Relazione per ‘Calculator Collection’

Roberto Lepore, Riccardo Tassinari, Riccardo Alni, Alessandro Agosta, Marco Pesic

25 Aprile 2022

Indice

Capitolo 1
Analisi

1.1 Requisiti

Il software vuole mettere a disposizione una calcolatrice capace di risolvere tipi diversi di problemi, dalle espressioni algebriche più semplici ai problemi di calcolo differenziale e combinatorio. Il riferimento principale è l'applicazione Calcolatrice di Windows, sia per funzionalità che per design visivo.

Requisiti funzionali

- La calcolatrice deve riuscire a risolvere accuratamente espressioni appartenenti ai seguenti domini:
    1. Algebra elementare.
    2. Algebra di Boole.
    3. Calcolo combinatorio.
    4. Calcolo differenziale, in particolare: limiti, derivate e integrali definiti.
- L'applicativo deve permettere di tracciare il grafico di una funzione data.
- La calcolatrice deve permettere di applicare conversioni di base e svolgere operazioni bitwise.
- Dev'essere possibile visionare tutti i calcoli svolti nella sessione corrente.

Requisiti non funzionali

- I domini applicativi devono essere divisi in più 'sotto-calcolatrici', in modo da creare un'interfaccia semplice e intuitiva per l'utente.

1.2 Analisi e modello del dominio

L'applicazione sarà suddivisa in più calcolatrici, ciascuna capace di svolgere un determinato insieme di calcoli. La logica principale dell'applicazione, che chiameremo Manager, dovrà essere in grado di delegare i calcoli alla corretta calcolatrice e di farsi dire dall'utente quale calcolatrice vuole utilizzare. Inoltre, il Manager dovrà mantenere una memoria per l'input e una per la cronologia dei calcoli svolti. Ciascuna calcolatrice sarà composta da una componente visiva con cui l'utente può interagire, un insieme di operatori binari e unari che può utilizzare e una logica per comunicare con il Manager. Sarà importante, quindi, astrarre le funzionalità comuni a tutte le calcolatrici in modo da semplificare il dominio.

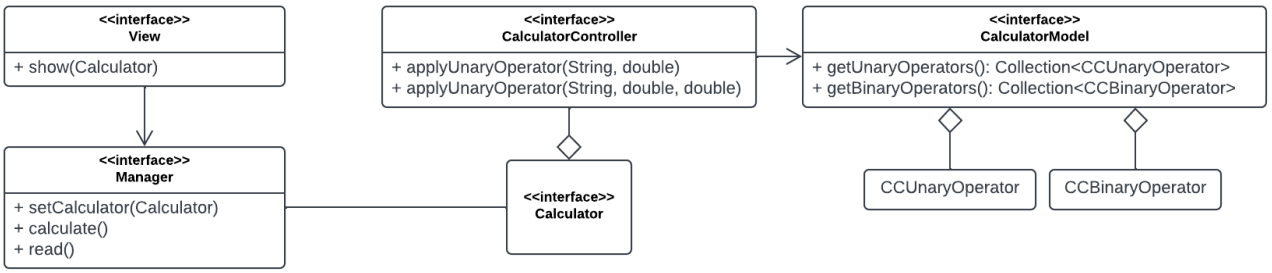

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro.

Capitolo 2
Design

2.1 Architettura

L'architettura di Calculator Collection segue il pattern MVC.
Il controller principale è il Manager, che delega le sue responsabilità a due entità, MemoryManager e EngineManager, che si occupano della gestione della memoria e dello svolgimento dei calcoli rispettivamente. EngineManager comunica con un'altra parte di controller, specifico per ciascuna calcolatrice, che implementa l'interfaccia CalculatorController. Inoltre le logiche di ciascuna calcolatrice hanno il compito di collegare la specifica view e il CalculatorController, manipolando i dati in ingresso e gestendo l'interazione con l'utente.

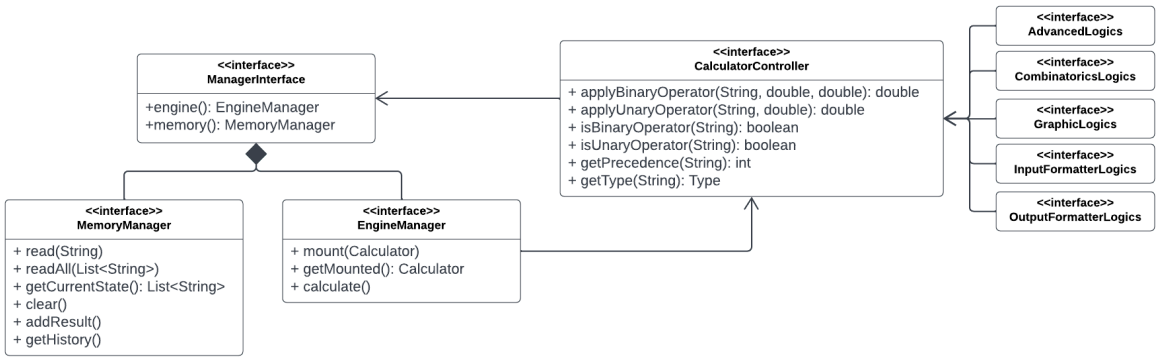

Figura 2.1: Schema UML della componente controller del pattern MVC.

Il model di ciascuna calcolatrice implementa l'interfaccia CalculatorModel e contiene l'insieme di tutti gli operatori che la calcolatrice può gestire con le rispettive implementazioni. Il CalculatorModel viene interrogato dal CalculatorController durante il calcolo. Il model del MemoryManager implementa l'interfaccia MemoryModelInterface, contiene un buffer di memoria per l'input dell'utente e una memoria per la cronologia dei calcoli. Il model del EngineManager implementa l'interfaccia EngineModelInterface, contiene un riferimento alla calcolatrice attualmente utilizzata e l'enumerazione Calculator contenente tutte le calcolatrici disponibili al sistema.

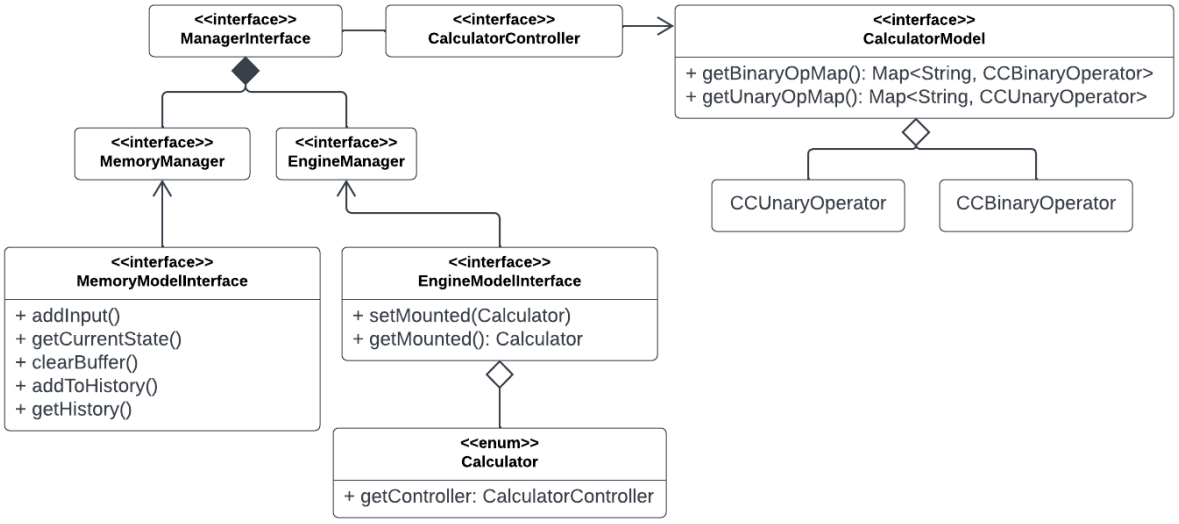

Figura 2.2: Schema UML della componente model del pattern MVC.

L'applicazione si compone di una sola view principale, la cui logica comunica con il manager, contenente le view di tutte le calcolatrici. In questo modo l'architettura è completamente indipendente dalla tecnologia implementata, ma l'aggiunta di una nuova view implica una modifica alla view principale.

