Relazione di “Converger”

Dario Pavllo Gabriele Graffieti

26 febbraio 2015

Sommario

Converger è un CAS (Computer Algebra System) open source scritto in Java, e distribuito sotto licenza GPLv3. Brevemente, consiste in un sistema capace di leggere, stampare e manipolare espressioni matematiche, sia in modo simbolico che in modo numerico.

Il software è stato realizzato come progetto per l'esame di Programmazione ad Oggetti del corso di Ingegneria e Scienze Informatiche (Università di Bologna, A.A. 2014/2015).

Indice

Capitolo 1

Analisi

1.1 Requisiti

Essendo un Computer Algebra System, Converger dovrà fornire all'utente un sistema in grado di lavorare con le espressioni matematiche, al fine di facilitare lo svolgimento di calcoli lunghi e/o complessi.

L'applicazione sarà suddivisa in due blocchi principali: il *framework matematico*, che si occuperà di manipolare le espressioni e sarà potenzialmente utilizzabile come libreria a sè stante, e l'*interfaccia grafica*, che si occuperà di gestire le interazioni con l'utente.

Funzionalità del framework

- Parsing di espressioni matematiche in notazione infissa, permettendo anche di usare una sintassi sufficientemente elastica
- Manipolazione simbolica di espressioni: semplificazione algebrica, sostituzione di variabili con sottoespressioni, calcolo della derivata di una funzione, serie di Taylor
- Manipolazione numerica (cioè applicando algoritmi numerici che forniscono un risultato approssimato): approssimazione del valore di una funzione in un punto, risoluzione di equazioni, integrali definiti
- Stampa di espressioni matematiche, sia come testo normale che come testo in linguaggio LaTeX

Funzionalità della GUI

- Creazione di un ambiente di lavoro user-friendly, nello stile del noto software Derive[1] sviluppato da Texas Instruments. L'ambiente di lavoro dovrà inoltre dare la possibilità all'utente di inserire espressioni matematiche sia inserendole direttamente da tastiera, sia aiutandosi con appositi pulsanti.
- Interfaccia grafica totalmente indipendente dal framework, ovvero dovrà essere possibile cambiare la tecnologia della GUI senza intaccare minimamente parti interne del software.
- Visualizzazione delle espressioni matematiche in formato LaTeX.
- Salvataggio e caricamento di documenti in un apposito formato.
- Visualizzazione di grafici di funzioni in una variabile, in cui sarà possibile zoomare sia in tutto il grafico, sia nei singoli assi.

1.2 Problema

I problemi che l'applicazione si pone di risolvere sono innumerevoli e molto diversificati tra loro, per questo essi sono divisi in due macro-categorie, che rappresentano anche le due parti principali del software.

Framework

Un CAS è un sistema intrinsecamente complesso, e la sua progettazione richiede un'attenta valutazione. Una delle maggiori difficoltà è quella di strutturare il sistema in modo che sia facilmente scalabile, cioè che sia possibile aggiungere funzionalità senza applicare modifiche drastiche al codice.

Altra problematica, è quella di fare in modo che sulla stessa espressione matematica si possano eseguire più operazioni, anche molto diverse tra loro. Ad esempio, un'espressione matematica può essere semplificata algebricamente, oppure approssimata in modo totalmente numerico, restituendo un numero reale (senza coinvolgere alcun tipo di calcolo simbolico). E' quindi necessario scegliere le strutture dati adatte, e progettare il sistema in modo opportuno, tenendo a mente che un eventuale difetto di progettazione può compromettere la scalabilità del sistema, rendendo difficile o addirittura impossibile aggiungere certe funzionalità.

[1] `http://it.wikipedia.org/wiki/Derive`

La progettazione sarà realizzata in modo da semplificare l'implementazione delle features citate nel capitolo precedente, permettendone anche l'aggiunta di altre in futuro. Tuttavia, dato che la matematica è una materia estremamente vasta, per trattare ulteriori campi di essa (al livello dei CAS commerciali) sarebbe necessaria un'analisi più approfondita, che richiederebbe molto più tempo del monte ore previsto.

Una volta completata la progettazione, i problemi saranno di natura implementativa. La funzione più difficile da implementare è la semplificazione, poiché richiede di specificare numerose regole algebriche in base al tipo di dato che si sta manipolando. Ad esempio, la funzione seno avrà le proprie regole di semplificazione (come $\sin(0) = 0$), così come la moltiplicazione avrà le proprie (ad esempio 0*x = 0). Lo stesso si applica alle altre funzionalità: dev'essere possibile specificare, per ognuna di esse, una serie di regole che varia in base al tipo di dato. Per la derivata, è necessario specificare come derivare una funzione composta, qual è la derivata di ogni funzione, come derivare gli operatori, ed il tutto dev'essere gestito in maniera modulare.

Interfaccia grafica e controller

L'interfaccia grafica dovrà essere sconnessa dal resto dell'applicazione, e poter essere cambiata all'occorrenza, senza che ciò implichi di dover mettere mano a parti interne del software. Questo problema, che è uno dei principali su cui volge tutta l'applicazione, sarà materia di studio nelle fasi embrionali della costruzione del software e la sua implementazione avrà ripercussioni sul design generale del sistema.

Un altro aspetto fondamentale sarà quello di generalizzare l'interazione tra il framework e l'utente. Il core dell'applicazione ha bisogno, per svolgere alcune operazioni, di dati immessi direttamente dall'utilizzatore finale, come ad esempio in quale variabile derivare un'espressione o gli estremi per il calcolo di un integrale definito. Questo scambio di informazioni tra utente e software dovrà essere implementato nella maniera più generica possibile, in modo da poter aggiungere facilmente nuove funzioni all'applicazione. Proprio per questo motivo le cosidette *framework operations*, ovvero quelle funzionalità che richiedono l'intervento del framework dovranno essere il più possibile generalizzate, in modo da poterne aggiungere altre senza intaccare la struttura del programma.

Le espressioni matematiche dovranno essere graficate in modo comprensibile e standardizzato e, naturalmente, dovrà esserci un meccanismo per selezionare le formule inserite sulle quali si vogliono eseguire le operazioni messe a disposizione dal framework. Le formule matematiche verranno gra-

ficate in LaTeX, il che richiederà uno studio a parte sulla soluzione migliore da adottare.

Dovrà inoltre essere possibile graficare funzioni in una variabile tramite un grafico cartesiano. La difficoltà primaria di questa parte dell'applicazione sarà quella di implementare lo zoom sul grafico e anche sui singoli assi, avvalendosi di strumenti matematici come le trasformazioni lineari.

L'applicazione dovrà inoltre essere veloce e supportare il *multithreading*, in modo che il framework matematico sia in esecuzione su un thread diverso rispetto all'interfaccia grafica. In questo modo l'applicazione sarà sempre reattiva anche a fronte delle operazioni più lunghe e dispendiose. Sarà inoltre possibile, da parte dell'utente, fermare il thread su cui il framework sta eseguendo un'operazione quando essa diventa troppo dispendiosa in termini di tempo. Quest'ultimo punto richiederà uno studio specifico sul design dell'applicazione e sulla comunicazione tra thread diversi che non potrà essere effettuato nel molte ore previsto. Tuttavia la parte riguardante il solo framework matematico sarà svolta, lasciando a lavori futuri il compito di implementare completamente la tecnologia multithreading nell'applicazione.

Capitolo 2

Design

Come già precedentemente visto il *framework matematico* può essere usato anche come libreria a sè stante, e quindi deve essere il più indipendente possibile dal resto dell'applicazione. Anche *l'interfaccia grafica* deve essere indipendente dal software, poiché uno degli obiettivi è proprio quello di poter cambiare l'interfaccia utente senza intaccare gli aspetti interni dell'applicazione.

