Boiler++

Final Project Report (Draft)
6 December 2009

Sponsor: Cyberonics
Representative(s): Durga Kulkarni
Jim Mapel

Project Team: FUDG (Framework for Unit-testing Development Group)
Members:
Lauren Stuart
Nathanael Frost
Jennifer Long
Michael Noah
Bradley Van Dyk

Course: CS49000-020: Software Testing
Fall 2009

Instructor: Aditya P. Mathur

Department of Computer Science
Purdue University
West Lafayette, IN 47907

Table of Contents

A. List of Figures

B. Change Log

Version 1	Initial document / Interim Report 1: introduction, system functions, use cases, external interface requirements
Version 2, 9/29/09	Revised Interim Report 1
Version 3, 10/30/09	Interim Report 2: added design sections, task list, acknowledgements, changed formatting and look
Version 4, 11/6/09	Interim Report 2: improved design discussion and organization, added table of figures and change log, minor edits
Version 5, 11/23/09	Final Report Rough Draft: significantly expanded and reorganized discussion of system features implementation, made suggested edits from Interim Report 2 version 2.
Version 6, 12/4/09	Final Report Draft 2: text/content edits as suggested; modification of existing diagrams; addition of new diagrams, screenshots, user manual, and task list; removed Use Cases as redundant
Version 7, 12/6/09	Final Report Draft 3: formatting edits as suggested, extended/updated table of contents & list of figures

1. Introduction

1.1. *Purpose*

The purpose of this document is to define specific requirements, features, and functionality for the Cyberonics unit test framework under development by the student developer team. This document is meant to be reviewed by Cyberonics and the course instructor as a reference for using the product, determining whether the product meets their needs, and understanding the state of the product. This document provides a description of the entire project, including requirements, use cases, features, use and installation guides, and information about the development of the product. This document should be used as a reference for the project team members, the course instructor, and Cyberonics.

1.2. *Project Synopsis*

This project was created by Cyberonics for students of CS4900-020 Software Testing. Its end-product is a customized unit testing framework that developers can use with the Pocket PC 2003. CPPUnit is a full-featured open source unit testing framework and will serve as the foundation of the project. To meet the requirements of Cyberonics, CPPUnit will be ported into the Pocket PC 2003 environment and a customized interface for running tests will be built.

1.3. *Project Environment*

The framework (in this document referred to as “the testing framework”, “the framework”, or “Boiler++”) will be compiled with eMbedded Visual C++ 4.0 SP3 using the 2003 Pocket PC SDK. Unit testing and the test-running interface will be executed on the Pocket PC itself.

2. System Functions and Requirements

2.1. *System Functions*

2.1.1. Unit Test Framework

2.1.1.1 The main component of the software solution is a framework that will provide Cyberonics with ease of designing, developing, and executing unit tests.
Language: C++ using eVC++ 4.0 IDE
Operating System: Pocket PC 2003 (Testing Environment), Windows XP (Development Environment)

2.1.2. The test framework will allow Cyberonics to perform unit tests. A unit can be a group of files, a file, class or a function.

2.1.3. The test framework will integrate into the Cyberonics code base.

2.1.2. Unit Test Constructs

2.2.1. The framework will have unit test features common in most unit testing frameworks. It will:

- Run tests on units such as classes and methods
- Assert state of variables and methods
- Compute the elapsed time for each test to execute
- Manage test suites for automation

2.1.3. Test Automation

The framework will allow Cyberonics to select single or multiple tests to be executed as a single script. It will also allow Cyberonics to define test suites that can be executed when new code is updated. A test suite is a class that contains multiple tests.

2.1.4. Test Summary

After running multiple tests, the framework will enable a tester to generate a report with information including success/fail status, details of failures if any, timestamp for the tests, and the time taken to execute each individual test and all tests together.

2.2. *Nonfunctional Requirements*

2.2.1. Performance Requirements

CPPUnit maintains a reasonable responsiveness when operating in a Pocket PC 2003 environment.

The framework provides an accurate picture of current progress by supplying the user with information on the progress of the current test run.

2.2.2. Extendibility

The framework should accommodate future additions of features not implemented in the initial release due to limited development time.

2.2.3. Reliability

The framework must be able to handle test program faults and continue operation without causing failure in the testing application.

The framework cannot be the source of any fault or error.

2.2.4. Security Requirements

No security measures will be implemented in the testing application. This is due to the testing application only running in a desktop or Pocket PC environment. No security will be necessary since the desktop environment used by Cyberonics is considered secure.

2.3. *External Interface Requirements*

The user is able to define a test list, being able to include multiple tests from multiple test suites. After the user has generated a list of tests, they will be able to easily run this list of tests using the framework's test runner. During the testing process, the framework displays its current progress graphically. Upon completion of a test run or a user-aborted run, the framework displays a high-level report of pass/fail/(aborted) data, with further details available in a stored report file.

3. System Behavior

3.1. *Introduction*

Figure 1 shows the major modes of interaction with the testing framework. On the desktop side, the test developer uses the development environment (eVC++4, etc.) to create tests and deploy them onto the Pocket PC 2003 device or emulator. Once the tests are downloaded to the device or emulator, the tester runs tests by interacting with the Boiler++ GUI, which controls running of tests using the CPPUnit library (and our additions) through the Boiler++ backend.

The majority of our work is concentrated on the elements represented in Figure 1 (b).

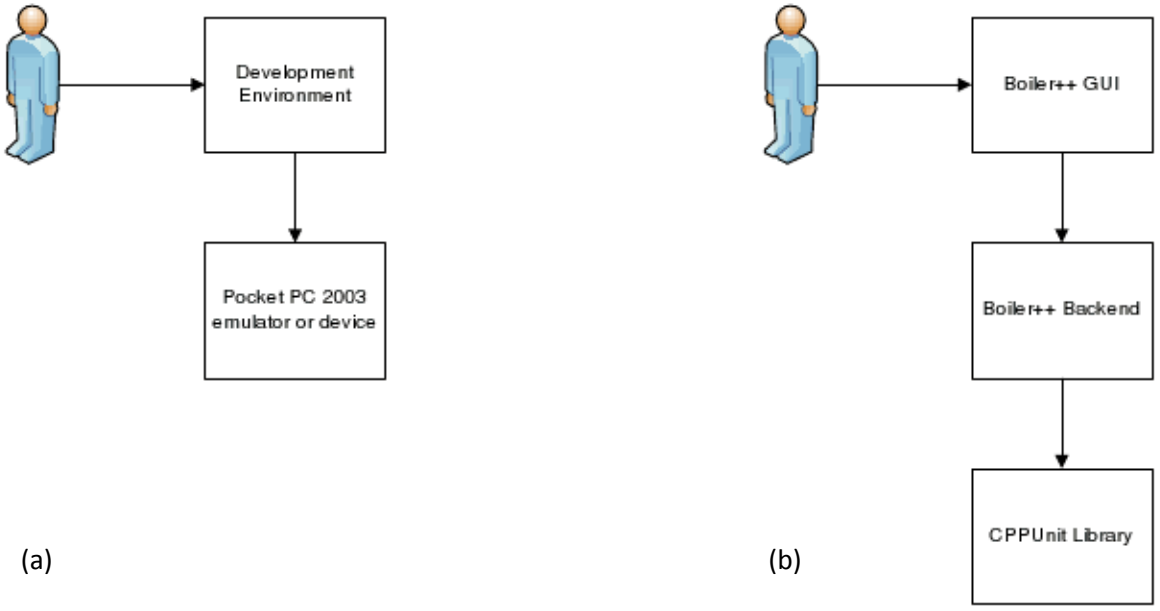


Figure 1. *Modes of user interaction with the testing framework*. There are two particular roles in using the testing framework: (a) that of the test developer on the desktop environment, and (b) that of the tester on the Pocket PC 2003 device or emulator.

In (a), the test developer interacts with the Development Environment to create tests, and the Development Environment compiles these tests then downloads them to the Pocket PC device – if an emulator is being used, the Environment starts up the emulator.

In (b), the tester interacts with the application GUI on the Pocket PC device or emulator. The GUI controls the intermediary (which we call the "Backend") between it and the test-running/reporting library.

3.2. *Select Tests*

On startup of Boiler++, the application presents a list of the current test cases and suites in the project. The list is presented in the form of a checkbox tree, which echoes the hierarchy of the test cases and suites. Figure 2 is a screenshot from the application which shows the checkbox tree. The user selects the test he or she wishes to run by checking the boxes (selection of a suite automatically selects the suite's entire contents for testing) then clicks a button to run the tests.

Figure 2. *The test selection checkbox tree.* Test cases and suites are displayed in a checkbox tree that represents their hierarchy: the *TestMemorySmoke* entries are individual test methods which are members of the suite *CritterTest3*. It is possible to select a family of tests by selecting the root of the family; in this case, to run all of the *TestMemorySmoke* tests in *CritterTest3*, it suffices to only select *CritterTest3* by checking its box. Likewise, checking the *All Tests* box selects all tests.

In CPPUnit, test selection is normally done programmatically. In order to provide a user interface to select tests, several new features were added. A logical breakdown of the test selection feature follows.

3.2.1. Display Behavior

The user interacts with the checkbox tree to indicate which tests or suites must be run. In some cases, simple selection is augmented by helpful selection from the interface: if all of a node's descendants are selected, it is logical to say that the user has also selected the node. If only some of a node's descendants are selected, the user should be able to see that this is so without fully expanding the tree.

3.2.1.1 Classes

Details of class interactions in this feature are illustrated in Figure 3.

PrototypeDlg This is the class for the dialog window in the testing framework, which is the component the user interacts with. When the user clicks a checkbox in the tree, PrototypeDlg determines which node was clicked and passes that on to TreeCheckBox.

TreeCheckBox This class is original to the project, and is the display class for the test selection checkbox tree. When TreeCheckBox receives the clicked node, it toggles the selection status of the node and updates the node's children and parent if needed.

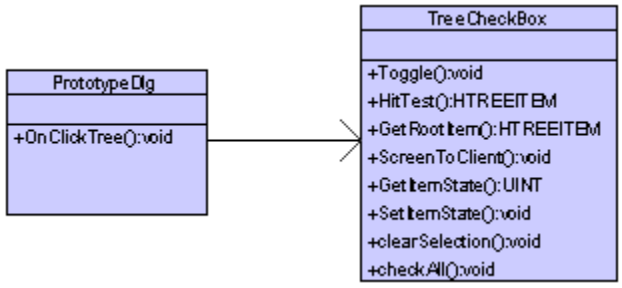


Figure 3. *Extended checkbox functionality: class hierarchy.* Two main classes (colored blue here because they are FUDG original classes) interact to provide the extended checkbox behavior.
The arrow denotes an association: PrototypeDlg manipulates TreeCheckBox.

3.2.1.2 Function

Upon user interaction with a checkbox, the parent and children of that checkbox are recursively updated. This means that placing a checkmark on a suite will update all of its children with checkmarks, and will update its parent to partially-selected. If all of a node's children are selected, the node will be marked as selected. Similarly, if a suite is unselected, all of its children will be unselected. If a parent is marked as selected, and one of its children is deselected, then the parent will become partially selected as long as at least one of its children is still selected. The tester may clear all selections by choosing "Clear Selection" from the application menu.

The different states of a checkbox are illustrated in Figure 4, and Figure 5 details the component interactions involved in the functionality.

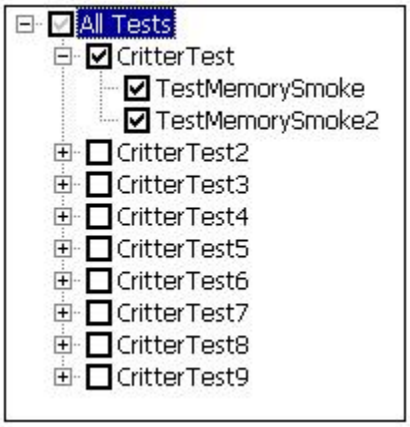


Figure 4. *Illustration of checkbox states.* Here we see extended behavior of the checkbox tree. *CritterTest* has two test cases, *TestMemorySmoke* and *TestMemorySmoke2*, which are both selected. As

a result, *CritterTest* is also displayed as "selected" so that the tester may see that all its component test cases are selected without expanding the *CritterTest* subtree. The display looks exactly the same as if the tester had selected *CritterTest* alone; the two children are updated to show as "selected". Next, because not all of the children of *All Tests* are selected, the *All Tests* checkbox is a gray check, which indicates "partially selected".

