Università degli Studi di Firenze
Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

Tesi di Laurea Triennale in Ingegneria Informatica

Analisi e sviluppo di un componente Java per la simulazione interattiva di reti di Petri stocastiche

Candidato
Tommaso Scarlatti

Relatore
Prof. Enrico Vicario

Correlatore
Ing. Marco Biagi

Anno Accademico 2016-2017

Indice

Ringraziamenti **iii**

Introduzione **i**

1 Preliminari **1**

1.1 Rete di Petri 1

1.1.1 Concetti base 1

1.1.2 Una definizione formale 2

1.1.3 Dinamica della rete 3

1.1.4 Estensioni 4

1.2 ORIS tool 5

1.2.1 Panoramica 5

1.2.2 Caratteristiche 6

1.2.3 Struttura interna 7

2 Analisi **9**

2.1 Studio di fattibilità 9

2.2 Analisi dei requisiti 10

2.2.1 Casi d'uso 11

2.2.2 Analisi della GUI 13

3 Sviluppo **17**

3.1 Sviluppo dell'API 17

3.1.1 Implementazione 17

3.1.2 Dettagli implementativi 20

3.1.3 Testing 22

3.2 Sviluppo della GUI 23

3.2.1 Swing 23

3.2.2 Implementazione 24

3.2.3 Dettagli implementativi 27

4 Conclusioni **34**

Bibliografia **36**

Ringraziamenti

Desidero ringraziare tutti coloro che mi hanno aiutato nella stesura della tesi attraverso consigli, suggerimenti, osservazioni, o semplicemente interessandosi al mio lavoro. E' stato un percorso complesso ma estremamente interessante, formativo, e di cui vado molto fiero.

Ringrazio anzitutto il relatore, il Prof. Enrico Vicario, per avermi proposto il lavoro e per il tempo dedicatomi. Un ringraziamento speciale va al mio correlatore, l'Ing. Marco Biagi, per avermi seguito passo passo in questi mesi ed essere sempre stato chiaro, disponibile ed esauriente in ogni occasione.

Il ringraziamento più grande va ai miei genitori, che mi hanno insegnato il valore del lavoro e del sacrificio e che da ventidue anni mi sostengono in tutto quello che faccio. Senza il loro appoggio non avrei raggiunto questo risultato. Ringrazio mia nonna, che quasi tiene più di me a questa laurea, per essermi stata sempre vicino e ringrazio i miei zii e i miei cugini per il supporto continuo e per dimostrarmi sempre l'affetto che ci lega. Voi siete la mia famiglia, le mie radici e in questo risultato c'è una parte di ognuno di voi.

Un ringraziamento collettivo va a tutti i miei amici, specialmente a quelli di vecchia data, che mi hanno permesso di trascorrere questi tre anni circon-

dato da serenità e simpatia. Non posso però non ringraziare direttamente Matteo, migliore amico e compagno di una vita.

Ringrazio infine tutti quei compagni di corso che sono stati parte fondamentale della mia vita accademica in questi tre anni. Con loro ho condiviso le gioie e le difficoltà della vita universitaria e sono fiero di aver conosciuto così tante persone in gamba e disponibili.

E' stato un percorso lungo e complesso, coronato dal presente lavoro di tesi. Mi ritengo soddisfatto del percorso formativo che ho portato a termine e, in generale, di questi tre anni della mia vita, in cui l'università ha sicuramente avuto un ruolo principale, senza però impedirmi di tenere vivi i miei interessi e le mie passioni.

Introduzione

Con il presente lavoro di tesi si vuole documentare, seguendo pratiche consolidate dell'Ingegneria del Software, la progettazione e lo sviluppo di un token game per reti di Petri all'interno di ORIS, un tool sviluppato presso il Software Technologies Lab (STLAB) dell'Università degli Studi di Firenze.

Le motivazioni che hanno portato alla nascita di questo progetto sono principalmente due: da un lato vi era il desiderio di arrichire il tool ORIS con una nuova feature (per altro già presente in molti altri software di modellazione delle reti di Petri, vedi GreatSPN [1]), dall'altro si voleva trarre beneficio dalle funzionalità che un componente come il token game può offrire. Si tratta di un'interfaccia grafica interattiva che mostra l'evoluzione della marcatura di una rete di Petri analizzata, e che permette di valutarne alcune proprietà quali la raggiungibilità in modo immediato.

Il progetto di tesi in questo documento è stato strutturato considerando i tre momenti chiave che hanno portato al rilascio finale del token game.

Nel primo capitolo vengono presentati brevemente i concetti preliminari necessari ad una corretta comprensione del lavoro svolto. Nella prima sezione viene presentata la rete di Petri come formalismo grafico/matematico e vne

engono citate alcune estensioni. Nella seconda sezione si offre una panoramica sul tool ORIS, mostrandone le funzionalità principali.

Nel secondo capitolo si documenta la fase di analisi dei requisiti facendo ricorso a diagrammi UML dei casi d'uso e a mockups dell'interfaccia, concentrandosi prima sull'API (Application Program Interface) e successivamente sulla GUI (Graphical User Interface). All'analisi vera è propria viene fatto precedere uno studio di fattibilità del progetto, in termini di tempo e conoscenze da integrare.

Il terzo ed ultimo capitolo documenta la fase di implementazione del token game in Java. La prima sezione è interamente dedicata allo sviluppo dell'API, in cui si riporta in dettaglio il percorso seguito attraverso l'utilizzo di diagrammi UML di classe e di sequenza. Un paragrafo è dedicato alla fase di testing condotta utilizzando la libreria JUnit. La seconda sezione mostra le problematiche affrontate durante lo sviluppo della GUI utilizzando la libreria Swing. Vengono inoltre spiegati in dettaglio i passi principali dello sviluppo che hanno richiesto l'impiego di alcuni design pattern.

Capitolo 1

Preliminari

1.1 Rete di Petri

1.1.1 Concetti base

Una rete di Petri è un formalismo grafico/matematico per la modellazione di sistemi dinamici a eventi discreti. Si tratta di uno strumento assai versatile: da un lato può essere utilizzata come strumento di comunicazione visiva, alla stregua dei diagrammi di flusso e dei diagrammi a blocchi, dall'altro permette l'utilizzo di modelli matematici che ne governino il comportamento. Storicamente parlando, fu inventata nel 1962 dal matematico e informatico tedesco Carl Adam Petri, durante la propria tesi di dottorato. [2]

Una rete di Petri consiste di posti, transizioni e archi diretti. Possono esserci archi tra posti e transizioni, ma non tra posti e posti o transizioni e transizioni. Questo è dovuto al fatto che la rete è un grafo bipartito, ossia un particolare tipo di grafo caratterizzato da un insieme dei nodi che può essere partizionato in due sottoinsiemi, tali che ogni vertice di una di queste due parti è collegato solo a vertici dell'altra. Ogni posto può contenere un certo

numero di token o marche. La distribuzione dei token sui posti della rete viene definita **marking**. In un'ottica di modellazione, i posti rappresentano le condizioni, mentre le transizioni rappresentano gli eventi. Quest'ultime hanno un certo numero di posti in input ed in output che rappresentano rispettivamente le pre-condizioni e le post-condizioni. La regola principale che governa la rete è relativa alle transizioni, ed è definita regola di scatto (*firing rule*). Una transizione è infatti abilitata (*enabled to fire*) se può scattare, ossia quando tutti i posti in ingresso sono marcati. Quando la transizione scatta, consuma i token dei posti in input e marca tutti posti in output secondo uno specifico peso.

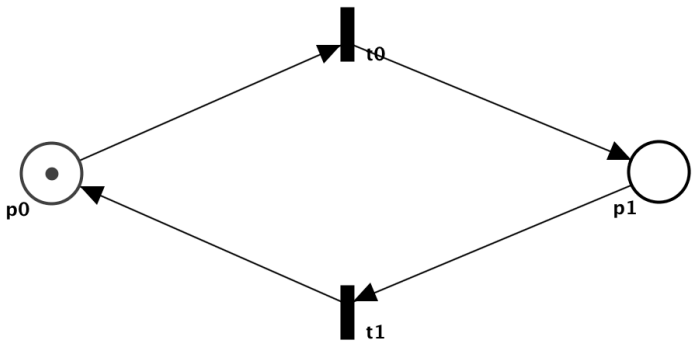

Figura 1.1: Una rete di petri (P/T) ordinaria. La transizione t0 è abilitata.

1.1.2 Una definizione formale

Una rete di Petri è una quintupla $PN = (P, T, F, W, M)$, ove:

1. $P = \{p_1, p_2, \ldots, p_m\}$ è un insieme finito di posti
2. $T = \{t_1, t_2, \ldots, t_n\}$ è un insieme finito di transizioni
3. $F \subset (T \times U) \cup (U \times T)$ è un insieme di flussi relazionali chiamati "archi"

4. $W : F \to \{1, 2, 3, \ldots\}$ è la funzione di peso

5. $M : P \to \{0, 1, 2, \ldots\}$ è il marking. Spesso viene rappresentato come un vettore di m componenti, una per ogni posto. La p-esima componente di M, espressa come $M(p)$, è il numero di token contenuti nel posto p.

