Alma Mater Studiorum - Università di Bologna
Campus di Cesena

SCUOLA DI SCIENZE
Corso di Laurea in Ingegneria e Scienze Informatiche

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

II sessione di laurea
Anno Accademico 2016 - 2017

Parole chiave

Alchemist

Simulatore

JavaFX

Interfaccia grafica

Effetto

Sommario

Lo scopo di questa tesi verte intorno allo studio del simulatore Alchemist e 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 seguente trattazione è strutturata su tre capitoli: nel capitolo 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 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 analizza i risultati ottenuti, interpretandoli anche in ottica di miglioramenti futuri.

Indice

Capitolo 1

Introduzione

1.1 Alchemist

Alchemist [1, 18] è un meta-simulatore estendibile completamente *open-source* che esegue su *Java Virtual Machine* (JVM), nato all'interno del'Università di Bologna e distribuito su licenza GNU GPLv3+ con *linking exception*; il codice è reperibile su GitHub[1], 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* [3] 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) [2, 6]: 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 [9]).

Per perseguire questo obiettivo, la progettazione dell'algoritmo è partita dallo studio del lavoro di Gillespie del 1977 [10] e di altri scienziati nell'ambito della simulazione

[1] `https://github.com/AlchemistSimulator/Alchemist`

chimica. 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 [8] 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 1.1.2 a pagina 4) su cui lavora sono le seguenti:

Molecule 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.

Concentration 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 fonrire informazioni in merito alla posizione dei *Nodi* nello spazio, alla distanza tra loro e al loro vicinato; opzionalmente, 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.

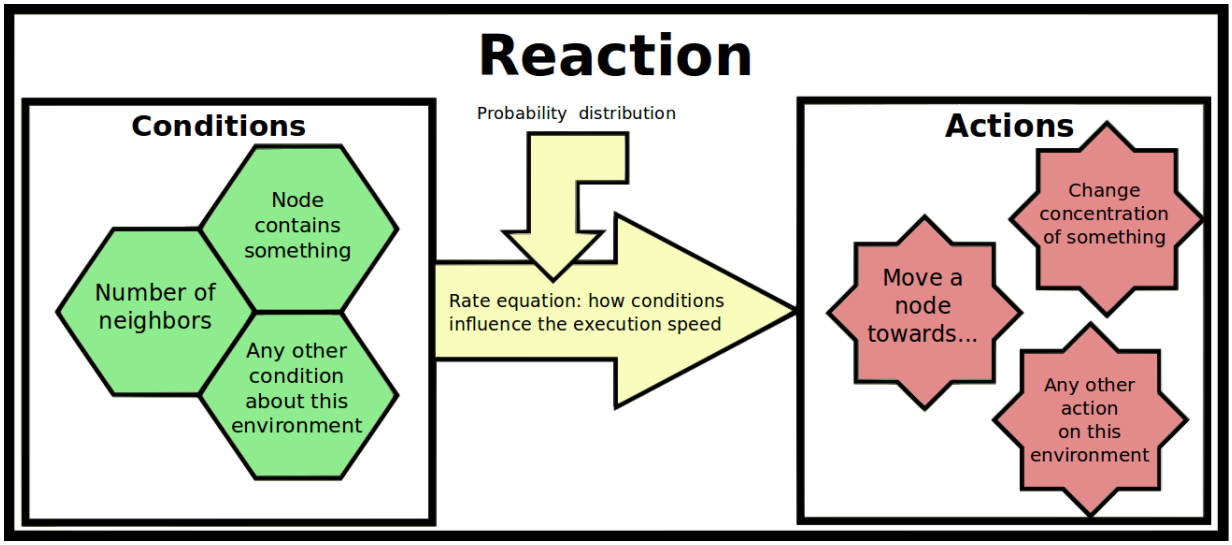

Figura 1.1: La figura, rivisitata da quella disponibile sul sito ufficiale [1], offre una rappresentazione grafica della *Reazione*.

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 1.1.2) 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*.