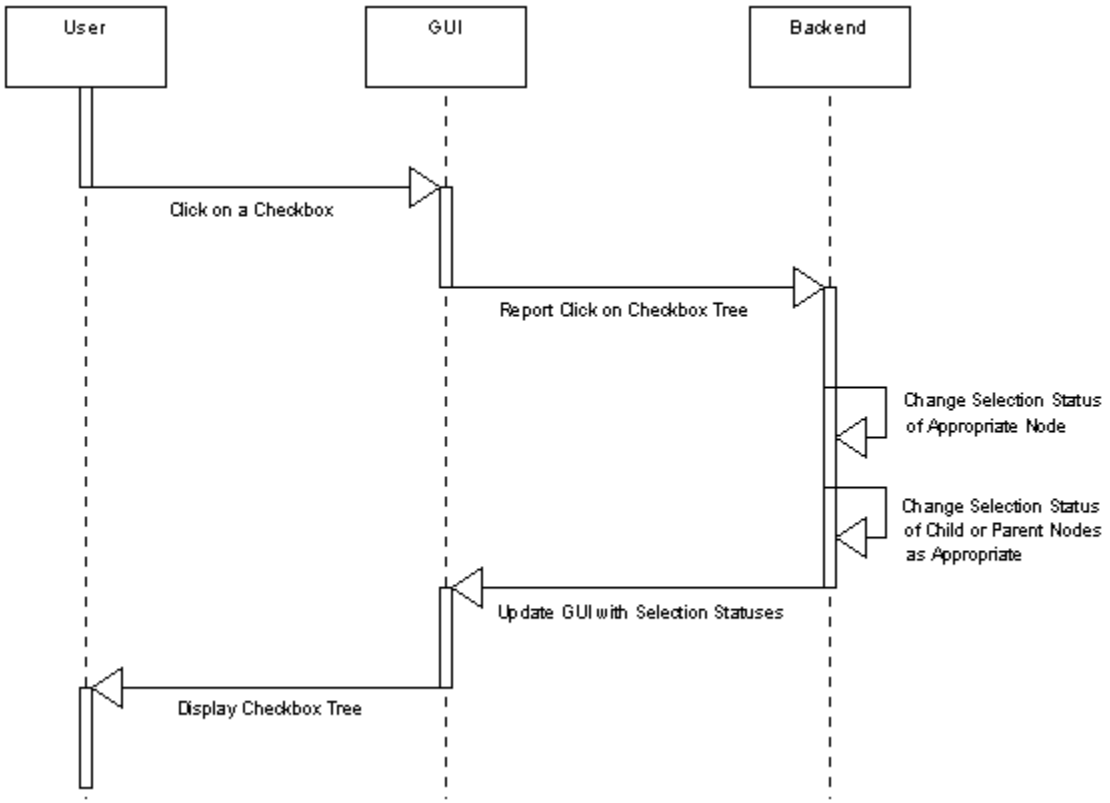


Figure 5. *Interaction among system components in extended checkbox functionality.* There are three logical layers involved in the checkbox functionality: the *User*, the application *GUI* with which the User interacts with the application, and the *Backend*, which handles the logic of updating checkbox statuses.

3.2.2. Saving a Selection Set

If the project has a large number of tests, it is convenient to be able to save a set of tests selected for ease of use the next time the set must be run. These sets can also correspond to, for instance, subsections of the code exercised, testing criteria to be met, or steps to reproduce a bug.

3.2.2.1 Classes

Details of class interactions in this feature are illustrated in Figure 3.

PrototypeDlg This class manages the file-saving dialog and coordinates the save operation.

TreeCheckBox The checkbox tree examines itself for selected leaves (nodes with no children are test cases) and writes the names of those tests to the file passed by the PrototypeDlg.

ZVector<TestItem> The collection of TestItems that holds suites and tests which the checkbox tree represents. TreeCheckBox iterates over it to find selected tests or suites (and drills down into the suites until it gets to test-case level).

TestItem This class represents a test case or suite in the checkbox tree; it is examined for its selection status and, if it is selected and has no children (leaf nodes are test cases only), its test's full name.

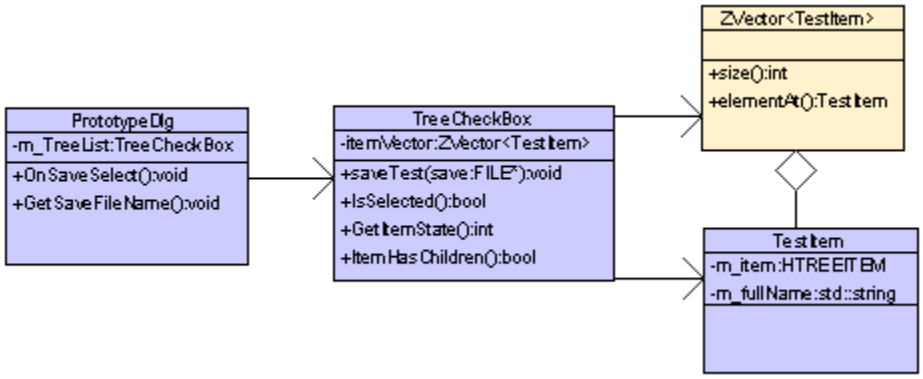


Figure 6. *Save selection set: class hierarchy.* Four main classes work together to deliver the save functionality.

3.2.2.2 Function

After the user has finished selecting the tests to be run, the user chooses the "Save Selection" option from the Selection Menu (illustrated in Figure 7). This will bring up a "Save As" dialog where the user is prompted to provide a filename for the test selection set. If the user provides a filename that already exists, the system will ask the user if he or she wishes to overwrite the existing file. The project will then open a file for writing and write the names of the selected tests into the file. Upon completion, the program will return to the test selection tree. Figure 8 illustrates the system component interactions involved in saving a selection set.

Figure 7. *The Selection menu.* The Selection menu is located in the menu bar of Boiler++. Here, the *Save Selection* menu option is selected. Only a portion of the entire screen is shown here.

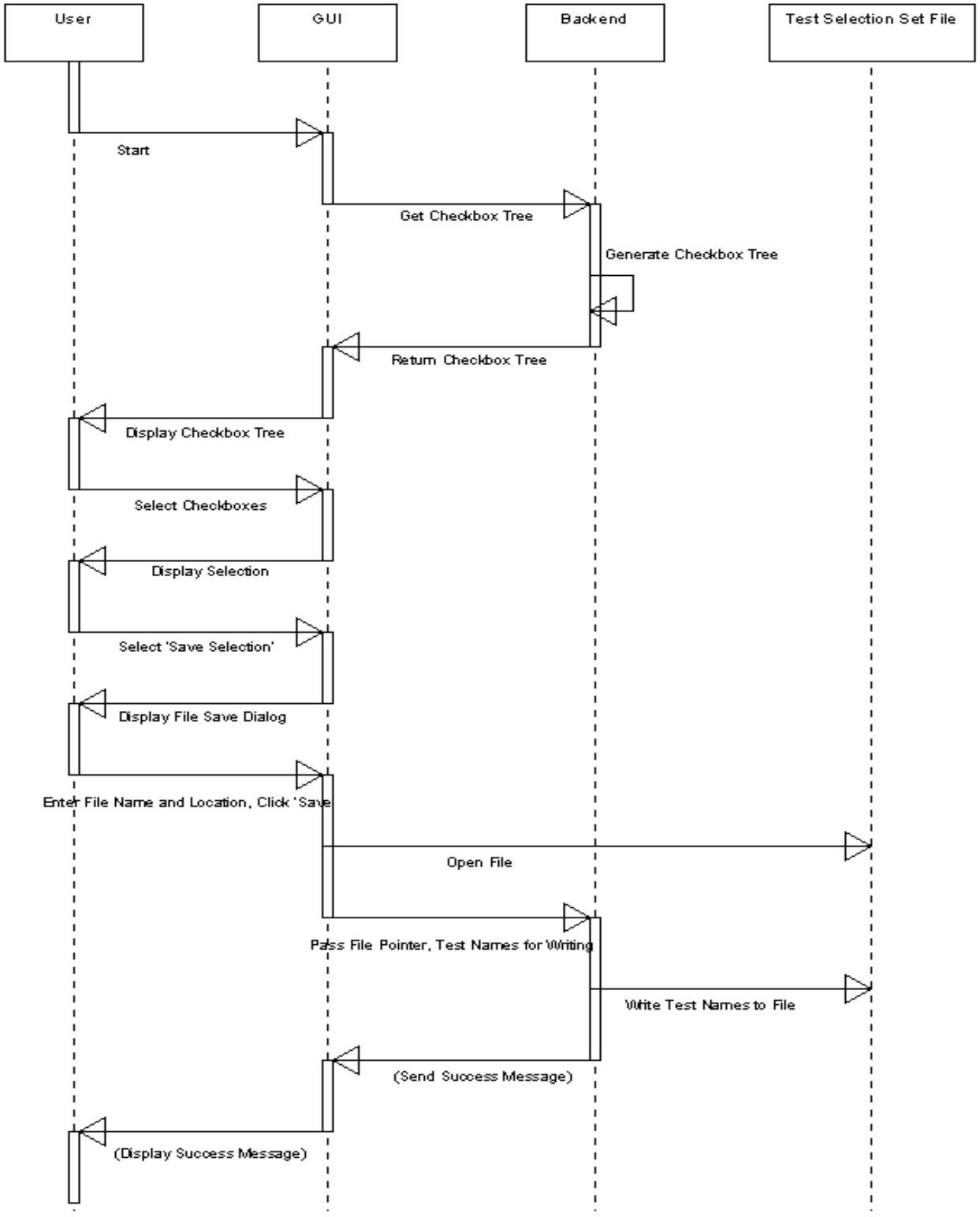


Figure 8. *Interactions among system components during selection set saving.* There are four logical layers in the operation of saving a test selection set. Three are recognizable from the previous interaction diagram; the new layer is the file to which the test names are written. The Boiler++ Backend deals with it directly.

3.2.3. Loading a Selection Set

If a particular test set (or combination of sets) is of interest, the user saves time by being able to quickly select sets. The load function makes it possible to re-use test selections without re-selecting the tests afresh.

3.2.3.1 Classes

Details of class interactions in this feature are illustrated in Figure 9.

PrototypeDlg Upon selection of the Load option, OnLoadSelect is called. This will open a file for reading as specified from the user and pass the file pointer to loadTest in the TreeCheckBox class.

TreeCheckBox LoadTest will read the names of the tests in the file and search our list for the test. Once found it will call SetCheck to place a checkmark on that test. SetCheck will then call updateChildren and updateParent to maintain the proper checkbox display behavior.

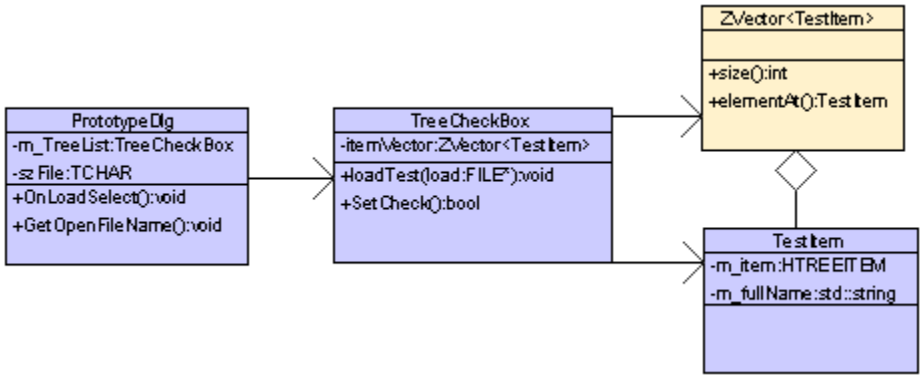


Figure 9. *Load selection set: class hierarchy*. The interactions in this feature are very like those in the save feature.

3.2.3.2 Function

At any time during the test selection process, the user may select “Load Selection” from the Selection Menu (also depicted in Figure 7). This will bring up an “Open” dialog that will list all of the previously-saved test selection files. When a user selects a file, the program opens it, reads the test names from it, and checks the appropriate checkboxes in the test tree. No checkboxes are unchecked during this process, so any test that was selected by the user prior to loading a test selection set will remain selected. This allows the user to load multiple saved test files so as to run the sets together. When a selection set has been loaded and the appropriate tests marked as selected, the program will display a confirmation box, which shows the number of tests successfully marked and the number of tests not found in the tree. The class interactions which deliver the load functionality are depicted in Figure 10.

