

Requirements document

Rowan University Software Engineering

UNIT TEST GENERATOR TOOL

Professor Jack Myers



October 13, 2017

TEAM TIGERS

Prepared by: Aanchal Chaturvedi

TABLE OF CONTENTS

1. **INTRODUCTION………………………………………………………………………………………………………………..……….…. 2**
   1. DOCUMENT PURPOSE……….……………………………………………………………………………………..………….…….2
2. **SYSTEM DESCRIPTION……………………………………………………………………………………………………..…….…….. 3**
3. **DETAILED REQUIREMENTS………………………………………………………………………………………..……………....... 5**
   1. FUNCTIONAL REQUIREMENTS…………………………………………..………………….……..……………………..…….. 5
      1. FRONT END REQUIREMENTS………………………………………………………………………………………… 5
      2. MIDDLE END REQUIREMENTS………………………………………………………………………..…………….. 6
      3. BACK END REQUIREMENTS…………………………………………………………………………………..………. 6
   2. NON-FUNCTIONAL REQUIREMENTS…………………………..………………………….………………………..…………. 7
4. **USE CASE DIAGRAMS…………………………………………………………………………………………………………………… 8**
5. INTRODUCTION
   1. **Document Purpose**

The requirements document is a formal statement of the application functional and operational requirements. The developers of the Tiger team agree to provide the capabilities specified. The client agrees to find the product satisfactory if it provides the capabilities specified in the Requirements Document.

The goal of this project is to create a tool that will generate unit tests for C++ classes. Users will input C++ classes for which they want the test to be generated. The tool will return components needed for testing the file. The *System Requirements* section of this document describes that in detail. Other requirements which include storing data, performance and reliability of the tool will be included in the *Detailed Requirements* section of the document.

1. SYSTEM DESCRIPTION

The Unit Test Generator tool is designed to help fasten the testing of C++ classes. In complicated systems, unit testing is often neglected in pressure to meet increment deadlines on time. Designing a software tool which by itself generates unit tests for the files being inputted into it will save time for the developer, who otherwise will have to create unit tests for all the classes individually.

The tool functionality will be divided into three different sections namely, *the front end, the middle end and the back end*[[1]](#endnote-1). It is important to break down the functionality into different sections to ensure that each end has its individual function and they work co-dependently to produce the desired output. The output for the tool shall be files that can be fed into CPP Unit Lite for testing. In *CPP Unit Lite*[[2]](#endnote-2) to write individual tests for C++ classes the user creates test groups.

Test groups can be populated with tests which check Boolean results, memory leaks[[3]](#endnote-3), relational operators between two entities, and other equivalents. Every test group shall have its own setup and teardown method. The *setup method* will be called before each test and the *teardown method* is called after each test. The testing framework should work for individual tests as well as libraries of tests.

1. Front end: part of the program which is directly accessed by the user; Middle end: part of the program that acts as a glue between the user interface and the data; Back end: part of the program which resides away from the user’s vision and performs specialized function on behalf of the user. [↑](#endnote-ref-1)
2. CPP Unit Lite is a C/C++ based unit testing framework for unit testing and test-driving your code. It is written in C++ but is used in C and C++ projects. The core design principles include its simple design and user friendly framework. The unit testing framework is portable to both old and new platforms which makes it popular as a C++ unit testing tool. [↑](#endnote-ref-2)
3. Memory leaks are failure in program to release discarded memory.

   \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

   3. detailed requirements

   **3.1 Functional Requirements**

   3.1.1 Unit Test Generator Tool – Front End

   |  |  |
   | --- | --- |
   | Requirement Reference # | Description |
   | 3.1.1.1 | The tool shall have a graphical user interface through which the user can select files to test. |
   | 3.1.1.2 | The graphical user interface (GUI) shall have a browse button. |
   | 3.1.1.3 | The browse button shall have the ability to search the user’s computer for a specific file. |
   | 3.1.1.4 | The browse button should have the ability to select single files as well as multiple files. |
   | 3.1.1.5 | The GUI shall have a text box area which displays name of the file or files that are selected. |
   | 3.1.1.6 | The front end of the tool shall have the ability to show a default output file name for the output result but also allow the user to modify it. |
   | 3.1.1.7 | The front end of the tool should have a drop-down feature in the text box which contains recently opened files. |
   | 3.1.1.8 | The front end of the tool shall have a small area designated to display a brief visual preview of the final output. |
   | 3.1.1.9 | The front end of the tool shall have the ability to display the given error information when test frameworks fails. |
   | 3.1.1.10 | The front end of the tool shall have the ability to display a pop-up dialogue box which allows user to select one file if two files of the same name exist. |
   | 3.1.1.11 | The graphical user interface should allow parameter selection using toggles and switches. |

   **3.1 Functional Requirements**

   3.1.2 Unit Test Generator Tool – Middle End

   |  |  |
   | --- | --- |
   | Requirements Reference # | Description |
   | 3.1.2.1 | The middle end of the tool shall have a search file system which uses a pop-up box to display a ‘found message’ if the file is found and a ‘not found message’ if the file is not found. |
   | 3.1.2.2 | The middle end of the tool shall implement a method which avoids performing file search on the unit test directories |
   | 3.1.2.3 | The middle end of the tool shall be able to create a makeFile for the corresponding class when found. |
   | 3.1.2.4 | The middle end of the tool shall be able to create a testFixture for the corresponding class when found. |
   | 3.1.2.5 | The middle end of the tool shall be able to create a Unit Test file for corresponding class when found. |
   | 3.1.2.6 | The middle end of the tool shall develop a file parser which reads any class inputted into it. |
   | 3.1.2.7 | The middle end of the tool shall include a simple API to easily interface with the Front and Back End. |