Una transizione senza nessun posto in input è definita *source transition*, mentre una senza nessun posto in output è chiamata *sink transition*. La prima è abilitata incondizionatamente, la seconda, una volta scattata, consuma tokens ma non ne produce alcuno. Una coppia (p, t) dove p è un posto e t una transizione si definisce un *self loop* se p è sia un posto di input che di output per t. Una rete di Petri è definita pura (*pure*) se non ammette al suo interno dei self-loop. Una rete di Petri si definisce invece ordinaria (*ordinary*) qualora tutti i pesi degli archi siano pari ad 1.

1.1.3 Dinamica della rete

Una rete di Petri può quindi rappresentare un insieme di stati discreti di cardinalità infinita con un numero finito di posti. La sua evoluzione è legata all'occorrenza di eventi. Come è stato delineato nei paragrafi precedenti, si può avere un'occorrenza di evento quando (almeno) una transizione è abilitata. In corrispondenza di un evento avviene un cambio di marcatura della rete. [3] Formalmente, una transizione t_j è abilitata quando:

$$M(p_i) \geqslant W(p_i, t_j) \forall p_i \in I(t_j)$$

dove $I(t_j)$ è l'insieme dei posti di input della transizione t_j. Come viene modificata la marcatura quando scatta una transizione abilitata?

- in tutti i posti in ingresso alla transizione *si consumano* un numero di token pari al peso dell'arco che collega il posto alla transizione

- in tutti i posti in uscita dalla transizione *si generano* un numero di token pari al peso dell'arco che collega la transizione al posto

In questo modo è possibile far comparire/scomparire risorse all'interno della rete. Se infatti la somma di tutti i pesi degli archi di input di una transizione è maggiore della somma dei pesi di quelli di ouput, allora si generano token, viceversa si eliminano.

Cosa succede se una particolare marcatura abilita più transizioni insieme? Nelle reti standard, per convenzione, si assume che possa scattare una sola transizione alla volta, scelta a caso. Tra i programmi di simulazione delle reti di Petri vi è un'ulteriore regola assai diffusa, che consiste nel far scattare insieme tutte le transizioni abilitate e non in conflitto. Per quelle in conflitto, ossia in cui si hanno uno più posti di ingresso in comune, ma aventi un numero di token non sufficienti a farle scattare tutte, vi sono molte varianti di scelta: scelta casuale, priorità lessicografica, priorità con memoria...

1.1.4 Estensioni

Esistono molte estensioni della rete di Petri. Alcune di esse sono completamente retro-compatibili (ad esempio le reti di Petri colorate) con la rete di Petri originaria, ossia possono essere trasformate con opportune formule matematiche nell'originale rete, altre invece aggiungono proprietà che non non modellabili nella rete di Petri d'origine (ad esempio le reti di Petri tempificate). Ecco alcune estensioni del formalismo base posto/transizione:

- **Reti di Petri colorate:** in una rete di Petri standard i token sono indistinguibili tra di loro. In una rete di Petri colorata ogni token ha un valore. Il valore è tipizzato e può essere testato e manipolato attraverso un linguaggio di programmazione funzionale.

- **Reti di Petri con priorità:** suddividono le transizioni in gruppi di priorità. Una transizione, anche se abilitata, non può scattare se ve n'è una abilitata con più alta priorità.

- **Reti di Petri ad oggetti:** in questo particolare formalismo si utilizzano oggetti complessi (definiti in quelache linguaggio object-oriented) come tokens. Una rete di Petri di questo tipo può contenere anche diverse reti come suoi token, stabilendo una gerarchia di reti di Petri innestate che comunicano tra di loro sincronizzando le transizioni su diversi livelli.

- **Reti di Petri temporizzate:** sono reti di Petri nelle quali ad ogni transizione può essere associato un range di tempo non deterministico in cui la transizione, se abilitata, può scattare.

- **Reti di Petri stocastiche:** le reti temporizzate non forniscono alcuna informazione aggiuntiva sul fatto che una transizione abbia più probabilità o meno di scattare in un determinato istante del proprio range. Le reti di Petri Stocastiche non hanno tempi non deterministici, ma la durata di una transizione è descritta da una distribuzione di probabilità (uniforme, esponenziale,...)(Fig. 1.2).

1.2 ORIS tool

1.2.1 Panoramica

ORIS è un tool sviluppato dal *Software Technologies Lab* (STLAB) del Dipartimento di Ingegneria dell'Informazione dell'Università Degli Studi di Firenze. Permette la modellazione e l'analisi di sistemi reattivi temporizzati

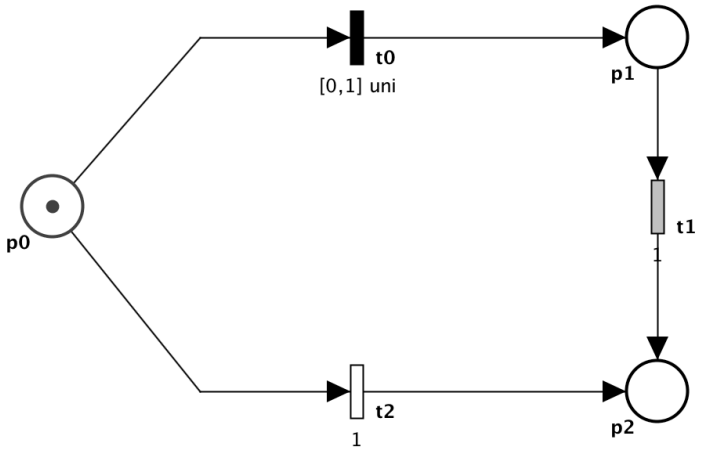

Figura 1.2: Una rete di Petri stocastica

basati su varie classi di reti di Petri. Centrale è la capacità del tool di supportare tecniche di analisi qualitativa e quantitativa basate su un'enumerazione simbolica dello spazio degli stati. [4]

1.2.2 Caratteristiche

- **Editor grafico delle reti di Petri:** Le reti di Petri possono essere modificate graficamente, associando ad ogni transizione un tempo minimo e massimo per scattare (reti di Petri temporizzate), o una funzione di densità di probabilità con supporto finito o infinito (reti di Petri stocastiche). L'editor inoltre include features quali: undo, cut-and-paste, zoom, magnetic grid, alignment and even spacing of elements, SVG export, sticky notes.

- **Analisi non-deterministica delle reti di Petri:** il grafico della classe di stato di reti di Petri temporizzate può essere computato e visualizzato graficamente. Per reti di Petri stocastiche e temporizzate,

la classe di stato può evidenziare i punti di rigenerazione ed escludere gli scatti con probabilità nulla.

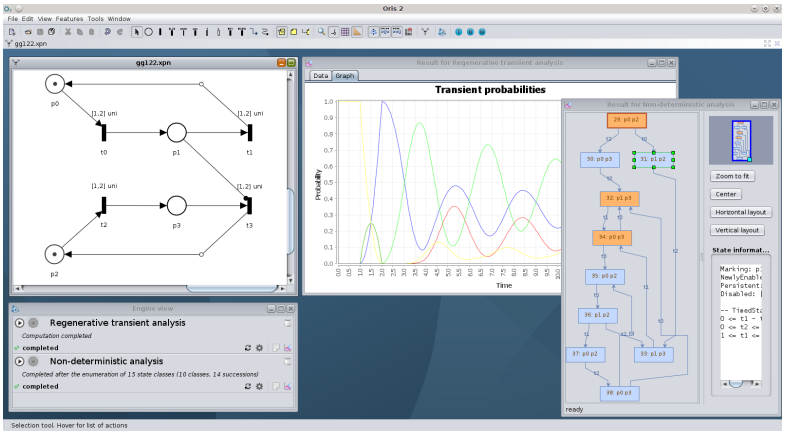

Figura 1.3: Interfaccia di ORIS

- **Analisi transiente di reti di Petri stocastiche non-markoviane:** le probabilità dello stato transiente di reti di Petri stocastiche e temporizzate può essere computato attraverso l'enumerazione delle classi di stato stocastiche (distribuzione dei time-to-fires dopo ogni scatto) entro un limite di tempo. L'analisi rigenerativa combina l'enumerazione delle classi di stato stocastiche fino ai punti di rigenerazione con sistemi di equazioni integrali, al fine di sfruttare la struttura ripetitiva del processo stocastico sottostante.

1.2.3 Struttura interna

Oris è un progetto Java Maven [5] formato da vari moduli. Ogni modulo ha una sua specifica funzione che brevemente introduciamo qui di seguito:

- **OrisRunner:** Serve ad avviare l'applicazione.

