Software Requirements Specification

for

Automatic Random Regression Testing with Human Oracle

Prepared by: Ivan Maguidhir
Date: January 2011

Table of Contents

1. Introduction....4
  1.1 Purpose....4
  1.2 Document Conventions....4
  1.3 Intended Audience and Reading Suggestions....4
  1.4 Product Scope....4
  1.5 References....4
2. Overall Description....4
  2.1 Product Perspective....4
  2.2 Product Functions....5
  2.3 User Classes and Characteristics....5
  2.4 Operating Environment....5
  2.5 Design and Implementation Constraints....5
  2.6 User Documentation....5
  2.7 Assumptions and Dependencies....5
3. External Interface Requirements....5
  3.1 User Interfaces....5
  3.2 Hardware Interfaces....6
  3.3 Software Interfaces....6
  3.4 Communications Interfaces....6
4. System Features....6
  4.1 Create Tests....6
    4.1.1 Description and Priority....6
    4.1.2 Stimulus/Response Sequences....6
    4.1.3 Functional Requirements....7
  4.2 Run tests....7
    4.2.1 Description and Priority....7
    4.2.2 Stimulus/Response Sequences....7
    4.2.3 Functional Requirements....8
  4.3 List tests....8
    4.3.1 Description and Priority....8
    4.3.2 Stimulus/Response Sequences....8
    4.3.3 Functional Requirements....8
  4.4 Configure testing....9
    4.4.1 Description and Priority....9
    4.4.2 Stimulus/Response Sequences....9
    4.4.3 Functional Requirements....9
  4.5 Remove Tests....9
    4.5.1 Description and Priority....9
    4.5.2 Stimulus/Response Sequences....9
    4.5.3 Functional Requirements....10
Appendix A: Glossary....11
Appendix B: Analysis Models....11
  Use Case Diagram....11
  Detailed Use Cases....12
    Use Case 1: Create tests....12
    Use Case 2: Run tests....14
    Use Case 3: List tests....15
    Use Case 4: Configure testing....16

Use Case 5: Remove tests....17
Simple Examples....18
Configuration....18
Create and Run Tests....18
List Tests....18
Remove Tests....19
Appendix C: Supplementary Specification....20
Functional Requirements....20
Auditing 20
Localization....20
Reporting....20
System management....21
Workflow....21
Usability, Reliability, Performance and Supportability Requirements....21
Usability....21
Reliability....21
Supportability....22
Localization....22
Configurability....22
Compatiblity....22
Design, Implementation, Interface and Physical Requirements....22
Design....22
Implementation....22
Interface....22

1. Introduction

1.1 Purpose

This document describes the requirements of the software product AutoUnit. AutoUnit is an implementation of the Automatic Random Regression Testing with Human Oracle project proposal for the ANSI C programming language. The scope of this document is to describe the requirements of the AutoUnit product in its entirety.

1.2 Document Conventions

Source code and console output are indicated with `Courier 10 Pitch` font.

1.3 Intended Audience and Reading Suggestions

This document is intended for developers, testers and documentation writers. Sections 2, 3 and 4 are essential reading for any of these audiences.

1.4 Product Scope

AutoUnit is a software tool for analysing ANSI C source code, provided to it by a user, generating tests for each function in the source code using the CppUnit implementation of the xUnit testing framework and running these tests periodically. The objectives of the software are:

- To reduce the effort required by the developer for testing by suggesting tests and running them automatically going forward.
- To reduce errors in the developers code and therefore save time which would normally be spent debugging.
- To continuously attempt to uncover errors in previously error-free code

From a management and business point of view the objective of the software is to reduce the time and cost associated with development and the provision of technical support.

1.5 References

Meudec, Christophe. IT Carlow Project Proposals, Software Engineering Honours Degree 2010-2011

2. Overall Description

2.1 Product Perspective

AutoUnit is a new, self contained unit testing tool for the ANSI C programming language which uses the CppUnit library(an implementation of xUnit for C++) to perform tests. The system consists of:

- A library containing all of the functionality of the software
- A user interface in the form of a command line application which makes use of and documents all of the functions provided by the library

2.2 Product Functions

- Create tests
- Run tests
- List existing tests
- Configure testing
- Remove tests

2.3 User Classes and Characteristics

The product is intended for use by people with a high level of technical expertise, specifically:

- Developers
- Test Engineers
- Software QA Staff

2.4 Operating Environment

The software is intended to be cross platform i.e. any operating system specific code will be implemented using a cross platform library (such as Qt or wxWidgets). We are focusing on developing the product for the Linux and Microsoft Windows platforms.

