Corso di Laurea in Ingegneria e Scienze Informatiche

Planted

(OOP15 - Process Report)

Ashley Caselli (ashley.caselli@studio.unibo.it)
Matricola 635153

15 maggio 2016

Indice

1 Introduzione

Questo documento descrive lo sviluppo software del progetto Planted, sviluppato per il corso di Programmazione ad Oggetti.

2 Requisiti

Il software mira alla costruzione di un editor di testo, che attraverso l'integrazione con la libreria PlantUML, renda possibile ai suoi utilizzatori la definizione di modelli e diagrammi UML. L'UML è un linguaggio di modellazione basato sul paradigma ad oggetti.

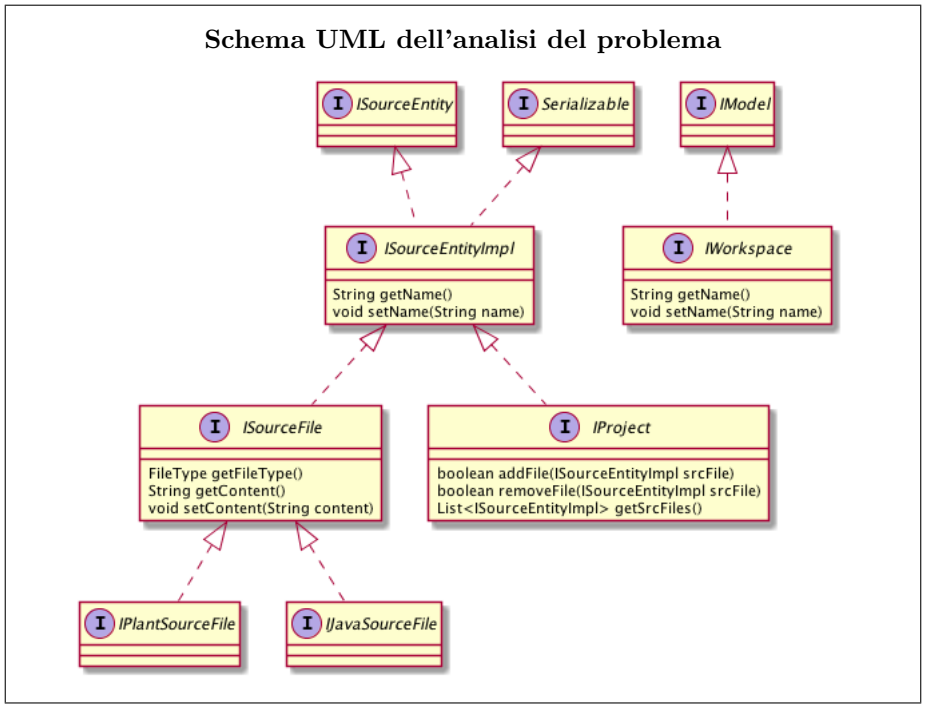
Requisiti concordati

- Il suddetto software dovrà essere un semplice editor di testo che permetterà all'utente di scrivere codice conforme al linguaggio usato dalla libreria PlantUML (`http://plantuml.com/`), e mediante l'integrazione con essa, supportare la visualizzazione real time del modello che si sta creando.
- Il modello dovrà poter essere esportato in un formato grafico (es: png).
- Nel caso in cui si stia definendo un modello UML delle classi, dovrà permettere all'utente non solo di esportare il modello in formato grafico (questa funzione dovrà essere disponibile per ogni modello o diagramma), ma anche di generare codice sorgente a partire dal modello delle classi che si sta definendo. L'utente quindi dovrà avere la possibilità di scegliere il linguaggio nel quale vuole che il codice sorgente sia scritto.
- Planted dovrà fornire all'utente la possibilità di importare file sorgenti esistenti al suo interno.
- Il modello UML delle classi potrà essere generato anche a partire da codice sorgente scritto in un linguaggio diverso da quello utilizzato dalla libreria PlantUML (es: codice scritto in linguaggio Java).

3 Analisi

Planted dovrà avere uno spazio di lavoro, detto **workspace (IWorkspace)**, all'interno del quale sarà possibilie gestire un insieme di entità sorgenti **ISourceEntity**.
Un progetto sarà un'entità sorgente e potrà contenere al suo interno più file sorgenti, anche di natura diversa, purchè contengano un linguaggio con una grammatica definita.
Un **progetto (IProject)** è quindi un contenitore di file sorgenti che possono essere scritti in linguaggi diversi tra loro.
Un **file sorgente (ISourceFile)** presente all'interno di un progetto rappresenta

perciò un semplice file di testo, caratterizzato da un'estensione, che ne definisce la natura del contenuto. Le 3 entità appena descritte sono rappresentate nel modello UML sottostante:



4 Design

Pattern

- **MVC** verrà utilizzato per definire l'architettura del sistema
- **Observer** verrà utilizzato ampiamente all'interno del sistema
- **Bridge** è stato utilizzato per la progettazione dell'entità **ISourceEntity**
- **Singleton** verrà utilizzato per la creazione di Model e Controller

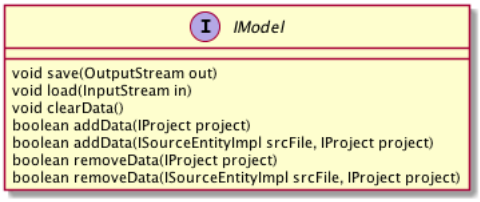
4.1 Architettura

Per lo sviluppo del progetto si è scelto di utilizzare il *pattern architetturale MVC*, creando una suddivisione logica dei componenti come segue:

- **Model** si occupa dei dati del sistema, in particolare fornisce funzionalità per la persistenza dei dati all'interno del sistema e fornisce accesso ai dati al controller.
- **View** si occupa di fornire all'utente la possibilità di interagire con il sistema (a livello grafico), mostrando l'editor di testo, l'albero dei progetti all'interno del workspace e tutte le possibili azioni che l'utente può fare (comandi che saranno passati al controller) con il controller.
- **Controller** contiene la business logic del sistema. Si occupa dell'interazione tra model e view, in particolare riceve i comandi che l'utente da al sistema attraverso la view e li attua, aggiornando in seguito sia model che view.