- **OrisSirio:** Funge da interfaccia da ORIS verso Sirio in quanto il primo può essere utilizzato su qualsiasi analizzatore. La classe Abstract Engine gestisce il pannello dove sono presenti le varie simulazioni. La classe PNDefinition crea la rete di Petri Sirio a partire da quella di Oris.
- **OrisStochastic:** Rappresentazione grafica delle reti di Petri stocastiche.
- **OrisTime:** Rappresentazione grafica delle reti di Petri temporizzate.
- **OrisPNEditor:** Definisce il documento di tipo "rete di Petri", definendo quindi tutte le caratteristiche di tali documenti e come appare la loro interfaccia grafica di editing.

Di seguito mostriamo un diagramma dei componenti di ORIS (Fig. 1.4). Sirio è il motore per le analisi ed espone delle API utilizzate da ORIS. La Token Game API fa utilizzo sia di Sirio che della PetriNetLib, ed è a sua volta utilizzata dall'interfaccia grafica del Token Game presente nel tool ORIS.

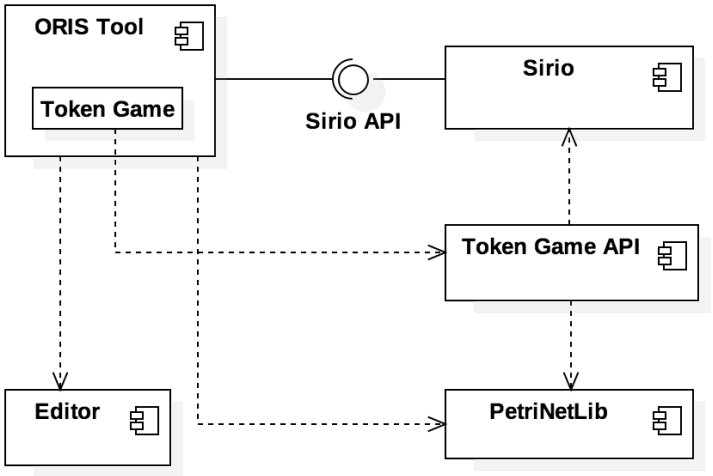

Figura 1.4: Diagramma dei componenti di ORIS

Capitolo 2

Analisi

Si vuole sviluppare un token game, ossia aggiungere una feature ad ORIS in modo da garantire all'utente la possiblità di visualizzare e governare l'evoluzione della marcatura della rete. Per farlo, in primo luogo, si è studiato come sviluppare l'API (Application Program Interface) e solo in seguito si è passati allo studio della GUI (Graphical User Interface) . Questo ha permesso di separare la logica di dominio del token game dalla sua rappresentazione grafica, garantendo la possibilità di accedere alle sue funzioni anche esclusivamente tramite API.

2.1 Studio di fattibilità

Inizialmente si è eseguito uno studio di fattibilità del progetto in termini di tempo e conoscenze da integrare a quelle acquisite durante i corsi. E' emersa fin da subito la necessità di approfondire le funzionalità messe a disposizione dell'IDE Eclipse per velocizzare lo sviluppo e garantire una sua terminazione entro la data stabilita (luglio 2017, complessivamente cinque mesi di lavoro). Inoltre si è avvertito il bisogno di ampliare le conoscenze e la

dimestichezza con l'utilizzo dei design pattern [6] orientati al linguaggio Java e con i cosiddetti "Java idioms" [7] al fine di elevare la correttezza e la qualità del codice cercando di uniformarsi a quello preesistente di ORIS. Il codice di quest'ultimo, inoltre, fa largo utilizzo della cosiddetta Context Dependency Injection (CDI), il che ha reso necessaria una documentazione adeguata sull'argomento seguendo alcune lezioni dedicate del corso *Software Architecture and Methodologies* tenuto dal Prof. E. Vicario presso l'Università degli Studi di Firenze. [8]

2.2 Analisi dei requisiti

L'analisi dei requisiti funzionali per lo sviluppo del token game in ORIS è stata condotta tramite una pratica di successo e consolidata nella rappresentazione di quest'ultimi: i **casi d'uso**. I casi d'uso danno struttura e metodo all'idea di catturare i requisiti funzionali a partire da esempi e scenari di interazione dell'utente con il sistema. [9]

Un caso d'uso è avviato da un attore, ossia un entità esterna al sistema (nel nostro caso avremmo un solo utente) e si conclude con successo con il raggiungimento dell'obiettivo preposto. [10] L'insieme dei casi d'uso per disegnare il diagramma è stato definito assumendo un livello di astrazione "User goal" (o "sea level" utilizzando la metafora del mare), ossia vengono prese in considerazione le interazioni circoscritte tra un attore e il sistema che tipicamente hanno durata compresa tra un paio di minuti e mezz'ora. [11]

Quindi, in primo luogo, è stato necessario stabilire in quanti e quali modi l'attore avrebbe potuto interagire con il sistema. Ne sono stati evidenziati quattro, riportati nel diagramma UML dei casi d'uso di Fig. 2.1.

Oltre ai quattro casi d'uso principali, ve n'è un quinto che utilizza la

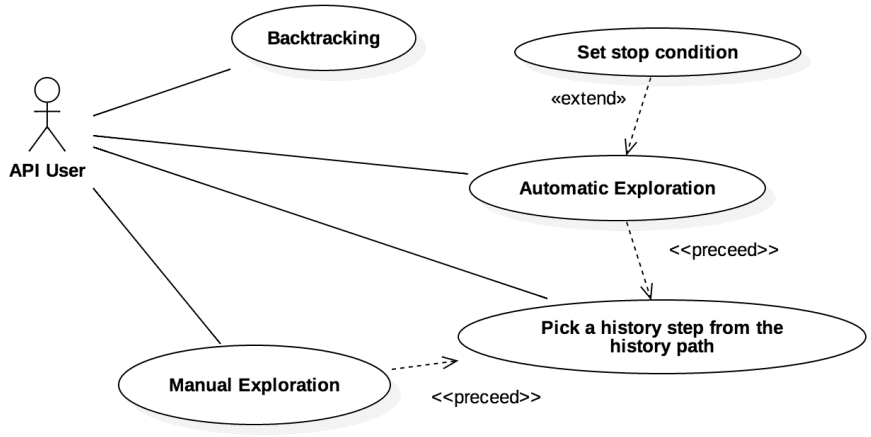

Figura 2.1: Diagramma UML dei casi d'uso dell'API

specifica *«extend»* che implica la definizione di un punto di estensione al caso d'uso. Vediamoli in dettaglio.

2.2.1 Casi d'uso

- **Manual exploration:** la funzione di esplorazione manuale permette all'utente di scegliere una transizione da far scattare tra quelle disponibili. Le transizioni disponibili, come descritto nel primo capitolo, saranno quelle con tutti i posti in ingresso marcati.

- **Automatic exploration:** l'esplorazione manuale permette all'utente di decidere una sequenza ordinata di transizioni da far scattare. Tuttavia questa operazione può diventare tediosa al crescere della lunghezza della sequenza, oppure l'utente potrebbe essere interessato ad osservare l'evoluzione "casuale" della rete, ossia senza specificare direttamente le transizioni da far scattare ad ogni passo. L'esplorazione automatica assolve questa necessità permettendo di settare due parametri:

  - **Steps:** numero massimo di transizioni che possono scattare in sequenza
  - **Stop condition:** particolare marcatura della rete che se raggiunta termina l'esplorazione

  In caso di più transizioni disponibili a scattare, che meccanismo di scelta usa il token game? Qualora si tratti di una rete di Petri ordinaria o temporizzata (timed) si utilizza una distribuzione di probabilità uniforme, ossia tutte le transizioni abilitate hanno la stessa probabilità di scattare. Se la rete di Petri è stocastica, ad ogni transizione è associata una probabilità di scattare, detta *firing probability*.

- **Backtracking:** tramite questa funzionalità si dà la possibilità all'utente di fare backtracking, ossia di risalire all'indietro nella sequenza di transizioni scattate per il numero di passi specificato. Da questa si evince la necessità di tenere traccia della "storia" dell'evoluzione della rete.

- **Pick a history step:** la possibilità di tener traccia della storia dell'evoluzione della rete porta a definire un'ulteriore possibilità di interazione per l'utente: selezionare direttamente un passo della storia per vedere quale fosse lo stato della rete, quale transizione è scattata e con quale probabilità. Per ovvie ragioni, non si può selezionare un passo della storia se ancora non ve n'è alcuno. Di conseguenza sia l'esplorazione automatica che quella manuale devono precedere logicamente la selezione di uno step (*«preceed»*).

