ReDiA-VeriFInt

***Re**alizzatore **Di**agrammi **A**utomatico con**Veri**fica **F**ormale **Int**egrata*

tesina di

Metodi Formali nell'Ingegneria del Software

a.a. 2007-2008

SAPIENZA – Università di Roma

Autori:
Fabio D'Aprano
Claudio Di Ciccio

con la collaborazione di:
Pier Andrea Federici

Supervisore: Prof. Toni Mancini

Versione **0.8** – Settembre 2008

Capitolo 1

Introduzione

1.1 Il progetto ReDiA-VeriFInt

I metodi formali, dagli inizi degli anni '90, stanno entrando a far parte del processo di sviluppo software moderno e vengono impiegati attualmente per realizzare applicazioni di rilevante complessità. Essi si basano, a partire dalle specifiche di progetto, sulla stesura e la verifica di particolari formule, espresse in un linguaggio formale, quali ad esempio logica proposizionale, logica del prim'ordine e logica temporale, utili alla dimostrazione di determinate proprietà da garantire nell'applicazione.

Ad un progetto relativamente complesso corrispondono formule complesse, le proprietà per le quali vogliamo dimostrare la validità diventano estremamente difficili da valutare per un essere umano. Per questa attività sorge quindi il bisogno di *tool* in grado di gestire l'aspetto computazionale della verifica; a tale scopo sono attualmente presenti sulla scena specifici pacchetti software, quali sono *zChaff* per la logica proposizionale, *Prover9*, *Alloy*, per la logica del primo ordine e *NuSMV*, *Spin*, per quanto riguarda la logica temporale.

Purtroppo gli strumenti elencati sopra non sono focalizzati sulla verifica formale del software, bensì sono assimilabili a dimostratori universali di teoremi, oppure a ricercatori di modelli. Al fine di utilizzarli per i nostri obiettivi occorre di conseguenza scrivere manualmente il file di input, a partire dalla particolare istanza del diagramma UML (delle classi, o degli stati e transizioni...), per poi procedere alla stesura delle formule corrispondenti alle proprietà da dimostrare, ed infine alla relativa verifica automatica.

Da qui nasce l'idea del progetto CASE: costruire un ambiente di sviluppo integrato, in grado di assistere il progettista/programmatore in tutte le fasi di progettazione e realizzazione delle proprie applicazioni, validando continuamente le sue scelte progettuali/implementative, fin dalla raccolta ed analisi dei requisiti.

ReDiA-VeriFInt si inserisce in questo contesto, come progetto pilota

dell'intera applicazione. Il presente documento tratterà lo sviluppo della prima parte di questo strumento, allo stadio attuale; più in particolare, descriverà le tecniche algoritmiche, l'architettura software e le tecnologie impiegate per la traduzione automatica di diagrammi UML delle classi, concettuali, in un insieme equipollente di formule della logica di prim'ordine, secondo gli algoritmi descritti in [CM07a], con relativa rappresentazione grafica. La *knowledge base* così rappresentata sarà riproposta, in particolare, sottoforma di direttive di input per il dimostratore di teoremi *Prover9*[1]. Grazie a questo, sarà possibile dimostrare proprietà formali mediante lo stesso programma, espresse sulla base del diagramma dato.

Per ottenere questo doppio passaggio, si è scelto di usare tecnologie *cutting-edge*, basate su XML: al momento della validazione formale, l'intero diagramma viene tradotto in XML, così che, a seconda del *foglio di stile* XSLT[2], possa essere trasformato ed importato da un diverso dimostratore automatico di teoremi.

ReDiA-VeriFInt è stato sviluppato interamente sulla piattaforma *Java* 1.5, la quale, estesa con i package opportuni, peraltro presenti nella versione 1.6, ha fornito l'infrastruttura software sufficiente a garantire l'esportazione dei dati di *back-end* in formato XML, grazie alla tecnologia *JAXB*[3], su cui successivamente applicare XSLT per tradurre il tutto in una serie di stringhe sintatticamente valide per il dimostratore scelto. Tali tecnologie, e la realizzazione del progetto, in ogni particolare, saranno descritte esaurientemente nei prossimi capitoli.

[1] `http://www.cs.unm.edu/~mccune/prover9/`
[2] eXtensible Stylesheet Language Transformations
[3] Java Architecture for XML Binding

Capitolo 2

Architettura del sistema

2.1 Presentazione

Il sistema è stato interamente costruito tenendo in mente che sarebbe stato in seguito ripreso da altri sviluppatori per incrementarne le funzionalità: uno dei requisiti principali era dunque l'*estendibilità*.
Dunque, per favorire il lavoro dei programmatori, è stato deciso di seguire delle soluzioni progettuali *strutturate*, *interoperabili* e *modulari*. Cosa ciò abbia comportato in sede di realizzazione diverrà più evidente tramite i dettagli forniti nelle sezioni a seguire.
Ora, premettiamo che abbiamo innanzi tutto seguito il *pattern architetturale* **MVC** (***M**odel **V**iew **C**ontroller*), così da rendere, fra le altre cose, più espliciti, ai prossimi sviluppatori, i "punti di aggancio" cui fare riferimento (ad es., se si vuole tradurre lo schema concettuale dell'utente in un altro linguaggio di input per dimostratori automatici, occorre intervenire sullo strato "Control"; per fornire di un'interfaccia grafica questa nuova peculiarità, si deve mettere mano al livello "View"...).

Analizziamo a questo punto l'architettura del livello concettuale (*Model*) su cui si basa l'intero sistema, non prima di aver specificato che per ogni livello le classi realizzative sono separate all'interno di *package* omonimi. In seguito approfondiremo anche le architetture di *View* e *Control*.

2.2 Model

2.2.1 Studi preliminari

Prima dell'avvio di questo progetto, Michele Proni, studente di Ingegneria Gestionale de La Sapienza, in [Pro07], aveva operato, all'interno del suo studio condotto come tesi (il cui relatore era il Prof. Mancini), un'analisi approfondita della modellazione dei dati necessari alla rappresentazione di

varie tipologie di diagramma UML. Fra queste, anche quella che faceva al caso nostro, ovvero i Diagrammi delle Classi Concettuali (vedi [Pro07][parr. 2.1.2, 3.1.1] e [MS08a]).
Abbiamo deciso di seguire in modo pedissequo le soluzioni da lui adoperate, salvo operare minime modifiche, in modo tale da adattare il suo modello concettuale alle nostre esigenze, estendendo, e non ripetendo, il suo ottimo lavoro. Questo ha consentito di sfruttare l'esperienza da lui maturata, risparmiare tempo in sede di analisi e progettazione, e non produrre ulteriore documentazione sulle motivazioni delle scelte effettuate, oltre quelle già da lui redatte.
Il codice realizzativo, invece, è stato completamente ricostruito.

2.2.2 Uno sguardo d'insieme

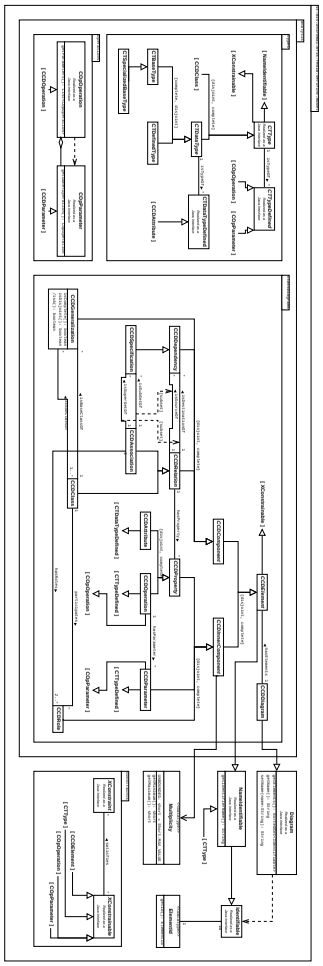

Figura 2.1: Model: uno sguardo d'insieme

Come si può notare dal diagramma della Figura[1] 2.1, l'intero livello *Model* dell'applicazione risiede all'interno del package it.uniroma1.dis.mfis.redia.verifint.model. Il nostro progetto aveva come obiettivo realizzare il motore per la gestione di Diagrammi delle Classi Concettuali (*Conceptual Class Diagrams* – normalmente stilati in fase di analisi) di cui verificare tramite dimostratore automatico determinate proprietà: ecco perché il pacchetto più esteso è anlysis.classDiagram.

[1] I diagrammi delle classi di questa sezione sono concettuali, tuttavia, per migliorare l'accessibilità della documentazione, preferiamo fin da subito esplicitare la divisione in *package*.

2.2.3 Componenti di uso generale

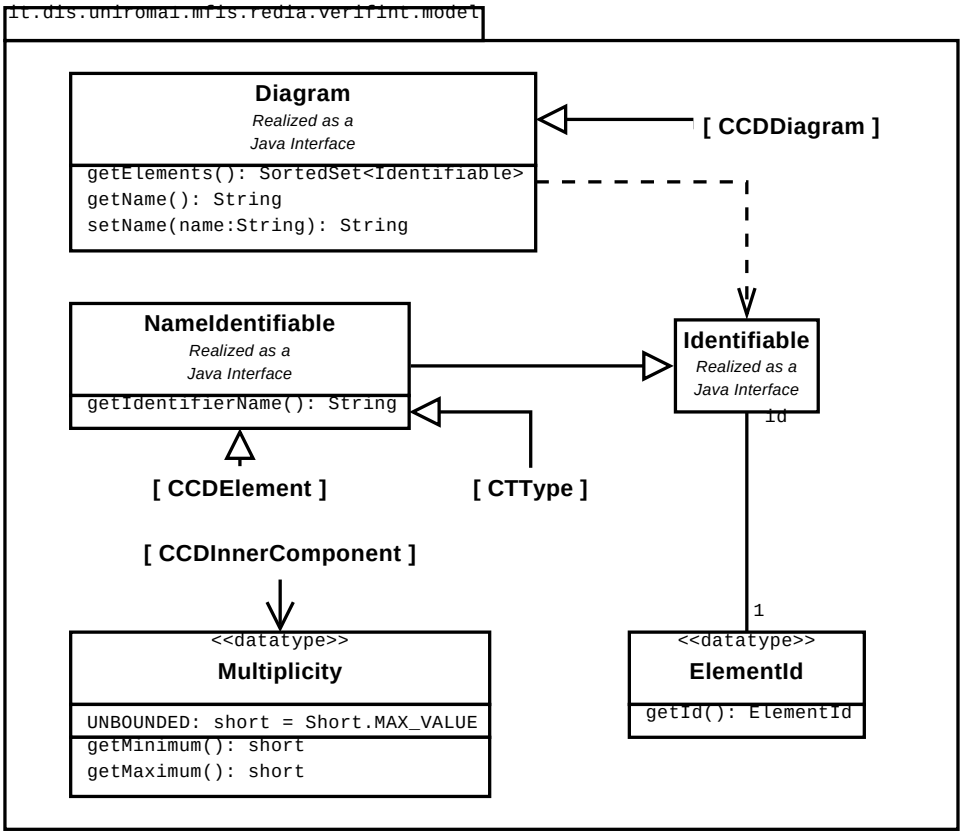

Figura 2.2: Il package it.uniroma1.dis.mfis.redia.verifint.model

Si presuppone che ogni *diagramma* (sia esso delle *Classi*, o degli *Stati e Transizioni*, o dei *Casi d'Uso*) debba essere un'istanza di Diagram. Ogni diagramma è un insieme ordinato di IdentifiableElement, ossia elementi fra loro identificabili (Identifiable) tramite una particolare chiave di ricerca, univoca, di tipo ElementId.
Essendo però noi a conoscenza del fatto che la maggioranza degli elementi sarebbero stati trasposti in predicati della logica (del primo ordine, nel caso dei Diagrammi delle Classi), abbiamo ritenuto opportuno fornire ogni elemento del Diagramma Concettuale di un'ulteriore identificatore più leggibile, basato sul nome effettivo dato all'elemento. Tale nome sarebbe divenuto il predicato con cui identificare, univocamente, in un file di input per i dimostratori automatici, gli elementi stessi (si veda la Sezione 4.2 per un esempio che chiarisca questo aspetto).
All'interno di questo pacchetto si ha pure il tipo di dato Multiplicity, per via della sua generalità d'uso.

2.2.4 Il diagramma delle classi concettuale

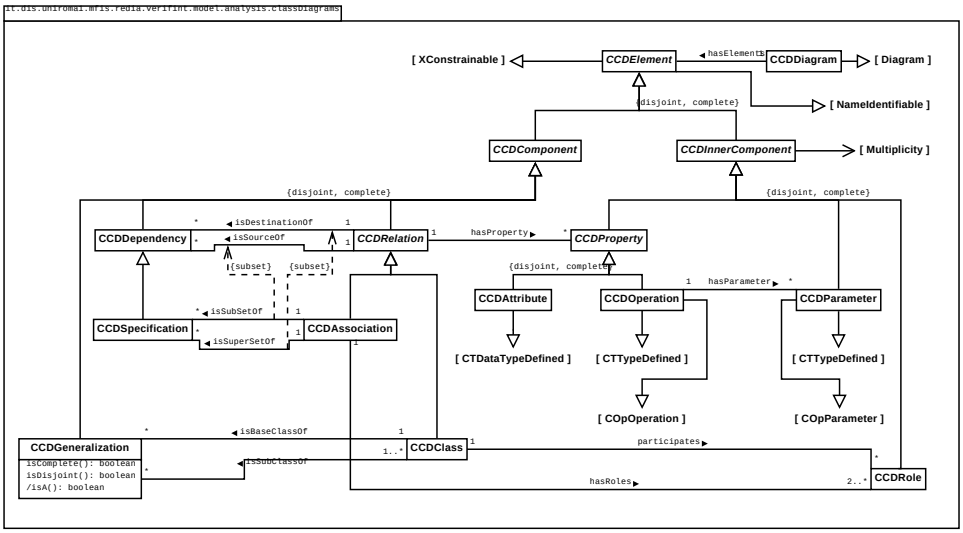

Figura 2.3: Il package it.uniroma1.dis.mfis.redia.verifint.model.analysis.classDiagram

Il package it.uniroma1.dis.mfis.redia.verifint.model.analysis.classDiagram ripropone quasi integralmente le soluzioni progettuali proposte in [Pro07], salvo ridenominazioni orientate ad una maggiore leggibilità (ad esempio, il comune prefisso CCD). Pertanto, non approfondiremo eccessivamente le scelte adottate: quando si rivelano coincidenti, rimandiamo al documento di cui sopra.
Le istanze **CCDDiagram** contengono al loro interno oggetti **CCDElement**, rispettivamente figli di Diagram e NameIdentifiableElement. CCDElement è classe madre di una gerarchia, il cui primo livello vede la distinzione fra **CCDComponent** e **CCDInnerComponent**. Questa riflette la differenza che intercorre fra i componenti auto-sufficienti (come una classe) e quelli che devono necessariamente riferirsi ai primi per la loro identificazione (ad esempio, gli attributi).