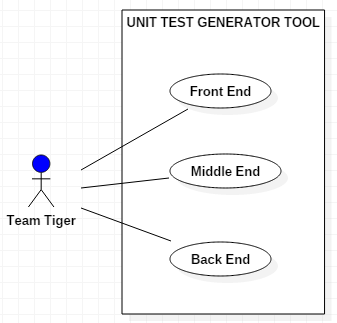
   3.1.3 Unit Test Generator Tool – Back End

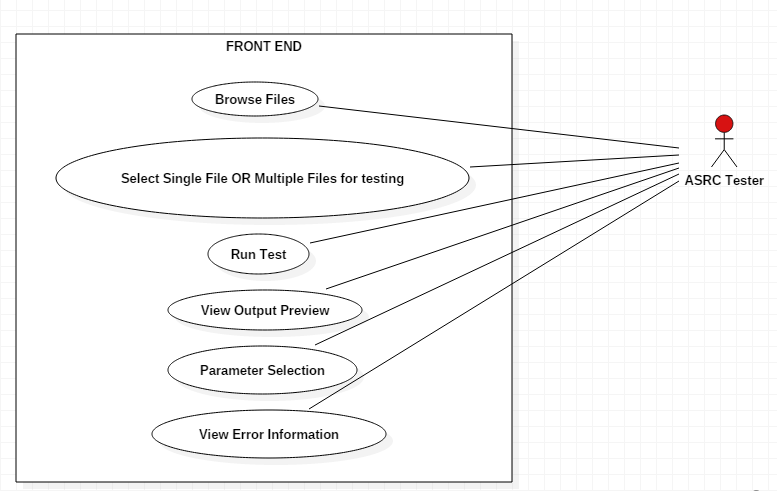
   |  |  |
   | --- | --- |
   | Requirements Reference # | Description |
   | 3.1.3.1 | The back end of the tool shall have the ability to save the output to a specified location |
   | 3.1.3.2 | The back end of the tool shall have the ability to save and display logs for tests generated previously. |
   | 3.1.3.3 | The back end of the tool shall have functionality to generate basic C++ Unit Test Code. |
   | 3.1.3.4 | The back end of the tool shall have a simple API to easily interface with the middle end. |

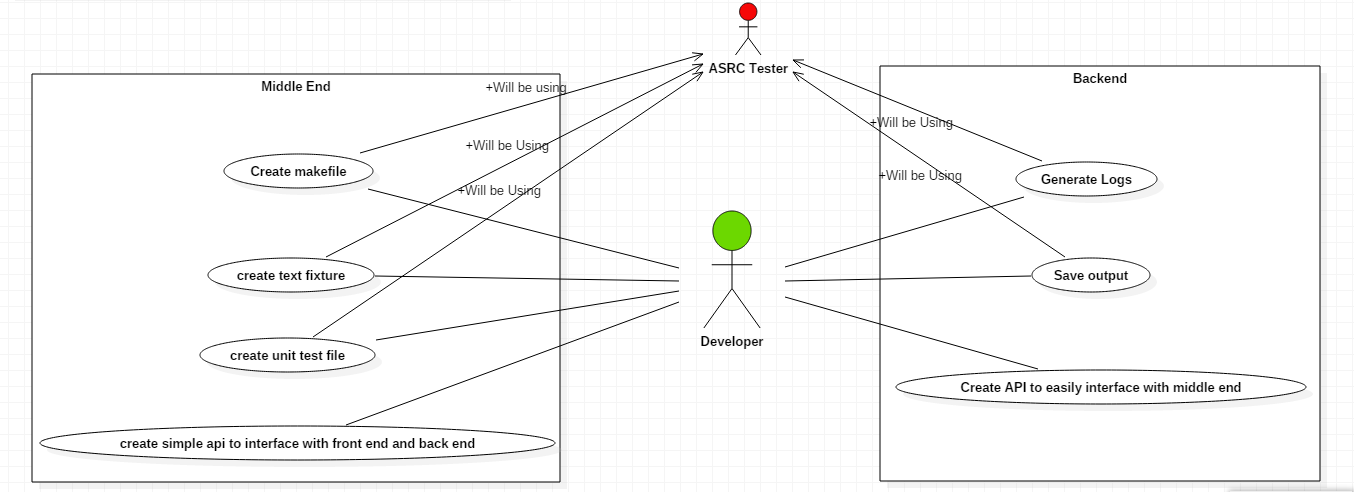
   **3.2 Non Functional Requirements**

   |  |  |
   | --- | --- |
   | Requirements Reference # | Description |
   | 3.2.1 | The source code for the tool must be uploaded on GitHub to ensure transparency. |
   | 3.2.2 | The front end tool must be simple and easy to use. |
   | 3.2.3 | The middle end and back end of the tool must comply with C++ and OOP design patterns. |
   | 3.2.4 | The source code for the tool must be neat, organized and documented to ensure easy readability. |
   | 3.2.5 | The tool must be developed using Agile Methodologies. |
   |  | |

   4. USE CASE DIAGRAMS

    [↑](#endnote-ref-3)