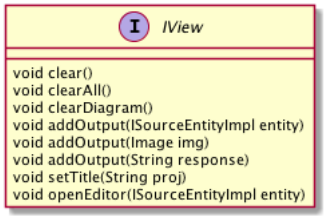
4.1.1 Model

Il livello di Model fornisce alla business logic, che è rappresentata dal livello denominato Controller, l'accesso ai dati del sistema.
L'entità **IModel** rappresenta il punto di accesso al livello di Model. Essa rappresenta quindi il vero e proprio livello di Model del sistema. L'entità che implementerà questa interfaccia dovrà utilizzare il pattern creazionale Singleton, per non permettere che esistano due istanze di essa all'interno del sistema. Le funzionalità che essa fornisce sono descritte dal diagramma UML seguente:



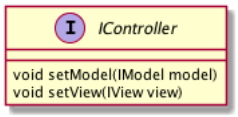
4.1.2 View

Il livello di View è ciò che l'utente vede e con il quale interagiragisce, inviando comandi al Controller.
L'entità **IView** rappresenta questo livello, e fornisce le funzionalità sotto descritte:

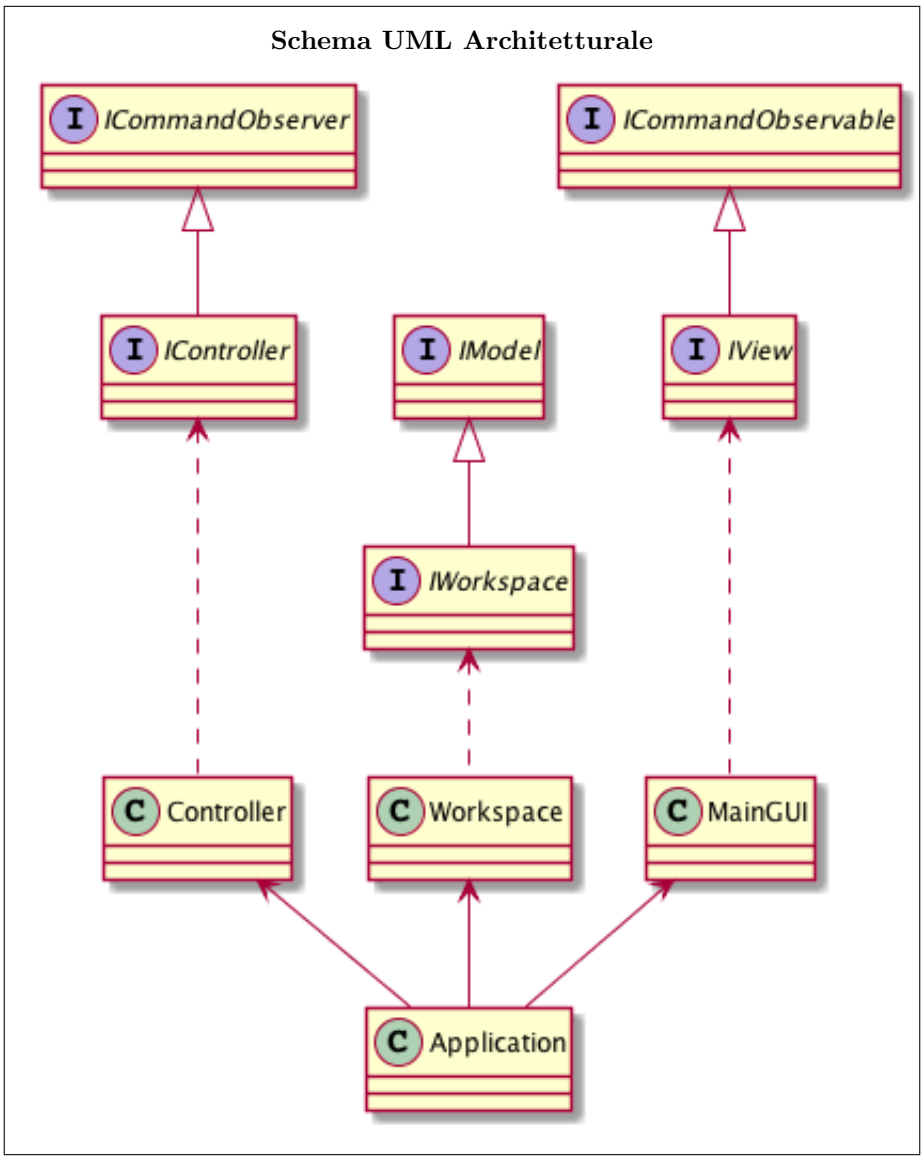


4.1.3 Controller

Il livello Controller è il vero e proprio core del sistema. Esso rappresenta la business logic del sistema, ciò che collega tra loro i due livelli sopra citati: Model e View. L'entità che implementerà questa interfaccia dovrà utilizzare il pattern creazionale Singleton, per non permettere che esistano due istanze di essa all'interno del sistema.
L'entità che rappresenta tale livello è chiamata **IController**:



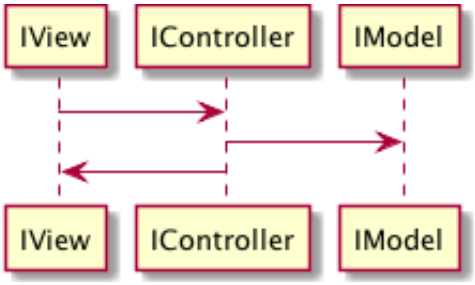
Tutta la logica dell'applicazione sarà incapsulata nel componente che implementerà l'interfaccia di IController. In questo modo la logica del sistema rimane indipendente dal modello dei dati (Model) e dalla loro rappresentazione (View).



Interazione dei livelli logici

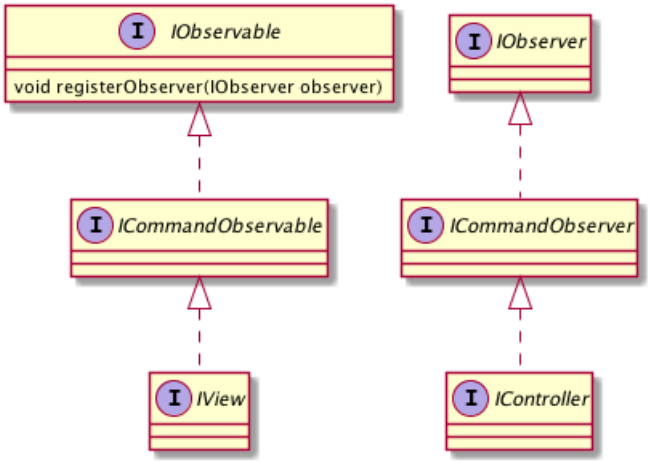
Vengono ora descritte le funzionalità del sistema attraverso alcuni diagrammi di interazione, in particolare il modo con cui i tre livelli descritti in precedenza

interagiscono tra loro.
La modalità di interazione tra i tre livelli è la seguente:

