Project Report

JUnit Test Generator

I.T. Carlow

Software Engineering

Bachelor Degree (With Honours)

Sergio Alcocer Vázquez

C00132732

Table of Contents

1. Introduction

The JUnit Test Generator with Human Oracle (from now on, “the application”) is a tool that helps developers to create JUnit Tests, providing a graphical interface that displays all the methods' information, allowing to associate instructions to test each method, create objects to use in the tests following the constructors. It is written in Java and delivered as a jar file and some other files (images and another Jar file that provides the randomization.

The aim of this document is to have a review of the following points:

- What was the first idea of the application.
- Which are the differences between the original product idea and the result
- What work is already done
- What work is left, tips to future developers, some suggestions about additional functionality
- Problems found.
- What would I do differently if I had to start over again.
- What have I learned
- Conclusion

2. Original Idea

Nowadays it is common for developers to create software. As almost all the simple applications are already created, it is required to create applications that would be more complex. Would do more things and, of course, help and reduce the amount of work that humans have to do.

In order to create complex software, it is needed to use modules. Modules that the programmer would have to code (not always) and put all of them together. Once they start to combine with each other (some function of one module is called, and the information retrieved is used by another function, etc.), is when the problems could appear.

The aim of this software was to allow to create JUnit Tests automatically in order to help developers to test their sources. By testing sources in a Unit way, it is possible to discover plenty of the errors, helping to reduce the time spent to find the errors around the whole code.

3. Current Application

The application, as it is right now, is more a wizard that helps newbie developers familiarize with JUnit Testing, providing a graphical interface in which it is possible to create asserts and add custom code to the methods of a Java code, without having to worry about the structure of the file. To explain how it does it, we will have to analyse the stages the program goes through.

3.1. Lexer

This stage is crucial. With this, it is possible to forget about the letters while reading the file. This part of the program takes the letters and figures out what they are together (which token they form), if they are a reserved word like *private, public, etc*. That is a huge help in order to understand what is the structure of the code in the file. It also takes out useless information, such as line comments, multi-line comments, spaces, enters, etc.

3.2. Parser

This stage uses the tokens provided from the previous stage and creates a data-structure with all the useful information. For example, it takes out the body of the methods, what is not important for our application. We only care of the the following information:

- Class package
- List of imports
- Class name
- Methods with their access, returned type, name and parameters.
- Constructors with their access, and parameters.

3.3. User Interface

The User Interface is the way how the user of the application interacts with it. It should show all the relevant information that the user might need to know and, at the same time, should remain simple. It is divided in three parts.

– Menu, with all the options to add a method, add objects and instructions.

– Left list, with all the methods and constructors from the file that has been read or the file the user is creating.

– Right list. It is only used when the user loads a Java file. It shows either the objects declared (when a constructor is selected on the left list) or the instructions associated to a certain method (when a method is selected in the left list).

3.3.1. JUnit Test Mode

The application has two ways (or different modes) of working. This mode is the main one, allowing the user to create objects to be used on the tests, and also create instructions and associate them to a method.

While creating an object, the user has the possibility of clicking a button that fills the fields for him (all of them (if the Randomizer is properly filled) but the object name, that it would be left in blank in order the user to fill it as he/she wishes.

While creating an instruction, if the method associated returns nothing (void), the application would only allow to declare instructions manually. Otherwise, it would also allow to create an assertTrue or assertFalse of a call to that method from an object declared previously.

3.3.2. Skeleton Mode

This mode is pretty basic, and is not one of the main goals of the project. However, it has been included because it is kind of useful while familiarizing the user with the interface.

It allows to create a class skeleton, by just saying the name of the file, its package, the imports, name of the methods, returned types, parameters types and access. (There is not much code that is added, but as it has been said before... it helps to get used to the interface.

3.4. Randomizer <external project>

With the project, it is attached another jar file (from a project that is not the JUnit Test Generator (It is considered an external project because is compiled apart)), that works as a small library.

This jar file is compiled from the Randomizer class.

This class (and also the jar) is provided to add some flexibility to the application, by allowing the developer to create its own random types.

For example if the developer wants to create String from 5 to 10 characters, including numbers and special symbols, the only thing required is to create the proper function in that class.

Another example would be if the developer has its own type from another class, it would be possible to create a function that randomly returns something as “new Whatever(23,323)”.