2.2 Design dettagliato

Roberto Lepore

Uno dei primi problemi ad essere emerso consiste nella necessità di utilizzare un unico Manager in tutta l'applicazione, condiviso tra le calcolatrici. Questo è necessario perché il Manager tiene traccia della calcolatrice da utilizzare e memorizza la cronologia, per cui non può essere sostituito da un nuovo Manager durante l'esecuzione dell'applicativo. Inoltre questa doppia funzione violerebbe il single responsibility principle, per cui ho deciso di delegare le due funzionalità a MemoryManager e EngineManager. In questo modo il Manager ha solo il compito di fornire alle calcolatrici i riferimenti ai due specifici Manager e mantenerli uniti. Ho deciso di non utilizzare il pattern Singleton perché non risulta particolarmente complesso passare alle calcolatrici il riferimento al Manager. Il Manager viene inizializzato dalla logica della View principale.

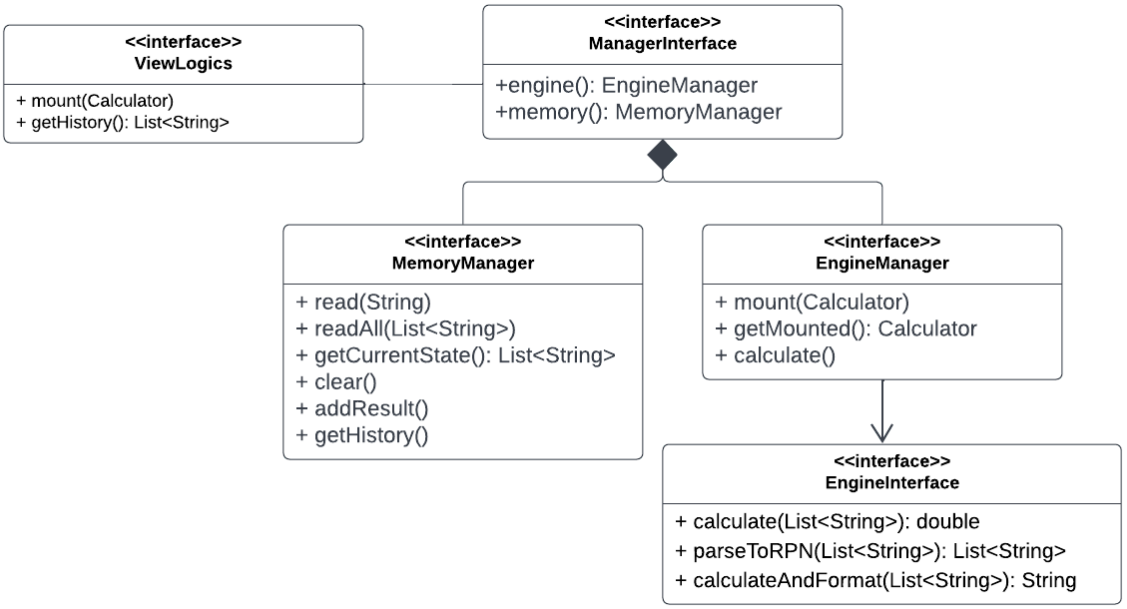

Figura 2.3: Schema UML della delegazione dei compiti del Manager.

Per permettere una completa indipendenza dell'architettura dalla tecnologia scelta per la GUI, si memorizzano le componenti grafiche delle calcolatrici in una struttura dentro la View principale, la quale mostrerà la GUI corretta su richiesta e comunica con il Manager attraverso una componente di logica ViewLogics. Lo svantaggio di questa soluzione consiste nella necessità di modificare l'implementazione della View principale nel caso di aggiunta di nuove GUI e calcolatrici.

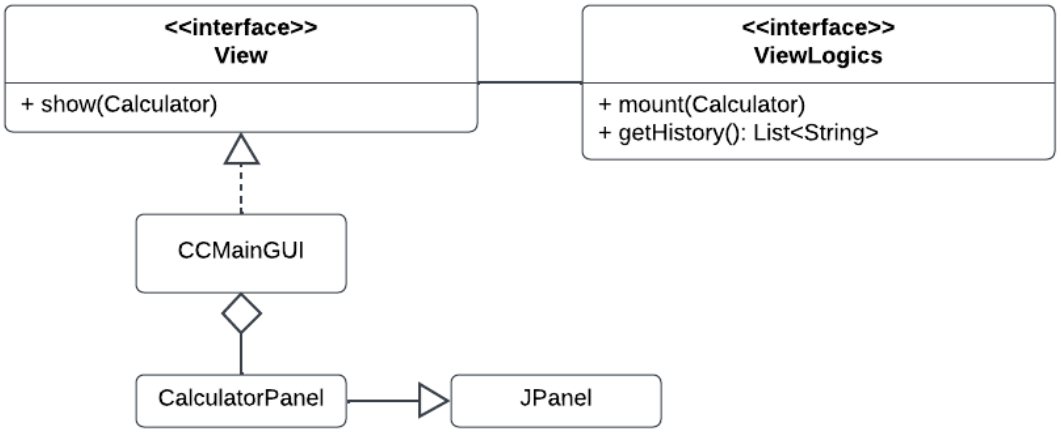

Figura 2.4: Schema UML della view principale.

Le calcolatrici sono modellate come una enumerazione Singleton innestata nell'interfaccia EngineManagerModel. Ho preso questa decisione, nonostante i problemi di estendibilità a cui porta e nonostante non sia strettamente necessaria la non-istanziabilità, principalmente perchè

sono identificate da un nome e perchè è molto utile ottenere i relativi controller in modo semplice in molte classi. Inoltre in questo modo si può essere sicuri che il controller del Calculator sia stato inizializzato correttamente.

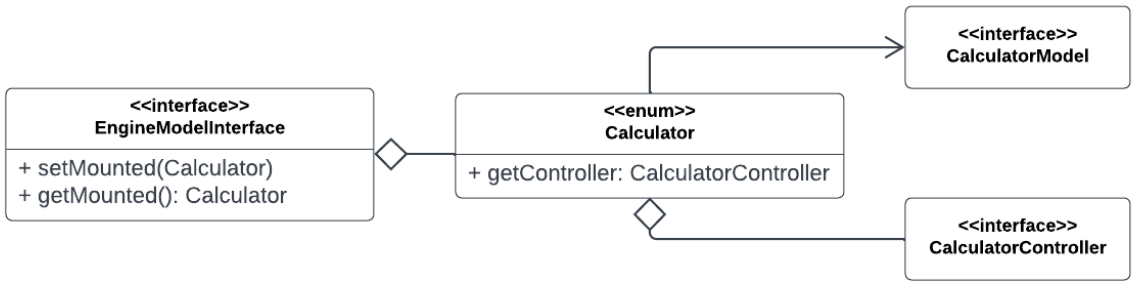

Figura 2.5: Schema UML della enumerazione Calculator.

Riccardo Alni

Abbiamo iniziato implementando il modo in cui venivano memorizzate le operazioni: sono state quindi realizzate le classi CCBinaryOperator e CCUnaryOperator. Queste classi contengono tutte le informazioni utili per ogni operatore tra cui un Unary/BinaryOperator<Double>, una classe di java.util.function che è il fulcro dell'operatore in quanto contiene il suo nome e soprattutto la sua Function/BiFunction.

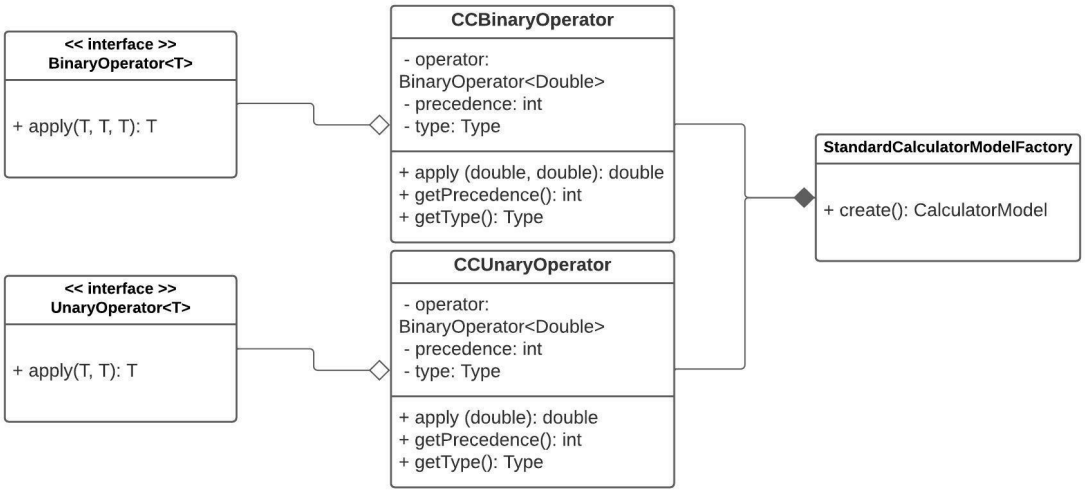

Figura 2.6: Schema UML degli operatori.