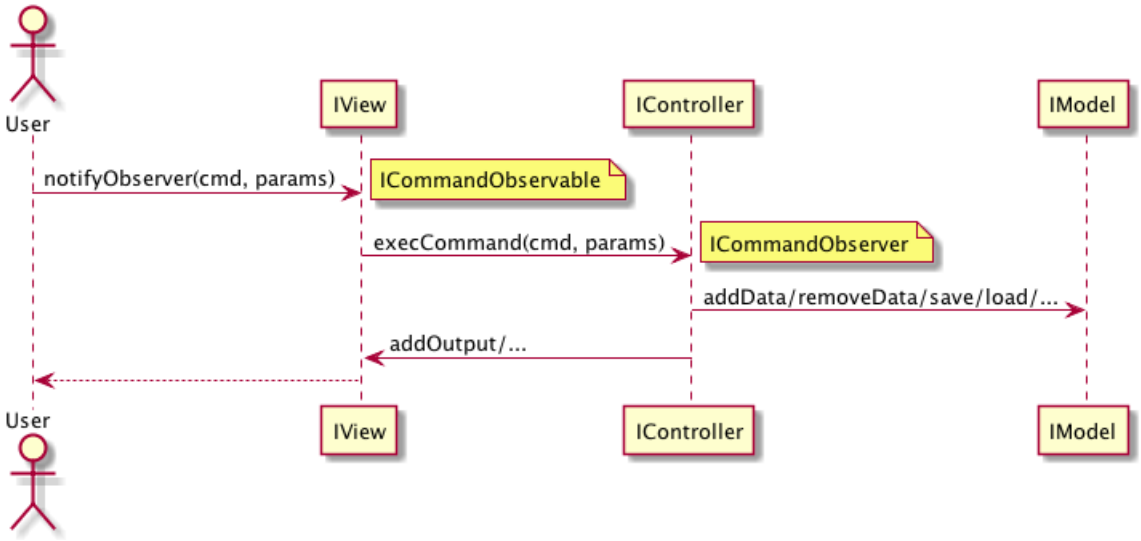


La comunicazione tra il livello View ed il livello Controller avviene attraverso il *pattern observer*. In particolare il Controller agisce da osservatore e la View da sorgente di eventi.

Come si può notare dal diagramma sottostante le due entità Controller e View implementano rispettivamente l'interfaccia *ICommandObserver* ed *ICommandObservable* che a loro volta estendono le interfacce *IObservable* ed *IObserver*.



Le due interfacce *ICommandObserver* ed *ICommandObservable* rappresentano due entità in grado di inviare ed ascoltare comandi, ed interagiscono tra loro tramite il *pattern observer*. L'interazione tra loro è alla base dell'interazione che c'è tra View e Controller.

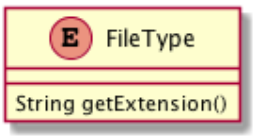


Esempio - Interazione tra i livelli mediante pattern observer per l'esecuzione di comandi

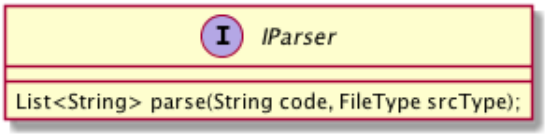
4.2 Design Dettagliato

Di seguito saranno illustrate ulteriori entità che andranno ad ampliare il modello del dominio già definito in fase di analisi.

Come già detto ogni file sorgente sarà caratterizzato da un'estensione che indentifica la natura del suo contenuto. Questa estensione è descritta dal **FileType**.



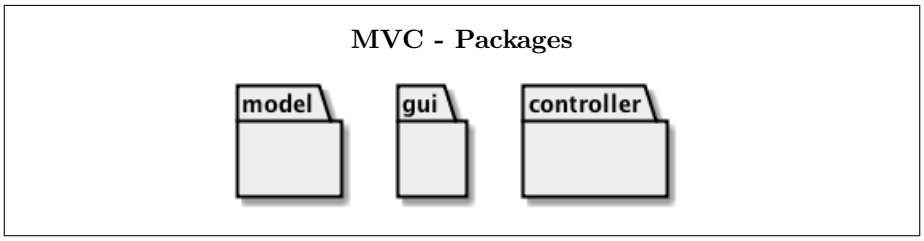
Il sistema inoltre avrà la capacità di interpretare il contenuto testuale di un file per poter generare codice sorgente nel linguaggio scelto. Per soddisfare tale requisito è necessario definire un'entità denominata **parser (IParser)**.



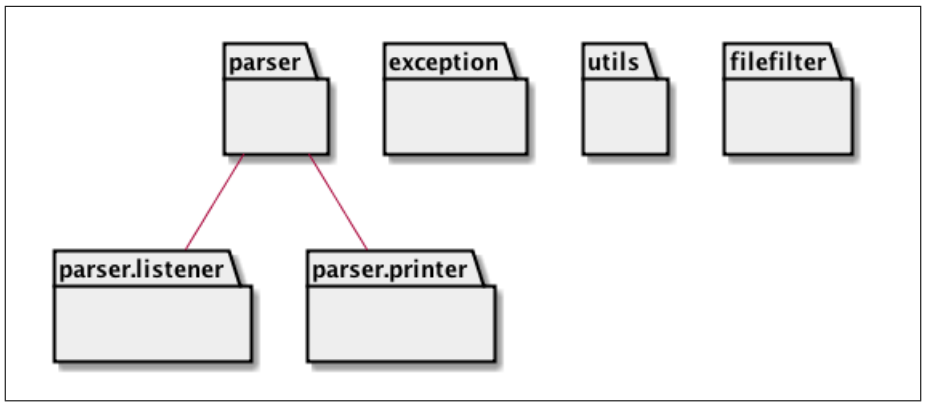
L'interazione tra il livello di View e Controller, come già detto, è basata sul *pattern observer*. Tale interazione avverrà attraverso lo scambio di comandi, che sono definiti all'interno dell'entità **Command**.

4.2.1 Suddivisione dei package

Il sistema è suddiviso in package, ognuno dei quali conterrà una parte di sistema.
Ogni package avrà al suo interno tutto ciò che si occupa della parte del sistema per cui è stato pensato.
Il primo package descritto è chiamato **interfaces**. Al suo interno si troveranno le interfacce delle entità che compongono il modello del dominio. Queste interfacce saranno scritte nel linguaggio che si sceglierà per lo sviluppo del sistema software e dipendono esclusivamente dal meta modello creato per il sistema. Al suo interno non ci deve essere nessuna implementazione e logica del sistema.
La logica del sistema è contenuta nel package definito **controller,** che conterrà esclusivamente la business logic.
Il package **model** è stato pensato per contenere le implementazioni delle entità del modello del dominio che riguardano il livello logico di Model dell'architettura MVC. È bene suddividere i package del sistema in modo che salti subito all'occhio di quale livello un'entità fa parte ma sopratutto per avere una suddivisione più marcata tra le entità.
Per contenere il livello di View è stato creato il package **gui.**



Inoltre sono stati definiti ulteriori package per avere una maggiore suddivisione logica della componenti del sistema. Per questo motivo sono stati creati i package **parser**, **parser.listener** e **parser.printer** che, come si può intuire dal nome, conterranno tutto ciò che riguarda la funzionalità di parsing, che verrà descritta in seguito. Infine sono stati definiti altri package utili per la futura realizzazione del sistema: il package **utils** che conterrà al suo interno le utilità del sistema e le impostazioni; il package **exception**, al cui interno si troveranno le eccezioni del sistema; ed il package **filefilter** che sarà utile per la gestione dei file di progetto.