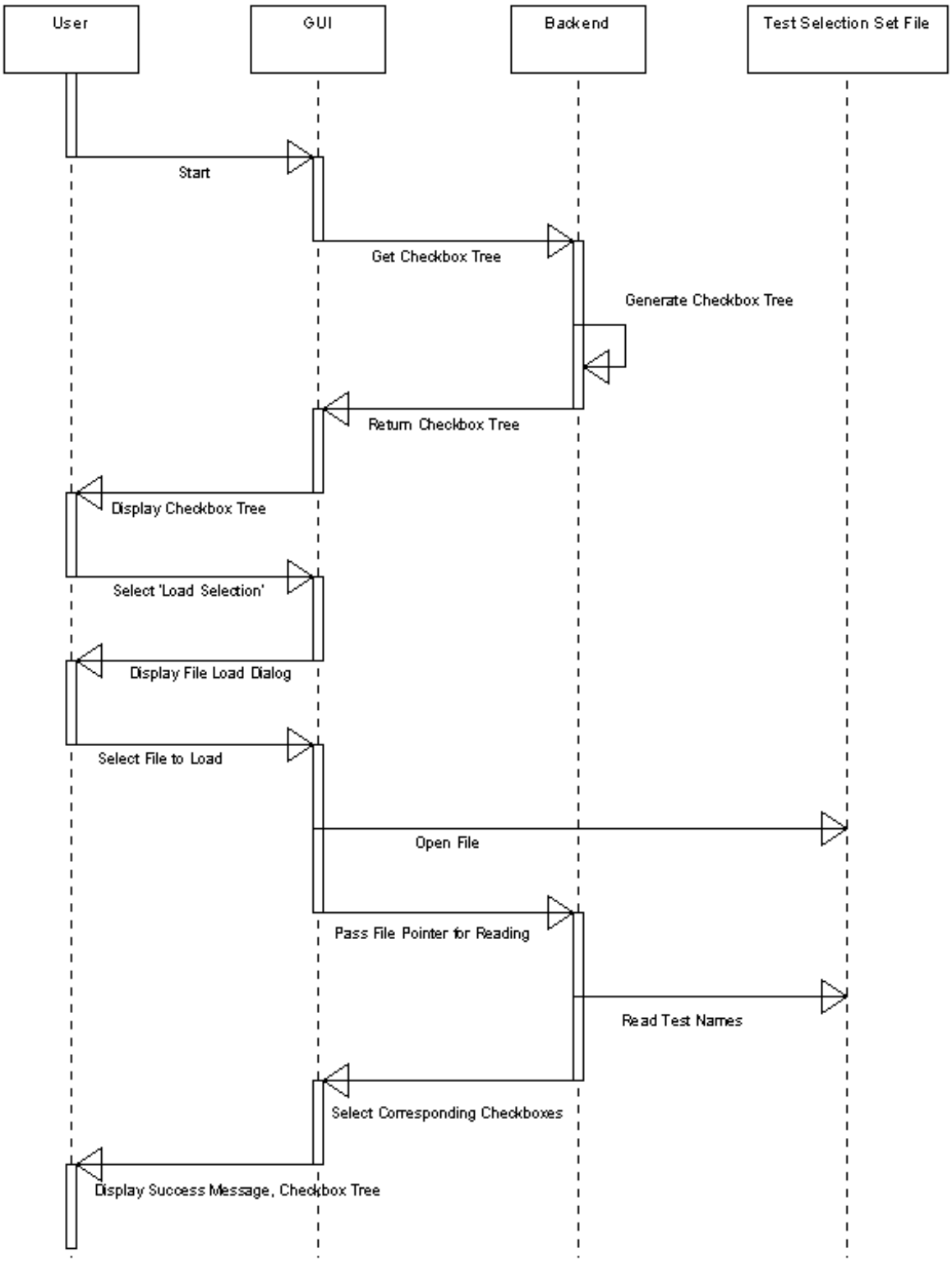


Figure 10. *Interactions among system components during selection set loading.* The actions here are much like those of the save operation, except that the interaction with the test selection set file is in reading test names out rather than writing them in.

3.2.3.3 Faults, Anomalies, and Enhancement

Because of the recursive nature of the process used to update checkbox statuses (according to the number of children selected), the number of tests being loaded affects the amount of time required to load the list. *(further content pending due to ongoing development)*

3.2.3. Registering a Selection Set

A selection of tests is a subset of the entire set of tests registered with the test runner. Some manipulation is required in order to run only the desired tests.

3.2.3.1 Classes

Classes involved in this function, and the relationships between them, are illustrated in Figure 11.

PrototypeDlg When Start is clicked, this class begins the process of running the tests.

TestRunner This class is responsible for coordinating the test run.

TestFactoryRegistry This class creates test registries, which are lists of tests to pass to TestRunner to manage.

TreeCheckBox This class is original to the project, and is the display class for the test selection checkbox tree. The items in the tree are TestItems.

TestItem This is another original class which links a test case to its representation in the selection tree.

ZVector This is a generic collection class, created by Craig Muller, which is used to store the individual tests that were selected by the user.

Test This is the class that all test cases and test suites inherit from. The class allows access to a suite's children (test cases).

Test Suite This class inherits from Test; it has the access method used to add tests to the test registry.

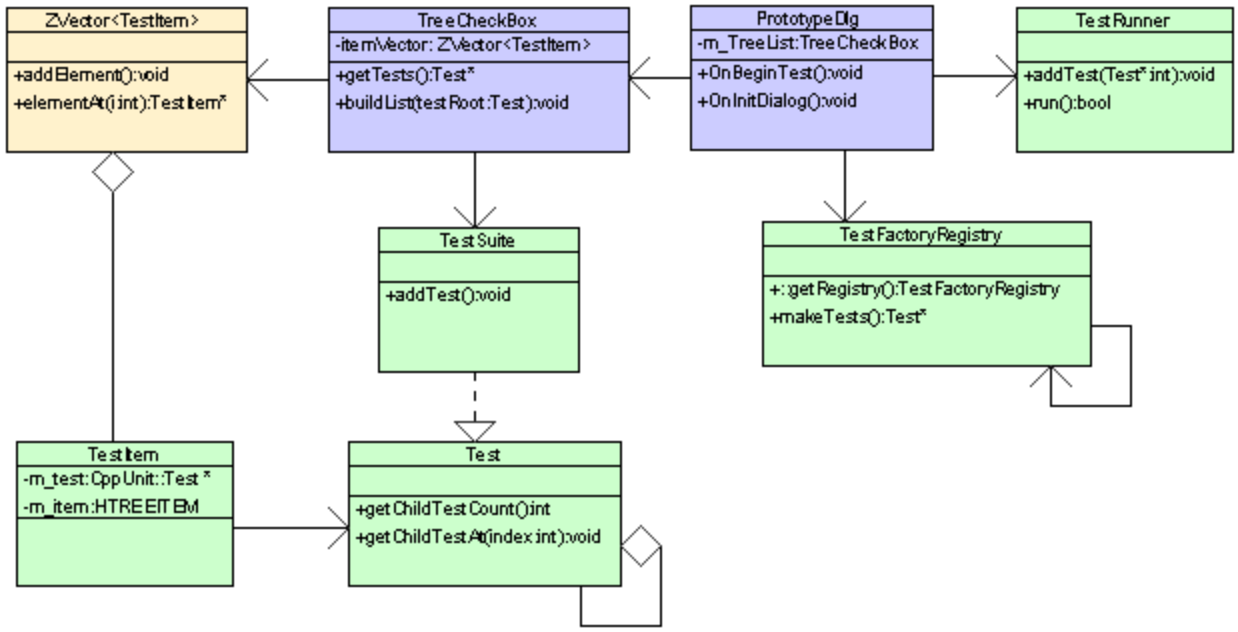


Figure 11. *Test selection: classes hierarchy.* There are several CPPUnit, original, and third-party classes (colored in green, blue, and orange respectively) interacting to provide the test selection functionality. Here we only show those classes that are directly involved with test selection and the subsequent required operations to create a registry of only the selected tests to run. An arrow denotes an association; a dotted line with a triangle: a generalization; a diamond: a collection (for instance, ZVector is a collection of TestItem objects).

3.2.3.2 Function

When the user clicks the "Run" button, the application creates a new test registry out of the selected tests. It examines the tree, accesses the selected tests through their links in the TestItem objects in the selection tree, and then adds each of those tests to a new registry. The test runner takes the registry, runs the tests, and reports back the results. The interaction of all the layers is broken down sequentially in Figure 12.

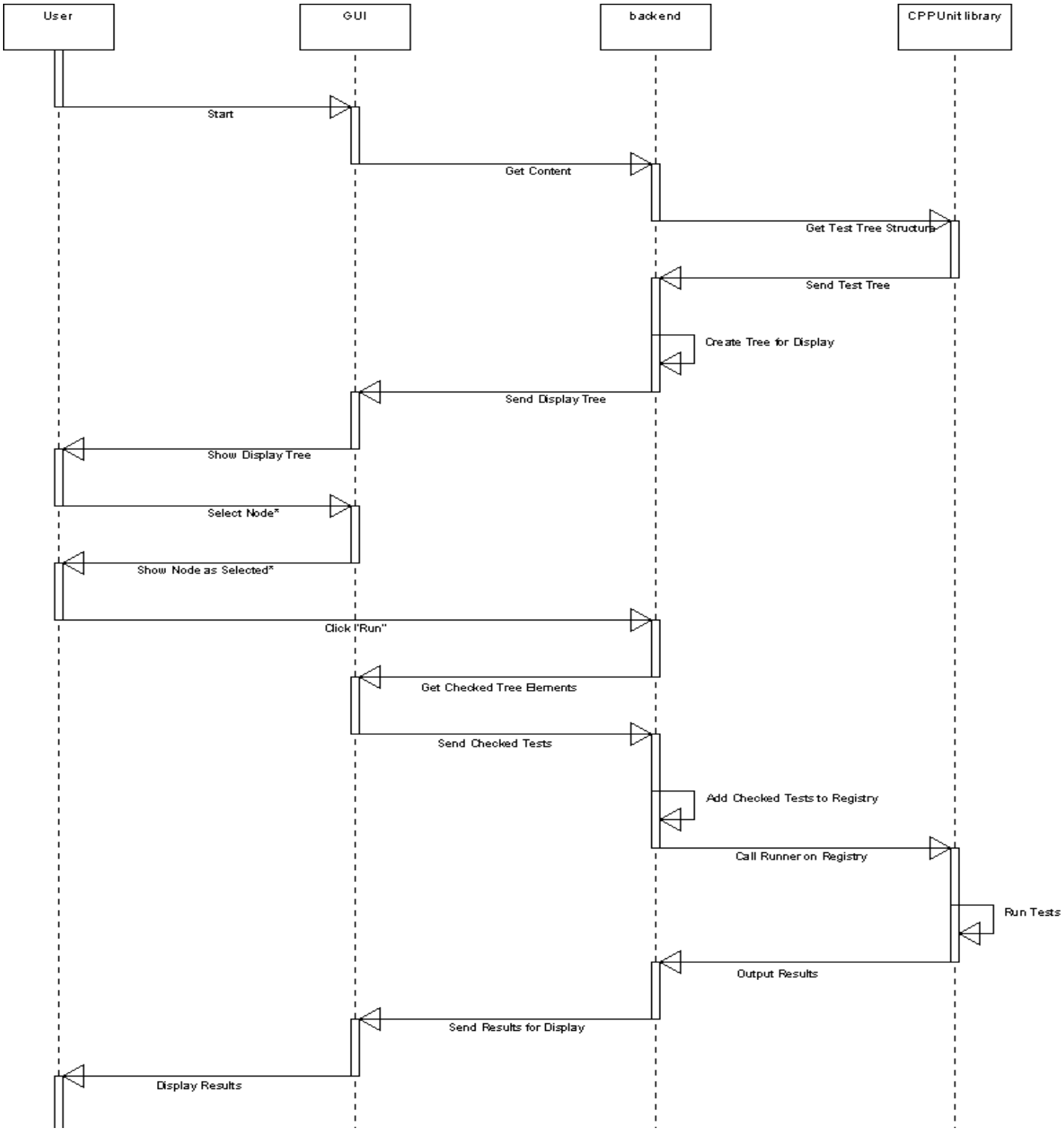


Figure 12. *Interaction among system components during test selection*. There are four logical layers of interaction in test selection: the User, the GUI the user directly interacts with, the Boiler++ backend intermediary between the GUI and the test-running library, and the CPPUnit library itself, which handles test running.

