

**Progettazione object oriented
di un'interfaccia grafica JavaFX
per il simulatore Alchemist**

Tesi in Programmazione ad Oggetti

Relatore:
Prof. Mirko Viroli

Correlatore:
Ing. Danilo Pianini

Presentata da:
Niccolò Maltoni

Parole chiave

Alchemist

Simulatore

JavaFX

Interfaccia grafica

Effetto

Sommario

Lo scopo di questa tesi verte intorno allo studio del simulatore Alchemist al fine di progettare un’interfaccia 2D potenziata per l’ambiente grafico relativo alla simulazione. La nuova interfaccia permette di interagire con la simulazione a tempo di esecuzione e di vedere chiaramente rappresentate informazioni su di essa; in particolare, è supportata una struttura modulare di effetti che per rendere ancora più facilmente osservabili determinate entità del sistema ed eventuali loro proprietà. Si è scelto di mantenere un’interfaccia il più possibile *user-friendly*, mantenendo un design più simile ai simulatori a scopo videoludico per favorire l’utilizzo da parte di utenti inesperti. La libreria grafica utilizzata per l’implementazione è stata JavaFX.

La seguente trattazione è strutturata su tre capitoli: nel capitolo 1 a pagina 1 viene introdotto il contesto nel quale il lavoro descritto nella tesi ha preso parte, introducendo il simulatore Alchemist, la sua interfaccia grafica classica e il framework JavaFX; nel capitolo 2 a pagina 16 si espone l’intero contributo fornito al progetto, analizzando singolarmente le fasi di analisi dei requisiti, design e progettazione e in ultimo implementazione della nuova interfaccia; infine, il capitolo 3 a pagina 31 analizza i risultati ottenuti, interpretandoli anche in ottica di miglioramenti futuri.

Indice

Sommario	iii
1 Introduzione	1
1.1 Alchemist	1
1.1.1 Introduzione ad Alchemist	1
1.1.2 Astrazioni e modello di Alchemist	2
1.1.3 Interfaccia utente classica	4
Esperienza utente	5
Swing	6
Gli effetti e l'interfaccia Effect	7
1.2 JavaFX	8
1.2.1 Introduzione al framework JavaFX	8
1.2.2 Architettura del framework JavaFX	9
1.2.3 Struttura di una Applicazione JavaFX	10
1.2.4 Vantaggi di JavaFX su Swing	13
1.3 Interfaccia JavaFX per Alchemist: motivazioni	14
2 Contributo	16
2.1 Analisi dei requisiti	16
2.1.1 Requisiti funzionali	16
Rappresentazione dell'ambiente di simulazione	16
Gestione degli effetti	17
Effetti standard per nodi e collegamenti	17
Interazione con simulazione e ambiente rappresentato	17
Rappresentazione di ambienti con mappa	17
2.1.2 Requisiti non funzionali	19

JavaFX	19
Performance	19
Supporto Hi-DPI	19
Serializzazione <i>Human-readable</i>	19
2.2 Fonti d’ispirazione	19
2.2.1 Simulatori a scopo videoludico	20
SimCity	20
Universe Sandbox	21
2.2.2 Material Design	22
2.3 Design dell’interfaccia	22
2.4 Progettazione	24
2.4.1 L’architettura degli effetti	24
I singoli effetti e l’interfaccia EffectFX	24
I gruppi di effetti e l’interfaccia EffectGroup	25
Caricamento, salvataggio e modifica di gruppi di effetti	26
2.4.2 La struttura dei drawer e le proprietà osservabili	27
2.4.3 Nodi grafici come monitor per la simulazione	27
2.4.4 Costruzione e avvio dell’interfaccia	28
2.5 Dettagli implementativi	28
2.5.1 La barra inferiore	28
2.5.2 Drawer, liste e celle	29
2.5.3 Librerie esterne utilizzate	29
ControlsFX	30
GSON	30
JFoenix	30
jIconFont	30
2.5.4 Strumenti utilizzati	30
Qualità del codice e controllo del software	30
Controllo di versione	30
Automazione dello sviluppo e integrazione continua	30
Ambiente di sviluppo integrato	30
3 Conclusioni	31
3.1 Risultati	31
3.2 Lavori futuri	31

A Codice relativo all'interfaccia classica	32
A.1 L'interfaccia <code>Effect</code>	32
B Codice relativo al contributo	34
B.1 L'interfaccia <code>EffectFX</code>	34
B.2 L'interfaccia <code>EffectGroup</code>	35
B.3 Implementazioni di <code>EffectFX</code> e Proprietà serializzabili	38
Bibliografia	39
Ringraziamenti	42

Capitolo 1

Introduzione

1.1 Alchemist

Alchemist [1, 31] è un meta-simulatore estendibile completamente *open-source* che esegue su *Java Virtual Machine* (JVM), nato all'interno dell'Università di Bologna e distribuito su licenza GNU GPLv3+ con *linking exception*; il codice è reperibile su GitHub¹, dove chiunque fosse interessato può collaborare sviluppando nuove estensioni, migliorando funzionalità esistenti e risolvendo possibili bug.

1.1.1 Introduzione ad Alchemist

In generale, una *simulazione* [4] è una riproduzione del modo di operare di un sistema o un processo del mondo reale nel tempo. L'imitazione del processo del mondo reale è detta *modello*; esso risulta essere una riproduzione più o meno semplificata del mondo reale, che viene aggiornata ad ogni passo di esecuzione della simulazione.

Alchemist rientra nell'archetipo dei simulatori ad eventi discreti (DES) [3, 13]: gli eventi sono strettamente ordinati e vengono eseguiti uno alla volta, mentre il tempo viene fatto avanzare parallelamente ad ogni passo (detto *tick*). L'idea dietro al progetto è quello di riuscire ad avere un framework di simulazione il più possibile generico, in grado di simulare sistemi di tipologia e complessità diverse, mantenendo le prestazioni dei simulatori non generici (come ad esempio quelli impiegati in ambito chimico [18]).

Per perseguire questo obiettivo, la progettazione dell'algoritmo è partita dallo studio del lavoro di Gillespie del 1977 [19] e di altri scienziati nell'ambito della simulazione chimica.

¹<https://github.com/AlchemistSimulator/Alchemist>

Nonostante siano presenti algoritmi in grado di eseguire un numero di reazioni addirittura in tempo costante, la scelta dell'algoritmo è infine ricaduta su una versione migliorata dell'algoritmo SSA di Gillespie, il Next Reaction Method [17] di Gibson e Bruck: ad ogni passo di simulazione, esso è in grado di selezionare la reazione successiva in tempo costante e richiede un tempo logaritmico per aggiornare le strutture dati interne al termine dell'esecuzione dell'evento.

1.1.2 Astrazioni e modello di Alchemist

Il modello di astrazione di Alchemist è ispirato dal lavoro della comunità scientifica nell'ambito dei simulatori a scopo di ricerca chimica e, dunque, ne riprende la nomenclatura, seppur con alcune libertà per favorire le entità (visibile in Figura 1.2 a pagina 4) su cui lavora sono le seguenti:

Molecola Una *Molecola* rappresenta il nome dato ad un particolare dato all'interno di un *Nodo*, del quale ne astrae parte dello stato.

Un parallelismo con la programmazione imperativa vedrebbe la *Molecola* come un'astrazione del nome di una variabile.

Concentrazione La *Concentrazione* di una *Molecola* è il valore associato alla proprietà rappresentata dalla *Molecola*.

Mantenendo il parallelismo con la programmazione imperativa, la *Concentrazione* rappresenterebbe il valore della variabile.

Node Il *Nodo* è un contenitore di *Molecole* e *Reazioni* che risiede all'interno di un *Ambiente* e che astrae una singola entità.

Environment L'*Ambiente* è l'astrazione che rappresenta lo spazio nella simulazione ed è l'entità che contiene i nodi.

Esso è in grado di fornire informazioni in merito alla posizione dei *Nodi* nello spazio, alla distanza tra loro e al loro vicinato; optionalmente, l'*Ambiente* può offrire il supporto allo spostamento dei *Nodi*.

Linking rule La *Regola di collegamento* è la funzione dello stato corrente dell'*Ambiente* che associa ad ogni *Nodo* un *Vicinato*.

Vicinato Un *Vicinato* è un'entità costituita da un *Nodo* detto “centro” e da un insieme di altri *Nodi* (i “vicini”).

L’astrazione dovrebbe avere un’accezione il più possibile generale e flessibile, in modo da poter modellare qualsiasi tipo di legame di vicinato, non solo spaziale.

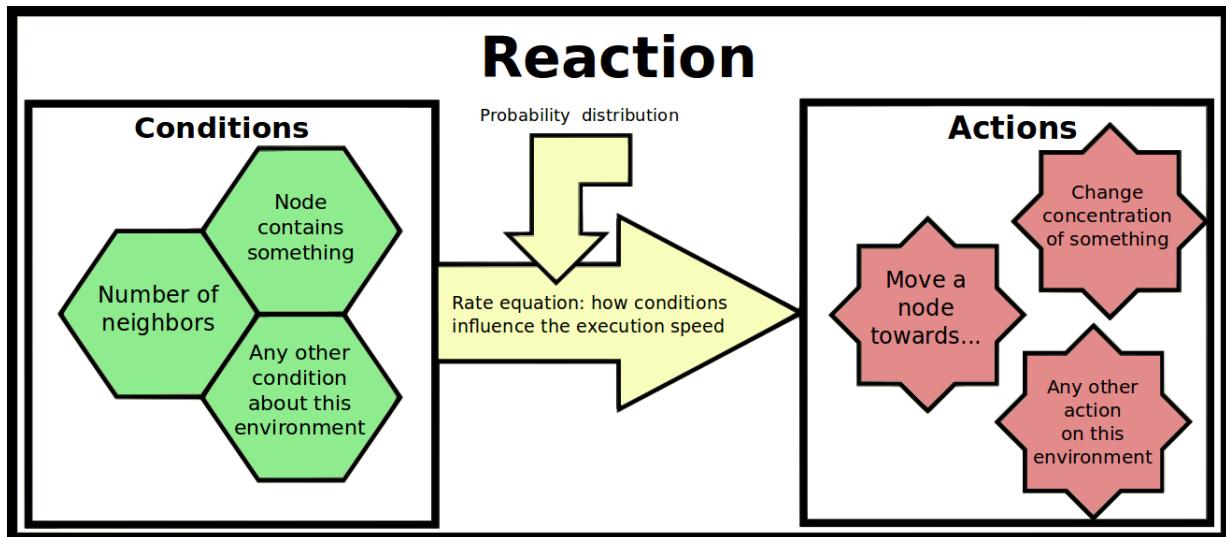


Figura 1.1: La figura, rivisitata da quella disponibile sul sito ufficiale [1], offre una rappresentazione grafica della *Reazione*.

Reaction Il concetto di *Reazione* è da considerarsi molto più elaborato di quello utilizzato in chimica: in questo caso, si può considerare com un insieme di *Condizioni* sullo stato del sistema, che qualora dovessero risultare vere innescherebbero l’esecuzione di un insieme di *Azioni*.

Una *Reazione* (di cui è possibile vederne una rappresentazione grafica in Figura 1.1) è dunque un qualsiasi evento che può cambiare lo stato dell’*Ambiente* e si compone di un insieme di condizioni, una o più azioni e una distribuzione temporale.