Il modo migliore di soddisfare questi requisiti è di costruire l'applicazione attorno al pattern MVC (model-view-controller), in cui ogni componente è del tutto sconnesso dagli altri, in modo da potere cambiare alcune parti del software senza effetti collaterali.

2.1 Architettura

Come accennato precedentemente l'applicazione è principalmente costruita attorno al pattern MVC, il quale permette un totale disaccoppiamento tra il core e la parte grafica dell'applicazione. In particolare sia il framework che l'interfaccia grafica sono visti dall'esterno solo attraverso un'interfaccia, che mette a disposizione soltanto i metodi necessari a risolvere i requisiti principali dell'applicazione, come mostrato in Figura 2.1.

In questo modo tutta la parte interna dell'applicazione è nascosta al Controller e quindi qualsiasi cambiamento interno di qualunque di queste parti non interferirà minimamente con le altre. In questo modo è possibile ad esempio passare da una interfaccia grafica a una interfaccia console line senza dover modificare parti dell'applicazione che non facciano parte della parte di interazione con l'utente. Basterà perciò far si che la nuova interfaccia grafica implementi l'interfaccia `UserInterface`, il comportamento interno delle classi potrà essere qualunque.

Inoltre anche modificare internamente il framework matematico non ha ripercussioni sugli altri componenti del software, e ciò è ancora più importante che per l'interfaccia grafica dato che esso è stato costruito per poter essere utilizzato anche come libreria a se stante.

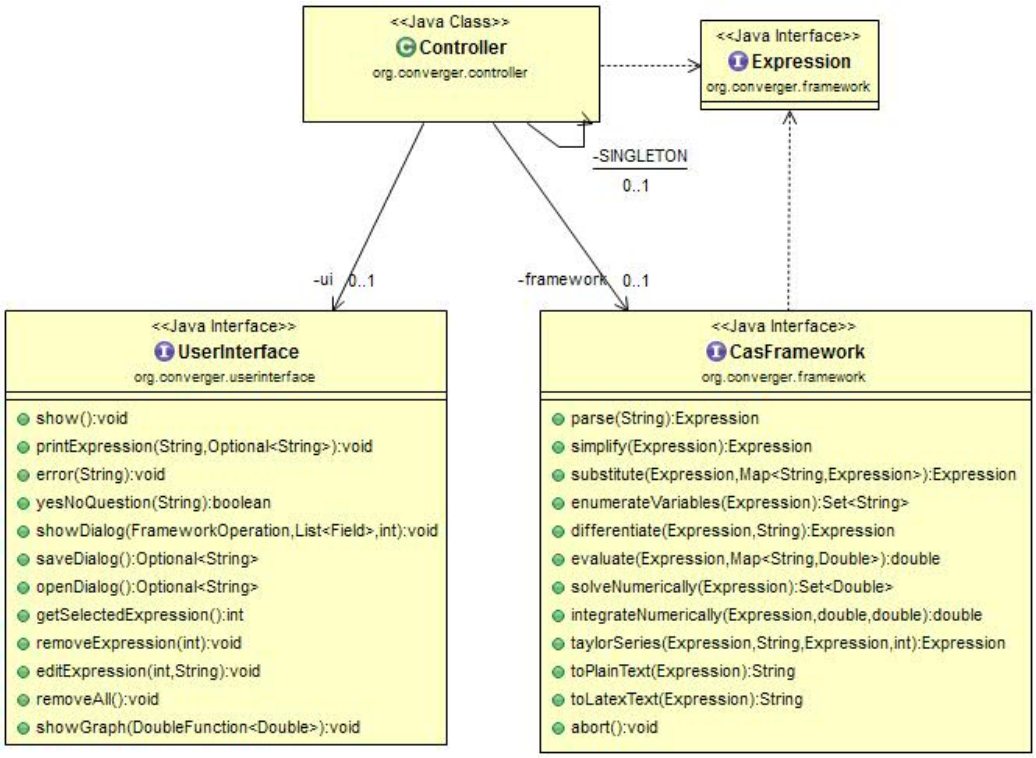

Figura 2.1: Schema UML architetturale. Si noti sopratutto l'uso del pattern MVC e il fatto che le 3 parti principali dell'applicazione (model, view e controller) sono sconnesse tra loro. Il controller vede soltanto le funzionalità messe a disposizione da view e model, ma non può vedere come internamente queste funzionalità sono state implementate. In questo modo è molto semplice sostituire in blocco sia la view che il model, basta che essi implementino rispettivamente le interfacce `UserInterface` e `CasFramework`. E' bene anche notare che il Controller lavora con una generica interfaccia `Expression` (che modella un'espressione matematica), la cui rappresentazione interna è nascosta all'utilizzatore.

2.2 Design dettagliato

Framework

Le operazioni del framework accessibili dall'esterno sono disponibili nell'interfaccia `CasFramework`, e lavorano su degli oggetti di tipo `Expression`. Quest'ultima, è l'interfaccia su cui si basano tutte le espressioni matematiche, che si possono specializzare nel seguente modo:

- `Constant`: rappresenta una costante numerica, e contiene un singolo numero naturale
- `Variable`: rappresenta una variabile
- `BinaryOperation`: rappresenta un'operazione binaria (potenza e divisione), cioè coinvolge due operandi qualsiasi di tipo `Expression`
- `NAryOperation`: rappresenta un'operazione n-aria, composta da un numero qualsiasi di operandi. Viene usata per gli operatori che godono della proprietà associativa e distributiva (addizione e moltiplicazione)
- `FunctionOperation`: rappresenta l'esecuzione di una funzione f(x)
- `Equation`: rappresenta un'equazione contenente due membri, uniti da una condizione di uguaglianza

Per rappresentare espressioni complesse, questi oggetti vengono collegati assieme, creando una struttura ad albero. Tale struttura, che assume il nome di *Syntax Tree*, realizza il pattern *Composite*. Per rendere il sistema scalabile, è necessario disaccoppiare completamente la parte rappresentativa dell'espressione dalla parte algoritmica. Per questo motivo, le classi che implementano `Expression` sono dei semplici contenitori, e per attraversare l'albero si utilizza il pattern *Visitor*. Ogni funzionalità del framework (semplificazione, derivazione, approssimazione, ecc...) implementa l'interfaccia `Expression.Visitor` ed accede all'albero tramite essa (Figura 2.2). Tutti i visitors si trovano nel package `org.converger.framework.visitors`, ed implementano quasi tutte le funzionalità principali del framework.

Alcune di queste classi necessitano di più informazioni: nel caso delle operazioni binarie ed n-arie sarà necessario conoscere qual è l'operatore da eseguire (somma, prodotto, ...), e nel caso dell'esecuzione di una funzione, sarà necessario sapere la funzione coinvolta (seno, coseno, ...). Per questo motivo, sono presenti delle enumerazioni: `BinaryOperator`, `NAryOperator`, `Function`, che contengono la lista delle possibili azioni nel loro dominio. Per