Dato che il Model gestisce allo stesso modo ogni operatore di ogni calcolatrice(in particolare sono la Standard, Scientific, Programmer e Combinatorics a implementare gli operatori) abbiamo pensato di creare un unico CalculatorModel che memorizza gli operatori di ogni calcolatrice in due mappe(una per gli unari e una per i binari). Ogni calcolatrice ha una Static Factory che, richiamata dall'enumerazione Calculators, crea un CalculatorModel contenente le due mappe. Abbiamo usato questo pattern per una più semplice leggibilità e per evitare di istanziare un oggetto Factory all'interno del costruttore dell'enumerazione Calculators, purtroppo non potendo

inserire metodi statici senza body nelle interfacce non siamo riusciti a creare un'interfaccia comune per le 4 CalculatorModelFactory.

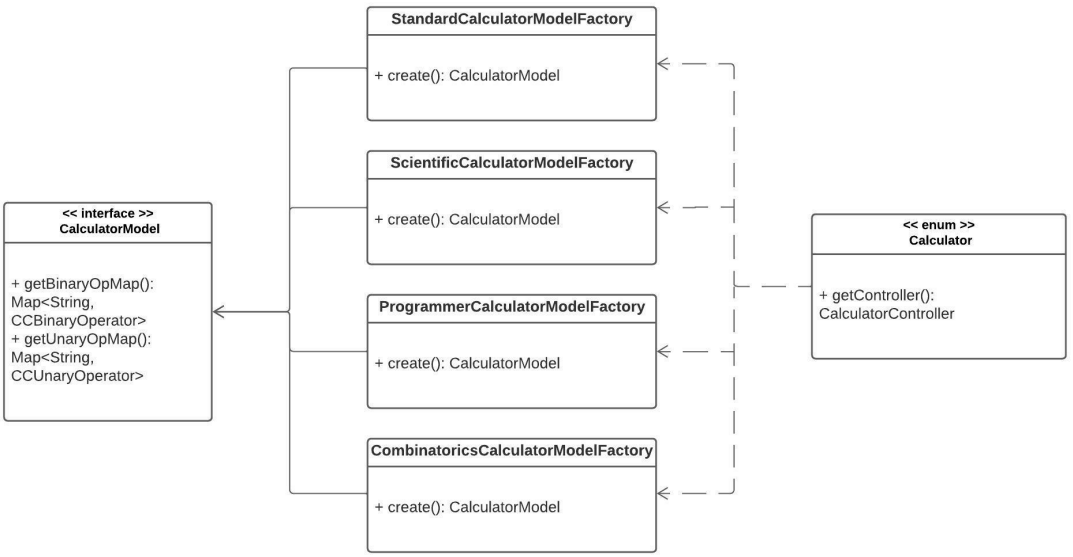

Figura 2.7: Schema UML della Static Factory del Model generale delle calcolatrici.

Il Controller che gestisce il CalculatorModel e comunica con il Manager è CalculatorController. Come nel caso del CalculatorModel, anche CalculatorController è in comune tra le calcolatrici Standard, Scientific, Programmer e Combinatorics e l'unica differenza tra questi controller è il CalculatorModel che viene utilizzato per ricavare le mappe. Per questo l'enumerazione Calculators istanzia i CalculatorController attraverso una Simple Factory a cui viene passato come parametro del metodo *create* il CalculatorModel.

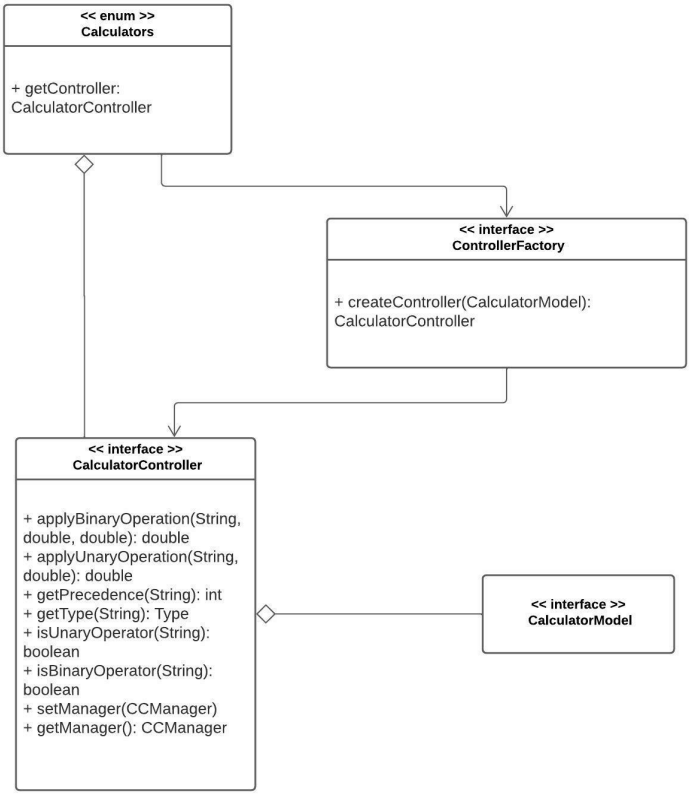

Figura 2.8: Schema UML della Simple Factory del Controller generale delle calcolatrici.

Infine era da gestire il modo in cui i dati in input venivano passati al CalculatorController per poi essere dati in output alla View. Si è creato un nuovo controller CombinatoricsLogics che agisce da ponte tra la View(CombinatoricsCalculatorPanel)e CalculatorController.
CombinatoricsLogics agisce anche come pattern Strategy per CombinatoricsCalculatorPanel.

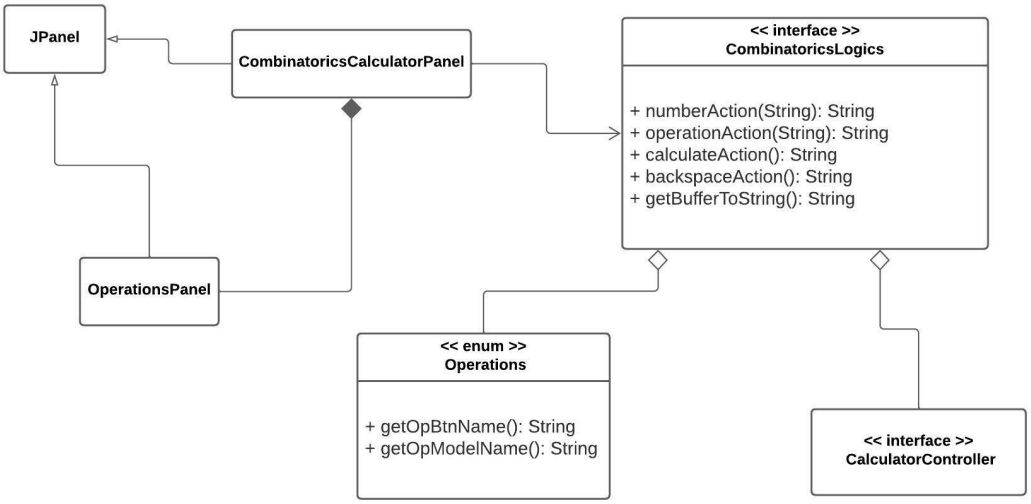

Figura 2.9: Schema UML del rapporto View-Controller della CombinatoricsCalculator

Riccardo Tassinari

Calcolatrice Scientifica

Per quanto riguarda la calcolatrice scientifica mi sono reso conto che, come nella calcolatrice di Windows, essa avrebbe dovuto mettere a disposizione sia i suoi operatori esclusivi, sia gli operatori della calcolatrice standard. Per risolvere il problema, il metodo statico create della ScientificCalculatorModelFactory inizializza le sue binaryOpMap e unaryOpMap con due HashMap e successivamente, con un putAll, chiama il metodo create della StandardCalculatorModelFactory per aggiungervi le binaryOpMap e unaryOpMap della calcolatrice standard.
Sempre riguardo al model, ho optato per una scrittura di codice il più compatta possibile tramite l'utilizzo di espressioni lambda e successiva implementazione del body per ogni operatore, il tutto dentro al metodo create.
La parte di view della calcolatrice scientifica comunica due interfacce InputFormatterLogics e OutputFormatterLogics, le quali comunicano con il controller.