2.2.2 Analisi della GUI

Una volta condotta l'analisi dei possibili casi d'uso dell'API, si è passati a delineare le caratteristiche dell'interfaccia grafica, tenenedo ben presenti i requisiti funzionali (implementare tutte le funzioni dell'API) e qualitativi (dividere logicamente le varie funzionalità, garantire una corretta visione della rete). Lo strumento utilizzato in questa fase dell'analisi è stato quello dei *mockups*, bozze dell'interfaccia grafica esposta all'utente. Si tratta di uno strumento assai utile: a fronte di un basso costo di creazione e di evoluzione, l'immediata concretezza del mockup stimola il coinvolgimento di esperti di dominio e degli stessi utenti finali. Inoltre i mockups sono estremamente versatili riguardo al loro utilizzo nelle varie fasi del ciclo di vita del software. Per la loro creazione si è fatto affidamento al tool Balsamiq Mockups. [12]

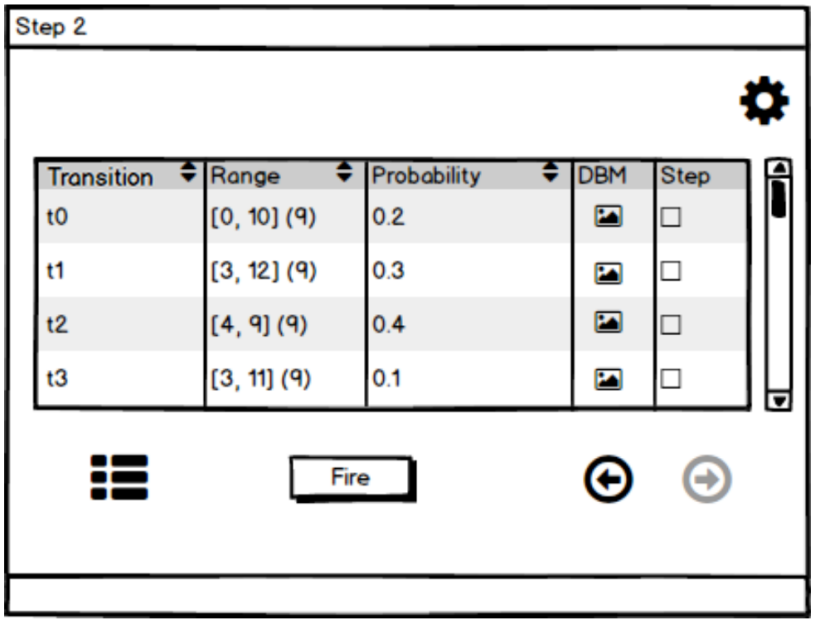

Figura 2.2: Primo mockup realizzato in fase di analisi del progetto

Il mockup di Fig. 2.2 è stato creato in una fase embrionale del progetto. Si è rivelato assai utile per fissare le idee sulla futura implementazione e sul-

la fattibilità della stessa, tuttavia, proprio per la sua naturale inclinazione all'evoluzione, è stato cambiato radicalmente. Nonostante ciò, si può osservare come siano già presenti le proprietà di una transizione che si desiderano visualizzare: il nome della transizione, il range di tempo in cui la transizione è abilitata a scattare (se temporizzata) e la proprabilità di scattare (se stocastica) e la DBM (Difference Bound Matrix). Il pulsante a sinistra avrebbe aperto un menù di opzioni avanzate (poi deprecato), *Fire* avrebbe permesso ad una transizione di scattare (nel caso fosse stata selezionata con l'apposita checkbox), mentre i pulsanti sulla destra avrebbero permesso di muoversi avanti/indietro nei vari passi della storia. La versione definitiva del mockup (Fig. 2.3) presenta notevoli cambiamenti ed una struttura assai più compatta.

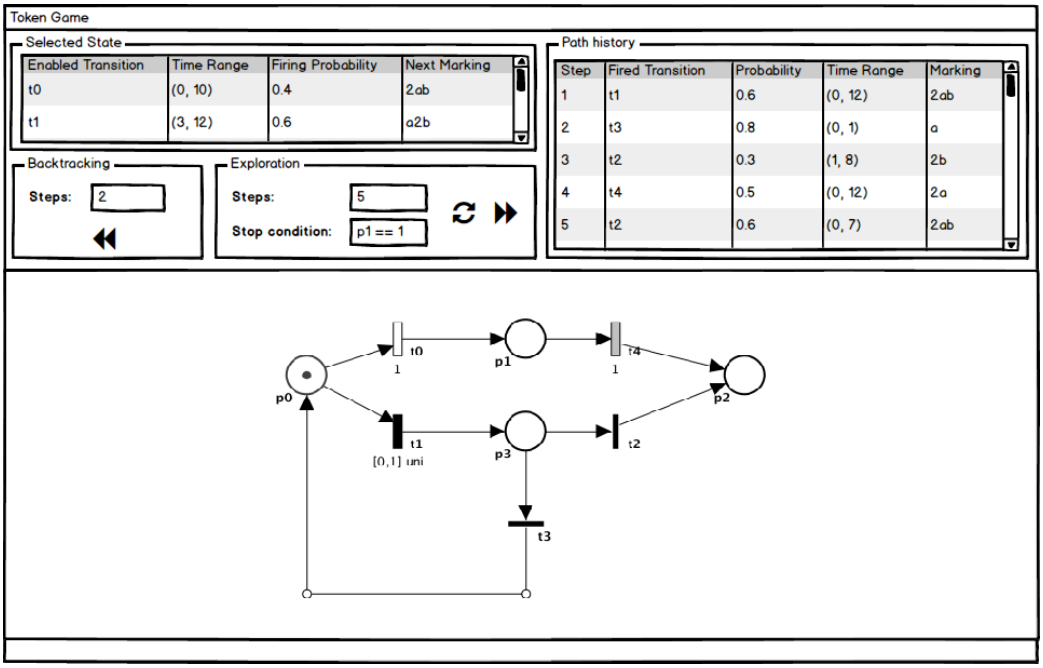

Figura 2.3: Mockup definitivo del token game

La finestra è stata divisa in tre unità logiche principali, qui brevemente

descritte:

- **Exploration:** permette di visualizzare lo stato corrente della rete e riunisce tutte le operazioni che possono modificarla. Si divide a sua volta in:
  - Selected state
  - Backtracking
  - Automatic exploration
- **Path history:** rappresenta il "cammino" dell'esplorazione. Contiene tutti gli stati in cui è transitata la rete e permette, tramite click da parte dell'utente, di selezionare un particolare passo del cammino.

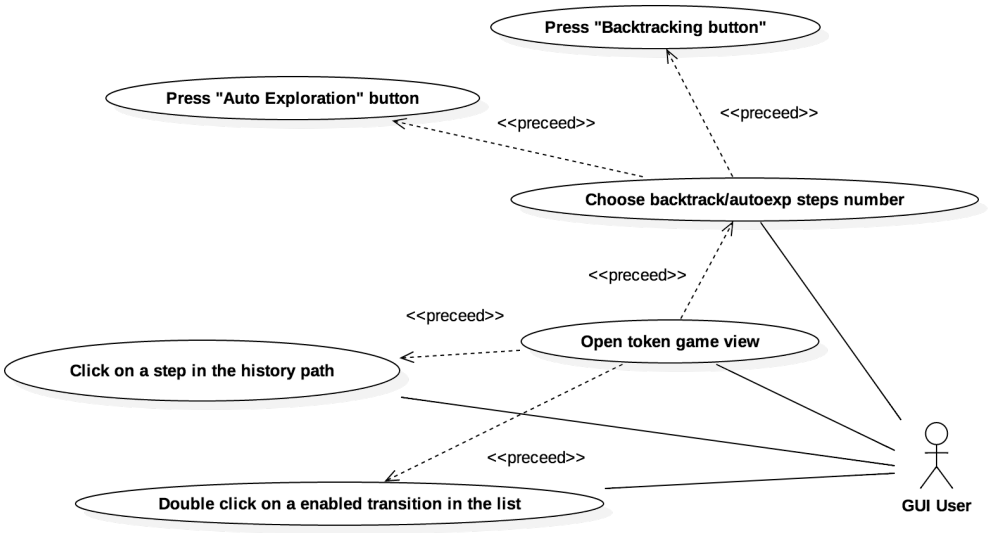

Figura 2.4: Diagramma UML dei casi d'uso della GUI

- **Petri net:** metà della finestra è garantita ad un'adeguata visualizzazione della rete di Petri. Quest'ultima, creata precedentemente attra-

verso l'editor grafico di ORIS, mostra la marcatura associata allo stato selezionato.

Alla stregua di quanto fatto per l'API, anche per la GUI si è realizzato un diagramma dei casi d'uso (Fig. 2.4). Questo può essere collegato al diagramma in Fig. 2.1 in quanto ogni interazione dell'utente con l'interfaccia grafica viene mappata ad un metodo dell'API.

