# Functional Specification

# JUnit Test Generator

I.T. Carlow

Software Engineering

Bachelor Degree (With Honours)

Sergio Alcocer Vázquez

C00132732

## Table of Contents

# 1. Vision

## 1.1. Introduction

The JUnit Test Generator is a system that allows the user (usually a software developer) to generate JUnit Tests early in the project semi-automatically and, if the project is already finished, to generate JUnit Tests to test it. The program will gather information from the user during all the process of creation of the tests (with a wizard) such as input examples (type is recognized, but not its format if exists), but it won't require the user to be an expert on JUnit Testing, only a brief knowledge. If the tests are for a project that hasn't been started yet, it will allow to generate its skeleton. And all the information gathered and all the settings should be kept for futures executions.

## 1.2. Business Case

There are other programs that tent to do the same, but as they are not under support any more, and Java is a language that is changing (little by little, but changing) they are outdated and not working for the current needs at all. Some others are expensive and comes with several extras (included in the price) that maybe the developer don't want to use it. Our program, apart from offering a much cheaper tool, it will also be so simple to use that the developer could be learning how to create JUnit Tests by hand at the same time (s)he use it.

## 1.3. Problem Statement

Nowadays, the Software Engineering is very used on all projects, but still, too many project don't include Testing as part of the development process, what leads in a bad quality software. The main reason of that is that there are a lot of software developers don't know about Testing (not only Unit Testing), or the know about it, but they don't have time to learn it. With our program, software developers, not only would be able to start using JUnit Tests answering some simple questions, but also they will learn how to use JUnit.

## 1.4. Stakeholders

Stakeholders for this project are developers that want to add JUnit Testing to a Project that doesn't have it and those who want to start a project using JUnit and want the test part to be generated automatically as they go typing code for the application.

## 1.5. Non-Functional Requirements

| Requirement | Reason |
|---|---|
| Java Language | Since the source analysed is in Java, it is convenient to use the same language in order not to force users to install extra software. |
| Maintainability | Since Java is a language that is changing continuously, the code of the project should be easily modifiable to support these changes. |
| Intuitive Interface (Easy to use) | As the user could be a non-expert on JUnit then, it should be enough easy to use that the user actually would be able to use it. |
| No Performance | The response time is not critical. It doesn't matter whether it takes 2 or 30 seconds to work-out the tests classes. |
| No Scalability | As it is a JUnit Test Generator, the classes to be analysed cannot be increasing too much (otherwise, it is a bad design) |
| No Security | As there is no personal information involved, no life is in danger, etc. |
| Medium Reliability | Should be have enough reliability to trust it as a tool, but it may fail and the programmer should be able to notice that. |
| Availability ? | It's not an on-line tool (not right now), so the concept of availability is not required. If it is decided to develop a web-based version, this requirement would become more important. |

## 1.6. Functional Requirements

| Requirement | Priority | Concerns |
|---|---|---|
| Create Test for a certain function | Critical | Creating a test for a method of the class (that includes increasing it) |
| Analyse Project's Class | Critical | Must generate a tree of public methods, recognise inputs, etc. |
| Auto-save | High | Should be saving all the information gathered from the user in order to use it in future executions. |
| Import / Export Configuration in XML | Optional | To follow standards and be able to easy use it with other future applications, and to be easy modifiable and readable by humans, saved information should be importable and exportable through XML files. |
| Web Integration | Optional | Allow people using the program from a website without having to run it. Just uploading the project folder. |
| Update Manager | Optional | Will automatically check (if Internet connection is available) for updates of the software and ask the user if (s)he wants to install it. |

## 1.7. System Usage

There are two main usages of the application. One of them as a code analyser (for projects that already exists) and the other one is as a pre-development process, that will trigger automatically the wizard to guide the user.