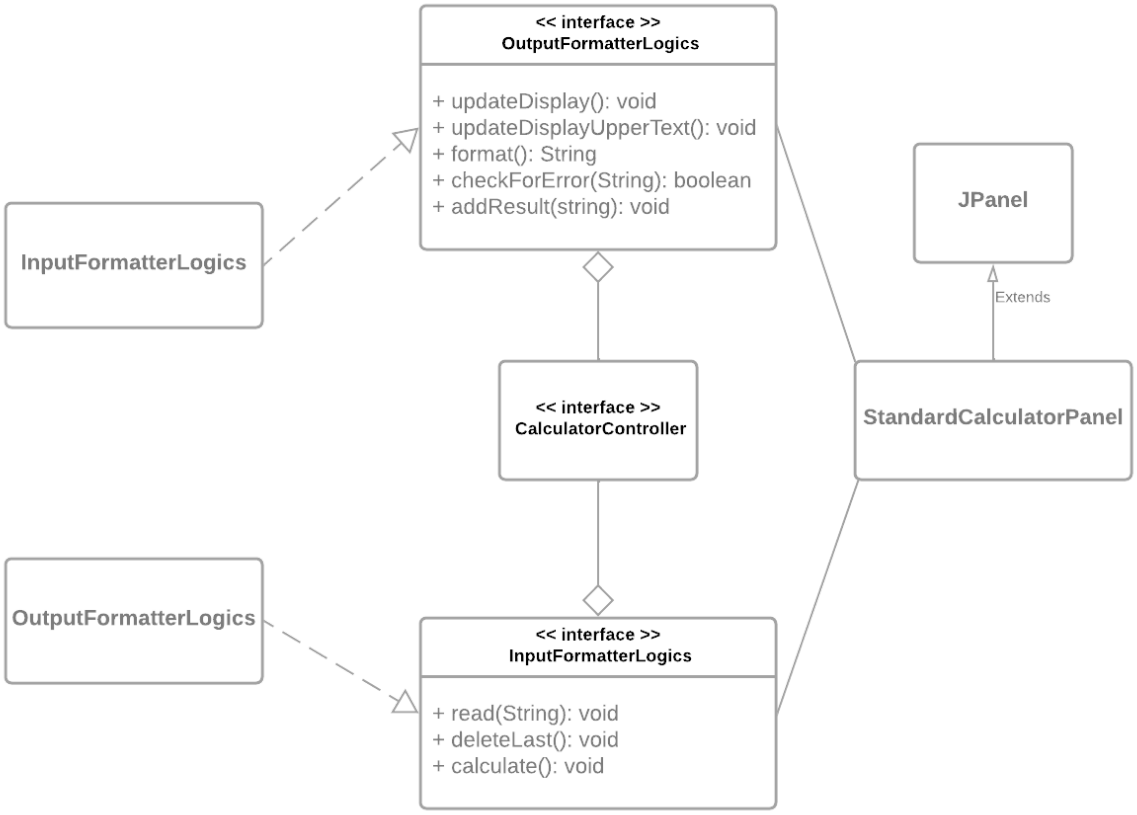

Figura 2.10: Schema UML della ScientificCalculator

Calcolatrice Grafica

Ho avuto chiaro sin dall'inizio che per disegnare il grafico di una funzione avrei dovuto ottenere una stringa rappresentante una funzione f(x), calcolare la funzione per un numero definito di x e poi disegnarla su un JPanel.

Ho creato quindi tre classi specializzate nella risoluzione di ognuno di questi problemi
- **FunctionGrapher**.
Il suo scopo è quello di memorizzare e disegnare le funzioni sul JPanel. Ha due metodi pubblici addFunction e deleteFunction che vengono chiamati dal FunctionInsertionPanel.
- **FunctionInsertionPanel**.
Gli viene passato come parametro il FunctionGrapher e istanzia GraphicLogics.
Dopo aver ottenuto la stringa dall'utente, chiama il metodo calculate di GraphicLogics, che non restituisce però subito il risultato ma lo salva in un campo. Il risultato viene restituito tramite la chiamata del metodo getResult.
Chiama poi il metodo addFunction del FunctionGrapher e passa come parametro il metodo getResult.
Il compito di questa classe quindi è sì quello di ottenere la funzione da input, ma anche quello di passare i dati elaborati dal GraphicLogics al FunctionGrapher.
Chiama anche il metodo deleteLast di FunctionGrapher.
- **GraphicLogicsImpl** che implementa l'interfaccia **GraphiLogics**.
Il suo scopo è quello di prendere una stringa, rappresentante una funzione, e calcolare un numero definito di punti ottenuti dalla sostituzione delle x con valori crescenti.
Per tradurre la stringa letta in una List<String> leggibile poi dal manager, istanzio un Tokenizer, classe creata e utilizzata anche dal mio collega Marco Pesic. Comunica poi il manager per eseguire le operazioni tradotte tramite il tokenizer. La calcolatrice grafica non ha un model tutto suo, bensì si avvale del model della calcolatrice scientifica.

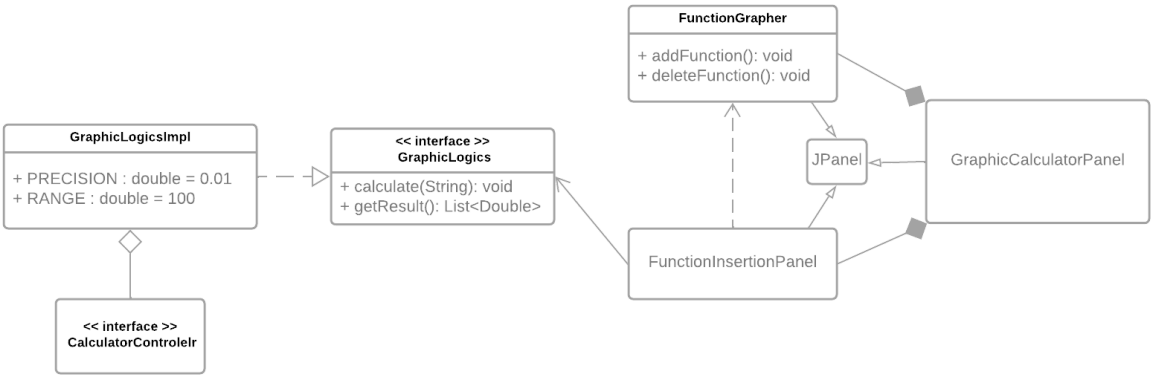

Figura 2.11: Schema UML della GraphicCalculator

Alessandro Agosta

Per meglio gestire l'input e l'output delle calcolatrici è stato necessario creare due interfacce InputFormatterLogics e OutputFormatterLogics implementate per le calcolatrici **Standard** e **Programmer**.

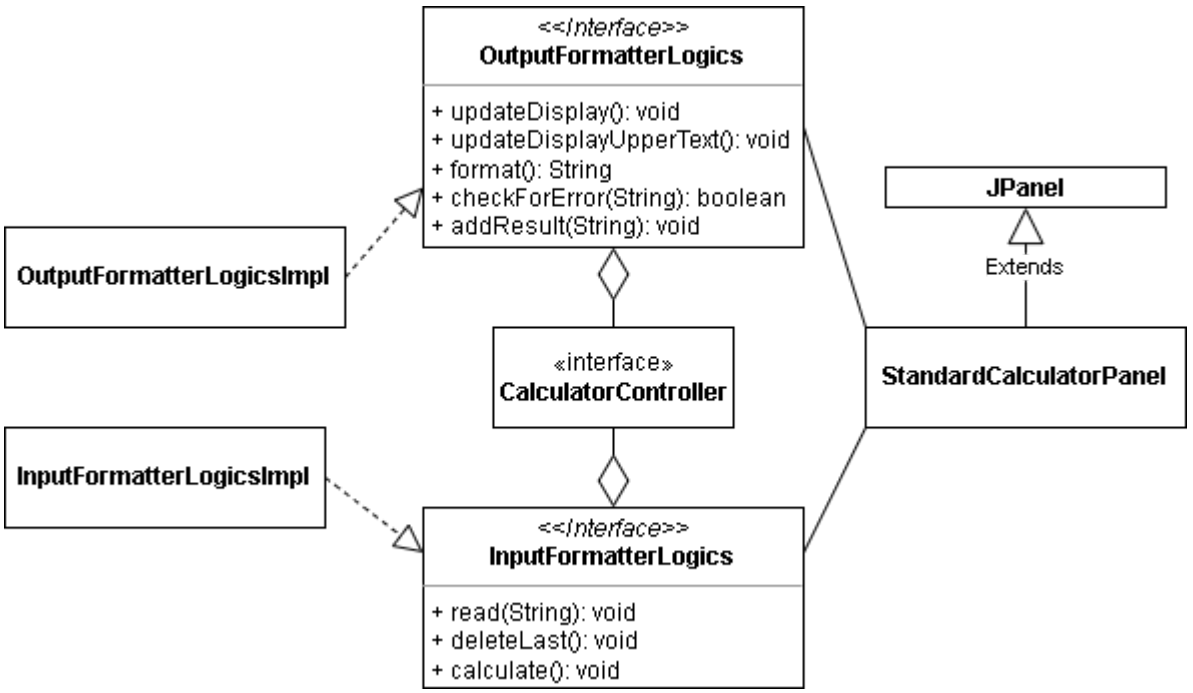

Figura 2.12: Schema UML della StandardCalculator

Le due interfacce sono la parte Logic del MVC e mediano il Panel assegnato ed il CalculatorController.