2.5 Design and Implementation Constraints

- Use of a cross platform library may be required so that the product will compile and behave consistently across all target platforms.
- The software must work with both the GCC and Microsoft Visual Studio compilers and linkers.

2.6 User Documentation

- User Manual
- Library Documentation

2.7 Assumptions and Dependencies

No assumptions. The software is dependent on the CppUnit library and any cross-platform library that is used.

3. External Interface Requirements

3.1 User Interfaces

Command line interface

3.2 Hardware Interfaces

None

3.3 Software Interfaces

The library which contains all the functionality of the software. This is used by our command-line application and can be used by licensed 3rd party applications (e.g. IDE plug-ins).

3.4 Communications Interfaces

None

4. System Features

4.1 Create Tests

4.1.1 Description and Priority

Allow the user to create tests for specified source code. High priority.

4.1.2 Stimulus/Response Sequences

1. The user opens a terminal / command-line window
2. The user enters the name of the command-line application
3. The user enters an option as a parameter to indicate they wish to generate tests
4. The user enters the path of the source-code file they wish to generate tests for as a parameter
5. The user optionally enters flags to:
   - Set compiler and linker flags/options for the source-code
   - Set the number of tests to be generated for each function
   - Generate tests for all functions without asking
6. The user hits enter/return
7. The user is asked to set compiler and linker flags/options for the specified source-code if they have not done so using a flag already
8. For each function in the source-code file the user is asked if they want a test generated unless they have used the flag to generate tests for all functions.
9. For each test generated the test is run and the user is shown the result and asked to verify it
10. The user is asked after each test is generated and verified if they want to generate further tests for the same function unless they have set the number of tests to be generated using a flag in which case that exact number of tests is generated.

4.1.3 Functional Requirements

- The application must be able to parse the source code and find each function it contains
- The application must be able to detect errors in the source code either through its own mechanism or by calling the compiler
- If errors are found in the source code the user must be notified and the operation aborted
- When asked for compiler and linker flags the user should be shown the options which were previously used as a hint/default option
- The application must be able to modify/update the source-code to allow code which calls CppUnit to be inserted into it
- The process of copying, modifying and running tests on a particular source-code file is repeated every time the test is run going forward. i.e. the code is not updated once and left in its modified state.
- The application must be able to launch the compiler and linker to compile and link the updated source code
- If errors occur the user must be notified, shown the compiler/linker error and the operation aborted
- The application must be able to launch the generated executable in order to run the test and receive the result
- The application must be able to save the details of each new test (source-code path, function name, test values and a unique ID) to a “tests file” or “test database” for later retrieval

4.2 Run tests

4.2.1 Description and Priority

Run the tests which have been added to the test file/database. High priority.

4.2.2 Stimulus/Response Sequences

1. A task scheduler configured by the user calculates that the interval between running tests has elapsed
2. The task scheduler runs the command-line application with an option as a parameter indicating that tests should be run
3. The application runs the tests

or

1. The user opens a terminal / command-line window
2. The user enters the name of the command-line application
3. The user enters an option as a parameter to indicate they wish to run tests now
4. The application runs the tests

4.2.3 Functional Requirements

- The application must be able to read the list of tests from the test file/database
- The application must be able to read the source-code referred to by each test
- The application must be able to detect errors in the source-code either through its own mechanism or by calling the compiler
- The application needs to be able to tell if the source code for a specific function has been modified since it was run last (e.g. using a hash which is updated after run of the test)
- The application must be able to modify/update the source-code to allow code which calls CppUnit to be inserted into it
- The application must be able to launch the compiler and linker to compile and link the updated source code
- The application must be able to launch the generated executable in order to run the test and receive the result
- If compiler or linker errors are found in the source code the current test is aborted, detailed information is logged for the user and the application moves on to the next test
- The application, before running any tests, needs to build a list of the tests it will run based on which functions have changed since they were last tested and random selection. This list of tests will be a subset of all tests in the test file/database

4.3 List tests

4.3.1 Description and Priority

Display a list of existing tests to the user. Medium priority.

