Design Manual

JUnit Test Generator

I.T. Carlow
Software Engineering
Bachelor Degree (With Honours)

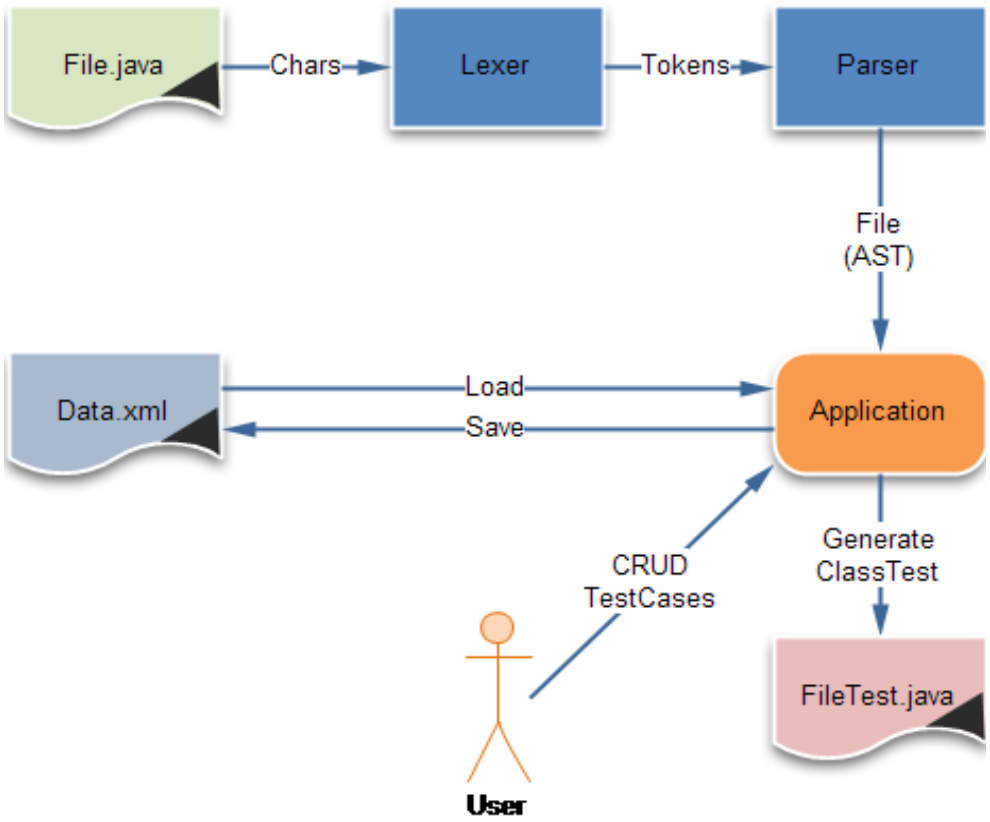
Sergio Alcocer Vázquez
C00132732

Table of Contents

1. Introduction

This is the Design manual of the JUnit Test Generator. In this document is displayed some of the reasons of some decisions made, explain briefly the structure of the Parser, including the Tokens used, the grammar, etc. To help understanding, there are some class diagrams.

A simple overview of the application would be as shown in the following image



2. Decisions made – reasons

As in any other project, several decisions have to be made. For this project, the first decision was to choose JUnit from all the XUnit frameworks (PyUnit, Cunit, etc) The main reasons for this choice were that the project was open in this area, and that the developers felt more comfortable with Java than with any other language, being platform free an added value.

Once this choice was made, the language to code the application was easy to choose, Java. There are several reasons to choose Java within the rest. First of all, it is the language used in JUnit. How that is an advantage?. Easy. It doesn't require the user to install any other osftware. If is able to make a program in Java, one of the first things he should worry about is to have JDK installed. Then, no further installs are needed to run our application. If instead of Java we chose C, for example, the user must be aware of the libraries used for the applications that are not included, the operating system and processor's bits to compile the sources, what implies that the user may need to install things he/she doesn't want for anything else.

It has been decided to use a hand-made parser and a Graphical User Interface. It might look as a waste of time to code all this stuff, but is not totally true.