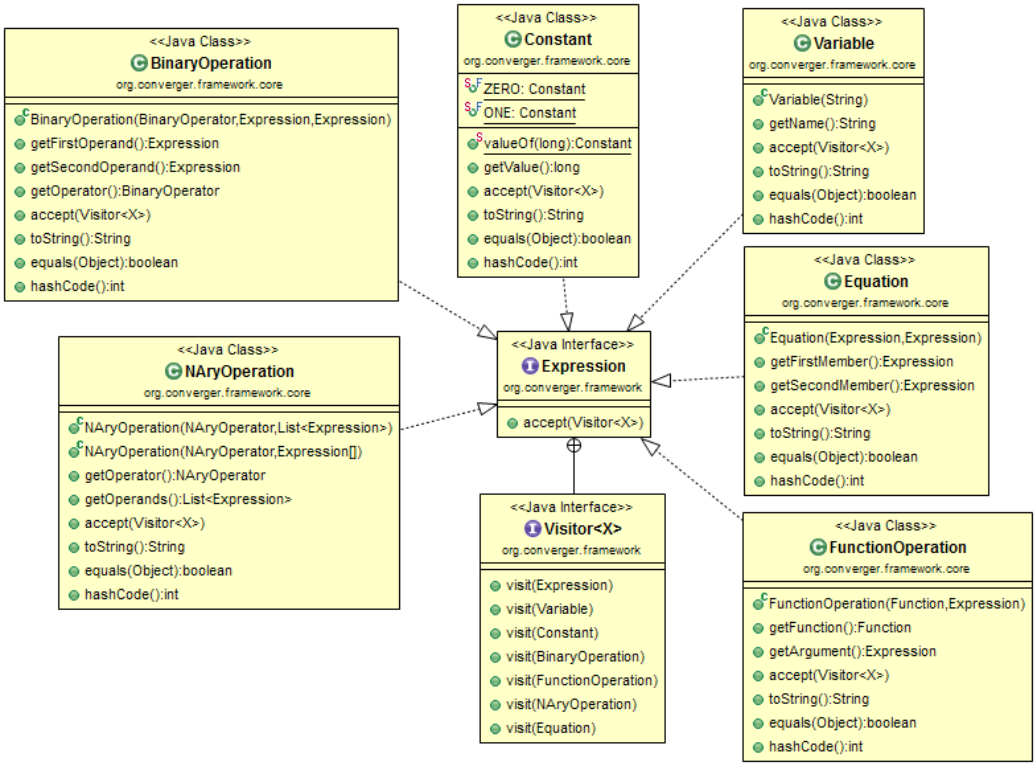

Figura 2.2: Schema UML delle classi che implementano l'interfaccia `Expression`. Notare l'interfaccia `Expression.Visitor`: ogni classe che la implementa deve specificare come visitare nello specifico ogni tipo di nodo contenuto nell'albero.

permettere la massima elasticità del sistema, anch'esse hanno la propria interfaccia `Visitor`. Questo permette di specificare come comportarsi ad ogni singola operazione: ad esempio, nel caso della derivazione, si può definire la derivata di ogni funzione, come si può definire la derivata del prodotto, della somma, della divisione, della potenza, e così via. Inoltre, si evitano costrutti *switch* di notevoli dimensioni, il design è più pulito, e le prestazioni del sistema sono ottimali, dato che il flusso di chiamate ai metodi è diretto. Una funzionalità non deve necessariamente implementare tutti i visitor per accedere all'albero, ma solo quelli di cui ha bisogno. Ad esempio, nella sostituzione delle variabili, viene implementato solamente `Expression.Visitor`, che visita ricorsivamente l'albero senza fare distinzioni tra i vari operatori.

Per l'implementazione di alcune specifiche funzionalità sono stati usati altri pattern di programmazione. Ad esempio, per generalizzare la stampa tra il formato plain text ed il formato LaTeX sono stati scritti i due visitor `BasicPrinter` e `LatexPrinter`, che derivano entrambi da `AbstractPrinter`

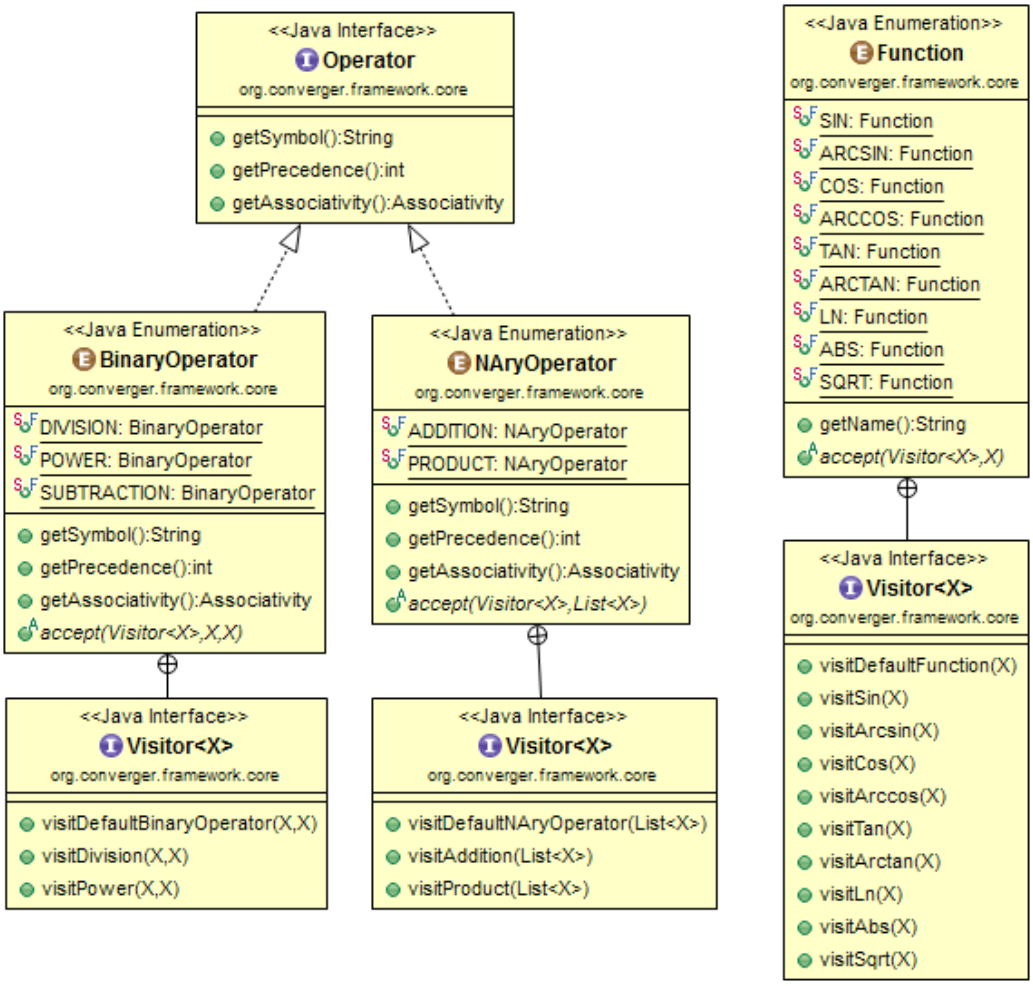

Figura 2.3: Schema UML delle enumerazioni, con le rispettive interfacce `Visitor`. Qualora si volessero specializzare solo certi operatori/funzioni, è possibile definire anche un comportamento di default. I metodi nell'interfaccia `Operator` sono di interesse del parser, per confrontare la precedenza tra gli operatori e la loro associatività.

e specializzano le parti differenti mediante l'uso del pattern *Template Method*. Per generare facilmente alcune espressioni tramite codice si utilizza una *Static Factory*. Nel calcolo della derivata, per integrare le interfacce `Visitor` tra loro si utilizza un sistema assimilabile al pattern *Strategy*.

Come nota, anche se la funzionalità di multithreading non è stata realizzata, il framework è stato predisposto a tale scopo. La classe `CasManager`, che è un *Singleton*, può istanziare oggetti di tipo `CasFramework`. Inoltre, quest'ultimo fornisce le primitive thread-safe per permettere di interrompere

il calcolo in corso, chiamando il metodo `CasFramework.abort` da un altro thread. Ciò si rende necessario per permettere l'interruzione cooperativa, in quanto la primitiva `Thread.stop` di Java è stata deprecata.

Aspetti implementativi del framework più specifici saranno trattati nell'apposito capitolo.