5 Sviluppo

La fase di sviluppo del sistema software richiede innanzitutto la scelta del linguaggio di programmazione con il quale verrà scritto il sistema software. Essendo tale progetto sviluppato per il corso di OOP15-16, la scelta ricade senza alcun dubbio sull'utilizzo di Java (versione 8). Questa fase deve iniziare tassativamente solo dopo che si sono ultimate le fasi di analisi (dei requisiti e del problema) ed è stato definito il design dettagliato del sistema.

5.1 Testing automatizzato

La fase di testing è necessaria in qualunque sistema software. Tale fase consente di delineare le funzionalità, in modo che non ci siano modifiche di base in aggiornamenti futuri che comprometterebbero tutto il sistema.

Sono stati effettuati test case automatizzati su alcune entità del modello, in particolare:

- **ModelTest** testing delle funzionalità dell'entità IModel
- **ProjectTest** testing delle funzionalità dell'entità IProject
- **WorkspaceTest** testing delle funzionalità dell'entità IWorkspace
- **PlantSourceFileTest** testing delle funzionalità dell'entità IPlantSourceFile
- **JavaSourceFileTest** testing delle funzionalità dell'entità IJavaSourceFile

Con gli ultimi due test case descritti, si sono testate anche le funzionalità dell'entità **ISourceFile**.

5.2 Metodologia di lavoro

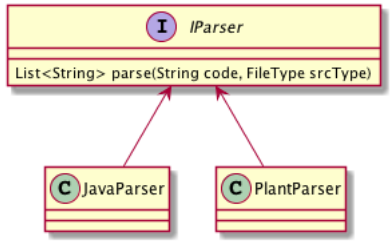
L'utilizzo di Java mi ha portato a basare lo sviluppo su un gestore di progetti Java e build automation. In questo modo l'implementazione del progetto non dipenderà da nessun IDE, ma solo ed esclusivamente dai file di configurazione di tale gestore di progetti.
In questa fase ho iniziato col creare le implementazioni delle entità che definiscono il modello del dominio. Seguendo questa politica, ho successivamente implementato tutte le componenti essenziali per l'architettura MVC in modo da avere un core dell'applicazione pronto per essere ampliato con le relative funzionalità.
Implementare le componenti essenziali per l'architettura MVC consiste nello scrivere una prima, anche se scarna, business logic del sistema, che sarà in seguito ampliata per soddisfare tutte le funzionalità che il sistema dovrà avere. Per fare ciò bisogna definire un comportamento per ogni metodo che si era definito nel modello del Controller.

Molto importante nella fase di sviluppo è la creazione di classi di supporto e per le configurazioni del sistema. In questo modo risulta più semplice la manutenzione del sistema finale e c'è una maggiore divisione logica dei compiti. In particolare ho creato una classe, chiamata **SysKB**, che ha come unico scopo il raccoglimento di tutti quei parametri usati nel sistema come predefiniti ma che possono essere oggetto di future modifiche non preventivate in questo progetto (es: path delle cartelle dove si serializzeranno i dati). Oltre alla buona organizzazione ed ad una semplificazione di manutenzione, questo permette anche di estendere alcune funzionalità del sistema e modificarle senza grossi cambiamenti.

5.2.1 Parsing

Una funzionalità del sistema consiste nel generare automaticamente codice sorgente scritto in un linguaggio scelto a partire dal modello UML delle classi e viceversa. Per soddisfare questa funzionalità c'è la necessità di sviluppare all'interno del sistema un'entità in grado di prendere in input uno stream di dati e fornire in output un diverso stream di dati ma strettamente correlato a quello fornitogli in ingresso. Questà entità deve quindi essere in grado fare un'analisi lessicale e sintattica dello stream passatogli in input, e fornire in output uno stream conforme lessicalmente e sintatticamente a ciò che viene richiesto (sia esso codice Java, codice per la libreria PlantUML, o altro codice con una grammatica definita).
Il comportamento descritto sopra viene detto *parsing,* ovvero l'insieme dell'analisi lessicale ed analisi sintattica di uno stream di dati.
Tale funzionalità può essere sviluppata in più modi.
Il primo modo, nonchè il più semplice da costruire senza avere nessuna conoscenza avanzata sulla costruzione di un parser, è quello di sviluppare alcune classi che fungono da riconoscitori di grammatica e "convertitori" dello stream di dati passatogli in ingresso verso un nuovo stream di dati il cui contenuto

dovrà essere conforme al linguaggio richiesto. In questo modo ho costruito due classi in grado di prendere in ingresso uno stream di dati (ed il tipo del file, che indica la natura dello stream), analizzarne il contenuto e creare uno stream di dati in uscita che conterrà lo stream di dati preso in ingresso ma convertito verso il linguaggio scelto.



In questo modo se si vogliono aggiungere uno o più linguaggi a quelli che il sistema è già in grado di parserizzare, bisogna creare un'entità che estende l'interfaccia **IParser** ed al suo interno costruire un riconoscitore per la grammatica usata dal linguaggio che si vuole aggiungere.

La costruzione di un riconoscitore di grammatica in questo modo però ha molti limiti, che si evidenziano all'aumentare dei possibili token che la grammatica utilizza. Per questo motivo (e non solo) ho deciso di sviluppare la funzionalità di Parsing attraverso l'utilizzo di un generatore di parser, e dopo essermi documentato su come si costruisce un "vero" parser ho optato per l'utilizzo di ANTLR v4.

5.2.2 ANTLR - Definizione delle Grammatiche

ANTLR è un generatore di parser per leggere, processare e/o tradurre testo strutturato. Il suo funzionamento si basa sulla definizione di una grammatica, ed a partire da essa il tool genera le classi necessarie (lexer, parser, listener) per eseguire le varie operazioni sullo stream di input.