2.1. Why a hand-made parser?

As long as parser generators might include general purpose code and need to define the grammar and the tokens, there is a little step left to make to have a hand-made one. And if any problem occurs, it would be easier to track-it.

2.2. Why a hand-made Graphical User Interface?

Once you have some experience coding GUI, create it is straight forward, a bit long, but just that. It is possible to save time coding by-hand, because Drag-n-Drop tools usually includes unnecessary code that changes the layout of the whole Frame, forcing the developers to waste time fixing the mess made. In the other hand, if you code it, you don't have that problem (you only add the code you need).

3. The parsing

The project has been split into two different, but complementary phases.

3.1. Lexical analysis

The aim of the lexical analysis is to translate form single characters, to Tokens, that has a real meaning. For this program, the Tokens used are the following:

- EOF
- IMPORT
- PACKAGE
- VALUE
- SC
- PUBLIC
- PRIVATE
- PROTECTED
- ABSTRACT
- STATIC
- FINAL
- OPEN_CURLY
- CLOSE_CURLY
- CLASS
- OPEN_BRACK
- CLOSE_BRACK

3.2. Syntax Analysis

The aim of the syntax analysis is to check if the tokens that forms the file are placed in the correct order. For example, there should be the same amount of OPEN_CURLY as CLOSE_CURLY.

To be able to do this, the following grammar has been used.

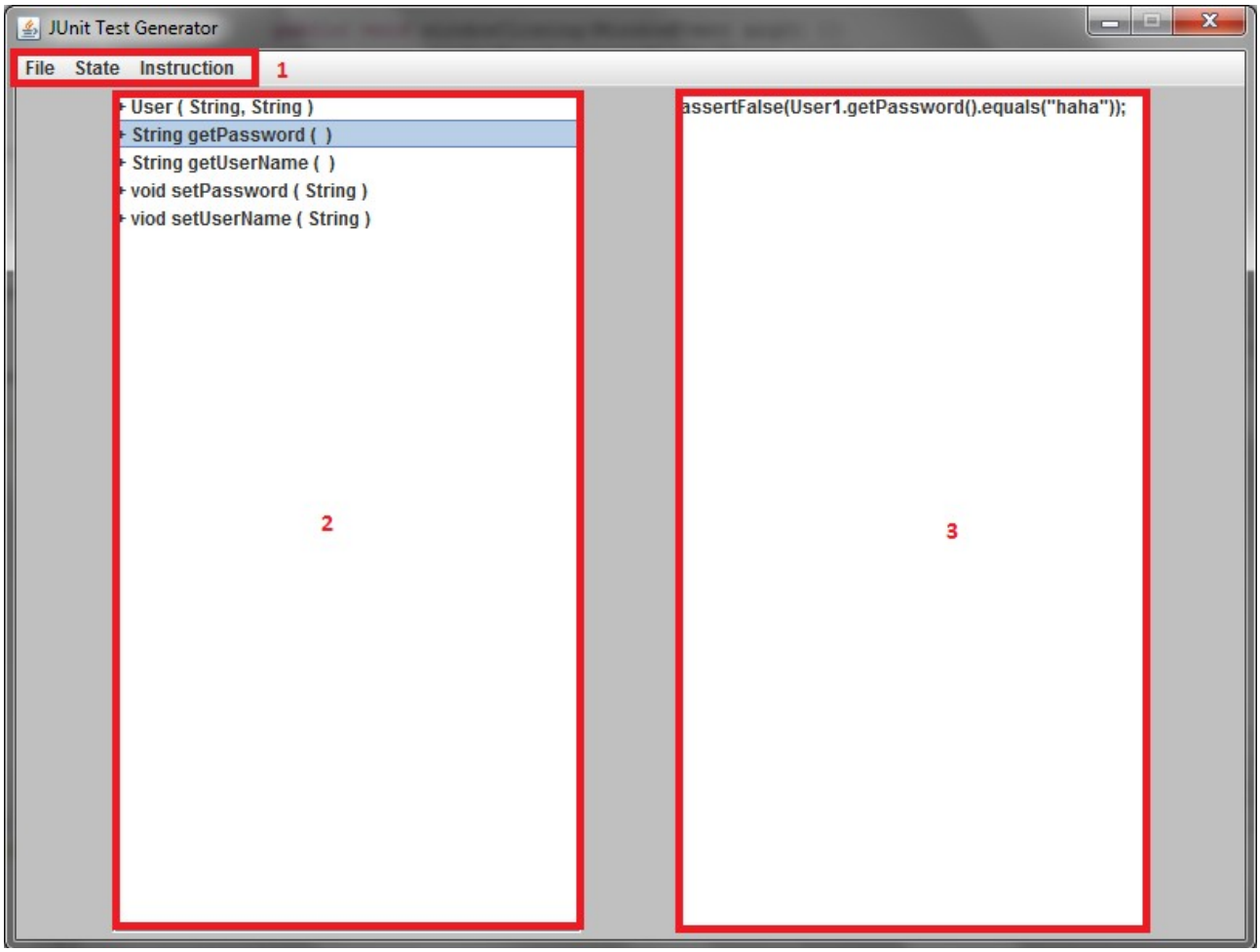
Note: the Production MethodBody is not displayed in the grammar because is a Token Burner, that burns until it gets the same number of CLOSE_CURLY as OPEN_CURLY.