Capitolo 3

Sviluppo

Una volta conclusa la fase di analisi si è passati allo sviluppo del progetto in Java. Come evidenziato nel capitolo precedente, lo sviluppo ha seguito due fasi principali: l'implementazione dell'API e quella della GUI.

3.1 Sviluppo dell'API

3.1.1 Implementazione

Dopo aver preso confidenza con l'organizzazione interna di ORIS e dei suoi componenti, si è passati alla creazione di un diagramma UML delle classi che ha permesso di fissare le idee sviluppate durante la fase di analisi. Il punto di forza del class diagram è proprio la sua capacità nel catturare l'organizzazione delle classi in modo immediato, mediante l'allocazione e la partizione delle responsabilità tra esse. La visione d'insieme che fornisce il diagramma, tuttavia, è una visione prettamente statica. Per questo, nel corso dello sviluppo, sono stati creati dei diagrammi di sequenza al fine di documentare l'interazione dinamica tra le varie classi.

Il diagramma di classe dell'API (Fig. 3.1) presenta una spiccata centralità nella classe *TokenGameExploration*, la classe madre del progetto. Quest'ultima riunisce tutte le operazioni attuabili sulla rete e tiene traccia al suo interno della rete stessa e dello stato selezionato.

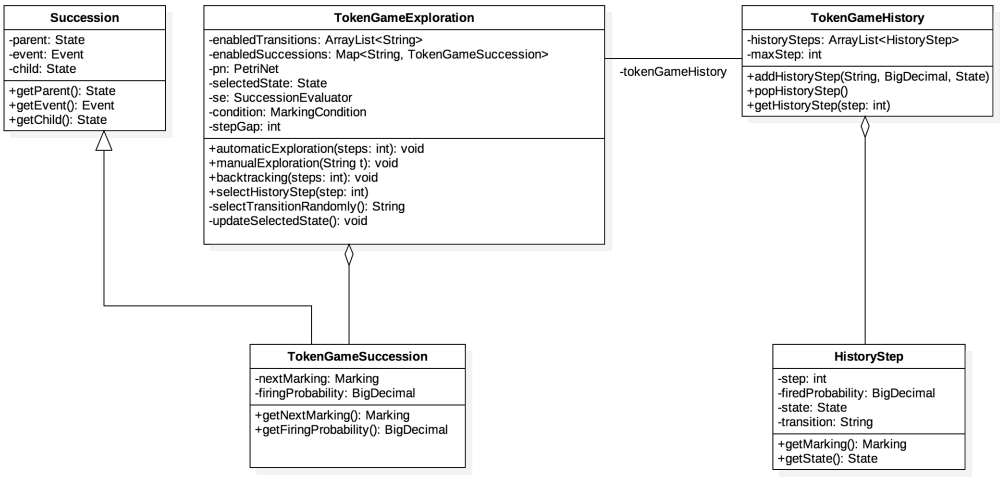

Figura 3.1: Class diagram dell'API

Andiamo ora a vedere in dettaglio tutte le classi dell'API e le interazioni che intercorrono tra di esse:

- **TokenGameExploration:** è la classe centrale del progetto. Tra gli attributi principali notiamo: la rete di Petri, lo stato selezionato, il *SuccessionEvaluator* (una classe di Sirio che permette il calcolo del nuovo stato a seguito di una transizione scattata), l'eventuale condizione di stop, la lista delle transizioni e delle successioni abilitate. Soffermiamoci in dettaglio su quest'ultimo attributo: *enabledSuccession*. Si tratta di una mappa che collega una transizione (abilitata) ad un oggetto del tipo *TokenGameSuccession*. La classe di quest'ultimo eredita direttamente dalla classe *Succession* di Sirio. In questo modo (come richiesto

dal mockup di Fig. 2.3) è possibile mostrare quale sarà il marking futuro della rete se una determinata transizione scatta, mediante un calcolo preventivo della successione. Tra i metodi principali troviamo quattro metodi pubblici che rappresentano le funzionalità delineate nella fase di analisi. I due metodi privati, invece, forniscono le seguenti funzionalità d'appoggio:

  - *selectTransitionRandomly()* restituisce una stringa che rappresenta una transizione scelta a caso tra quelle abilitate. L'algoritmo varia in funzione del tipo di rete di Petri analizzata.
  - *updateSelectedState()* aggiorna gli attributi *enabledSuccession* e *enabledTransitions* in funzione del nuovo stato raggiunto.

- **TokenGameSuccession:** estende la classe Succession già presente in ORIS arricchendo i suoi attributi con il marking futuro e la firing probability. La classe rappresenta appunto una "successione" tra uno stato padre (parent) e uno stato figlio (child) generata dall'occorrenza di un evento (event). Tutte le transizioni abilitate saranno gli eventi degli oggetti TokenGameSuccession contenuti nella mappa *enabledSuccession*, e lo stato padre sarà comune a tutti gli oggetti.

- **TokenGameHistory:** questa classe rappresenta il container di tutti i passi del cammino dell'esplorazione della rete. E' contenuta a sua volta all'interno della classe TokenGameExploration, che si preoccupa di instanziarla appoggiandosi ad essa per eseguire le proprie operazioni. E' modellata come una pila a stack gestita con una politica FIFO [13] (le esplorazioni aggiungono passi, il backtracking li elimina).

- **HistoryStep:** rappresenta un singolo passo del cammino. Come è possibile notare dal diagramma di Fig. 3.1, al suo interno è memorizzato il numero dello step e lo stato associato, parametro fondamentale per recuperare informazioni cruciali quali la marcatura della rete e la lista delle transizioni abilitate a fronte di una richiesta da parte dell'utente.

3.1.2 Dettagli implementativi

Per scegliere in modo casuale la transizione da far scattare ad ogni passo di un'esplorazione automatica si è utilizzata un'implementazione del pattern *Strategy*. Il metodo *selectTransitionRandomly()*, infatti, può avere due implementazioni differenti a seconda del tipo di rete di Petri trattata. Nel caso sia una rete ordinaria o temporizzata, si numerano le transizioni da 0 ad n e si genera casualmente un numero compreso in quel range. Il numero generato corrisponderà ad una transizione che verrà fatta scattare. Nel caso di una rete di Petri stocastica, ad ogni transizione abilitata è associata una probabilità di scattare normalizzata. Come è possibile notare in Fig. 3.2, le transizioni abilitate (in questo caso t0, t1, t2 e t3) vengono disposte in ordine in un intervallo compreso tra 0 e 1.

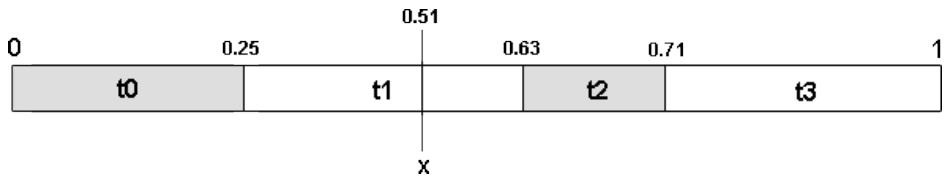

Figura 3.2: Meccanismo di selezione casuale in una rete di Petri stocastica

L'ampiezza di ogni intervallo è direttamente proporzionale alla probabilità associata alla transizione. Viene generato un numero casuale x compreso tra 0 ed 1. L'intervallo in cui ricade (in questo caso t1) determina la transizione che verrà fatta scattare.

Vediamo ora in dettaglio il metodo *automaticExploration(int steps)* che permette l'esplorazione automatica della rete di Petri analizzata. Per mostrare come opera il metodo si è fatto ricorso ad un diagramma di sequenza (Fig. 3.3). Un diagramma di sequenza, infatti, descrive la collaborazione di un gruppo di oggetti che devono implementare collettivamente un comportamento.