La frequenza di accadimento può dipendere da:

- Un tasso statico;
- Il valore di ciascuna *Condizione*;
- Una equazione che combina il tasso statico e il valore delle *Condizioni*, restituendo un “tasso istantaneo”;
- Una distribuzione temporale.

Ogni *Nodo* è costituito da un insieme (anche vuoto) di *Reazioni*.

Condition Una *Condizione* è una funzione che associa un valore numerico e un valore booleano allo stato corrente di un *Ambiente*.

Action Un' *Azione* è una procedura che provoca una modifica allo stato dell'*Ambiente*.

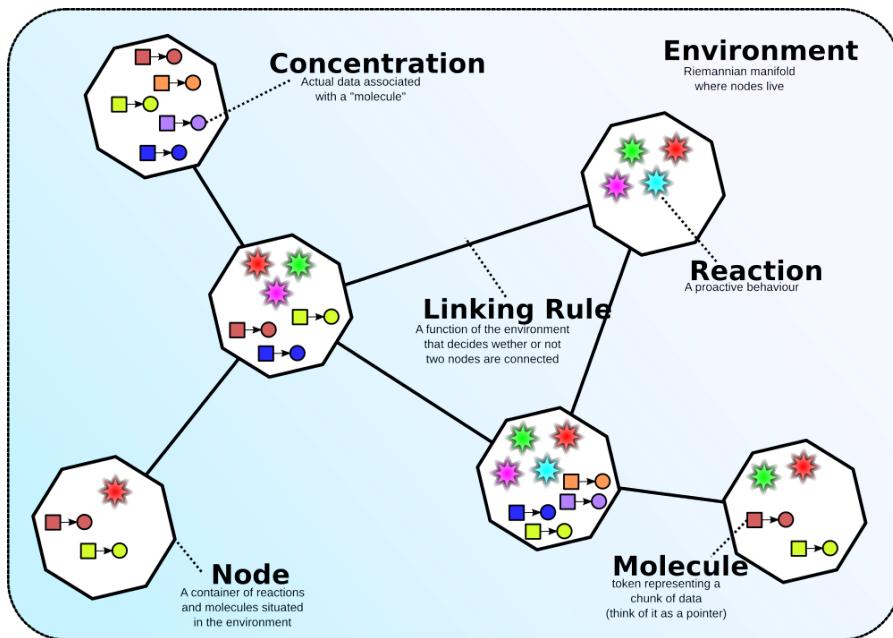


Figura 1.2: La figura, presa dal sito ufficiale [1], offre una rappresentazione grafica delle diverse entità. All'interno di un ambiente, che modella il sistema, si trovano i nodi connessi tra loro attraverso dei collegamenti; ogni nodo è composto da reazioni e molecole, ognuna delle quali ha associata una concentrazione.

1.1.3 Interfaccia utente classica

L'architettura di Alchemist è progettata con paradigma *Model-View-Controller* (MVC) [23], di conseguenza la suddivisione tra componente grafica (*View*) e il blocco “logico” composto da *Model* e *Controller* è netta. Questa distinzione è evidente anche per quanto riguarda l'utilizzo pratico del software: una simulazione su Alchemist può venire lanciata da terminale, senza che alcuna interfaccia grafica sia necessaria per tutta la durata del periodo di esecuzione, oppure essere inizializzata, lanciata e controllata in tempo reale dalla sua interfaccia grafica.

Per lo scopo di questa tesi, tratteremo esclusivamente della GUI.

Esperienza utente

Un’interfaccia grafica (detta anche GUI, *graphical user interface* [30]) è l’insieme dei componenti grafici con i quali l’utente può interagire per impartire comandi ad un programma del computer, che si contrappone ad un altro metodo di interazione, l’interfaccia a riga di comando (o CLI, *Command Line Interface*).

L’interfaccia grafica è stata ideata negli anni ‘80 a partire da un’esigenza di maggiore usabilità rispetto dalla riga di comando, derivante soprattutto dall’affermarsi degli studi di usabilità [29] e di ergonomia cognitiva di quel periodo.

Più ampio e moderno è invece il concetto di esperienza utente [20] (spesso abbreviata in UX, *User eXperience*): l’ISO 9241-210 [10] al definisce come “le percezioni e le reazioni di un utente che derivano dall’uso o dall’aspettativa d’uso di un prodotto, sistema o servizio”. Di fatto, essa descrive la reazione dell’utente di fronte all’interazione con il programma o lo strumento in base a tre dimensioni:

- *Dimensione pragmatica*: funzionalità e usabilità del sistema;
- *Dimensione estetica/edonistica*: piacevolezza estetica, emotiva e ludica del sistema;
- *Dimensione simbolica*: attributi sociali, forza del brand, identificazione.

L’usabilità, invece, fa riferimento unicamente ai soli aspetti pragmatici (la capacità di svolgere un compito con efficienza ed efficacia).

L’interfaccia utente classica di Alchemist è caratterizzata da un’usabilità appena sufficiente, funzionale alle necessità di un utilizzatore esperto, ma non adeguato a fornire un’esperienza completa e *user-friendly* ad un utente “standard”.

Grazie a contributi recenti [6], la GUI ha subito un parziale rinnovamento, limitato alla parte di ambiente integrato che accoglie l’utilizzatore che stia lanciando il simulatore senza una simulazione specificata; questa parte non è oggetto del lavoro illustrato in questa tesi. Al contrario, è interessante analizzare lo stato dell’interfaccia relativa all’ambiente di esecuzione della simulazione.

La criticità principale, che va a minare non solo il livello di esperienza utente, ma anche il concetto di usabilità “classico”, è evidente nella non intuitività dei controlli: come è possibile vedere in Figura 1.3 nella pagina successiva, non sono presenti bottoni di interazione per, ad esempio, avviare o fermare la simulazione o per cambiare la modalità di interazione con la zona in cui viene rappresentato l’ambiente; questo perché molte possibilità di controllo

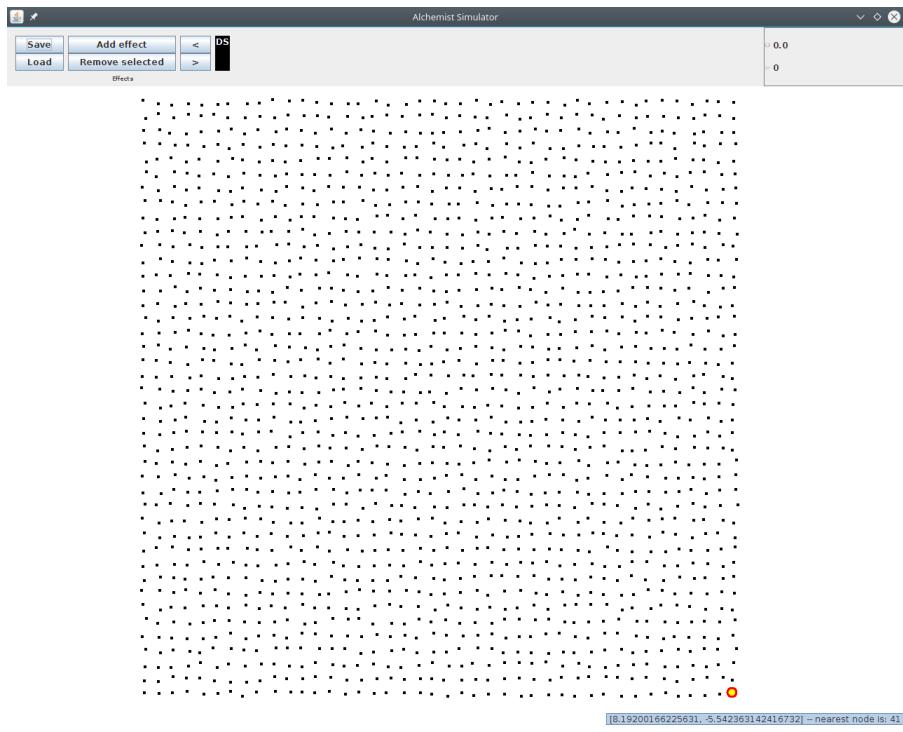


Figura 1.3: Vista principale di una simulazione con l’interfaccia classica

sono limitate a scorciatoie da tastiera non modificabili e non descritte altrove se non nella documentazione.

Un’ultima criticità che esula dal contesto pratico, ma che rientra appieno nel contesto estetico importante per una buona UX è, appunto l’aspetto grafico: l’intera interfaccia di simulazione è implementata sfruttando le impostazioni di base del framework Swing, senza alcun tipo di personalizzazione al *look’n’feel* che potesse identificare uno stile originale di Alchemist, né che lo allineasse alle direttive di un design grafico ben definito (come il Material Design [24] di Google o Modern UI e Fluent Design System [14] di Microsoft), né che si adattasse al *look’n’feel* generico del sistema operativo.

Swing

Come detto, Alchemist utilizzava Swing come strumento per implementare l’interfaccia grafica. Java Swing è un framework per lo sviluppo di GUI in Java, parte delle *Java Foundation Classes* (JFC) insieme ad AWT (*Abstract Window Toolkit*) e *Java 2D*.

Come è possibile vedere in Figura 1.4 a fronte, la libreria sfrutta i componenti forniti da AWT, mettendo a disposizione nuovi componenti in grado di risolvere diverse debolezze del

precedente standard grafico per il linguaggio di Oracle:

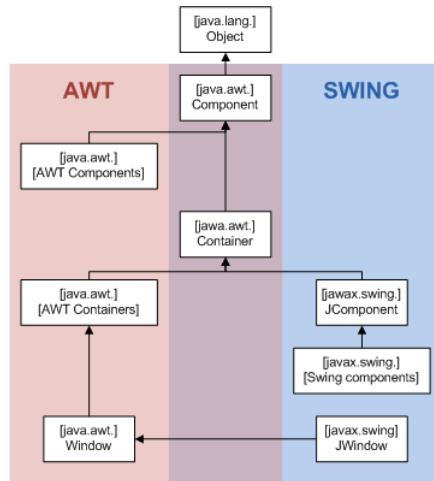


Figura 1.4: Struttura delle classi di Swing e AWT, by Jakub Závěrka (Jakub Závěrka - own work) [Public domain], via Wikimedia Commons

- Swing è molto più facilmente estendibile e rende possibile un controllo della presentazione grafica dei componenti (il *look'n'feel*) trasparente, non necessitando più di classi specifiche per ogni aspetto grafico.
- I componenti forniti da Swing permettono inoltre di realizzare un’interfaccia più leggera di quella di AWT: essa sfrutta infatti le API fornite da Java 2D, anziché chiamare il *toolkit* di interfacce native del sistema operativo; nel contempo, appoggiandosi al container di AWT, sfrutta l’accesso al framework di gestione delle GUI fornito dall’OS, traducendo gli eventi specifici dell’OS in eventi Java disaccoppiati dalla piattaforma su cui gira la JVM, semplificando la gestione da parte dello sviluppatore.
- Swing rende più semplice appoggiarsi al pattern MVC per implementare software con GUI, separando le classi di modello da quelle grafiche e di controllo.

Gli effetti e l’interfaccia Effect

Una parte consistente della visualizzazione di una simulazione di Alchemist, nell’interfaccia classica come in quella attuale, è costituita dagli effetti.

Un *effetto* in Alchemist è una rappresentazione grafica di “qualcosa” nell’ambiente; costituisce di fatto una modalità semplificata per l’utente di cogliere quanto accade nella simulazione.