Classi figlie di **CCDComponent** sono:

CCDRelation classe che denota l'insieme unione di *classi* (**CCDClass**) ed *associazioni* (**CCDAssociation**);

CCDDependency per rappresentare le *dipendenze* (di cui le *specializzazioni* – **CCDSpecialization** – sono un sottoinsieme);

CCDGeneralization per rappresentare le *generalizzazioni* e le *gerarchie*.

L'altro ramo della gerarchia, la cui madre è **CCDInnerComponent** comprende:

CCDProperty classe che denota le *proprietà*, ossia l'insieme unione di *attributi* (**CCDAttribute**) ed *operazioni* (**CCDOperation**);

CCDParameter per rappresentare i *parametri* di ingresso delle operazioni;

CCDRole per rappresentare i *ruoli* ricoperti dalle classi all'interno delle associazioni.

Si noti che poiché l'associazione **hasProperty** va da CCDRelation a CCDProperty, è previsto che sia classi che associazioni possano avere sia attributi che operazioni. Si noti inoltre che qualunque CCDInnerComponent è associato ad un valore Multiplicity: ciò significa che non solo gli attributi ed i ruoli ma anche le operazioni[2] ed i relativi parametri hanno una molteplicità.

2.2.5 Le operazioni

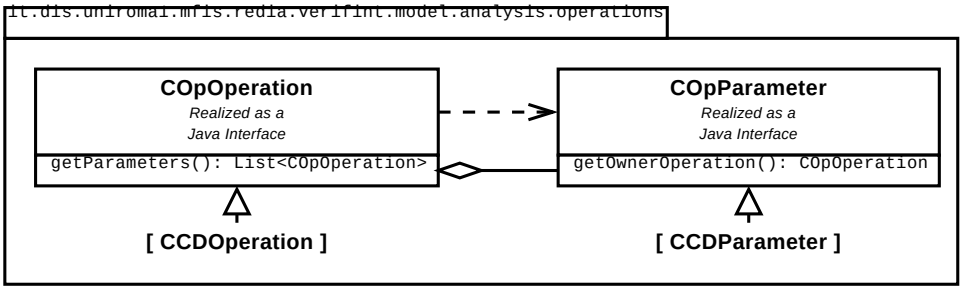

Figura 2.4: Il package it.uniroma1.dis.mfis.redia.verifint.model.analysis.operations

Il prefisso di tutte le classi e le interfacce del package inerente questo aspetto della modellazione, che si riferisce alla semantica delle operazioni concettuali, denominato it.uniroma1.dis.mfis.redia.verifint.model.analysis.operations, è "COp" (*Conceptual Operation*)[3].
COpOperation indica le *operazioni concettuali*, di cui le istanze **COpParameter** costituiscono i *parametri*.

[2]Per molteplicità delle operazioni si intende quella del parametro di uscita (ritorno).

[3]Per motivi di leggibilità, dalla figura 2.4 sono state omesse le estensioni di entrambe le classi COpOperation e COpParameter verso XConstrainable e CTTypeDefined, esplicitate rispettivamente nelle figure 2.6 e 2.5

2.2.6 I tipi

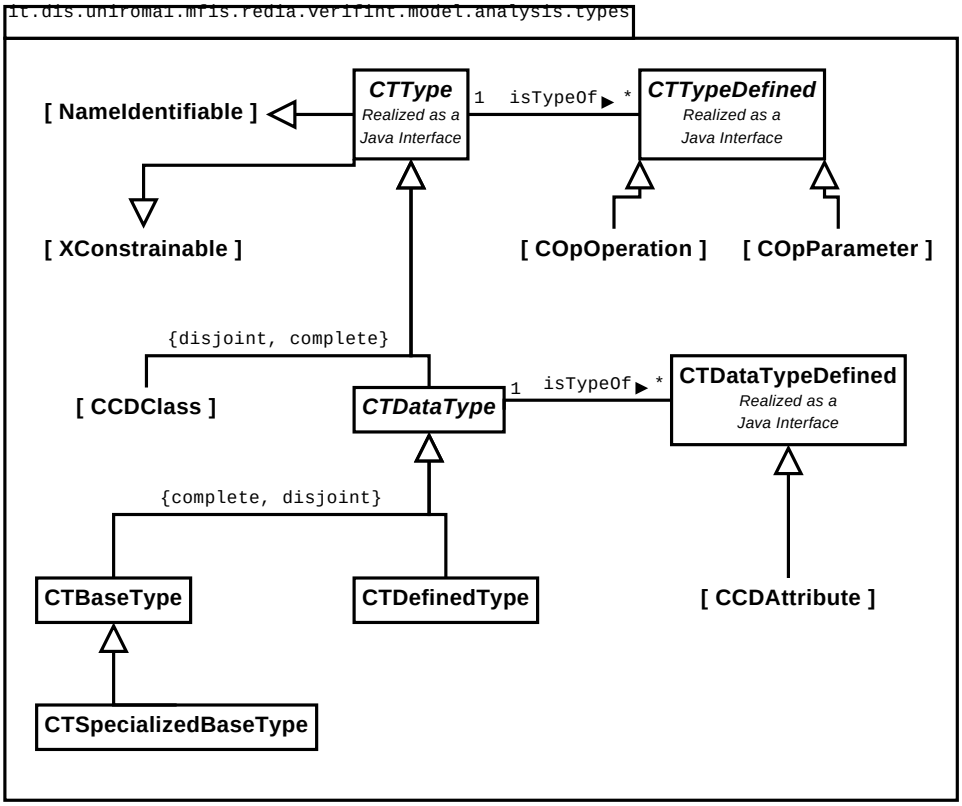

Figura 2.5: Il package it.uniroma1.dis.mfis.redia.verifint.model.analysis.types

Il prefisso di tutte le classi e le interfacce di questo package, denominato it.uniroma1.dis.mfis.redia.verifint.model.analysis.types, è "CT" (*Conceptual Type*). Madre della gerarchia nel pacchetto è **CTType**, ossia il generico *tipo*, unione di *classi* (**CCDClass**) e *tipi di dato* (**CTDataType**).
Si noti come la semantica di classe del diagramma concettuale come un *tipo* (CTType) venga disaccoppiata dalla semantica di elemento del diagramma delle classi (CCDComponent), sfruttando il meccanismo di ereditarietà multipla.

I tipi di dato (CTDataType) sono suddivisi a loro volta in tipi di dato *base* (**CTBaseType** – come *Stringa*, *Intero*, *Data*...) e tipi di dato *definiti dall'utente* (**CTDefinedType**)[4]. Si è infine previsto che i tipi di dato base possano essere specializzati (**CTSpecializedBaseType** – ad esempio: *URI*,

[4]non essendo tema di analisi per questo progetto, non si è approfondito l'ambito dei tipi di dato, dunque se prevedessero o meno *side-effect*, con quali attributi e quali operazioni; tuttavia, si è ritenuto comunque opportuno fornire dei punti di "aggancio" a futuri sviluppi di questa applicazione fin da subito

InteroPositivo...).
Si noti che, per correttezza, alcuni elementi dei diagrammi possono essere di un qualunque tipo (anche classi del diagramma concettuale), come le operazioni[5] ed i parametri, mentre altri solo di un tipo di dato, come gli attributi. Per demarcare questa differenza, sono state introdotte le classi **CTTypeDefined** e **CTDataTypeDefined**.

2.2.7 I vincoli

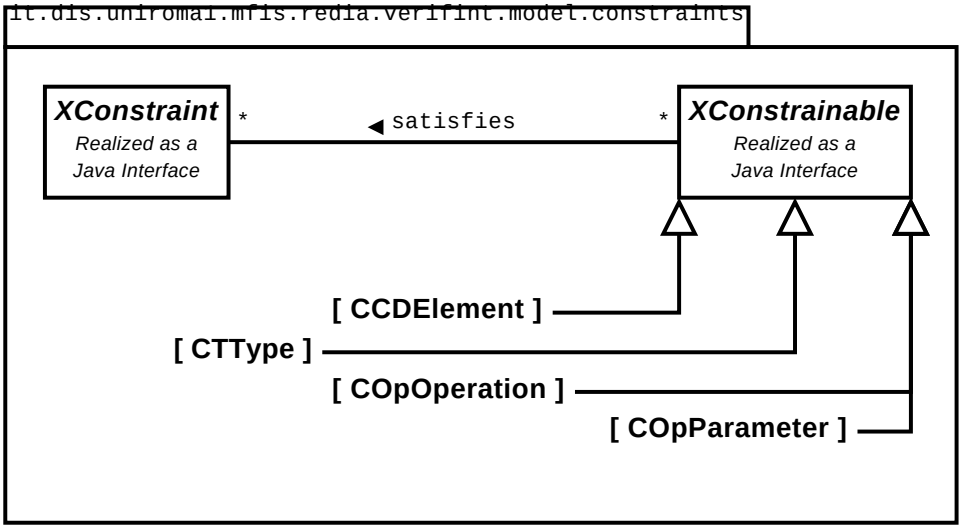

Figura 2.6: Il package it.uniroma1.dis.mfis.redia.verifint.model.analysis.constraints

Il prefisso di tutte le classi e le interfacce del package in analisi, denominato it.uniroma1.dis.mfis.redia.verifint.model.analysis.constraints, è "X"[6].
Anche questo pacchetto, come già it.uniroma1.dis.mfis.redia.verifint.analysis.types, è lasciato all'implementazione di chi, nell'estensione dell'applicazione, provvederà ad esplicitare i vincoli (**XConstraint**) espressi sugli elementi dei diagrammi (che estendono **XConstrainable**). Per ora, si è provveduto a rendere noto che un vincolo può essere riferito ad un qualunque numero di tipi, elementi del diagramma delle classi, operazioni, o parametri!

[5] Si intende come tipo dell'operazione il tipo del parametro di ritorno

[6] la "X" è usata per ricordare che i vincoli possono essere riferiti a più elementi contemporaneamente, ma soprattutto perché usare la prima, o le prime due lettere di "constraint", a mo' di acronimo, avrebbe potuto confondere con gli altri prefissi già usati – cosa che avrebbe annullato il compito di rendere più leggibile il codice!

2.3 Architettura: View

2.3.1 Progettazione

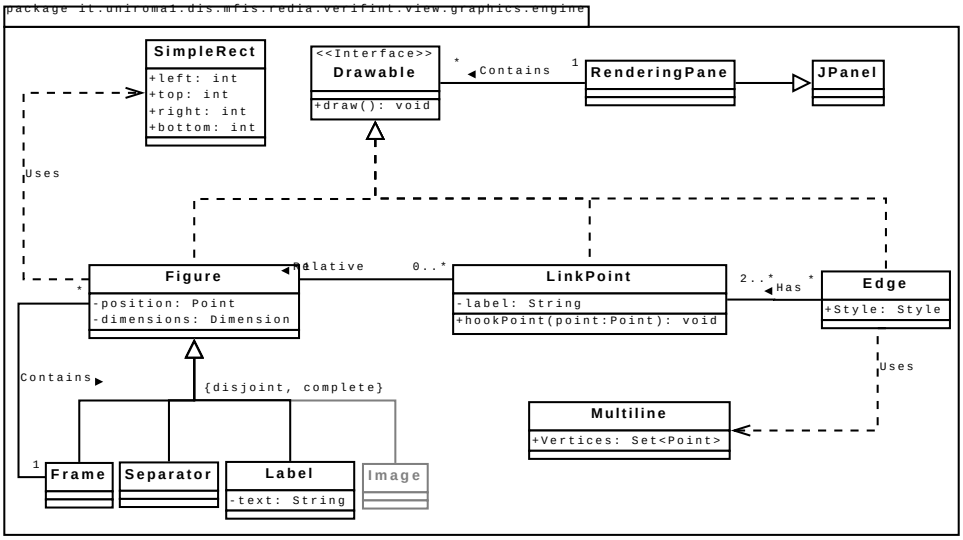

Figura 2.7: View: Class Diagram

La figura 2.7 mostra il diagramma delle classi del package `it.uniroma1.dis.mfis.redia.verifint.view.graphics.engine`. Questo package si occupa di mettere a disposizione le interfacce che forniscono le funzionalità occorrenti per rappresentare un qualsiasi diagramma delle classi memorizzato nel livello *model*. Sviluppi futuri implementeranno anche altri diagrammi UML quali diagramma degli stati e transizioni e use case. Questa libreria grafica contiene strutture dati che operano a basso livello, cioè non hanno alcuna dipendenza con la semantica definita dalla logica applicativa degli altri livelli. Nelle prossime sezioni verrà descritto il funzionamento e le relazioni tra le classi che compongono il package.

Interfaccia Drawable

Questa interfaccia definisce il metodo `draw()` che accomuna tutti gli oggetti che possiedono una particolare visualizzazione grafica. Tale metodo viene ridefinito in modo tale che ogni oggetto che lo implementa, può autogestire il proprio processo di disegno. Questa scelta è dettata dal bisogno di avere un insieme di oggetti omogeneo, memorizzato in una collezione come Liste o Set, da renderizzare su un qualsiasi JPanel di *Swing*.

L'interfaccia **Drawable** costituisce un punto di forza per l'estendibilità della libreria in quanto consente di definire facilmente nuovi oggetti grafici generici.

Classe **Figure**

La classe **Figure** implementa l'interfaccia **Drawable** e definisce un oggetto grafico inscrivibile in un rettangolo (con relativa posizione e dimensioni bidimensionali) comunemente chiamato anche *Bounding Box*. Questa classe fornisce anche un metodo di utilità definito come `hookPoint(p: Point) : Point` che, dato un punto qualsiasi, restituisce il punto "agganciato" alla *Bounding Box* della figura, in altre parole il punto localizzato sul bordo della figura più vicino al punto dato.

Classi figlie di **Figure** sono:

Label si tratta di un oggetto grafico che contiene al suo interno una stringa, indicato in questo progetto come *Label*, ovvero *Etichetta*. La sua proprietà fondamentale è `text` e stabilisce il suo contenuto testuale. È naturalmente possibile impostare lo stile grafico dell'oggetto come il colore, il tipo di carattere, e l'allineamento del testo nel caso l'etichetta venga inserita all'interno di un contenitore in grado di gestirlo, come ad esempio un oggetto di tipo **Frame**.