### 1.7.1. Code Analyser

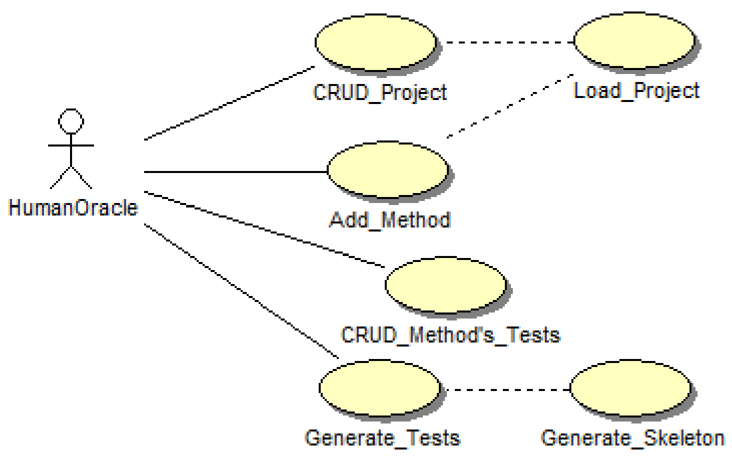
The user must specify source files, and the application would scan them to know dependencies between classes. After that, a Wizard would guide the User through all the steps to add new tests, by displaying options to choose which method would the User like to test, what parameters to use, and all the related options. Will suggest outputs (based on current output of the code), etc.

### 1.7.2. Wizard

A wizard will guide the user through all the steps, creating test for functions that doesn't exist, type of parameters, etc. And maybe will generate a skeleton of the class needed to fulfil the test (at least the public methods)

# 2. Functional Model

## 2.1. Use Cases



| | |
|---|---|
| Use Case | UC01 |
| Name | CRUD_Project |
| Actors | HumanOracle |
| Priority | Critical |
| Details | The Create's basic flow is as follows:<br><br>1. &lt;HumanOracle&gt; chooses Create Project Option.<br>2. &lt;HumanOracle&gt; browses the destination folder.<br>3. &lt;System&gt; creates the project.<br>4. &lt;System&gt; stores some information in the folder chosen.<br>5. The use case ends.<br><br>Alternative flows:<br><br>If in step &lt;2&gt;, the folder contains a project.<br>3. &lt;Load Project&gt; use case is triggered.<br>4.1. If a JUnit project exists, &lt;System&gt; loads the information.<br>4.2. Otherwise, &lt;System&gt; creates file configuration, etc.<br>5. The Use Case ends.<br><br>Alternatives:<br>Read Project.<br>In step &lt;1&gt;,<br>&lt;HumanOracle&gt; chooses Load Project Option.<br>In step &lt;2&gt; destination folder must contain a ...<br>...JUnit Project.<br><br>The Update's basic flow is as follows:<br>Precondition: a JUnit Project should have been loaded.<br>1. &lt;HumanOracle&gt; changes some settings.<br>2. &lt;System&gt; stores the changes<br>3. The Use Case ends.<br><br>The Delete's basic flow is as follows:<br>Precondition: a JUnit Project should have been loaded.<br><br>1. &lt;HumanOracle&gt; chooses Delete Project Option<br>2. &lt;System&gt; asks to delete either the Junit Project or also the Java one<br>3. &lt;HumanOracle&gt; chooses what to delete.<br>4. &lt;System&gt; deletes the information requested.<br>5. The Use Case ends. |

| | |
|---|---|
| Use Case | UC02 |
| Name | Load_Project |
| Actors | HumanOracle |
| Priority | Critical |
| Details | The basic flow is as follows:<br><br>1. <HumanOracle> choose the Java project to be loaded, or is triggered by <CRUD_Project> use case.<br>2. <System> analyse the sources and display a "working" message<br>3. <System> uses <Add_Method> use case to add the loaded class methods.<br>4. <System> changes the view (User Interface) and show the tree of methods, proper menus, etc.<br>5. The Use Case ends.<br><br>Alternative flows:<br><br>If in step <3>, there is no information about the class, will skip the step and resume to step <4> |

