disporre i componenti, occupando una sola cella o più righe o colonne.
- *AnchorPane* (Figura 2.3f) - permette di ancorare i componenti in uno dei quattro lati del pannello o al centro, consentendogli di non cambiare la posizione una volta ridimensionata la finestra.
- *FlowPane* (Figura 2.3g) - dispone i componenti in modo consecutivo, avvolgendoli in un bordo.
- *TilePane* - molto simile al FlowPane, colloca i componenti in una griglia di celle di dimensioni uniformi.



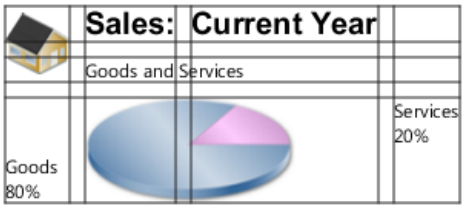
(a) BorderPane



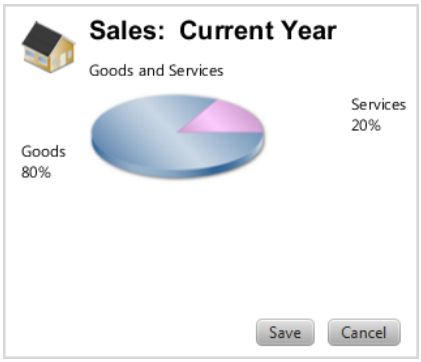
(b) VBox

(c) HBox

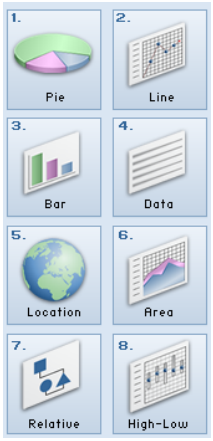
(d) StackPane



(e) GridPane

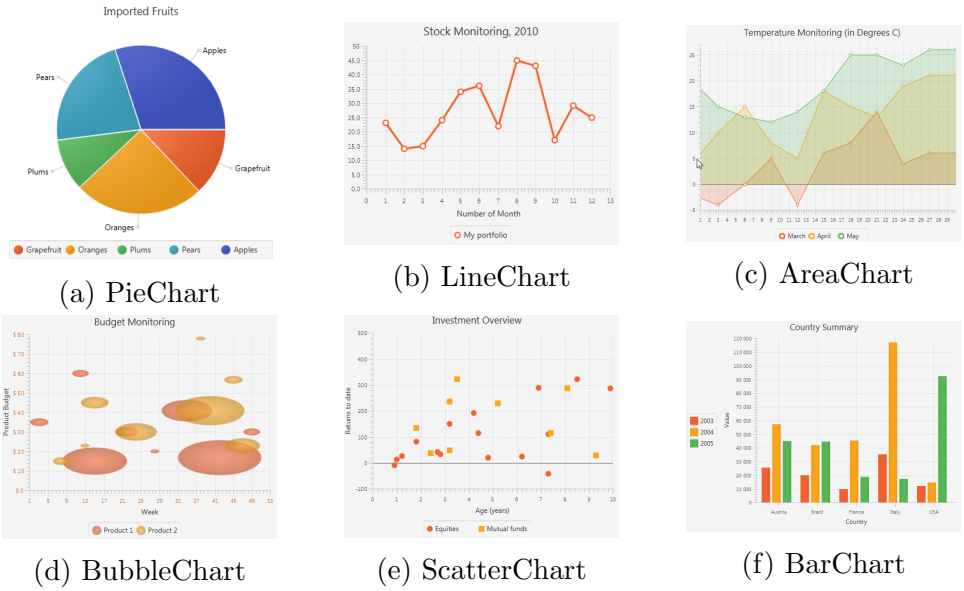


(f) AnchorPane



(g) FlowPane

Figura 2.3: Figure di esempio che mostrano i tipi di layout disponibili per i pannelli in JavaFX [11].



(a) PieChart (b) LineChart (c) AreaChart

(d) BubbleChart (e) ScatterChart (f) BarChart

Figura 2.4: JavaFX Charts messi a disposizione dalla libreria [12].

2.2.2 Componenti

In ogni pannello possono poi essere aggiunti i componenti, come ad esempio label, button, table view, tree view, ... Oltre ai componenti tradizionali, nella libreria di JavaFX sono disponibili nuovi componenti che permettono all'utente di realizzare grafici.

I *JavaFX Charts* sono disponibili in sei tipologie diverse (Figura 2.4), ognuna delle quali può essere personalizzata a piacimento del programmatore. Infatti è possibile impostare gli intervalli di valori mostrati sugli assi, i titoli degli assi cartesiani e il titolo dell'intero grafico.

2.3 Stile delle interfacce

Come detto all'inizio del capitolo, una interfaccia utente realizzata con JavaFX può essere personalizzata, applicando su di essa un foglio di stile.

Le proprietà grafiche di ogni componente possono essere settate tramite codice Java, come è sempre stato fatto anche per Swing, ma, in questo modo, non si consente di separare chiaramente la struttura dalla decorazione. Con JavaFX, invece, il compito della personalizzazione è assegnato a un foglio di stile CSS, così da avere tutto ciò in un solo punto. Ciò permette di dividere

l'aspetto dalla struttura e consente di applicare modifiche dell'ultimo minuto alla grafica, senza dover andare a toccare il codice Java.

Per poter utilizzare il foglio di stile, il file CSS deve essere applicato alla scena di interesse con un metodo Java oppure all'interno del codice FXML, se utilizzato per realizzare l'interfaccia.

Capitolo 3

Requisiti e analisi

In questo capitolo verranno analizzati i requisiti dell'interfaccia utente sviluppata, ossia ciò che l'applicazione deve fare.

L'obiettivo principale è quello di integrare l'interfaccia grafica esistente del simulatore Alchemist (vedi Figura 3.1) aggiungendo la possibilità di creare progetti, al fine di semplificare l'adozione del simulatore da parte di utenti inesperti.

3.1 Requisiti funzionali

I requisiti funzionali descrivono il comportamento che il sistema deve avere.

Qui di seguito vengono descritti tutti i requisiti richiesti, mostrati anche nel diagramma di Figura 3.2.

3.1.1 Selezione di un file di simulazione

Alchemist, per poter simulare, ha la necessità di un file dove è presente la descrizione del modello.

Il file suddetto deve perciò poter essere selezionato dalla cartella in cui si trova, così che Alchemist possa leggerla, caricarla e infine eseguirla.

L'interfaccia deve permettere all'utente di selezionare il file YAML contenente la simulazione.

3.1.2 Selezione di un file degli effetti

Il simulatore Alchemist permette di associare a ogni simulazione uno stack di effetti grafici, in modo da offrire un riscontro grafico, che mostra ciò che sta succedendo durante la simulazione. Questi effetti possono anche essere creati direttamente durante l'uso del simulatore.

Figura 3.1: Immagine che mostra l'interfaccia grafica del simulatore Alchemist utilizzata fino a questo momento. La sezione mostrata è quella in cui si può eseguire una simulazione.

L'interfaccia deve permettere all'utente di selezionare il file contenente gli effetti ed applicarlo alla simulazione.

3.1.3 Creazione di nuovi file

Un file di simulazione o degli effetti deve poter essere creato direttamente all'interno dell'interfaccia usando l'editor preferito.

Ciò rende il processo di scrittura della simulazione più veloce, immediato e automatizzato, in quanto l'utente non dovrà più andare a cercare un editor di testo e salvare il file, con il formato appropriato, in una specifica cartella.

3.1.4 Selezione di una cartella di output

Una volta avviata la simulazione, Alchemist può resistuire, su richiesta dell'utente, un file di output dove stampa i valori ottenuti. Tipicamente questa operazione è svolta con la CLI.

Nella nuova interfaccia, invece, si vuole dare la possibilità all'utente di scegliere se avere o meno in output i risultati. Se l'utente scegliesse di averli in output, allora deve essergli permesso di selezionare la cartella, all'interno del progetto, dove andare a salvare i file di output. Inoltre, sempre se l'utente ha scelto questa possibilità, può decidere il nome del file di output e il tempo di

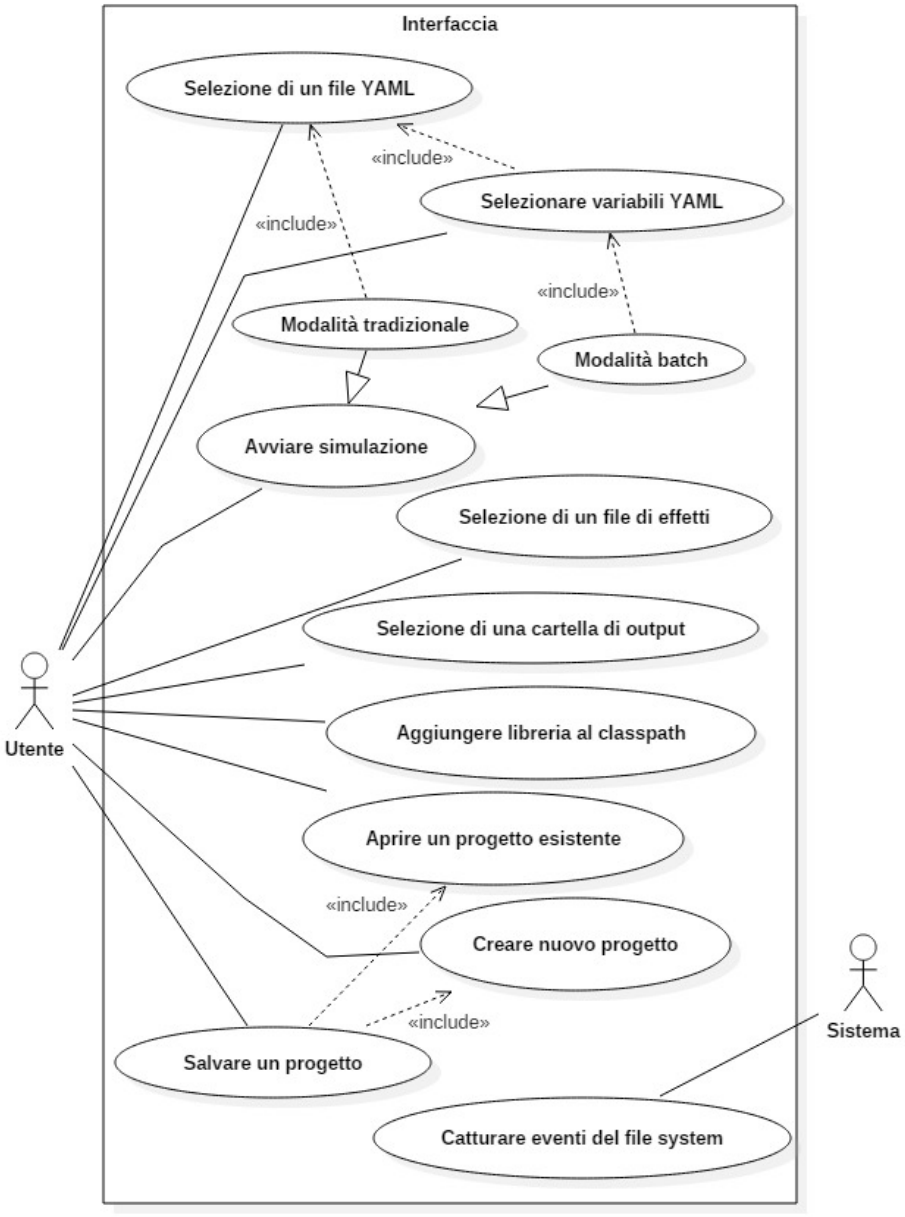


Figura 3.2: Casi d'uso dell'interfaccia grafica

campionamento, cioè ogni quanto tempo vengono stampati i risultati ottenuti all'interno del file di output.

3.1.5 Selezione delle variabili YAML e avvio modalità batch

Alchemist permette di lanciare le simulazione in modalità normale con una propria interfaccia grafica oppure in modalità batch, ossia una modalità in cui sono eseguite tante simulazioni in cui vengono combinate le variabili passate in input.

Si è voluto, quindi, dare la possibilità all'utente di decidere se attivare o meno la sezione che deve mostrare le variabili disponibili nel file di simulazione selezionato. Nel caso in cui l'utente decida di attivare questa sezione, deve poter selezionare le variabili che gli interessano e indicare il numero di thread da usare durante la simulazione. Sempre se la sezione è attiva, si dà la possibilità all'utente di avviare la simulazione in modalità batch con uno specifico pulsante.

3.1.6 Gestione del classpath

Per i più esperti, esiste anche la possibilità di aggiungere le proprie librerie, file o cartelle al classpath di Alchemist, che poi potranno essere utilizzate durante l'esecuzione delle simulazioni.

Questa azione può essere svolta direttamente dentro la vecchia interfaccia, aggiungendo i JAR di interesse, oppure può essere svolta direttamente dall'utente a mano, da linea di comando.

Per rendere tutto questo lavoro più semplice e chiaro, si vuole permettere all'utente di selezionare il file o la cartella che desidera inserire o rimuovere dal classpath. È poi compito del sistema gestire l'aggiunta o la rimozione di tale file o cartella.

Nel caso in cui si aggiunga una libreria, questa dovrà essere posizionata come prima occorrenza del classpath. Questo è importante per l'utente nel caso in cui voglia aggiungere una classe dove sono presenti delle modifiche da lui apportate, in quanto il caricamento delle classi in Java avviene consultando il classpath in modo ordinato, di conseguenza, in caso di classi omonime, verrà caricata la prima classe incontrata.

3.1.7 Avviare una simulazione

Una volta configurata la simulazione deve poter essere possibile lanciarla.

Questa azione deve essere necessariamente presente all'interno della nuova interfaccia grafica, anche perché si tratta del cuore del programma.

3.1.8 Creazione, apertura e salvataggio di un progetto

In Alchemist, dopo aver impostato la simulazione da avviare, gli effetti, il file di output e il classpath, dopo aver avviato la simulazione e dopo aver chiuso il simulatore, vengono perse tutte le impostazioni applicate.

Per evitare che ciò avvenga, si desidera permettere all'utente di salvare la configurazione della simulazione, creando di fatto il concetto di "progetto".

Questa decisione porta alla possibilità di poter anche selezionare un progetto salvato per poterlo riaprire all'interno dell'interfaccia.

Oltre alle due azioni precedentemente descritte, si vuole anche dare la possibilità all'utente di creare un nuovo progetto, dove potrà andare ad inserire al suo interno i file di simulazione, i file degli effetti e le librerie, e dove potrà salvare i file di output.

3.1.9 Template predefiniti

La creazione di un nuovo progetto deve anche dare la possibilità all'utente di scegliere tra creare un progetto vuoto, oppure creare un progetto con all'interno già dei file di simulazione disponibili. Nel secondo caso, si dà la possibilità agli utenti, che non hanno mai utilizzato Alchemist, di impratichirsi prima di procedere con una propria simulazione.

3.1.10 Gestione degli eventi del file system

Dato che con la nuova interfaccia si vanno a utilizzare file e cartelle, che durante la simulazione potrebbero venire modificate, si vuole poter gestire gli eventi che si verificano all'interno della cartella del progetto.