4.3.2 Stimulus/Response Sequences

1. The user opens a terminal / command-line window
2. The user enters the name of the command-line application
3. The user enters an option as a parameter to indicate they wish to see a list of existing tests
4. The user optionally enters a flag followed by a function name or a flag followed by a source-code path
5. The user hits enter/return
6. The application displays a list of existing tests on the console

4.3.3 Functional Requirements

- The application must be able to read the list of existing tests from the tests file/database where they were previously saved. If this is inaccessible the user must be notified and the operation aborted
- The application should pause every screen-full of tests to allow the user an opportunity to read the details of each test

- The application should be able to limit the list of tests returned using a function name or source-code path provided by the user and only display tests associated with that function name or source-code file.

4.4 Configure testing

4.4.1 Description and Priority

Configure how tests are picked to run and how many are run. High priority.

4.4.2 Stimulus/Response Sequences

1. The user opens a terminal / command-line window
2. The user enters the name of the command-line application
3. The user enters a parameter to indicate they want to change settings
4. The user optionally enters an option as a parameter to indicate they wish to configure the maximum number of tests per run followed by an integer representing the maximum number of tests for each run
5. The user optionally enters an option as a parameter to indicate they wish to configure how tests are picked to run followed by a numeric value
6. The user hits enter/return
7. The application updates the configuration file
8. The application confirms to the user that settings have been updated

4.4.3 Functional Requirements

- The application must be able to read from and write to the configuration file if it cannot the user must be notified and the operation aborted
- The application must update the configuration file to reflect the new settings
- The numeric value entered by the user to configure how tests are picked to run represents whether tests are chosen based on the fact that the source-code of the functions they test has changed since the last test or if tests are to be chosen randomly or a combination of both with priority given to tests whose functions have changed since the last test.

4.5 Remove Tests

4.5.1 Description and Priority

Remove a test or tests. Medium priority.

4.5.2 Stimulus/Response Sequences

1. The user opens a terminal / command-line window
2. The user enters the name of the command-line application
3. The user enters an option as a parameter to indicate they wish to remove tests

4. The user enters a flag followed by a test ID or a flag followed by a source-code file path
5. The user hits enter/return
6. The application removes the test with the specified ID or tests associated with the specified source-code file from the test file/database
7. The application tells the user that the tests have been removed