Nella **Programmer** viene utilizzata una classe di utility che converte i dati nelle varie basi numeriche, tutte le operazioni matematiche vengono effettuate in base decimale, mentre l'output nelle altre basi viene visualizzato da ConversionPanel.

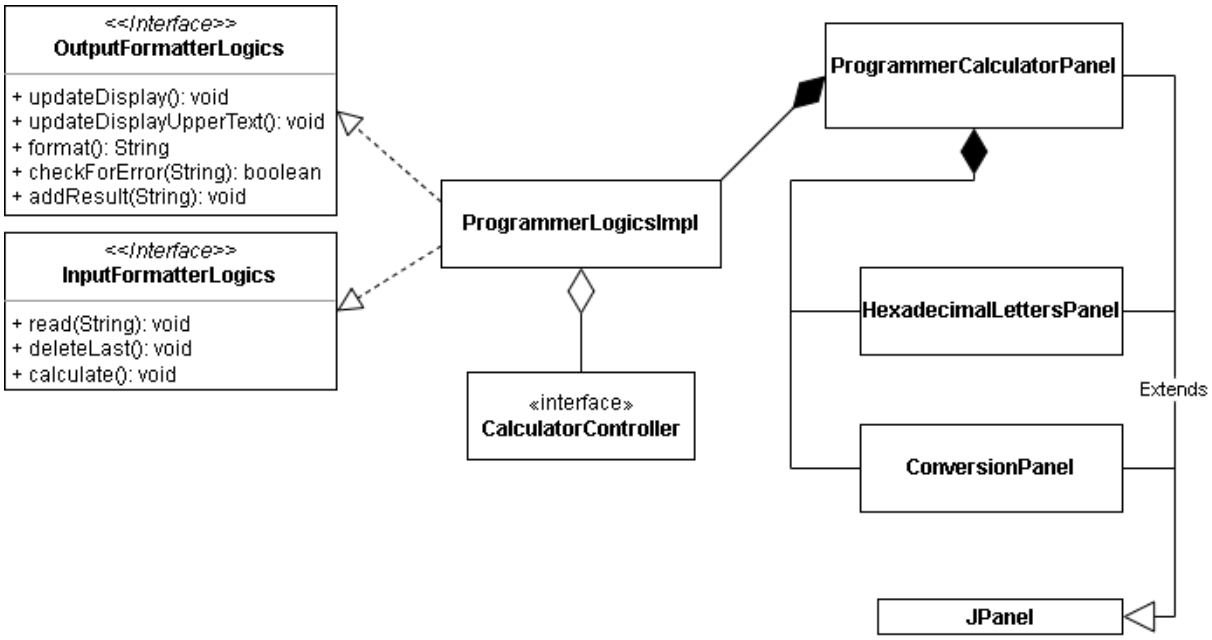

Figura 2.13: Schema UML della ProgrammerCalculator

Marco Pesic

La mia parte del progetto richiedeva l'esecuzione e il calcolo simbolico di derivate, integrali definiti e limiti calcolati numericamente e infine la parte di GUI della mia calcolatrice. Naturalmente queste componenti richiedono un modo di gestire espressioni matematiche con presenza di incognite , in questo caso solo una. La soluzione a cui sono arrivato è quella di utilizzare un Abstract Syntax Tree per memorizzare l'espressione e per poi valutarla. Ho diviso la costruzione dell'albero in varie interfacce . Tutto viene gestito dal MathematicalExpression che una volta ottenuta la stringa di input , passa l'output al componente successivo in questo caso il risultato del MathematicalParser al TreeEvaluator. A sua volta il MathematicalParser adopera altre interfacce per assolvere il suo compito.

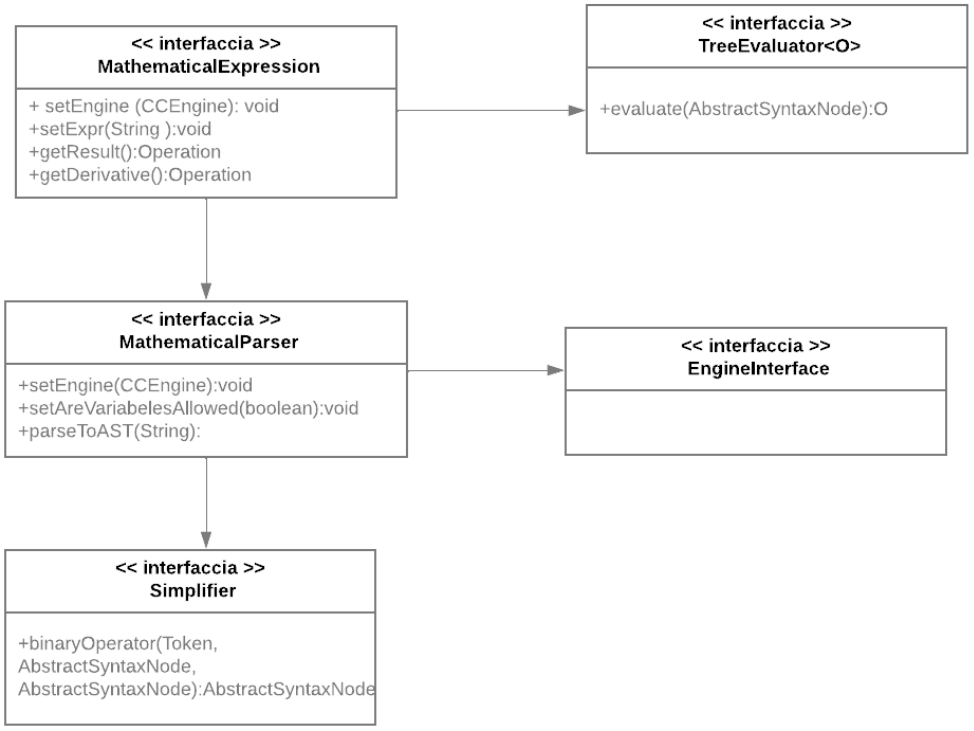

Figura 2.14: Schema UML per la valutazione di espressioni matematiche.

In questo caso sfruttiamo l'interfaccia EngineInterface per trasformare l'espressione in un formato polacco, per poi passare dal MathematicalParser per trasformare l'espressione in formato polacco in un Abstract Syntax Tree.Passando prima dal Simplifier per simplificare determinate espressioni prima di inserirle nell'albero. Per calcolare l'espressione il treeEvaluator esegue un post-order traversal sia per calcolare l'espressione in un determinato punto oppure per derivare l'espressione.

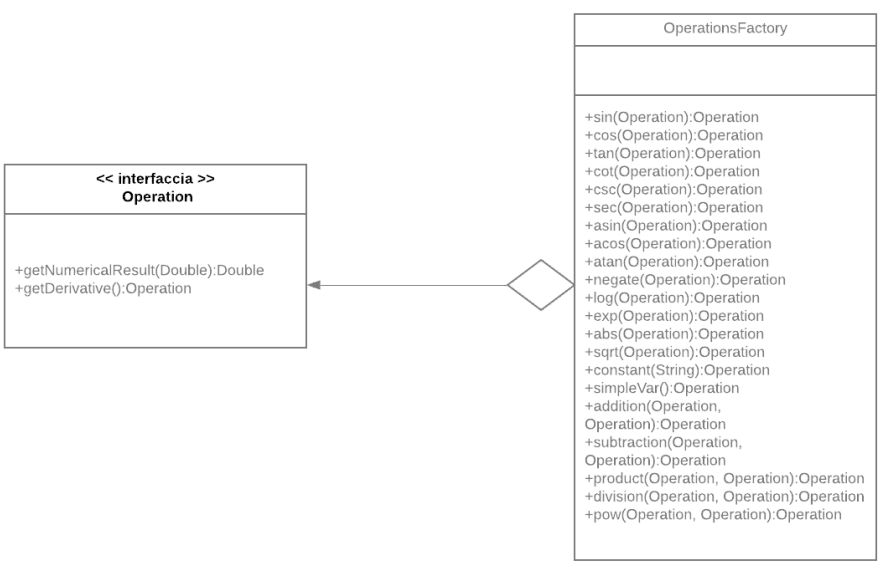

Figura 2.15: Schema UML per come vengono gestite le varie Operazioni.