Utilizzando questo tool ho scritto due grammatiche:

- **Plant** grammatica di base per il linguaggio usato dalla libreria PlantUML

```
grammar Plant;

// Parser Rules

plantDeclaration : '@startuml'
    classDeclaration* '@enduml';
classDeclaration : 'class' TEXT classBody;
classBody : '{' classBodyDeclaration* '}';
```

```
classBodyDeclaration: fieldDeclaration |
    methodDeclaration ;
fieldDeclaration : modifierDeclaration?
    nameDeclaration ':' typeDeclaration;
methodDeclaration: modifierDeclaration?
    returnTypeMethodDeclaration?
    methodNameDeclaration paramDeclaration  ;
returnTypeMethodDeclaration: 'int' | 'String'
     | 'void';
paramDeclaration: '(' paramBodyDeclaration*
    ')';
firstParamBodyDeclaration:
    typeParamDeclaration TEXT;
paramBodyDeclaration:
    firstParamBodyDeclaration
    otherParamBodyDeclaration*;
otherParamBodyDeclaration: ','
    typeParamDeclaration TEXT;
typeParamDeclaration: 'int' | 'String';
typeDeclaration: 'int' | 'String';
modifierDeclaration :  MODIFIER;
nameDeclaration : TEXT;
methodNameDeclaration : TEXT;

// Lexer Rules

TEXT : ('a'..'z' | 'A'..'Z')+ ;
MODIFIER : '+' | '-' | '~' | '#';
WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\
    u000C' )+ -> skip ;
```

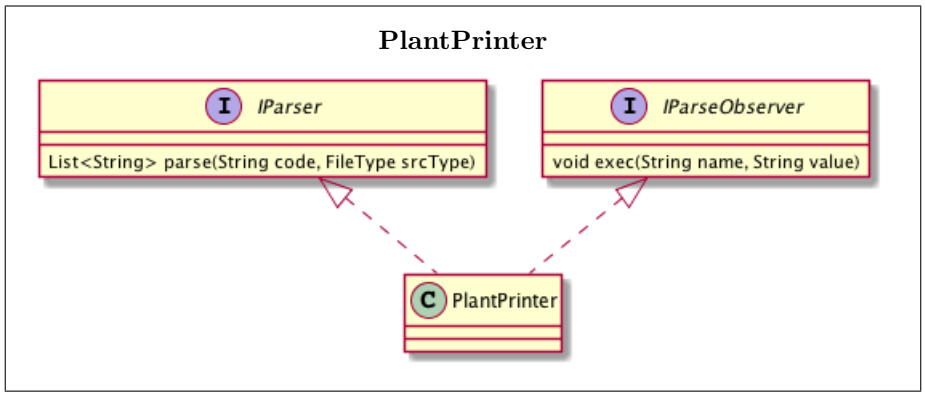
- **SimpleJava** grammatica di base per il linguaggio Java

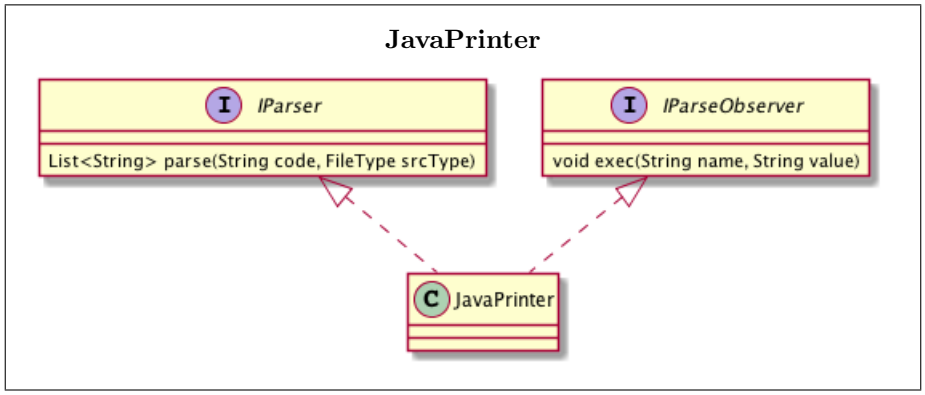
```
grammar SimpleJava;

// Parser Rules

classDeclaration : modifier 'class' TEXT
    classBody;
classBody: '{' classBodyDeclaration* '}';
classBodyDeclaration: fieldDeclaration |
    methodDeclaration;
methodDeclaration: modifier? methodReturnType
     methodName paramDeclaration
    methodBodyDeclaration;
methodName: TEXT;
```

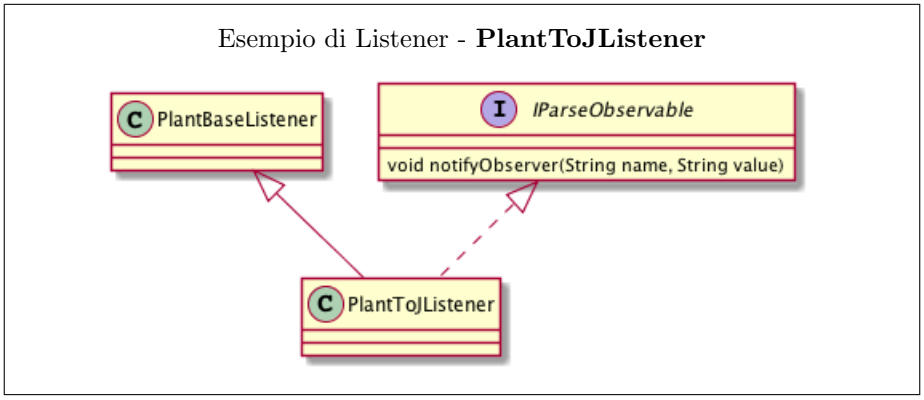
```
methodBodyDeclaration: '{' methodBody '}';
methodBody: ((TEXT | '.' | '=')* ';')*;
paramDeclaration:'(' paramBodyDeclaration* ')'
    ';
paramBodyDeclaration:
    firstParamBodyDeclaration
    otherParamBodyDeclaration*;
firstParamBodyDeclaration: paramType
    paramName;
otherParamBodyDeclaration: ',' paramType
    paramName;
paramName : TEXT;
fieldDeclaration: modifier? fieldType
    fieldName fieldInitializion? ';';
fieldInitializion: '=' TEXT;
fieldName: TEXT;
modifier
  :      'public'
  |      'protected'
  |      'private'
  ;
methodReturnType: 'void' | type;
type: 'int' | 'String' | 'boolean';
fieldType: type;
paramType: type;

// Lexer Rules

TEXT : ('a'..'z' | 'A'..'Z')+ ;
WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\
    u000C' )+ -> skip ;
```