È necessario catturare tutti gli eventi che si verificano all'interno, come ad esempio la creazione, la modifica o la cancellazione di un nuovo file o cartella.

L'aspetto dell'interfaccia dovrà poi essere consistente con lo stato del file system.

3.2 Requisiti non funzionali

I requisiti non funzionali descrivono le proprietà non comportamentali che il sistema deve possedere, come la sicurezza, la performance, ...

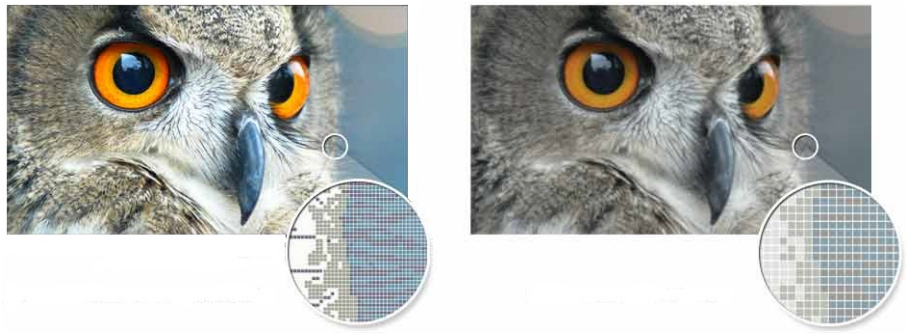


Figura 3.3: Immagine[1] che mostra la differenza di densità di pixel tra uno schermo 4K (a sinistra) e uno 1080p (a destra). Come si può notare la risoluzione dell'immagine è migliore nella prima rispetto alla seconda.

3.2.1 Supporto Hi-DPI

Recentemente sono stati adottati monitor con alta risoluzione.

Gli schermi 4K [5], detti anche *Ultra HD*, sono schermi con un'alta risoluzione, in quanto hanno a disposizione un elevato numero di pixel. La differenza tra questo tipo di schermi e quelli HD tradizionali è mostrata nella Figura 3.3.

Eseguire applicazioni su questo tipo di schermi, in moltissime occasioni, causa problemi di visualizzazione. Questo si verifica perché l'interfaccia grafica delle applicazioni è stata sviluppata inserendo valori assoluti, cioè in pixel. Data l'alta concentrazione di pixel in uno schermo 4K, se si indica la dimensione della GUI in pixel, si ottiene un'interfaccia grafica molto piccola e soprattutto quasi totalmente inutilizzabile dall'utente.

Per l'interfaccia grafica da realizzare, si vuole fare in modo che possa essere correttamente visualizzata da qualsiasi tipo di schermo, cioè che supporti l'Hi-DPI. Per fare questo si devono quindi impostare grandezze relative e sfruttare al meglio le funzionalità offerte da JavaFX.

[1] http://www.aiseesoft.com/resource/what-is-4k-resolution.html

Capitolo 4

Architettura e Progettazione

Separare l'aspetto dalla specifica comportamentale dell'interfaccia grafica è una buona pratica da seguire per ottenere un programma ben organizzato e ben strutturato.

La divisione di questi due aspetti permette di sviluppare un programma senza doversi preoccupare del suo aspetto esteriore.

In questo capitolo vengono quindi descritte e discusse le scelte progettuali e di design, che hanno portato all'implementazione dell'interfaccia utente.

4.1 Progettazione dell'interfaccia grafica

L'interfaccia grafica del simulatore deve essere progettata in modo che sia di facile utilizzo per l'utente, ma che consenta anche di svolgere funzionalità avanzate in modo semplice e intuitivo.

Deve essere disposta in modo che tutti i componenti siano ben visibili in un'unica finestra, così che l'utente possa avere tutto ciò che gli serve per svolgere una simulazione a portata di mano. Ciò significa che nella nuova interfaccia grafica di Alchemist non può essere presente una sezione a tab, come nell'interfaccia precedente.

La progettazione dell'interfaccia grafica del simulatore Alchemist ha richiesto una serie di fasi in cui sono stati realizzati schizzi del risultato che si voleva ottenere.

4.1.1 Fase 1: disegno dell'interfaccia

In questa prima fase è stata disegnata un'interfaccia, mostrata in Figura 4.1, che risponde ai requisiti di selezione di una simulazione, di selezione di un effetto grafico e di aggiunta di una libreria al classpath.

Si è quindi prevista la presenza di una barra del menu in alto, una colonna a sinistra, un corpo sulla destra e una barra di pulsanti in basso.

Il menu in alto, contenente la sola voce "File", permette all'utente di scegliere diverse azioni, come aprire una simulazione e salvare una simulazione.

Nella colonna a sinistra si deve mostrare tutto l'albero di cartelle e file presenti all'interno del file system del computer utilizzato, da cui si possono selezionare i file o le cartelle da utilizzare durante la simulazione.

Nel corpo centrale sono previste tre sezioni:

- *Sezione per selezionare il file di simulazione.* In questa sezione si permette all'utente di impostare il file, selezionato nella colonna a sinistra, come file di simulazione oppure, nel caso non lo si volesse più, si permette di deselezionarlo. Inoltre si permette all'utente di modificare il file appena selezionato, aprendolo in un editor di testo di sistema.

- *Sezione per selezionare il file degli effetti.* Come la precedente sezione, questa deve permettere all'utente di impostare il file selezionato a sinistra, come file degli effetti, di deselezionare il file oppure di modificare il file selezionato all'interno di un editor di testo.

- *Sezione per aggiungere librerie al classpath.* Quest'ultima sezione deve permettere all'utente di aggiungere al classpath i file o le cartelle selezionate a sinistra o, nel caso, rimuoverle dal classpath.

Infine, nella barra in basso sono disponibili due pulsanti, dove uno fa partire l'esecuzione della simulazione, mentre l'altro permette di selezionare il file di output.

4.1.2 Fase 2: aggiunta di sezioni opzionali

Nella seconda fase di progettazione dell'interfaccia è stato realizzato uno schizzo, mostrato in Figura 4.2, che risulta molto simile a quello precedente, ma a cui sono state aggiunte diverse sezioni e sono stati affinati alcuni dettagli.

Nel corpo centrale, infatti, sono state aggiunte altre tre sezioni:

- *Sezione per selezionare la radice del progetto.* Nella sezione si dà il permesso di impostare una cartella del file system, selezionata nella colonna a sinistra, come cartella radice del progetto.

- *Sezione opzionale per selezionare il file di output.* Questa sezione dà la possibilità all'utente di scegliere se avere o meno un file di output.

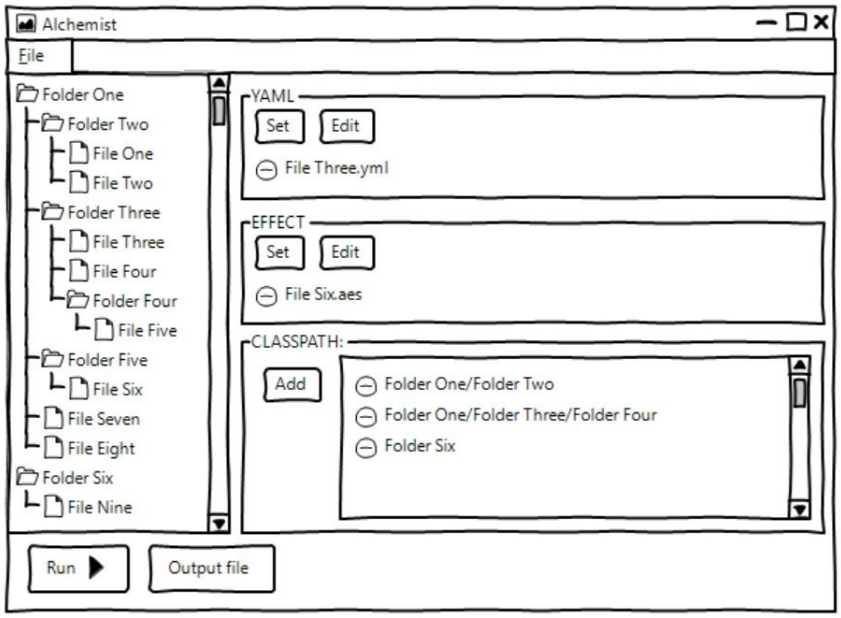


Figura 4.1: Prima bozza dell'interfaccia grafica da realizzare, dove a sinistra compare l'albero del file system e a destra sono presenti una sezione per selezionare il file di simulazione, una sezione per selezionare il file degli effetti e una sezione per aggiungere le librerie al classpath. In fondo sono disponibili due pulsanti per avviare la simulazione e per selezionare il file di output.

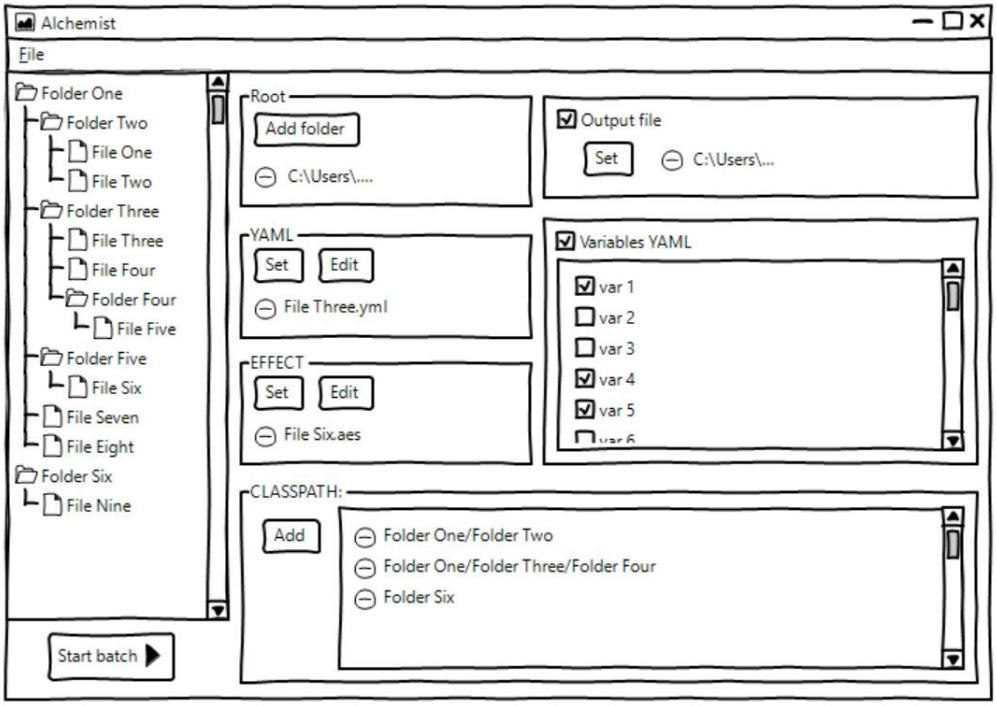


Figura 4.2: Seconda bozza dell'interfaccia grafica da realizzare, dove a sinistra si trova la colonna con l'albero del file system, mentre a destra, rispetto alla prima bozza, sono state aggiunte una sezione per selezionare la cartella radice del progetto, una sezione per settare la cartella di output e una sezione per selezionare le variabili YAML.

Nel caso in cui si scegliesse di attivare tale sezione, essa permette di impostare il file di output, selezionandolo sempre a sinistra. L'introduzione di questa sezione ha segnato, quindi, la rimozione del pulsante, che nella versione precedente dell'interfaccia era situato nella barra in basso.

- *Sezione opzionale per selezionare le variabili presenti nel file YAML della simulazione.* Come per la sezione precedentemente descritta, si dà la possibilità di scegliere se attivarla o meno. Nel caso in cui venisse attivata, si permettere di selezionare le variabili di interesse, per poterle così utilizzate durante la simulazione.

4.1.3 Fase 3: affinamento dell'interfaccia

Nella terza fase, l'interfaccia grafica è stata affinata, in modo che potesse rispondere sempre di più ai requisiti.

Come mostrato dalla Figura 4.3, in questa fase è stato deciso di sostituire la barra del menu con una barra di pulsanti che permettessero di creare un nuovo progetto, aprire un progetto esistenze, salvare il progetto attuale e importare un progetto. La scelta è stata preferita, rispetto a una tradizionale barra del menu, in quanto permette all'utente di avere a portata di mano tutta una serie di azioni.

È anche data la possibilità all'utente di poter creare direttamente il file di simulazione o il file degli effetti, grazie all'introduzione di un nuovo pulsante presente nelle relative sezioni. Nel caso in cui si scegliesse questa opzione, si chiede all'utente di inserire il nome del nuovo file da creare, che sarà poi aperto con l'editor di testo di sistema.

Per rendere il tutto, esteticamente più bello e più intuibile, i due checkbox, che permettevano all'utente di scegliere se attivare o meno la sezione di output e quella delle variabili YAML, sono sostituiti da due switch, i quali attivano o disattivano le due sezioni.

Infine, sono stati definiti meglio i compiti dei pulsanti "Run" e "Batch", in modo che il primo avvii la simulazione con l'interfaccia grafica, mentre il secondo la avvii in modalità batch. Il secondo pulsante, inoltre, può essere usato solamente, nel caso in cui, la sezione delle variabili YAML è attiva.

4.1.4 Fase 4: ultime correzioni

Nell'ultima fase di progettazione dell'interfaccia, mostrata in Figura 4.4, sono stati applicati gli ultimi ritocchi per renderla più funzionale.

Nella barra del menu sono stati eliminati due tasti, che per il momento non avevano una loro utilità.

Inoltre, nella colonna a sinistra è stata tolta la possibilità di vedere l'intero albero del file system. In questa colonna, infatti, è stato deciso che viene mostrato l'albero di cartelle e file che formano la cartella del progetto.

Proprio per via di questo cambiamento di utilizzo della colonna a sinistra, è stata tolta la sezione dove si impostava la cartella radice, in quanto questa viene definita in automatico all'apertura del progetto.