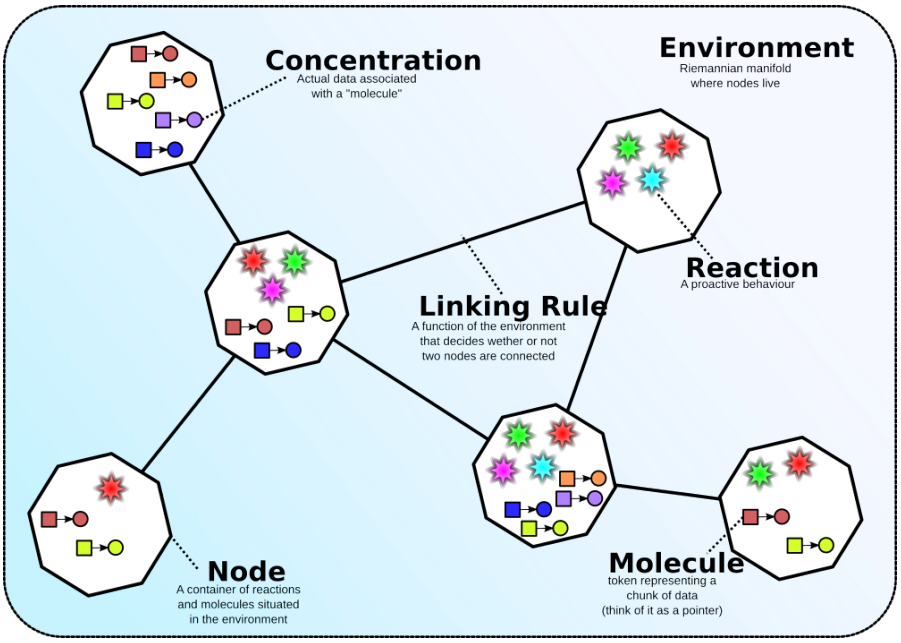

Figura 1.2: La figura, presa dal sito ufficiale [1], offre una rappresentazione grafica delle diverse entità. All'interno di un ambiente, che modella il sistema, si trovano i nodi connessi tra loro attraverso dei collegamenti; ogni nodo è composto da reazioni e molecole, ognuna delle quali ha associata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

1.1.3 Interfaccia utente classica

L'architettura di Alchemist è progettata con paradigma *Model-View-Controller* (MVC) [13], 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* [17]) 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 soprattuto dall'affermarsi degli studi di usabilità [16] e di ergonomia cognitiva di quel periodo.

Più ampio e moderno è invece il concetto di esperienza utente [11] (spesso abbreviata in UX, *User eXperience*): l'ISO 9241-210 [5] 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'usbilità 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 [4], la GUI ha subito un parziale rinnovamento, limitati 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 1.1.3 nella pagina seguente, non sono presenti bottoni di interazione per, ad esempio, avviare o fermare la simulazione o per cambiare la modalità di interazione con la zone in cui viene rappresentato l'environment; questo perché molte possibilità di