L’interfaccia Java che implementa questo tipo di astrazione prima del lavoro svolto con questa tesi è la classe `Effect` (codice in Appendice A.1). L’effetto è in grado di rappresentare qualsiasi proprietà di un nodo dato; è concepito come un oggetto serializzabile, in modo da semplificare il salvataggio e il caricamento di intere rappresentazioni tramite la serializzazione di collezioni di essi.

1.2 JavaFX

Nel mese di maggio del 2007 alla conferenza annuale JavaOne, Sun Microsystems annuncia JavaFX Script (chiamato anche F3, *Form Follows Function*), un DSL (*Domain Specific Language*, linguaggio di dominio specifico) pensato per lo sviluppo di interfacce grafiche di Rich Internet Applications [28], e JavaFX Mobile, un sistema software per dispositivi mobili basato su Java e ispirato all’allora neonato iPhone, che avrebbe avuto come cavallo di battaglia la possibilità di sviluppare app mobile in grado di condividere codice e asset grafici con le controparti desktop e web, semplificando lo sviluppo di ecosistemi strutturati.

Incluso nella versione 1.0 del pacchetto JavaFX rilasciato nel dicembre del 2008, JavaFX Script verrà però abbandonato da Oracle (che nel frattempo aveva acquisito Sun Microsystems) meno di 2 anni dopo, in contemporanea con l’ampliamento della disponibilità delle JavaFX API agli altri linguaggi disponibili per JVM; anche JavaFX Mobile, con l’avvento di OS mobili moderni come Android e iOS, può considerarsi deprecato.

JavaFX continua invece lo sviluppo come framework per la gestione di interfacce grafiche per Java ed altri linguaggi JVM-compatibili, andando di fatto a sostituire Swing e AWT.

In questo capitolo si intende analizzare il framework e le sue funzionalità fino alla versione utilizzata nella stesura del codice, nonché l’ultima versione stabile all’atto di inizio del lavoro illustrato in questa tesi: JavaFX 8.

1.2.1 Introduzione al framework JavaFX

La prima versione di JavaFX ad abbandonare JavaFX Script e JavaFX Mobile, con i quali il framework era nato, per andare ad affiancarsi a Swing è la versione 2.0, distribuita parzialmente su licenza *open-source* verso la fine del 2011. Essa introduceva un nuovo linguaggio XML dichiarativo, l’FXML, in grado di fornire una struttura grafica all’applicazione coinvolgendo minimamente il codice Java, oltre a migliorare il supporto *multi-thread*. Con le successive versioni 2.1 e 2.2, rilasciate nell’arco del 2012, viene esteso il supporto a MacOS e Linux.

La prima versione ad essere parte del JRE/JDK è JavaFX 8, rilasciata il 18 marzo 2014 insieme a Java 8; essa diventa di fatto la nuova libreria di riferimento per lo sviluppo di applicazioni grafiche per ambiente JVM.

Essa si presenta come fortemente orientata verso i pattern di progettazione *Model-View-Controller* e *Model-View-Presenter*; la suddivisione infatti è netta:

- La *componente visiva* è definita su file di markup FXML, logicamente separati da qualsiasi componente Java che non siano le loro classi *Controller*; anche la *presentazione*, definibile attraverso fogli di stile CSS, è indipendente dalle altre componenti Java e XML e può essere anche sostituita a tempo di esecuzione senza difficoltà;
- Il *controllo* dell'applicazione è circoscritto a classi Java specifiche, che vengono associate al caricamento del documento di markup corrispondente; per design sono facilmente sostituibili da differenti implementazioni progettate per interagire con gli oggetti che il parser di JavaFX riconosce nel file FXML;
- In una implementazione che sfrutti appieno gli strumenti messi a disposizione del framework, per design il *modello* non viene coinvolto dalle suddette componenti e resta dunque distaccato dalle suddette componenti.

Oltre al già citato miglioramento per quanto riguarda il *look'n'feel* (che ora può vantare la semplificazione data dai fogli di stile CSS), un ulteriore flessibilità grafica è il supporto Hi-DPI, che permette alle GUI di adattarsi a qualsiasi risoulzione di schermo senza comprometterne la qualità.

1.2.2 Architettura del framework JavaFX

Come è possibile osservare in Figura 1.5 nella pagina successiva, l'architettura interna di JavaFX è costituita da diversi livelli, ciascuno dei quali sfrutta le funzionalità messe a disposizione dai livelli inferiori per offrire nuove API ai livelli superiori e allo sviluppatore finale.

Il livello più elevato per la costruzione di una applicazione JavaFX è il grafo delle scene (*Scene Graph*): esso ospita un albero gerarchico di nodi, ciascuno dei quali rappresentante un elemento visivo dell'interfaccia utente. Questo livello si occupa anche di intercettare gli input e di mettere a disposizione le JavaFX API pubbliche.

Un singolo elemento del grafo delle scene è chiamato *nodo*. Ogni nodo possiede un ID, una classe di stile e un volume delimitato; fatta eccezione per il nodo radice, ogni nodo

possiede un solo nodo genitore e può essere a sua volta genitore di uno o più altri nodi. Su ogni nodo possono essere definiti effetti grafici (come blur e ombre) e livello di opacità, nonché stati specifici per l'applicazione e comportamenti in caso di eventi specifici.

Il livello subito inferiore allo *Scene Graph* è costituito dal *JavaFX Graphics System* (in Figura 1.5 è rappresentato dagli elementi in azzurro), che attraverso il *Quantum Toolkit* e *Prism* mette a disposizione funzionalità più a basso livello per rappresentazioni 2D e 3D.

I processi di *Prism* si occupano del rendering; possono eseguire sia con accelerazione hardware che senza, ed effettuano sia rendering 2D che 3D. Attraverso questi processi vengono eseguite le rasterizzazioni e i rendering di tutti i grafi delle scene.

Il *Quantum Toolkit* collega invece *Prism* al *Glass Windowing Toolkit* e gestisce le regole di threading per rendering e gestione degli eventi.

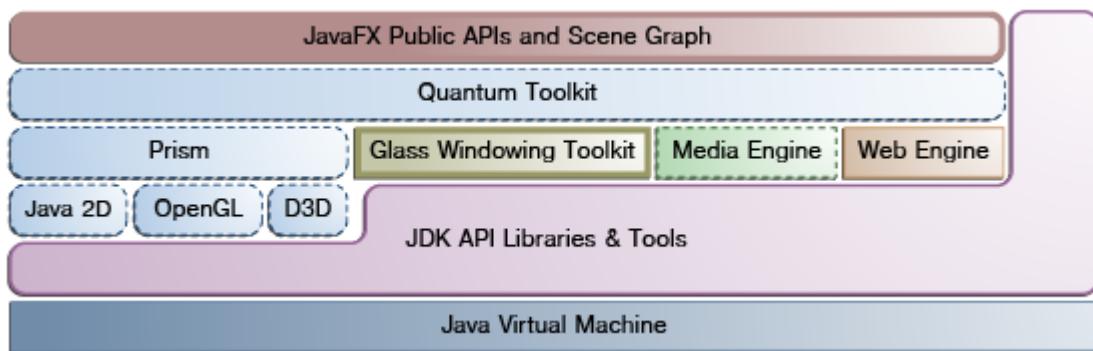


Figura 1.5: La figura, presa dalla documentazione ufficiale di Oracle [21], rappresenta i diversi livelli che caratterizzano il framework JavaFX

Il terzo livello è costituito dal sopra citato *Glass Windowing Toolkit*, che rappresenta il livello più basso del *JavaFX graphics stack*. Esso si occupa di gestire i servizi nativi forniti dai sistemi operativi per la gestione delle finestre e delle code degli eventi; costituisce la parte *platform-dependent* di JavaFX.

Media Engine e *Web Engine* si occupano del supporto per i file multimediali audiovisivi e per i linguaggi web.

1.2.3 Struttura di una Applicazione JavaFX

JavaFX fornisce le classi di base per strutturare una applicazione completa, delimitando linee guida ben specifiche nella suddivisione della struttura.

La classe principale per una applicazione JavaFX deve estendere da `javafx.application.Application`. Il metodo `start()` è il punto di ingresso principale: lanciando una

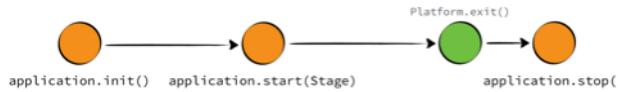


Figura 1.6: La figura rappresenta il ciclo di vita di una applicazione JavaFX

JavaFX Application attraverso il metodo `launch()` della classe di supporto `javafx.application.Platform`, verrà inizializzato il framework e poi verranno chiamati in ordine i metodi `init()` e `start(javafx.stage.Stage)`, che rappresentano di fatto il ciclo di inizializzazione ideale dell'applicazione. Il metodo `main()` non deve essere necessariamente implementato perché l'applicazione possa essere avviata: esso viene infatti creato da `JavaFX Packager tool` durante l'inserimento del *JavaFX Launcher* nel file JAR.

Il layout con cui un'applicazione JavaFX è costituita a livello grafico si struttura gerarchicamente su tre sezioni principali, visibili in Figura 1.7 nella pagina seguente:

Stage In JavaFX, una finestra è astratta tramite la classe `javafx.stage.Stage`: letteralmente “*palcoscenico*”, è il contenitore di livello più elevato e funge da “guscio esterno” per ogni altro componente grafico e pannello.

Lo **Stage** primario viene costruito dalla piattaforma all'atto di avvio, ma ulteriori **Stage** possono venire costruiti durante l'esecuzione, purché attraverso il *JavaFX Application Thread*.

Scene Il contenuto dello **Stage** è rappresentato dalla classe `javafx.scene.Scene`, che è a sua volta contenitore di ogni nodo appartentente a quello specifico *scene graph*.

Pane Come già affermato nella Sezione 1.2.1 a pagina 8, ogni elemento appartenente ad una *scena* è detto *nodo*. Il terzo livello è costituito da un particolare tipo di nodo, detto *pannello* (modellato dalla classe `javafx.scene.layout.Pane`), che costituisce il contenuto di una scena e gestisce la disposizione dei nodi al suo interno sullo schermo; è possibile costruire una struttura gerarchica inserendo all'interno di un pannello radice altri pannelli.

Node La classe `javafx.scene.Node` modella ogni singolo componente dello *Scene Graph*, dai già citati pannelli ai bottoni, canvas, campi di testo e così via.

In Figura 1.8 nella pagina seguente è possibile vedere una rappresentazione gerarchica delle diverse tipologie di *nodo*.