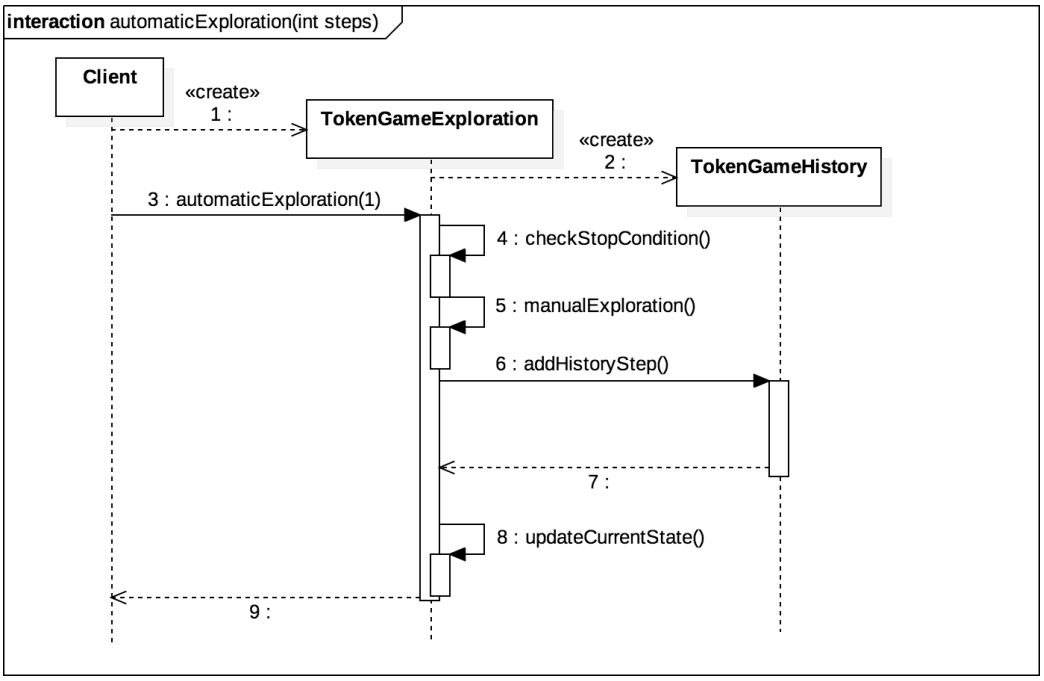

Figura 3.3: Sequence diagram del metodo automaticExploration()

Il client chiama il metodo sull'oggetto TokenGameExploration instanziato in precedenza. Qualora all'attributo *stopCondition* sia stato assegnato un valore, in primo luogo si controlla se la condizione è verificata, ossia se la marcatura corrente la soddisfa (nel diagramma abbiamo supposto di no). Viene poi invocato il metodo *manualExploration()* che a sua volta chiama *addHistoryStep()* sull'oggetto TokenGameHistory. Quest'ultimo metodo va ad

aggiungere un nuovo passo alla storia del cammino, restituendo il nuovo stato raggiunto dall'esplorazione. Infine, proprio lo stato restituito viene utilizzato dal TokenGameExploration per chiamare il metodo *updateCurrentState()* che, come abbiamo visto nelle sezioni precedenti, aggiorna tutti gli attributi della classe alla luce del nuovo stato raggiunto. L'interazione delineata nel diagramma di sequenza fa riferimento al primo passo dell'esplorazione automatica. I successivi sono esattamente analoghi.

3.1.3 Testing

Durante l'implementazione dell'API si è fatto uso del framework JUnit, perseguendo l'idea di sviluppo guidato da test (in inglese Test Driven Development). [14] Questo ha permesso di verificare la correttezza di ogni singolo metodo in corso d'opera, velocizzando notevolmente lo sviluppo. Il test di un software, infatti, viene effettuato a più livelli: al livello più basso il test di unità verifica se una funzione, un modulo o una classe siano corretti rispetto al loro presunto utilizzo. In particolare nel caso dell'API si è svolto un testing funzionale, detto anche *black box testing*, eseguito sulla base dei requisiti funzionali ma tralasciando la struttura interna del codice. [15]

Di seguito si riporta il codice del metodo di testing *wrongTransitionTest()*, che verifica che venga lanciata un'eccezione al momento della chiamata del metodo manualExploration() su una transizione non abilitata a scattare.

```
@Test(expected=IllegalArgumentException.class)
public void wrongTransitionTest() {
   tge.manualExploration("t10");
}
```

3.2 Sviluppo della GUI

Le interfacce di ORIS sono state realizzate tramite Java Swing, in particolare sono state generate usando il tool grafico di Eclipse: Window Builder. [16] Per lo sviluppo dell'interfaccia grafica del token game si è fatto lo stesso, ai fini di rendere il progetto ORIS il più uniforme possibile.

3.2.1 Swing

Swing è un framework per Java, appartenente alle Java Foundation Classes (JFC) e orientato allo sviluppo di interfacce grafiche. Swing inoltre implementa e fornisce una libreria di componenti grafici (widgets) indipendente dalla piattaforma sottostante e basata sul pattern architetturale MVC (Model/View/Controller). [17] È un'estensione del precedente Abstract Window Toolkit (AWT). La differenza principale tra i due è che i componenti Swing sono scritti completamente in codice Java. Di seguito si rappresenta la gerarchia di base della libreria (Fig. 3.4).

Ogni applicazione che utilizza componenti Swing ha almeno un top-level container che rappresenta la radice di una gerachia di contenimento ad albero. Ogni top-level container ha un *content pane* che contiene (direttamente o indirettamente) i componenti visibili della GUI.

I componenti con cui possono essere popolati i top-level container sono tra i più vari: pannelli, bottoni, barre di scorrimento... Questi sono organizzati dal *Layout Manager*, un oggetto che gestisce il loro dimensionamento e posizionamento. Esistono numerose implementazioni predefinite di *LayoutManager*, ciascuna corrispondente ad una particolare politica di posizionamento.

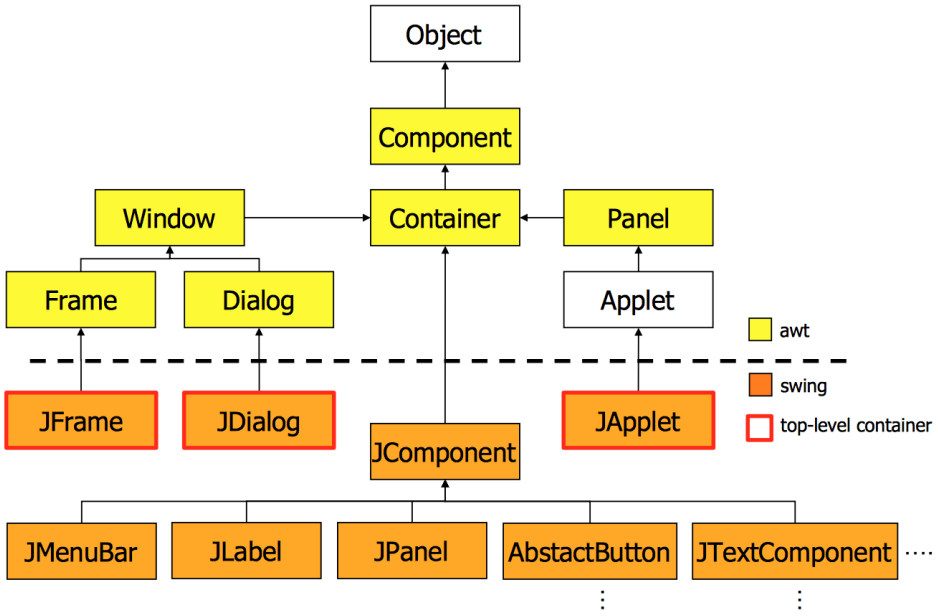

Figura 3.4: Gerarchia di base della libreria Swing

3.2.2 Implementazione

Per l'implementazione dell'interfaccia grafica è stato necessario, in primo luogo, aggiungere un pulsante alla toolbar di ORIS che permettesse di aprire la **token game view**, ossia la vista del token game.

Figura 3.5: Toolbar delle views in ORIS

Questa si aggiunge a due viste già presenti (Fig. 3.5): l'editor view, che permette la modifica della rete di Petri (a sinistra), e l'engine view, che fornisce un'interfaccia per l'analisi della rete (a destra). Inizialmente era stato concepito un logo scheumorfico (Fig. 3.6) per la vista del token game, poi abbandonato per problemi di rendering.

Figura 3.6: Logo originale del token game

Lo sviluppo della GUI è stato condotto facendo riferimento all'ultimo mockup creato in fase di analisi (Fig. 2.3). Data la disposizione degli elementi principali si è scelto di utilizzare un'implementazione di *LayoutManager* che fornisce un posizionamento a griglia: *GridBagLayout*. Il content pane del frame principale è stato così suddiviso in una tabella 2x2, con la rete di Petri ad occupare tutta la riga inferiore. Le due celle superiori sono state riservate (da sinsitra verso destra) all'esplorazione e alla storia del cammino rispettivamente. Al pannello di esplorazione è stato a sua volta applicato un *GridBagLayout*, per separare logicamente la lista delle transizioni abilitate dai pannelli di backtracking e di esplorazione automatica (Fig. 3.7).

Per contenere la tabella delle transizioni è stato utilizzato un *JScrollPane*, il quale permette la comparsa di una barra di scorrimento verticale quando il numero di transizioni abilitate diventa elevato. L'interfaccia è semplice, ma integra perfettamente tutte le funzionalità richieste nella fase di analisi dei requisiti.

Alla luce di quanto visto nel Cap. 2, si vuole raccordare le funzionalità sviluppate nell'API con le interazioni dell'utente con l'interfaccia grafica. Per farlo, si è reso necessario creare due oggetti di tipo *Listener* che stessero in ascolto degli eventi generati dall'utente.