Sono, poi, stati aggiunti alcuni campi mancanti, necessari però per l'esecuzione della simulazione. I campi sono:

- campo per settare il periodo di tempo trascorso il quale la simulazione termina;

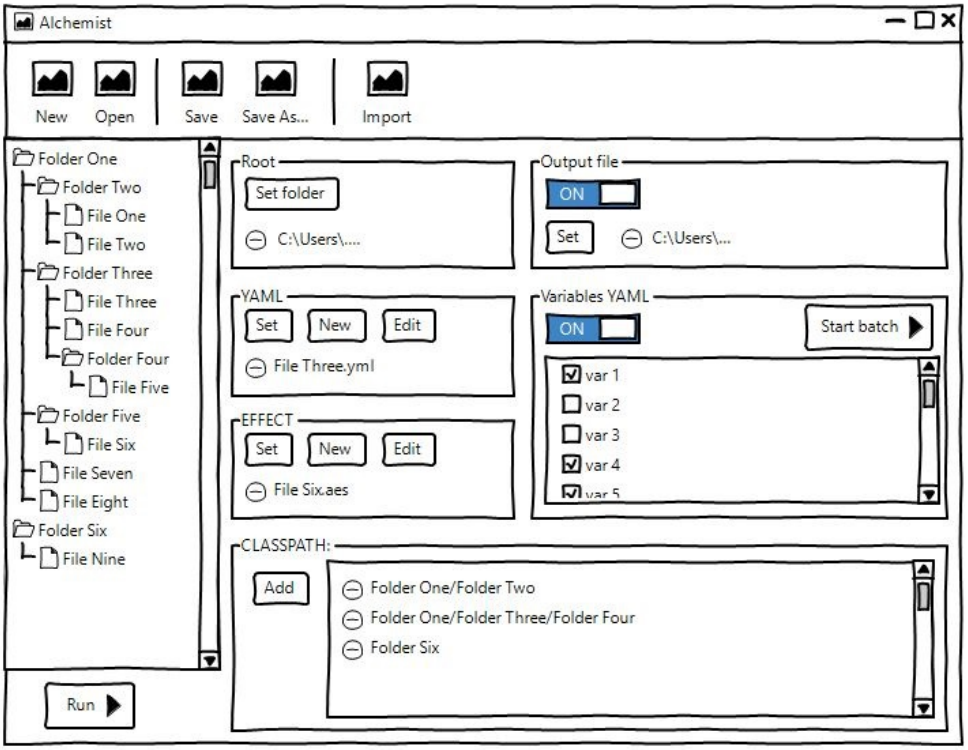


Figura 4.3: Terza bozza dell'interfaccia grafica da realizzare, dove la barra del menu è stata sostituita da una serie di pulsanti che svolgono le stesse funzioni e le due sezioni opzionali (output file e variables YAML) possono essere attivate da uno switch.

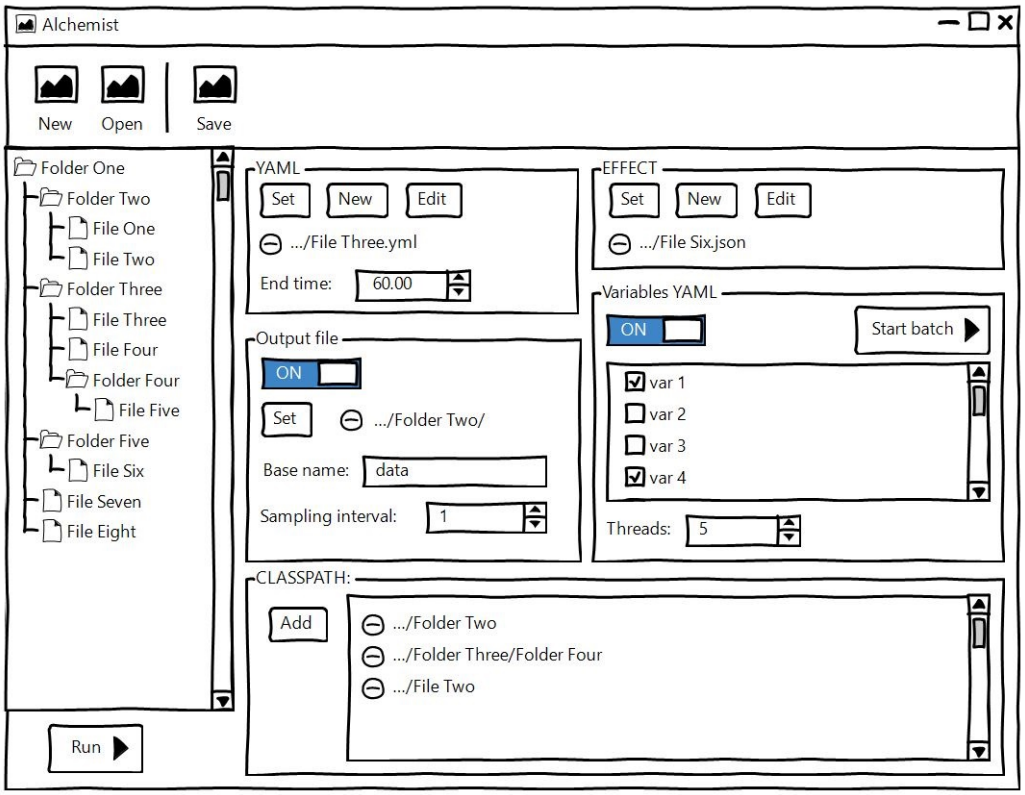


Figura 4.4: Quarta bozza dell'interfaccia grafica da realizzare, dove la sezione per selezionare la radice del progetto è stata eliminata e nella colonna a sinistra viene mostrato solamente l'albero del progetto.

- campo per inserire il nome del file di output;
- campo per settare ogni quanto tempo viene campionato il risultato della simulazione da inserire nel file di output;
- campo per settare il numero di thread da utilizzare in modalità batch.

Inoltre non viene più selezionato il file di output, ma viene selezionata la cartella in cui il file di output deve essere salvato.

4.2 Salvataggio dei dati

Il salvataggio della simulazione corrente deve essere progettato in modo tale da garantire la corretta scrittura e lettura dei dati da parte dell'applicativo.

Il file di salvataggio è scritto in linguaggio JSON ed ha un nome uguale per tutti i progetti Alchemist, così che il programma possa riconoscerlo per poterlo caricare.

JSON[1] o *JavaScript Object Notation* [4] è un formato di testo per la serializzazione dei dati strutturati, basato su JavaScript. Questo semplice formato per lo scambio di dati risulta essere facile da leggere e scrivere per le persone e facile da generare e analizzarne la sintassi per le macchine. È un formato di testo indipendente dal linguaggio di programmazione, ma utilizza convenzioni riconosciute dalla maggior parte dei programmatori di linguaggi come C, Java, ... Esso si basa su due strutture: un insieme di coppie nome-valore e un elenco ordinato di valori.

È stato scelto JSON per strutturare il file di salvataggio, perché è *human readable* ed è conosciuto alla maggior parte degli utenti.

4.3 Creazione di un nuovo progetto

Come detto nella sottosezione 3.1.8, si vuole dare la possibilità all'utente di creare un nuovo progetto.

Un progetto, per definirsi tale, deve avere una struttura del tipo mostrata in Figura 4.5, dove:

- nella cartella `data`, vanno posizionati tutti i file di output;
- nella cartella `java`, vanno posizionati i file e le cartelle delle librerie da aggiungere al classpath;
- nella cartella `json`, vanno posizionati tutti i file degli effetti;
- nella cartella `yaml`, vanno posizionati tutti i file di simulazione;
- il file `.alchemist_project_descriptor.json` è il file di salvataggio.

Questa struttura è necessaria per considerare quella cartella la cartella radice di un progetto Alchemist.

Nella creazione di un nuovo progetto deve prima essere richiesto all'utente la selezione di una cartella radice per il progetto che si va a creare. Bisogna però prevedere il caso in cui tale cartella non sia vuota, perché si potrebbero verificare ambiguità. Nel caso in cui l'utente scegliesse una cartella non vuota, gli verrà chiesto se si intenda procedere con il suo svuotamento oppure se si preferisca selezionarne un'altra.

In seguito alla selezione della cartella radice, è data la possibilità all'utente di scegliere se creare un progetto vuoto oppure utilizzando un template.

[1] http://www.json.org/index.html

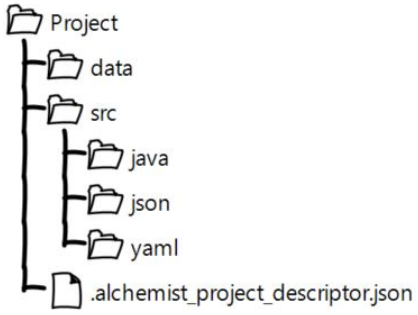


Figura 4.5: Immagine che mostra la struttura interna che deve avere l'albero di un progetto di Alchemist.

4.3.1 I template

I template previsti sono tre:

- Sapere
- Protelis
- Biochemistry

Come si può notare dal nome dei tre template, essi sono le tre incarnazioni di Alchemist attive al momento.

La selezione di uno dei template provoca la creazione di un progetto con la stessa struttura mostrata in Figura 4.5, ma all'interno delle cartelle sarà possibile trovare già delle simulazioni e dei file di effetti grafici, per poter avviare subito una prima simulazione.

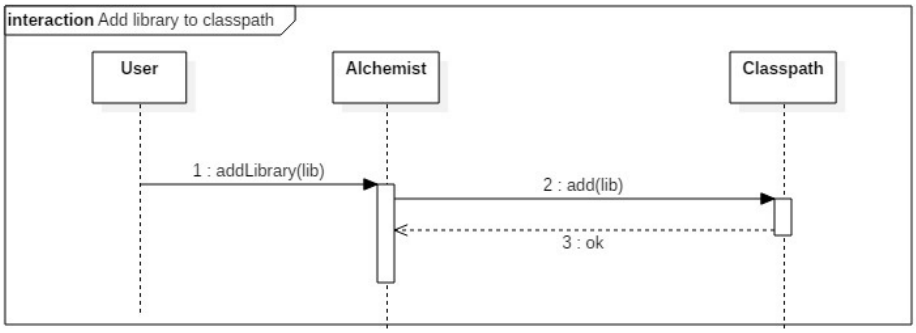
La strutturazione di questi template deve essere fatta in modo che, se in un futuro prossimo saranno create altre incarnazioni di Alchemist, queste possano essere mostrate senza dover modificare il codice Java scritto.

4.4 Aggiornamento del classpath

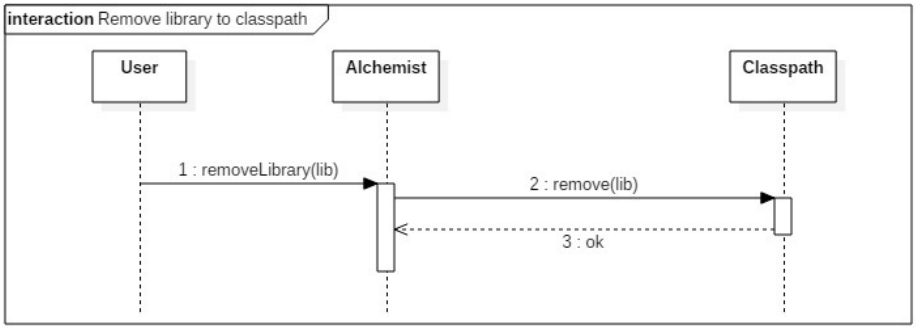
Aggiungere librerie al classpath è un lavoro complesso e delicato, che richiede l'utilizzo di particolari librerie che devono svolgere questo lavoro.

Il risultato che si vuole ottenere è avere un metodo efficiente e veloce, che permette di aggiungere senza problemi il file o la cartella selezionata al classpath.

Il metodo utilizzato per aggiornare il classpath è quindi quello di aggiungere o rimuovere la libreria non appena questa viene selezionata. Questo



(a) Aggiunta di una libreria al classpath.



(b) Rimozione di una libreria al classpath.

Figura 4.6: Diagrammi di sequenza che mostrano l'aggiornamento del classpath.

risulta essere il metodo più semplice, perché la rimozione di una libreria potrebbe causare qualche problema. Nel caso specifico, ogni volta che una libreria viene aggiunta al classpath, questa sarà anche memorizzata in una lista di librerie e ogni volta che viene rimossa, essa sarà rimossa anche dalla lista.

Il motivo di tale scelta è dovuto al fatto che, nel caso in cui la libreria viene aggiunta al classpath solamente poco prima dell'esecuzione della simulazione, si ha che la selezione dell'utente provoca il solo aggiornamento della lista di librerie. Nel caso in cui una libreria già precedentemente aggiunta al classpath deve essere rimossa, la deselezione dell'utente porta alla sola rimozione della libreria dalla lista. Quindi significa che nel momento in cui si deve eseguire la simulazione, non si sa più quale libreria va rimossa e quale tenuta.

Il funzionamento dell'aggiornamento viene mostrato nei diagrammi di sequenza di Figura 4.6.

4.5 Avvio della simulazione

L'avvio della simulazione in Alchemist consiste nel passare al simulatore una serie di dati, il quale li deve leggere e utilizzare per poter procedere con la simulazione.

L'azione appena descritta è svolta da una classe appartenente al package principale del simulatore. Essa prende gli argomenti passati da linea di comando, setta i campi utili al simulatore e dopo che ha controllato se deve eseguire la simulazione in modalità batch, decide come avviare la simulazione.

L'idea è quindi quella di rifattorizzare tale classe in una nuova. Questa nuova classe deve andare a settare tutti i componenti necessari a lanciare la simulazione. Ciò deve essere gestito dal pattern *Builder*, che mette a disposizione metodi che settano i componenti, grazie ai quali è poi possibile costruire l'istanza della classe. All'interno di tale classe devono, inoltre, essere disponibili due metodi: uno gestisce l'avvio della simulazione distinguendo i casi in cui sono o meno presenti le variabili YAML, così da distinguere se si è in modalità batch o no, e l'altro restituisce le variabili del file di simulazione.

Grazie all'utilizzo di questo pattern, si riesce a creare un'istanza dell'esecutore della simulazione in modo semplice e veloce, che sarà usata per avviare la simulazione oppure per ottenere le variabili presenti all'interno della simulazione.

La nuova classe verrà poi chiamata all'interno della classe che definisce l'istanza del progetto, all'interno del metodo chiamato per avviare l'esecuzione della simulazione. In questo luogo andranno fatti tutti i controlli sui componenti per evitare che si incappi in errori durante la costruzione del builder.

4.5.1 Pattern Builder

Come detto poco prima, per realizzare la rifattorizzazione della classe di Alchemist, è stato deciso di utilizzare il pattern creazionale builder.

Il builder, il cui schema UML è mostrato in Figura 4.7, è uno dei pattern fondamentali della programmazione ad oggetti, originariamente definito dalla "Gang of Four" all'interno del loro libro "*Design Pattern: Elements of Reusable Object-Oriented Software*".

Questo pattern creazionale permette di separare la costruzione di un oggetto complesso dalla sua rappresentazione. Ciò significa che la classe originale si concentra sul funzionamento degli oggetti, mentre il builder si occupa della corretta costruzione di una singola istanza dell'oggetto.

In sostanza, il builder definisce una strategia step-by-step per la costruzione di un oggetto, perchè mette a disposizione una serie di metodi per

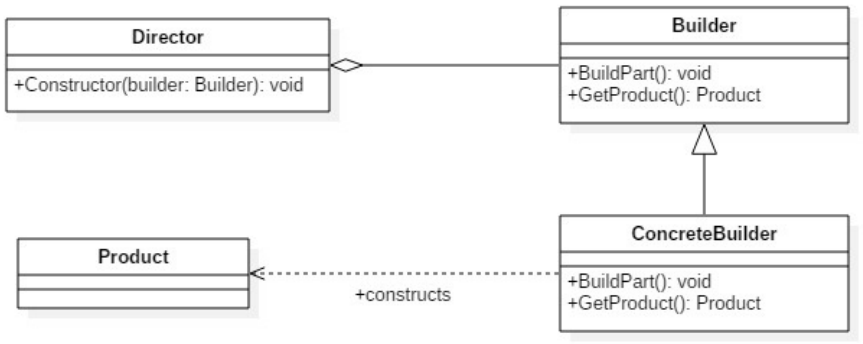


Figura 4.7: Diagramma delle classi che illustra il pattern Builder.

definire le sue proprietà e alla fine dà la possibilità di chiamare un metodo di building per costruirlo [18].