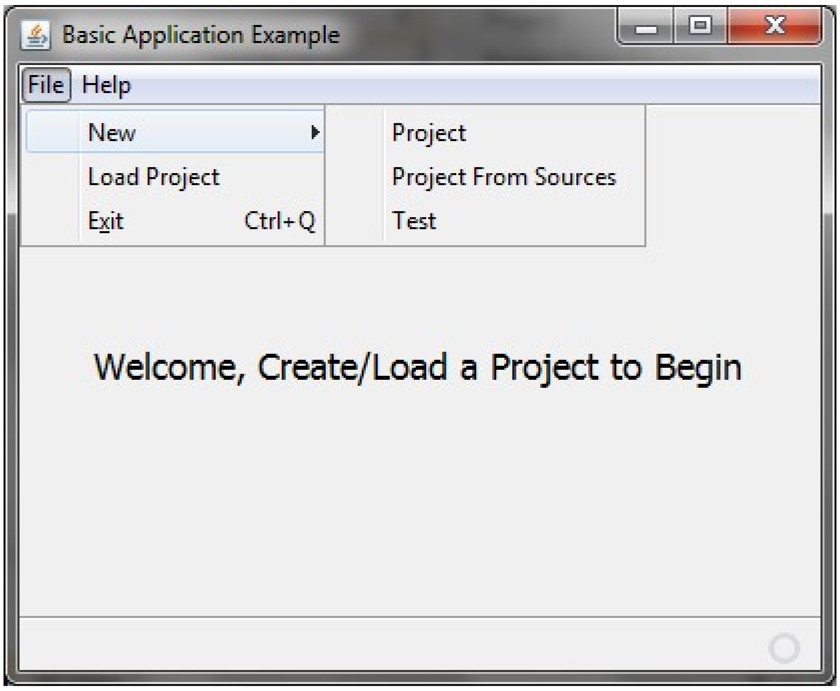
| | |
|---|---|
| Use Case | UC03 |
| Name | Add_Method |
| Actors | HumanOracle |
| Details | This Use Case is only available if the user is creating the project from scratch (won't be available if is creating tests for an existing project unless is triggered by the system. In that case, <HumanOracle> interaction would be filled by <System>). 1. <HumanOracle> give a name to the method 2. <HumanOracle> sets returning value 3. <HumanOracle> add a parameter name and type. 4. <HumanOracle> repeats step <3> until all methods have been introduced. 5. <System> updates list of methods. |

| | |
|---|---|
| Use Case | UC04 |
| Name | CRUD_Method's_Tests |
| Actors | HumanOracle |
| Details | Preconditions:<br><br>A class method has to be selected.<br>While Reading, Updating and Deleting, a Test Case has to be chosen<br><br>The Create's basic flow is as follows:<br><br>1. &lt;System&gt; will show a window with suggestions for the inputs and/or output.<br>2. &lt;HumanOracle&gt; chooses the auto-generated inputs<br>3. &lt;HumanOracle&gt; writes the expected output<br>4. &lt;HumanOracle&gt; click on save<br>5. &lt;System&gt; add the new Test to the list and save it in the datastore.<br>6. &lt;System&gt; return to Main View (User interface)<br>7. The Use Case ends.<br><br>Alternative flows:<br>If on step &lt;2&gt; the user doesn't want the auto-generated inputs<br>2. &lt;HumanOracle&gt; writes his own inputs<br>3. Resume to Step &lt;3&gt; (basic flow).<br><br>Read flow:<br><br>1. &lt;System&gt; gets and shows current inputs and the output<br>2. The use case ends<br><br>Update flow:<br>1. &lt;System&gt; will show a window with the current inputs and output.<br>2. &lt;HumanOracle&gt; modifies input and/or the output<br>3. &lt;HumanOracle&gt; click on Save Changes<br>4. &lt;System&gt; updates the new Test Information in the list and in the datastore.<br>5. &lt;System&gt; return to Main View (User interface)<br>6. The Use Case ends<br><br>Delete flow:<br>1. &lt;System&gt; deletes the Test Case from the list and from the datastore.<br>2. The Use Case ends. |

| | |
|---|---|
| Use Case | UC05 |
| Name | Generate_Test |
| Actors | HumanOracle |
| Details | 1. <HumanOracle> Chooses Generate Test Option<br>2. <System> with the information from the class(es) and the information about the methods and the Test cases, it builds the test class.<br>3. The use case ends. |

| | |
|---|---|
| Use Case | UC06 |
| Name | Generate_Skeleton |
| Actors | HumanOracle |
| Details | This use case is only available when the project opened is one created from scratch.<br><br>1. <HumanOracle> Chooses Generate Skeleton option.<br>2. <System> generates the skeleton with the information gathered in the project's folder<br>3. Use case ends |