Le grammatiche di cui sopra, non comprendono tutta la grammatica del linguaggio Java e nemmeno tutta quella del linguaggio usato dalla libreria PlantUML, quindi non ho riscritto un parser per questi due linguaggi, ma ho solamente scritto due grammatiche che comprendono i concetti base dei due linguaggi.
A partire da queste grammatiche ho utilizzato il tool per generare il necessario per effettuare il parsing.
Come nella prima soluzione adottata per la costruzione del parser, ho costruito un'entità in grado di convertire lo stream di dati preso in ingresso in uno stream di dati di natura diversa. L'utilizzo di ANTLR, non solo è ottimo per quanto riguarda la costruzione del parser, ma anche per quanto riguarda la costruzione di quest'ultima entità che viene creata in maniera molto semplice utilizzando i listener generati automaticamente da ANTLR.
I **listener** di ANTLR seguono il funzionamento definito dal *pattern observer*, e permettono di definire un comportamento specifico ogni qualvolta si entra ed

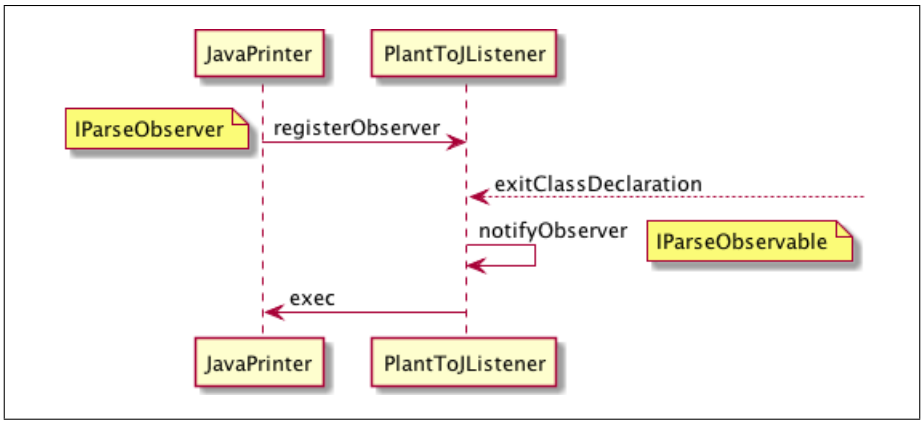
esce da una regola definita nella grammatica. In questo caso, implementando i listener delle relative grammatiche, ho costruito uno stream di dati che, non appena conclusasi l'analisi dello stream in input, conterrà l'output finale desiderato. Il listener relativo per una grammatica viene creato runtime da un'entità che implementa l'interfaccia *IParser*, che seleziona quale listener creare in base al tipo dello stream di dati che gli viene passato come input.







La creazione dello stream dati di output avviene mediante l'utilizzo del *pattern observer* tra Printer e Listener. Le due entità fungono rispettivamente da *observer* ed *observable*. Lo stream di dati in output viene creato all'interno del Listener che, non appena ne ha completato la creazione, notifica l'ascoltatore (Printer).
Un esempio del funzionamento di tale sistema è spiegato nel seguente diagramma di interazione:



5.3 Note di Sviluppo

La fase di sviluppo del sistema software è avvenuta attraverso l'utilizzo di vari tool e librerie.

Il codice è stato scritto in Java 8, e salvato mediante l'utilizzo di *git* su GitLab (`https://gitlab.com/`). Successivamente, come richiesto dal prof. Mirko Viroli, il codice è stato importato su BitBucket (`https://bitbucket.org/ashleycaselli/oop15-planted`) all'interno di un progetto appositamente crea-

to utilizzando *mercurial* per il controllo di versione.
Come descritto in precedenza ho usato un gestore di progetti, chiamato Maven (`http://maven.apache.org/`), integrato all'interno dell'IDE di sviluppo Eclipse (`https://eclipse.org/`).
Per la fase di testing ho utilizzato il tool JUnit 4 (`http://junit.org/junit4/`).
Ho utilizzato il generatore di parser ANTLR v4 (`http://www.antlr.org/`) per implementare la funzionalità di parsing, ed infine ho utilizzato la libreria PlantUML (`http://plantuml.com/`) per effettuare l'esportazione dei grafici/diagrammi in formato grafico.

NB: utilizzando Maven come gestore del progetto, esso non potrà essere importato direttamente in Eclipse come "progetto Eclipse".