3.3. *Run Tests*

3.3.1. Progress Bar

The progress bar displays how many tests are completed out of the total number of tests being tested.

Figure 13. *The progress bar during running tests.* As tests are executed, the progress bar is updated to show how far along the test run has progressed.

3.3.1.1 Classes

The interactions and hierarchy of these classes is detailed in Figure 14.

TestRunner This is the class that runs the tests. TestRunner also owns a TestResult object which coordinates the tests and listeners.

TestResult This is the class which calls the individual tests and communicates with the test listeners. We will refer to it most often as “Event Manager” (the name of the variable in TestRunner), which is a more descriptive name for its function.

TestListener This is an interface which provides a template for test listeners. Listeners keep track of test information such as pass/fail status.

FUDGProgressBarListener This class is added to the TestRunner as a listener, inheriting from the TestListener interface. The GUI adds a function pointer to this listener which will be called at the end of each test case. The function passed has access to the progress bar.

PrototypeDlg This class initializes the progress bar object and test runner, and owns the function mentioned above which updates the progress bar.

CProgressCtrl This is the MFC wrapper for the progress bar. This class provides methods to set the limits and position of the progress bar.

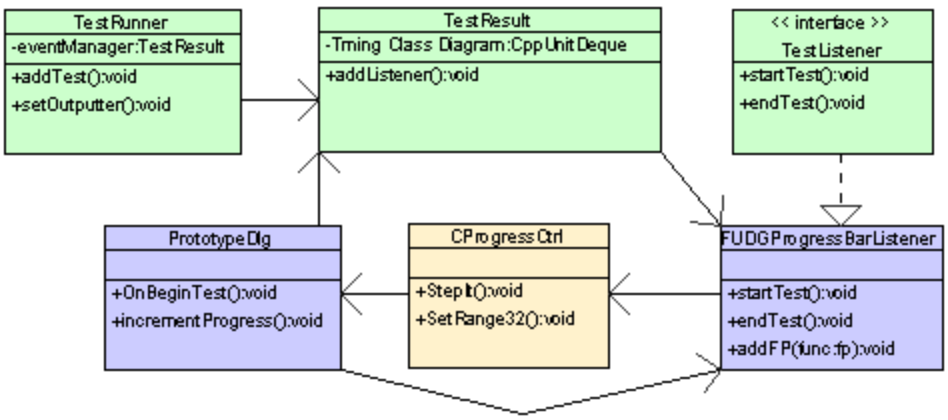


Figure 14. *Progress bar: class hierarchy*.

3.3.1.2 Function

Once the start test button is clicked the progress bar becomes visible starting at zero percent. After a test case is complete the progress bar is incremented by the percentage of one test case out of total test cases. Once the tests are complete the progress bar disappears.

(figure forthcoming)

Figure 15. *Interaction among system components updating the progress bar.*

3.4. *Report Results*

When the tests are run, the test runner keeps statistics on the tests. It is desirable for those statistics to be put into a report that is easily readable. To achieve this goal, reports are generated in XML form (Figure 16, formatted output in Figure 17). XML reporting was chosen for a few reasons. First, it is easy to extend what is reported by CPPUnit's XMLOutputter through the use of an XMLHook which can add features such as timing to the results. Next, formatting is simple with XSL stylesheets, which adds to the ease of making large sets of results readable. Display of selected information immediately after the test run – which we will refer to as "immediate results" – provides the user a summary of statistics on the screen of the testing application, communicating which tests have passed and failed without requiring the user to port the full XML results to a desktop computer for viewing.

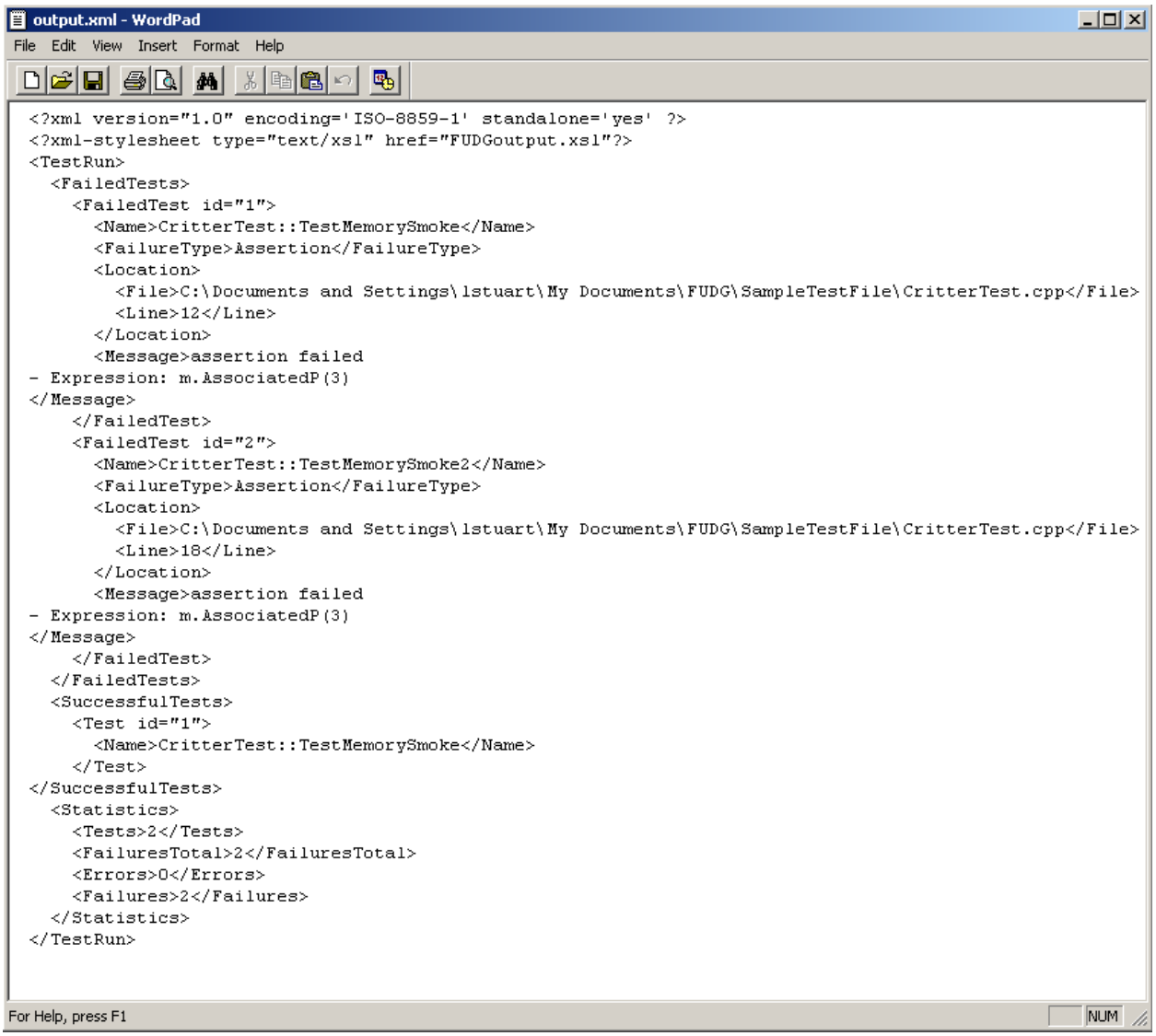


Figure 16. *Raw XML output.* The XML-formatted results include information on each test case such as pass/fail status, failure location, and error message. There are separate subtrees for failed tests, successful tests, and overall test statistics.

(figure forthcoming)

Figure 17. *Formatted test run report as Viewed on desktop.*

3.4.1. Test Timing

CPPUnit does not offer timing information about tests; however, one of the options for results output was in XML form. The XML output can also be added to by using XML hooks, which we use to add the timing information to each test's individual report. The information about each test is gathered by listeners during the test run and used in output after the run is over (note timing data in Figures 16 and 17); we added a listener for the timing information.

3.4.1.1 Classes

Classes involved in this function, and the relationships between them, are illustrated in Figure 18. This function shares

TestRunner This is the class that runs the tests. Here, we need to set its outputter in order to have XML-formatted results. TestRunner also owns a TestResult object which coordinates the tests and listeners.

TestResult This is the class which calls the individual tests and communicates with the test listeners. We will refer to it most often as "Event Manager" (the name of the variable in TestRunner), which is a more descriptive name for its function.

TestListener This is an interface which provides a template for test listeners. Listeners keep track of test information such as pass/fail status.

FUDGListener This class implements the TestListener interface and keeps track of time for each test and for the whole test run. In the future we may extend it for other purposes related to monitoring test runs.

XmlHook This is another interface, which provides the XML output writer access to the information that a listener has collected.

FUDGXmlHook This class implements the XmlHook interface and specifically hooks into the FUDGListener class for timing information.

XmlOutputter This class writes the results output. It uses XmlHooks to gather information to write.

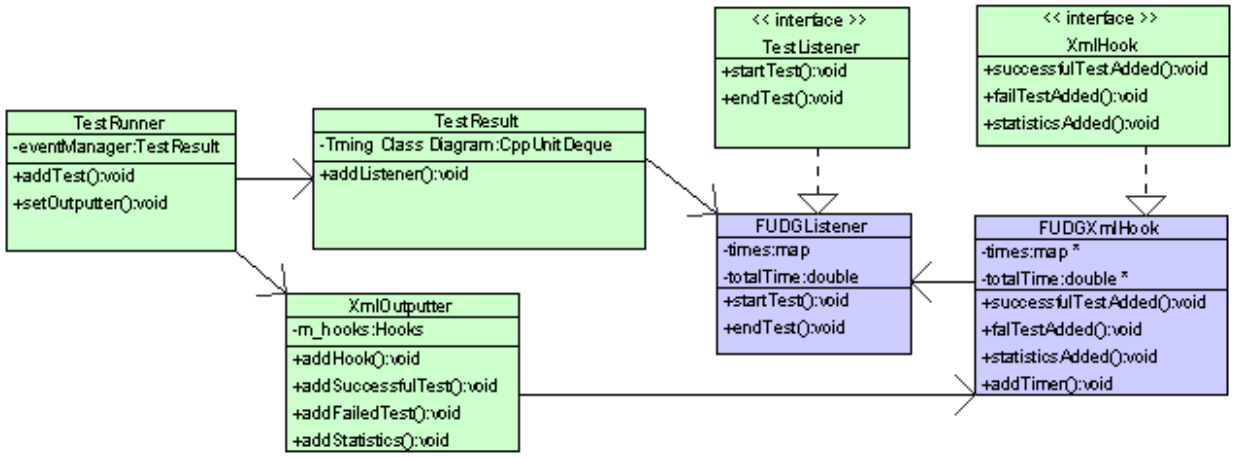


Figure 18. *Test timing class hierarchy*. There are several CPPUnit and original classes (colored in green and blue respectively) involved in running and reporting on tests. Here we show only the classes directly involved in the timing functionality.

3.4.1.2 Function

When the TestRunner starts, it has an outputter specified, a registry to run, and a set of listeners added to the Event Manager. The Event Manager goes through its list of tests: it starts a test, starts a listener, and then ends a listener when the test has ended. The listeners are specialized; we have the FUDGListener keeping track of timing, and a TestResultCollector which keeps track of which tests were successfully run and which failed. When all the tests have been attempted, the Event Manager returns control to the Test Runner, which begins to write output. The XML outputter accesses the

TestResultCollector information then uses its XML hooks to add the other information to the XML results output file. This process is illustrated in Figure 19.