4.5.3 Functional Requirements

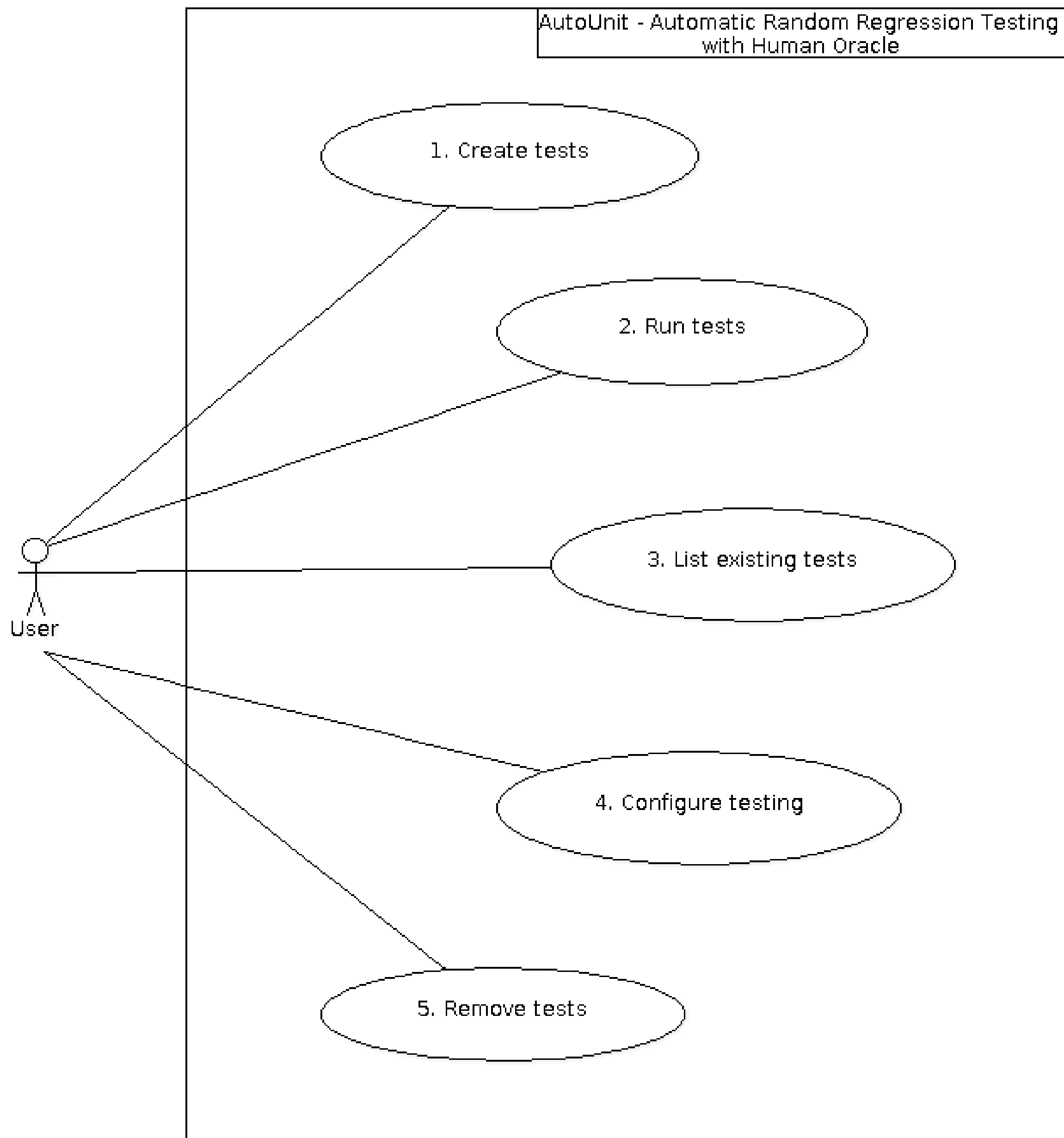
- The application must be able to read from and write to the test file/database. If it cannot the user must be notified and the operation aborted

Appendix A: Glossary

Appendix B: Analysis Models

Use Case Diagram

AutoUnit Use Case Diagram



Detailed Use Cases

Use Case 1: Create tests

Use Case UC1 : Create tests
Scope : AutoUnit - Automatic Random Regression Testing with Human Oracle
Level : User Goal
Primary Actor : User
Stakeholders and Interests :
User: Wants tests generated for the specified source code.

Preconditions:
The system is installed.

Success Guarantee (aka Postconditions):

- Results of the tests generated for the chosen source-code have be verified by the user.
- The system has added the location of the source-code, the function signature and the test values to the tests file/database.

Main Success Scenario (or Basic Flow):

1. The user opens a terminal / command-line window
2. The user enters the name of the command-line application
3. The user enters a command as a parameter to indicate they wish to generate tests
4. The user enters the path of the source-code file they wish to generate tests for as a parameter
5. The user optionally enters flags to:
    - Set compiler and linker flags/options for the source-code
    - Set the number of tests to be generated for each function
    - Generate tests for all functions without asking
6. The user hits enter/return
7. The system reads the most recently used compiler and linker flags/options from its configuration file
8. The user is asked to set compiler and linker flags/options for the specified source-code with the most recently used flags/options displayed as a default/hint
9. The system parses the source-code file
10. The system identifies a section of code as being a function definition
11. The name of the function is displayed on the console and the user is asked if they want a test generated for this function
12. The user tells the system that they want a test generated.
13. The system generates a test for this function
14. The system runs the test and displays the result
15. The system asks the user if the result is correct

16. The user verifies that the result is correct
17. The system adds the location of the source-code, the function signature, test values and result to the tests file/database.
18. The user is asked if they want further tests generated
19. The user tells the system they want further tests generated for this function. Repeat Step 13.