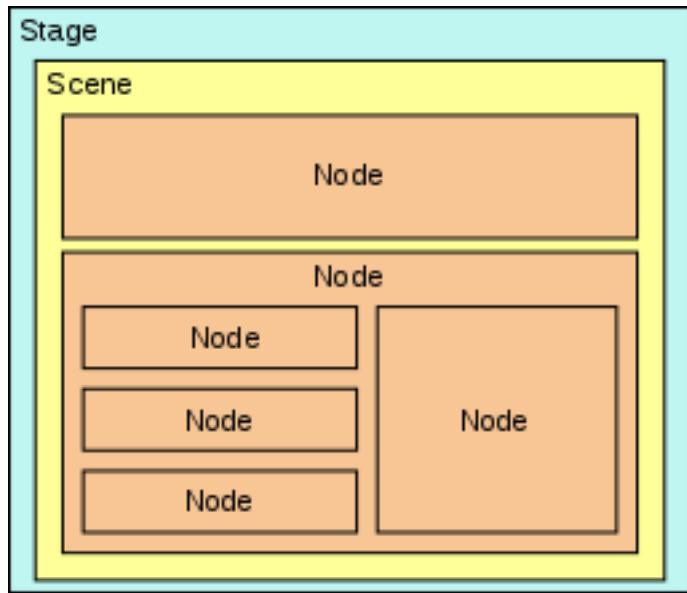


Figura 1.7: Struttura del layout di una applicazione JavaFX, by Stkl [CC BY-SA 4.0], via Wikimedia Commons

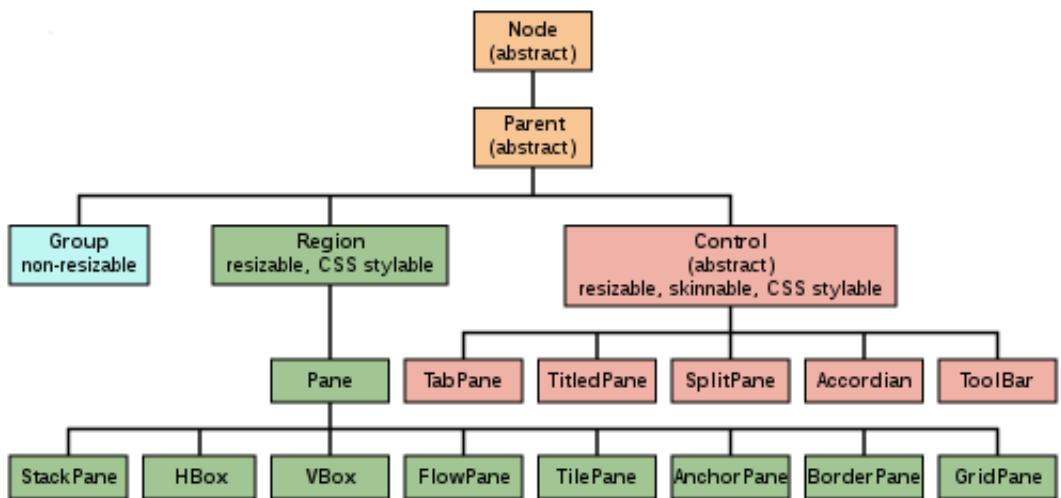


Figura 1.8: Struttura dei *nodi* di JavaFX, by Stkl [CC BY-SA 4.0], via Wikimedia Commons

1.2.4 Vantaggi di JavaFX su Swing

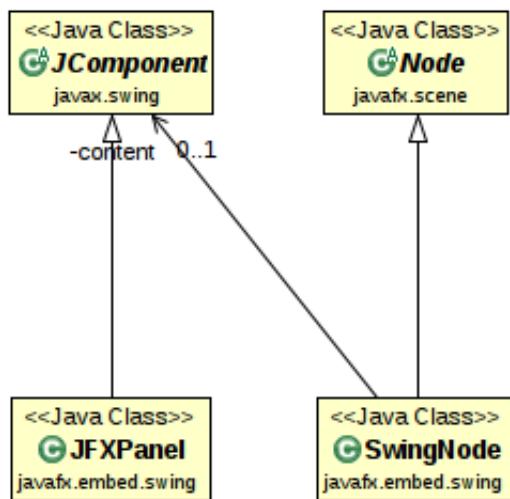


Figura 1.9: Diagramma delle classi *adapter* per *nodi JavaFX* e *componenti Swing*

Nonostante sia presente un livello di interoperabilità fornito dalle classi *adapter* `JFXPanel` e `SwingNode` del package `javafx.embed.swing` (Figura 1.9), come già sostenuto nella sottosezione 1.2.1 lo sviluppo del framework JavaFX è orientato prima ad affiancare poi a sostituire Swing e le *Java Foundation Classes* come ambiente grafico per applicazioni su *Java Virtual Machine*.

Escludendo il fatto di essere la libreria grafica non di terze parti più moderna per Java e il cosiddetto “eye candy factor”, JavaFX può vantare diversi vantaggi su Swing:

FXML L'utilizzo di un linguaggio di markup basato su XML per la definizione della componente di UI offre un approccio dichiarativo nel processo di costruzione che Swing non poteva offrire; inoltre, questo porta ad un incapsulamento della compionente *View* dei pattern di progettazione MV* sostanzialmente “automatico”.

Da non sottovalutare è anche la presenza di un software ufficiale per il design dell'interfaccia in FXML: *Scene Builder*. Esso offre un'interfaccia *drag'n'drop* per l'aggiunta dei componenti grafici, ed è perfettamente integrato in tutte le IDE più utilizzate (Eclipse, IntelliJ IDEA, Netbeans). La versione più recente del software creato da Oracle è distribuito da Gluon².

Skin e CSS Altro miglioramento in merito alla flessibilità riguarda il supporto ai fogli di stile CSS (*Cascading style sheet*): ampiamente utilizzati nello sviluppo web, essi

²<http://gluonhq.com/products/scene-builder/>

permettono di sviluppare temi per una applicazione JavaFX senza coinvolgere il codice Java, a differenza dei *look'n'feel* di Swing, né i file FXML, encapsulando completamente la componente di presentazione.

Supporto multimediale Grazie a JavaFX Media, la piattaforma supporta nativamente i più comuni formati audio (MP3, AIFF, WAV, AAC) e video (FLV con compressione VP6, MP4 con compressione H.264/AVC) senza richiedere l'utilizzo di librerie interne. Il supporto è incluso anche nella *WebView*.

Animazione Rispetto a Swing, il supporto alle animazioni (sia per quanto riguarda il movimento di componenti grafiche che gli effetti di transizione) è stato notevolmente alleggerito; è stato inoltre introdotto il supporto alle alterazioni regolate dal tempo, precedentemente molto complesse da implementare a causa del paradigma di rappresentazione tramite doppio buffer di default.

Supporto HTML Completamente assente in Swing era la possibilità di rendering di contenuti HTML; con JavaFX è stato inserito il supporto completo a Javascript, HTML5 e CSS tramite motore di rendering WebKit (utilizzato da browser come Safari, Google Chrome e Opera) e nodi *WebView* dedicati.

Scaling e Hi-DPI Recentemente si sta verificando un progressivo innalzarsi delle risoluzioni degli schermi: i sempre più comuni display 4K UltraHD o display retina dei dispositivi Apple recenti sono alcuni esempi. Eseguire applicazioni su questo tipo di schermi in moltissime occasioni causa problemi di visualizzazione: infatti l'utilizzo di valori assoluti per quanto riguarda la dimensione dei componenti grafici rende la GUI molto piccola e soprattutto quasi totalmente inutilizzabile dall'utente, a causa dell'elevato rapporto di pixel per unità di dimensione.

JavaFX 8 introduce il supporto a Hi-DPI, assente in Swing.

1.3 Interfaccia JavaFX per Alchemist: motivazioni

Come spiegato nella sottosezione 1.1.3 a pagina 5, l'esperienza utente dell'interfaccia utente classica di Alchemist è estremamente limitata. Miglioramenti recenti [6] sono stati apportati a parte dell'interfaccia, impiegando la libreria JavaFX per implementare un'esperienza d'uso più intuitiva, simile all'utilizzo di una IDE. Evidente era dunque la necessità di una nuova interfaccia grafica per l'ambiente di simulazione, che potesse avvicinare

utilizzatori meno esperti in ambito informatico all'utilizzo dei Alchemist per impieghi scientifici nei loro campi di competenza.

Si è scelto dunque di effettuare la reimplementazione in JavaFX anche per questa parte per diversi motivi:

- Poiché l'alternativa valutabile era Swing, la prima motivazione è composta da tutte le nuove funzionalità elencate nella sottosezione 1.2.4; in particolare, era importante il supporto ad uno scaling corretto dell'interfaccia su tutti gli schermi e una gestione delle animazioni più leggera che potesse impattare in modo inferiore le risorse della macchina e lasciare maggiore potenza computazionale al motore di simulazione.

Inoltre, è stato considerato interessante l'orientamento “intrinseco” che l'utilizzo di file FXML ha verso il pattern MVC, già utilizzato in Alchemist per il design della struttura dei moduli.

- Una seconda motivazione riguarda il supporto futuro: come illustrato precedentemente, JavaFX si presenta come il nuovo punto di riferimento per quanto riguarda l'implementazione di GUI per applicazioni JVM, dunque è stato considerato più conveniente abbandonare la soluzione *legacy* Swing per la più nuova alternativa.
- In ultimo, vi è una motivazione prettamente estetica: la maggiore flessibilità a livello di personalizzazione grafica che JavaFX è in grado di offrire ha permesso di adottare le direttive grafiche di design molto apprezzati come, in questo caso, il Material Design definito da Google.

Capitolo 2

Contributo

In questo capitolo verrà analizzato il contributo fornito al progetto, elencando i requisiti necessari e analizzando il processo di soddisfazione degli stessi.

L'obiettivo principale è quello di integrare una nuova interfaccia per la simulazione, al fine di semplificare l'adozione del simulatore da parte di utenti inesperti.

2.1 Analisi dei requisiti

Lo studio del lavoro illustrato in questa tesi ha inizio con l'analisi dei requisiti dell'interfaccia utente, ossia cosa l'applicazione deve mostrare a schermo.

Questa sezione si occuperà di enunciare i requisiti funzionali e non funzionali individuati.

2.1.1 Requisiti funzionali

I requisiti funzionali (Figura 2.1) descrivono il comportamento che il sistema deve avere: descrivono le funzionalità del sistema software, in termini di servizi che il sistema software deve fornire, di come il sistema software reagisce a specifici tipi di input e di come si comporta in situazioni particolari.

Rappresentazione dell'ambiente di simulazione

Essendo la componente grafica da reimplementare quella legata alla simulazione in esecuzione, requisito fondamentale è che la GUI possa rappresentare l'ambiente con le maggiori possibilità di dettaglio possibile.

Di conseguenza, deve essere presente uno spazio disegnabile in cui si possa avere una rappresentazione grafica di quanto accade, ma anche contatori che mostrino l'avanzamento della simulazione in termini di tempo (secondi) trascorso e passaggi (*step*) effettuati.

Gestione degli effetti

La nuova interfaccia deve rendere possibile all'utente di poter aggiungere nuovi effetti allo *stack* di rappresentazione e modificarne le proprietà a tempo di esecuzione.

Inoltre attraverso la GUI l'utente deve poter salvare lo stack di effetti presente in quel momento e caricarlo in un secondo momento, mantenendo tutte le proprietà definite manualmente.

Infine, deve essere possibile nascondere singoli effetti o gruppi di essi senza rimuoverli dallo *stack*.

Effetti standard per nodi e collegamenti

Devono essere implementati effetti adibiti alla rappresentazione dei singoli nodi come punti e dei collegamenti tra i nodi di un vicinato.

Questi effetti dovranno essere caricati automaticamente al lancio dell'applicazione, salvo diversamente specificato.

Interazione con simulazione e ambiente rappresentato

L'interfaccia deve mettere a disposizione dell'utente la capacità di interagire con la simulazione, potendo fermarla e riavviarla, interagire con i nodi e spostarsi tra essi. Deve essere possibile effettuare pan e zoom sull'ambiente rappresentato.