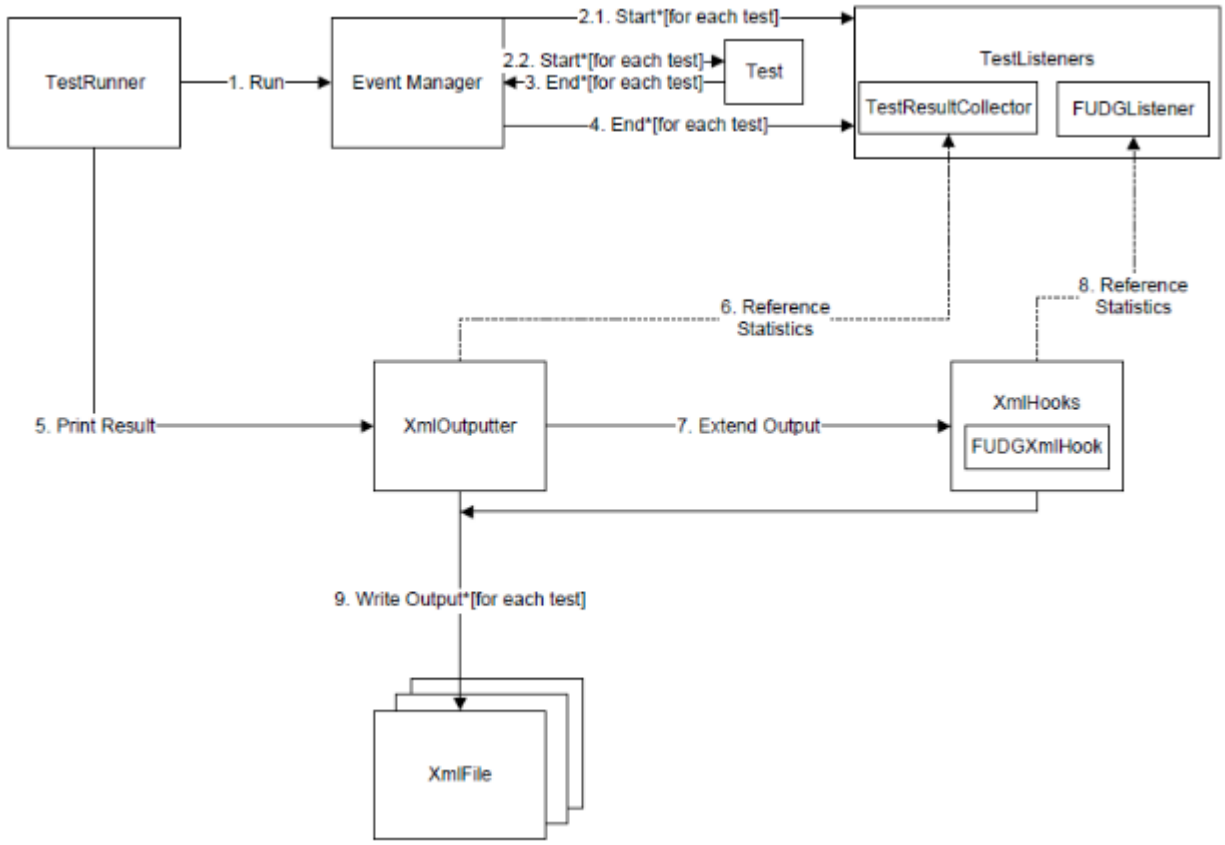


Figure 19. *Interaction among system components during test running.* The process of running and reporting on tests involves several objects and classes; here we show the process from when the Test Runner is started to the writing of the XML output file. The steps are numbered sequentially; steps which share numbers are part of the same action (for instance: '2.1. Start' and '2.2. Start': test listeners are started alongside tests).

4. Problems, Conflicts, and Resolution

4.1. *Testing Framework Selection*

The first issue we encountered was finding a framework that fulfilled Cyberonics requests and would work in eMbedded Visual C++ 4.0 (eVC4) with minimal or no changes to the chosen framework.

4.1.1. Approach

To find the right framework we had to add some additional requirements to remove the ambiguity of making our decision: high usability, low setup time in other environments, presence of existing documentation, and amount of available features. With better guidelines in place we constructed a list of potential frameworks to investigate further.

4.1.2. Solution

After completing research we decided on CPPUnit because it has the best user history, available features, and available documentation of all the frameworks we investigated. Specifically, we chose CPPUnit over WCEUnit due to the built-in functionality already available.

4.2. *Development Environment Setup*

After selecting our framework we needed to set up our development environment. From communication with Cyberonics we knew that we were to use eVC4 with the Pocket PC 2003 emulator. Once we figured out the setup, we also needed a dedicated development environment in which to work, as the machines in our chosen lab would reset every night and erase all our setup work.

4.2.1. Approach

At first during installation we encountered Windows XP blocking the Pocket PC 2003 emulator from running. After that we needed to decide which Service Pack was sufficient for getting CPPUnit to work, without causing issues for Cyberonics. We also needed to run a project to see if our installation was successful. Lastly, we needed to work with the CS Facilities team to establish a persistent environment in the Windows lab.

4.2.2. Solution

We found information on why Windows XP was blocking the emulator from running, and adjusted the registry accordingly. Next, through research on the different service packs, and what Cyberonics currently uses we deemed Service Pack 3 to be sufficient for our development. Then, we used the example “Hello World” program provided, and after some investigation on the emulator we finished our installation by running the program. Lastly, we provided setup instructions to and met with CS Facilities staff to ensure that we had the desired environment.

4.3. *CPPUnit Port into eVC4*

CPPUnit is not designed to work in eVC4, and we needed to find out how to make it compatible.

4.3.1. Approach

Our first goal was to set CPPUnit up as a project in Visual Studio 2005/2008 in order to get an idea of what was involved. Even when following steps available online, this proved to be very difficult. Once we started trying to get CPPUnit to compile in eVC4, we found that the lessons we learned in Visual Studio did not apply to eVC4. Another issue we encountered involved us trying to create a Dynamic Library File (.dll) instead of a Static Library File (.lib). Lastly we found out that in order to use CPPUnit we would need the RITTI library.

After compiling the library another conflict arose: how to use the library in a testing program. Once again we could not find good resources on what we could use in eVC4 versus what was needed to run a CPPUnit test.

4.3.2. Solution

After much experimentation we used the information learned from the Visual Studio 2005/2008 documentation to figure out that a Static Library File would be more appropriate for our needs over a Dynamic Library File: a Static Library File is easier to develop and to include in development. With the aforementioned documentation we found instructions on how to install the RITTI libraries into the eVC4 environment. With the RITTI libraries and the decision to go with a Static Library File, we were able to compile the CPPUnit library.

When trying to figure out how to create and run a test, we searched for existing C++ projects that would work in the eVC4 environment. After some searching we found a project that was compatible with eVC4 and successfully ran a CPPUnit test.

4.4. *Parallel Development Project Merge*

In the beginning of development, we had two versions of the project being developed – one team was working on the XMLOutputter additions and one on developing the GUI. When we tried to merge these two development paths, we encountered a bug with a library that was being used in the XMLOutputter which could not be used with the MFC GUI.

4.4.1. Approach

A library for using STD::Map (an implementation of a hash map) was not compatible with the MFC compiler, leaving us with the options of: trying to fix the linking error, finding another hash map library that is compatible with MFC, or not using a hash map.

4.4.2. Solution

After many unsuccessful tries to fix the linking errors, we decided to no longer pursue that course of action. Due to eVC4's compatibility issues with our first choice of a hash map implementation we decided to use a linked list instead for simplicity.

5. Acknowledgments

· Kip Granson and Melanie Church with CS Facilities, Purdue University, for setting up machines for our use

· Dr. Aditya Mathur, Software Testing professor, for his guidance and insight

· Sourceforge.net for CPPUnit documentation and the CPPUnit community for tutorials and setup instructions: http://sourceforge.net/apps/mediawiki/cppunit/index.php

· TopCoder for the TopCoder UML Tool, which we used to make some of our diagrams: http://www.topcoder.com/wiki/display/tc/TopCoder+UML+Tool

· Craig Muller at CodeProject, for ZVector: http://www.codeproject.com/KB/trace/zvector.aspx

· Sean Laxton with ITaP (Information Technology at Purdue) for coordinating our videoconferences with Cyberonics

6. References

1. **Pocket PC 2003**

Emulator: http://www.microsoft.com/downloads/details.aspx?familyid=57265402-47A8-4CE4-9AA7-5FE85B95DE72&displaylang=en

2. **CPPUnit**

Main Page: http://sourceforge.net/apps/mediawiki/cppunit/index.php?title=Main_Page

3. **eMbedded Visual C++ 4.0**

Download: http://www.microsoft.com/downloads/details.aspx?FamilyId=1DACDB3D-50D1-41B2-A107-FA75AE960856&displaylang=en

Developer Resources (for use with an emulator): http://msdn.microsoft.com/en-us/library/aa446907.aspx

eVC4 Service Pack 3

Download: http://www.microsoft.com/downloads/details.aspx?FamilyId=5bb36f3e-5b3d-419a-9610-2fe53815ae3b&displaylang=en

Information: http://msmobiles.com/news.php/2314.html

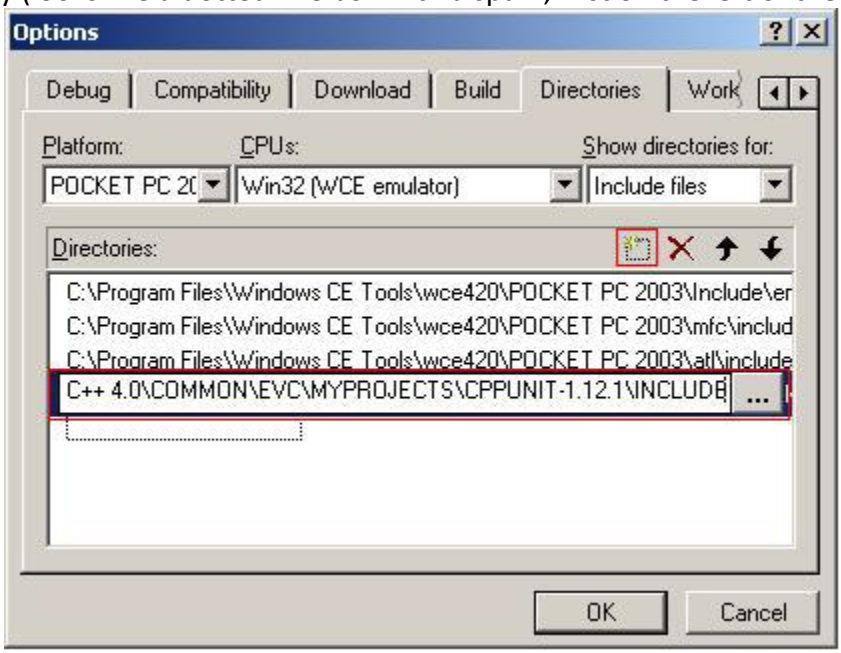
Appendix A: User Manual

1 eVC4 Environment Setup

This setup needs to be done on each computer that will be running CPPUnit.
1. Download the file in this link:
http://support.microsoft.com/default.aspx?scid=kb;en-us;830482
Copy the Ccrtrtti.lib and Ccrtrtti.pdb files from the Emulator folder in the RITTI download to the folder that is named Emulator inside the \Lib folder of the Pocket PC 2003 SDK.
The default path name for computers with Pocket PC 2003 SDK is:
"C:\Program Files\Windows CE Tools\wce420\POCKET PC 2003\Lib"

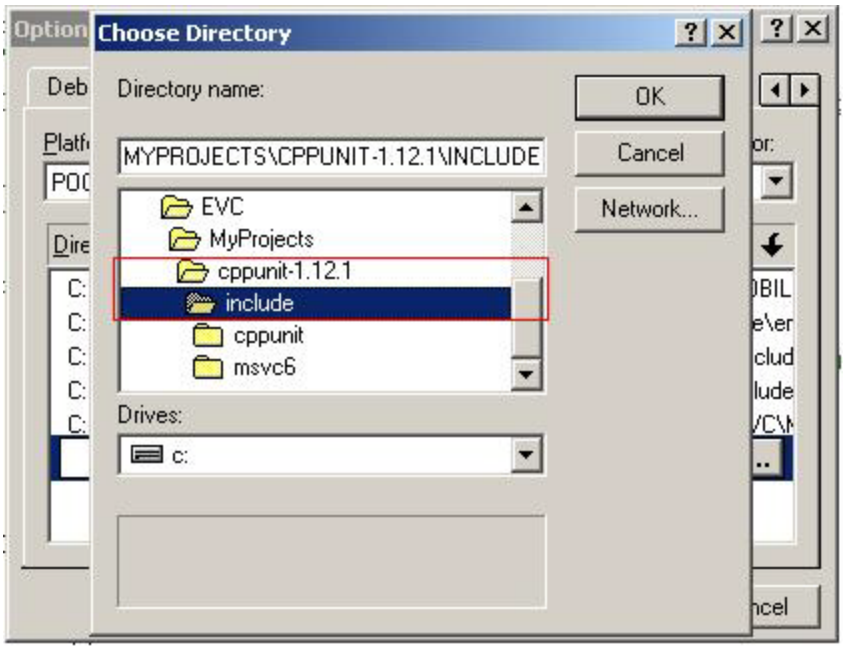
- Detailed instructions are located in the provided link if they are needed.