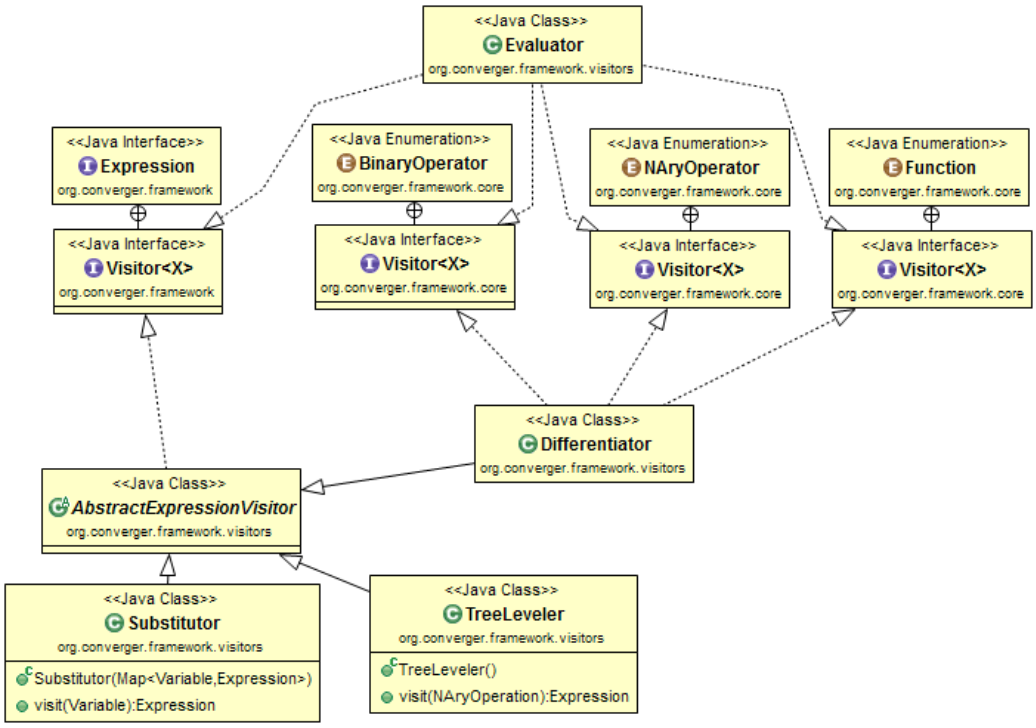

Figura 2.4: Schema UML semplificato di alcuni visitors. `AbstractExpressionVisitor` non è altro che un visitor astratto che prende in ingresso un'espressione, la attraversa ricorsivamente, e restituisce una nuova espressione manipolata. `Evaluator` è la classe che approssima il risultato di un'espressione, restituendo un numero reale: implementa tutti i visitor, dato che specifica un comportamento per ognuno di essi. Lo stesso si applica per `Differentiator`, che calcola la derivata di una funzione. `Substitutor` esegue la sostituzione di una variabile con una sottoespressione, ed infatti contiene solo l'override del visit di una variabile. `TreeLeveler` è uno specifico semplificatore che compatta l'albero, comprimendo gli operatori n-ari ridondanti (ad esempio (x+y)+(z+w) diventa x+y+z+w). I visitor sono numerosi, e non sono stati inclusi tutti in questo diagramma per questioni di chiarezza, inoltre, i metodi di alcuni visitor qui presenti sono stati nascosti.

Interfaccia grafica e controller

Come già ripetuto più volte precedentemente, uno dei target nella costruzione dell'interfaccia grafica è la sua facile sostituibilità con altre tecnologie (Swing, JavaFX) o con altre implementazioni (GUI, CLI). Come si è constatato nel capitolo precedente il pattern MVC offre una soluzione pulita al problema: l'interfaccia grafica può essere sostituita anche in blocco senza dover toccare aspetti interni all'applicazione, basterà che la nuova implementazione aderisca all'interfaccia `UserInterface`.

Tuttavia l'implementazione interna del front-end grafico non è una cosa banale, poiché esso deve offrire molte funzionalità, anche molto diverse tra loro. Per questo motivo esso è stato implementato seguendo il design pattern *Facade*, in cui una classe sola implementa l'interfacca `UserInterface`, implementandone anche tutti i suoi metodi. Questa classe (la classe `GUI` in `org.converger.userinterface.gui`) non è altro che una helper class, che semplifica l'accesso a molte funzionalità implementate da una gerarchia di classi a lei sottostanti, come mostrato nella Figura 2.5. Seguendo questo design pattern non solo è possibile costruire classi più snelle, ma anche dividere le funzionalità in modo più consistente (ad esempio la classe `BodyImpl` che ha il compito di gestire la visualizzazione e la selezione delle espressioni non ha niente a che fare con la classe `Footer` che si occupa dell'inserimento di formule matematiche). In questo modo l'interfaccia grafica è scomposta in un insieme di componenti distinte che vengono "montate" al momento della creazione della GUI. Così facendo risulta immediato sostituire un componente con una nuova implementazione (ad esempio un nuovo visualizzatore di espressioni), poichè basterà creare una nuova implementazione della classe, senza toccare gli altri componenti dell'interfaccia grafica.

Un altro problema evidenziato in fase di analisi è stato quello di generalizzare le operazioni che richiedono l'intervento del framework matematico (framework operations), come ad esempio la derivazione o la sostituzione di variabile. Queste operazioni, che sono il cuore dell'applicazione, possono richiedere anche interazione con l'utente, come ad esempio chiedere in che punto sviluppare una serie di Taylor.

Le *framework operations* sono state implementate attraverso un enum (`org.converger.controller.FrameworkOperation`), che ha al suo interno due metodi astratti: *requestFields* e *execute*. Queste due funzioni sono implementate dentro ogni voce dell'enum, sfruttando il fatto che ogni elemento di un enumeratore in java è considerato una classe. Quando l'utente richiede di eseguire una *framework operation* su una espressione viene quindi chiamato il metodo *executeFrameworkOperation* passandogli come parametro l'operazione che si vuole eseguire. Esso chiamerà i due metodi astratti

della *FrameworkOperation*, il primo per farsi restituire una lista di `Field`, con cui verrà costruita dinamicamente la dialog di comunicazione con l'utente. Se la lista di `Field` è vuota significa che non è richiesta nessuna interazione con l'utente e viene direttamente eseguita la *FrameworkOperation* tramite il metodo astratto *execute*. Se invece si richiede comunicazione tra utente e framework viene dinamicamente costruita una dialog e alla pressione del tasto OK viene eseguita la *FrameworkOperation* con i dati inseriti. Si è quindi usato il pattern *Template method* per generalizzare un'operazione. In questo modo è molto facile inserirne altre framework operation, basterà aggiungere una voce all'enum `FrameworkOperation` e implementare i due metodi astratti *requestFields* e *execute* come mostrato in Figura 2.6.

Come si evince dall'implementazione delle *framework operation* il controller dell'applicazione è strutturato in modo da funzionare alla maniera di un comune server web. L'utente tramite la pressione dei tasti comunica ad esso le operazioni da svolgere, il controller risponderà a sua volta, tramite i metodi messi a disposizione dalla GUI, richiedendo informazioni aggiuntive. Quando questi dati sono stati ottenuti viene inviata una nuova richiesta al controller con le informazioni immesse dall'utente, il controller userà le funzionalità del framework (impiegato quasi come fosse soltanto una libreria) per ottenere una risposta che sarà a sua volta mostrata all'utente modificando la GUI tramite i metodi messi a disposizione dall'interfaccia `UserInterface`. Il controller si comporta quindi come uno strato intermedio tra interfaccia utente e core dell'applicazione.

Per questo motivo il controller deve essere univoco all'interno dell'applicazione. Per far si che ciò accada è stato usato il pattern *Singleton*, come si può notare nella Figura 2.1. In questo modo il controller è istanziato una sola volta. Ciò è necessario anche perché esso al suo interno contiene lo stato corrente dell'ambiente di lavoro, con le espressioni matematiche immesse dall'utente o ottenute a seguito di operazioni del framework.