Frame definisce una generica cornice che contiene una composizione di oggetti di tipo **Figure** memorizzati al suo interno attraverso una lista concatenata lineare. Attualmente supporta composizioni di oggetti di tipo **Label** e **Separator**. Questi oggetti vengono renderizzati in maniera puramente sequenziale. Dalla figura 2.8 è possibile osservare un'istanza di **Frame** composta rispettivamente da un'etichetta che ha allineamento centrato e testo grassetto, un separatore, 3 etichette con formattazione di default, un separatore e infine due etichette con formattazione di default. Naturalmente è possibile impostare il colore di sfondo di ogni oggetto di tipo **Frame**.

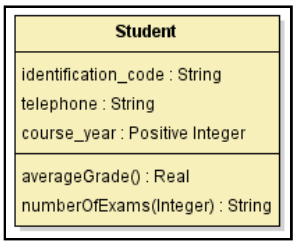

Figura 2.8: Esempio di renderizzazione di un **Frame** composto da etichette e separatori

Separator si tratta di una classe vuota figlia della classe **Figure**. Serve per indicare all'oggetto di tipo **Frame** di disegnare una linea separatrice in corrispondenza di un oggetto di tipo **Separator** all'interno della lista contenuta nel frame **Frame**.

Image nella figura 2.7 questa classe è stata rappresentata in grigio in quanto non effettivamente implementata nella release attuale del codice. È stata tuttavia inserita per mostrare la possibilità di aggiungere e implementare in modo relativamente semplice altri oggetti grafici, come immagini (che di fatto sono di tipo ***Figure*** in quanto sono definite da una *Bounding Box*).

Classi utilizzate da **Figure** sono:

SimpleRect questa classe non fa altro che definire una semplice figura rettangolare definita da quattro proprietà: *left*, *top*, *right*, *bottom*. **SimpleRect** è una classe di utilità pensata per semplificare la scrittura di algoritmi grafici dal momento che Java non fornisce alcuna struttura dati che definisce un rettangolo in base alle coordinate del punto a nord ovest e il punto a sud est.

Classe Edge

La classe **Edge** definisce e calcola il comportamento degli archi di connessione tra una collezione di oggetti di tipo **Figure** e per questo costituisce una componente algoritmica rilevante all'interno del package. Per semplicità si è assunto che un arco (**Edge**) ha una serie di oggetti di tipo **Figure** di partenza e un solo oggetto di tipo **Figure** di arrivo, in questo modo si possono graficare non solo gerarchie, che hanno sempre una o più classi figlie e una sola classe padre, ma anche associazioni di arità *n*. Ogni oggetto di tipo **Edge** possiede uno stile di visualizzazione che stabilisce il tratteggio e le caratteristiche della freccia. Gli stili predefiniti sono selezionabili tra quelli standard degli archi UML: ISA, IMPLEMENTS, SIMPLEASSOCIATION, HAS, USES, AGGREGATION, COMPOSITION[7]. Infine anche **Edge** implementa l'interfaccia **Drawable** perché si tratta di un oggetto grafico visualizzabile a schermo, ma allo stesso tempo non è una **Figure** perché non è caratterizzato da una *Bounding Box*.

Classi utilizzate da **Edge** sono:

LinkPoint questa classe ha una funzione fondamentale per determinare il corretto comportamento di un arco di connessione. Un oggetto di tipo **Edge** possiede in particolare una collezione di **LinkPoint** di partenza e un solo **LinkPoint** di arrivo, il loro insieme ha il riferimento alle **Figure** coinvolte nell'associazione o gerarchia. La classe rappresenta il punto in cui un arco deve "agganciarsi" ad una figura e deve trovarsi necessariamente sul suo bordo. Dal momento che non è possibile assegnare un punto qualsiasi ad un oggetto di tipo **LinkPoint**, esso non fornisce

[7]Questi stili regolano automaticamente il tratteggio dell'arco (continuo, tratteggiato) e la forma e il colore della freccia (cuspide, triangolo, rombo)

un metodo di settaggio diretto, ma mette a disposizione invece il metodo `hookPoint()` descritto relativamente alla classe **Figure**. Infine anch'essa implementa l'interfaccia **Drawable** in quanto, contenendo le etichette che rappresentano il ruolo e la molteplicità riferita alla **Figure** associata al **LinkPoint**, una routine di disegno è necessaria.

Multiline per consentire in futuro il disegno di archi ortogonali o archi spezzati in generale, è stata implementata anche la classe **Multiline**. Si tratta di una struttura dati che definisce una linea spezzata costituita da una collezione di vertici.

Classe RenderingPane

La classe **RenderingPane** eredita direttamente dall'oggetto delle librerie *Swing* **JPanel** e mette a disposizione il contesto grafico sul quale il diagramma verrà rappresentato. Essa contiene un ***TreeSet***[8] di oggetti che implementano **Drawable**. Un semplice ciclo for provvede a disegnare su schermo ogni oggetto contenuto nella lista tramite il metodo `draw()`. Lo stub principale degli eventi di disegno sarà aggiunto dal livello *Control* su questo pannello come un **ActionListener**. È possibile osservare un esempio dell'applicazione di test del motore grafico in figura 2.9.

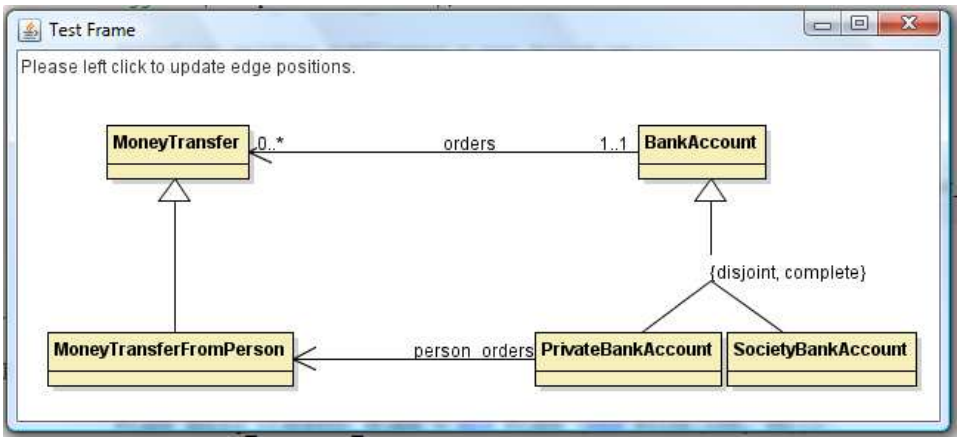

Figura 2.9: Esempio di diagramma delle classi disegnato dal motore grafico di ReDiA-VeriFInt

[8]Il **TreeSet** è una collezione di dati univoci e ordinati, è stato scelto per evitare oggetti duplicati e per dare un preciso ordine di disegno agli oggetti (*ZOrder*)

2.3.2 Widget di I/O

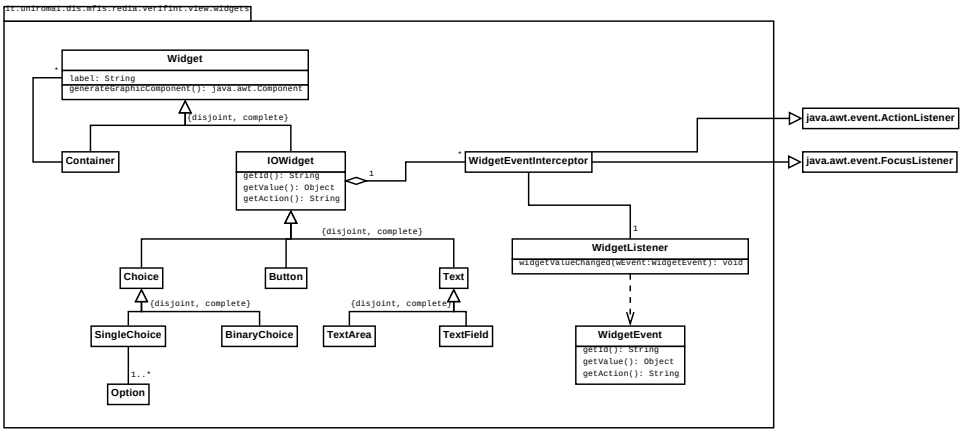

Figura 2.10: Il package it.uniroma1.dis.mfis.redia.verifint.view.widgets

Questa sezione è dedicata a delle classi di utilità per poter velocemente presentare e rendere modificabili a livello grafico le proprietà degli elementi dei diagrammi. Non si tratta di una caratteristica sperimentata ed integrata completamente con il resto del sistema, ma la sua realizzazione ha avuto inizio agli esordi del progetto, e potrebbe costituire un buon punto di partenza per gli sviluppatori che si occuperanno dell'aggiornamento interattivo dei dati del sistema. In virtù di ciò, però, non viene trattato il tutto approfonditamente. Si rimanda, piuttosto, alla documentazione JavaDoc in allegato al codice sorgente stesso, per maggiori informazioni.

A grandi linee, notiamo che tutto parte dalla classe madre **Widget**, da cui tutte le figlie ereditano, estendendola, la capacità di restituire un componente grafico di interazione corrispondente ai dati interni (`generateGraphicComponent()` restituisce infatti un oggetto java.awt.Component, utilizzabile all'interno di qualunque pannello grafico in Java). La gerarchia al di sotto opera una suddivisione tra **Container** ed **IOWidget**, laddove questi ultimi sono effettivamente elementi di interfaccia grafica verso l'utente, mentre i primi sono deputati a contenere altri Widget (quindi ulteriori contenitori) o degli IOWidget. Gli IOWidget possono essere di tipo testuale (**Text**), a scelta tra valori preimpostati (**Choice**), o pulsanti (**Button**). Quel che li rende utili, a livello applicativo, è il fatto che prevedono già la presenza di *Listener* (**WidgetListener**, attivati da oggetti intercettori di eventi primitivi propri delle Swing, **WidgetEventInterceptor**): il meccanismo di interazione basata sugli eventi è tipico di Java, e tramite esso diviene possibile il collegamento a funzioni di livello *Control* – se ne discuterà ulteriormente nella Sezione 2.4.2. Per capire quale si voglia sia l'azione, con quale valore, da compiere sull'oggetto correlato, gli eventi

WidgetEvent forniscono i metodi `getAction()`, `getValue()`, `getId()` (dove l'*id* è la rappresentazione in forma di stringa dell'identifidcatore di cui si parla nella Sezione 2.2.3), rispettivamente.

2.4 Architettura: Control

2.4.1 Uno sguardo d'insieme

Nel livello *Control* si possono trovare due sezioni fondamentali (v. Figura 2.11). Una sezione, ad oggi incompleta ma base per le future estensioni del progetto, relativa all'interazione con i dati, *da* e *verso* l'utente; l'altra, completata per quel che riguarda gli scopi del progetto svolto, relativa alla validazione dei diagrammi. È dunque in questo strato applicativo che si trova il cuore del meccanismo di dimostrazione automatica di proprietà formali, punto focale del *framework* adottato.

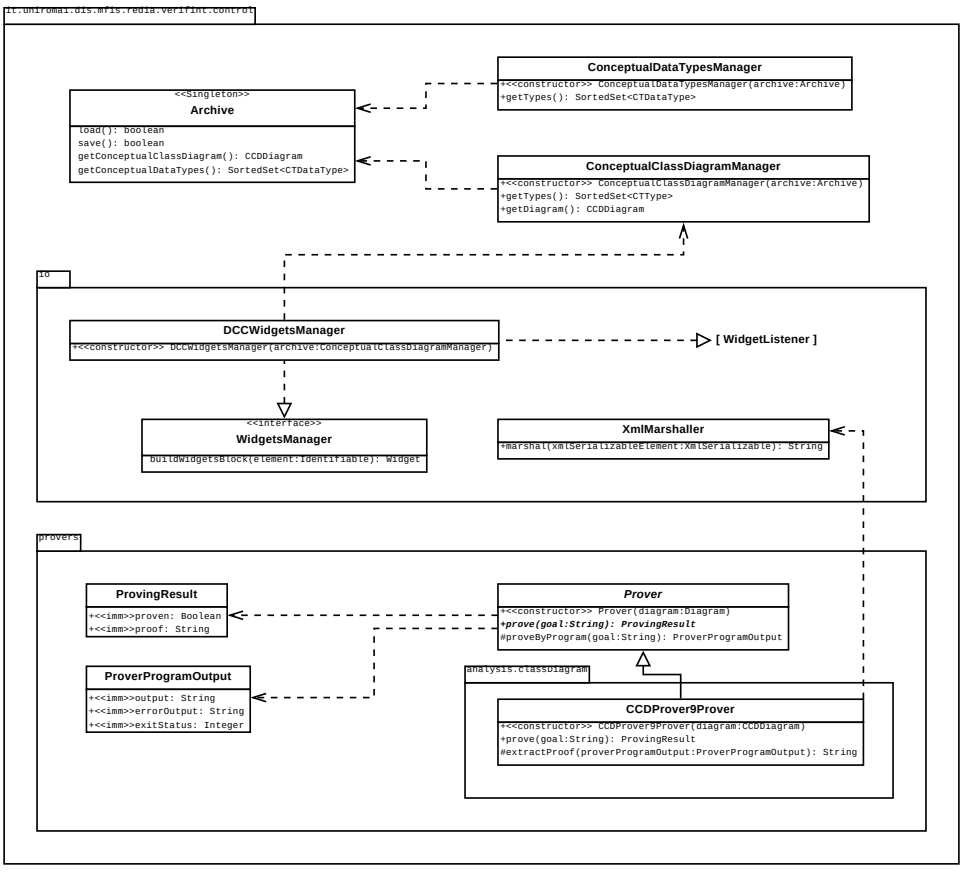

Figura 2.11: Il package it.uniroma1.dis.mfis.redia.verifint.control

2.4.2 Interazione con i dati

Questa sezione del programma non è completa, ma in larghe parti il codice è stato scritto e verificato, pertanto riteniamo opportuno, per successive estensioni, documentarlo fin d'ora.

Il ruolo chiave è svolto da **Archive**, classe implementante il pattern **Singleton** essendo fondamentalmente il gestore unico del *repository* dei dati (svolge un ruolo simile all'*EntityManager* della *Java Persistence API*); preposto al salvataggio e caricamento in formato binario delle informazioni su disco rigido (parliamo rispettivamente dei metodi `load()` e `save()`). Da esso le classi di gestione dei tipi (**ConceptualTypeManager**) e del diagramma concettuale (**ConceptualClassDiagramManager**) ricavano i propri dati. Le istanze di classi implementanti l'interfaccia **WidgetsManager** (in particolare quella di nostra competenza, **DCCWidgetsManager**), si occuperanno di creare, a partire da elementi identificabili (da elementi del diagramma concettuale delle classi) i Widget descritti nella Sezione 2.3.2, e di gestirli (infatti DCCWidgetsManager implementa l'interfaccia WidgetListener). In questa maniera, si realizza il disaccoppiamento tra gli oggetti di interazione prettamente grafici, i quali assolvono solo un compito di input/output tramite interfaccia utente, e quelli di interazione con i dati, in modo che i secondi usino i primi per mostrare e ricavare i dati, senza che i primi debbano attivare esplicitamente gli altri, in alcun caso.