<u>Alternative Flows</u>:

8a. The system has not been configured (no configuration file exists).

9. A message is displayed to the user to tell them they must configure the system before creating tests and the operation is aborted.

8a. The user has already supplied compiler and linker flags at the command-line. Perform Step 9.

9a. The system cannot locate the specified source-code

10. A message is displayed to the user to tell them that the source-code cannot be found and the operation is aborted.

7a. The user entered an unrecognised value as a command-line parameter

8. A message is displayed to the user to tell them that an unrecognised parameter has been entered and an outline of the command-line syntax is displayed on the console.

9b. The source-code file contains errors

10. The user is notified that the source-code contains errors and the system exits.

18a. The user has already indicated that a fixed number of tests are to be generated for each function using a command-line flag. Repeat step 9 for all functions in the source-code.

18b. The fixed number of tests specified by the user have been generated. Repeat Step 9.

19a. The user tells the system they don't want further tests generated for this function.

20. Perform Step 9.

10a. No further functions can be found in the source-code file. The user is notified that the operation is complete and the system exits.

16a. The user tells the system that the result is incorrect.
17. The system asks the user to specify what the correct result should be.
18. The system adds the location of the source-code, the function signature, test values and the

corrected result provided by the user to the tests file/database.
19. Perform Step 18 in Basic Flow.

Use Case 2: Run tests

Use Case UC2 : Run tests
Scope : AutoUnit - Automatic Random Regression Testing with Human Oracle
Level : User Goal
Primary Actor : User
Stakeholders and Interests :
User: Wants tests previously added to the tests file/database to be run

Preconditions:
The system is installed.

Success Guarantee (aka Postconditions):
Results for any tests in the tests file/database have been saved

Main Success Scenario (or Basic Flow):

1. The user opens a terminal / command-line window
2. The user enters the name of the command-line application
3. The user enters a command as a parameter to indicate they wish to run tests now
4. The system reads configuration options from the configuration file
5. The system runs each of the tests stored in the tests file/database
6. The system saves the test results to a file named according to the current date and time

Alternative Flows:

1a. A task scheduler configured by the user calculates that the interval between running tests has elapsed

2. The task scheduler runs the command-line application with an option as a parameter indicating that tests should be run

3. Goto Step 4.

5a. The system has not been configured (no configuration file exists).

6. A message is output to the application log and console (if available) stating that the system must be configured before tests are run. This could occur because:

- An attempt was made to run tests using a fresh installation of the application
- The configuration file has been deleted since the application was configured

Use Case 3: List tests

Use Case UC3 : List tests
Scope : AutoUnit - Automatic Random Regression Testing with Human Oracle
Level : User Goal
Primary Actor : User
Stakeholders and Interests :
User: Wants to see a list of existing tests

Preconditions:
The system is installed.

Success Guarantee (aka Postconditions):
A list of tests has been displayed.

Main Success Scenario (or Basic Flow):

1. The user opens a terminal / command-line window
2. The user enters the name of the command-line application
3. The user enters a command as a parameter to indicate they wish to see a list of existing tests
4. The user optionally enters a flag followed by a function name or a flag followed by a source-code path
5. The user hits enter/return
6. The system displays a list of tests on the console

Alternative Flows:

6a. The user entered an unrecognised value as a command-line parameter

7. A message is displayed to the user to tell them that an unrecognised parameter has been entered and an outline of the command-line syntax is displayed on the console.

6a. The user entered a flag followed by a path to a source-code file indicating that only tests contained within that file should be shown. The system displays any tests contained in the specified source-code file on the console.

6b. The user entered a flag followed by a function name indicating that only tests associated with functions of that name should be shown. The system displays any tests associated with functions of the specified name on the console.