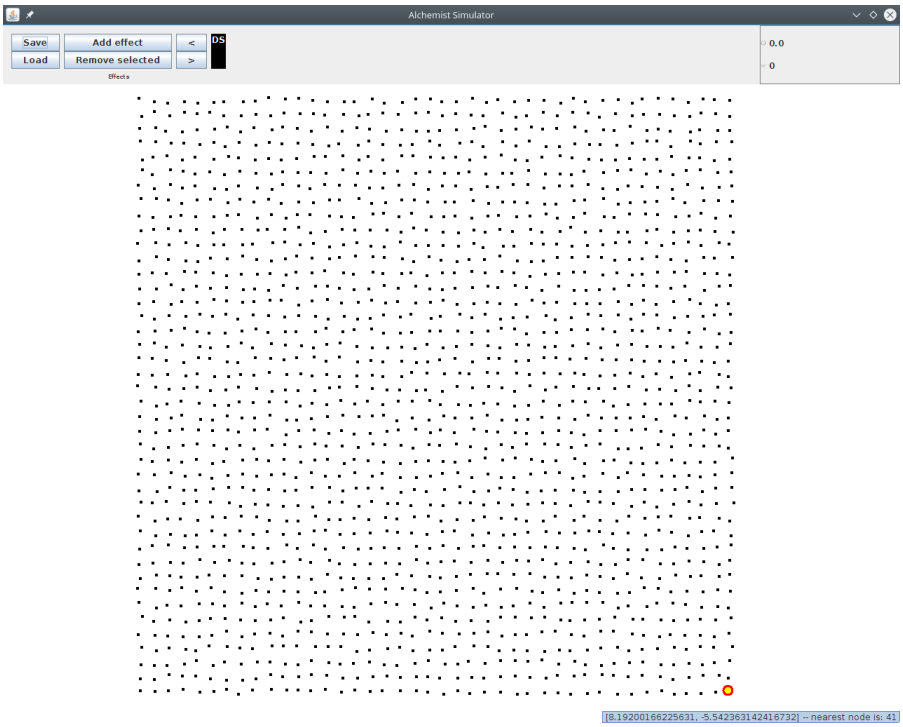

Figura 1.3: Vista principale di una simulazione con l'interfaccia classica

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

Un'ultima criticità che esula dal contesto pratico, ma che rientra appieno nel contesto estetico/edonistico 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 estetica che rispettasse le direttive di un design grafico ben definito (come il Material Design [14] di Google o Modern UI e Fluent Design System [7] di Microsoft) o che mantenesse il design fornito dal 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 1.1.3 nella pagina successiva, 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:

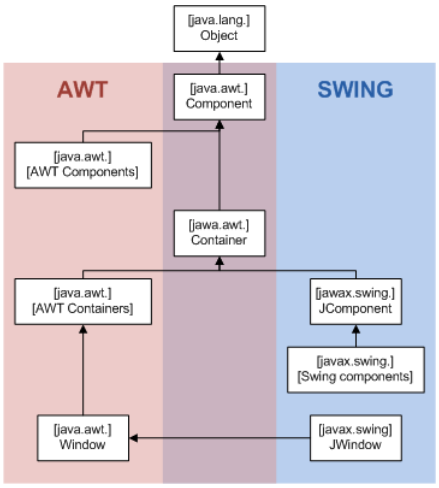

Figura 1.4: Struttura delle classi di Swing e AWT, by Jakub Závěrka (Jakub Závěrka - own work) [Public domain], via Wikimedia Commons

- 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`. 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 rapprsentazioni 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 [15], 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 delle 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 1.2.2 nella pagina seguente, 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 1.2.2 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.

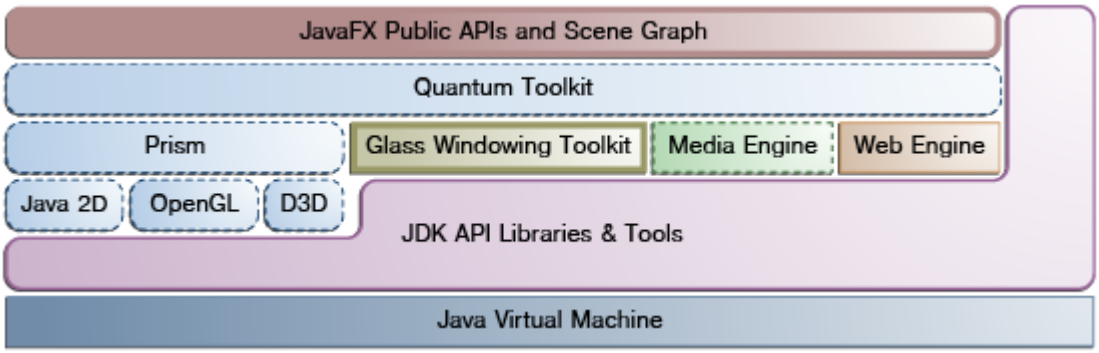

Figura 1.5: La figura, presa dalla documentazione ufficiale di Oracle [12], rappresenta i diversi livelli che caratterizzano il framework JavaFX

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.

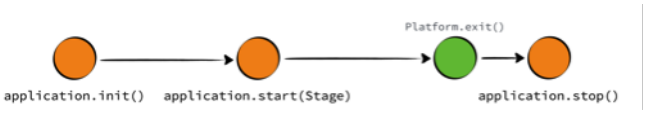

Figura 1.6: La figura rappresenta il ciclo di vita di una applicazione JavaFX

La classe principale per una applicazione JavaFX deve estendere da `javafx.application.Application`. Il metodo `start()` è il punto di ingresso principale: lanciando una 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 l*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 1.2.3:

`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*.