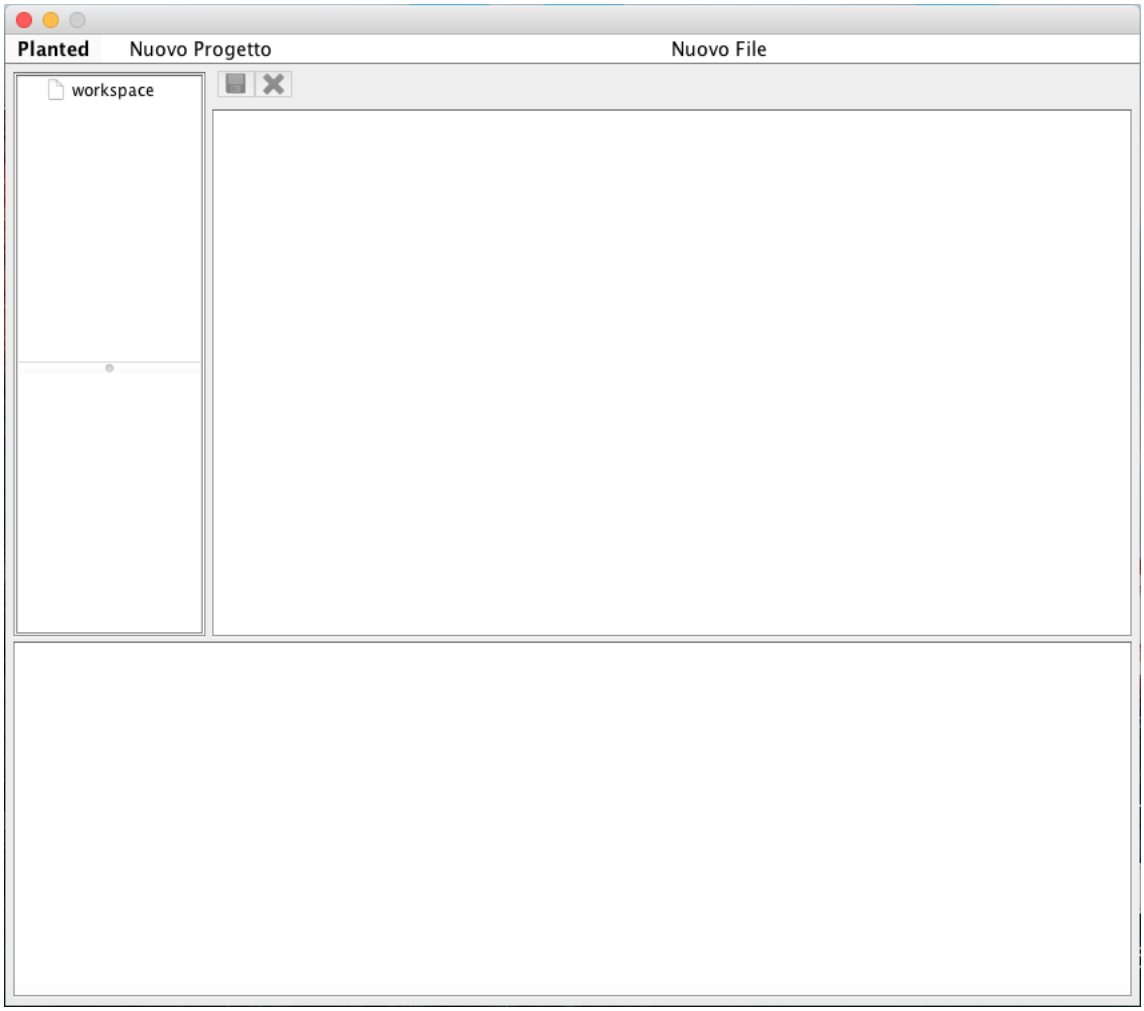
6 Commenti finali

Non sono pienamente soddisfatto del risultato finale ottenuto anche se sono state implementate tutte le funzionalità di base del sistema. Ciò che non mi rende completamente soddisfatto del lavoro svolto è la progettazione del sistema. In particolare avrei voluto che la definizione del modello del dominio fosse ancora più estendibile per modifiche future.
Nonostante ciò, vedo ampi margini di estensione nel progetto. Le grammatiche dei linguaggi attuali possono essere ampliate, e possono essere supportati ulteriori linguaggi (es: PHP, C#). Grazie al disaccoppiamento Model-View-Controller, sarebbe opportuno modificare in blocco la View, costruendo un ambiente di modellazione e sviluppo più simile ad un IDE.

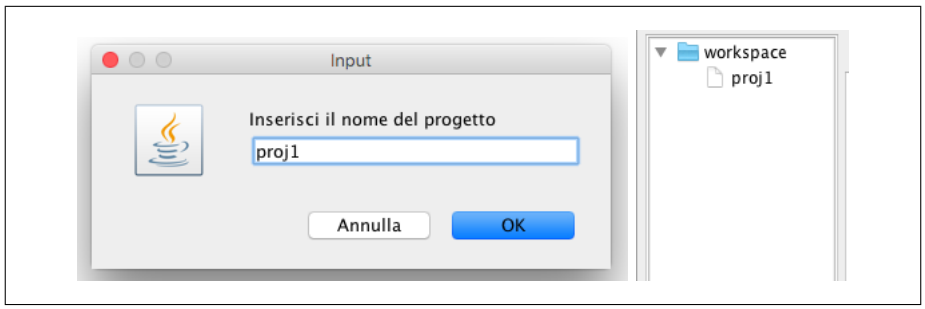
Sono invece molto soddisfatto di aver potuto ampliare le mie conoscenze di alcuni tool mai studiati nel corso della mia carriera universitaria, ma che ora ritengo di fondamentale importanza per lo sviluppo di un sistema software.
Inoltre, e non ultimo per importanza, ho avuto la possibilità di guardare avanti nella mia carriera universitaria ed anticipare alcuni argomenti che verranno studiati nel corso di laurea magistrale (utilizzo di ANTLR e costruzione di grammatiche/linguaggi).

Appendice A Guida Utente

La prima volta che si esegue l'applicazione Planted sarà visualizzata questa schermata, con il pannello dei progetti vuoto.

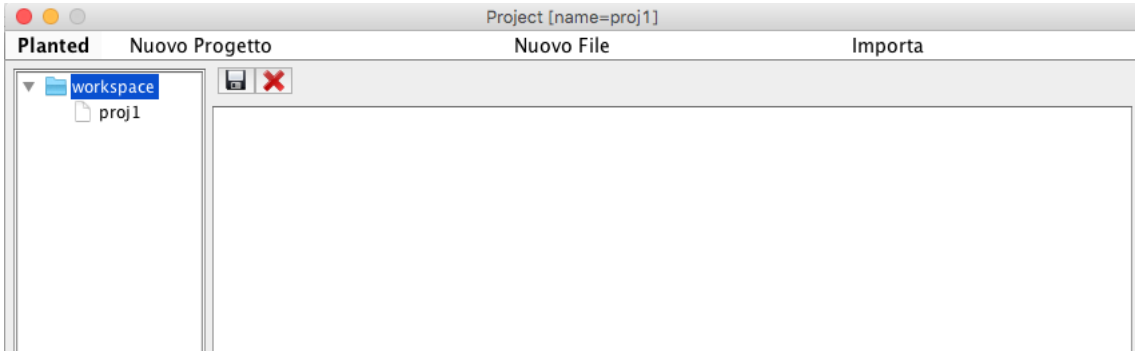


Il primo passo da fare è creare un progetto attraverso la voce *Nuovo Progetto* del Menù. Alla creazione del progetto verrà richiesto il nome, ed una volta inserito, il progetto, sarà visualizzato nel pannello dei progetti a sinistra.



Appendice A - Figura 1: Creazione di un nuovo progetto

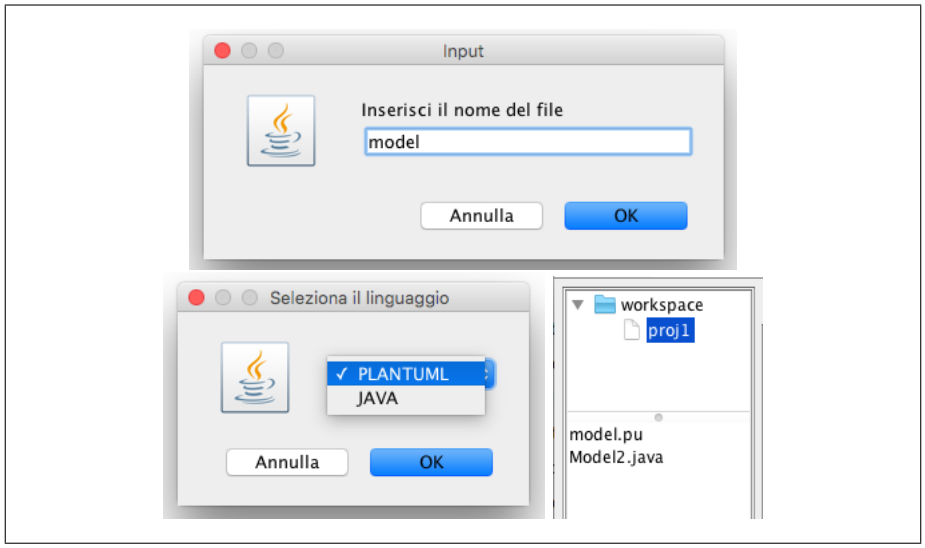
A questo punto, abbiamo creato il nostro primo progetto. L'alto della finestra ci indica il progetto corrente, su cui stiamo lavorando. Ogni volta che si crea un nuovo progetto, esso diventerà il progetto corrente. Nel caso in cui si avessero più di due progetti, si può selezionare il progetto che si vuole fare diventare il corrente semplicemente con un click su di esso dal pannello dei progetti posto a sinistra.