4.6 Gestore degli eventi

La gestione degli eventi che si verificano all'interno del file system è molto complicata, in quanto bisogna tenere conto di tutti i tipi di eventi che si possono verificare e bisogna gestire l'ascoltatore che deve dirigere tutta questa operazione.

L'idea di base è quella di avere un gestore che si mette in ascolto sulla cartella di radice del progetto. Deve tenere in memoria tutte le sottocartelle e i file che fanno parte del progetto e ogni volta che all'interno della cartella radice si verifica un evento, il gestore deve svolgere un'azione precisa. Infine questo gestore deve chiudersi con la chiusura del progetto.

Per implementare questo gestore degli eventi è stata utilizzata l'interfaccia `WatchService`.

4.6.1 WatchService

L'interfaccia WatchService[2], introdotta con Java 7, permette di implementare un watcher che registra al suo interno degli oggetti per cui è necessario catturare le modifiche e gli eventi.

Un oggetto può essere registrato all'interno di un watcher invocando il metodo di registrazione, che restituisce a sua volta una chiave che rappresenta la registrazione stessa. Questa chiave è necessaria, perché nel momento in cui

[2]https://docs.oracle.com/javase/8/docs/

viene catturato un evento esso si riferisce a una certa chiave, la quale, a sua volta, si riferisce a uno specifico oggetto. In seguito alla cattura dell'evento, la chiave deve poi essere ripristinata, in modo che possa essere pronta per catturare l'evento successivo.

Oltre alla registrazione, un oggetto può anche essere deregistrato dal watcher, invocando un metodo che annulli la chiave associata a tale oggetto.

Per implementare il WatchService è necessario, per prima cosa, ottenere il file system sottostante, in quanto dipendente da esso, e crearvi il nuovo watcher.

Una volta creato un nuovo watcher, si deve registrare la cartella radice del progetto, incluse tutte le sue sottocartelle. Ciò è necessario, perché il watcher altrimenti catturerebbe solamente gli eventi all'interno della cartella radice e non quelli all'interno delle sue sottocartelle. Per registrare tutto l'albero di cartelle appartenenti alla cartella radice, si deve compiere una serie di registrazioni ricorsive, in modo che tutto l'albero venga registrato.

La registrazione di un oggetto avviene invocando un metodo dell'interfaccia. Questo metodo necessita del watcher a cui va fatta la registrazione e di tutti gli eventi che si vogliono catturare. Bisogna notare, però, che gli unici oggetti che vengono registrati nel watcher sono cartelle, perché è al loro interno che si possono verificare nuovi eventi, cosa che non avviene per i file, i quali generano eventi solo per se stessi.

Una volta che il watcher viene mandato in esecuzione, questo si mette in attesa dell'arrivo di un evento. La cattura dell'evento può essere continua, cioè appena l'evento si verifica il watcher la cattura, oppure può essere fatta dopo che è trascorso un intervallo di tempo, impostato dal programmatore.

In seguito alla cattura di un evento, il watcher identifica il tipo di evento verificatosi ed esegue una opportuna operazione. Le tipologie di evento catturate dal watcher sono trattate nella sottosezione successiva.

Quando l'esecuzione dell'operazione è terminata, la chiave dell'oggetto viene resettata per poter così essere pronta a catturare l'evento successivo.

All'apertura di un nuovo progetto o alla chiusura del programma, il watcher deve essere chiuso, in modo che non continui a catturare eventi che non sono più di interesse.

Il watcher quindi, durante la sua esistenza, attraversa una serie di stati in cui può trovarsi, mostrati in Figura 4.8.

Il gestore degli eventi deve, quindi, essere sempre funzionante mentre il programma è in attività. Questa affermazione, ci porta a dire che è necessario posizionare il watcher all'interno di un thread separato in modo che la sua attività non blocchi l'attività dell'intero programma. L'utilizzo di un thread implica che la classe del watcher implementa l'interfaccia funzionale `Runnable`.

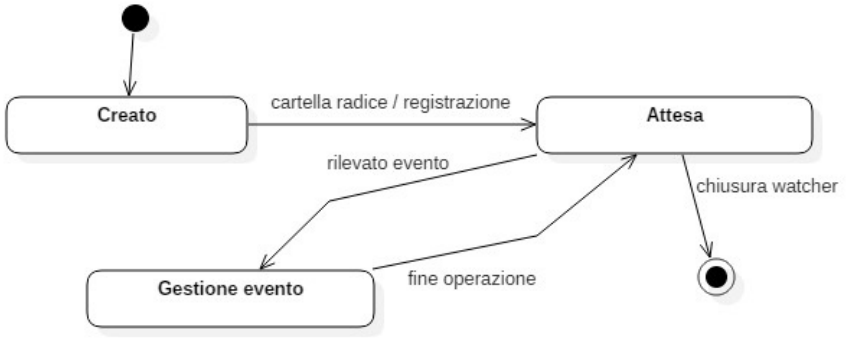


Figura 4.8: Diagramma di stato del watcher.

4.6.2 Eventi catturati

Come detto nella sottosezione precedente, il watcher deve catturare tutta una serie di eventi verificatisi all'interno della cartella radice del progetto.

L'interfaccia WatchService permette di catturare quattro tipi di eventi, forniti dalla classe `StandardWatchEventKinds`:

- `ENTRY_CREATE` - indica che un file o una cartella sono stati creati;
- `ENTRY_DELETE` - indica che un file o una cartella sono stati eliminati;
- `ENTRY_MODIFY` - indica che un file o una cartella sono stati modificati;
- `OVERFLOW` - indica che l'evento è stato perso oppure scartato.

Per catturare questi eventi, è necessario inserirli al momento della registrazione dell'oggetto al watcher, ad esclusione dell'evento `OVERFLOW`, il quale è sempre registrato implicitamente.

Come mostrato nel diagramma di attività di Figura 4.9, a seconda dell'evento catturato, il watcher deve compiere un'azione diversa.

Ogni azione, che il watcher deve eseguire sull'interfaccia grafica, deve essere invocata con il metodo `Platform.`*`runLater`*, il quale prende in ingresso un `Runnable`. Questa invocazione è necessaria in quanto il watcher e l'interfaccia dell'applicazione lavorano su due thread separati, che in un caso come l'esecuzione di una operazione sull'interfaccia causa conflitto. Grazie a questo metodo, l'operazione chiamata viene messa in una coda di eventi, che saranno poi evasi in un momento imprecisato del futuro. Una volta accodata l'operazione da svolgere nella coda di eventi, il controllo torna al thread del watcher, il quale può riprendere il suo precedente lavoro.

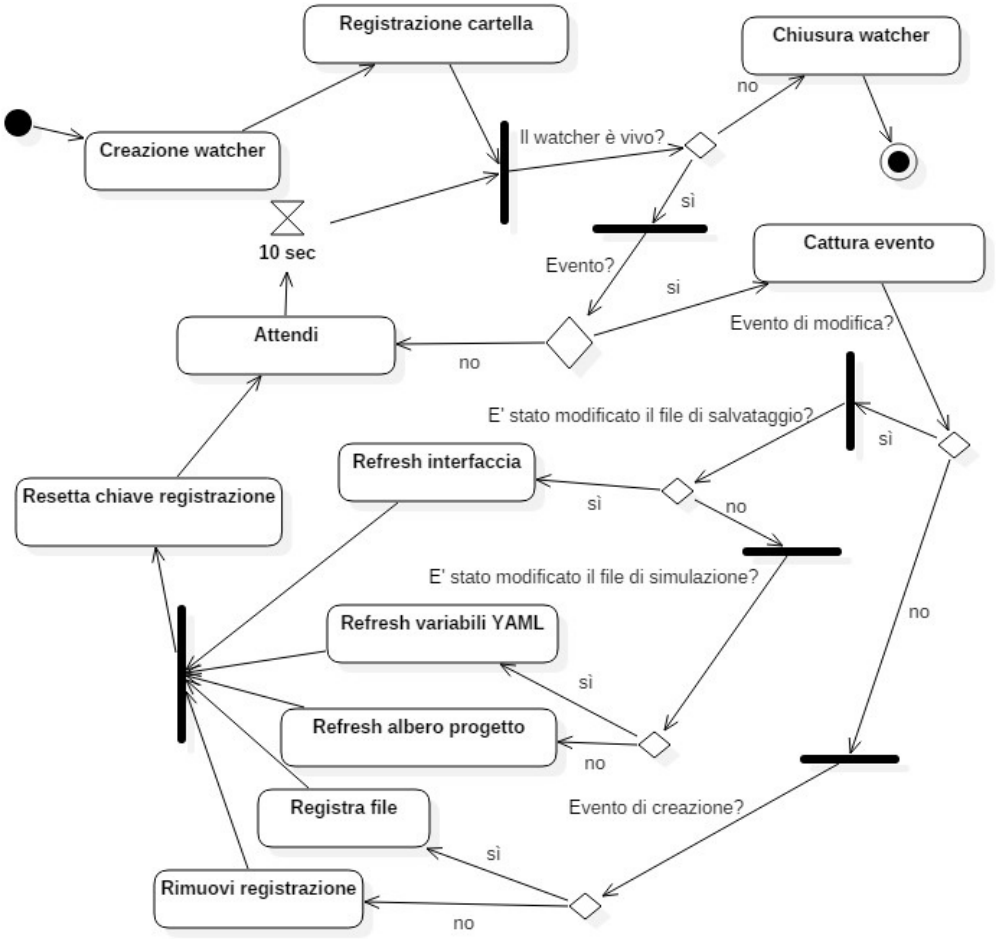


Figura 4.9: Diagramma di attività del watcher.

ENTRY_CREATE

L'evento `ENTRY_CREATE` è scatenato quando un nuovo file o una nuova cartella vengono creati o aggiunti alla cartella radice del progetto Alchemist.

Nel caso in cui questo evento viene catturato, si deve registrare il nuovo oggetto al watcher, così che, nel futuro, il watcher stesso possa catturarne gli eventi generati.

ENTRY_DELETE

L'evento `ENTRY_DELETE` è generato quando un file o una cartella, presenti all'interno della cartella radice del progetto, vengono eliminati o spostati in un'altra posizione del file system. In sostanza si verifica quando il file o la cartella vengono rimossi dalla cartella radice.

Se viene catturato questo evento, il watcher dovrà controllare se si tratta di una cartella o di un file. Nel caso in cui sia stato eliminato un file, il watcher deve controllare che quel file non fosse utilizzato all'interno della simulazione corrente, invece se l'oggetto eliminato era una cartella, il watcher deve rimuovere la sua registrazione e controllare che la cartella non fosse in uso nella simulazione.

ENTRY_MODIFY

L'evento `ENTRY_MODIFY` è provocato quando un file o una cartella sono stati in qualche modo modificati. Ad esempio, un semplice evento di modifica di una cartella è l'aggiunta di un nuovo file al suo interno.

Per la cattura di questo evento, il watcher può compiere tre azioni diverse:

- *refresh della GUI*, nel caso in cui l'oggetto modificato è il file di salvataggio della simulazione. Questo è necessario, perché l'utente potrebbe aver modificato alcune voci presenti e tali modifiche devono essere riportate anche sull'interfaccia grafica;

- *refresh delle variabili*, nel caso in cui l'oggetto modificato è il file di simulazione. Come si può intuire, modificando il file di simulazione, potrebbero essere state aggiunte o rimosse delle variabili, perciò è assolutamente necessario aggiornare l'elenco delle variabili mostrate all'interno dell'interfaccia;

- *refresh dell'albero delle cartelle del progetto*, per tutti gli altri casi che causano una modifica all'oggetto. Questo è dovuto al fatto che con tutta probabilità sono stati aggiunti nuovi file o cartelle, che quindi devono essere mostrati all'interno dell'albero del progetto.

4.7 Localizzazione

La maggior parte del software è sempre disponibile in più lingue, permettendo agli utenti di avere le interfacce grafiche nella propria lingua.

La localizzazione (detta anche *l10n*) è un processo che adatta un'applicazione ad un nuova lingua.

Attualmente, Alchemist supporta solamente l'inglese americano, ma dato che prevede la localizzazione, ha fatto in modo che se in un futuro si volesse aggiungere una nuova lingua, questa operazione sarebbe supportata.

Deve quindi essere creato un file di proprietà dove per ogni chiave è associata una stringa, così che per ogni componente grafico possa essere applicato il testo appropriato.

Per prima cosa è rifattorizzata la classe `LocalizedResourceBundle`, la quale permette di ottenere il `ResourceBundle` del file di proprietà, che è il riferimento alla risorsa, e per ogni chiave passata in input restituisce la stringa associata.

In ogni file in cui sarà necessario utilizzare tali stringhe, si creerà una istanza statica della classe `LocalizedResourceBundle`, a cui sarà passato il nome della risorsa da utilizzare, e poi si invocherà il metodo per ottenere le stringhe necessarie, in referimento a una data chiave.

Capitolo 5

Strumenti e metodi di sviluppo

Nel capitolo sono esposti tutti gli strumenti utilizzati per sviluppare il progetto.

5.1 Qualità del codice e controllo del software

Alchemist, come ogni progetto di medio-grandi dimensioni, deve mettere in pratica regole e tecniche che gli permettano di essere portabile e riproducibile, facendo anche uso di strumenti che controllino la qualità del codice.

5.1.1 Static source code analysis

Gli strumenti di qualità del codice sono strumenti molto utili in quanto permettono di revisionare il codice in modo sistematico, così da evitare errori che a volte possono verificarsi.

Questo tipo di analisi del codice viene effettuata senza bisogno che il programma venga realmente eseguito, in quanto viene analizzato il codice sorgente per individuare errori, come i potenziali bug o la duplicazione del codice, oppure per indicare i possibili miglioramenti e ottimizzazioni.

L'analisi è eseguita solitamente da uno strumento automatico. Gli strumenti utilizzati all'interno di Alchemist sono *FindBugs*, *CheckStyle* e *PMD*, con le configurazioni consigliate dal supervisore del progetto.

FindBugs

FindBugs[1] [1] [7] è un analizzatore di codice statico open source che rileva possibili bug all'interno del codice Java.

[1] http://findbugs.sourceforge.net/

Classifica i potenziali errori in categorie, per dare un'idea allo sviluppatore di quale potrebbe essere il loro impatto sul programma. Alcuni esempi di bug che può individuare sono la comparazione di stringhe con ==, quando si dovrebbe usare il metodo `equals()`, oppure la presenza di cicli ricorsivi infiniti o il cattivo utilizzo di librerie Java.

Questo analizzatore opera direttamente sul bytecode di Java, a differenza di tanti altri strumenti che operano sul codice sorgente.

CheckStyle

CheckStyle[2] [7] è utilizzato per lo sviluppo di codice software Java che aderisce a uno standard specifico di codifica. Permette così di risparmiare al programmatore un lavoro che, seppur molto importante, è alquanto noioso. È configurabile, e utilizzabile per diversi linguaggi e stili.

In pratica, effettua un controllo sulla presentazione e non sul contenuto, perciò non permette di assicurare la correttezza e la completezza del programma. Alcuni esempi di errori individuati sono la mancanza di commenti per la Javadoc, le spaziature non corrette oppure la presenza di magic numbers.