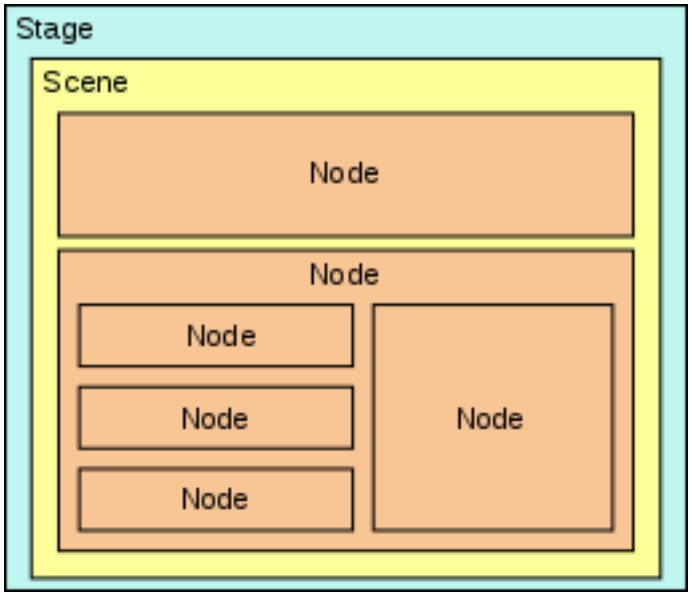

Figura 1.7: Struttura del layout di una applicazione JavaFX, by Stkl [CC BY-SA 4.0], via Wikimedia Commons

`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.

1.2.4 Vantaggi di JavaFX su Swing

1.3 Interfaccia JavaFX per Alchemist: motivazioni

Capitolo 2

Contributo

2.1 Analisi dei requisiti

2.1.1 Requisiti funzionali

2.1.2 Requisiti non funzionali

2.2 Fonti d'ispirazione

2.2.1 Simulatori a scopo videoludico

Universe Sandbox

Universe Sandbox 2

SimCity

2.2.2 Material Design

2.3 Design dell'interfaccia

2.4 Progettazione

2.4.1 La barra inferiore

2.4.2 La struttura a drawer

2.4.3 L'architettura degli effetti

2.5 Dettagli implementativi

Capitolo 3

Conclusioni

3.1 Risultati

3.2 Lavori futuri

Appendice A

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);
}
```

Bibliografia

[1] *Alchemist*. URL: http://alchemistsimulator.github.io/.

[2] 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.

[3] J. Banks et al. *Discrete-Event System Simulation: Pearson New International Edition*. Pearson Higher Ed, 2013.

[4] E. Casadio. «Revisione e refactoring dell'interfaccia utente del simulatore Alchemist». Tesi di dott. URL: http://amslaurea.unibo.it/12310/.

[5] 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).

[6] G. S. Fishman. «Principles of discrete event simulation.[book review]». In: (1978).

[7] *Fluent Design System by Microsoft*. URL: https://fluent.microsoft.com/.

[8] 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.

[9] 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.

[10] D. T. Gillespie. «Exact stochastic simulation of coupled chemical reactions». In: *The journal of physical chemistry* 81.25 (1977), pp. 2340–2361.

[11] 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.

[12] *Java Platform, Standard Edition (Java SE) 8*. URL: https://docs.oracle.com/javase/8/javase-clienttechnologies.htm.

[13] 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.

[14] *Material Design by Google*. URL: https://material.io/.

[15] F. Moritz. «Rich Internet Applications (RIA): A Convergence of User Interface Paradigms of Web and Desktop-Exemplified by JavaFX». In: *University of AppliedScience Kaiserslautern, Deutschland* (2008).

[16] D. A. Norman. *La caffettiera del masochista (The psychology of everyday things)*. Inglese. 1988.

[17] T. C. O'rourke et al. *Graphical user interface*. US Patent 5,349,658. Set. 1994.

[18] 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.

Ringraziamenti