2.4.3 Serializzazione XML e verifica di proprietà formali

Per disaccoppiare quanto più possibile fra loro i livelli, abbiamo ritenuto di operare come in Figura 2.12, per poter collegare un diagramma alla sua traduzione in formato XML. Ispirandoci alla ben nota interfaccia ja-

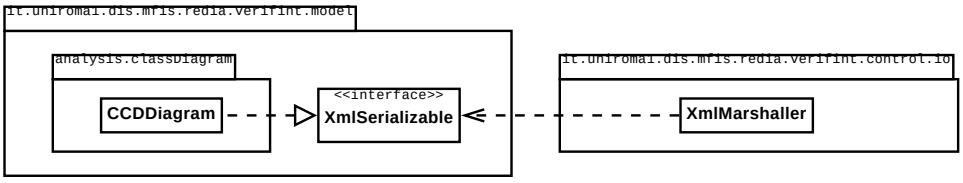

Figura 2.12: L'interfaccia it.uniroma1.dis.mfis.redia.verifint.model.XmlSerializable

va.lang.Serializable, puramente dichiarativa, anche in questo caso abbiamo deciso di utilizzare un'interfaccia, **XmlSerializable**, il cui unico compito fosse indicare che la classe implementante fornisse delle regole di serializzazione. Operando in tal modo, la classe preposta alla traduzione (*marshalling*) XML, **XmlMarshaller**, avrebbe agito solo con oggetti di tipo XmlSerializable. Conseguenza: nel momento in cui altri sviluppatori decideranno di estendere con la gestione di altri diagrammi questo sistema, sarà loro sufficiente

dichiarare il nuovo diagramma implementante XmlSerializable, ed esso potrà essere tradotto, senza modificare di una sola riga il codice di XmlMarshaller.

Prover è la classe madre (astratta) di una gerarchia *in fieri* di classi preposte alla gestione della verifica automatica di proprietà di diagrammi (cioè classi figlie di Diagram), tramite l'interazione con dimostratori automatici. **CCDProver9Prover** è la classe figlia di Prover che abbiamo implementato: si basa su diagrammi delle classi concettuali, e lavora con *Prover9* (vedi [McC08]), successore di *Otter* (vedi [McC03]).

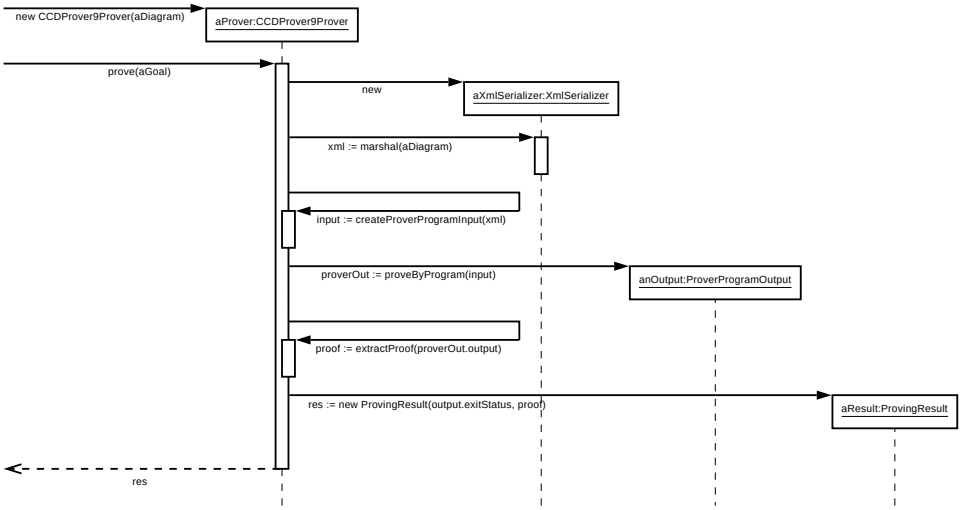

Figura 2.13: L'esecuzione del metodo di verifica di una proprietà di un diagramma

Il funzionamento dell'interazione fra queste classi (fulcro dello strato *Control* dell'applicazione attuale) è descritto sinteticamente nel Diagramma delle Sequenze di Figura 2.13.
Si può vedere come, ad una chiamata del metodo `prove (String)`, corrisponda dapprima una chiamata su un'istanza XmlMarshaller del metodo `marshal (XmlSerializable)`, la cui stringa restituita (in formato XML) viene tradotta in un input per il dimostratore automatico di teoremi, tramite la successiva invocazione del metodo `createProverProgramInput (String)`. Poi, con l'input prodotto, viene richiamato il dimostratore stesso, tramite l'attivazione del metodo `proveByProgram (String)` in un processo separato, i cui output, standard e di errore, e il codice di uscita, vengono inseriti all'interno dell'oggetto ProverProgramOutput; lo standard output bufferizzato viene a quel punto filtrato, per ottenere da esso la sola dimostrazione (`extractProof (String)`), da restituire all'interno di un oggetto ProvingResult.

Capitolo 3

Realizzazione

3.1 Model

3.1.1 Strutturazione dei package

Il passaggio alla realizzazione di quanto indicato nella precedente Sezione 2.2 è stato effettuato in modo estremamente meccanico, seguendo le regole dettate dalle slide [MS08b]. In più, abbiamo deciso di inserire dei *meta-*package, ossia utili a separare le classi realizzative da quelle di utilità (come le classi di associazione, o le eccezioni applicative), dalle realizzazioni dei tipi di dato e dalle interfacce: essi sono facilmente identificabili, rispetto agli altri, dal fatto che il loro prefisso è "_" (_links per le associazioni, _exceptions per le eccezioni, _dataTypes per i tipi di dato, _interfaces per le interfacce). Questo principio è stato seguito in tutti i package, non solo in quelli del *Model*.

Tutte le associazioni fra le classi del Model sono a responsabilità doppia, onde facilitare il compito di individuare le modifiche grafiche da apportare per un editor grafico, nel caso di cambiamenti nel diagramma (per esempio: sottolineare in rosso, o sostituire con un punto interrogativo, le voci type di tutti i parametri delle operazioni il cui tipo concettuale sia una classe appena cancellata).

3.1.2 Il pattern *Factory*

Teniamo a specificare che abbiamo realizzato tutte le associazioni con responsabilità doppia. Per ovviare al problema di gestire associazioni con responsabilità multipla e vincolo di partecipazione abbiamo adottato il *design pattern Factory*; ad esempio, i costruttori di CCDParameter (le cui istanze devono essere associate ad un'istanza di CCDOperation e ad una di CCDDiagram) sono dichiarati ad accesso ristretto di package, laddove sono *public* i metodi con equivalenti parametri (salvo l'aggiunta di ulteriori argomenti, per avere le istanze cui associare l'istanza creata e rispettare le molteplicità

minime) all'interno di CCDParameter_Factory: saranno questi ultimi a poter essere chiamati per ottenere nuove istanze CCDParameter (il fatto che solo le classi *Factory* si trovino nello stesso package delle classi realizzative garantisce che i costruttori di queste ultime non siano visibili dall'esterno, ma solo dalle prime). Per maggiori dettagli sul funzionamento, rimandiamo alla documentazione JavaDoc, allegata al codice sorgente, e al Capitolo 4.

3.1.3 Regole di serializzazione XML

Listato 1 Annotazioni JAXB nel codice delle classi del *Model*.

```
public abstract class CCDProperty extends CCDInnerComponent {
  ...

  @XmlJavaTypeAdapter(
    type=CTType.class,
    value=CTTypeXmlImplementation.Adapter.class)
  @XmlAttribute(required=true)
  @XmlIDREF
  protected CTType type;

  @XmlTransient
  protected CCDRelation ownerRelation;

  ...
}
```

Spesso si possono notare, all'interno del codice, annotazioni come nel Listato 1, il cui prefisso è "`@Xml`". Si tratta di annotazioni *JAXB*, che vengono usate come regole di serializzazione XML dei singoli campi (si rimanda all'Appendice A per una spiegazione più dettagliata inerente il loro funzionamento).

3.2 Control

3.2.1 File di setup

Essendo a conoscenza del fatto che questa applicazione dovrà essere eseguita su più piattaforme, per aumentarne la *portabilità*, abbiamo deciso di includere nella distribuzione del programma un file di setup (`setup.ini`) da cui l'applicazione potesse leggere le direttive parametriche fondamentali, come il percorso interno al FileSystem locale in cui è stato installato

il verificatore automatico, il comando da *shell* con il quale avviare quest'ultimo, ecc. Si intuisce che, per eventuali estensioni, come la verifica tramite altri dimostratori automatici, occorrerà inserire nuove voci al suo interno. Il gestore di questo file, istanziato al lancio dell'applicazione, è denominato ExecutionProperties, e sfrutta, all'uopo, la classe del *core Java* java.util.Properties.

3.2.2 Serializzazione XML

La classe di questo livello preposta a trasporre i dati del *Model* all'interno di un documento XML è XmlMarshaller (v. Sezione 2.4.3). Grazie al fatto che già nelle classi del *Model*, tramite annotazioni, è stato specificato il modo in cui ognuna andasse tradotta per il *marshalling* XML, il codice all'interno del metodo `marshal (XmlSerializable)` è di dimensioni estremamente ridotte, come si può notare all'interno del Listato 2.

JAXBContext e Marshaller sono classi del package javax.xml.bind. Il significato delle righe del codice è abbastanza autointuitivo, pertanto non ci dilunghiamo sull'argomento; per maggiori informazioni rimandiamo all'Appendice A.

3.2.3 Dalla serializzazione XML all'input per Prover9

Per la trasposizione dalla serializzazione XML al testo di input per il verificatore automatico, abbiamo utilizzato dei fogli XSLT (v. Appendice B per ulteriori informazioni). Ciò su cui vogliamo focalizzare l'attenzione, anche in questo caso, è il risparmio di codice interno alle classi Java: le regole di traduzione non sono *hard-wired* nel codice delle classi, bensì interamente delegate ai fogli di stile. Si intuisce pertanto che, per eventuali estensioni, come la verifica tramite altri dimostratori automatici, sarà sufficiente scrivere nuovi documenti XSLT, ed aggiungere una nuova classe figlia di Prover – si badi che, pur essendo essa astratta, fornisce metodi già codificati, di uso comune, come quello del Listato 3, che si occupa proprio del *task* che titola questa sezione, ma anche `proveByProgram (String)`, il quale, dato l'input, invoca il dimostratore predefinito dalla classe figlia, e ne restituisce il risultato (per ulteriori informazioni, v. Sezione 2.4.3).

Listato 2 Metodo per il *marshalling* XML di un oggetto XmlSerializable.

```
public class XmlMarshaller {
  public String marshal(
      XmlSerializable xmlSerializableElement) {

    Object xmlSerializableObject =
      (Object)xmlSerializableElement;

    StringWriter marshalledInstanceWriter =
      new StringWriter();

    try {
      JAXBContext jaxbContext =
        JAXBContext.newInstance(
          xmlSerializableObject.getClass());

      Marshaller marshaller = jaxbContext.createMarshaller();
      ...
      marshaller.marshal(
          xmlSerializableElement, marshalledInstanceWriter);
    }
    catch (Exception e) {
      e.printStackTrace(System.out);
    }
    return marshalledInstanceWriter.toString();
  }
}
```

Listato 3 Metodo di traduzione della serializzazione XML del diagramma in un testo di input per *Prover9*

```
public abstract class Prover {
  protected String createProverProgramInput(String diagramXml)
  {
    String proverInput = "";
    try {
      // read the XML input
      Source xmlSource =
        new StreamSource(new StringReader(diagramXml));

      // read the XSLT sheet
      Source xsltSource = new StreamSource(
          new FileReader(
              System.getProperty("user.dir") +
              this.getXsltPath()));

      // transform the XML following the XSLT templates
      TransformerFactory transFactory =
        TransformerFactory.newInstance();
      Transformer trans =
        transFactory.newTransformer(xsltSource);
      StringWriter newXmlWriter =
        new StringWriter();
      trans.transform(
        xmlSource, new StreamResult(newXmlWriter));

      // delete indentations
      String[] lines =
        newXmlWriter.toString().split("\\n");
      StringBuffer stringBuffer = new StringBuffer();
      for(String line : lines) {
        line = line.trim();
        if (!line.equals("")) {
          stringBuffer.append(line + "\n");
        }
      }
      proverInput = stringBuffer.toString();
    }
    catch (Exception e) { ... }
    return proverInput;
  }
  ...
}
```

Capitolo 4

Esempi

In questa sezione vedremo alcuni esempi di funzionamento del sistema, con il codice delle classi di test, e l'output fornito. Tramite questi, sarà certamente possibile prendere maggiore confidenza con il codice e le strutture dati.

4.1 Traduzione in formato XML

Supponiamo di voler tradurre, secondo il formato XML dell'applicazione, il diagramma in Figura 4.1.