PMD

PMD[3] [7] è uno strumento di analisi del codice sorgente statico di Java, il quale prevede un set di regole per definire quando un pezzo di sorgente è errata. Questo insieme di regole può anche essere personalizzato dal programmatore.

In generale, gli errori segnalati non sono veri e propri errori, ma porzioni di codice subottimali. Ciò significa che il programma potrebbe continuare a funzionare anche se non sono immediatamente corretti gli errori segnalati.

PMD è utilizzato per trovare imperfezioni nel codice, come il mancato uso di `final`, variabili inutilizzate, la creazione di oggetti inutili oppure la duplicazione del codice. Quest'ultimo controllo è possibile grazie all'utilizzo del *Copy/Paste Detector* (o CPD), che è incluso all'interno di PMD.

5.1.2 Distributed Version Control

Per lo sviluppo di Alchemist si è fatto ricorso a un *Distributed Version Control System* (DVCS) [14], che permette agli sviluppatori di lavorare in modo concorrente su un progetto, oltre a tenere traccia delle modifiche apportate al codice sorgente e delle sue versioni.

[2] http://checkstyle.sourceforge.net/
[3] https://pmd.github.io/

Alchemist utilizza Git, che è stato creato nel 2005 da Linus Torvalds. In particolare, nel progetto si usa un flusso di lavoro adottato sovente da coloro che utilizzano Git chiamato *Git flow*.

Git flow

Git flow prevede che un progetto venga diviso in più branch, come mostrato in Figura 5.1, dove ognuno dei quali ha un proprio compito e un certo tempo di vita.

Si struttura nei seguenti branch:

- *Master*. È il ramo principale del progetto, che vive sempre. Qui il codice sorgente è in uno stato pronto per la produzione.
- *Develop*. Nasce dal master e su questo branch si sviluppa il codice per una release successiva del progetto.
- *Feature branches*. Sono utilizzati per sviluppare nuove funzionalità per il progetto, mentre altri team sviluppano altre feature. Partono dal branch develop e si uniscono sempre a develop.
- *Release branches*. Sono utilizzati per preparasi alla nuova versione del programma e correggere bug minori, infatti è solamente in questa occasione che si assegna un nuovo numero di versione. Partono dal branch develop e finiscono in develop o master.
- *Hotfix branches*. Si sviluppano dal branch master e sono creati nei casi in cui si vuole agire immediatamente sul codice per correggere un bug critico. Sono utilizzati quando la correzione del bug non può attendere che venga rilasciala la release successiva.

5.1.3 Build automation

Una buona pratica seguita da Alchemist è quella di avvalersi di due strumenti che si occupano di fare la build e la continuous integration.

La continuous integration si basa sul presupposto che un programma deve sempre funzionare in qualsiasi situazione. Per fare in modo che questo sia possibile è necessario eseguire le build di frequente per controllare che ogni modifica apportata non causi errori e che funzioni anche su sistemi freschi e puliti. La continuous integration, quindi, riduce la possibilità di bug e

[4]http://nvie.com/posts/a-successful-git-branching-model/

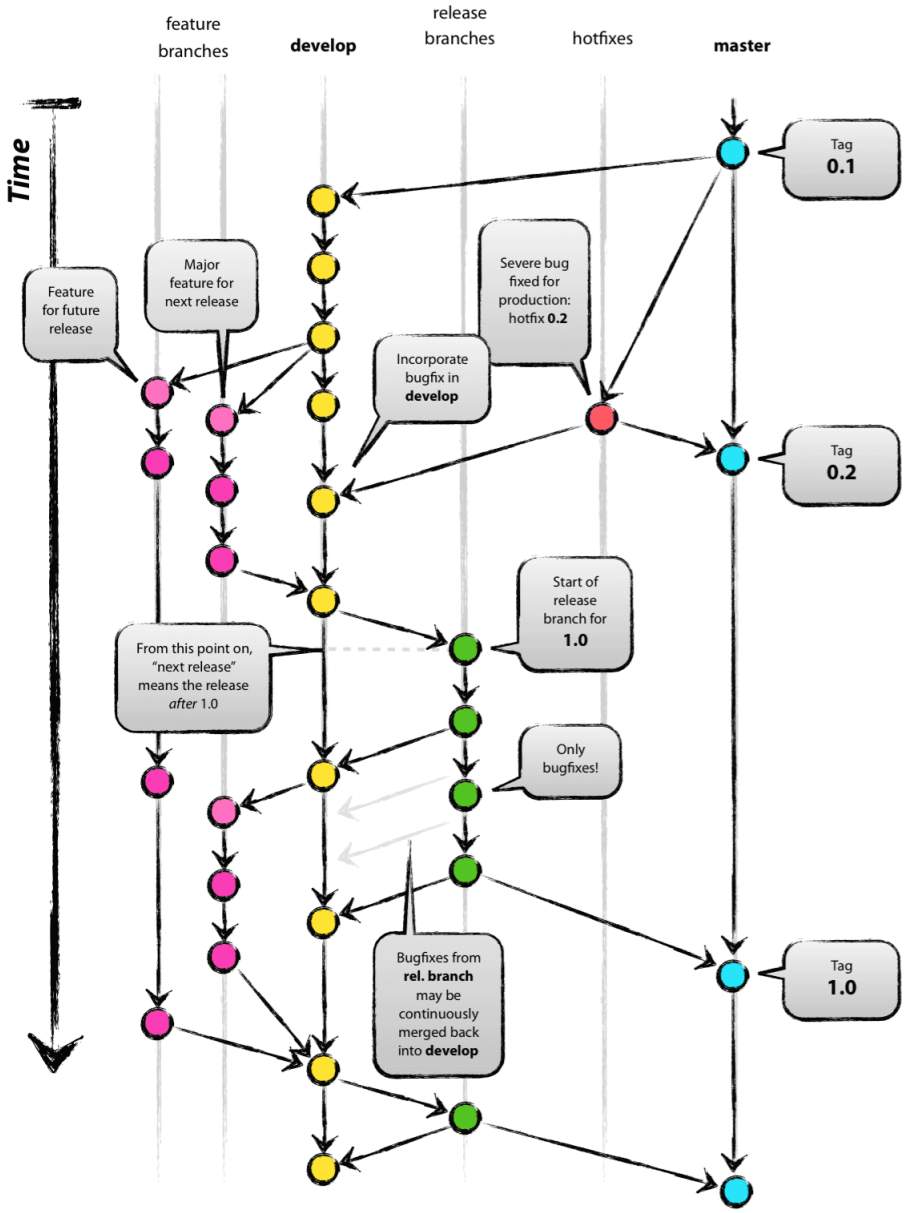


Figura 5.1: Diagramma[4] che illustra come, con lo scorrere del tempo, possano essere creati tanti branch e tutte le interazioni tra essi.

favorisce l'integrazione delle parti di software sviluppate da programmatori diversi, soprattutto quando si tratta di progetti molto grandi.

I due strumenti utilizzati sono *Gradle* e *Travis CI*.

Gradle

Gradle[5] [9] è un sistema per l'automazione dello sviluppo, nato per includere tutte le caratteristiche provenienti da Apache Ant, Apache Maven, Ivy e Groovy. È scritto in Java e Groovy e supporta molti linguaggi come Java e Scala. Questo sistema permette di avere una buona gestione delle dipendenze (o *dependency management*), in quanto scarica in automatico, durante la compilazione, le librerie e tutte le loro dipendenze necessarie per far funzionare il programma.

Le librerie scaricate da Gradle, all'interno di Alchemist, devono essere distribuite da *Maven Central*. Maven Central è un repository utilizzato per la deployment automatization, che permette di pubblicarvi le librerie personali solamente se si utilizzano le librerie da lui distribuite. Proprio per questo motivo, dato che Alchemist vuole pubblicare le proprie librerie sul repository, allora utilizza le librerie di Maven. Inoltre, all'interno del repository, vige la politica della "non retraibilità", ciò significa che una volta pubblicati gli artefatti, non possono più essere modificati in alcun modo.

Travis CI

Travis CI è un servizio di continuous integration distribuito, utilizzato per costruire e testare i progetti ospitati su GitHub[6] (servizio di hosting per Git). Esso permette il testing gratuito per i progetti open source. Per poterne usufruire è necessario configurare un file YAML presente nella directory del repository, dove va indicato il linguaggio di programmazione utilizzato nel progetto, la building desiderata e l'ambiente di test. Travis CI esegue la build sul repository indicato oppure su solamente una parte di esso. Una volta configurato l'ambiente, Travis CI avvia il test, a conclusione del quale, notifica i risultati ottenuti allo sviluppatore. Il suo funzionamento è mostrato in Figura 5.2.

[5] https://gradle.org/
[6] https://github.com/
[7] http://decks.eric.pe/pantheon-ci/

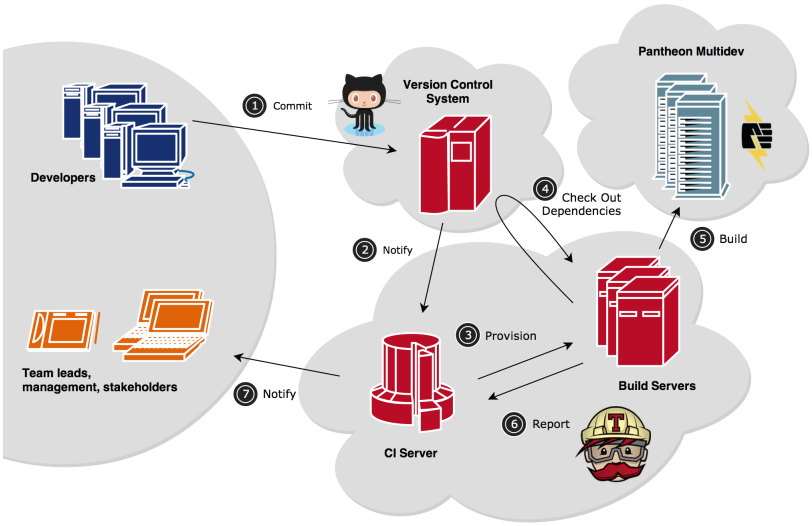


Figura 5.2: In questo schema[7] viene mostrato il funzionamento di Travis CI in abbinamento con GitHub. Come si può notare Travis prende il progetto da GitHub e tutte le sue dipendenze, ne costruisce la build e, una volta terminato il testing, notifica al team i risultati ottenuti.

5.2 Strumenti utilizzati

In questa sezione verranno presentati tutti gli strumenti utilizzati per sviluppare il codice dell'interfaccia grafica, descrivendone la loro utilità e il loro impiego.

5.2.1 IDE di sviluppo

L'IDE di sviluppo o *Integrated Development Environment* è un software che aiuta i programmatori nello sviluppo del codice sorgente di un programma. Esso mette a disposizione tutta una serie di strumenti e funzionalità utili per lo sviluppo e il debugging. Solitamente uno strumento di questo tipo mette a disposizione un editor di testo, un compilatore o un interprete (o entrambi) e un debugger.

L'IDE utilizzato per lo sviluppo dell'interfaccia grafica è *Eclipse*, software libero che mette a disposizione un ambiente multi-linguaggio. Questa piattaforma è incentrata sull'uso di plug-in, componenti software con un compito specifico, che ne permettono di ampliare le funzionalità.

Anche se si è utilizzato Eclipse come IDE di sviluppo, si è fatto in modo che il codice sia indipendente dall'IDE, così che possa funzionare su una qualsiasi piattaforma.

5.2.2 Plug-in

Un plug-in è un programma autonomo che interagisce con un altro programma per ampliarne le funzionalità originarie.

Come detto nella sottosezione precedente, Eclipse ne fa largo uso. Tra i plugin utilizzati per il progetto possiamo individuare quelli per la static source code analysis (*FindBugs*, *CheckStyle* e *PMD*) e quello per implementare progetti Gradle (*Buildship*).

Un altro plug-in di cui si è fatto largo uso è *e(fx)clipse*, il quale permette di creare progetti di applicazioni JavaFX. Grazie a questo plug-in è possibile aggiungere ai progetti file in linguaggio FXML, utilizzate per realizzare le view dell'interfaccia, e file CSS per implementare la grafica.

È stato utilizzato anche il plug-in di git, il quale è integrato all'interno di Eclipse dalle sue ultime versioni.

5.2.3 Scene Builder

Scene Builder è uno strumento di layout visivo che consente agli utenti di progettare rapidamente le interfacce utente delle applicazioni in JavaFX, generando automaticamente il codice FXML. Lo sviluppatore può prendere i componenti e trascinarli all'interno della zona di lavoro, ne può modificare le proprietà e applicarvi i fogli di stile.

Esso può essere utilizzato in combinazione con qualsiasi IDE Java, ma è integrato principalmente in NetBeans IDE e in Eclipse. La versione utilizzata di questo software è quella distribuita da Gluon[8].

La scelta di utilizzare questo strumento è stata dettata dalla semplicità con cui si possono realizzare le interfacce, in quanto i componenti disponibili possono essere posizionati in modo drag-and-drop, ottenendo direttamente il codice FXML all'interno del file designato. Inoltre offre la possibilità di vedere in anteprima il risultato finale, in modo che si possano effettuare immediatamente le modifiche necessarie alla disposizione dei componenti.

Le uniche cose che non sono state impostate direttamente all'interno di Scene Builder sono le icone dei pulsanti, i testi dei componenti, i controller e il foglio di stile. Questo è dovuto al fatto che Scene Builder permette di impostare tutte queste componenti solamente se esse sono collocate nello stesso

[8]http://gluonhq.com/labs/scene-builder/

Figura 5.3: ToggleSwitch messo a disposizione dalla libreria `controlsfx` nei suoi due possibili stati (on e off).

package del file FXML. Alchemist, però, per avere un codice bene organizzato, prevede che le classi siano posizionate in package diversi a seconda della loro destinazione d'uso, mentre le immagini e i fogli di stile devono essere situati nella cartella delle risorse.

5.2.4 Librerie esterne

Per realizzare questa interfaccia grafica è stato necessario utilizzare delle librerie esterne che forniscono ulteriori funzionalità, rispetto a quelle già fornite dalle librerie di base.

Le librerie esterne utilizzate all'interno del progetto sono: `controlsfx` e `gson`.

Controlsfx

Controlsfx è una libreria open source per JavaFX, che mette a disposizione controlli per interfacce di alta qualità e altri strumenti. È stata realizzata per la versione 8 di JavaFX.

Questa libreria, nella versione 8.40.12, è stata impiegata per realizzare gli switch, che la libreria interna di JavaFX non mette a disposizione.

Per implementare gli switch, è stata utilizzata la classe `ToggleSwitch`, che realizza il componente mostrato in Figura 5.3.

Gson

Gson è una libreria Java open source che è utilizzata per convertire gli oggetti Java nella loro rappresentazione JSON oppure per convertire una stringa JSON nell'equivalente oggetto Java.

Questa libreria è stata utilizzata per implementare la lettura e la scrittura del file JSON di salvataggio.

Capitolo 6

Implementazione

In questo capitolo vengono trattate le scelte implementative più rilevanti attuate per la realizzazione dell'interfaccia grafica di Alchemist sviluppata.

L'intero progetto è stato sviluppato nel linguaggio di programmazione Java, il quale è il linguaggio utilizzato per realizzare l'intero simulatore di Alchemist, e facendo largo uso del software applicativo JavaFX.