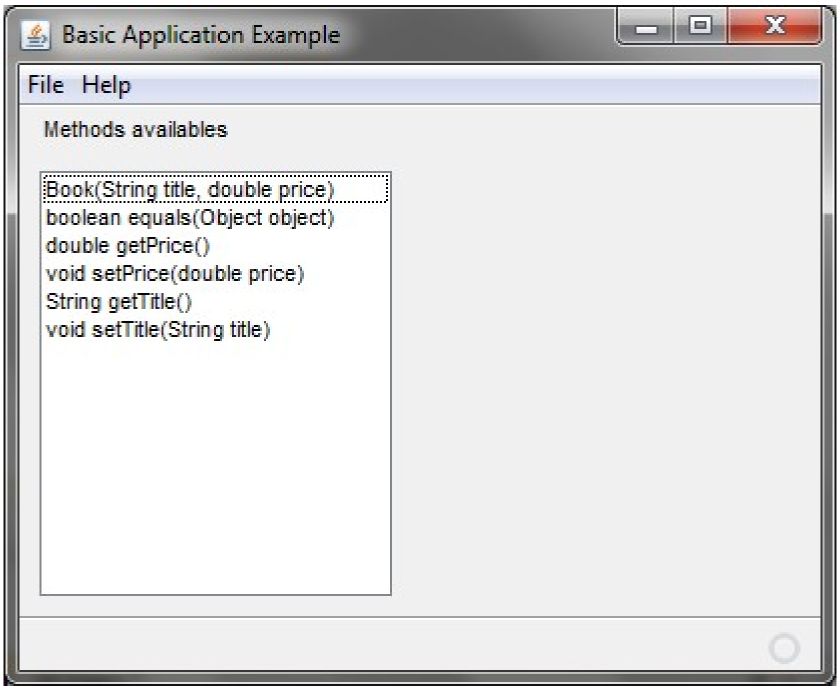
## 2.2. Screen Shots



Welcome, Create/Load a Project to Begin

This is the starter User Interface. No project has been created nor loaded yet. It gives the user the choice to either create a new project from scratch or from existing Sources.

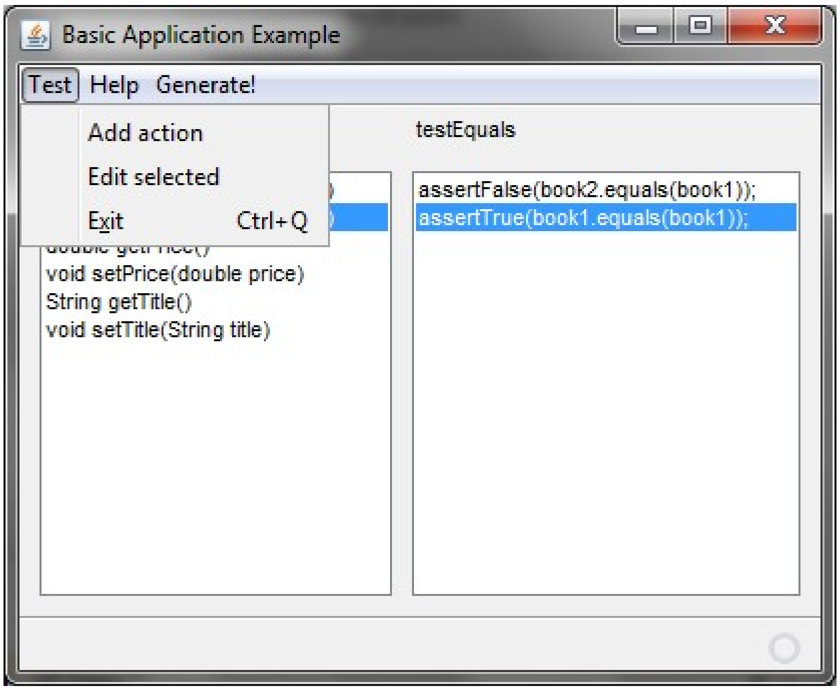
If *Project From Sources* is chosen, the user will be requested to give the location of the source file (browsing it), after that, the program will analyse the code and prompt a list of functions (public methods of the class), to start adding tests.



Methods availables

Book(String title, double price)
boolean equals(Object object)
double getPrice()
void setPrice(double price)
String getTitle()
void setTitle(String title)

A list of functions will be shown, (once analysis finishes or as the user types them). Selecting one of them, will list the actions associated to that method as shown below.



testEquals

void setPrice(double price)
String getTitle()
void setTitle(String title)

assertFalse(book2.equals(book1));
assertTrue(book1.equals(book1));

The constructor method (left list) is used to store almost all the objects used for the whole test. (will be shown on the right list).

From this view, the user can, either edit a current action, insert new ones, Generate the code, etc.

Generating the code may mean generating also the skeleton of the classes if is done from scratch and the user give his permission.

As this is a document released at the beginning of the development process, is not possible to assure that User interfaces won't change. Actually, they are likely to change to fit the needs during the Modelling Process and the Coding of the application.