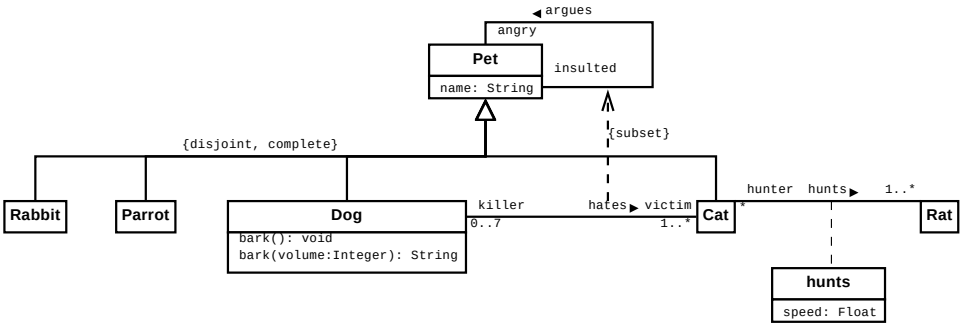

Figura 4.1: Il diagramma di esempio

È evidente che questo diagramma contenga vari costrutti. Vediamo come introdurlo nella nostra applicazione, e tradurlo in formato XML, nella classe di test il cui codice è riportato nel listato di codice a seguire.

```
public class DiagramTest {

  // Animals diagram
  public static CCDDiagram prepareAnimalsTestDiagram() {
    CCDDiagram diagram = new CCDDiagram("AnimalsDiagram");

    CCDClass_Factory class_factory = new CCDClass_Factory();
    CCDOperation_Factory op_factory = new CCDOperation_Factory();
    CCDAttribute_Factory attribute_factory = new CCDAttribute_Factory();
    CCDAssociation_Factory association_factory = new CCDAssociation_Factory();
    CCDParameter_Factory parameter_factory = new CCDParameter_Factory();
    CCDSpecialization_Factory specialization_factory = new CCDSpecialization_Factory();

    // Parent class: Pet
    CCDClass pet_class = class_factory.createInstance(diagram, "Pet");
    // The pet has a name
    CCDAttribute name_attribute = attribute_factory.createInstance(diagram,
        pet_class, CTBaseType.STRING_TYPE, "name");

    // Pet's child class: Dog
    CCDClass dog_class = class_factory.createInstance(diagram, "Dog");
    // The dog can bark
    CCDOperation bark_operation = op_factory.createInstance(diagram,
        "bark", dog_class);

    // The dog can bark at any volume, too!
    CCDOperation bark_volumed_operation = op_factory.createInstance(
        diagram, "bark", dog_class);
    CCDParameter bark_volume_parameter = parameter_factory.createInstance(
        diagram, bark_volumed_operation, CTBaseType.INTEGER_TYPE,
        "volume");
    CTType_isTypeOf_CTTypeDefined.addLink(CTBaseType.STRING_TYPE,
        bark_volumed_operation);
    // Pet's child class: Cat
    CCDClass cat_class = class_factory.createInstance(diagram, "Cat");

    // Pet's child class: Parrot
    CCDClass parrot_class = class_factory.createInstance(diagram, "Parrot");

    // Pet's child class: Rabbit
    CCDClass rabbit_class = class_factory.createInstance(diagram, "Rabbit");

    // Set of child classes
    TreeSet<CCDClass> pet_subClasses = new TreeSet<CCDClass>();
    pet_subClasses.add(dog_class);
    pet_subClasses.add(cat_class);
    pet_subClasses.add(parrot_class);
    pet_subClasses.add(rabbit_class);

    // Doggies and kitties are Pets
    CCDGeneralization pets_generalization = new CCDGeneralization_Factory()
        .createInstance(diagram, true, true, pet_class, pet_subClasses);

    // Dogs don't love cats
    List<CCDAssociation_Factory.Role_TemporaryContainer> hatred_roles =
      new ArrayList<CCDAssociation_Factory.Role_TemporaryContainer>(2);
    // The first in the list is the Dog
    hatred_roles.add(new CCDAssociation_Factory.Role_TemporaryContainer(
        "killer", new Multiplicity((short) 0, (short) 7),
        dog_class));
    // The second in the list is the Cat, thus the association is going to
    // have an implicit direction
    hatred_roles.add(new CCDAssociation_Factory.Role_TemporaryContainer(
        "victim", new Multiplicity((short) 1, Multiplicity.UNBOUNDED),
        cat_class));

    CCDAssociation_Factory.Association_Roles_Container hatred_container =
      association_factory
        .createInstances(diagram, "hates", hatred_roles);

    // Pets argue among them, sometimes
    List<CCDAssociation_Factory.Role_TemporaryContainer> argue_roles =
      new ArrayList<CCDAssociation_Factory.Role_TemporaryContainer>(2);
    // The first in the list is a Pet
    argue_roles.add(new CCDAssociation_Factory.Role_TemporaryContainer(
        "insulted", pet_class));
    // The second in the list is another Pet, thus the association is going
    // to have an implicit direction
    argue_roles.add(new CCDAssociation_Factory.Role_TemporaryContainer(
        "angry", pet_class));

    CCDAssociation_Factory.Association_Roles_Container argue_container =
      association_factory
        .createInstances(diagram, "argues", argue_roles);
```

```
    // When a Dog hates a Cat, there's always something to argue (it's a
    // specialization!)
    CCDSpecialization hatred_argue_specialization = specialization_factory
        .createInstance(diagram, argue_container.association,
            hatred_container.association);

    // Another class: Rat (it's not a pet, usually)
    CCDClass rat_class = class_factory.createInstance(diagram, "Rat");
    // Cats hunt rats!
    List<CCDAssociation_Factory.Role_TemporaryContainer> hunt_roles =
      new ArrayList<CCDAssociation_Factory.Role_TemporaryContainer>(
        2);
    hunt_roles.add(new CCDAssociation_Factory.Role_TemporaryContainer(
        "hunter", cat_class));
    hunt_roles.add(new CCDAssociation_Factory.Role_TemporaryContainer(
        new Multiplicity((short)3, Multiplicity.UNBOUNDED), rat_class));

    CCDAssociation_Factory.Association_Roles_Container hunt_container =
      association_factory
        .createInstances(diagram, "hunts", hunt_roles);
    CCDAttribute hunting_speed = attribute_factory.createInstance(diagram,
        hunt_container.association, CTBaseType.FLOAT_TYPE, "speed");

    return diagram;
  }

  public static String transform(CCDDiagram diagram) {
    return new XmlMarshaller().marshal(diagram);
  }

  public static void main(String[] args) {
    CCDDiagram diagram = prepareAnimalsTestDiagram();
    System.out.println(transform(diagram));
    System.exit(0);
  }
}
```

Il codice, che pure sembra lungo e complesso, ad una lettura più attenta si rivela piuttosto schematico e ripetitivo: ogni qual volta si voglia istanziare un componente, si deve richiamare il metodo `createInstance(...)` sull'opportuna istanza *Factory*; qualora il componente abbia associazioni con altri, nella maggior parte dei casi il metodo stesso provvederà a gestirle, altrimenti è sufficiente chiamare il metodo *static* `addLink(...)` sulla classe di associazione correlata. Per maggiori ragguagli sui parametri del caso, rimandiamo alla documentazione JavaDoc del codice. Il risultato del *marshalling* XML (notare: si tratta di una sola riga!) è il seguente.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ccdDiagram>
 <name>AnimalsDiagram</name>
 <components>
  <class identifierName="AnimalsDiagram_Pet" id="DCC_Element_1">
   <name>Pet</name>
   <properties>
    <attribute type="String"
    identifierName="AnimalsDiagram_Pet_name" id="DCC_Element_2">
     <name>name</name>
     <multiplicity maximum="1" minimum="0"/>
    </attribute>
   </properties>
  </class>
  <generalization
  complete="true" disjoint="true"
  identifierName="AnimalsDiagram__Pet__generalizes__Dog_Cat_Parrot_Rabbit"
  id="DCC_Element_10">
   <name></name>
   <subClasses>
    <subClass>AnimalsDiagram_Dog</subClass>
    <subClass>AnimalsDiagram_Cat</subClass>
    <subClass>AnimalsDiagram_Parrot</subClass>
    <subClass>AnimalsDiagram_Rabbit</subClass>
   </subClasses>
   <baseClass>AnimalsDiagram_Pet</baseClass>
  </generalization>
  <association
  identifierName="AnimalsDiagram_hates__associates__Dog_killer_Cat_victim"
  id="DCC_Element_11">
   <name>hates</name>
   <properties/>
   <roles>
    <role
    identifierName="AnimalsDiagram_killer__roleFor__Dog__inAssociation__hates"
    id="DCC_Element_12">
     <name>killer</name>
     <multiplicity maximum="7" minimum="0"/>
     <participantClass>AnimalsDiagram_Dog</participantClass>
    </role>
    <role
    identifierName="AnimalsDiagram_victim__roleFor__Cat__inAssociation__hates"
    id="DCC_Element_13">
     <name>victim</name>
     <multiplicity maximum="32767" minimum="1"/>
     <participantClass>AnimalsDiagram_Cat</participantClass>
    </role>
   </roles>
  </association>
  <association
  identifierName="AnimalsDiagram_argues__associates__Pet_insulted_Pet_angry"
  id="DCC_Element_14">
   <name>argues</name>
   <properties/>
   <roles>
    <role
    identifierName="AnimalsDiagram_insulted__roleFor__Pet__inAssociation__argues"
    id="DCC_Element_15">
     <name>insulted</name>
     <multiplicity maximum="32767" minimum="0"/>
     <participantClass>AnimalsDiagram_Pet</participantClass>
    </role>
    <role
    identifierName="AnimalsDiagram_angry__roleFor__Pet__inAssociation__argues"
    id="DCC_Element_16">
     <name>angry</name>
     <multiplicity maximum="32767" minimum="0"/>
     <participantClass>AnimalsDiagram_Pet</participantClass>
    </role>
   </roles>
  </association>
  <specialization
  identifierName="AnimalsDiagram_{subset}_hates__isSubSetOf__argues"
  id="DCC_Element_17">
   <name>{subset}</name>
   <destination>
    AnimalsDiagram_argues__associates__Pet_insulted_Pet_angry
   </destination>
   <source>
    AnimalsDiagram_hates__associates__Dog_killer_Cat_victim
   </source>
  </specialization>
  <class identifierName="AnimalsDiagram_Rat" id="DCC_Element_18">
   <name>Rat</name>
```

```
   <properties/>
  </class>
  <association
  identifierName="AnimalsDiagram_hunts__associates__Cat_hunter_Rat_"
  id="DCC_Element_19">
   <name>hunts</name>
   <properties>
    <attribute type="Float"
    identifierName=
    "AnimalsDiagram_hunts__associates__Cat_hunter_Rat__speed"
    id="DCC_Element_22">
     <name>speed</name>
     <multiplicity maximum="1" minimum="0"/>
    </attribute>
   </properties>
   <roles>
    <role
    identifierName=
    "AnimalsDiagram_hunter__roleFor__Cat__inAssociation__hunts"
    id="DCC_Element_20">
     <name>hunter</name>
     <multiplicity maximum="32767" minimum="0"/>
     <participantClass>AnimalsDiagram_Cat</participantClass>
    </role>
    <role
    identifierName="AnimalsDiagram___roleFor__Rat__inAssociation__hunts"
    id="DCC_Element_21">
     <name></name>
     <multiplicity maximum="32767" minimum="3"/>
     <participantClass>AnimalsDiagram_Rat</participantClass>
    </role>
   </roles>
  </association>
  <class identifierName="AnimalsDiagram_Dog" id="DCC_Element_3">
   <name>Dog</name>
   <properties>
    <operation
    identifierName="AnimalsDiagram_Dog_bark_void" id="DCC_Element_4">
     <name>bark</name>
     <multiplicity maximum="1" minimum="0"/>
     <inputParameters/>
     <returnTypeString>void</returnTypeString>
    </operation>
    <operation type="String"
    identifierName="AnimalsDiagram_Dog_bark_volume_Integer_String"
    id="DCC_Element_5">
     <name>bark</name>
     <multiplicity maximum="1" minimum="0"/>
     <inputParameters>
      <parameter type="Integer"
      identifierName=
      "AnimalsDiagram_Dog_bark_volume_Integer_String__parameter__volume"
      id="DCC_Element_6">
       <name>volume</name>
       <multiplicity maximum="1" minimum="0"/>
      </parameter>
     </inputParameters>
     <returnTypeString>String</returnTypeString>
    </operation>
   </properties>
  </class>
  <class identifierName="AnimalsDiagram_Cat" id="DCC_Element_7">
   <name>Cat</name>
   <properties/>
  </class>
  <class identifierName="AnimalsDiagram_Parrot" id="DCC_Element_8">
   <name>Parrot</name>
   <properties/>
  </class>
  <class identifierName="AnimalsDiagram_Rabbit" id="DCC_Element_9">
   <name>Rabbit</name>
   <properties/>
  </class>
 </components>
</ccdDiagram>
```

In questa struttura XML, che riprende completamente la struttura specificata nel Model, ci soffermiamo su tre aspetti principali.

Innanzi tutto, notiamo la presenza per ogni nodo (cioé ogni InnerComponent) di un identificatore (`id`) ed un nome identificativo (`identifierName`), la cui costruzione consente comunque di distinguere, ad esempio, tra metodi per i quali è stato effettuato *overloading*, come nel Listato estratto 4.

Listato 4 Gestione dell'*overloading* fra operazioni di una classe nella traduzione XML.

```
<operation
identifierName="AnimalsDiagram_Dog_bark_void" id="DCC_Element_4">
 <name>bark</name>
 <multiplicity maximum="1" minimum="0"/>
 <inputParameters/>
 <returnTypeString>void</returnTypeString>
</operation>
<operation type="String"
identifierName="AnimalsDiagram_Dog_bark_volume_Integer_String"
id="DCC_Element_5">
 <name>bark</name>
 <multiplicity maximum="1" minimum="0"/>
 <inputParameters>
  <parameter type="Integer"
  identifierName=
  "AnimalsDiagram_Dog_bark_volume_Integer_String__parameter__volume"
  id="DCC_Element_6">
   <name>volume</name>
   <multiplicity maximum="1" minimum="0"/>
  </parameter>
 </inputParameters>
 <returnTypeString>String</returnTypeString>
</operation>
```

In secondo luogo, notiamo come tutti i nodi corrispondenti a istanze di classi figlie di CCDInnerComponent siano sempre interni, mai figli diretti del nodo radice, `ccdDiagram`.

Infine, nella traduzione dal grafo delle istanze all'albero XML, sono stati risolti i possibili *cicli*: essendovi tutte responsabilità doppie, ogni associazione ha riferimenti da tutte le classi partecipanti: se non si fosse indicato come, e quando, interrompere la traduzione nel momento in cui si trovava un riferimento, per così dire, di ritorno, si sarebbe instaurato un *loop* infinito (una spiegazione più dettagliata è descritta in [SM08])! Ecco perché, per esempio, in CCDRelation si ha l'indicazione che andranno riportati i dati delle proprietà (CCDProperty) in formato XML, all'interno del proprio nodo (tramite l'annotazione `@XmlElements`):

```
public abstract class CCDRelation
extends CCDComponent {
  @XmlElementWrapper(name="properties")
  @XmlElements(value={
      @XmlElement(name="attribute",
          type=CCDAttribute.class, required=false),
      @XmlElement(name="operation",
          type=CCDOperation.class, required=false)
    }
```

```
  )
  private SortedSet<CCDProperty> properties;
  ...
}
```

mentre in CCDProperty viene indicato che la classe, o l'associazione, di cui fa parte, non va riportata (`@XmlTransient`):

```
public abstract class CCDProperty
extends CCDInnerComponent {
  ...
  @XmlTransient
  protected CCDRelation ownerRelation;
  ...
}
```

4.2 Verifica di una proprietà formale

Prendiamo come esempio il Diagramma delle Classi tratto dalle dispense del corso di Metodi Formali nell'Ingegneria del Software, [CM07b], basato su un'idea del Prof. Franconi. In esso, vogliamo dimostrare che non esistono *latin-lover*.

Per farlo, usiamo questo codice di test:

```
public class PropertyProvingTest {
  public static void main(String[] args) {
    // prepare the diagram to test
    CCDDiagram testDiagram = prepareEnglishItalianTestDiagram();