Un altro aspetto importante dell'applicazione è la gestione degli eventi della tastiera. E' possibile ad esempio, selezionando una espressione matematica, copiare il suo testo con la pressione dei tasti `CTRL + C`, e poi incollarlo sia dentro che fuori dall'applicazione. Al momento questa è l'unico evento della tastiera che viene osservato, ma l'architettura implementata, attraverso il pattern *Observer*, consentirà in futuro di aggiungere diverse shortcut da tastiera velocemente.

Come mostrato in Figura 2.7 il pattern è stato implementato nella maniera classica, con una `Source`, che può avere più osservatori, e un `Observer` a cui vengono notificati gli eventi prodotti dalla sorgente. I diversi tipi di eventi che possono essere notificati sono implementati tramite un enum (`KeyboardEvent`).

Un altro importante aspetto di design è la gestione del grafico di una funzione. Esso è gestito principalmente attraverso la classe `PlotWindow` nel package `org.converger.plot`. Essa rappresenta il componente grafico in cui sarà disegnato il grafico cartesiano. Questa classe nel costruttore prende come parametro una `DoubleFunction (java.util.function)` che viene passata attraverso una lambda expression, essendo `DoubleFunction` un'interfaccia funzionale. La funzione passata come parametro rappresenta la funzione matematica da graficare, che verrà valutata tramite il metodo *apply* in diversi punto per produrre il grafico. Questa è una sorta di implementazione del pattern *Strategy*, utilizzando però ciò che le librerie Java mettono a disposizione nativamente.

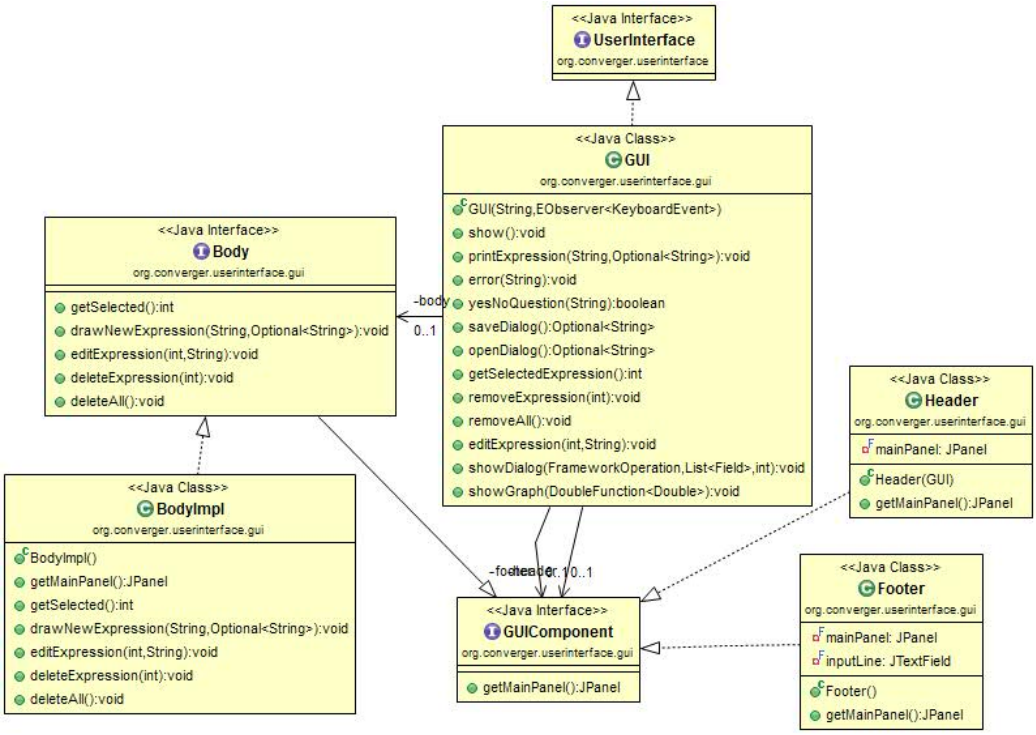

Figura 2.5: Schema UML della GUI e delle classi sottostanti, in cui si evidenzia il pattern *Facade*. La classe `GUI` ha come metodi pubblici soltanto quelli derivati dall'implementazione dell'interfaccia `UserInterface`. Questi metodi non fanno altro che eseguire i metodi implementati nelle classi sottostanti. In questo modo la vera implementazione delle funzioni della `UserInterface` è lasciata alle 3 classi `Header`, `BodyImpl` e `Footer`, la classe `GUI` funge solo da raccordo tra queste. Inoltre la classe `GUI` è anche responsabile della gestione delle dialog, ovvero della comunicazione tra framework e utente. Le dialog sono implementate altrove, mentre la classe `GUI` è responsabile solo della loro visualizzazione nell'interfaccia grafica. In questo modo è addirittura possibile cambiare solo alcuni componenti del frontend grafico senza dover toccare nient'altro: ad esempio se si volesse fare una nuova implementazione del Body potrebbe essere fatta senza toccare Footer e Header.

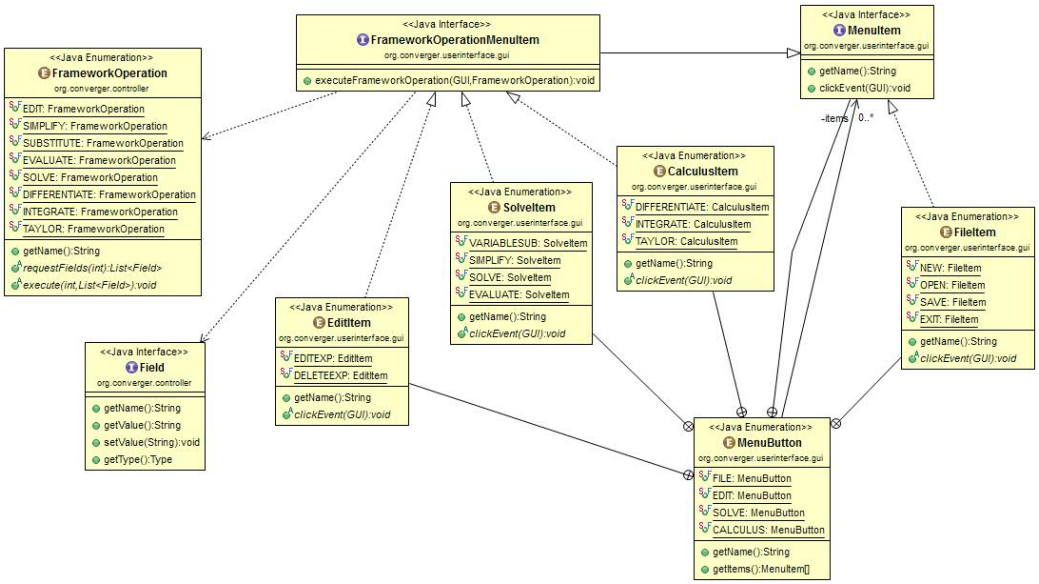

Figura 2.6: Schema UML che mostra le varie interazioni tra classi per eseguire una framework operation. Quando l'utente preme un pulsante del menù per eseguire una framework operation su una espressione viene eseguito il metodo *executeFrameworkOperation* nell'interfaccia `FrameworkOperationMenuItem`, che ha un'implementazione di default. Esso chiama i due metodi astratti della framework operation (*requestFields* e *execute*), il primo resituisce una lista di `Field`, oggetti usati per rappresentare un valore richiesto dal framework all'utente, il secondo esegue materialmente l'operazione servendosi degli eventuali valori immessi dall'utente. Se la lista di `Field` è vuota significa che non è richiesta interazione con l'utente e l'operazione viene subito eseguita. Altrimenti si apre una `Dialog` (`org.converger.userinterface.gui.dialog`) che viene costruita dinamicamente in base ai `Field` restituiti. In questa `Dialog` l'utente inserisce i valori richiesti e alla pressione del tasto OK viene eseguito il metodo *execute* della framework operation richiesta. E' stato quindi usato una sorta di *Template method*, anche se non nella sua forma standard per poter generalizzare il più possibile il concetto di *framework operation*.