Le possibilità di interazione non devono essere vincolate al puntatore del mouse, ma deve supportare anche le scorciatoie da tastiera.

Rappresentazione di ambienti con mappa

Deve essere fornito il supporto alle mappe come sfondo degli ambienti nella rappresentazione di simulazioni che coinvolgano questo aspetto.

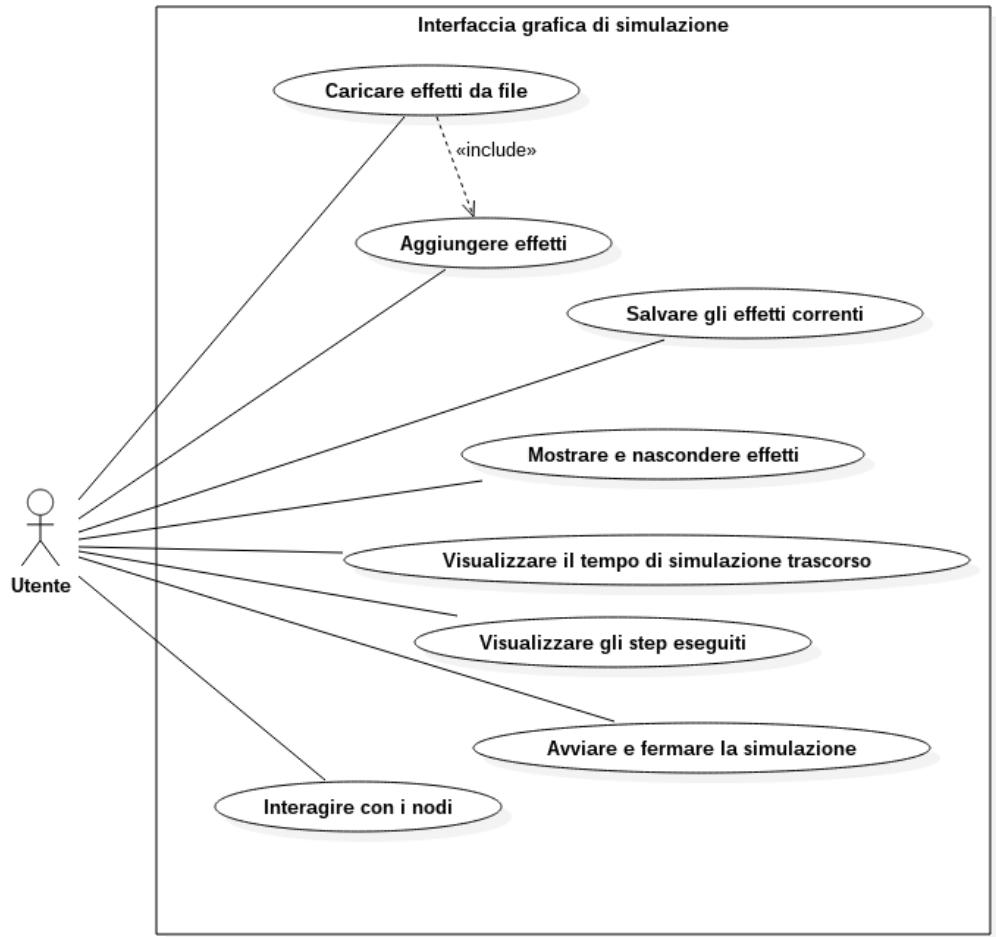


Figura 2.1: Requisiti funzionali principali

2.1.2 Requisiti non funzionali

I requisiti non funzionali descrivono le proprietà non comportamentali che il sistema deve possedere, come efficienza, affidabilità, sicurezza, performance, ma anche caratteristiche del processo di sviluppo e caratteristiche esterne.

JavaFX

Come specificato nella sezione 1.3 a pagina 14, il processo di sviluppo deve coinvolgere la libreria JavaFX come framework per la costruzione dell'interfaccia.

Performance

L'interfaccia grafica deve quanto più possibile non gravare sulle prestazioni del motore di simulazione; in particolare, poiché JavaFX non è nativamente *thread-safe*, è necessario gestire la concorrenza in modo oculato.

Supporto Hi-DPI

L'interfaccia non deve perdere di usabilità e qualità di rappresentazione su alcun tipo di schermo, indipendentemente dalla risoluzione e dalla densità di pixel. Per fare questo si devono quindi utilizzare quanto più possibile grandezze relative e sfruttare al meglio in tal senso le funzionalità offerte da JavaFX.

Serializzazione *Human-readable*

Deve essere possibile serializzare gli effetti in un formato testo, in modo tale che possa essere facilmente creato e/o modificato manualmente in modo semplice, senza coinvolgere necessariamente l'interfaccia di Alchemist.

2.2 Fonti d'ispirazione

Una volta chiariti i requisiti dell'interfaccia, il passo successivo riguarda la progettazione dell'interfaccia.

Per poter disegnare dei mockup da utilizzare come bozzetti, sono state fatte ricerche in merito alle interfacce grafiche utilizzate da altri simulatori, anche a scopo non strettamente scientifico.

Infine, si è scelto tra i design moderni più comuni e apprezzati uno da adottare per fornire un aspetto grafico a cui l'utente medio fosse già abituato e che potesse fornire un'esperienza di utilizzo più gradevole.

2.2.1 Simulatori a scopo videoludico

Come già segnalato nelle sezioni precedenti, è importante che l'interfaccia grafica si presenti semplice e immediata anche per l'utente non avanzato. Di conseguenza, si è scelto di analizzare con più attenzione le GUI di simulatori sviluppati a scopo prettamente videoludico, in quanto più orientati all'immediatezza d'uso rispetto ai simulatori di concezione scientifica.

Tra i videogiochi di simulazione più famosi, è stato interessante analizzare SimCity, il quale all'epoca del lancio fu molto apprezzato [15] appunto per il gameplay e l'interfaccia piuttosto innovativi, e i giochi della serie Universe Sandbox del team Giant Army.

SimCity

La celebre serie di videogiochi di simulazione SimCity [34], ideata da Will Wright tra gli anni '80 e gli anni '90 ispirandosi alla ricerca contenuta nel saggio di architettura *A pattern language* [22, 2], sviluppata da Maxis e distribuita da Electronics Arts, è tutt'ora considerata una delle più innovative per quanto riguarda la storia dei videogiochi di simulazione.



Figura 2.2: SimCity (2013), sviluppato da Maxis e distribuito da EA, tutti i diritti riservati ai ripetitivi proprietari

Universe Sandbox

Altra serie di videogiochi simulativi analizzata è Universe Sandbox. Dopo oltre 15 anni di sviluppo [8], il primo capitolo [11] è stato rilasciato dallo sviluppatore e artista Dan Dixon nel 2008. Il responso positivo lo ha portato a continurare lo sviluppo, tanto da fondare la compagnia Giant Army [12], che ha rilasciato nel 2015 la seconda iterazione della saga, Universe Sandbox² [16].

Per il design dell’interfaccia, è stato proprio il secondo capitolo a fungere da maggior fonte d’ispirazione. Essa va a riprendere la classica interfaccia utilizzata da videogiochi simulativi come il già citato SimCity (Figura 2.2), ma andando a rimuovere buona parte degli ornamenti grafici tipici delle GUI a scopo videoludico, andando a preferire uno stile molto più pulito e semplificato; il sistema di interazione a sviluppo verticale e a popup viene sostituito con uno sviluppo orizzontale di pannelli (definito *Modello multi-paned* [7]) che vanno a raccogliere tutte le impostazioni e i parametri.

La simulazione viene rappresentata sullo sfondo, come in SimCity, ma con effetti di trasparenza non presenti nel gioco di EA.

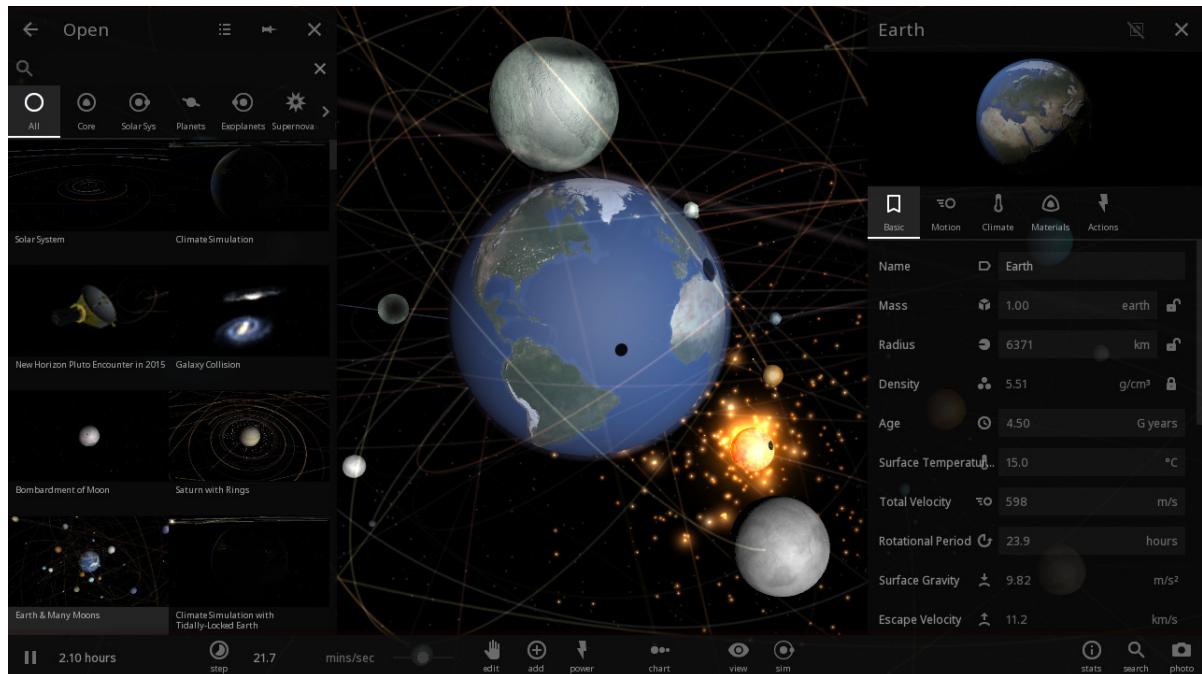


Figura 2.3: Universe Sandbox² (2015), sviluppato e distribuito da Giant Army, tutti i diritti riservati ai ripetuti proprietari

2.2.2 Material Design

Uno dei maggiori motivi che hanno portato l’interfaccia grafica di Alchemist a necessitare di un rinnovamento è stata l’intenzione di semplificarla all’occhio dell’utente non esperto, fornendo un’esperienza completa e gradevole. Era dunque necessario scegliere uno stile grafico familiare, moderno e facilmente adattabile a quella che sarebbe essere la nuova interfaccia che si stava progettando.

Prendendo come base l’interfaccia di Universe Sandbox illustrata nella sezione 2.2.1, è possibile notare che il design di base sia estremamente “flat”; si è deciso di valutare i possibili design a cui adeguare la UX che si aveva intenzione di progettare.

La scelta è infine ricaduta sul Material Design [24] sviluppato da Google: dal suo annuncio nel giugno del 2014 alla conferenza del Google I/O [32] esso è stato almeno parzialmente adottato in molte applicazioni web, mobile e desktop, e ben si si presta all’implementazione di un’interfaccia semplice e minimale.

Si è deciso di utilizzare le icone¹ e le direttive in merito a dimensioni e palette di colore² fornite da Google.