File	→	Package	Imports	Class			
Package	→	**PACKAGE**	**VALUE**	**SC**			
Package	→	**^**					
Imports	→	**IMPORT**	**VALUE**	**SC**			
Imports	→	**^**					
Class	→	ClassModifier	**CLASS**	**VALUE**	**OPEN_CURLY**	ClassBody	**CLOSE_CURLY**
ClassModifier	→	**PUBLIC**					
ClassModifier	→	**^**					
ClassBody	→	Access	**VALUE**	ClassBody2	ClassBody		
ClassBody	→	**^**					
ClassBody2	→	ClassBody3					
ClassBody2	→	**VALUE**	ClassBody3				
ClassBody3	→	**SC**					
ClassBody3	→	**OPEN_BRACK**	Params	**CLOSE_BRACK**	**OPEN_CURLY**	BodyMethod	**CLOSE_CURLY**
Params	→	**VALUE**	**VALUE**	Params			
Params	→	**^**					
Access	→	**PRIVATE**					
Access	→	**PUBLIC**					

4. GUI

The Graphical User Interface is how the user interacts with the application. It is made by four different Screens.

4.1. Main Screen

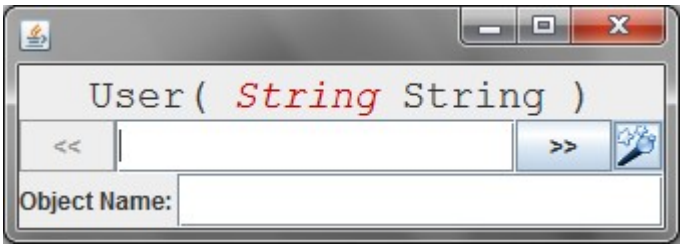


1 – Is the menu. This menu will change with the available options on each moment

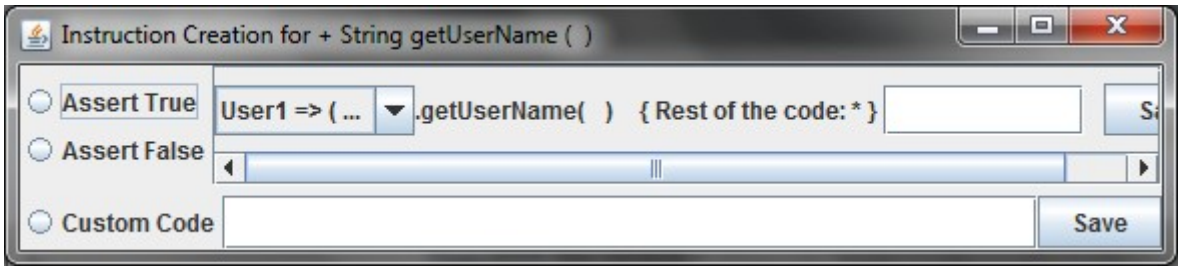
2 – Is the methods and constructors view. In here is where all the methods and constructors are shown.

3 – Actions and Objects view. In here is where all the Instructions (either assert or customized) and the objects created would be displayed.