- **ExplorationListener:** sta in ascolto di eventi confinati all'interno

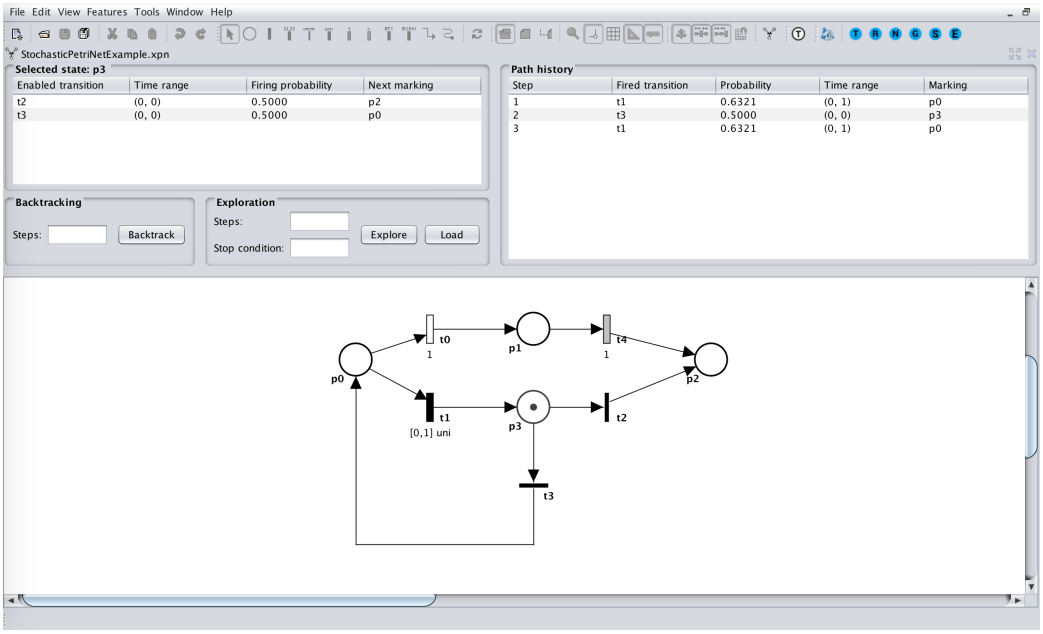

Figura 3.7: Vista del token game

della sezione di esplorazione. In particolare sono cinque gli eventi possibili:

- Click sul bottone *Backtracking*
- Click sul bottone *Exploration*
- Click sul bottone *Load* per caricare la rete di Petri dall'editor view. La token game view infatti non permette di modificare la rete. Qualora si fosse modificata la rete nell'editor sarebbe necessario caricarla per aggiornare le modifiche.
- Doppio click su una riga della tabella *Selected state* per far scattare la transizione associata (esplorazione manuale)
- Click sull'header della colonna *Time Range* per mostrare la DBM relativa

- **HistoryListener:** sta in ascolto di un unico evento, ossia il click di una riga della tabella *Path history* che permette di recuperare lo stato associato ed aggiornare di conseguenza la tabella *Selected state*.

L'implementazione della GUI ha richiesto inoltre di inserire dei messaggi di avviso o di errore per avvisare l'utente di un'operazione non disponibile o che potrebbe causare effetti non desiderati. Nella fattispecie, qualora si recuperi uno stato della storia e si decida di far ripartire l'esplorazione da quel punto, si è ritenuto opportuno mostrare uno warning che avverta del seguente "effetto collaterale": i passi compresi tra quello selezionato e l'ultimo verranno cancellati dalla storia. Allo stesso modo, se si cerca di caricare la rete dalla editor view, viene eseguito un controllo preventivo sull'omogeneità dei tipi di transizione e, in caso negativo, si visualizza un messaggio di errore. E ancora: se viene inserito un valore diverso da un intero positivo nel campo *Steps*, il bottone di esplorazione automatica o di backtracking viene momentaneamente disabilitato.

3.2.3 Dettagli implementativi

In questa sezione si è deciso di riportare in dettaglio i procedimenti più raffinati per ovviare ai problemi che si sono presentati nel corso dell'implementazione del token game.

Variare la token game view in base al tipo di rete di Petri

Le transizioni di una rete di Petri possono essere opzionalmente arricchite con due tipi di features: timed o stochastic. La token game view può essere aperta solo quando la rete considerata è omogenea rispetto ad esse, ossia quando contiene solo transizioni timed, stochastic, o prive di features. In

base al tipo di rete, vorremmo mostrare o meno determinate informazioni e variare il comportamento dell'API sottostante.

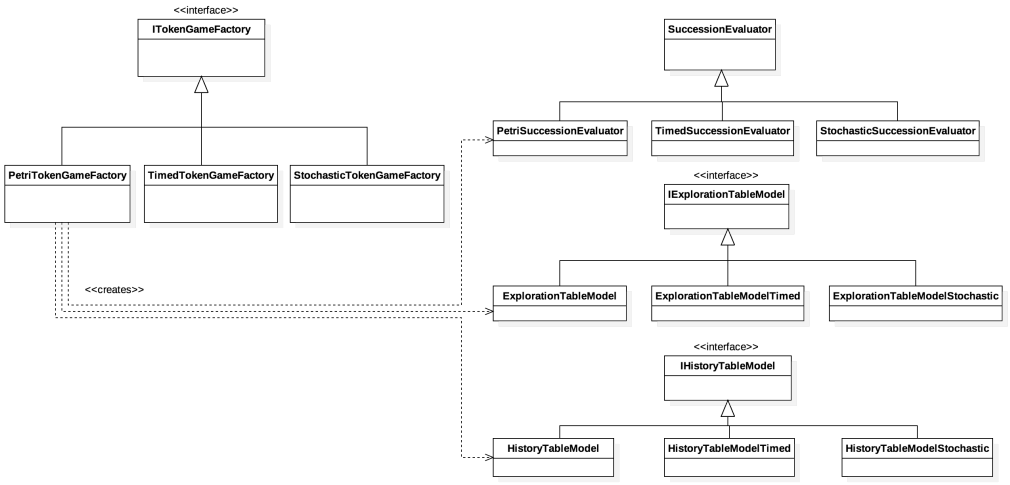

Figura 3.8: Abstract factory pattern applicato al token game

Il pattern Abstract Factory si dimostra perfettamente funzionale al nostro scopo, fornendo un'interfaccia per creare famiglie di oggetti connessi o dipendenti tra loro. Come possiamo osservare dal diagramma UML di Fig. 3.8, si sono create tre differenti "fabbriche" di oggetti, una per ogni tipo di rete di Petri. Per rendere il diagramma più facile alla lettura sono stati omessi tutti i metodi di creazione e le relazioni della *TimedTokenGameFactory* e della *StochasticTokenGameFactory*.

- *SuccessionEvaluator* funge da discriminante nell'API per valutare in che modo calcolare la successione degli stati e soprattutto quale algoritmo di scelta casuale utilizzare per far scattare automaticamente le transizioni.

- *IExplorationTableModel* e *IHistoryTableModel* sono due interfacce le cui implementazioni governano l'organizzazione e la disposizione dei

dati all'interno delle due omonime tabelle.

L'utilizzo del pattern permette al sistema di essere indipendente dall'implementazione degli oggetti concreti. In questo modo, qualora si voglia analizzare, ad esempio, una rete di Petri priva di features, verranno omesse dalla tabella le colonne "time range" e "probability" e ogni transizione abilitata avrà la stessa probabilità di scattare delle altre.

Mantenere separate l'editor view e la token game view

L'editor view permette la creazione di una rete di Petri mediante un sistema di drag and drop dei singoli componenti. La token game view mostra il comportamento della rete creata, permettendo di testarne rapidamente proprietà quali la raggiungibilità. Dato che la seconda opera sulla rete creata dalla prima, affinchè le reti delle due viste possano evolvere separatamente è necessario ricorrere ad una deep copy. In particolare, l'oggetto da copiare è un oggetto di tipo *PNEditorDocument*, la cui classe è contenuta all'interno del progetto OrisPNEditor. Il *PNEditorDocument* contiene una mappa che descrive tutti gli elementi del documento "rete di Petri" che lo compongono:

```
private Map<String, PNEntity> entities = new LinkedHashMap<>();
```

In particolare ogni entità è definita come una classe che estende *PNEntity*, quindi archi, posti, transizioni e via dicendo sono tutte definite da classi che estendono *PNEntity*. Ricordiamo che di base hanno tutte coordinate sotto forma di (x, y) ma non se ne può fare direttamente il render grafico.