2.3 Design dell’interfaccia

Una volta chiariti i requisiti e le possibili fonti di ispirazione per la struttura della GUI da realizzare, sono stati disegnati dei mockup che potessero rappresentare una linea guida per l’implementazione concreta dell’interfaccia.

Come è possibile vedere dalla Figura 2.4 nella pagina successiva, si è scelto di adottare un’interfaccia composta da uno spazio disegnabile centrale, al quale viene sovrapposta nella parte inferiore una barra contenente dei controlli che permettono un’interazione semplice e diretta con le funzionalità di base:

Play/Pausa Partendo da destra, è presente un bottone che permette di avviare e mettere in pausa la simulazione.

Esso funge anche da indicatore per lo stato attuale della simulazione: qualora essa venga avviata o fermata da terminale o tramite una scorciatoia da tastiera, l’icona rappresentata sul bottone viene aggiornata per adeguarsi al nuovo stato.

¹<https://material.io/icons/>

²<https://material.io/color/>

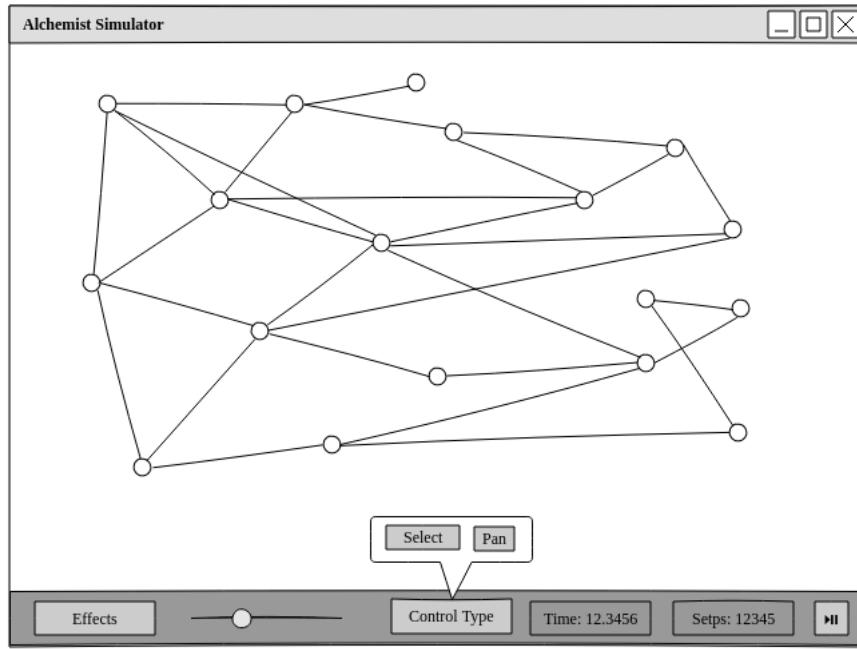


Figura 2.4: Mockup dell'interfaccia principale

Avanzamento in termini di tempo e step Continuando verso sinistra, si trovano spazi dedicati al numero di secondi di simulazione rappresentati e di step effettuati; essi vengono aggiornati durante tutto l'avanzamento del motore di simulazione.

Gestione del sistema di controllo Poiché l'interazione tramite mouse deve permettere sia di spostarsi nell'ambiente che selezionare i nodi e interagirvi, è presente un bottone che apre un pannello che permette di scegliere tra spostamento (*pan*) e selezione.

Gestione della velocità Una barra a scorrimento permette di regolare la velocità di rappresentazione della simulazione.

Gesione degli effetti Un bottone sul lato sinistro della barra permette di aprire un pannello sul medesimo lato della finestra per poter controllare gli effetti con i quali rappresentare cosa sta avvenendo nella simulazione.

Nelle Figure da 2.5(a) a 2.5(c) nella pagina seguente è possibile osservare i diversi livelli del drawer laterale degli effetti.

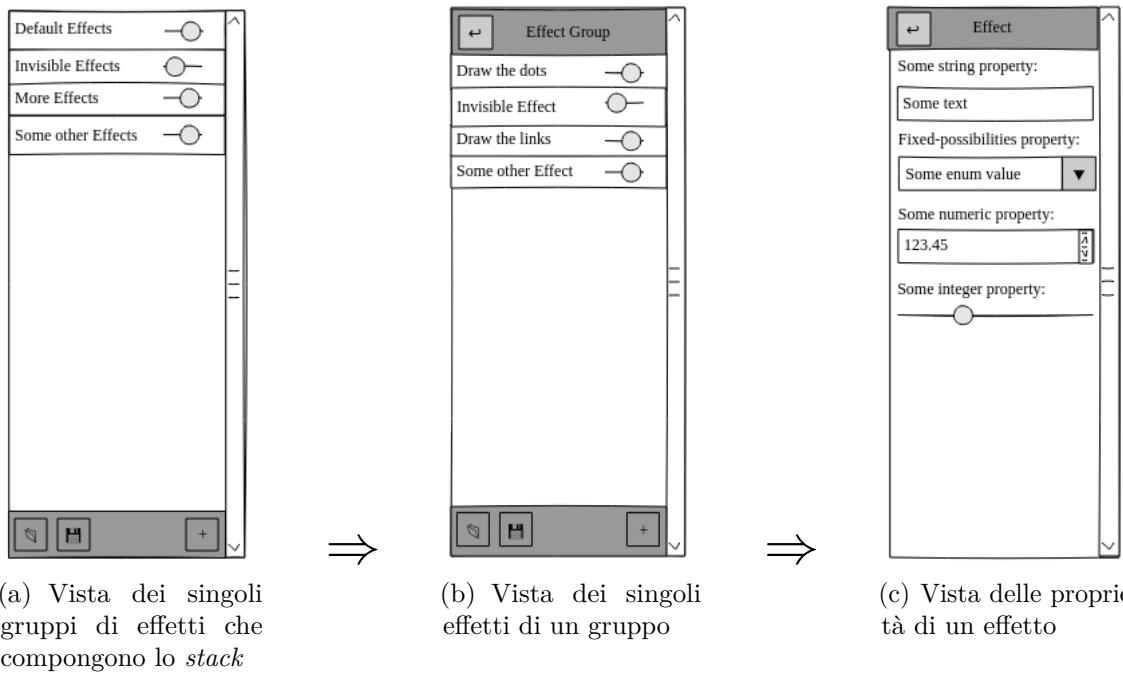


Figura 2.5: Vista del pannello laterale degli effetti nei diversi livelli

2.4 Progettazione

Terminata la realizzazione dei mockup, il passo successivo riguardava la progettazione della struttura del sistema software. Durante la progettazione, in molti casi ci si è serviti di pattern di progettazione specifici, secondo quanto definito dalla cosiddetta *Gang of Four*³ nel loro celebre saggio [33] del 1995.

2.4.1 L'architettura degli effetti

I singoli effetti e l'interfaccia `EffectFX`

La componente architettonica più complessa da progettare probabilmente è costituita dagli effetti. Infatti, concettualmente il nuovo archetipo di effetti (della cui interfaccia è possibile vedere il codice in Appendice B.1) rappresenta un oggetto completamente diverso:

- esso non si relaziona più con il singolo nodo, del quale può considerare le proprietà, bensì con l'intero ambiente; in questo modo, l'effetto agisce in blocco su un determinato tipo di entità allo stesso modo, garantendo una migliore uniformità di applicazione.

³John Vlissides, Richard Helm, Ralph Johnson, Erich Gamma

- esso ha un *nome* che lo identifica dalle altre istanze della medesima classe; questo permette all’utente di identificarlo con più semplicità e gestirlo in modo più naturale, soprattutto nel caso si trovi a gestire, attraverso l’interfaccia grafica, una moltitudine di effetti.
- esso possiede un campo di *visibilità* individuale, che permette di nasconderlo temporaneamente, aumentando le possibilità di rappresentazione anche per blocchi di effetti predefiniti.
- le proprietà peculiari di ciascun effetto sono pensate per implementare il pattern *observer* [26], permettendo di effettuare collegamenti con l’interfaccia in modo trasparente ed ottimizzato, in quanto gestito completamente dal framework di JavaFX.
- infine, l’effetto viene serializzato in formato *human-readable*, il quale facilita la creazione e la modifica anche al di fuori dell’interfaccia di Alchemist.

Si è scelto di utilizzare il formato JSON (JavaScript Object Notation [9])⁴: esso è un formato di testo per la serializzazione dei dati strutturati, basato sugli oggetti JavaScript, che 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 programmatore di linguaggi.

Come rappresentato graficamente nel diagramma UML in Figura 2.6 nella pagina successiva, un effetto viene concretizzato secondo il pattern *template method* [27]: la struttura di funzionamento di base viene parzialmente definita da una classe astratta che implementa il metodo principale dell’interfaccia effetto, `computeDrawCommands()`, come metodo template, il quale chiama i due metodi astratti `getData()` e `consumeData()` per adempiere al proprio compito. La suddivisione permette a ciascun effetto concreto di separare le procedure che coinvolgono l’interrogazione del modello da quelle che portano alla costruzione della coda di comandi per effettuare la rappresentazione grafica.

I gruppi di effetti e l’interfaccia `EffectGroup`

L’esigenza di permettere all’utente di poter realizzare rappresentazioni complesse con numerosi effetti ha portato alla definizione di una classe collezione specifica:

⁴<https://www.json.org/index.html>

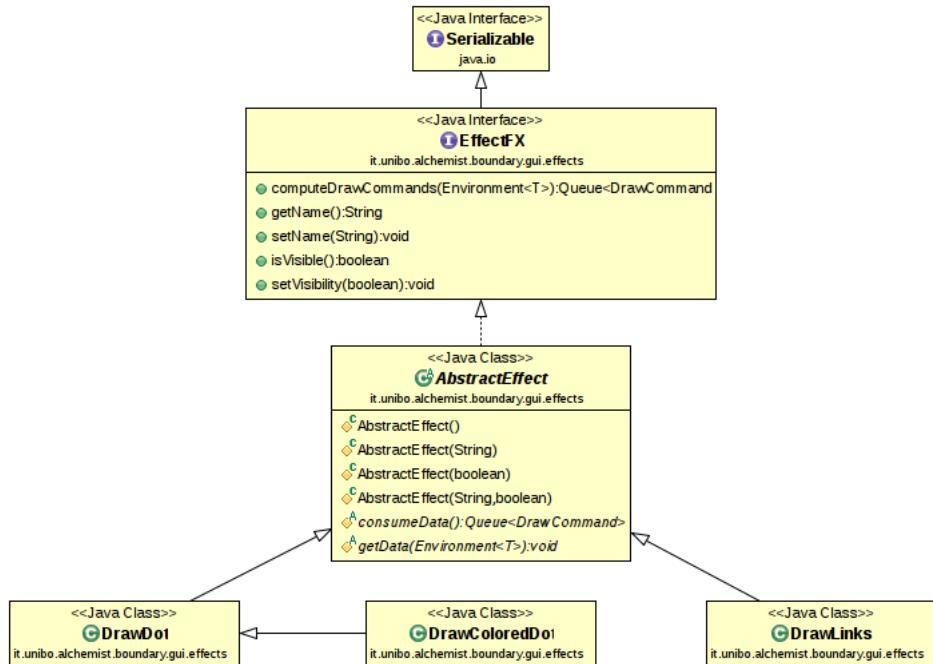


Figura 2.6: Diagramma UML delle classi che modellano la nuova struttura di effetti; maggiori dettagli in Appendice B.1

- la classe, secondo il contratto classico definito dall’interfaccia `Collection` di Java, è pensata per essere iterabile [25] e per permettere l’applicazione in blocco degli effetti che la compongono;
- anche il gruppo di effetti è definito da un nome che lo distingue dagli altri, permettendo all’utente di distinguere in modo più immediato;
- altra proprietà che accomuna la collezione con gli oggetti che è stata pensata per contenere è la visibilità: essa permette di mostrare e nascondere in blocco tutti i suoi effetti senza andare a modificare la visibilità di ciascuno degli effetti; l’interfaccia consente comunque di agire anche singolarmente sulla visibilità dei singoli nodi.