Use Case 4: Configure testing

Use Case UC4 : Configure testing
Scope : AutoUnit - Automatic Random Regression Testing with Human Oracle
Level : User Goal
Primary Actor : User
Stakeholders and Interests :
User: Wants to set options which will dictate how tests are selected and how many of them are selected.

Preconditions:
The system is installed.

Success Guarantee (aka Postconditions):
The configuration file for the system has been updated or created.

Main Success Scenario (or Basic Flow):

1. The user opens a terminal / command-line window
2. The user enters the name of the command-line application
3. The user enters a command as a parameter to indicate they want to perform configuration
4. The user optionally enters flags (at least one must be entered) to:
    - Set the path for the compiler
    - Set the path for the linker
    - Set the default compiler flags/options
    - Set the default linker flags/options
    - Set the default number of tests to be generated for each function
    - Set a numeric value representing the maximum number of tests per run
    - Set a numeric value representing how tests are picked to run
5. The user hits enter/return
6. The system updates the configuration file
7. The system confirms to the user that settings have been updated

Alternative Flows:

6a. The user entered an unrecognised value as a command-line parameter

7. A message is displayed to the user to tell them that an unrecognised parameter has been entered and an outline of the command-line syntax is displayed on the console and the system exits.

7a. The user did not input any parameters to change settings. The system tells the user than no

settings have been changed and the system exists.

Use Case 5: Remove tests

Use Case UC5 : Remove tests
Scope : AutoUnit - Automatic Random Regression Testing with Human Oracle
Level : User Goal
Primary Actor : User
Stakeholders and Interests :
User: Wants to remove tests from the tests file/database

Preconditions:
The system is installed.

Success Guarantee (aka Postconditions):
The specified tests have been removed from the tests file/database

Main Success Scenario (or Basic Flow):

1. The user opens a terminal / command-line window
2. The user enters the name of the command-line application
3. The user enters an option as a parameter to indicate they wish to remove tests
4. The user enters a flag followed by a test ID or a flag followed by a source-code file path
5. The user hits enter/return
6. The system displays a list of affected tests and prompts the user to confirm the action
7. The system removes the test with the specified ID or tests associated with the specified source-code file from the test file/database
8. The system confirms to the user that the tests have been removed

Alternative Flows:

6a. The user entered an unrecognised value as a command-line parameter

7. A message is displayed to the user to tell them that an unrecognised parameter has been entered and an outline of the command-line syntax is displayed on the console and the system exits.

6b. No tests associated with the specified source-code file exist in the tests file/database

7. A message is displayed to the user to tell them that there are no tests for the specified source-code file in the database and the system exits.

6c. No tests with the specified ID exist in the tests file/database

7. A message is displayed to the user to tell them that there are no tests for functions with the specified ID in the database and the system exits.

Simple Examples

Configuration

The user has installed the system and wants to configure it so that they can start adding source-code to be tested.

Required, the user runs the application and:

- specifies the configuration command to set the compiler path
- specifies the configuration command to set the linker path

Optional, the user runs the application and:

- specifies the configuration command and flag to set the default compiler flags
- specifies the configuration command and flag to set the default linker flags
- specifies the configuration command and flag to set the default number of tests to be generated for each function
- specifies the configuration command and flag to set the maximum number of tests per run
- specifies the configuration command and flag to set how tests are picked to run

Once the two required values have been set the system considers itself configured.

Create and Run Tests

The user has a source-code file which contains functions they want to test.

The system must already be configured.

The user runs the application and specifies the command for creating tests followed by a flag and parameter which specify the source-code path.

For each function in the specified source-code the application:

- generates test values
- runs a test on the function with those values
- prompts the user to validate the result
- updates the result to what it should be (if the user rejected the result)
- saves the test in the test database for future use

List Tests

The user wants to cross-reference a listing of their source-code directory with the contents of the test database to see what proportion of their source-code is currently being tested by the application.

The user runs the application and specifies the command for listing tests with no flags.

The application outputs a list of all tests to the console which the user can pipe into a text document if they require it.

Remove Tests

A change has recently occurred where several source-code files have been merged. The test database therefore contains references to files that no longer exist.

The user wants to remove any tests corresponding to these non-existent files.