6.1 Interfaccia grafica

L'interfaccia del simulatore di Alchemist, ottenuta dopo lo sviluppo di tutto il progetto, è quella mostrata in Figura 6.1, la quale è composta da tre sezioni, ognuna delle quali è gestita da un proprio controller. Queste sezioni sono:

- la barra in alto, dove sono presenti i pulsanti per creare un nuovo progetto, aprire un progetto esistente e salvare il progetto corrente;
- la colonna a sinistra, dove è presente un TreeView per rappresentare l'albero della directory del progetto e il pulsante per eseguire la simulazione corrente;
- il corpo centrale, diviso in cinque sezioni dove si permette all'utente di configurare la simulazione da avviare, gli effetti da applicare, il file di output, le variabili di simulazione e il classpath. È anche presente un pulsante per avviare la simulazione in modalità batch.

Qui di seguito verranno spiegate le implementazioni di alcune parti dell'interfaccia più rilevanti.

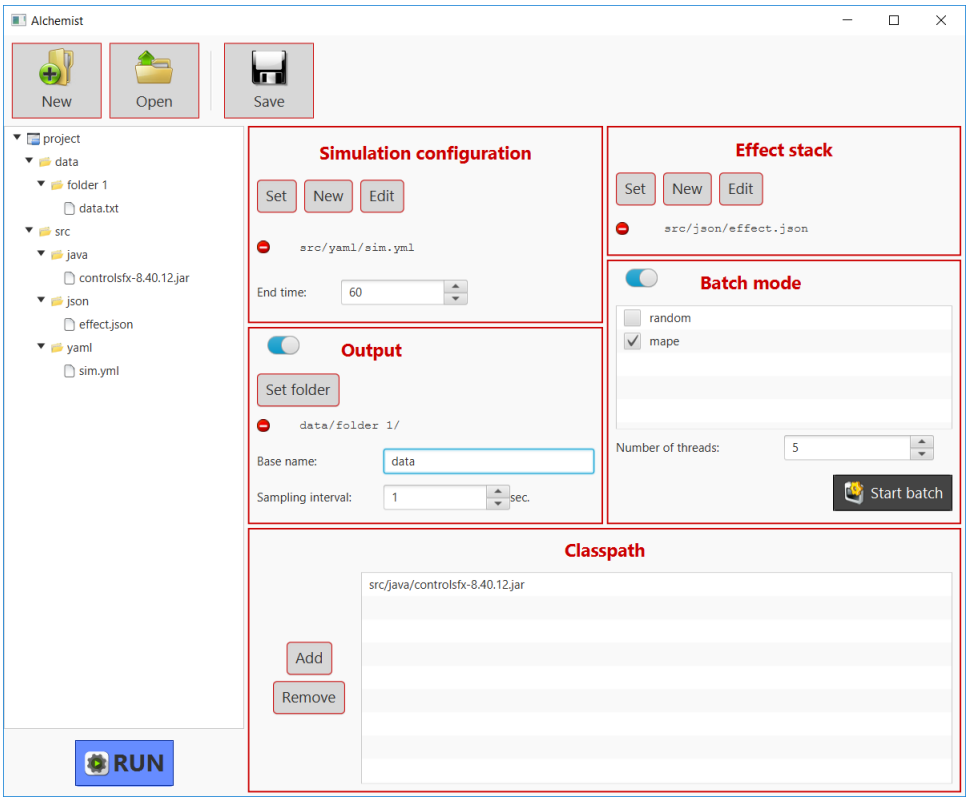


Figura 6.1: Immagine che mostra come si presenta il simulatore di Alchemist durante la configurazione di una simulazione.

6.1.1 Albero della directory

L'albero della directory deve mostrare tutti i file e le cartelle che compongono la cartella radice del progetto.

Per realizzare tale albero è stato utilizzato il componente di JavaFX chiamato `TreeView`, il quale permette di impostare un nodo radice a cui si possono associare tutta una serie di foglie o altri nodi. È quindi stato necessario scorrere l'intera cartella del progetto in modo ricorsivo, per poter creare la struttura corretta, facendo in modo che il file di salvataggio non venisse mostrato.

Come si può notare dal Codice A.1 presente nell'Appendice A, l'utente, sull'albero, può compiere tre tipi di azioni, ognuna delle quali è stata gestita in modo appropriato:

- *click del mouse su un elemento.* Cliccando su un elemento si seleziona l'oggetto, che poi potrà essere usato come file di simulazione, file degli effetti, ... a seconda della configurazione settata.

- *doppio click del mouse su un file.* Il doppio click su un file permette all'utente di aprirlo in un editor di testo, così che possa modificarlo. Questa azione non vale per le cartelle, in quanto il doppio click fa espandere o ridurre il suo sottoalbero.

- *click con il tasto destro del mouse su un elemento.* Il click del tasto destro su un elemento permette di aprire un menu contestuale (mostrato in Figura 6.2) in cui è possibile selezionare due azioni: la creazione di una nuova cartella o la creazione di un nuovo file. Per entrambe queste azioni si aprirà una finestra in cui è possibile inserire il nome del file o della cartella da creare. Il nuovo file o la nuova cartella sarà posizionato nella cartella in cui si trova il file selezionato oppure nella cartella selezionata con il tasto destro.

6.1.2 Variabili di simulazione

Per implementare la lista di variabili di simulazione selezionabili è stato necessario creare una `ListView`, dove ogni elemento è stato definito come un `CheckBoxListCell`. Settando ogni elemento in questo modo, è stato reso possibile aggiungere un checkbox per ogni elemento della lista, permettendo così all'utente di selezionare o deselezionare le variabili. Per catturare l'evento di selezione e di deselezione, è stato associato a ogni checkbox un listener, come mostrato nel Codice A.2 presente nell'Appendice A.

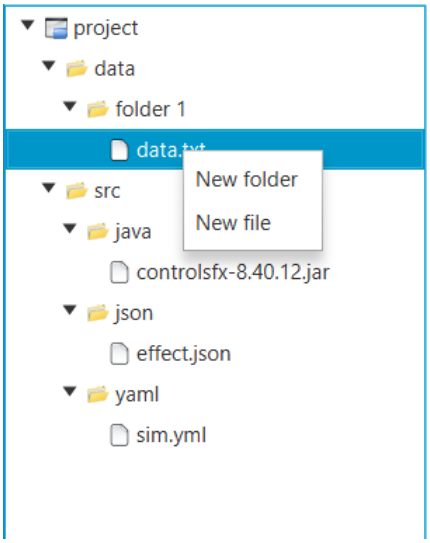


Figura 6.2: Immagine che mostra il `TreeView` con il menu contestuale aperto.

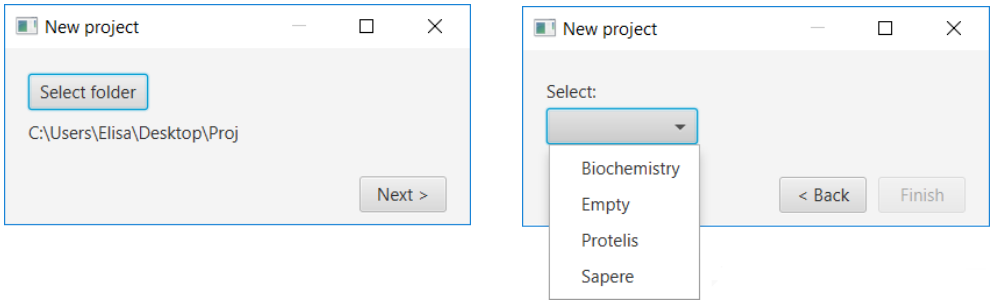
6.1.3 Creazione di un nuovo progetto

Per creare un nuovo progetto, l'utente deve cliccare sul pulsante "*New*". In seguito a questa azione si apre una finestra uguale a quella di Figura 6.3a, in cui è possibile scegliere la cartella vuota per il nuovo progetto. Se la cartella selezionata non è vuota, viene chiesto all'utente se la vuole reinizializzare, altrimenti sarà costretto a selezionare un'altra cartella. Una volta selezionata, si attiva il pulsante *Next*, premuto il quale fa cambiare la scena. Nella nuova scena, mostrata in Figura 6.3b, l'utente può selezionare il tipo di template che vuole per creare il nuovo progetto. Una volta selezionato il template si può cliccare sul pulsante *Finish*, che andrà quindi a creare la struttura del progetto nella cartella indicata.

Come già detto in precedenza i template disponibili sono tre, come le incarnazioni di Alchemist attive al momento, e in più si dà la possibilità di creare anche un progetto con il template vuoto. La lista di elementi del `ChoiceBox` viene popolata prendendo il nome dei package presenti nelle risorse del progetto, posizionati sotto il package `templates`. Questo permette ogni volta di creare la corretta struttura del progetto.

6.2 Salvataggio e caricamento dati in JSON

Come spiegato nella sezione 4.2 è stato implementato il salvataggio della simulazione corrente in un file JSON, inoltre è stato anche implementato il



(a) Selezione della cartella. (b) Selezione del template.

Figura 6.3: Immagine che mostra le finestre che si aprono quando si vuole creare un nuovo progetto. Prima si apre la finestra di Figura a, poi una volta premuto il pulsante *Next* si apre la finestra di Figura b.

caricamento dei dati presi dal file JSON di salvataggio.

Per poter però salvare i dati presenti nell'interfaccia all'interno del file JSON oppure leggerli, è necessario convertire gli oggetti Java nella loro rappresentazione JSON. Questo ci porta alla conclusione che è necessario realizzare una struttura di classi che permetta di creare un oggetto che dovrà in seguito essere convertito in JSON utilizzando la libreria `gson`.

Sono quindi state create tre classi (Figura 6.4):

- `Project` - è l'istanza del progetto, dove vengono memorizzati tutti i dati relativi alla simulazione. Questa classe deve permettere di avviare la simulazione e di trovare le variabili provenienti dal file YAML.
- `Output` - è l'istanza della sezione di selezione del file di output, dove vengono memorizzati tutti i dati relativi, come ad esempio la cartella di output, il nome del file, ...
- `Batch` - è l'istanza della sezione di selezione delle variabili YAML, dove vengono memorizzati tutti i dati relativi, come le variabili selezionate o il numero di thread.

Oltre a queste classi, è stata implementata una classe di utilità che permette, attraverso due metodi statici, di leggere o scrivere il file JSON.

Questa classe, chiamata `ProjectIOUtils`, è composta dai seguenti metodi:

- `loadFrom` (Codice A.3, Appendice A): metodo che, data la directory della cartella radice del progetto, restituisce l'entità del progetto, all'interno della quale è stato deserializzato il reader di JSON.

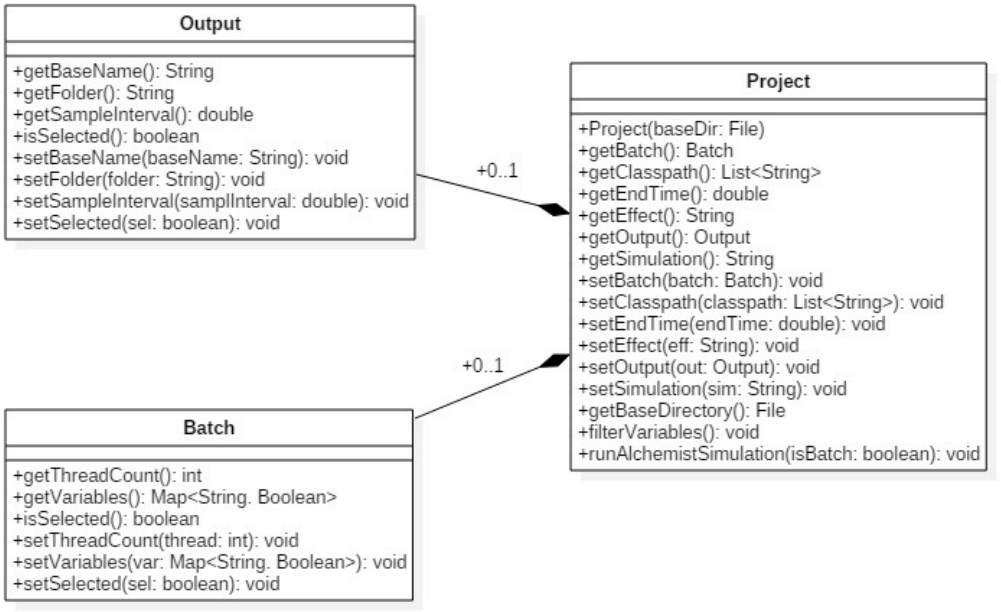


Figura 6.4: Diagramma delle classi utilizzate per il salvataggio dei dati.

- `saveTo` (Codice A.4, Appendice A): metodo che, data la directory della cartella radice e l'entità del progetto, scrive all'interno del file designato la rappresentazione JSON dell'entità passata in input. Il file di salvataggio che si ottiene con questa operazione ha una struttura uguale a quella mostrata nel JSON schema di Codice A.5 (Appendice A).

6.3 Watcher

L'implementazione del gestore degli eventi è stata fatta realizzando la classe `Watcher`, che implementa l'interfaccia `Runnable`. Tale classe istanzia l'entità di un WatcherService, così da poter controllare gli eventi che si verificano all'interno di una certa directory.

6.3.1 Registrazione al watcher

La registrazione al watcher deve essere fatta in modo ricorsivo, in quanto è necessario registrare tutte le cartelle presenti all'interno della cartella radice del progetto, altrimenti non sarebbe possibile catturare gli eventi che si verificano all'interno di queste.

Per realizzare la registrazione ricorsiva si è scritto un metodo privato, richiamato dal metodo pubblico `registerPath`, il quale visita tutto il sottoalbero della directory passata in input e registra le cartelle al watcher.

Ogni coppia formata da chiave di registrazione e posizione della cartella sono inserite all'interno di una `Map`, dove sono mantenute tutte le coppie chiave-directory. Mantenere questa mappa è necessario in quanto serve per controllare la cancellazione delle chiavi alla cattura dell'evento `ENTRY_DELETE`. Infatti se non esistesse tale oggetto, alla cattura dell'evento di cancellazione, sarebbe eliminata sempre la chiave di registrazione anche quando si tratta di un file. Ciò porta alla deregistrazione di una cartella, che in realtà è ancora in vita e che può generare altri eventi.

Durante la realizzazione del Codice A.6 (Appendice A), è stato notato un bug che si verifica solamente nel caso di utilizzo del sistema operativo Windows. Infatti, durante la registrazione, veniva in alcuni casi lanciato un errore. Questo si verificava perché Windows dà la possibilità all'utente di rinominare la nuova cartella appena creata, ma, in alcuni casi, si tiene il controllo troppo a lungo, creando un conflitto con il watcher che tenta la registrazione di tale cartella. L'unico modo per evitare questo problema, è stato quello di introdurre un ciclo di tre tentativi in cui il watcher cerca di registrare la cartella. Nel caso in cui viene catturata l'eccezione, il watcher attende dieci millisecondi e poi ritenta la registrazione.

6.3.2 Cattura degli eventi

Le operazioni indicate all'interno del metodo, mostrato nel Codice A.7 (Appendice A), sono svolte fino a quando il watcher è in vita, altrimenti viene chiusa l'istanza del `WatchService`.