2. The CPPUnit include file needs to be added to the execution paths:
Go to menu options Tools->Options. Then go to the tab Directories.
Leave CPUs box on Win32(WCE ARMV4) option (or change to this option if not selected):
*****Add picture for ARMV4 option********
Add a New entry (looks like a dotted line box with a spark, first on the left of the icons)

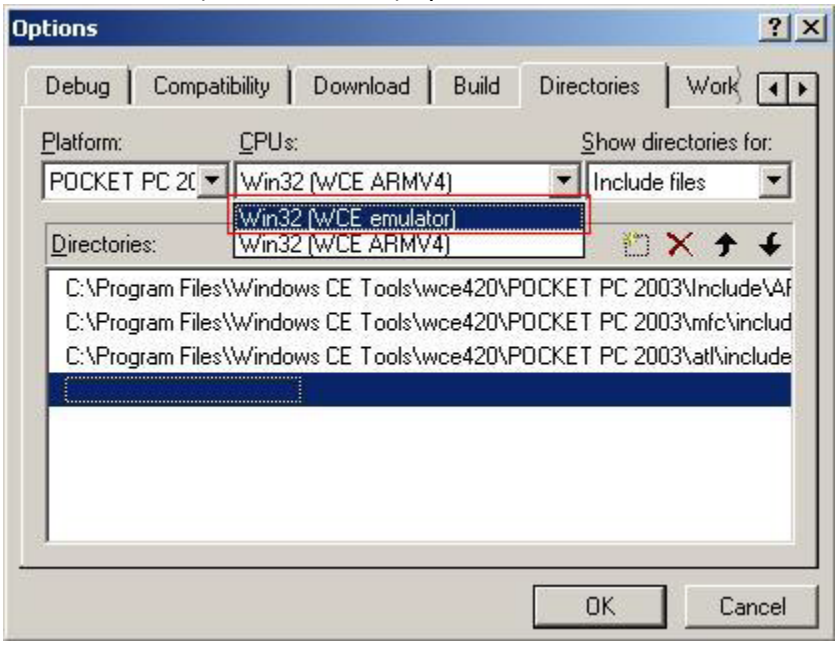


Browse for CPPUnit include folder. This folder will be located where the Boilerpp source files are, or where the CPPUnit source files was unzipped to.

- Example: C:\Documents and Settings\Desktop\Boilerpp\cppunit-1.12.1\include.



Repeat the previous steps for the Win32(WCE Emulator) option:
Change CPUs box to the Win32(WCE Emulator) option:



Repeat rest of the steps to finish setting up the Win32(WCE Emulator) option.

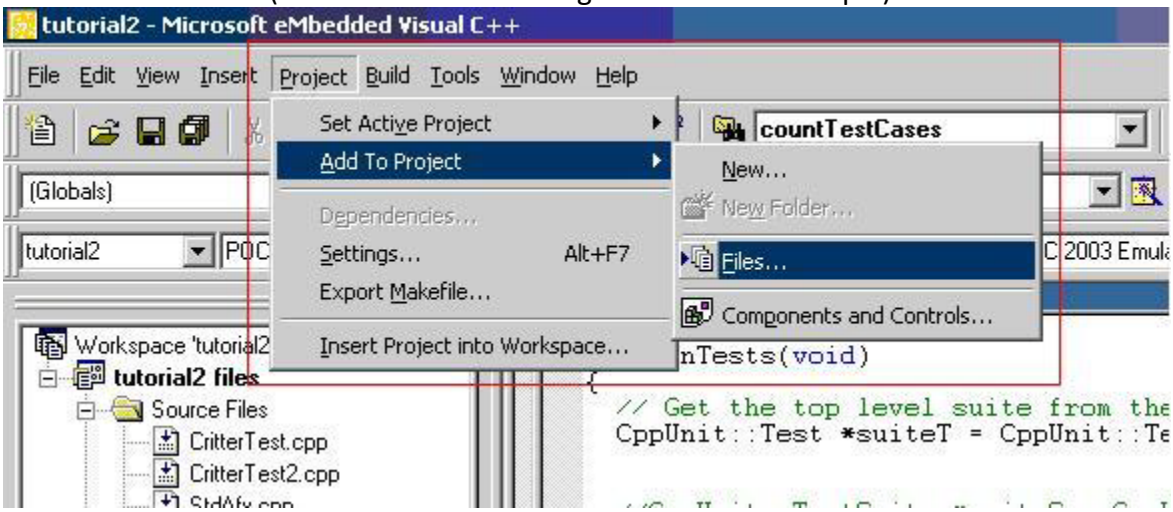
2 Boilerpp Project Setup

The Boilerpp Project will work without any modification. The following steps are to add code to be tested.

Note: To run the Boilerpp project on the Pocket PC 2003 device change the compile option Win32(WCE ARMV4) option. Otherwise this program will run on the Pocket PC 2003 emulator (Win32(WCE Emulator) option).

*******add picture of this option change**********

1. Create/Add the source files being tested, if any. Follow Project->Add to Project->Files... and browse for files to add. (Critter.h is the file being tested in the example)



2. Create "...test.cpp" and "...test.h" to set up the tests to run the source files. Again follow the example test program. (CritterTest.h and CritterTest.cpp are the testing files for Critter.h)
**********add picture here with the GUI files present *********
3. #include "StdAfx.h" in all of the new (or added) .cpp files.

CritterTest2.cpp

```
#include "StdAfx.h"
#include <cppunit/extensions/HelperMacros.h>
#include "Critter2.h"
#include "CritterTest2.h"

CPPUNIT_TEST_SUITE_REGISTRATION( CritterTest2 );

void CritterTest2::TestMemorySmoke2(void) {
  Memory2 m;
  CPPUNIT_ASSERT(  m.AssociatedP(0) );
  CPPUNIT_ASSERT(! m.AssociatedP(42) );
  CPPUNIT_ASSERT(  m.AssociatedP(3) ); /* this should fail */
}
```

4. Repeat steps 1-3 to add more files to test.
5. Follow example test documentation, and CPPUnit documentation, to add more testcases.
http://cppunit.sourceforge.net/doc/1.0/cppunit_cookbook.html
This link is the best source for test setup information.

3 Running the Test Project

To compile the project click the Rebuild Button.

If compile is successful the Device or Emulator will have the Boilerpp Test Project loaded on to it.

3.1 Location of the Project in the Device/Emulator

1. Initial Emulator view

2. Select Programs from the Windows Menu Dropdown

3. Select File Explorer

4. Select My Device from the My Documents dropdown

5. Select "Prototype" to run the Testing files **********edit, fix with final project name

3.2 Testing Options

This section outlines what all of the different options for selecting tests.

3.3 Test Selection

Clicking All Tests will check all of the suites and testcases.

Clicking a suite will run all of the tests in that suite. Also, unless all suites are checked All Tests will have a partial check mark.

Clicking a testcase will run that test case. Also, unless all testcases in a suite are checked the suite, as well as All Tests, will have a partial checkmark.

If you click a checked checkbox it will uncheck itself and all tests/suites beneath it (if there are any).
Before:


Pocket PC 2003
Emulator Help
Pocket PC
Prototype
8:30
ok
All Tests
CritterTest
TestMemorySmoke
TestMemorySmoke2
CritterTest2
CritterTest3
CritterTest4
CritterTest5
CritterTest6
CritterTest7
CritterTest8
CritterTest9
Start Test
Selection Results

After:

Last there is a Clear Selection option that will clear all selected tests.

3.4 Load/Save

3.4.1 Save

Once tests have been selected the tests can be saved to a file.

1. After "Save" has been selected the Save As window will popup.

2. Once "OK" has been clicked the file will be located at the location chosen, and will be available for loading.

3.4.2 Load

A test can be loaded and those tests will be checked if they exist. In addition any previously loaded or selected tests will remain selected.

1. Once "Load" has been selected the Load window will popup.

2. After the file has been selected the application returns to the Test Selection screen and shows how many tests were successfully loaded and how many tests could not be loaded. If they couldn't be loaded it means they no longer exist in the current test set, which could be that they have been deleted or renamed.

4 Viewing Output

4.1 Immediate Result Viewing

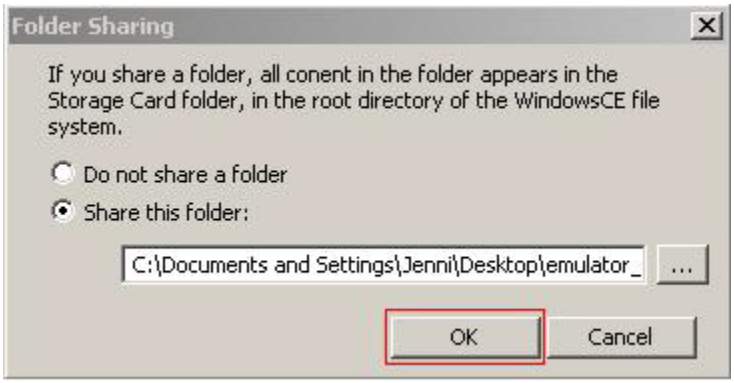
After running the test file a window will appear with which tests passed, failed, or were not ran.
******Show picture with a test passed, failed, and not ran*******
The green box shows a test that passed. The red box shows a test that failed. The black box shows a test that wasn't ran.

4.2 Viewing XML Output

To view the XML Output first it needs to be transferred to a computer.

1. On the emulator window: Emulator>Folder Sharing...

2. Choose the folder you want to share & click OK



3. The folder is now in My Device as Storage Card

4. Open the XML document and select Tools->Options

5. Select Storage Card from the Save to drop down:

Now the XML output is available in the folder selected in step 2.

******add any steps to finalize being able to view the XML doc****

Once transfer is complete the XML output can be viewed using our XML style sheet.

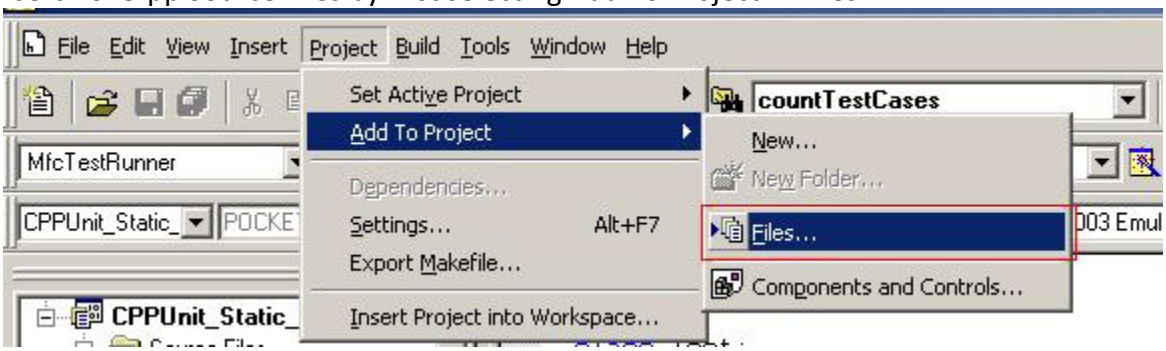
******finished xml view pic*******

5 Creating a new Boilerpp Project

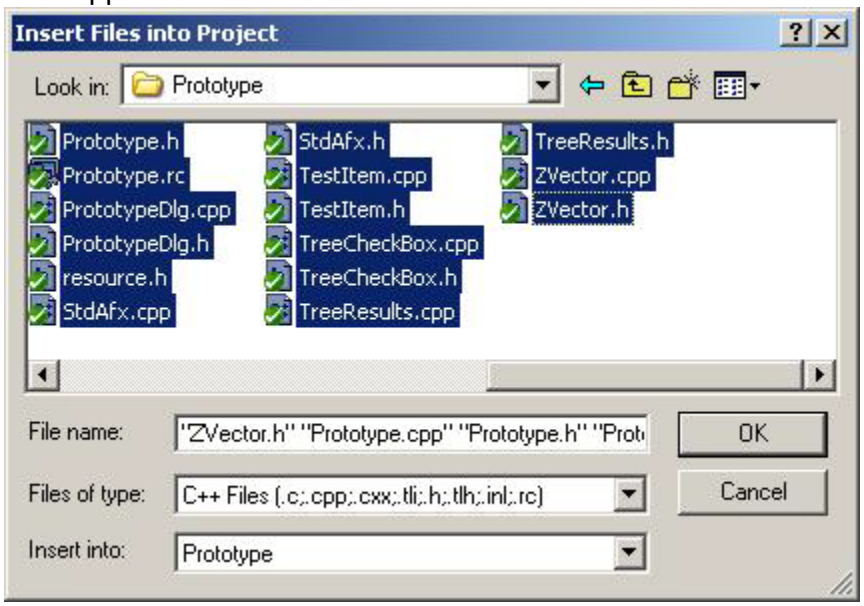
Following these steps will create a new Boilerpp Project.