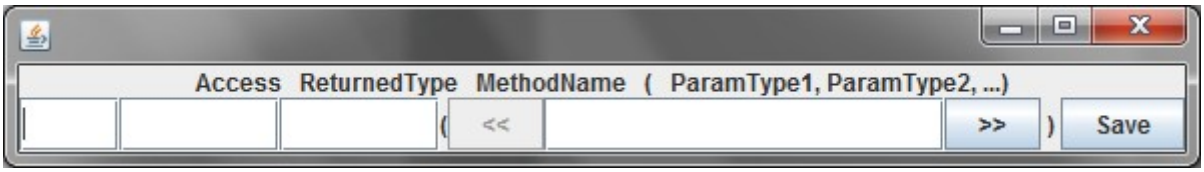
4.2. Object Creation



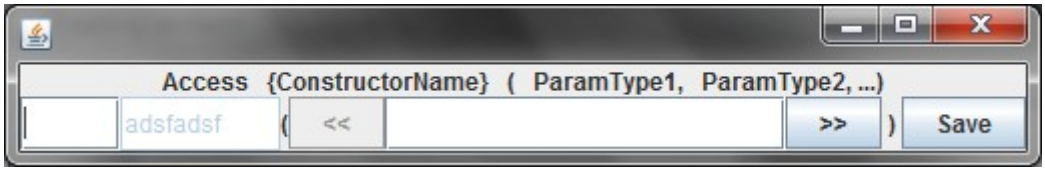
4.3. Instruction Creation



4.4. Method Creation

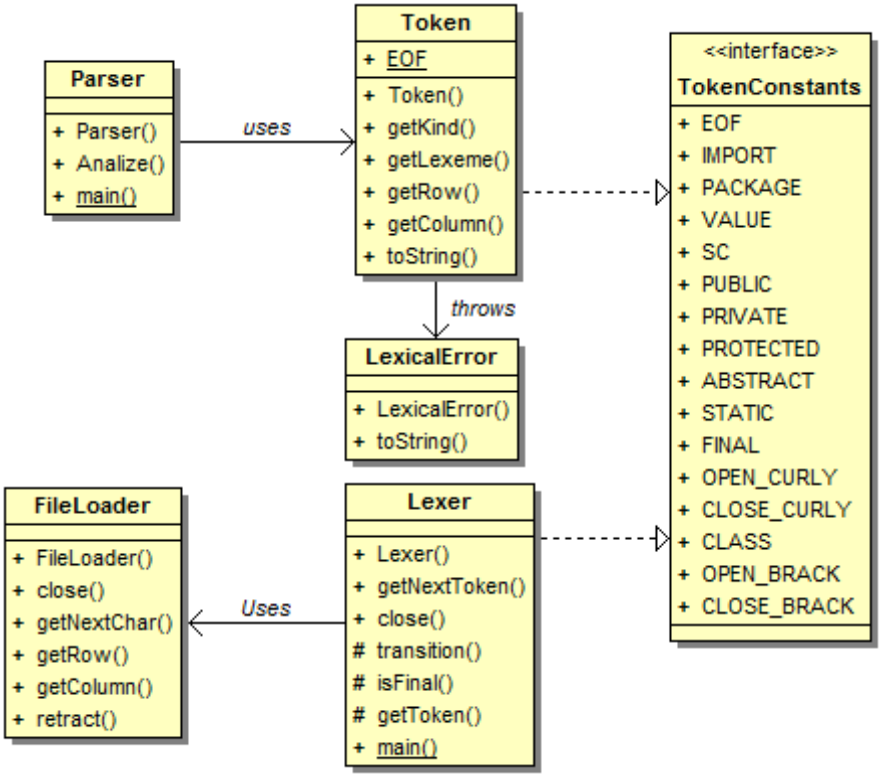


4.5. Constructor Creation



5. Class Diagrams

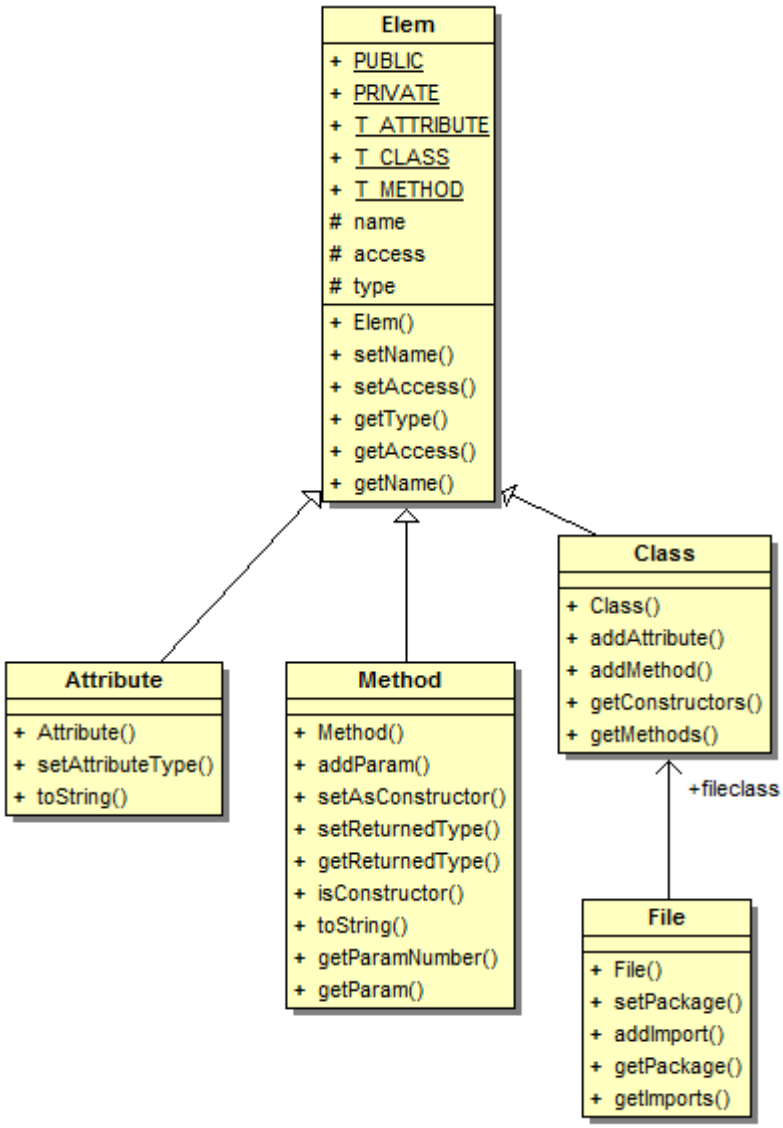
5.1. Analyzer



Notes:

- In **TokenConstants** there are more tokens than the ones used, but they could be used in future releases
- **FileLoader** is a class that simplifies the way to move through a file, allowing to get the next character, go back if needed and count the line and the row of the characters.
- **Lexer** gets tokens.
- **Token** has all the information needed of a certain token.

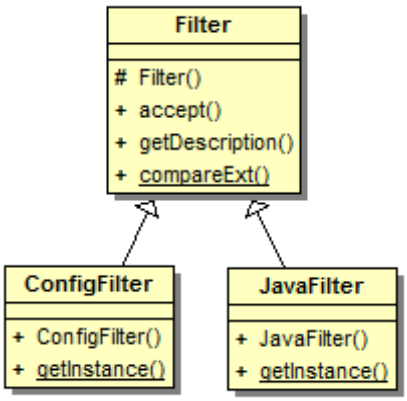
5.2. Abstract Syntax Tree



Notes:

- **File** only allows storing a single class per file
- **Method** provides a class to store methods information and also constructors.