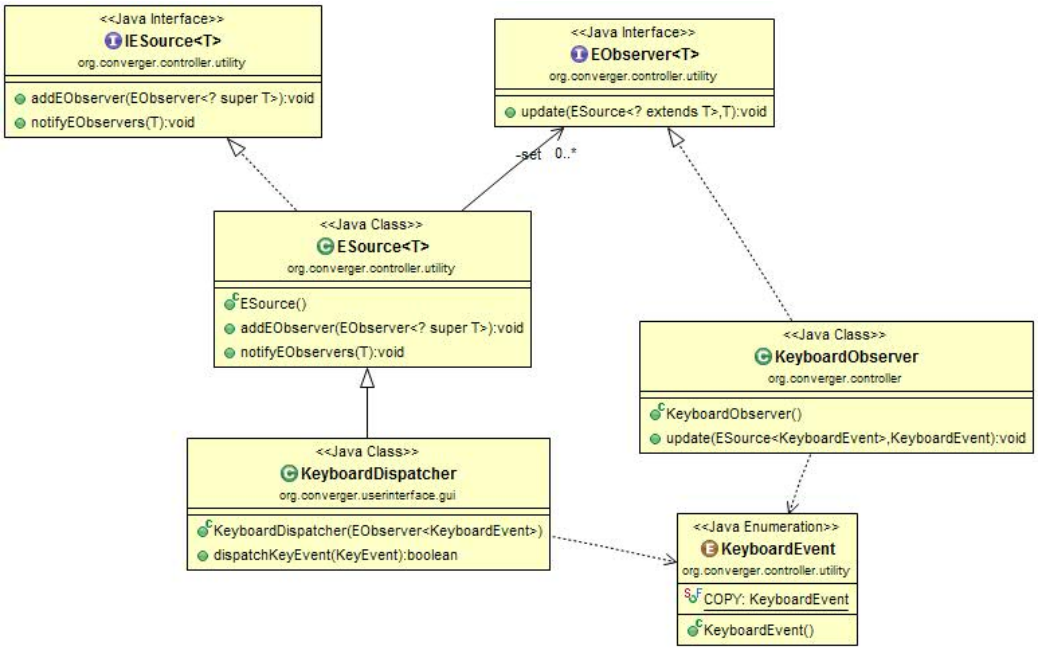

Figura 2.7: Schema UML del pattern *Observer* utilizzato per gestire le shortcut da tastiera. La classe `KeyboardDispatcher` si occupa di intercettare gli eventi della tastiera. Quando questi combaciano con gli eventi da osservare viene chiamato il metodo *notifyEObserver* che notifica l'evento all'osservatore, in questo caso rappresentato dalla classe `KeyboardObserver`. Questa tramite il metodo *update* esegue la corretta sequeza di istruzioni in base al tipo di evento notificato, contenuto nell'enum `KeyboardEvent`.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per quel che riguarda il framework, sono stati inseriti diversi test automatizzati per verificarne le funzionalità. Tali test, implementati con JUnit, si trovano nel package `org.converger.framework.test`. Essi si preoccupano di verificare principalmente le funzionalità legate al parsing delle espressioni e all'approssimazione numerica. Ad esempio, per il risolutore di equazioni vengono testati dei polinomi generati in modo random, e per gli integrali si confrontano i risultati prodotti con valori noti.

Per quel che riguarda l'interfaccia grafica, non essendo ovviamente possibile testarla in modo automatico, sono stati eseguiti diversi test manuali per verificarne il corretto funzionamento, anche nei vari casi d'errore. Inoltre, l'applicazione è stata testata su diversi sistemi operativi, facendo uso di macchine virtuali. Sono stati eseguiti test in ambiente Windows e Linux (nello specifico, le distribuzioni Ubuntu e Fedora).

3.2 Divisione dei compiti e metodologia di lavoro

Come citato precedentemente, l'applicazione è divisa in due blocchi principali: il framework matematico ed il frontend grafico (composto da GUI e Controller). Il lavoro è stato suddiviso come segue:

- Dario Pavllo: progettazione ed implementazione del framework matematico
- Gabriele Graffieti: progettazione ed implementazione della GUI e del Controller

- Parti in comune: progettazione architetturale del sistema, progettazione del layout dell'applicazione, funzionalità accessorie del framework (nella fattispecie, risoluzione di equazioni, integrali definiti, serie di Taylor), implementazione della parte che si occupa di disegnare il grafico di una funzione

Per coordinare il lavoro è stato creato un repository Mercurial su Bitbucket, con uno schema del tipo Shared Repository, adatto ai software con pochi mantenitori. Il progetto è stato successivamente svolto in maniera cooperativa, cercando di disaccoppiare al massimo il lavoro tra le due parti, anche per via della difficoltà di incontrarsi (per diversi impegni). Ognuno ha lavorato sulla propria parte in parallelo ed autonomia, eseguendo i push e merge all'occorrenza (cioè quando le features erano arrivate ad una certa stabilità). Le parti che hanno richiesto un incontro hanno riguardato principalmente il design di alcuni aspetti architetturali dell'applicazione, ed il testing di alcune funzionalità in modo da concordarsi sul risultato sperato.

Ognuna delle due parti ha esposto le proprie interfacce senza farle dipendere dagli aspetti implementativi interni, e di conseguenza è stato relativamente semplice integrare il lavoro tra le due parti. Dato che il framework è stato concepito come libreria, i suoi aspetti visibili dall'esterno sono stati semplificati al massimo per permetterne un utilizzo intuitivo. Grazie al grande disaccoppiamento delle funzionalità, non ci sono stati particolari problemi di coordinazione in fase implementativa.

3.3 Note di sviluppo

Framework

Il framework, per svolgere le proprie funzioni, fa uso di numerosi algoritmi e sistemi complessi. Il sistema è suddiviso nelle seguenti parti:

Core (`org.converger.framework.core`)

E' il cuore del sistema, ed include i contenitori che rappresentano le varie parti dell'espressione, le enumerazioni, le costanti matematiche, e varie utilità. Non include alcun algoritmo e alcuna regola algebrica in senso stretto.

Parser (`org.converger.framework.parser`)

Contiene le interfacce e le classi che si occupano di eseguire il parsing delle espressioni testuali. E' suddiviso in 3 parti:

- Tokenizer: divide la stringa in token, cioè separa le variabili, le costanti, gli operatori, ed ogni eventuale elemento facente parte della notazione matematica.

- Parser: prende in input i token prodotti precedentemente, e li interpreta come se rappresentassero un'espressione in notazione infissa (cioè la notazione comunemente usata). Un parser deve implementare l'interfaccia `Parser`, e nel nostro caso, l'implementazione è stata realizzata tramite l'algoritmo Shunting-Yard di Dijkstra[1], che viene usato per produrre un output in notazione polacca inversa (RPN, detta anche notazione postfissa). L'algoritmo è stato modificato per supportare la moltiplicazione implicita ed alcuni operatori unari (di segno, nello specifico).

- TreeBuilder: interpreta l'output precedente in RPN, e ne costruisce un albero sintattico (Syntax Tree), cioè la struttura dati finale, composta da `Expression`.