Note: If the compile option is changed from Win32(WCE Emulator) option (to run on the emulator) to the Win32(WCE ARMV4) option (to run on the device) then steps 3 and 4 will need to be repeated.

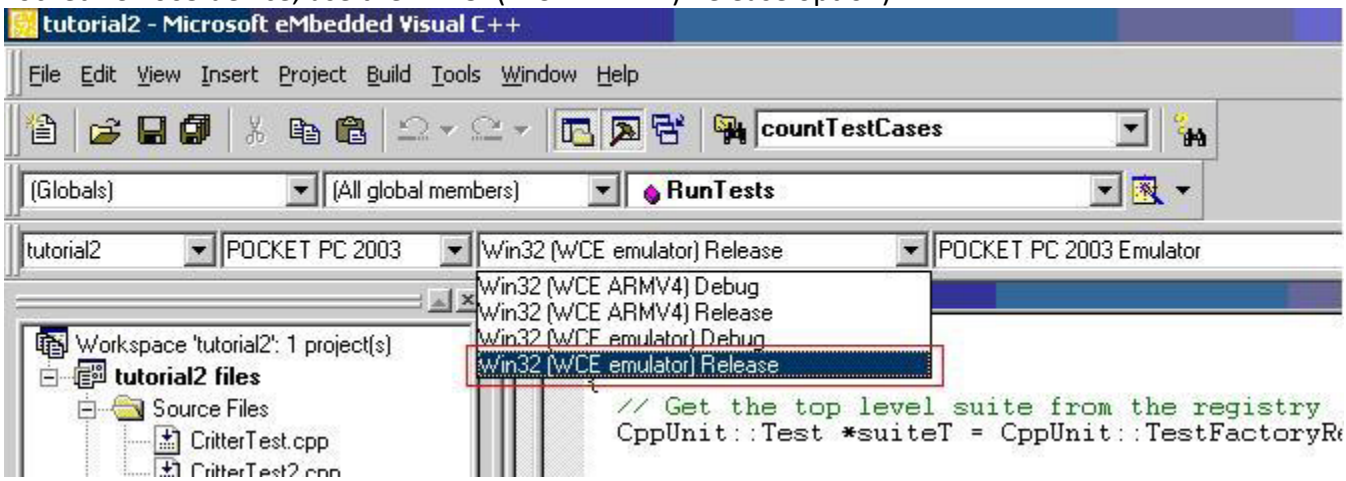
1. Insert Boilerpp Source Files by first selecting Add To Project -> Files…



Then select the Boilerpp Source Files

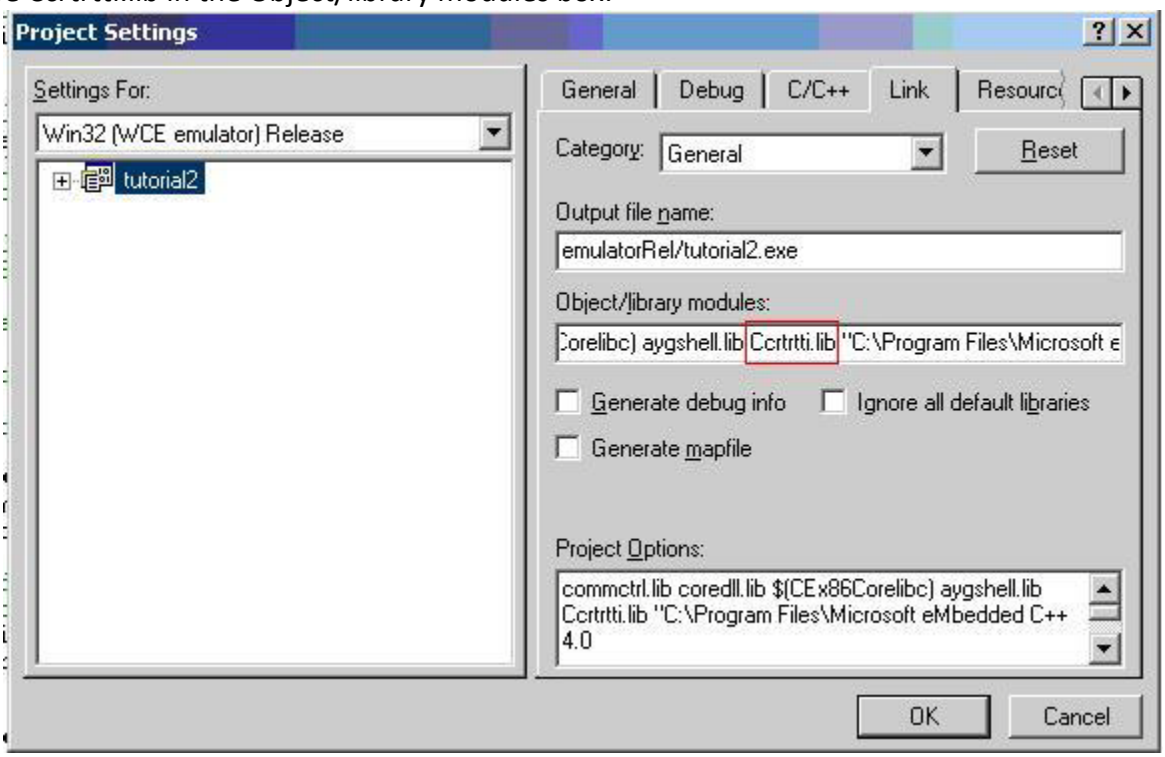


2. Change compile option to Win32(WCE emulator) Release (if compiling for operation on a Pocket PC 2003 device, use the Win32(WCE ARMV4) Release option):

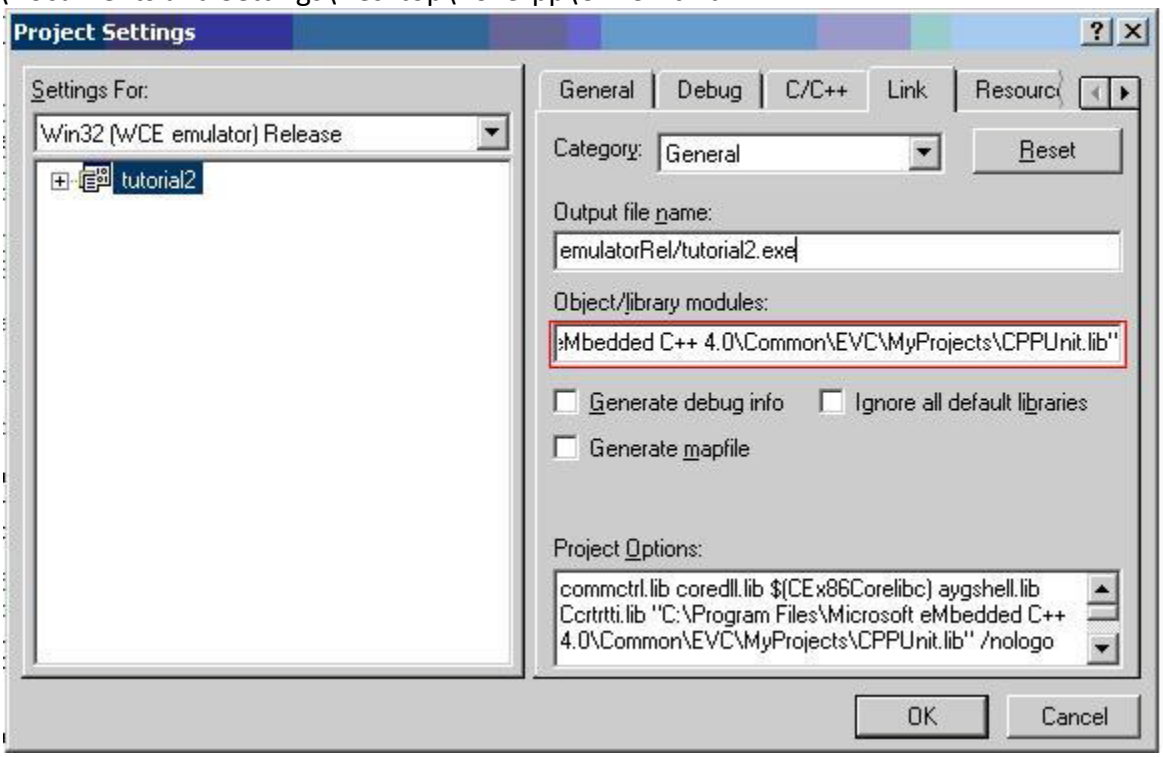


3. Go to Project->Settings->Link

Type Ccrtrtti.lib in the Object/library modules box:

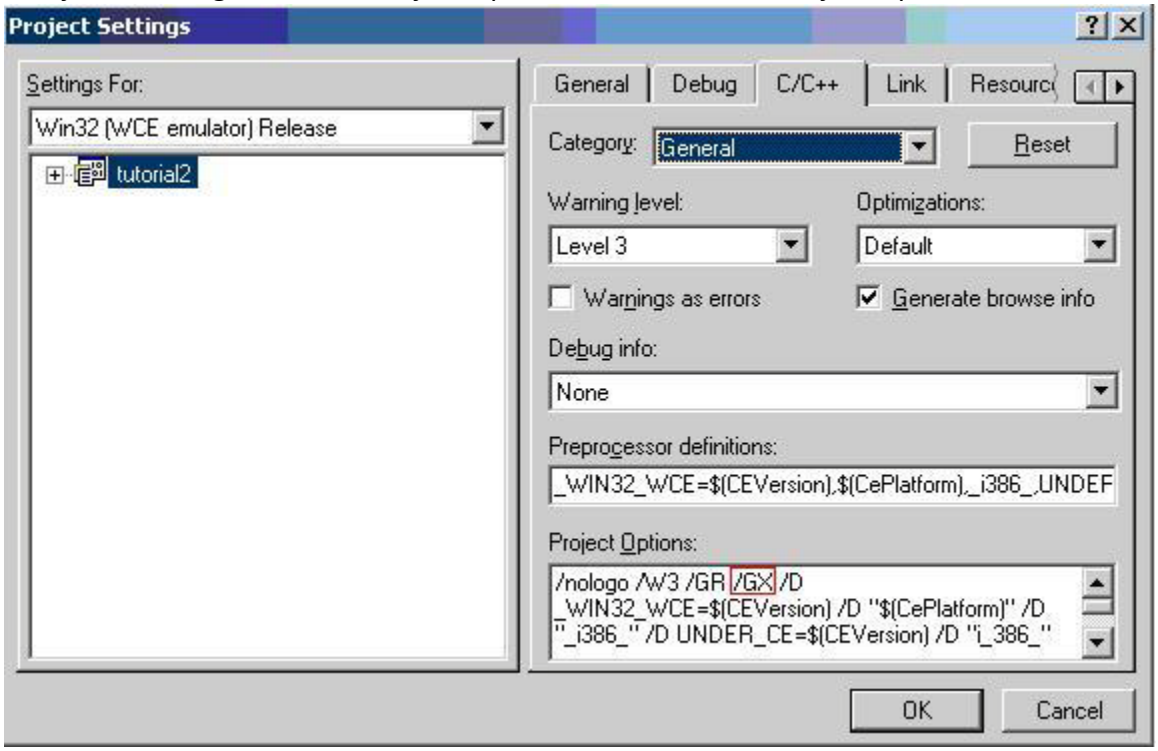


Type the full path name of CPPUnit.lib in quotes in the Object/library modules box. For example "C:\Documents and Settings\Desktop\Boilerpp\CPPUnit.lib":



Note: The libraries for running on the emulator and running on the device are different. The emulator library is compiled with the Win32(WCE Emulator) option selected; where the device is compiled with the Win32(WCE ARMV4) option selected.