    // marshal the given diagram in XML
    String xmlString = new XmlMarshaller().marshal(testDiagram);

    /*  the formula to prove about the given diagram:
     *  no latin lover can exist!  */
    String testGoal = "all X (-EnglishAndItalians_LatinLover(X)).";

    // proving
    ProvingResult proof =
     new CCDProver9Prover(testDiagram).prove(testGoal);

    // from now on, print the results
    System.out.println("\n" +
        "########################################" + "\n" +
        "####   Given the model:                #" + "\n" +
        "########################################" + "\n"
    );
    System.out.println(xmlString);
    System.out.println("\n" +
        "########################################" + "\n" +
        "####   Given the goal:                 #" + "\n" +
        "########################################" + "\n"
    );
    System.out.println(testGoal);
    System.out.println("\n" +
        "########################################" + "\n" +
        "####   The formula is:                 #" + "\n" +
        "########################################" + "\n"
    );
    System.out.println(proof.proven);
    System.out.println("\n" +
        "########################################" + "\n" +
        "####   The proof is:                   #" + "\n" +
        "########################################" + "\n"
    );
    System.out.println(proof.proof);
    System.exit(0);
  }

  /**
   * Builds a sample diagram, taken from Prof. E. Franconi's idea.
   * @return A sample diagram
   */
  public static CCDDiagram prepareEnglishItalianTestDiagram() {
    CCDDiagram diagram = new CCDDiagram("EnglishAndItalians");

    // factories
    CCDClass_Factory class_factory = new CCDClass_Factory();
    CCDGeneralization_Factory generalization_factory =
      new CCDGeneralization_Factory();

    // classes
    CCDClass person_class = class_factory.createInstance(
        diagram, "Person");
    CCDClass italian_class = class_factory.createInstance(
        diagram, "Italian");
    CCDClass english_class = class_factory.createInstance(
        diagram, "English");
    CCDClass lazy_class = class_factory.createInstance(
        diagram, "Lazy");
    CCDClass latinLover_class = class_factory.createInstance(
        diagram, "LatinLover");
    CCDClass gentleman_class = class_factory.createInstance(
        diagram, "Gentleman");
    CCDClass hooligan_class = class_factory.createInstance(
        diagram, "Hooligan");

    // children in hierarchies
    Set<CCDClass> itaEnBrothers = new TreeSet<CCDClass>();
    itaEnBrothers.add(italian_class);
    itaEnBrothers.add(english_class);

    generalization_factory.createInstance(
        diagram, true, false, person_class, itaEnBrothers);

    Set<CCDClass> italianChildrenBrothers = new TreeSet<CCDClass>();
    italianChildrenBrothers.add(lazy_class);
    italianChildrenBrothers.add(latinLover_class);
```

```
    generalization_factory.createInstance(
        diagram, true, true, italian_class, italianChildrenBrothers);

    Set<CCDClass> englishChildrenBrothers = new TreeSet<CCDClass>();
    englishChildrenBrothers.add(gentleman_class);
    englishChildrenBrothers.add(hooligan_class);

    generalization_factory.createInstance(
        diagram, false, false, english_class, englishChildrenBrothers);

    // a latin lover is a gentleman
    generalization_factory.createInstance(
        diagram, gentleman_class, latinLover_class);

    return diagram;
  }
}
```

Da cui segue l'output

```
########################################
####   Given the model:                #
########################################

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ccdDiagram>
 <name>EnglishAndItalians</name>
 <components>
  <class identifierName="EnglishAndItalians_Person" id="DCC_Element_1">
   <name>Person</name>
   <properties/>
  </class>
  <generalization complete="false" disjoint="false"
  identifierName="EnglishAndItalians__English__generalizes__Gentleman_Hooligan" id="DCC_Element_10">
   <name></name>
   <subClasses>
    <subClass>EnglishAndItalians_Gentleman</subClass>
    <subClass>EnglishAndItalians_Hooligan</subClass>
   </subClasses>
   <baseClass>EnglishAndItalians_English</baseClass>
  </generalization>
  <generalization complete="false" disjoint="false"
  identifierName="EnglishAndItalians__Gentleman__generalizes__LatinLover" id="DCC_Element_11">
   <name></name>
   <subClasses>
    <subClass>EnglishAndItalians_LatinLover</subClass>
   </subClasses>
   <baseClass>EnglishAndItalians_Gentleman</baseClass>
  </generalization>
  <class identifierName="EnglishAndItalians_Italian" id="DCC_Element_2">
   <name>Italian</name>
   <properties/>
  </class>
  <class identifierName="EnglishAndItalians_English" id="DCC_Element_3">
   <name>English</name>
   <properties/>
  </class>
  <class identifierName="EnglishAndItalians_Lazy" id="DCC_Element_4">
   <name>Lazy</name>
   <properties/>
  </class>
  <class identifierName="EnglishAndItalians_LatinLover" id="DCC_Element_5">
   <name>LatinLover</name>
   <properties/>
  </class>
  <class identifierName="EnglishAndItalians_Gentleman" id="DCC_Element_6">
   <name>Gentleman</name>
   <properties/>
  </class>
  <class identifierName="EnglishAndItalians_Hooligan" id="DCC_Element_7">
   <name>Hooligan</name>
   <properties/>
  </class>
  <generalization complete="false" disjoint="true"
  identifierName="EnglishAndItalians__Person__generalizes__Italian_English" id="DCC_Element_8">
   <name></name>
   <subClasses>
    <subClass>EnglishAndItalians_Italian</subClass>
    <subClass>EnglishAndItalians_English</subClass>
   </subClasses>
   <baseClass>EnglishAndItalians_Person</baseClass>
  </generalization>
  <generalization complete="true" disjoint="true"
  identifierName="EnglishAndItalians__Italian__generalizes__Lazy_LatinLover" id="DCC_Element_9">
   <name></name>
   <subClasses>
    <subClass>EnglishAndItalians_Lazy</subClass>
    <subClass>EnglishAndItalians_LatinLover</subClass>
   </subClasses>
   <baseClass>EnglishAndItalians_Italian</baseClass>
  </generalization>
 </components>
</ccdDiagram>


########################################
####   Given the goal:                 #
########################################

all X (-EnglishAndItalians_LatinLover(X)).

########################################
```

```
####   The formula is:                #
########################################

true

########################################
####   The proof is:                  #
########################################


============================== prooftrans ============================
============================== Prover9 ===============================
Prover9 (32) version Dec-2007, Dec 2007.
Process 28342 was started by cdc on ubuntu.cdc.dis.uniroma1.it,
Tue Sep 23 13:11:05 2008
The command was "./prover9".
============================== end of head ===========================

============================== end of input ==========================

============================== PROOF =================================

% -------- Comments from original proof --------
% Proof 1 at 0.01 (+ 0.00) seconds.
% Length of proof is 15.
% Level of proof is 4.
% Maximum clause weight is 2.
% Given clauses 0.

1 (all X (EnglishAndItalians_Gentleman(X) -> EnglishAndItalians_English(X))).  [assumption].
3 (all X (EnglishAndItalians_LatinLover(X) -> EnglishAndItalians_Gentleman(X))).  [assumption].
6 (all X (EnglishAndItalians_Italian(X) -> -EnglishAndItalians_English(X))).  [assumption].
8 (all X (EnglishAndItalians_LatinLover(X) -> EnglishAndItalians_Italian(X))).  [assumption].
12 (all X -EnglishAndItalians_LatinLover(X)).  [goal].
13 -EnglishAndItalians_LatinLover(x) | EnglishAndItalians_Gentleman(x).  [clausify(3)].
14 -EnglishAndItalians_Gentleman(x) | EnglishAndItalians_English(x).  [clausify(1)].
20 -EnglishAndItalians_Italian(x) | -EnglishAndItalians_English(x).  [clausify(6)].
21 -EnglishAndItalians_LatinLover(x) | EnglishAndItalians_Italian(x).  [clausify(8)].
28 -EnglishAndItalians_LatinLover(x) | EnglishAndItalians_English(x).  [resolve(13,b,14,a)].
29 EnglishAndItalians_LatinLover(c1).  [deny(12)].
31 -EnglishAndItalians_LatinLover(x) | -EnglishAndItalians_English(x).  [resolve(21,b,20,a)].
34 EnglishAndItalians_English(c1).  [resolve(28,a,29,a)].
36 -EnglishAndItalians_English(c1).  [resolve(31,a,29,a)].
37 $F.  [copy(36),unit_del(a,34)].

============================== end of proof ==========================
```

In fondo, si legge la dimostrazione fornita da *Prover9* e filtrata da *Prooftrans*, programma fornito con il pacchetto di installazione di *Prover9*. Un passaggio intermedio, non stampato a video, in quanto non reso esplicito tramite accesso *public* al metodo opportuno (`createProverProgramInput(String)`), ha prodotto il seguente input per *Prover9* (codice in cui è evidente l'uso degli attributi identifierName dei nodi XML visti in precedenza per avere nomi univoci da assegnare ai predicati):

```
formulas(assumptions).
%%% Translation of conceptual class diagram: EnglishAndItalians
%%%
%%% [Generalization]  : Gentleman + Hooligan < English
%%%
all X ( EnglishAndItalians_Gentleman(X) -> EnglishAndItalians_English(X) ).
all X ( EnglishAndItalians_Hooligan(X) -> EnglishAndItalians_English(X) ).
%%% [Generalization]  : LatinLover < Gentleman
%%%
all X ( EnglishAndItalians_LatinLover(X) -> EnglishAndItalians_Gentleman(X) ).
%%% [Generalization]  : Italian + English < Person
%%%
all X ( EnglishAndItalians_Italian(X) -> EnglishAndItalians_Person(X) ).
all X ( EnglishAndItalians_English(X) -> EnglishAndItalians_Person(X) ).
%%% >Disjointness
%%%
all X ( EnglishAndItalians_Italian(X) -> - EnglishAndItalians_English(X) ).
%%% [Generalization]  : Lazy + LatinLover < Italian
%%%
all X ( EnglishAndItalians_Lazy(X) -> EnglishAndItalians_Italian(X) ).
all X ( EnglishAndItalians_LatinLover(X) -> EnglishAndItalians_Italian(X) ).
%%% >Disjointness
%%%
all X ( EnglishAndItalians_Lazy(X) -> - EnglishAndItalians_LatinLover(X) ).
%%% >Completeness
%%%
all X ( EnglishAndItalians_Italian(X) -> (
EnglishAndItalians_Lazy(X) |
EnglishAndItalians_LatinLover(X)
)
).
%%% Every variable must be an instance, or a value
%%%
all X (
EnglishAndItalians_Person(X) |
EnglishAndItalians_Italian(X) |
EnglishAndItalians_English(X) |
EnglishAndItalians_Lazy(X) |
EnglishAndItalians_LatinLover(X) |
EnglishAndItalians_Gentleman(X) |
EnglishAndItalians_Hooligan(X)
).
%%% Implicit disjointness
%%%
%%% >Implicit disjointness for classes not linked by hierarchies or generalizations, among them, and with respect to types
%%%
%%% >Implicit disjointness among types
%%%
end_of_list.
formulas(goals).
all X (-EnglishAndItalians_LatinLover(X)).
end_of_list.
```

4.3 Disegno di un diagramma tramite il motore grafico di ReDiA-VeriFInt

Supponiamo di voler graficare, utilizzando il motore grafico di ReDiA-VeriFInt, il diagramma in Figura 4.2.

Di seguito è riportato il codice sorgente Java necessario per ottenere il risultato grafico visualizzato in figura 4.3

```
public class GraphicDiagramTest {
    private Set<Drawable> createTestDiagram()
    {
        ogg = new TreeSet();
        Frame person_frame = new Frame(new Point(10, 100));
        Label person_label = new Label("Person");
        person_label.setFont(fontBold);
        person_label.setTextAlignment(Label.ALIGNMENT_CENTER);
        person_frame.getFigure().add(person_label);
        person_frame.getFigure().add(new Figure());
        person_frame.getFigure().add(new Label("name : String"));
        person_frame.getFigure().add(new Label("surname : String"));
        person_frame.getFigure().add(new Label("birth_date : Date"));
        person_frame.getFigure().add(new Label("regional_code : String"));
```

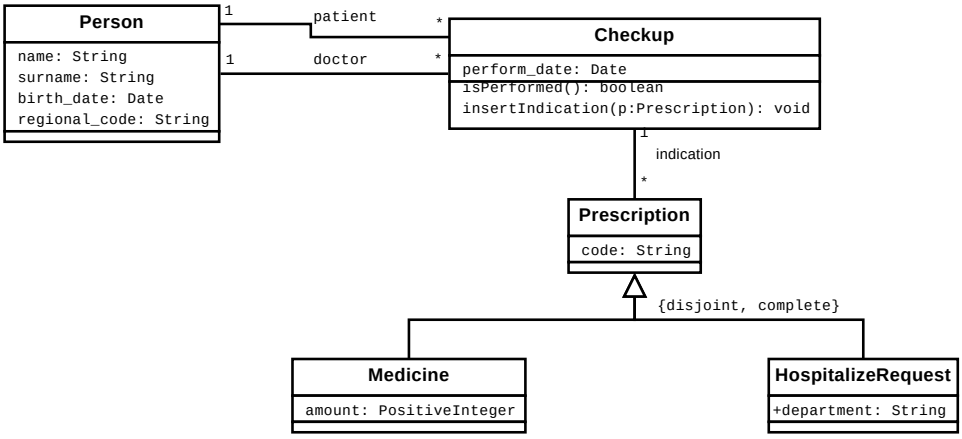

Figura 4.2: Un diagramma di esempio che si vuole graficare

```
ogg.add(person_frame);

Frame checkup_frame = new Frame(new Point(250, 50));
Label checkup_label = new Label("Checkup");
checkup_label.setFont(fontBold);
checkup_label.setTextAlignment(Label.ALIGNMENT_CENTER);
checkup_frame.getFigure().add(checkup_label);
checkup_frame.getFigure().add(new Figure());
checkup_frame.getFigure().add(new Label("perform_date : Date"));
checkup_frame.getFigure().add(new Figure());
checkup_frame.getFigure().add(new Label("isPerformed() : Boolean"));
checkup_frame.getFigure().add(new Label("insertIndication(p: Prescription) : Void"));
ogg.add(checkup_frame);

Frame prescription_frame = new Frame(new Point(300, 250));
Label prescription_label = new Label("Prescription");
prescription_label.setFont(fontBold);
prescription_label.setTextAlignment(Label.ALIGNMENT_CENTER);
prescription_frame.getFigure().add(prescription_label);
prescription_frame.getFigure().add(new Figure());
prescription_frame.getFigure().add(new Label("code : String"));
ogg.add(prescription_frame);