L'interfaccia generica evaluator nel mio caso ritorna un tipo operation che possiamo vedere nella figura 2. . L'interfaccia operation permette essenzialmente di accumulare il risultato di tutta l'espressione in un solo nodo, per poi chiamare uno dei 2 metodi per ottenere il risultato richiesto ;la valutazione in punto, oppure la derivata simbolica. Utilizzo una **Static Factory** per la creazione delle diverse istanze di Operation, soprattutto dal punto di vista del numero di classi che diventerebbe molto grande altrimenti.

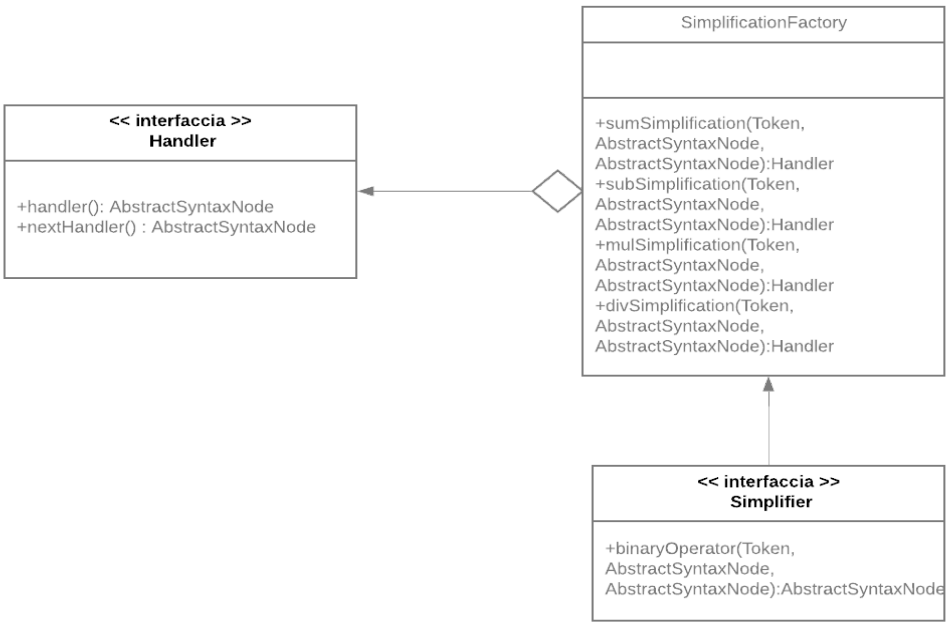

Figura 2.16: Schema UML della gestione delle semplificazione delle espressioni.

Per quanto riguarda il simplifier di figura 2.16 , questo era neccessario per ridurre il numero di nodi possibili , infatti per tipi di operazioni binarie come la somma, sottrazione, prodotto e divisione ci possono essere determinate sottoespressioni semplificabili (e.g. 5/1 -> 5) che quindi sprecherebbero soltanto spazio in memoria. Siccome per ogni operazione esistono più di una "soluzione banale" (e.g. per la moltiplicazioe 5*0 = 0, 5*1 = 5)e dovendo tenere in conto anche l'ordine dei nodi, ovvero nodo sinistro e nodo destro. Ho deciso di utilizzare una **chain of responsability** per ogni operazione e nel caso nessuno degli Handler riesca a semplificare la sottoespressione questa viene delegata al "Identity Handler" che restituisce la sottoespressione iniziale. Per creare le varie chain utilizzo la SimplificationFactory che è una **static factory** che vien utilizzata dall'interfaccia Simplifier.

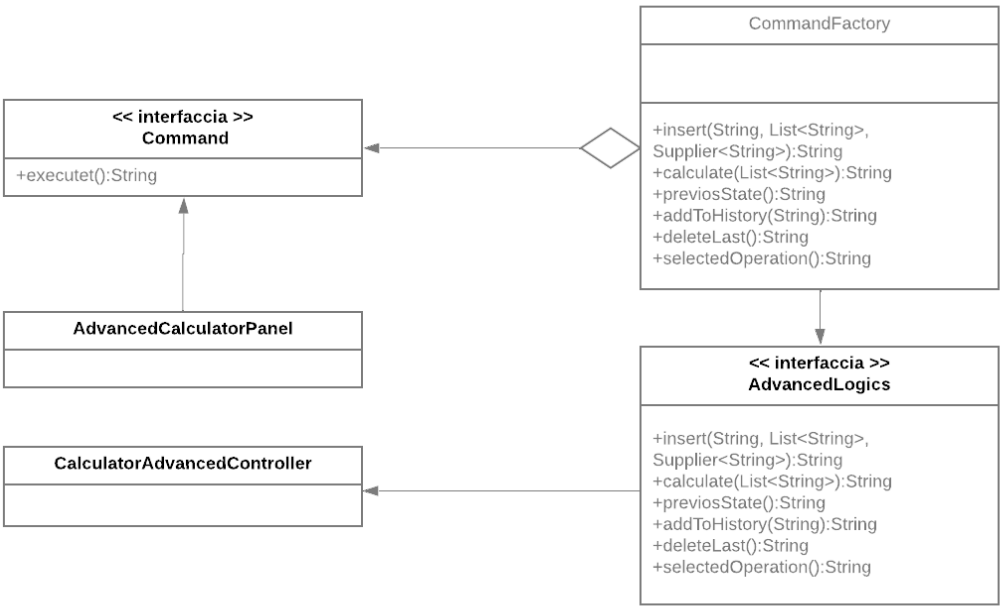

Figura 2.17: Schema UML della GUI

Nella parte grafica , separo la logica di business dalla view utilizzando una **Strategy** e poi adoperando un **Command** che ha come receiver l'AdvancedLogics.

Capitolo 3
Sviluppo

3.1 Testing automatizzato

Abbiamo scritto diverse classi di test JUnit, una per ogni funzionalità principale del software.

- BitwiseOperationsTest: testa tutte le operazioni bit a bit della calcolatrice programmatore.
- CombinatoricsOperationTest: testa tutte le operazioni della calcolatrice combinatoria.
- ConversionsTest: testa le conversioni tra le basi binaria, ottale, decimale e esadecimale, messe a disposizione dalla calcolatrice programmatore.
- DerivateTest: testa il calcolo di derivata di semplici espressioni.
- FormatterTest: testa numerosi edge cases della formattazione personalizzata dei numeri, funzionalità messa a disposizione dalla classe NumberFormatter.
- ManagerTest: testa il corretto svolgimento di semplici calcoli attraverso il Manager, della lettura di input e del calcolo simbolico.
- ScientificCalculatorOperationsTest: testa tutte le operazioni della calcolatrice scientifica.

Il testing della GUI è avvenuto in modo manuale, in modo da poter verificare la semplicità di utilizzo da parte dell'utente, la correttezza dei grafici costruiti e il corretto funzionamento del cambio di calcolatrice in tutte le sue parti.

3.2 Metodologia di lavoro

Abbiamo dedicato la prima parte di progettazione ad una analisi del problema, svolta tutti insieme, cercando di individuare i principali elementi del dominio e di dividerci il lavoro in modo equo. Purtroppo l'analisi iniziale si è rivelata superficiale e ha portato a individuare solo in un secondo momento la possibilità di generalizzare alcuni elementi e la necessità di inserire alcuni metodi. Ciò ha portato alla modifica di codice e interfacce in corso d'opera. Per esempio, ci siamo resi conto successivamente della possibilità di usare delle factory per controller e model delle calcolatrici.
Insieme abbiamo discusso di cosa potesse essere necessario nelle interfacce di controller e model delle calcolatrici e della struttura degli operatori (CCBinaryOperator e CCUnaryOperator), in quanto sono parti che interessano tutti. Abbiamo cercato di sviluppare parallelamente parti di progetto il più possibile indipendenti tra loro, comunicando nel caso si rivelasse necessaria una dipendenza o una modifica in una classe di un altro componente del gruppo.

Di seguito indichiamo, in una sezione dedicata a ciascun componente del gruppo, la divisione del carico di lavoro effettuata. Riteniamo che ognuno abbia avuto la possibilità di sviluppare parti di model, view e controller.