Visitors (`org.converger.framework.visitors`)

Il package contiene tutti i visitors, cioè le classi che implementano il pattern Visitor e contengono le funzionalità base del framework. Ad esempio, per la semplificazione (la feature più complessa), vengono eseguiti in serie i seguenti visitors: `TreeLeveler`, `AlgebraicSimplifier`, `RationalSimplifier`, `Collector`, `ConstantFolder`, `TreeSorter`. Il primo, ad esempio, comprime l'albero rimuovendo le operazioni commutative e associative ridondanti. Ad esempio, (a+b)+(c+d) viene semplificato in a+b+c+d. Ciò si rende sempre necessario, anche perché il parser restituisce sempre operazioni binarie, come nell'esempio appena citato. Le altre classi applicano alcune regole algebriche appartenenti al proprio dominio (guardare i sorgenti, che sono corredati di documentazione, per maggiori informazioni). Dato che un solo passaggio nella maggior parte dei casi non è sufficiente, il processo di semplificazione viene ripetuto finché l'espressione non smette di variare: a quel punto il processo è completato.

Per la derivazione, ed altre funzionalità analoghe, si segue un approccio di backtracking. Prima si visitano le parti più in profondita nell'albero, e successivamente, per induzione, si innestano al nodo in oggetto. Nel caso della derivazione, ad esempio, per derivare la composizione di una funzione $f(g(x))$, si deriva la funzione esterna senza preoccuparsi del suo contenuto (come si farebbe con una comunissima tabella di derivazione), richiamando l'apposita

[1] `http://en.wikipedia.org/wiki/Shunting-yard_algorithm`

implementazione del visitor delle funzioni. Successivamente, dopo aver ottenuto $f'(g(x))$ si continua a visitare ricorsivamente l'albero ottenendo $g'(x)$ e lo si moltiplica all'espressione precedente, ottenendo $f'(g(x)) \cdot g'(x)$. Il metodo `Function.Visitor.visitSin(arg)` restituirà semplicemente $cos(arg)$, e non $cos(arg) \cdot arg'$. Sarà il metodo `Expression.Visitor.visit(Function)` a restituire $visitSin(arg) \cdot visit(arg)$.

Algoritmi (`org.converger.framework.algorithms`)

Il package contiene funzionalità ad alto livello, che eseguono operazioni complesse avvalendosi delle features basilari del framework. Contiene:

- `NumericalIntegrator`: un integratore numerico. Calcola gli integrali definiti di una funzione avvalendosi del *metodo dei trapezi*[2].
- `NumericalSolver`: approssima il risultato di un'equazione in una variabile utilizzando il *metodo delle tangenti di Newton*[3], un algoritmo per trovare le radici di una funzione. E' stato scelto perché, grazie alla conoscenza analitica della derivata (che il framework è in grado di calcolare), è possibile ottenere un'enorme rapidità di convergenza in poche iterazioni. Una funzione in ingresso del tipo $f(x) = g(x)$ viene convertita in $f(x) - g(x) = 0$, e successivamente viene eseguito l'algoritmo. Quest'ultimo, inoltre, è stato modificato al fine di trovare radici multiple, trasformando la funzione in modo da eliminare gli zeri già trovati. Sono presenti anche alcuni parametri per gestire l'errore minimo/massimo, e meccanismi per rilevare la divergenza, assenti nell'algoritmo originale.
- `TaylorSeries`: calcola le serie di Taylor di una funzione in modo algebrico. La funzionalità è molto semplice, dato che si limita ad applicare la definizione, ed è un esempio di funzionalità composte che si possono ottenere mettendo assieme funzionalità base (derivazione, semplificazione, sostituzione).

Note generali

Durante lo sviluppo del framework sono state prese delle decisioni di natura interna, relative alla rappresentazione dei dati. Una di queste è la presenza degli operatori n-ari, oltre ai semplici operatori binari. In teoria sarebbe stato sufficiente modellare l'addizione ed il prodotto come operatori binari, ma sono

[2] `http://en.wikipedia.org/wiki/Trapezoidal_rule`
[3] `http://mathworld.wolfram.com/NewtonsMethod.html`

stati implementati come n-ari perché ciò riduce l'ambiguità nella rappresentazione delle espressioni, aumenta le prestazioni, e permette di semplificare lo sviluppo di molte funzionalità, tra cui la derivazione e la semplificazione. Ad esempio, per implementare la legge dell'annullamento del prodotto (qualsiasi termine moltiplicato per 0 è uguale a 0) è sufficiente controllare se almeno un elemento nella lista dei fattori è zero. La derivata del prodotto è stata generalizzata ad N fattori, producendo un risultato più compatto.

Sempre per motivi analoghi, si è scelto di eliminare l'operatore sottrazione dal framework. Esso è presente nell'enumerazione solo ai fini del parsing, e successivamente, ogni sottoespressione di segno negativo viene trasformata in un addendo del tipo $(-1) \cdot x$. Ciò permette di preservare la commutatività dell'operatore addizione, permette di riordinare l'albero, e riduce ulteriormente le ambiguità nella rappresentazione dello stesso. Come contro, in fase di stampa è necessario reintegrarlo per produrre espressioni leggibili.

Alcune di queste idee sono state tratte informandosi sull'implementazione di CAS già esistenti.[4]

Interfaccia utente

Il frontend grafico è stato sviluppato come graphical user interface (GUI), e per questo tipo di interfaccia utente Java mette a disposizione diverse librerie. Per questo progetto è stato utilizzato Swing (`javax.swing`). Come supporto alla costruzione dei diversi componenti dell'interfaccia grafica è stato utilizzato il plugin per Eclipse WindowBuilder[5]. Naturalmente il codice da lui prodotto è stato revisionato e corretto dove necessario.

Per quel che riguarda il rendering delle espressioni matematiche in LaTeX è stata utilizzata una libreria esterna: JLaTeXMath[6]. La libreria è stata utilizzata nella classe `BodyImpl` nel package `org.converger.userinterface.gui` prendendo spunto dagli esempi presenti sul sito. La libreria è stata scelta sia perchè usata in famosi software come Scilab[7] o Geogebra[8] sia perchè compatibile con la licenza GPLv3 [9].

Nell'interfaccia grafica è stato utilizzato anche un sistema di caching per le immagini renderizzate di espressioni matematiche. Questo perchè il pannello in cui vengono mostrate le formule viene aggiornato di frequente (ad esempio quando una formula viene modificata o eliminata), e renderizzare

[4] `http://www.math.wpi.edu/IQP/BVCalcHist/calc5.html`
[5] `https://eclipse.org/windowbuilder`
[6] `http://forge.scilab.org/index.php/p/jlatexmath/`
[7] `http://www.scilab.org/`
[8] `http://www.geogebra.org/`
[9] `http://www.gnu.org/copyleft/gpl.html`

ogni volta tutte le espressioni inserite poteva comportare problemi di performance. Questo sistema di caching è stato implementato tramite una lista di `JPanel` i quali all'interno contengono l'immagine renderizzata in LaTeX. Ogni volta che il pannello principale viene aggiornato non si fa altro che eliminare le immagini su di esso e riaggiungerle di seguito scorrendo la lista. In questo modo le performance sono più che buone e l'aggiornamento del pannello richiede pochi millisecondi.

Per il grafico si è utilizzato un mini schema MVC. La view, in questo caso, è la plot window, che disegna il grafico vero e proprio. A livello implementativo, si utilizza la classe `AffineTransform` di AWT per eseguire le trasformazioni lineari, e si disegna utilizzando una `Path2D`, cioè segmento per segmento. Nonostante sia meno accurato di un disegno punto per punto, è più semplice da gestire, più performante, e permette di usufruire dell'antialiasing. Il Controller gestisce il comportamento dei pulsanti di zoom e manipola i parametri della trasformazione lineare, cambiando la scala, mentre la parte di Model non è altro che un'espressione lambda passata dal Controller principale, che, data la variabile indipendente, approssima il valore reale della variabile dipendente. A sua volta, quest'espressione sarà basata sulla funzionalità `evaluate` del framework.