Is a way of doing it more flexible and also to adapt it to the real needs of the developer and the code that is being produced.

3.5. Test Generator (file generator)

The responsibility of this part of the program is to dump all the information gathered (either from the current execution or from previous ones). To do that, it is used the class TestInfo, that stores all the objects created, all the imports, the package, the class name and a dictionary that associates the methods prototypes to a bunch of Instructions.

For convection, the name of the Test class would be **SourceClass** + **Test.** For example, if the class is **User,** the test generated would be **UserTest.** The program allows the user to choose the name of the Java file, but the class name would suffer no change, so, the user would have to take that into account.

3.6. Load/Save System

Another important point of the application is the possibility of saving the progress. By progress it is understood, instructions associated to the methods and objects associated to a file.

To do that, Java's config files has been used. They are much simpler than XML structure, they take less space and they are perfect for small amount of data.

This file stores:

- The amount of objects created, their names and parameters to initialize them.
- The amount of methods that has instructions (or that had instructions), the prototype of each of those methods, the amount of instructions per method and either their code or their parameters and type of assert.

4. Original Idea VS Product

At the very beginning, this project was that. An Idea. Almost every detail was open, even which technology to use.

As it has been said before, the idea was to create a software that, given a Java source, would generate tests automatically, and also Store the test information to be reapplied after some changes, and even change them by adding new tests.

The Idea was that the application would generate suggestions to create the asserts, and the developer would only have to check that they were right. It turned out that that was pretty complicated because, to be able to do that, our application would have to modify, compile and run the sources analysed to know the expected output for the instruction that is being created (or at least interpret the source code, what would be even more complicated)

Instead, what it has been provided is a tool that allows the developer to create pseudo-automatically, JUnit Tests, for simple pieces of code. The original idea of saving the progress to reuse the tests has been kept and is working.

To simulate the generation of the code, an external Jar is being used. The developer would have to code some methods and compile the library to adapt the generator to its needs.

Finally, a small tool (that would need some changes) to create a class skeleton. As the interface used is pretty similar to the other part of the program, it helps the final user to interact with the application and familiarize it -self with the interface.

5. What is left

In this section, not only is going to be shown what is left, but also some tips about how to fulfil them, and some suggestions that would include new functionality to the application.

First of all, the Lexer and the Parser should be modified to be able to recognise more complex Java files. To achieve this goal we should start by modifying the tokens list, the automaton that interpret the characters and finally the grammar and **File**'s structure (**File** is a class that belongs to the package **ast**)

To continue, more options should be given when adding an instruction. Not only AssertTrue, AssertFalse or Custom, but also AssertEquals, AssertNull, AssertNotNull, AssertSame, etc.). To do this, it is possible to reuse the **Assert** class. Add more types to the **static final** bunch of types of asserts, add the proper piece of UI, and using the parameters' to store the objects of the other asserts, for example.

Another thing that is left is to let the user of the application, change the methods description or the imports already inserted by using the skeleton mode. (Right now, if the user makes a mistake, he/she has two choices. Either keep going and then change the generated code or start over again.

Finally, the auto-generation of fields, should be improved, for example associating a name of a variable and a type to a method, or only the name of the variable. By doing so, it would be possible to have several ways of creating each type. (For example, one String for users, another String for a Traffic plate, etc)

6. Problems Found

Some of the problems found were part of the risk identified, some others came out in the middle of the project and some others, have been through all the project.

6.1. The client doesn't know what he wants

As this project was an open project, the requirements were not set in a strict way. It was my <job> to create the software I wanted. In this case, the client (me) didn't know what he wanted. As a consequence of that, the specification was changing little by little as the project advanced. The advantage of being the client and the developer is that the lines of communication were pretty fast.

I was able to ask myself if it was <that> what I wanted, say <no>, and change it to be more similar to my wishes.

6.2. Agile Methodology and Refactoring

One of the advantages of using an Agile Methodology is that you work on a prototype, and you start adding functionalities. The problem of that, is that some of them, might (an do) make you change a lot of code.

A simple example would be the following.

At the beginning, the asserts only stored the code. Once the editing functionality was going to be added, I had three choices.