Roberto Lepore

Mi sono occupato dello sviluppo del controller principale, che si è concretizzato nelle classi CCManager, CCMemoryManager, CCEngineManager e CCEngine (classe delegata allo svolgimento degli algoritmi necessari al calcolo). Il Memory Manager e l'Engine Manager manipolano i dati memorizzati nei rispettivi model CCMemoryModel e CCEngineModel. L'interfaccia EngineModelInterface contiene innestata l'enum Calculator, di cui ogni valore rappresenta una calcolatrice e crea il relativo controller. Come parte di view ho sviluppato il JFrame principale CCMainGUI, le componenti CCNumPad e CCDisplay e il panel che mostra la cronologia HistoryPanel. Inoltre ho sviluppato la classe di utilità NumberFormatter che permette di formattare i numeri in modo personalizzato, anche nascondendo alcuni eventuali errori di calcolo di Java (per esempio, l'errore di arrotondamento prodotto dal calcolo 0.1 + 0.2).

Riccardo Tassinari

Mi sono occupato dello sviluppo della calcolatrice scientifica e della calcolatrice grafica. Ciò mi ha portato a creare varie classi adibite a compiti diversi, soprattutto per quanto riguarda la calcolatrice grafica. Per la calcolatrice scientifica ho creato il ScientificCalculatorModelFactory e il rispettivo ScientificCalculatorPanel, che istanzia il CCDisplay, il CCNumpad, un JPanel per gli operatori della calcolatrice standard e un panel chiamato ScientificOperatorsPanel, presente nel package view.components. Ho anche lavorato inseme al mio collega Alessandro Agosta per realizzare e ottimizzare le due interfacce InputFormatterLogics e OutputFormatterLogics e le rispettive implementazioni, le quali vengono largamente utilizzate nei nostri panel.
Ho anche creato la classe ScientificCalculatorOperationsTest.
Per quanto riguarda la calcolatrice grafica ho creato l'interfaccia GraphicLogics e la rispettiva GraphicLogicsImpl, e le altre due classi FunctionGrapher e FunctionInsertionPanel, istanziate poi nel JPanel finale GraphicCalculatorPanel.

Riccardo Alni

Ho lavorato al design generale delle calcolatrici Standard, Scientific, Programmer e Combinatorics all'interno del pattern MVC. In particolare al modo in cui vengono memorizzati gli algoritmi risolutivi delle operazioni di ogni calcolatrice creando l'interfaccia CalculatorModel e la sua implementazione. Questa classe contenente gli operatori viene usata nella creazione di un CalculatorController attraverso ControllerFactory. Ho anche implementato varie classi di utility come CustomJToolTip e, insieme alla collaborazione dei miei compagni, CCBinaryOperator(e l'equivalente unaria), CalcException e l'enumerazione Type. Mi sono occupato poi dello sviluppo della calcolatrice combinatoria: ho sviluppato CombinatoricsCalculatorModelFactory,

CombinatoricsCalculatorPanel (e i file .txt contenuti in “resources”) e il relativo controller in controller.calculators.logics.

Agosta Alessandro

Oltre alla gestione della repository, la mia parte del progetto riguardava le calcolatrici Standard e Programmer.
Per l’implementazione ho utilizzato le seguenti classi:

- ConversionAlgorithms (utility che contiene algoritmi di conversioni tra le basi 2,8,10,16)
- ConversionPanel (selezione delle varie basi numeriche per l’input)
- ProgrammerCalculatorPanel
- StandardCalculatorPanel

Marco Pesic

Io mi sono occupato nello sviluppo della calcolatrice Advanced che permette calcolo di derivate simboliche, integrali definiti e limiti numerici. Per fare questo ho creato varie classi e interfacce: Tokenizer, ParserAST, EvaluatorAST, Expression, SimplifyingEngine, AdvancedCalculatorController, TokensFactory. Il tokenizer separa una stringa in token utili e sensati e permette l’utilizzo anche di moltiplicazione implicite. AdvancedCalculatorController permette di calcolare le varie operazioni integrandole con il manager e l’engine principale, sfruttando un metodo di evaluate diverso dalle altre calcolatrici. TokensFactory serve al tokenizer per separare in token la stringa, questo mi ha permesso di distinguere funzioni,operatori unari o meno.

Nell’utilizzo del repository Github abbiamo seguito la seguente procedura:

1. Ognuno sviluppa sul proprio fork del repo principale
2. Quando una feature o una sua parte è stata completata senza errori viene creata una pull request.
3. In seguito all’approvazione del gruppo viene eseguito il merge sul branch develop del repo principale.
4. Una volta che nel branch develop si trovano feature completate e necessarie al resto del progetto, si esegue il merge del branch develop sul branch principale master.

3.3 Note di sviluppo

Roberto Lepore

- **Optional**: utilizzato in CCMainGUI e nell'Engine Manager per la calcolatrice attualmente in uso.
- **Bounded Wildcards**: nella mappa per i JPanel delle calcolatrici in CCMainGUI.
- **Reflection**: per istanziare i JPanel delle calcolatrici su richiesta.
- **Lambda Expressions**
- **Stream**
- **Shunting Yard Algorithm:** per il parsing delle espressioni da notazione infissa a notazione polacca inversa [MR35]

Sono stati presi dal web e riadattati: ultima riga del metodo *trimZeros* in NumberFormatter (fonte: Remove trailing zero in Java - Stack Overflow) e l'algoritmo di valutazione di un'espressione in notazione polacca inversa, implementato nel metodo *evaluateRPN* in CCEngine (fonte: Parsing/RPN calculator algorithm - Rosetta Code)

Riccardo Tassinari

- **Lambda expressions**
- **Streams**

Per quanto riguarda lo studio delle librerie grafiche necessarie allo sviluppo della calcolatrice grafica, mi sono informato man mano che procedevo con l'implementazione, facendo occasionalmente modifiche per ottimizzare il codice il più possibile. Mi sono informato da numerosi siti, tra cui le javadoc, o video reperiti online, senza però copiare parti di codice, bensì per farmi un'idea su come partire e che strumenti usare per poi implementare il codice che avevo in mente.

Riccardo Alni

- **Lambda Expressions e Stream:** uso molto ristretto per snellire il codice e renderlo più leggibile
- **Algoritmo per i numeri di Fibonacci:** si considera un numero di Fibonacci F_n come il numero di sequenze binarie di lunghezza n che non contengono "1" consecutivi, per calcolarlo utilizzo $F_{n,k}$ ovvero il numero di sequenze binarie di lunghezza n in cui compaiono k "1" e nessuno è consecutivo ad un altro. Calcolo $F_{n,k}= \binom{n-k+1}{k}$ e F_n come la sommatoria di tutti i $F_{n,k}$

$$F_n = \sum_k \binom{n-k+1}{k}$$

- **Algoritmo per i numeri di Bell:** ho usato la formula iterativa sostituendo n con N=n-1 per calcolare B_N. Questo metodo è l'unico che risulta particolarmente pesante e a numeri relativamente alti il programma tende a metterci molto a calcolare il risultato e in certi casi manda in blocco anche il programma, il numero massimo per cui ritorna un risultato è 30

  $$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k.$$

  fonte: Bell number - Wikipedia
- **Algoritmo per i numeri di Stirling:** ho usato la formula iterativa

  $$S(n,k) = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^n$$

  fonte: Numeri di Stirling - Wikipedia
- **Algoritmo per il numero di Scombussolamenti:** applicando il principio di inclusione/esclusione sul numero di permutazioni si ottiene che l'insieme S degli scombussolamenti ha cardinalità

  $$|S| = \sum_{i=2}^{n} (-1)^i (n)_{n-i}$$

Di grande aiuto sono state le slide del corso di Matematica Discreta e Probabilità del prof. Caselli (lezioniNew.pdf). In particolare, dalle slide ho preso gli algoritmi sui numeri di Fibonacci e di scombussolamenti (derangements) e qualche esempio nei file di testo in resources.
La classe CustomJToolTip l'ho creata vedendo una risposta su StackOverflow: How to set the background of tooltip in Swing? - Stack Overflow, mentre la classe per la lettura da file all'interno del jar l'ho creata seguendo Java - Read a file from resources folder - Mkyong.com

Alessandro Agosta

- **Lambda expressions**
- **Streams**

Gli algoritmi di conversione sono di base già presenti in Java all'interno della classe Long, con modifiche per la gestione del segno.

Marco Pesic

- **Lambda expressions**
- **Streams**
- **Abstract Syntax Tree :** per il calcolo delle derivate
- **Optional**
- **Regola del trapezio:** per il calcolo degli integrali definiti

Per il calcolo di derivate è stato fondamentaleLa tesi di Laurea di Ivan Capponi. Per quanto riguarda l'implementazione del parser, evaluator e l'idea iniziale di adoperare un AST per

calcolare le espressione matematiche, mi sono venute leggendo questo blog che mi è stato di molto aiuto. Per quanto riguarda il calcolo di limiti Calculate Limits with java questo thread ha ispirato la mia implementazione. Per il Tokenizer ho preso spunto da questa libreria.