Capitolo 4

Commenti finali

In quest'ultimo capitolo si tirano le somme del lavoro svolto e si delineano eventuali sviluppi futuri.

4.1 Conclusioni e lavori futuri

Siamo più che soddisfatti del risultato ottenuto. Nel monte ore previsto sono state realizzate tutte le funzionalità previste, ed è avanzato anche del tempo per apportare delle modifiche migliorative alle prestazioni, condurre test più approfonditi, e rifinire alcuni aspetti grafici (come le icone). Per quel che riguarda il framework, il risultato è molto buono, in quanto ha permesso uno sviluppo molto lineare, cioè, superata la progettazione iniziale, è stato molto semplice implementare il resto delle funzionalità. Le prestazioni sono superiori alle aspettative, e nonostante il sistema non sia paragonabile ai CAS commerciali, il progetto è molto interessante da un punto di vista teorico.

Per quel che concerne l'interfaccia grafica, l'obiettivo di realizzare un sistema user-friendly è stato raggiunto. Inoltre, l'impatto visivo è notevole, anche in virtù del rendering delle espressioni usando LaTeX. L'architettura con cui è stata realizzata, inoltre, consente di intercambiare il frontend senza troppi problemi (passando ad esempio ad una Command Line Interface).

Il monte ore è stato sufficientemente ampio per implementare le funzionalità base di un CAS, ma dato che sarebbe stato necessario più tempo per realizzare un progetto di portata maggiore (al livello dei CAS commerciali), certe funzionalità non sono presenti. Ad esempio, non è possibile eseguire il calcolo analitico di equazioni, limiti ed integrali indefiniti, dato che ciò richiederebbe una mole di lavoro enorme. Inoltre, la funzionalità di semplificazione delle espressioni non è sufficientemente consistente, perché le regole algebriche di cui tener conto sono numerose, e quindi un'eventuale espressio-

ne potrebbe non essere semplificata nel modo in cui un utente si aspetterebbe (anche in alcuni casi considerati banali).

Lavori futuri

Il progetto verrà quasi sicuramente esteso in futuro, migliorando ulteriormente l'impatto visivo con l'utente e la fruibilità.

- Potrebbero venire aggiunte funzionalità satellite attorno al framework, come le sommatorie o un sistema di semplificazione più consistente rispetto a quello corrente.
- Supporto al multithreading: funzionalità per il quale il sistema è già stato predisposto, ma che non è stata implementata perché avrebbe richiesto un'analisi specifica per eseguire un lavoro ottimale.
- Miglioramenti alla Plot Window, ad esempio la possibilità di traslare il grafico e di disegnare più funzioni contemporaneamente, ed il supporto allo swipe e zoom con il mouse. Queste funzionalità possono essere implementate con relativa semplicità, dato che il grafico fa già uso di trasformazioni lineari (classe `AffineTransform` di AWT)

Appendice A

Guida utente

L'interfaccia è stata disegnata sulla falsa riga dei CAS esistenti, e quindi è già sufficientemente intuitiva e non presenta particolari difficoltà all'utilizzo. Nella barra delle espressioni si scrive l'equazione o l'espressione da manipolare, e la si invia all'applicazione premendo il tasto Invio o cliccando sul pulsante a sinistra.

Tutte le espressioni inserite verranno mostrate nella parte centrale della pagina, numerate per ordine di inserimento. Ogni riga è selezionabile, ed utilizzando i tasti in alto o le voci del menù è possibile eseguire le varie operazioni matematiche del framework. Selezionando un'espressione e premendo `CTRL + C`, il suo testo viene copiato nella clipboard del sistema operativo, e può essere incollata nell'applicazione o altrove. Attraverso le apposite opzioni, una riga può essere modificata o eliminata.

E' inoltre possibile salvare l'ambiente corrente o caricarne un altro tramite le apposite voci del menu File.

Sintassi delle espressioni

Il parser è molto elastico ed accetta espressioni scritte in vari modi, supportando anche la moltiplicazione implicita ed i segni unari. Esempi di sintassi accettate sono:

- 5sin(x) + 3cos(x) = 2atan(x)*ln(x)
- (-3xˆ2 + 2x + 2)ˆ2 = +y
- 3+2x+1*ln(x)*x-eˆxˆ2
  N.B.: la potenza ha associatività destra, quindi $e^{x^2} = e^{(x^2)}$
- pi*sin(x)cos(x) = 0

- 1.234 + 3 + sin(x)

Notare che è possibile anche inserire numeri decimali, che verranno automaticamente convertiti in frazioni (1.234 = 1234/1000).

Sono presenti anche dei bottoni sotto la barra delle espressioni, che aiutano l'utente nella composizione delle formule.

E' possibile trovare alcuni esempi di fogli di calcolo nell'archivio `Examples.zip`, scaricabile dal repository Bitbucket.

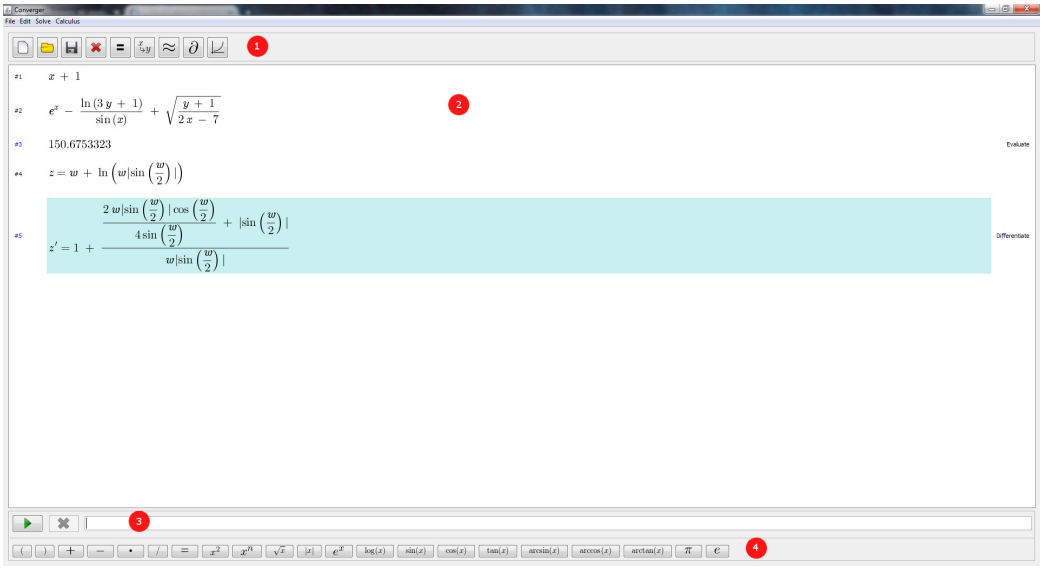

Figura A.1: Tipica schermata dell'applicazione. Sono visibili i pulsanti dell'header (1), che danno accesso a diverse funzionalità, tra cui quella di graficare una funzione di una variabile (ultimo pulsante a destra). Non tutte le funzionalità dell'applicazione sono qui rappresentate, infatti alcune sono presenti soltanto nelle voci del menù. Si può vedere la parte centrale dell'interfaccia grafica (2) in cui vengono visualizzate le espressioni tramite formato LaTeX. L'espressione selezionata è ben visibile, come è visibile la differenza tra le formule matematiche immesse (numero di riga nero) e quelle calcolate dal framework (numero di riga blu e nome dell'operazione che le ha generate sulla destra). Nella parte bassa dell'applicazione è posizionata l'input line (3) dove l'utente può inserire le espressioni matematiche, eventualmente aiutandosi con i pulsanti in basso (4) per comporre le formule.