- Don't add that functionality (ruled out)
- Create a parser that takes the code as input and figures out what each field belongs to each parameter, etc. (Complex and not efficient)
- Refactor **Assert** class to store all the information in such a way that could be retrieved to be edited. (What I did)

6.3. Information Storage

There are countless ways of storing information, and I only know about writing and reading a plain text file. But I knew that there was the possibility of doing this using other technology.

I thought about using a database (e.g. MySQL or Oracle). This would make the Test's information extremely interchangeable with external applications, but would require to set up a database, and would be too complicated for a single developer.

I also thought about using XML and when I was looking for information about how to code it, to see if Java had its own build-in technology to deal with XML, I found out that there was a less known alternative, that was the configuration files[1]. Then, I learned its structure and how to work with it and that is the technique used.

6.4. Loss of the project

Just after having the Project presentation, the pen-drive where all the project was stored (with no back-up) was lost. As a result of that, I had to start over again. Fortunately, all the documentation was printed out, but already delivered. It wasn't proper to ask for them. Luckily, with the same email that I attach the Project Presentation, I also attach the Design Manual. What relieve the shock of having to start everything over again. (by everything I mean all the code).

In the design manual where the tokens, the grammar, and the class structure I used consequently, I was able to catch up faster, as I didn't have to think about the LL(1) grammar, or how to design the parser and the UI.

6.5. Using methods through Strings

To be capable of showing some auto-generation feature, it was needed to have a method that given a Type, was able to call certain methods, without knowing their names in compilation time. To perform that operation, I used the getMethod and invoke methods.

7. What would I do different

First of all, I will back everything up, almost everyday in order to minimize the loss of information in case something happens. There is nothing more irritating and boring than having to do the same thing twice. It is easer to do it the second time, cause you already have an idea, and you do remember some mistakes that made you lose time, that now you won't have them.

Now that I have done a parser by hand, and now that I know how to workout a grammar in LL(1) (it is required to be able to use a descent recursive parser), I would probably use JavaCC. It would simplify the way of parsing the code and would probably be easier to modify in future iterations.

To continue, I would increase the amount of effort given to the design, mainly due to time lost in redesigning the application. And its consequent changes in the code. More time in design, would be rewarded as less variations on the code. What, in turn, would save time. And in bigger programs, would save a lot of trouble having to say a whole team of coders that they have to change half of their code.

To conclude, the last thing I would do different is the hand-made Graphical Interface. It is possible that, having done it by hand, it was easier to debug and it was even more efficient, but for such a small application (graphically speaking) is not worthy to spend (waste) that amount of time on coding and modelling the GUI when using a drag-n-drop tool would be much faster.

8. What have I learned

To commence, I have learned the importance of having good lines of communications with the client and also with the people that is going to use the program. It is very hard to get what the client wants. The real specification.

Usually, the client has an idea in mind and is our job to model it, and guess if its what he wants. Sometimes, he has an idea and the real users of the application has a complete different idea. Only by having interviews, interactivity and showing the prototypes, can we get the proper feedback.

Agile Methodology has, apart from the disadvantages written before, good things. It allows to show application's prototypes to the customer and have a feedback of, either the functionality, the GUI, etc.

It is also important to have good means of communication. If each time we, as developers, have some doubts, we have to wait one month to solve them, it is likely that what we have assume, was wrong, having to change some code and, in the worst case, throw everything away and start over.

I have learned the importance of doing JUnit Tests, their structure, and that they are there for a reason, avoid stupid mistakes to hide in classes when we try the whole application and fails.

I have also learned a lot about Java, that there are things that I didn't know that it was possible to do them, such as configuration files, that are pretty easy and I would have used them years ago if I had known about them.

9. Conclusions

- Developing software is a science that needs psychology and guessing skills
- Doing things by hand when there are tools that helps, is a bad Idea.
- It is vital to have back-ups of everything.
- Sometimes the customer doesn't even know what he wants. In these cases, the developer should get the information for the applications as the police takes information for their Identikit picture.
- Agile Methodology helps to detect misunderstandings between what the client wants and what the developers think the client wants. It is also good because it allows fixing problems as you go, but under some cost. (having to redo/change code from time to time)

10. References

[1] Properties (The Java TM Tutorials > Essential Classes > The Plataform Environment

http://java.sun.com/docs/books/tutorial/essential/environment/properties.html