Caricamento, salvataggio e modifica di gruppi di effetti

Come detto nella pagina precedente, il salvataggio e il caricamento degli effetti avvengono tramite file JSON. Si è deciso di modellare una *stateless utility class* che si comportasse da intermediario con la libreria utilizzata per la serializzazione, comportandosi secondo il pattern strutturale di tipo *Façade* [33]. È possibile vedere la rappresentazione UML in Figura 2.7 nella pagina successiva.

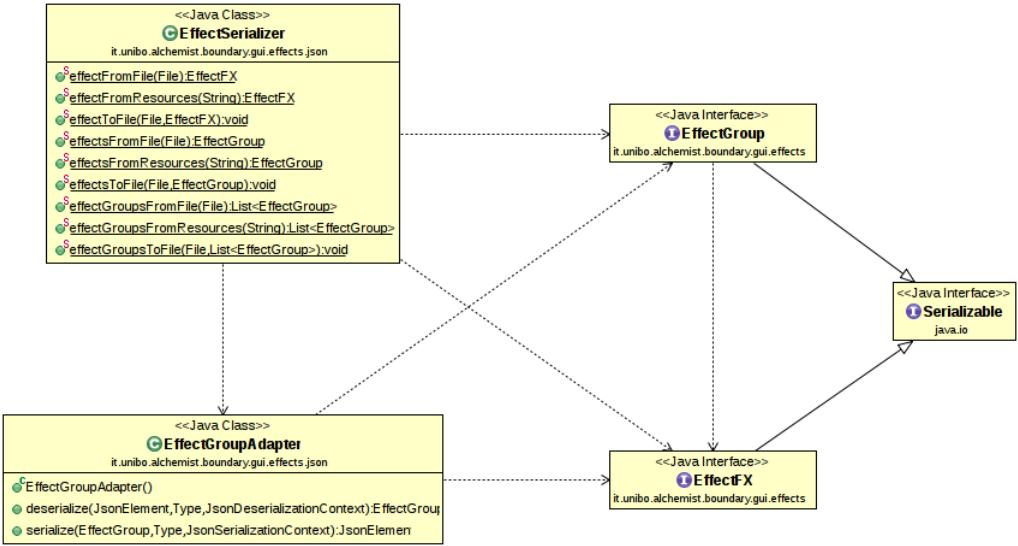


Figura 2.7: Diagramma UML delle classi che modellano la logica di serializzazione degli effetti tramite *Façade stateless*

2.4.2 La struttura dei drawer e le proprietà osservabili

Una volta realizzato il mockup illustrato nella Sezione 2.3, l'attenzione si è spostata su progettare la struttura a livello software. Trascurando i dettagli implementativi prettamente legati all'implementazione grafica, di cui si parla nella Sezione 2.5 nella pagina seguente, è stato importante progettare la gestione degli eventi di modifica di proprietà di effetti e gruppi. Come detto nella Sottosezione 2.4.1 a pagina 25, si è deciso di fare ampio uso del pattern *Observer*, ciascun effetto è progettato per implementare proprietà osservabili (UML in Figura 2.8 nella pagina seguente), che possono implementare ascoltatori di eventi dedicati in modo semplice, potendosi avvalere delle API messe a disposizione dal framework JavaFX.

2.4.3 Nodi grafici come monitor per la simulazione

Il canvas in cui vengono rappresentati gli effetti, il bottone che controlla l'avvio della simulazione e le etichette che mostrano il progresso della simulazione in termini di tempo e step sono indubbiamente nodi grafici; in fase di progettazione, si è scelto però di modellarli anche come monitor per il motore della simulazione, sfruttando ancora una volta il pattern *observer* per mettere in ascolto gli elementi della GUI legati alla simulazione stessa al fine di ricevere comunicazione di doversi aggiornare ad ogni step eseguito che li riguardi.

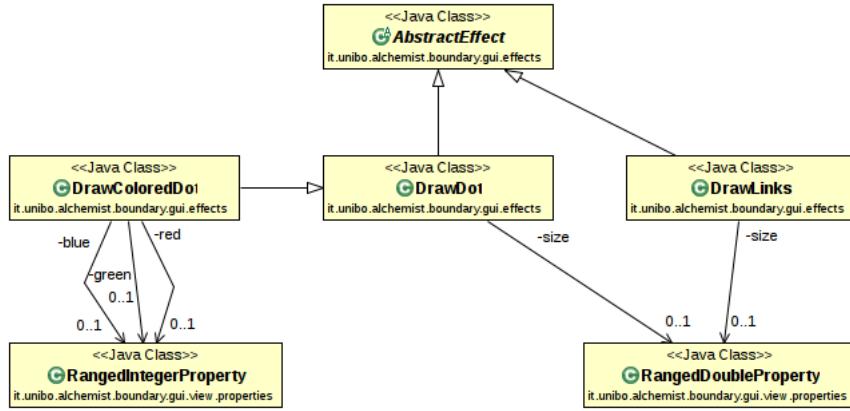


Figura 2.8: Diagramma UML delle classi che modellano la logica delle proprietà serializzabili e osservabili implementate

2.4.4 Costruzione e avvio dell’interfaccia

L’interfaccia classica gestiva l’avvio della GUI per la simulazione in tempo reale, costruendo l’interfaccia con componenti Swing specifici assemblati in base ai parametri richiesti.

La nuova interfaccia grafica è progettata conservando l’impiego di pattern *builder*, presentando un oggetto adibito all’interpretazione dei parametri e alla semplificazione del processo di costruzione. Si differenzia però per l’assenza di frammentazione della struttura: essa è modellata come un’unica applicazione JavaFX, con un layout predefinito, e il builder si occupa di modificare il comportamento e il tipo di canvas per adattarlo alla simulazione, oltre a caricare gli effetti e la simulazione stessa, ma non influenza l’aspetto della UI.

2.5 Dettagli implementativi

2.5.1 La barra inferiore

La barra inferiore (Figura 2.9 nella pagina successiva) è stata trasposta quasi perfettamente dal mockup al codice: si è utilizzato la classe `javafx.scene.control.ButtonBar` per modellare il layout della barra, mentre sono state impiegate le classi `JFXButton` e `JFXSlider` fornite nel package `com.jfoenix.controls` della libreria JFoenix per modellare i controlli mantenendoli sul prescelto stile del Material Design di Google. Il popup per la modifica del sistema di controllo è stato realizzato tramite la classe `org.controlsfx.control.PopOver` presente nella libreria ControlsFX.



Figura 2.9: La barra inferiore concretamente realizzata

Gli effetti di colore e trasparenza sono stati implementati con diverse specifiche inserite nel foglio di stile CSS, inline nel documento FXML e attraverso i suddetti componenti.

2.5.2 Drawer, liste e celle

L'impiego dei drawer, elemento tipico del Material Design, è stato possibile, ancora una volta, grazie alle classi `JFXDrawer` e `JFXDrawersStack` forniti da JFoenix.

La struttura di un drawer è costituita dagli elementi seguenti:

- una opzionale barra superiore, che va a rappresentare il nome dell'elemento di cui si va a modificare il contenuto (ed è dunque assente nella vista di tutti i gruppi di effetti caricati);
- una barra inferiore, che fornisce possibili interazioni:
 - tornare al drawer precedente nello stack; questa funzionalità è presente su ogni livello tranne il primo.
 - aggiungere nuovi effetti o gruppi di effetti, andando a cercare tramite *reflection* quelli presenti all'interno del classpath; questa funzionalità è assente nel drawer che mostra le proprietà di uno specifico effetto.
 - salvare e caricare gruppi di effetti da file; questa funzionalità è disponibile solo nel primo drawer.
- la parte centrale del drawer, che può essere costituita da una `ListView` di `EffectFX` o di `EffectGroup`, o da una rappresentazione delle proprietà dell'effetto costruita a tempo di esecuzione sfruttando la *reflection* per identificare quale nodo di controllo utilizzare.

Nel caso della lista, le celle sono state personalizzate per poter essere riordinate tramite *drag'n'drop* e per permettere la modifica della visibilità tramite un `JFXToggleButton`.

2.5.3 Librerie esterne utilizzate

Per realizzare questa interfaccia grafica è stato necessario utilizzare delle librerie esterne che forniscono ulteriori funzionalità. Le librerie esterne utilizzate sono le seguenti.

ControlsFX

GSON

JFoenix

jIconFont

2.5.4 Strumenti utilizzati

Qualità del codice e controllo del software

Date le dimensioni di Alchemist, è necessario fare uso di strumenti che controllino la qualità del codice e diano la possibilità di testarlo in modo immediato.

Gli strumenti di qualità del codice permettono di revisionare il codice in modo sistematico, così da evitare errori che a volte possono verificarsi, senza bisogno che il programma venga realmente eseguito: essi analizzano il codice sorgente per individuare potenziali bug o codice duplicato e per indicare i possibili miglioramenti e ottimizzazioni.

Il progetto Alchemist utilizza i seguenti:

FindBugs

Checkstyle

PMD

Controllo di versione

Automazione dello sviluppo e integrazione continua

Ambiente di sviluppo integrato

Capitolo 3

Conclusioni

3.1 Risultati

3.2 Lavori futuri

Appendice A

Codice relativo all'interfaccia classica

A.1 L'interfaccia Effect

```
public interface Effect extends Serializable {  
    /**  
     * Applies the effect.  
     *  
     * @param graphic  
     *          Graphics2D to use  
     * @param node  
     *          the node to draw  
     * @param x  
     *          x screen position  
     * @param y  
     *          y screen position  
     */  
    void apply(Graphics2D graphic, Node<?> node, int x, int y);  
  
    /**  
     * @return a color which resembles the color of this effect  
     */  
    Color getColorSummary();  
  
    @Override // Should override hashCode() method  
    int hashCode();  
  
    @Override // Should override equals() method  
    boolean equals(Object obj);
```


Appendice B

Codice relativo al contributo

B.1 L'interfaccia EffectFX

```
/**  
 * Graphical visualization of something happening in the  
 environment.  
 */  
public interface EffectFX extends Serializable {  
  
    /**  
     * Computes a queue of commands to Draw something.  
     *  
     * @param environment the environment to gather data from  
     * @param <T>           the {@link Concentration} type  
     * @return the queue of commands that should be run to draw  
 the effect  
     */  
    <T> Queue<DrawCommand> computeDrawCommands(Environment<T>  
environment);  
  
    /**  
     * Gets the name of the effect.  
     *  
     * @return the name of the effect  
     */  
    String getName();  
  
    /**
```

```

        * Sets the name of the effect.
        *
        * @param name the name of the effect to set
        */
void setName( String name);

/**
 * Gets the visibility of the effect.
 *
 * @return the visibility of the effect
 */
boolean isVisible();

/**
 * Sets the visibility of the effect.
 *
 * @param visibility the visibility of the effect to set
 */
void setVisibility( boolean visibility);
}