The user runs the application and specifies the command for removing tests followed by a flag indicating that a source-code path is being specified followed by that source-code path.

The application displays a list of the affected tests on the console

The application confirms with the user that it should remove the tests

The user confirms the action and the tests are removed from the test database

Appendix C: Supplementary Specification

Functional Requirements

Auditing

Each time the application is run it should record details of:

- Time the application started
- Time the application finished
- Any application errors which occurred (not to be confused with CppUnit test failures)

These details should be saved to an application log in the application folder in the form of a human-readable text document. A new application log should be written if the current application log goes missing (i.e. is deleted).

Localization

The application must use Unicode throughout in order to represent all character sets. This includes both internal representation of text and external e.g. UTF-8 or UTF-16 XML files. The XML files must be written with a byte-order mark (BOM).

Human language text displayed by the application to the user should be stored external to the application in XML format. The XML document should contain a section for each supported locale which contains each human language string and an ID. Terminology is not localizable and should be stored in the application itself. Examples of terms are: application name, command line flags and options.

Reporting

The application must generate a report each time tests are run. The report should contain details of:

- Time the application started
- Time the application finished
- Number of successful tests
- Number of failed tests
- Number of tests which could not be performed due to compiler/linker errors

For each test:

- Name of the test
- Signature of the function being tested
- Compiler/linker errors for the associated source-code
- Whether or not changes were detected in the source code
- Time the test was run
- Time the test finished
- Test values
- Test result
- If the expected result was returned

The report can be a simple text file or an XHTML document which can leverage our existing code for processing XML. The XHTML must be formatted so that it is human-readable as source.

System management

There are two important documents which the application stores in its application folder:

- Configuration File
- Test Database

Both of these files must be stored in human-readable XML format. This allows the system administrator to visually inspect the files and know what the values of settings are (in the case of the configuration file) and what tests are currently configured (in the case of the test database). This is useful if the configuration file and test database will be backed-up and restored. It also means that advanced users can copy and edit these files manually.

Workflow

All settings and commands available through the command-line interface must be individually accessible by a distinct command removing the need for interaction. This allows the configuration of the application and application commands to be scripted. For example, there should not be a single command “configure” which interacts with the user to prompt them to type each configuration value. Each setting should have its own distinct command e.g. “configure -compiler /usr/bin/gcc”

Usability, Reliability, Performance and Supportability Requirements

Usability

The command-line interface must provide the following:

- A usage screen for the user to learn application commands and options from the command-line. The usage screen should be shown when a command requesting help is used. When errors are found in input or when no input is provided a hint should be shown regarding the help command. Note: wxWidgets provides a command line parser which generates a usage screen automatically.
- Short and long versions of each command and option for example: “-c” and “--config”
- Detailed error messages regarding command line input so that the user can realize and correct mistakes easily.

Reliability

The system should be able to deal with missing files e.g. configuration file, test database, application log by alerting the user and creating new ones. This is a built-in / default behaviour as none of these files are present when the application is installed initially.

Note: The application will not be able to deal with a missing human language file, unless the default set of strings are hard-coded.

Supportability

Localization

As mentioned previously human language strings which the application uses will be stored externally in an XML file with sections for each locale e.g. “en-US”. The Ids associated with each string are common across locales. The localization engineer can easily add new translations using an XML editor by copying an existing section and updating the locale and string values.

Configurability

As mentioned previously:

- All commands and options can be accessed using distinct command-line inputs.
- Configuration settings are stored in an human-readable XML file

This allows the application configuration to be scripted and run on many machines or applied on one machine and copied to many others and also edited manually.

Compatibility

The application needs to be compatible with all major operating systems. The application will therefore use a cross-platform library (probably wxWidgets) to perform operating system specific functions. This will allow the application to be built for Linux, Windows and Mac OS X.

Design, Implementation, Interface and Physical Requirements

Design

The core functionality of the application must be written in the form of a library which the command-line interfaces documents and uses. This is so that the application can be used and extended to create plug-ins for IDEs and other tools.

Implementation

The application must be implemented using a cross-platform library so that it can be built and used on many operating systems.

Interface

The application must expose a software interface which can be used to access functions in the library described above.