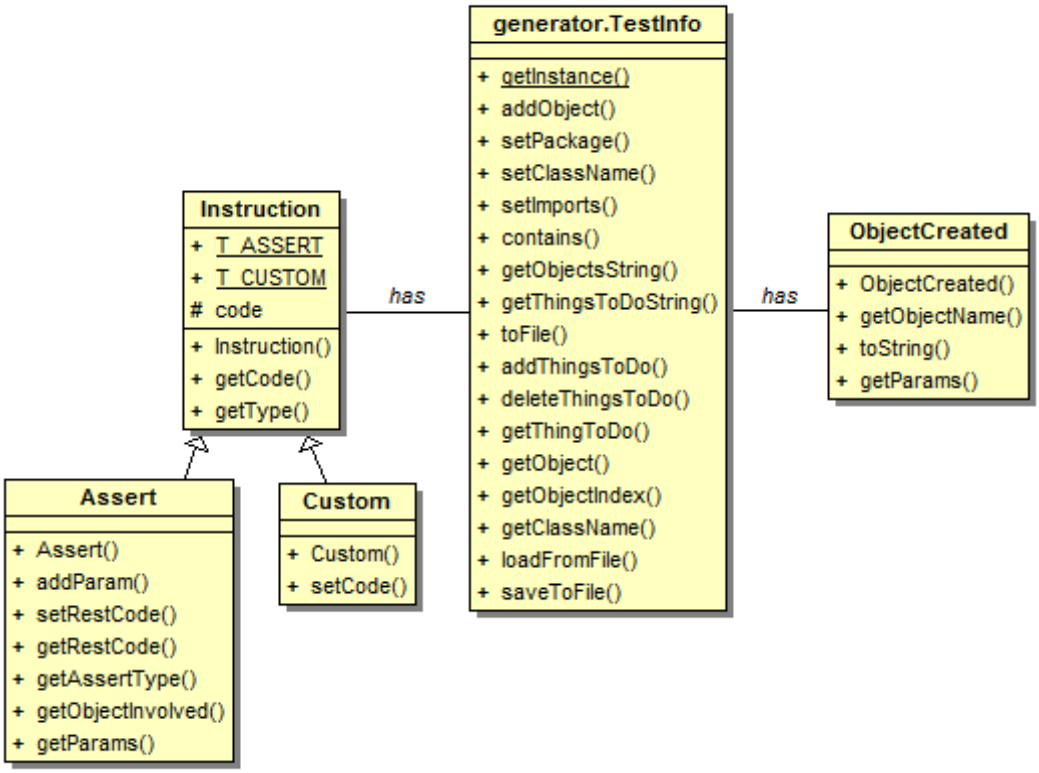
5.3. Filters



Note:

– This classes are needed to filter files in the **JfileChooser**

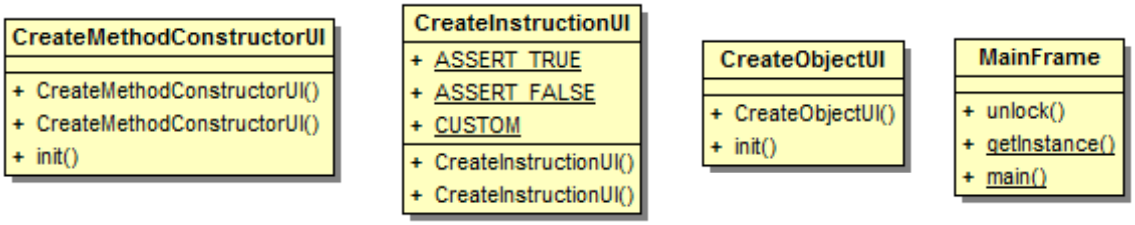
5.4. Generator



Notes:

- TestInfo stores:
  - an array of ObjectCreated
  - an array of imports
  - the test package
  - the class name
  - a dictionary of protype → Array of instructions
- TestInfo is the one that generates the Test file.
- Custom and Assert are the two kinds of instructions.

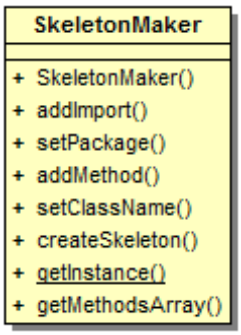
5.5. GUI



Notes:

- These are the user interfaces.
- **CreateMethodConstructorUI** provides two different and similar interfaces.
- **MainFrame.unlock()** unlocks the frame once a sub-frame has been closed.

5.6. Skeleton



Note:

- SkeletonMaker has all the information needed to create the skeleton of the class:
  - Array of methods and imports
  - Class package and class name

5.7. Randomizer

Note:

- Randomizer is a class that should be filled by the developer. In its bare version, it only has two methods:
  - getInstance() → static, in order to get a single instance
  - getRandom(String) → would call other functions created by the developer.

Final Note:

For more information about the classes and ints functionalities, please read the JavaDoc.