Come detto in precedenza, gli eventi possono essere catturati in modo continuo oppure dopo che è trascorso un certo lasso di tempo. La scelta è ricaduta sul secondo metodo, in quanto ha un costo meno elevato in termini di prestazioni. Il watcher, quindi, si mette in attesa per dieci secondi, poi si risveglia, per vedere se si sono verificati degli eventi.

Nel caso in cui si siano verificati, il watcher trova il tipo di evento verificatosi e l'oggetto che ha provocato questo evento, svolgendo un'operazione diversa a seconda dell'evento catturato. Infine resetta la chiave di registrazione, così da prepararla per la cattura dell'evento successivo.

Evento di modifica

Quando viene catturato l'evento di modifica, il watcher deve controllare il file a cui si riferisce la modifica, in modo da compiere le operazioni necessarie.

Nel caso in cui la modifica si riferisce al file di salvataggio, significa che l'utente avrà cambiato qualche parametro a mano, perciò l'intera interfaccia va ricaricata in modo da mostrare le modifiche applicate. Questa operazione (Appendice A, Codice A.8, metodo `refreshGrid`) viene svolta da un metodo del controller che gestisce la sezione centrale della GUI.

Nel caso in cui la modifica si riferisce al file di simulazione, può succedere che le variabili presenti al suo interno siano state modificate, ne siano state aggiunte nuove oppure eliminate. È quindi necessario ricaricare le variabili dal file di simulazione (Appendice A, Codice A.8, metodo `refreshVariables`).

In tutti gli altri casi di modifica, non previsti dai precedenti, l'unica azione necessaria è quella di ricaricare l'albero delle cartelle del progetto, in modo da mostrare tutte le modifiche apportate alla directory (Appendice A, Codice A.8, metodo `refreshTreeView`). Questa operazione viene svolta dal controller che gestisce la sezione della GUI in cui è presente il TreeView.

Evento di creazione

Alla cattura dell'evento di creazione l'unica cosa da fare è quella di richiamare il metodo di registrazione per registrare nel watcher il nuovo oggetto che è stato creato nella directory.

Un'altra operazione da svolgere è controllare se la cartella in cui è stato creato tale oggetto è uguale alla cartella radice del progetto. Questo controllo è necessario, in quanto la creazione di un file all'interno della cartella radice non genera l'evento di modifica per la cartella radice stessa, come invece avviene per le sue sottocartelle. Ciò significa che, in questo caso, si deve aggiornare appositamente l'albero della directory (Appendice A, Codice A.8, metodo `refreshTreeView`).

Evento di cancellazione

In seguito alla cattura dell'evento di cancellazione, si deve cercare la chiave da eliminare all'interno della mappa.

Se tale chiave è stata trovata, significa che l'oggetto eliminato era una cartella, quindi sarà necessario eliminare la chiave sia dalla mappa sia dal registro del watcher (Appendice A, Codice A.8, metodo `refreshFolder`). Inoltre si dovrà controllare che tale cartella non fosse utilizzata all'interno della simulazione, pena un messaggio di errore. Tale compito, però, spetta al controller che gestisce questi dati.

Se la chiave di registrazione non era presente nella mappa, significa che era stato eliminato un file. In questo caso va controllato che tale file non

fosse utilizzato nella simulazione corrente (Appendice A, Codice A.8, metodo `refreshFile`), altrimenti viene lanciato un messaggio di errore.

Sempre per lo stesso motivo indicato per l'evento di creazione, va controllato che la cartella da cui è stato eliminato l'oggetto non fosse uguale alla cartella radice, perché altrimenti andrà ricaricato l'albero delle directory.

6.4 Classpath

L'aggiunta e la rimozione di librerie dal classpath è stata implementata realizzando due metodi privati, in cui uno aggiunge la libreria, mentre l'altro la rimuove dal classpath. Questi metodi sono invocati nel momento in cui l'utente decide di aggiungere o rimuovere la libreria. Il motivo di tale scelta è stato spiegato nella sezione 4.4.

Il metodo per l'aggiunta di una libreria, mostrato nel Codice A.9 (Appendice A), crea un'istanza del classpath di sistema corrente. Invocando un metodo su questa istanza è quindi possibile aggiungere l'URL della libreria corrente, in coda a tutte le altre librerie già presenti. Nel caso in cui l'operazione di aggiunta di tale libreria non è andata a buon fine, il metodo restituisce `false`, così che possa essere gestito l'errore in modo opportuno.

La rimozione di una libreria, mostrata nel Codice A.10 (Appendice A), necessita di avere l'istanza del classpath corrente, in quando deve ricercare al suo interno la libreria da rimuovere e, una volta individuata, la elimina. Anche in questo caso si ha come valore di ritorno un booleano per poter controllare che la libreria sia stata correttamente rimossa.

Come si può notare, le librerie aggiunte vengono posizionate come ultime occorrenze della lista delle librerie del classpath. In realtà era stato richiesto che le librerie aggiunte dall'utente dovessero essere collocate in testa a tale lista. Questo purtroppo non si è potuto fare per motivi implementativi. È stato inoltre realizzato un test per verificare la correttezza dell'idea di base e la possibile realizzazione di questo metodo.

L'idea da cui si è partiti era quella di creare un proprio classpath in cui erano posizionate le librerie aggiunte dall'utente seguite da quelle del programma, poi questo classpath doveva essere utilizzato quando si mandava in esecuzione la simulazione.

Il primo test è stato eseguito per verificare la corretta creazione del nuovo classpath con le librerie aggiunte in testa alla lista. Il risultato ottenuto da tale test è stato positivo, in quanto nella lista delle librerie del classpath creato erano state correttamente concatenate le librerie appena aggiunte con le librerie già presenti.

Il secondo test è stato compiuto per verificare che il classpath appena creato potesse essere usato al posto del classpath di sistema. In questo caso il test ha restituito esito negativo, perché il sistema continuava ad utilizzare il suo classpath di base.

Capitolo 7

Conclusioni

L'obiettivo di questa tesi era quello di sviluppare un'interfaccia grafica del simulatore Alchemist che andasse ad integrare quella già esistente, permettendo la configurazione di una simulazione e il suo avvio.

Per lo sviluppo del progetto è stata utilizzata la libreria JavaFX, che ha permesso di mantenere separato l'aspetto del programma dalla specifica comportamentale. Sfruttando le potenzialità di questa libreria, l'interfaccia è stata realizzata in modo agevole e con la grafica del tutto personalizzata, grazie all'utilizzo del CSS.

I requisiti definiti in fase di analisi sono stati quasi tutti pienamente soddisfatti, portando l'interfaccia ad essere conforme per poter essere integrata in futuro all'interno del progetto Alchemist.

Un requisito che non si è potuto soddisfare riguarda la gestione del classpath. Non si è, infatti, potuto costruire in modo che le librerie aggiunte venissero posizionate prima di quelle già presenti. Questo non implica, però, che nel prossimo futuro non si riesca a trovare una soluzione a questo problema.

Anche il supporto Hi-DPI potrà essere soddisfatto solamente in futuro, in quanto attualmente JavaFX non lo estende per i sistemi operativi Linux e Windows. Si è comunque cercato di rendere l'interfaccia il più scalabile possibile, utilizzando unità di misura relative o che dipendano dalla risoluzione dello schermo, così che l'interfaccia possa essere usabile, mentre si attende che venga esteso il supporto.

L'introduzione del concetto di "progetto" ha permesso lo sviluppo di un ambiente più intuitivo per l'utente che lo deve andare ad usare, perché rende la sua esperienza d'uso molto simile a quella di un qualsiasi IDE di sviluppo. La speranza è che questa somiglianza porti un incremento nell'utilizzo del simulatore Alchemist, anche da parte di quei soggetti che non sono molto esperti.

Un punto di forza dell'interfaccia è quello di essere consistente con lo stato del file system. Ciò è molto importante perché in qualunque modo vengano modificate le simulazioni o la directory del progetto, l'ambiente si aggiornerà di conseguenza, evitando così di incorrere in errori dovuti alla perdita di dati.

In conclusione, l'interfaccia realizzata è sicuramente pronta per essere distribuita al pubblico, ma questo non significa che non possa essere integrata e ampliata nel futuro. Uno dei prossimi lavori previsti è quello di integrare tale interfaccia di un editor, permettendo così di realizzare un proprio file, senza dover abbandonare l'ambiente del simulatore Alchemist.

Ringraziamenti

Ringrazio il professor Mirko Viroli e il professor Danilo Pianini per la bella opportunità che mi hanno offerto, per tutto l'aiuto datomi per realizzare questo progetto e per i consigli ricevuti. Ringrazio anche gli amici che mi sono stati accanto durante questi mesi, ma soprattutto un particolare grazie ai miei genitori che mi sostengono da sempre.

Appendice A

Codice

A.1 Albero delle directory

Codice A.1: Codice che mostra come sono stati implementati l'albero della cartella radice del progetto e la cattura degli eventi che il `TreeView` può generare. Ad ogni nodo è stata associata un'icona che ne rappresentasse la tipologia, cioè se si tratta di una cartella radice, di una cartella o di un file.

```
1  public void setTreeView(final File dir) {
2   this.pathFolder = dir.getAbsolutePath();
3   final TreeItem<String> root =
4           new TreeItem<>(dir.getName(), new ImageView(
5                   new Image(ProjectGUI.class
6                           .getResource("/icon/project.png")
7                           .toExternalForm())));
8   root.setExpanded(true);
9   this.treeView = new TreeView<>(root);
10  displayProjectContent(dir, root);
11  this.pane.getChildren().add(this.treeView);
12  this.treeView.getSelectionModel()
13      .selectedItemProperty()
14      .addListener(
15          new ChangeListener<TreeItem<String>>() {
16   @Override
17   public void changed(
18       final ObservableValue<? extends TreeItem<String>>
             observable,
19       final TreeItem<String> oldVal,
20       final TreeItem<String> newVal) {
21    final TreeItem<String> selectedItem =
22                              (TreeItem<String>) newVal;
23    TreeItem<String> parent = selectedItem.getParent();
24    String path = File.separator +
```

```
25                          selectedItem.getValue();
26     while (parent != null)  {
27      if (parent.getParent() != null) {
28        path = File.separator + parent.getValue() + path;
29      }
30        parent = parent.getParent();
31     }
32     selectedFile = pathFolder + path;
33    }
34   });
35   this.treeView.setOnMouseClicked(
36                       new EventHandler<MouseEvent>() {
37    @Override
38    public void handle(final MouseEvent mouseEv) {
39     if (mouseEv.getClickCount() == 2
40           && new File(selectedFile).isFile()) {
41      final Desktop desk = Desktop.getDesktop();
42      try {
43         desk.open(new File(selectedFile));
44      } catch (IOException e) {
45         L.error("Error opening file.", e);
46         System.exit(1);
47      }
48     }
49    }
50   });
51   final ContextMenu menu = new ContextMenu();
52   final MenuItem newFolder =
53          new MenuItem(RESOURCES.getString("new_folder"));
54   newFolder.setOnAction(new EventHandler<ActionEvent>() {
55    @Override
56     public void handle(final ActionEvent event) {
57        loadLayout(true);
58     }
59   });
60   final MenuItem newFile =
61           new MenuItem(RESOURCES.getString("new_file"));
62   newFile.setOnAction(new EventHandler<ActionEvent>() {
63    @Override
64    public void handle(final ActionEvent event) {
65        loadLayout(false);
66    }
67   });
68   menu.getItems().addAll(newFolder, newFile);
69   this.treeView.setContextMenu(menu);
70  }
71
72  private void displayProjectContent(final File dir,
73                             final TreeItem<String> root) {
```

```
74    final File[] files = dir.listFiles();
75    if (files != null) {
76      for (final File file: files) {
77        if (!file.getName()
78          .equals(".alchemist_project_descriptor.json")) {
79          final TreeItem<String> singleFile;
80          if (file.isDirectory()) {
81            singleFile = new TreeItem<>(file.getName(), new
                 ImageView(this.folder));
82            displayProjectContent(file, singleFile);
83            root.getChildren().add(singleFile);
84          } else {
85            singleFile = new TreeItem<>(file.getName(), new
                 ImageView(this.file));
86            root.getChildren().add(singleFile);
87          }
88          root.setExpanded(true);
89        }
90      }
91    }
92  }
```

A.2 Variabili di simulazione

Codice A.2: Codice che mostra come è stata implementata una lista con un checkbox per ogni elemento. Inoltre è possibile notare come vengono catturati gli eventi di selezione e deselezione di un elemento.

```
1  public void setVariablesList() {
2   Loader fileYaml = null;
3   try {
4    fileYaml = new YamlLoader(new FileReader(
5                this.ctrlLeft.getPathFolder()
6                     + File.separator
7                     + this.pathYaml.getText()
8                          .replace("/", File.separator)));
9   } catch (FileNotFoundException e) {
10    L.error("Error loading simulation file.", e);
11  }
12  if (fileYaml != null) {
13    final ObservableList<String> vars =
14                      FXCollections.observableArrayList();
15    vars.addAll(fileYaml.getVariables().keySet());
16    if (this.variables.isEmpty()) {
17      for (final String s: vars) {
18        this.variables.put(s, false);
19      }
```

```
20      }
21      this.listYaml.setItems(vars);
22      this.listYaml.setCellFactory(
23        CheckBoxListCell.forListView(
24         new Callback<String, ObservableValue<Boolean>>() {
25        @Override
26        public ObservableValue<Boolean> call(
27                                          final String var) {
28          final BooleanProperty observable =
29                               new SimpleBooleanProperty();
30          if (variables.get(var)) {
31            observable.set(true);
32          }
33          observable.addListener(
34           (obs, wasSelected, isNowSelected) -> {
35            if (wasSelected && !isNowSelected) {
36               variables.put(var, false);
37            }
38            if (!wasSelected && isNowSelected) {
39               variables.put(var, true);
40            }
41          });
42          return observable;
43        }
44      }));
45   }
46  }
```

A.3 Salvataggio e caricamento dati

Codice A.3: Codice della classe `ProjectIOUtils`, dove viene mostrata l'implementazione del metodo `loadFrom`.

```
1  /**
2   * @param directory
3   *            The folder directory.
4   * @return Deserializing reader of Java.
5   */
6  public static Project loadFrom(final String directory) {
7   final Gson gson = new GsonBuilder()
8    .registerTypeAdapter(Project.class,
9                 new InstanceCreator<Project>() {
10      @Override
11      public Project createInstance(final Type type) {
12         return new Project(new File(directory));
13      }
14    })
```

```
15      .setPrettyPrinting()
16      .create();
17  final String actualPath = directory + File.separator + ".
        alchemist_project_descriptor.json";
18  if (new File(actualPath).exists()
19      && new File(actualPath).isFile()) {
20    try {
21      return gson.fromJson(new FileReader(actualPath),
22                           Project.class);
23    } catch (JsonSyntaxException
24             | JsonIOException
25             | FileNotFoundException e) {
26      throw new IllegalStateException(e);
27    }
28  } else {
29      return null;
30  }
31 }
```