Oris implementa il pattern MVC: il Model è rappresentato da classi che estendono *PNEntity*, mentre la View è rappresentata da classi che estendono *PNComponent*. Le classi che estendono *PNComponent* sono direttamente

renderizzabili, in quanto quest'ultimo estende a sua volta la classe JComponent di Swing. Di seguito mostriamo la tassonomia della classe astratta *PNEntity*, utile per comprendere le problematiche che si sono dovute affrontare:

- **PNConnector**
  - Arc
  - InhibitorArc
  - NoteConnector
- **PNNode**
  - Joint
  - Note
  - Place
  - Satellite
  - TextSatellite
  - Transition

PNConnector e *PNNode* sono anch'esse classi astratte e quindi non instanziabili. Avendo a disposizione una collezione di entità astratte e non potendo sapere a priori il tipo reale di ogni entità, si è delegato ad un **Visitor** l'operazione di copia. Il pattern è molto utile nel caso in cui si abbia una struttura di oggetti costituita da molte classi con implementazioni diverse, ed è necessario che un algoritmo esegua su ogni oggetto un'operazione differente a seconda dalla classe concreta dell'oggetto stesso.

Il problema principale, tuttavia, sta nella struttura interna delle entità. Queste formano collettivamente un grafo orientato, in cui anche gli archi originali della rete di Petri ne rappresentano un nodo (Fig. 3.9).

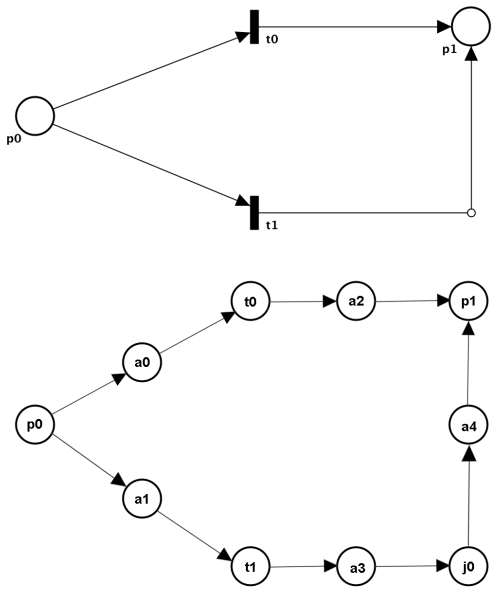

Figura 3.9: Rete di Petri e grafo orientato associato

Ogni entità ha registrato i propri nodi vicini sotto forma di attributi. Occorre però fare una netta distinzione nel caso siano essi nodi o connettori della rete di Petri originaria:

- **PNNode**
  - `Set<PNConnector> inConnectors = new HashSet<>()`
  - `Set<PNConnector> outConnectors = new HashSet<>()`
- **PNConnector**
  - `PNNode fromNode`
  - `PNNode toNode`

Quindi ragionevolmente un nodo può avere più connettori sia d'ingresso che di uscita mentre i connettori possono avere solo un nodo di partenza ed uno di terminazione.

Come attraversare il grafo? Dato che i vicini sono codificati come attributi di ogni entità possiamo ricorsivamente chiamare l'operazione di visita sui connettori di uscita, qualora sia un nodo, e sul nodo di terminazione qualora sia un connettore. In questo modo tramite il pattern Visitor si realizza una **DFS** del grafo (Depth First Search, ossia una visita in profondità). [18] Questa tuttavia presenta una criticità di fondo: quando si arriva all'utima delle chiamate ricorsive e si deve effettuare la copia vera e propria dell'entità, non si possono settare gli attributi "di input" in quanto ancora non sono stati copiati!

Per ovviare a questo problema si è proposto una soluzione che opera in tre passi distinti:

1. Si effettua una DFS del grafo settando per ogni entità solo gli **attributi di output**. Se la visita non può proseguire perchè ci sono nodi vicini di output che sono già stati visitati, ci troviamo in presenza di un ciclo: non settiamo gli attributi di output e salviamo l'entità in una lista temporanea *leftEntities*.

2. Si settano gli **attributi di input** di ogni entità. Ora è possibile, dato che di tutte le entità è stata effettuata correttamente una deep copy.

3. Si settano gli attributi di output delle **entità rimaste** (*leftEntities*), ossia quelle che danno origine ad un ciclo.

Per effettuare la DFS si è dovuto ricorrere ad un sistema di marcatura delle entità basato sui colori e così concepito: bianco, entità ancora non visitata; grigio, entità visitata ma ancora non copiata; nero, entità copiata.

In questo modo si è riusciti ad effettuare una deep copy del *PNEditorDocument* che ha permesso di mantenere le reti delle due view distinte.

Lo sviluppo del progetto seguendo una logica di divisione in moduli si inquadra nel concetto più generale di Separation of concerns (SoC), un principio di design secondo il quale un programma dovrebbe essere diviso in più sezioni, una per ogni "concern". Quando le sezioni sono correttamente separate tra loro, si semplifica il riuso del codice delle stesse e soprattutto la modifica. Una delle maggiori qualità del SoC è quella di poter modificare il codice di una sezione senza essere a conoscenza dei dettagli implementativi delle altre.

Capitolo 4

Conclusioni

L'elaborato svolto si è proposto di documentare tutte le fasi coinvolte nel ciclo di sviluppo del token game nel tool ORIS. Questo si inquadra all'interno dei principi formali delle metodologie di sviluppo dei sistemi informativi teorizzati dall'Ingegneria del Software.

Si è quindi cercato di definire un insieme di processi, ovvero sequenze di fasi che individuano tappe specifiche nella realizzazione di un sistema software tutte documentate e ispezionabili.

Il primo capitolo, dove vengono espressi i concetti base necessari alla comprensione del token game, può essere fatto confluire, insieme agli incontri preliminari tenuti con il relatore, il Prof. E. Vicario, nella fase di studio di fattibilità del progetto. In questa fase, infatti, si è effettuata un'identificazione preliminare del problema, delle alternative, e delle soluzioni di massima possibili.

Il secondo capitolo può essere ovviamente ricondotto alle fasi di analisi e

specifica dei requisiti e di progettazione, nella quale viene definita la struttura della soluzione proposta. Notiamo qui l'importanza della produzione di artefatti intermedi al passaggio tra le fasi (milestones), grazie anche all'utilizzo dei diagrammi UML che uniscono semplicità e capacità comunicativa.

Il terzo capitolo riguardante l'implementazione rappresenta sia la fase di sviluppo e test di unità che quella di integrazione e test di sistema. La scelta di sviluppare separatamente API e GUI ha permesso una netta distinzione tra quella che è la logica di dominio del token game e la sua interfaccia grafica (principio di Separation of Concern). In questo modo è possibile accedere alle funzionalità sviluppate anche solo tramite API.

Non dobbiamo però pensare a queste fasi come a compartimenti stagni. Grazie ai confronti settimanali con il correlatore, l'Ing. M. Biagi, è stato possibile correggere in corso d'opera le strategie di implementazione, o sperimentarne delle nuove non individuate in fase di analisi.

Si è trattato quindi di un lavoro estremamente dinamico e stimolante, che proprio per il modo in cui è stato concepito e progettato, si presta bene a futuri sviluppi più raffinati.

Bibliografia

[1] Università degli Studi di Torino | Dipartimento di Informatica. Greatspn. `http://www.di.unito.it/~greatspn/index.html`.

[2] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, Vol. 77(No. 4):541–580, apr 1989.

[3] A. De Luca. Petri nets. `http://www.diag.uniroma1.it/~deluca/automation/Automazione_RetiPetri.pdf`, sep 2016.

[4] E. Vicario, G. Bucci, M. Paolieri, J. Torrini, L. Carnevali, and J. Giuntini. Oris tool. `http://www.oris-tool.org/`.

[5] The Apache Software Foundation. Maven project. `https://maven.apache.org`.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, first edition, oct 2002.

[7] J. Bloch. *Effective Java*. Addison-Wesley, second edition, 2008.

[8] E. Vicario. Software architecture and methodologies. `http://www.dsi.unifi.it/~vicario/Teaching/SW_Architecture_and_Methodolog/sw_architecture_and_methodolog.html`, mar 2017.

[9] M. Fowler. *UML Distilled*. Pearson, fourth edition, mar 2010.

[10] E. Vicario. Use case diagrams and template. `http://www.dsi.unifi.it/~vicario/Teaching/SW_Engineering/UseCaseDiagsAndTemplates.pdf`, sep 2016.

[11] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[12] Balsamiq. Mockups. `https://balsamiq.com/products/mockups/`.

[13] E. Vicario. *Fondamenti di programmazione*. Società editrice Esculapio, third edition, sep 2011.

[14] K. Beck and E. Gamma. Junit. `http://junit.org/junit4/`.

[15] A. Fantechi. *Informatica Industriale*. CittàStudi, first edition, may 2009.

[16] The Eclipse Foundation. Window builder. `https://eclipse.org/windowbuilder/`.

[17] A. Russo. Progettazione di interfacce grafiche con java swing. `http://www.diag.uniroma1.it/arusso/didattica/ps1314/javaswing.pdf`, feb 2014.

[18] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduzione agli algoritmi e strutture dati*. McGraw-Hill, third edition, aug 2010.