Appendice A - Figura 2: Progetto corrente

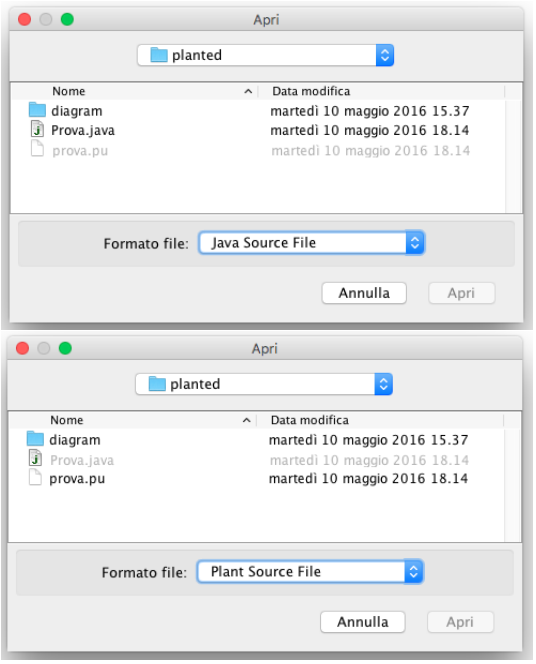
Ora non ci resta che inserire il primo file sorgente all'interno del progetto precedentemente creato. L'inserimento può avvenire in due modi:

- **Creando un nuovo file attraverso l'apposita voce *Nuovo File* del Menù.** Utilizzando questa modalità, dopo aver inserito il nome, deve essere scelto il tipo del file sorgente. Una volta creato il file, il sistema automaticamente lo inserirà, con la sua relativa estensione che dipende dalla scelta del tipo, all'interno del pannello progetti, nella sezione relativa ai file del progetto.



Appendice A - Figura 3: Creazione di un nuovo file sorgente

- **Importando un file esistente presente all'interno del nostro computer, utilizzando la voce *Importa* del Menù.** Utilizzando questa modalità si può scegliere il file esistente da importare all'interno del progetto attraverso il File Chooser. Esso da la possibilità di filtrare i file a seconda della loro natura.



Appendice A - Figura 4: Import di un file sorgente esistente

Planted permette all'utente di scrivere codice attraverso l'editor messo a disposizione, predisponendo un ambiente diverso per ogni tipo di file sorgente che viene aperto in esso e funzionalità diverse. Vediamo come sia possibile per esempio, quando si apre un file con estensione ".pu", effettuare sia la generazione di codice che l'esportazione grafica. Tali funzionalità non sono tutte presenti all'apertura di un file con estensione ".java", col quale è possibile solo utilizzare la funzionalità per generare il codice.
La funzionalità per generare il codice permette all'utente di scegliere il linguaggio con il quale il codice deve essere scritto, a prescindere dalla natura del file aperto nell'editor. In questo modo è possibile convertire file ".java" in file ".pu" e viceversa.

Inoltre l'editor presenta un pannello (posto in basso) di visualizzazione realtime del modello grafico, che è utilizzato esclusivamente quando si sta scrivendo o si effettuano modifiche su un file ".pu". Tale pannello contiene l'anteprima grafica di ciò che verrà salvato con il comando di "Export Diagram" in un file con formato grafico (es: ".png").

Le altre funzionalità di base come salva o elimina, non descritte in questa guida, permettono di effettuare il salvataggio del file che si sta editando e la rimozione

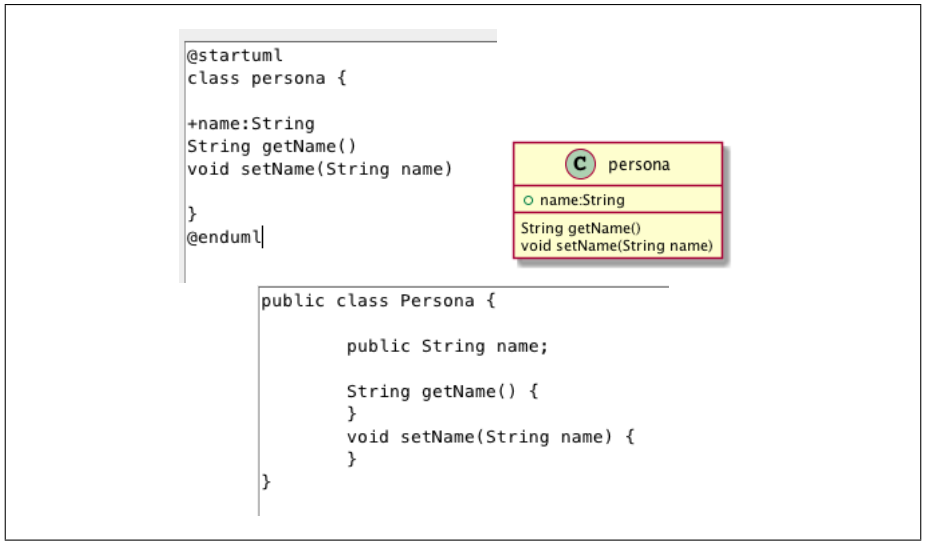
dell'elemento selezionato. Se l'elemento selezionato è un progetto, la rimozione comporta anche la rimozione di tutti i file presenti all'interno di esso.

Sintassi - Esempi di utilizzo

La sintassi del codice supportato dall'editor per la descrizione del modello delle classi attraverso il linguaggio PlantUML è la seguente:

- **classe**: "class" nome "" contenuto ""
- **modificatore**: è sempre opzionale, può avere valore +, -, # che indicano rispettivamente *public, private, protected*
- definizione di un **campo**: [modificatore] nome : tipo
- definizione di un **metodo**: [modificatore] [tipo] nome "("[parametri]")", il tipo è opzionale, se non esiste viene considerato *void*
- definizione dei **parametri**: tipo nome

Esempio - Generazione classe Java a partire dal modello



Esempio - Generazione del modello a partire da classe Java

```
public class Complex {
        int imm;
        int real;

        public int getReal() {
                return this.real;
        }
        public int getImm() {
                return this.imm;
        }
}
```

```
@startuml
class Complex {
imm:int
real:int
+int getReal()
+int getImm()

}
@enduml
```