Codice A.4: Codice della classe `ProjectIOUtils`, dove viene mostrata l'implementazione del metodo `saveTo`.

```
1  /**
2   * @param project
3   *          A entity Project.
4   * @param directory
5   *          A folder path.
6   */
7  public static void saveTo(final Project project,
8                            final String directory) {
9   if (new File(directory).exists()
10      && new File(directory).isDirectory()) {
11    try {
12      final Gson gson = new GsonBuilder()
13                              .setPrettyPrinting()
14                              .create();
15      Files
16        .write(gson.toJson(project),
17               new File(directory
18                 + File.separator
19                 + ".alchemist_project_descriptor.json"),
20               StandardCharsets.UTF_8);
21    } catch (IOException e) {
22      throw new IllegalStateException(e);
23    }
24  }
25 }
```

A.4 JSON schema del file di salvataggio

Codice A.5: JSON schema, che mostra come il file di salvataggio JSON deve essere strutturato al suo interno.

```
{
 "title": "Project",
 "description": "A project of Alchemist",
 "type": "object",
 "properties": {
  "simulation": {
   "description": "The relative path of the simulation file",
   "type": "string"
  },
  "endTime": {
   "description": "The end time of simulation",
   "type": "number"
  },
  "effect": {
   "description": "The relative path of effect file",
   "type": "string"
  },
  "output": {
   "description": "The output data",
   "type": "object",
   "properties": {
    "select": {
     "description": "If section output is selected",
     "type": "boolean"
    },
    "folder": {
     "description": "The relative path of the output folder",
     "type": "string"
    },
    "baseName": {
     "description": "The output file name",
     "type": "string"
    },
    "samplInterval": {
     "description": "The sampling interval",
     "type": "number"
    }
   }
  },
  "batch": {
   "description": "The batch data",
   "type": "object",
   "properties": {
    "select": {
```

```
45      "description": "If section batch is selected",
46      "type": "boolean"
47     },
48     "variables": {
49      "description": "The variables of simulation",
50      "type": "object",
51      "$ref": "#/definitions/var"
52     },
53     "thread": {
54      "description": "The number of threads using during the
           simulation",
55      "type": "integer"
56     }
57    }
58   },
59   "classpath": {
60    "description": "The list of relative path of libraries to
       add to classpath",
61    "type": "array",
62    "items": {
63      "type": "string"
64    }
65   }
66  },
67  "definitions": {
68   "var": {
69    "additionalProperties": false,
70    "type": "boolean"
71   }
72  }
73 }
```

A.5 Watcher

Codice A.6: Codice della classe `Watcher`, dove viene mostrato il metodo per la registrazione della directory al watcher.

```
1  /**
2   * @param path
3   *           A folder path of project.
4   */
5  public void registerPath(final String path) {
6   this.folderPath = path;
7   final Path dir = Paths.get(path);
8   recursiveRegistration(dir);
9  }
10
```

```
11
12 private void recursiveRegistration(final Path root) {
13  try {
14   Files.walkFileTree(root,
15             new SimpleFileVisitor<Path>() {
16    @Override
17    public FileVisitResult preVisitDirectory(final Path dir,
        final BasicFileAttributes attrs) {
18     try {
19      final WatchKey key =
20               dir.register(watcherServ,
21                   StandardWatchEventKinds.ENTRY_CREATE,
22                   StandardWatchEventKinds.ENTRY_DELETE,
23                   StandardWatchEventKinds.ENTRY_MODIFY);
24      keys.put(key, dir);
25     } catch (IOException e) {
26       L.error("Error register the folder path to
27            watcher. This is most likely a bug.", e);
28     }
29       return FileVisitResult.CONTINUE;
30     }
31    });
32  } catch (IOException e) {
33    L.error("There was an I/O error.", e);
34  }
35 }
```

Codice A.7: Codice della classe `Watcher`, dove viene mostrato il metodo per la cattura degli eventi.

```
1  @Override
2  public void run() {
3   while (this.isAlive) {
4    WatchKey key = null;
5    try {
6     key = this.watcherServ.poll(TIMEOUT,
7                                  TimeUnit.SECONDS);
8    } catch (InterruptedException e) {
9      L.error("Error watcher, because it was interrupted. This
         is most likely a bug.", e);
10   }
11   if (key != null) {
12    for (final WatchEvent<?> event : key.pollEvents()) {
13     final WatchEvent.Kind<?> kind = event.kind();
14      if (event.context() instanceof Path) {
15       final Path fileName = (Path) event.context();
16       if (StandardWatchEventKinds.ENTRY_MODIFY
17                                        .equals(kind)) {
18        if (fileName.toString()
```

```
19          .equals(".alchemist_project_descriptor.json")) {
20              refreshGrid();
21          } else if (this.ctrlCenter
22                          .getSimulationFilePath()
23                          .endsWith(fileName.toString())) {
24              refreshVariables();
25          } else {
26              refreshTreeView(this.folderPath);
27          }
28        } else if (StandardWatchEventKinds.ENTRY_CREATE
29                                            .equals(kind)) {
30          recursiveRegistration(resolvePath(key,
31                                        fileName));
32          if (this.folderPath.equals(FilenameUtils
33                      .getFullPathNoEndSeparator(
34                          resolvePath(key, fileName)
35                          .toFile().getAbsolutePath()))) {
36              refreshTreeView(this.folderPath);
37          }
38        } else if (StandardWatchEventKinds.ENTRY_DELETE
39                                            .equals(kind)) {
40          WatchKey keyToDelete = null;
41          final Path path = resolvePath(key, fileName);
42          for (final WatchKey w : keys.keySet()) {
43           if (keys.get(w).equals(path)) {
44              keyToDelete = w;
45           }
46          }
47          if (keyToDelete != null) {
48              refreshFolder(path);
49              keys.remove(keyToDelete);
50              keyToDelete.cancel();
51          } else {
52              refreshFile(path);
53          }
54          if (this.folderPath.equals(FilenameUtils
55                    .getFullPathNoEndSeparator(path
56                        .toFile().getAbsolutePath()))) {
57              refreshTreeView(this.folderPath);
58          }
59        } else {
60          throw new IllegalStateException("Unexpected event of
                kind " + kind);
61       }
62      }
63     }
64     key.reset();
65    }
66  }
```

```
67   if (!isAlive) {
68    try {
69      this.watcherServ.close();
70    } catch (IOException e) {
71      L.error("I/O error while closing of watcher.", e);
72    }
73   }
74  }
```

Codice A.8: Codice della classe `Watcher`, dove vengono mostrati i metodi delle operazioni da svolgere al verificarsi di un certo evento.

```
1  private void refreshTreeView(final String path) {
2    Platform.runLater(new Runnable() {
3      @Override
4      public void run() {
5        ctrlLeft.setTreeView(new File(path));
6      }
7    });
8  }
9
10 private void refreshGrid() {
11   Platform.runLater(new Runnable() {
12     @Override
13     public void run() {
14       ctrlCenter.setField();
15     }
16   });
17 }
18
19 private void refreshFolder(final Path path) {
20   Platform.runLater(new Runnable() {
21     @Override
22     public void run() {
23       ctrlCenter.setFolderAfterDelete(path);
24     }
25   });
26 }
27
28 private void refreshFile(final Path path) {
29   Platform.runLater(new Runnable() {
30     @Override
31     public void run() {
32       ctrlCenter.setFileAfterDelete(path);
33     }
34   });
35 }
36
37 private void refreshVariables() {
```

```
38    Platform.runLater(new Runnable() {
39      @Override
40      public void run() {
41        ctrlCenter.setVariablesList();
42      }
43    });
44  }
```

A.6 Gestione del classpath

Codice A.9: Codice che mostra l'implementazione del metodo `addPath`, utilizzato per aggiungere una nuova libreria al classpath.

```
1  private boolean addPath(final String path) {
2    URL url = null;
3    try {
4      url = new File(path).toURI().toURL();
5    } catch (MalformedURLException e) {
6      L.error("Error during the construction of the URL.", e);
7    }
8    if (url != null) {
9      final ClassLoader systemClassLoader = ClassLoader.
           getSystemClassLoader();
10     if (systemClassLoader instanceof URLClassLoader) {
11       final URLClassLoader urlClassLoader = (URLClassLoader)
             systemClassLoader;
12       final Class<?> urlClass = URLClassLoader.class;
13       final Method method;
14       try {
15         method = urlClass.getDeclaredMethod("addURL", new
               Class[]{URL.class});
16         method.setAccessible(true);
17         try {
18           method.invoke(urlClassLoader, new Object[]{url});
19           return true;
20         } catch (IllegalAccessException e) {
21           L.error("Error because the method is inaccessible."
                 , e);
22         } catch (IllegalArgumentException e) {
23           L.error("Error because the objects are not an
                 instance of the method.", e);
24         } catch (InvocationTargetException e) {
25           L.error("Error during invoke of method.", e);
26         }
27       } catch (NoSuchMethodException e) {
28         L.error("Error because no methods matching with \"
               addURL\".", e);
```

```
29        } catch (SecurityException e) {
30          L.error("Error because there is a security manager.",
                e);
31        }
32      }
33    }
34    return false;
35  }
```

Codice A.10: Codice che mostra l'implementazione del metodo `removePath`, utilizzato per rimuovere una libreria precedentemente aggiunta al classpath.

```
1   private boolean removePath(final String path) {
2     URL url = null;
3     try {
4       url = new File(path).toURI().toURL();
5     } catch (MalformedURLException e) {
6       L.error("Error during the construction of the URL.", e);
7     }
8     if (url != null) {
9       final ClassLoader systemClassLoader = ClassLoader
10                          .getSystemClassLoader();
11      if (systemClassLoader instanceof URLClassLoader) {
12        final URLClassLoader urlClassLoader
13                  = (URLClassLoader) systemClassLoader;
14        final Class<?> urlClass = URLClassLoader.class;
15        final Field ucpField;
16        try {
17          ucpField = urlClass.getDeclaredField("ucp");
18          ucpField.setAccessible(true);
19          try {
20            URLClassPath field = (URLClassPath) ucpField
21                                    .get(urlClassLoader);
22            if (field instanceof URLClassPath) {
23              try {
24                final URLClassPath ucp = field;
25                final Class<?> ucpClass
26                                    = URLClassPath.class;
27                final Field urlsField = ucpClass
28                                .getDeclaredField("urls");
29                urlsField.setAccessible(true);
30                final Stack<?> fieldStack
31                       = (Stack<?>) urlsField.get(ucp);
32                if (fieldStack instanceof Stack) {
33                  final Stack<?> urls = fieldStack;
34                  urls.remove(url);
35                  return true;
36                }
37              } catch (IllegalArgumentException e) {
```

```
38                L.error("Error because the objects are not an
                      instance of the field.", e);
39              } catch (IllegalAccessException e) {
40                L.error("Error because the method is
                      inaccessible.", e);
41              }
42            }
43          } catch (IllegalArgumentException e1) {
44            L.error("Error because the objects are not an
                  instance of the field.", e1);
45          } catch (IllegalAccessException e1) {
46            L.error("Error because the field is inaccessible.",
                  e1);
47          }
48        } catch (NoSuchFieldException e) {
49          L.error("Error because no fields matching with \"ucp
                \".", e);
50        } catch (SecurityException e) {
51          L.error("Error because there is a security manager.",
                e);
52        }
53      }
54    }
55    return false;
56  }
```

Bibliografia

[1] Nathaniel Ayewah et al. «Using static analysis to find bugs». In: *IEEE software* 25.5 (2008), pp. 22–29.

[2] Oren Ben-Kiki, Clark Evans e Brian Ingerson. «YAML Ain't Markup Language (YAML™) Version 1.1». In: *yaml. org, Tech. Rep* (2005).

[3] Jim Clarke, Jim Connors e Eric J Bruno. *JavaFX: Developing Rich Internet Applications*. Pearson Education, 2009.

[4] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627. `http://www.rfc-editor.org/rfc/rfc4627.txt`. RFC Editor, 2006. URL: `http://www.rfc-editor.org/rfc/rfc4627.txt`.

[5] Michael J Flynn e Ehsan Samei. «Experimental comparison of noise and resolution for 2k and 4k storage phosphor radiography systems.» In: *Medical physics* 26.8 (1999), pp. 1612–1623.

[6] Gabriele Graffieti. «Progettazione e implementazione di una incarnazione biochimica per il simulatore Alchemist». Tesi di laurea. Università di Bologna, 2016.

[7] Panagiotis Louridas. «Static code analysis». In: *IEEE Software* 23.4 (2006), pp. 58–61.

[8] Martin C. Maguire. «A review of human factors guidelines and techniques for the design of graphical human-computer interfaces». In: *Computers & Graphics* 9.3 (1985), pp. 221–235. DOI: `10.1016/0097-8493(85)90049-4`. URL: `http://dx.doi.org/10.1016/0097-8493(85)90049-4`.

[9] Matthew McCullough e Tim Berglund. *Building and Testing with Gradle*. "O'Reilly Media, Inc.", 2011.

[10] Sara Montagna, Danilo Pianini e Mirko Viroli. «A model for drosophila melanogaster development from a single cell to stripe pattern formation». In: *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*. 2012, pp. 1406–1412. DOI: 10.1145/2245276.2231999. URL: http://doi.acm.org/10.1145/2245276.2231999.

[11] Oracle. *JavaFX. Getting Started with JavaFX*. Ver. 8. 2014.

[12] Oracle. *JavaFX. Working with JavaFX UI Components*. Ver. 8. 2014.

[13] T.C. O'Rourke et al. *Graphical user interface*. US Patent 5,349,658. 1994. URL: https://www.google.com/patents/US5349658.

[14] Danilo Pianini. *Software development made serious*. Slide. Università di Bologna, 2016.

[15] Danilo Pianini, Sara Montagna e Mirko Viroli. «Chemical-oriented simulation of computational systems with ALCHEMIST». In: *J. Simulation* 7.3 (2013), pp. 202–215. DOI: 10.1057/jos.2012.27. URL: http://dx.doi.org/10.1057/jos.2012.27.

[16] Danilo Pianini, Mirko Viroli e Jacob Beal. «Protelis: practical aggregate programming». In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*. 2015, pp. 1846–1853. DOI: 10.1145/2695664.2695913. URL: http://doi.acm.org/10.1145/2695664.2695913.

[17] Daniel J. Power. «Decision Support Systems: A Historical Overview». In: *Handbook on Decision Support Systems 1: Basic Themes*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 121–140. ISBN: 978-3-540-48713-5. DOI: 10.1007/978-3-540-48713-5_7. URL: http://dx.doi.org/10.1007/978-3-540-48713-5_7.

[18] Mirko Viroli. *Pattern di Progettazione OO*. Slide. Università di Bologna, 2014.

[19] Mirko Viroli, Roberto Casadei e Danilo Pianini. «Simulating Large-scale Aggregate MASs with Alchemist and Scala». In: *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, September 11-14, 2016*. 2016, pp. 1495–1504. DOI: 10.15439/2016F407. URL: http://dx.doi.org/10.15439/2016F407.

[20] Franco Zambonelli et al. «Self-aware Pervasive Service Ecosystems». In: *Proceedings of the 2nd European Future Technologies Conference and Exhibition, FET 2011, Budapest, Hungary, May 4-6, 2011*. 2011, pp. 197–199. DOI: 10.1016/j.procs.2011.09.006. URL: http://dx.doi.org/10.1016/j.procs.2011.09.006.