Frame medicine_frame = new Frame(new Point(200, 380));
Label medicine_label = new Label("Medicine");
medicine_label.setFont(fontBold);
medicine_label.setTextAlignment(Label.ALIGNMENT_CENTER);
medicine_frame.getFigure().add(medicine_label);
medicine_frame.getFigure().add(new Figure());
medicine_frame.getFigure().add(new Label("amount : PositiveInteger"));
ogg.add(medicine_frame);

Frame hospitalize_request_frame = new Frame(new Point(350, 380));
Label hospitalize_request_label = new Label("HospitalizeRequest");
hospitalize_request_label.setFont(fontBold);
hospitalize_request_label.setTextAlignment(Label.ALIGNMENT_CENTER);
hospitalize_request_frame.getFigure().add(hospitalize_request_label);
hospitalize_request_frame.getFigure().add(new Figure());
hospitalize_request_frame.getFigure().add(new Label("department : String"));
ogg.add(hospitalize_request_frame);

Edge patient_edge = new Edge(person_frame, checkup_frame);
patient_edge.setLabelName("patient");
patient_edge.setMultiplicity(0, "1..1");
patient_edge.setMultiplicity(1, "0..*");
patient_edge.setStyle(Edge.STYLE_SIMPLEASSOCIATION);
ogg.add(patient_edge);

Edge doctor_edge = new Edge(person_frame, checkup_frame);
doctor_edge.setLabelName("doctor");
doctor_edge.setMultiplicity(0, "1..1");
doctor_edge.setMultiplicity(1, "0..*");
doctor_edge.setStyle(Edge.STYLE_SIMPLEASSOCIATION);
```

```
        ogg.add(doctor_edge);

        Edge indication_edge = new Edge(checkup_frame, prescription_frame);
        indication_edge.setLabelName("indication");
        indication_edge.setMultiplicity(0, "1..1");
        indication_edge.setMultiplicity(1, "0..*");
        indication_edge.setStyle(Edge.STYLE_SIMPLEASSOCIATION);
        ogg.add(indication_edge);

        ArrayList prescription_subClasses = new ArrayList();
        prescription_subClasses.add(medicine_frame);
        prescription_subClasses.add(hospitalize_request_frame);
        Edge prescription_generalization = new Edge(prescription_subClasses, prescription_frame);
        prescription_generalization.setLabelName("{disjoint, complete}");
        prescription_generalization.setStyle(Edge.STYLE_ISA);
        ogg.add(prescription_generalization);

        return ogg;
    }
}
```

Il codice genera tutte le strutture dati necessarie per visualizzare il diagramma riportato nella figura 4.3. Da ricordare c'è anche che il motore grafico supporta lo spostamento dinamico degli oggetti sullo schermo con relativo riadattamento delle posizioni degli archi.

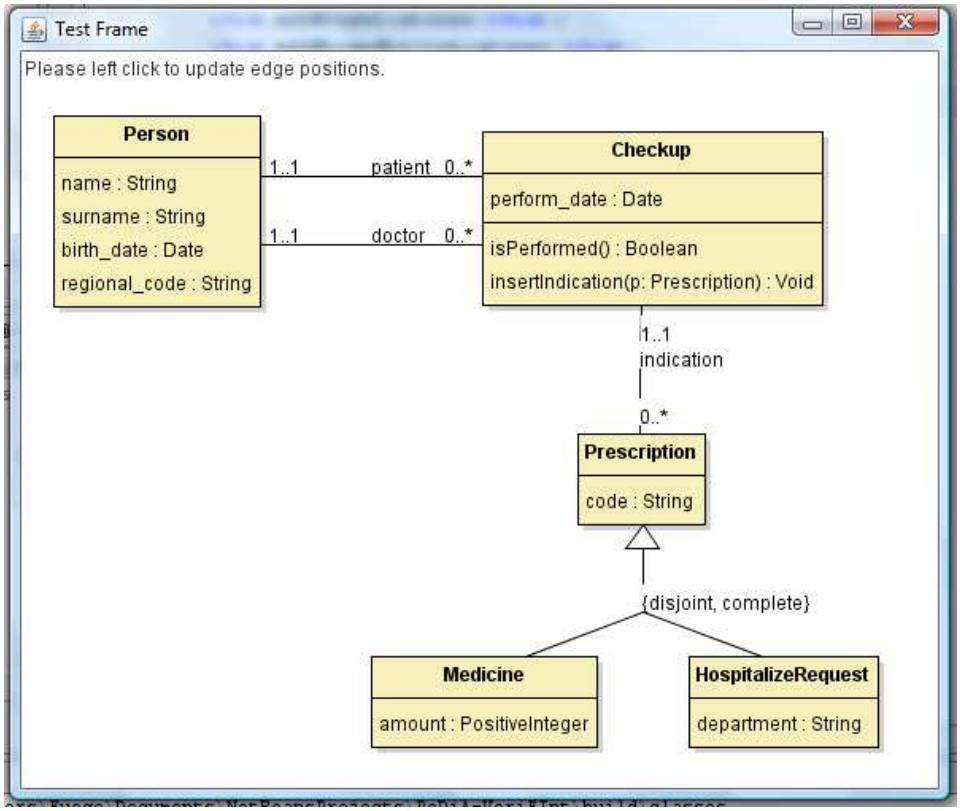

Figura 4.3: Il risultato generato dal motore di rendering

Capitolo 5

Sviluppi futuri

Suggeriamo, in quest'ultima sezione, dei punti da sviluppare in futuro, allo scopo di ampliare e completare questo progetto.

5.1 Estensioni

1. Supporto alla definizione di vincoli esterni ai diagrammi delle classi in *OCL*[1] (*Object Constraint Language*).
   (a) L'utente deve poter esprimere vincoli in OCL sugli elementi del diagramma
   (b) L'applicazione deve includere tali vincoli nel diagramma, nella relativa serializzazione XML, e tradurre il tutto in formule di input per il dimostratore automatico
   (c) L'utente deve poter esprimere le tesi da dimostrare direttamente in OCL (si deve dunque dotare l'applicazione di un *pannello di controllo* che consenta di farlo tramite interfaccia grafica)
2. Definizione e gestione dei tipi definiti dall'utente[2]
3. Supporto per la traduzione del diagramma, serializzato XML, in testo di input per un altro dimostratore automatico, *Alloy Analyzer*[3]
4. Attivazione automatica del dimostratore, per verificare tesi *ad hoc* al momento dell'aggiunta/modifica di componenti del diagramma[4] (ad esempio: al momento della creazione di una gerarchia, il programma deve essere in grado di verificare automaticamente che fra le classi figlie non ce ne siano di inconsistenti)

[1] Si può implementare l'interfaccia XConstraint (v. Sezione 2.2.7); la classe di associazione fra XConstraint e XConstrainable è già implementata

[2] Si estenda quanto già presente e descritto nella Sezione 2.2.6

[3] `http://alloy.mit.edu/community/`

[4] Si consiglia di usare degli *Interceptor* sui WidgetEvent – v. Sezione 2.3.2

Più avanti, sarà naturalmente possibile estendere ulteriormente l'applicazione, includendo la gestione di Diagrammi delle Classi (*realizzativi*), *Use Case*, degli *Stati e Transizioni*, ecc.

5.2 Perfezionamenti

1. Completamento dell'interfaccia grafica, al livello *View*, e delle classi di gestione del livello *Control*
   (a) Conclusione dell'implementazione dei Widget e del DCCWidgetsManager (v. Sezione 2.3.2)
   (b) Connessione dei componenti dell'*engine* grafico (v. Sezione 2.3.1) al diagramma memorizzato, a livello *Model*, tramite l'uso di primitive più ad alto livello
   (c) Creazione di un'interfaccia utente interattiva che includa ed organizzi i vari componenti grafici
2. Per la trasposizione in formule della logica del prim'ordine dei componenti del diagramma delle classi, restano da sviluppare o concludere:
   (a) la gestione di parametri di ingresso e parametri di ritorno, per le operazioni, con molteplicità diversa da { 0..1 }
   (b) la gestione di operazioni nelle associazioni
   (c) la gestione di ereditarietà multiple, nel caso in cui le classi madri non siano legate da un ramo di gerarchia condivisa

Appendice A

Tecnologia JAXB

In questa sezione verrà illustrata sinteticamente la tecnologia ***JAXB***. Per informazioni più dettagliate, raccomandiamo la lettura di [OM03] e [Sun], e delle citazioni menzionate via via lungo le sezioni a seguire.

A.1 Cos'è JAXB

JAXB è l'acronimo di **J**ava **A**rchitecture for **X**ML **B**inding ed indica la tecnologia, studiata appositamente per la piattaforma Java, che consente di convertire in XML determinate istanze di oggetti presenti in memoria e viceversa. L'azione di conversione da oggetti a XML è detta *Marshalling* mentre l'estrapolazione di oggetti da file XML è detta *Unmarshalling*: la figura A.1 fornisce una rappresentazione grafica di tali operazioni.

I vantaggi che queste interfacce portano sono innumerevoli; uno di questi è la mancata necessità di scrivere manualmente codice specifico utilizzando ad esempio *SAX*[1] o *DOM*[2]: si evita così di implementare un *parser* da zero; inoltre, ogni classe del *Model* definisce fin da subito il proprio *DTO* in formato XML.

Il ***JAXB***, essendo basato su *annotazioni* da inserire all'interno del codice Java riguardante la definizione delle classi, è particolarmente indicato nei progetti in cui i dati da convertire in XML hanno una certa complessità, o in cui la loro forma è soggetta a variazioni nel tempo.

[1] Un *framework* di gestione di documenti XML in base al quale si effettua il *parsing* basandosi su chiamate asincrone, *event-driven*

[2] Un altro *framework* di gestione di documenti XML che si basa sulla memorizzazione in una struttura dati di tutto l'albero XML, al fine di accedervi

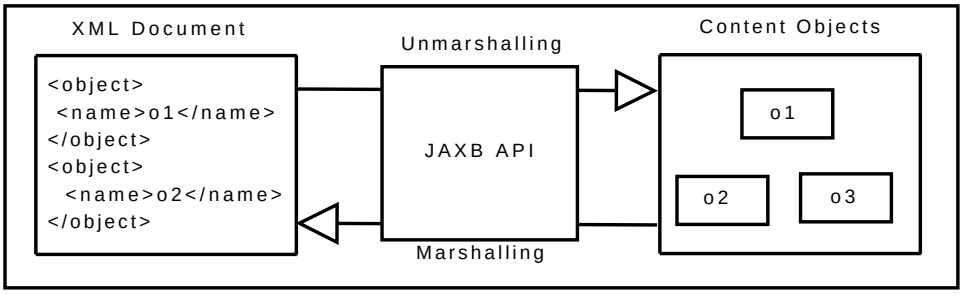

Figura A.1: Marshalling e Unmarshalling da e verso un file XML utilizzando l'interfaccia JAXB

A.2 Utilizzare JAXB

A.2.1 Annotazioni *XML*

Le *Annotations* JAXB si distinguono perché precedute dal prefisso `@Xml`; si riferiscono alla riga di codice successiva a quella in cui vengono riportate (può essere una dichiarazione di classe, di attributo, di metodo, o di package). Di seguito ne sono descritte molto brevemente alcune, più precisamente le più importanti tra quelle utilizzate all'interno del progetto ReDiA-VeriFIntPer un riferimento piú approfondito, v. [SMa].

@XmlRootElement viene inserito prima della riga di dichiarazione di una classe Java e indica che quella classe rappresenterà il nodo radice del file XML.

@XmlAccessorType(XmlAccessType) indica, tramite il passaggio di un valore dell'*Enum* ***XmlAccessType***, quali elementi della classe devono essere serializzati. Esempio:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class CCDDiagram
    implements Diagram,Comparable<CCDDiagram>,
    Serializable, XmlSerializable {
...
```

implica che i nella traduzione XML si avrà un documento il cui nodo radice sarà `<ccdElement/>` (viene effettuata una conversione in lettere minuscole durante il processo di serializzazione, laddove l'annotazione `@XmlRootElement` non abbia come parametro un valore esplicito per `name`), i cui figli saranno gli *attributi* non *static* della classe (indipendentemente dallo specificatore di accesso)

@XmlElement si antepone ad una qualunque proprietà (metodo o attributo), e crea un mapping tra un elemento XML e una proprietà di una classe, eventualmente sovrascrivendo il comportamento dettato da *@XmlAccessorType*

@XmlTransient indica che la proprietà così annotata non verrà riportata nell'albero XML risultante (un tipico caso d'uso è nella sezione 4.1), eventualmente sovrascrivendo il comportamento dettato da *@XmlAccessorType*. Esempio:

```
public abstract class CCDProperty ... {
...
    @XmlTransient
    protected CCDRelation ownerRelation;
...
```

comporta che non ci sarà la traduzione in XML dell'attributo CCDRelation all'interno del nodo rappresentante CCDRelation

Per usi di livello più avanzato, raccomandiamo l'ispezione del codice delle classi del package it.uniroma1.dis.mfis.redia.verifint.model:

analysis.classDiagrams.CCDDiagram per la gestione di collezioni di oggetti (una tecnica alternativa, basata sul *lazy binding*, si trova in classDiagrams.CCDRelation)

analysis.classDiagrams.CCDProperty per l'uso di *XmlJavaTypeAdapter*, ossia "traduttori" specializzati di oggetti (particolarmente utili nel caso in cui si vogliano serializzare oggetti il cui riferimento è tipizzato in base ad un'interfaccia – JAXB non gestisce interfacce!), e per l'uso degli identificatori XML

analysis.types._xml.CTTypeXmlImplementation per l'implementazione di *XmlJavaTypeAdapter*, e la specifica di identificatori XML

ElementId per la gestione di tipi di dato

A.2.2 Marshalling verso un file XML

L'operazione di *Marshalling*, come descritto precedentemente, consiste nel convertire le istanze delle classi presenti in memoria in un file strutturato XML. Questa procedura viene svolta esclusivamente dalle API Java fornite con ***JAXB***. Di seguito è riportata una parte di codice Java per eseguire il *Marshalling*, si assume che il package `javax.xml.bind.*` sia importato:

```
Marshaller marshaller = context.createMarshaller();
marshaller.marshal(
    xmlSerializableObject,
    new FileOutputStream("output.xml") );
```

A.2.3 Unmarshalling da file XML

L'operazione di *Unmarshalling* è inversa al *Marshalling* descritto sopra. Consiste nell'aprire un file XML e ricreare in memoria le istanze degli oggetti descritti all'interno di esso. Di seguito è riportata la parte di codice dedicata a tale operazione, si assume anche in questo esempio che il package `javax.xml.bind.*` sia importato e che il file XML di input sia scritto in formato compatibile con la classe **MyXmlSerializableObject** che si vuole caricare da file:

```
Unmarshaller unmarshaller = context.createUnmarshaller();
MyXmlSerializableObject xmlSerializableObject =
    (MyXmlSerializableObject)
    unmarshaller.unmarshal(new File("input.xml"));
```

Appendice B

Tecnologia XSLT

Andremo ora a presentare, sinteticamente, la tecnologia XSLT (e**X**tensible **S**tylesheet **L**anguage **T**ransformations). Per informazioni più dettagliate, consigliamo la lettura dell'ottimo [CB02], e, su web, di [W3S] come *tutorial* iniziale e [ZVO] a livello più avanzato – oltre naturalmente alla *Recommendation* del W3C, [W3C99b]. Soprattutto, consigliamo la consultazione della dispensa stilata da Andrea Frasca specifica per gli studenti di Metodi Formali nell'Ingegneria del Software, [Fra08]. Nel resto del capitolo, compaiono altri riferimenti specifici attraverso i quali poter approfondire.