Capitolo 4
Commenti finali

4.1 Autovalutazione e lavori futuri

Roberto Lepore

Mi sento soddisfatto del codice che ho prodotto a livello implementativo. Credo di aver scritto codice semplice e pulito, utilizzando i costrutti che durante il corso ci avete indicato come migliori e seguendo le convenzioni di programmazione.
Inoltre credo di essere riuscito a lavorare bene in gruppo e a dare un importante contributo allo sviluppo collaborativo del progetto.
D'altro canto non sono soddisfatto del design che ho prodotto. Non sono riuscito ad utilizzare particolari pattern di design, il che ha portato ad una architettura molto semplice. Questo è stato sicuramente causato da una fase di progettazione del design che non ha tenuto conto adeguatamente della qualità desiderata.

Riccardo Tassinari

Mi sento soddisfatto per quanto riguarda la calcolatrice grafica, anche se mi rendo conto che sia un pò limitata e avrei potuto fare meglio. Mi è dispiaciuto non essere stato in grado di aggiungere tutte le funzionalità che avrei voluto, ad esempio la possibilità di "muoversi" in altre zone del grafico, o di risolvere alcuni problemi, come generazione di parti di grafico non volute dovute a problemi con il dominio delle funzioni. Per quanto sia soddisfatto del risultato prodotto, sento anche che con una maggiore ricerca e un maggiore studio di tutta la branca riguardante la "grafica" sarei riuscito a produrre del codice con un design migliore.
Inoltre, inizialmente ero molto titubante sul se far scrivere o no le funzioni da tastiera.
L'idea di prendere stringhe da tastiera, per quanto mi piacesse, apriva la porta a molti problemi nei quali mi sarei potuto perdere facilmente, principalmente dovuti al fatto che sarebbe stato necessario convertire ciò che veniva letto in qualcosa che poi sarebbe stato possibile calcolare. Fortunatamente, parlando con il mio collega Marco Pesic, è venuto fuori che esso aveva già realizzato per necessità sua una classe di utility specializzata nella risoluzione di questo problema.
Non mi sento invece soddisfatto del mio lavoro riguardante la calcolatrice scientifica.
Sento di essermi interessato poco e di conseguenza di aver prodotto solo il codice essenziale per far sì che tutto funzionasse, senza interessarmi a possibili aggiunte o miglioramenti.
Inoltre mi sarebbe piaciuto rendere la calcolatrice scientifica più unica, magari anche con l'aggiunta di piccole funzionalità che non l'avrebbero resa solo una versione "con più operatori" della calcolatrice standard.

A fin dei conti sento di aver imparato molto da questo progetto, mi sono divertito a implementare la calcolatrice grafica e sono contento del risultato complessivo finale.

Riccardo Alni

Ho trovato stimolante la realizzazione del progetto soprattutto nelle sezioni più "di gruppo". Per quanto la mia parte di model non fosse particolarmente complessa mi sono comunque "divertito" nella progettazione del design ma, nonostante sia la parte che mi è piaciuta di più, penso sia anche una delle meno riuscite in quanto ci siamo visti più volte, anche e soprattutto in corso d'opera, a cambiare il design del progetto. Questo è stato dovuto secondo me ad una sottovalutazione del progetto e all'esserci "accontentati" troppo facilmente di un design funzionante senza sforzarci troppo a cercare uno più ottimale. I vari cambi apportati al design ci hanno fatto perdere molto tempo che potevamo investire nelle parti di model: nonostante sia abbastanza soddisfatto del codice che ho prodotto non ho avuto il tempo per implementare qualche funzione extra come il multithreading o l'uso di qualche libreria non studiata durante il corso.

Alessandro Agosta

L'uso del DVCS è stato implementato tenendo conto di quanto appreso in laboratorio,
l'ho ritenuto un valido strumento di sviluppo anche per futuri progetti in singolo.
L'utilizzo di repository isolate in remoto ha permesso lo sviluppo dell'applicazione in modo semplice ed efficace.
Per quanto avrei trovato interessante utilizzare più pattern o il multithreading ciò non avrebbe apportato miglioramenti sostanziali alle prestazioni.
La scelta di gruppo del design ha assorbito molto tempo a scapito di quanto avremmo voluto dedicare alle funzionalità opzionali ed alla qualità complessiva del progetto.

Marco Pesic

Il progetto si è rivelato più complicato del previsto da più angolazione. Sia dal punto di vista implementativo che dal punto di vista del Design. Anche se mi ritengo soddisfatto del codice scritto, sono sicuro che avrei dovuto passare più tempo a ragionare sull'utilizzo di pattern. Inoltre credo che avrei potuto integrare meglio la mia parte con le varie componenti dei miei colleghi. Ritengo che per certi componenti avrei potuto attuarli meglio, ad esempio la gestione del processo di semplificazione avrei dovuto applicarla sull'albero e non durante il parsing questo mi avrebbe permesso di non dover ricalcolare l'espressione due volte quando calcolo una derivata. Inoltre anche il calcolo dei limiti , mi sarebbe piaciuto utilizzare metodi più complessi che permettevano di risolvere ancora più limiti, rispetto ora.

4.2 Difficoltà incontrate e commenti per i docenti

Abbiamo usato la libreria grafica Swing per comodità dato che l'avevamo studiata a lezione e pensavamo bastasse per il nostro elaborato, tuttavia si è rivelata abbastanza ostica la gestione del ridimensionamento della finestra.

Appendice A
Guida utente

La calcolatrice mostrata in partenza è la calcolatrice standard. Può essere selezionata una differente calcolatrice in qualsiasi momento dal menù Select Calculator in alto a sinistra, e può essere mostrata/nascosta la cronologia dei calcoli premendo il tasto History in alto a destra.

La Graphic Calculator e l'Advanced Calculator utilizzano l'input da tastiera, che facilità all'utente l'inserimento di espressioni matematiche permettendo l'omissione del simbolo di moltiplicazione, dove possibile.
Gli operatori validi sono: **sin, cos, tan, csc, sec, cot, log, ln, abs, ^, root, +, -, *, /.**
Costanti valide sono: **pi, e.**

Graphic Calculator
Si scriva una funzione nella variabile *x* e si prema il pulsante ADD per mostrare la rispettiva curva, con un colore casuale, sul piano cartesiano. Possono essere aggiunte numerose curve e con il pulsante DELETE LAST si può cancellare l'ultima curva inserita.

Programmer Calculator
È possibile selezionare la base per l'input dei dati (esadecimale, decimale, ottale e binaria), tramite bottoni(azzurri).
La conversione viene mostrata anche durante la digitazione, mentre il risultato è visualizzato dopo l'esecuzione dell'operazione.
Il range delle operazioni è limitato a 64 bit, cioè un long con un valore massimo rappresentabile di 2^64-1.
Il segno viene visualizzato correttamente con "+" e "-" ed esprime il bit significativo del valore.
Gli operatori binari (roL, roR, shiftL, shiftR, nand, not, nor) implementano la funzione adeguandosi alla dimensione in byte degli operandi.
L'operatore unario (not) non ha bisogno di altri input.

Advanced Calculator
Dal menù a tendina a destra è possibile selezionare l'operazione da eseguire su una funzione nella variabile *x*. Il limite richiede un ulteriore input corrispondente al punto di accumulazione x_0. L'integrale richiede come ulteriori input gli estremi di integrazione. La derivata non chiede alcun input ulteriore. Inoltre nei parametri da inserire per le diverse operazione supporta ed espressioni e.g.(3+2).

Combinatorics Calculator

L'unico ordine consentito dalla calcolatrice per svolgere le operazioni è

primo operando -> *operazione unaria* -> *uguale*

primo operando -> *operazione binaria* -> *secondo operando* -> *uguale*

Ad ogni pulsante operazione è associato un pulsante '*?*' dotato di un tooltip che mostra brevemente il metodo algebrico dietro all'operazione combinatoria e se cliccato mostra sotto alla calcolatrice un'area di testo in cui è riportato un esempio di come è possibile utilizzare l'operazione combinatoria (l'area di testo scompare se viene premuto lo stesso pulsante '*?*' o se viene premuto il pulsante '=').

Appendice B
Esercitazioni di laboratorio

Roberto Lepore

Riccardo Tassinari

Riccardo Alni

Alessandro Agosta

Marco Pesic