```

B.2 L'interfaccia EffectGroup

```

/**
 * Models a group of effects. Each effect has a different
priority of
 * visualization.
 */
public interface EffectGroup extends Serializable , Queue<
EffectFX> {

/**
 * Computes all the commands for all the visible effects in
this group.
*
 * @param environment the environment to gather data from
 * @param <T>          the {@link Concentration} type
 * @return the queue of commands that should be run to draw
the effects of the group

```

```

        * @see EffectFX#computeDrawCommands( Environment )
        */
    <T> Queue<DrawCommand> computeDrawCommands( Environment<T>
environment);

    /**
     * Gets the name of the group.
     *
     * @return the name of the group
     */
    String getName();

    /**
     * Sets the name of the group.
     *
     * @param name the name of the group
     */
    void setName( String name);

    /**
     * Checks if an effect is present in the group.
     *
     * @param effect the effect to search
     * @return the position , or -1 if not present
     */
    int search(EffectFX effect);

    /**
     * Returns the visibility of the group.
     *
     * @return the visibility
     */
    boolean isVisible();

    /**
     * Sets the visibility of the group.
     *
     * @param visibility the visibility
     */
    void setVisibility( boolean visibility);

```

```

    /**
     * Returns the visibility of the specified effect.
     *
     * @param effect the effect
     * @return the visibility
     * @throws IllegalArgumentException if can't find the effect
     * @see EffectFX#isVisible()
     */
    boolean getVisibilityOf(EffectFX effect);

    /**
     * Sets the visibility of the specified effect.
     *
     * @param effect the effect
     * @param visibility the visibility to set
     * @throws IllegalArgumentException if can't find the effect
     * @see EffectFX#setVisibility(boolean)
     */
    void setVisibilityOf(EffectFX effect, boolean visibility);

    /**
     * Changes the specified offset priority of the specified
     * offset.
     *
     * @param effect the effect
     * @param offset the offset; it can be positive or negative
     * @throws IllegalArgumentException if can't find the effect
     */
    void changePriority(EffectFX effect, int offset);

    /* Is suggested to override Object default equals method. */
    @Override
    int hashCode();

    /**
     * Compares the {@link EffectGroup} {@link EffectGroup}s. The result
     * is true if and
     *   * only if the argument is not {@code null} and every {@link
     * EffectFX}
     *   * contained is not {@code null} and {@link EffectFX#equals(
     * Object)} equal to

```

```

        * the corresponding in the comparing {@code EffectGroup} (order is
        * important!) and the group has the same name, visibility
and transparency.

        *
        * @see Object#equals( Object )
        */
/* Is suggested to override Object default equals method. */
@Override
boolean equals( Object obj );
}

```

B.3 Implementazioni di EffectFX e Proprietà serializzabili

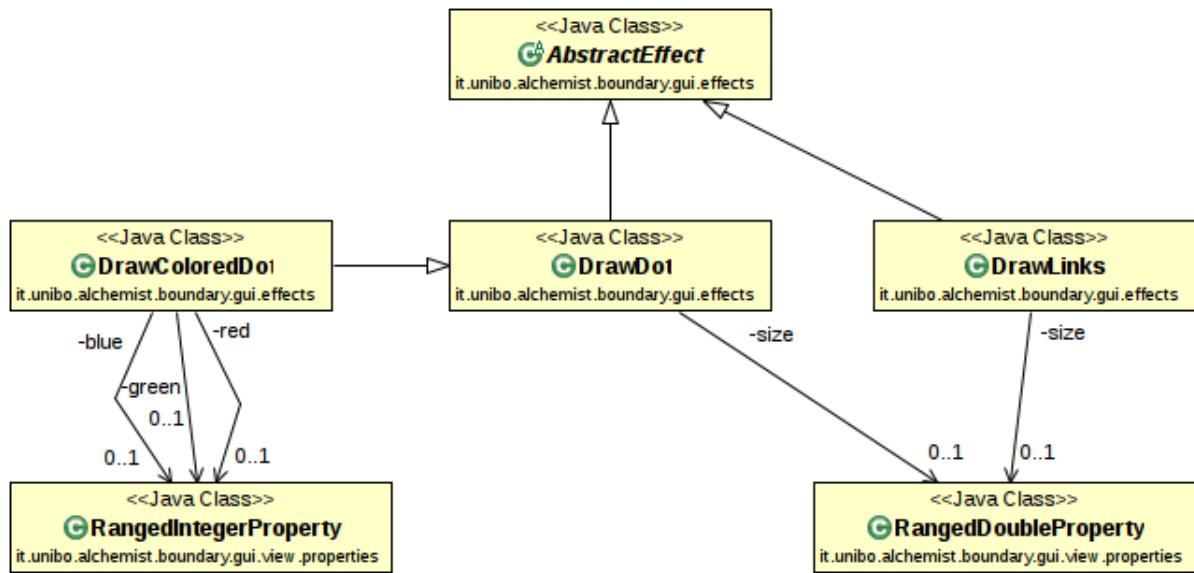


Figura B.1: Il diagramma UML delle classi mostra le relazioni tra gli effetti implementati e le proprietà custom utilizzate

Bibliografia

- [1] *Alchemist*. URL: <http://alchemistsimulator.github.io/>.
- [2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King e S. Angel. «A Pattern Language: Towns, Buildings, Construction(Center for Environmental Structure)». In: (1977).
- [3] E. Babulak e M. Wang. «Discrete event simulation: State of the art». In: *International Journal of Online Engineering (iJOE)* 4.2 (2007), pp. 60–63.
- [4] J. Banks, J. S. Carson, B. L. Nelson e D. M. Nicol. *Discrete-Event System Simulation: Pearson New International Edition*. Pearson Higher Ed, 2013.
- [5] J. Bloch. *Effective Java (2Nd Edition) (The Java Series)*. 2^a ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 0321356683, 9780321356680.
- [6] E. Casadio. «Revisione e refactoring dell’interfaccia utente del simulatore Alchemist». Tesi di dott. URL: <http://amslaurea.unibo.it/12310/>.
- [7] B. Clark e V. Feliberti. *Multi-pane navigation model for graphical user interfaces*. US Patent App. 11/333,164. Mag. 2006. URL: <https://www.google.com/patents/US20060101353>.
- [8] A. Cox. «How one man created his own universe - How Dan Dixon fashioned a whole universe out of mere bytes». In: (2008). URL: <http://www.techradar.com/news/software/computing/how-one-man-created-his-own-universe-470870>.
- [9] D. Crockford. «The application/json media type for javascript object notation (json)». In: (2006).
- [10] I. DIS. «9241-210: 2010. Ergonomics of human system interaction-Part 210: Human-centred design for interactive systems». In: *International Standardization Organization (ISO). Switzerland* (2009).
- [11] D. Dixon. *Universe Sandbox*. 2008.

- [12] D. Dixon, C. Herold, G. Steinrhder, T. Grnnelv e E. Hilton. *Universe Sandbox*. 2011.
- [13] G. S. Fishman. «Principles of discrete event simulation.[book review]». In: (1978).
- [14] *Fluent Design System by Microsoft*. URL: <https://fluent.microsoft.com/>.
- [15] T. Friedman. «Making sense of software: Computer games and interactive textuality». In: *Cybersociety; Computer-Mediated Communication and Community*. Thousand Oaks, Calif.: Sage Publications (1995).
- [16] Giant Army. *Universe Sandbox 2*. 2014.
- [17] M. A. Gibson e J. Bruck. «Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels». In: *The Journal of Physical Chemistry A* 104.9 (2000), pp. 1876–1889. DOI: 10.1021/jp993732q. eprint: <http://dx.doi.org/10.1021/jp993732q>. URL: <http://dx.doi.org/10.1021/jp993732q>.
- [18] D. T. Gillespie. «A general method for numerically simulating the stochastic time evolution of coupled chemical reactions». In: *Journal of computational physics* 22.4 (1976), pp. 403–434.
- [19] D. T. Gillespie. «Exact stochastic simulation of coupled chemical reactions». In: *The journal of physical chemistry* 81.25 (1977), pp. 2340–2361.
- [20] M. Hassenzahl. «User experience (UX): towards an experiential perspective on product quality». In: *Proceedings of the 20th Conference on l'Interaction Homme-Machine*. ACM. 2008, pp. 11–15.
- [21] *Java Platform, Standard Edition (Java SE) 8*. URL: <https://docs.oracle.com/javase/8/javase-clienttechnologies.htm>.
- [22] K. Kelly. *Will Wright: The Mayor of SimCity*. 1994.
- [23] G. E. Krasner, S. T. Pope et al. «A description of the model-view-controller user interface paradigm in the smalltalk-80 system». In: *Journal of object oriented programming* 1.3 (1988), pp. 26–49.
- [24] *Material Design by Google*. URL: <https://material.io/>.
- [25] J. E. McDonough. «Iterator Design Pattern». In: *Object-Oriented Design with ABAP: A Practical Approach*. Berkeley, CA: Apress, 2017, pp. 239–246. ISBN: 978-1-4842-2838-8. DOI: 10.1007/978-1-4842-2838-8_18. URL: https://doi.org/10.1007/978-1-4842-2838-8_18.

- [26] J. E. McDonough. «Observer Design Pattern». In: *Object-Oriented Design with ABAP: A Practical Approach*. Berkeley, CA: Apress, 2017, pp. 155–171. ISBN: 978-1-4842-2838-8. DOI: 10.1007/978-1-4842-2838-8_13. URL: https://doi.org/10.1007/978-1-4842-2838-8_13.
- [27] J. E. McDonough. «Template Method Design Pattern». In: *Object-Oriented Design with ABAP: A Practical Approach*. Berkeley, CA: Apress, 2017, pp. 247–254. ISBN: 978-1-4842-2838-8. DOI: 10.1007/978-1-4842-2838-8_19. URL: https://doi.org/10.1007/978-1-4842-2838-8_19.
- [28] F. Moritz. «Rich Internet Applications (RIA): A Convergence of User Interface Paradigms of Web and Desktop-Exemplified by JavaFX». In: *University of Applied Science Kaiserslautern, Deutschland* (2008).
- [29] D. A. Norman. *La caffettiera del masochista (The psychology of everyday things)*. Inglese. 1988.
- [30] T. C. O'rourke, B. T. O'neill, R. C. Cook, K. O. Taner, S. P. Synder, A. R. Joyner et al. *Graphical user interface*. US Patent 5,349,658. Set. 1994.
- [31] D. Pianini, S. Montagna e M. Viroli. «Chemical-oriented simulation of computational systems with ALCHEMIST». In: *Journal of Simulation* 7.3 (ago. 2013), pp. 202–215. ISSN: 1747-7786. DOI: 10.1057/jos.2012.27. URL: <https://doi.org/10.1057/jos.2012.27>.
- [32] S. Pichai. *Google I/O 2014-Keynote*. 2014.
- [33] J. Vlissides, R. Helm, R. Johnson e E. Gamma. «Design patterns: Elements of reusable object-oriented software». In: *Reading: Addison-Wesley* 49.120 (1995), p. 11.
- [34] W. Wright. *SimCity*. Erbe, 1996.

Ringraziamenti