Per comprendere a fondo il linguaggio XSLT, è propedeutico studiare la sintassi XPath (descritta nel già citato [CB02], al capitolo 6, e, sul web, anche in questo caso, c'è la *Recommendation* del W3C, [W3C99a]), usata per le espressioni di ricognizione dei nodi XML in analisi.

B.1 Cos'è XSLT

XSLT è un linguaggio standardizzato dal *W3C*, scritto in **sintassi XML**, che consente di trasformare la struttura di un documento XML. L'applicazione più consueta di XSLT è convertire un documento XML in un documento XHTML, ma qualunque formato di testo è ammesso in uscita al *processamento* XSLT, anche testo semplice o, come nel nostro caso, un documento la cui sintassi sia quella delle direttive di input di un dimostratore automatico di teoremi! Il ruolo del ***processore* XSLT** è quello di applicare le regole di trasformazione, date da un *foglio di stile* XSLT, ad un documento di origine XML. Attualmente, persino i browser più all'avanguardia possono svoglgere il ruolo di *processori* XSLT. In ReDiA-VeriFInt, usiamo il processore *Xalan*[1] della *Apache Software Foundation*[2], *bundled* nella JDK 1.6. Il codice che sfrutta il motore Xalan in ReDiA-VeriFInt è riportato nel Listato 3.

[1] `http://xml.apache.org/xalan-j/`
[2] `http://www.apache.org/`

Per ulteriori informazioni, consultare la documetazione in [SMb] e leggere [Bur01].

XSLT è un ***linguaggio dichiarativo***, nel senso che descrive il trasferimento dei dati, ma *senza istruzioni procedurali*. Ad esempio, si possono dichiarare variabili, ma non aggiornare i loro valori in seguito. Tuttavia, consente la ricursione (v. [RH06]), è ***Turing-completo***: per questo, possiamo delegare ad esso interamente la manipolazione dei documenti XML, senza usare una sola riga di codice Java nelle regole di trasformazione.

B.2 Utilizzare XSLT

Un esempio di come usare il processore è riportato in 3. Ci soffermeremo quindi sul foglio di stile XSLT adottato realmente in ReDiA-VeriFInt per tradurre il diagramma concettuale delle classi da XML a sintassi di input per *Prover9*.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns=""
>
  <xsl:output
    method="text"
    encoding="utf-8"
    omit-xml-declaration="yes"
    indent="no"/>
```

Innanzi tutto, il file inizia, come ogni file XML, con una *XML Declaration*. Segue il nodo radice, `stylesheet`, la quale indica la versione di XSLT in uso (noi usiamo la `1.0`, e non la `2.0`, seppure già disponibile, in quanto gran parte dei processori, ad oggi, non supportano il nuovo standard – tra questi, anche *Xalan*). Gli attributi `xmlns` specificano i *namespace* che identificano univocamente lo schema del documento. Per ReDiA-VeriFInt non è stato ancora definito, mentre per XSLT è noto, e pertanto va indicato, onde consentire al processore di capire con quali schemi di documento stia lavorando. Si è soliti specificare con un *namespace-prefix* (come `xsl`) gli elementi XSLT. Lo faremo anche noi.
Segue il nodo *output*, attraverso il quale si specificano le modalità di formattazione del documento in uscita. Nel nostro caso, deve essere un documento di testo, codificato `UTF-8`, senza *XML Declaration*, e non indentato.

```
  <xsl:variable name="UNBOUNDED_MULTIPLICITY">32767</xsl:variable>
  <xsl:variable name="VOID_RETURN">void</xsl:variable>
  <xsl:variable name="DEFAULT_SUBSET_NAME"><![CDATA[{subset}]]></xsl:variable>
```

Quella che precede é una dichiarazione di costanti; come vedremo più avanti, per accedere ad esse, occorre anteporre un simbolo di `$` al nome indicato come attributo.

```
  <xsl:template match="/">
    <!-- comment -->
```

```
  <xsl:text><![CDATA[%%% Translation of conceptual class diagram: ]]></xsl:text>
  <xsl:value-of select="ccdDiagram/name"/>
  <![CDATA[%%%]]>
  <!-- end of comment -->
  <xsl:apply-templates/>
  <xsl:apply-templates select="ccdDiagram" mode="CWA"/>
  <xsl:apply-templates select="ccdDiagram" mode="UNA"/>
</xsl:template>
```

A questo punto abbiamo il primo costrutto efficace: i nodi `template` dettano le regole di trasformazione al loro interno. L'attributo `match` specifica il percorso del nodo da analizzare, attraverso espressioni ***XPath***: in questo caso, ci si riferisce al nodo radice (*root*). Il nodo `text` indica che ciò che viene scritto al suo interno, diverrà direttamente testo all'esterno, non alterato (normalmente, ad esempio, nel testo interno ai nodi, gli invii a capo non vengono considerati, se non come normale spazio bianco), e senza che il testo in output venga scritto in una riga successiva, come accadrebbe scrivendo testo libero (ad esempio con il campo `CDATA` che segue). Il nodo `value-of` seleziona e stampa il contenuto degli elementi indicati dall'espressione *XPath* (valutata a partire dal nodo contestuale, indicato nel `template`) presente nell'attributo `select`. Infine, il nodo *apply-templates* invoca il processamento dei nodi figli, per default, o di quelli indicati, sempre tramite *XPath*, nell'attributo `select`. L'attributo *mode* consente di identificare regole di trasformazione diverse da attivare – a patto che esistano `template` che dichiarino lo stesso `mode`, come nel codice che segue:

```
<xsl:template match="ccdDiagram" mode="CWA">
  <!-- comment -->
  <![CDATA[%%%  Every variable must be an instance, or a value]]>
  <![CDATA[%%%]]>
  <!-- end of comment -->
  <![CDATA[all X (]]>
  <xsl:for-each select="//class">
    <xsl:if test="position() &gt; 1">
      <xsl:text><![CDATA[ |]]><!-- \n -->
      </xsl:text>
    </xsl:if>
    <xsl:value-of select="@identifierName"/>
    <xsl:text>(X)</xsl:text>
  </xsl:for-each>
  <xsl:for-each select="//@type[not(preceding::*/@type = .)
      and not(ancestor::ccdDiagram//class/@identifierName = .)]">
    <xsl:text><![CDATA[ |]]><!-- \n -->
    </xsl:text>
    <xsl:value-of select="."/>
    <xsl:text>(X)</xsl:text>
  </xsl:for-each>
  <![CDATA[).]]>
</xsl:template>
```

Di questo codice, esploriamo i costrutti più particolari:

- l'elemento `for-each` consente di ripetere, per tutti i nodi selezionati nell'espressione *XPath*, quanto descritto all'interno
- l'elemento `if` consente di eseguire o meno quanto descritto al proprio interno, a patto che sia rispettata la condizione specificata nell'attributo `test` (in questo caso, a patto che la posizione del nodo corrente sia, nella lista, maggiore di 1)

Le espressioni XPath possono raggiungere una complessità elevata: sono molto potenti! Ad esempio,

```
<xsl:for-each select="//@type[not(preceding::*/@type = .)
     and not(ancestor::ccdDiagram//class/@identifierName = .)]">
  ...
</xsl:for-each>
```

indica che si dovranno eseguire dei processamenti per tutti i nodi-attributo (`@`) di nome `type`, discendenti diretti o indiretti del nodo corrente (`//`), che rispettino la condizione fra parentesi quadre, ovvero tali che non ci siano nodi precedenti (*imparentati* o meno, basta che siano antecedenti nell'albero XML) il cui attributo `type` sia pari a quello corrente (`.`), e non ci siano nodi di nome `class` discendenti dell'elemento `ccdDiagram`, che del nodo corrente è antenato (`ancestor::`), i quali abbiano un attributo `identifierName` pari al corrente.

Per concludere, vediamo un esempio di *funzione ricorsiva*:

```
<!-- exists_zN_to_1(\$currentNumber) -->
<!--
  Writes, given \$currentNumber = 5,
  exists Z5 exists Z4 ... exists Z1
-->
<xsl:template name="exists_zN_to_1">
  <xsl:param name="currentNumber"/>
    <![CDATA[exists Z]]><xsl:value-of select="\$currentNumber"/>
    <xsl:text> </xsl:text>
    <xsl:if test="\$currentNumber \&gt; 1">
      <xsl:call-template name="exists_zN_to_1">
        <xsl:with-param name="currentNumber">
          <xsl:value-of select="\$currentNumber - 1"/>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:if>
</xsl:template>
```

In XSLT, *funzioni* e *template* condividono il nome identificativo, `template` appunto, (e infatti, anche i template possono essere *parametrizzati*, ma questo è un discorso avanzato che non è il caso di trattare in questa sede). Le funzioni sono caratterizzate dalla presenza dell'attributo `name`, e dal fatto che, per indicarne l'invocazione, non si usa l'elemento `<apply-templates select=.../>`, bensì `<call-template name=.../>`. Le funzioni prevedono eventuali parametri formali, dichiarati dai nodi `param`, il cui nome viene specificato nell'attributo `name`. Si badi che, all'interno del "corpo" della funzione, il parametro attuale viene indicato tramite il simbolo `$`, come avviene anche per le variabili globali (costanti). Il passaggio di parametri, all'interno dell'*invocazione* `call-template`, avviene attraverso l'elemento `with-param`, al cui interno viene posto il valore da attribuire. In questo ultimo tratto di codice, abbiamo visto dunque come si scriva una funzione ricorsiva, in XSLT, che emuli un *loop*.

Per applicazioni più complesse dei concetti fin qui espressi, magari dopo la lettura dei documenti consigliati, e un po' di esercizio, si può esplorare il codice del foglio di stile XSLT usato dall'applicazione, `prover9.xslt`, all'interno della directory `/resources/xml/xslt/ccd`. L'effetto della sua applicazione su un documento XML è stato mostrato in 4.2

Bibliografia

[Bur01] Eric M. Burke. Top Ten Java and XSLT Tips. Technical report, O'Reilly – OnJava.com, `http://www.oreillynet.com/pub/a/oreilly/java/news/javaxslt\_0801.html`, 2001.

[CB02] Kelly Carey and Stanco Blatnik. *Guida a XML (XML: Content and Data)*, chapter 6 - 7. McGraw-Hill, 2002.

[CM07a] Marco Cadoli and Toni Mancini. *Metodi Formali nell'Ingegneria del Software*, chapter 7 - Diagrammi UML delle classi. SAPIENZA - Dipartimento di Informatica e Sistemistica, 2007.

[CM07b] Marco Cadoli and Toni Mancini. *Metodi Formali nell'Ingegneria del Software*, chapter 7 - Diagrammi UML delle classi, page 104. SAPIENZA - Dipartimento di Informatica e Sistemistica, 2007. Figura 7.15.

[Fra08] Andrea Frasca. *XML e XSLT per la generazione automatica di codice*. SAPEINZA - Dipartimento di Informatica e Sistemistica, 2008.

[McC03] William McCune. *OTTER 3.3 Reference Manual*. `http://www.cs.unm.edu/~mccune/otter/Otter33.pdf`, 2003. Sito web ufficiale.

[McC08] William McCune. *Prover9 Manual*. `http://www.cs.unm.edu/~mccune/prover9/manual/2008-09A/`, 2008. Sito web ufficiale.

[MS08a] Toni Mancini and Monica Scannapieco. *Corso di Progettazione del Software per il Corso di Laurea in Ingegneria Gestionale*, chapter S.A.1. SAPIENZA - Dipartimento di Informatica e Sistemistica, 2008. `http://www.dis.uniroma1.it/~tmancini`.

[MS08b] Toni Mancini and Monica Scannapieco. *Corso di Progettazione del Software per il Corso di Laurea in Ingegneria Gestionale*, chapter S.R.1-S.R.5. SAPIENZA - Dipartimento di Informatica e Sistemistica, 2008. `http://www.dis.uniroma1.it/~tmancini`.

[OM03] Ed Ort and Bhakti Mehta. Java Architecture for XML Binding (JAXB). Technical report, Sun Microsystems, Inc., http://java.sun.com/developer/technicalArticles/WebServices/jaxb/, 2003.

[Pro07] Michele Proni. OOPS: un'applicazione per il supporto alla progettazione e alla realizzazione di software object oriented. Tesi di Laurea in Ingegneria Gestionale a.a. 2006-2007, presso SAPIENZA – Università di Roma, 2007.

[RH06] Elliotte Rusty Harold. Loop with recursion in XSLT. Technical report, IBM, Luglio 2006.

[SMa] Inc. Sun Microsystems. Javadoc: javax.xml.bind.annotation. http://java.sun.com/javase/6/docs/api/index.html?javax/xml/bind/annotation/package-summary.html.

[SMb] Inc. Sun Microsystems. Javadoc: javax.xml.transform. http://java.sun.com/javase/6/docs/api/javax/xml/transform/package-summary.html.

[SM08] Inc. Sun Microsystems. Mapping cyclic references to XML. Technical report, Sun Microsystems, Inc., https://jaxb.dev.java.net/guide/Mapping_cyclic_references_to_XML.html, 2008.

[Sun] Sun Microsystems, Inc., https://jaxb.dev.java.net/guide/index.html. *Unofficial JAXB Guide*.

[W3C99a] W3C, http://www.w3.org/TR/xpath. *XML Path Language (XPath) Version 1.0 W3C Recommendation*, Novembre 1999.

[W3C99b] W3C, http://www.w3.org/TR/xslt.html. *XSL Transformations (XSLT) Version 1.0 W3C Recommendation*, Novembre 1999.

[W3S] W3Schools. W3Schools XSLT Tutorial. http://www.w3schools.com/xsl/default.asp.

[ZVO] ZVON.org. ZVON XSLT Tutorial. http://www.zvon.org/xxl/XSLTutorial/Books/Output/contents.html.