4. In Project->Settings->C/C++->Project Options add /GX to the Project Options.

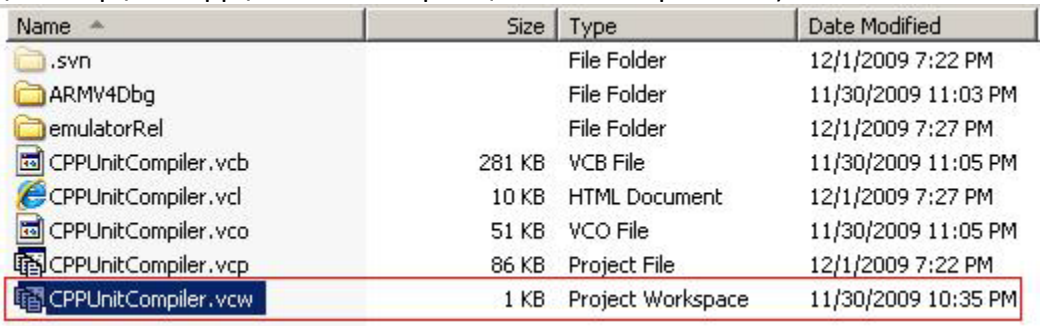


From here follow the Boilerpp Project Setup

6 FUDG CPPUnit Library

6.1 Compiling a FUDG CPPUnit Library

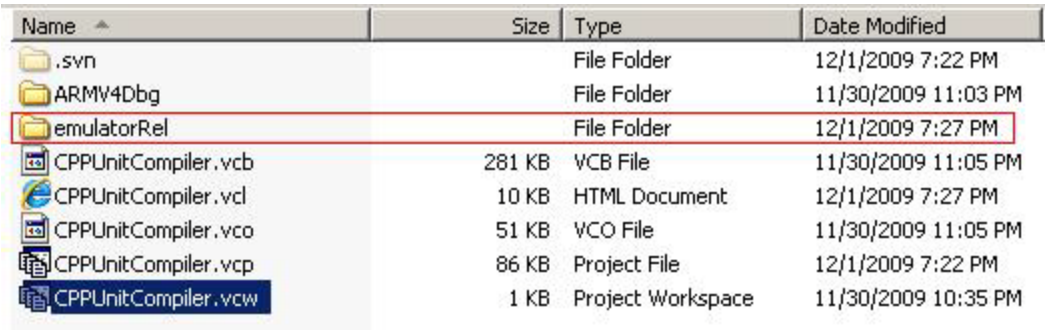
1. Open the FUDGCPPUnit project file (For example: C:\Documents and Settings\Desktop\Boilerpp\ CPPUnitCompiler\CPPUnitCompiler.vcw)



Name	Size	Type	Date Modified
.svn		File Folder	12/1/2009 7:22 PM
ARMV4Dbg		File Folder	11/30/2009 11:03 PM
emulatorRel		File Folder	12/1/2009 7:27 PM
CPPUnitCompiler.vcb	281 KB	VCB File	11/30/2009 11:05 PM
CPPUnitCompiler.vcl	10 KB	HTML Document	12/1/2009 7:27 PM
CPPUnitCompiler.vco	51 KB	VCO File	11/30/2009 11:05 PM
CPPUnitCompiler.vcp	86 KB	Project File	12/1/2009 7:22 PM
CPPUnitCompiler.vcw	1 KB	Project Workspace	11/30/2009 10:35 PM

2. Click the rebuild button

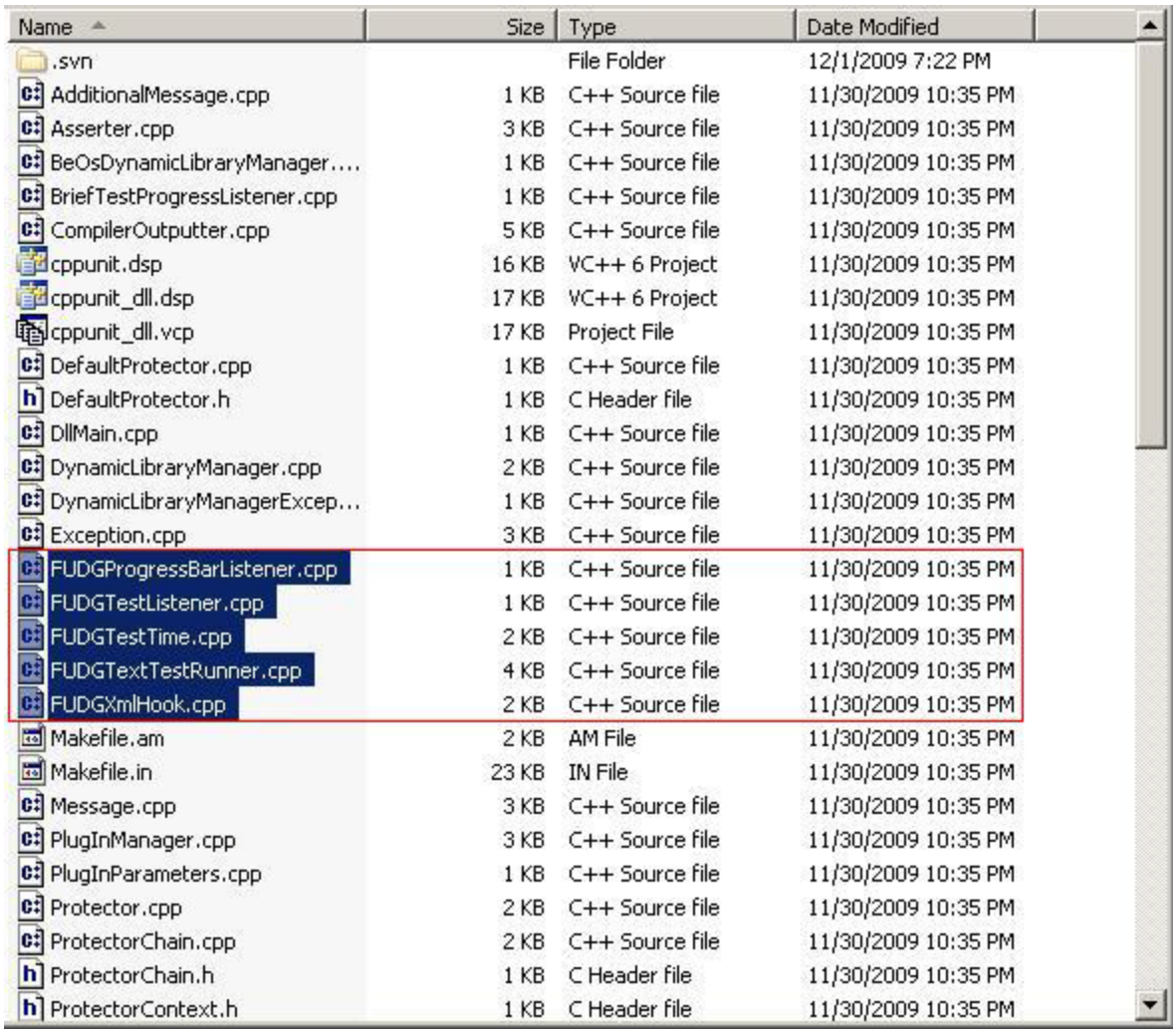
3. Your new library file will be in the emulatorRel file in the FUDGCPPUnit project file (if compiled with ARMV4 option, it will be in the ARMV4Rel file)



Name	Size	Type	Date Modified
.svn		File Folder	12/1/2009 7:22 PM
ARMV4Dbg		File Folder	11/30/2009 11:03 PM
emulatorRel		File Folder	12/1/2009 7:27 PM
CPPUnitCompiler.vcb	281 KB	VCB File	11/30/2009 11:05 PM
CPPUnitCompiler.vcl	10 KB	HTML Document	12/1/2009 7:27 PM
CPPUnitCompiler.vco	51 KB	VCO File	11/30/2009 11:05 PM
CPPUnitCompiler.vcp	86 KB	Project File	12/1/2009 7:22 PM
CPPUnitCompiler.vcw	1 KB	Project Workspace	11/30/2009 10:35 PM

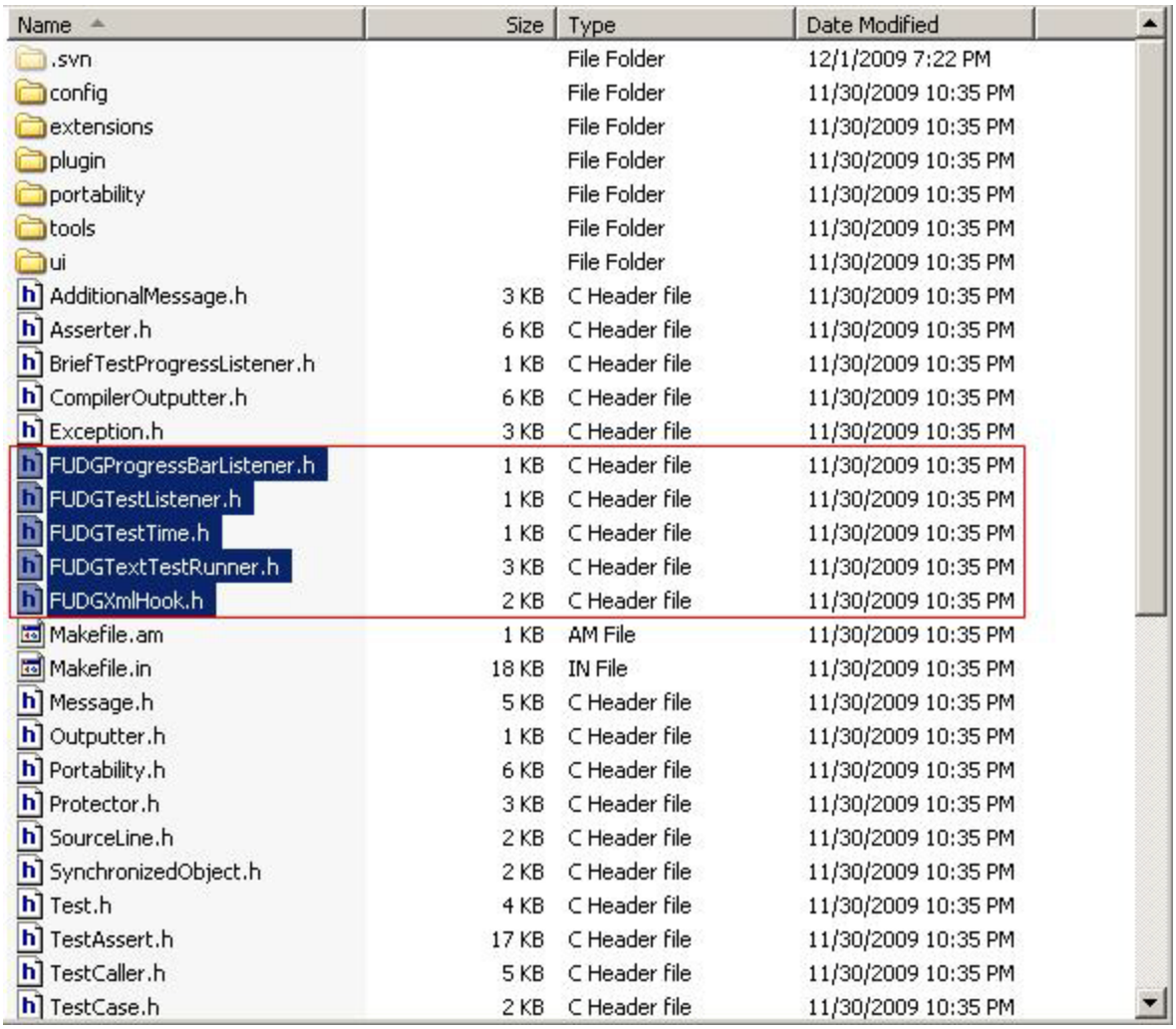
6.2 Creating a new FUDG CPPUnit Library Project

This is assuming that the CPPUnit framework being used does not yet include the FUDG CPPUnit additions.
1. Add the following files in the CPPUnit source and include with the FUDGCPPUnit files:
FUDGProgressBarListener (.cpp and .h)
FUDGTestListener (.cpp and .h)
FUDGTestTime (.cpp and .h)
FUDGTextTestRunner (.cpp and .h)
FUDGXmlHook (.cpp and .h)
Source:



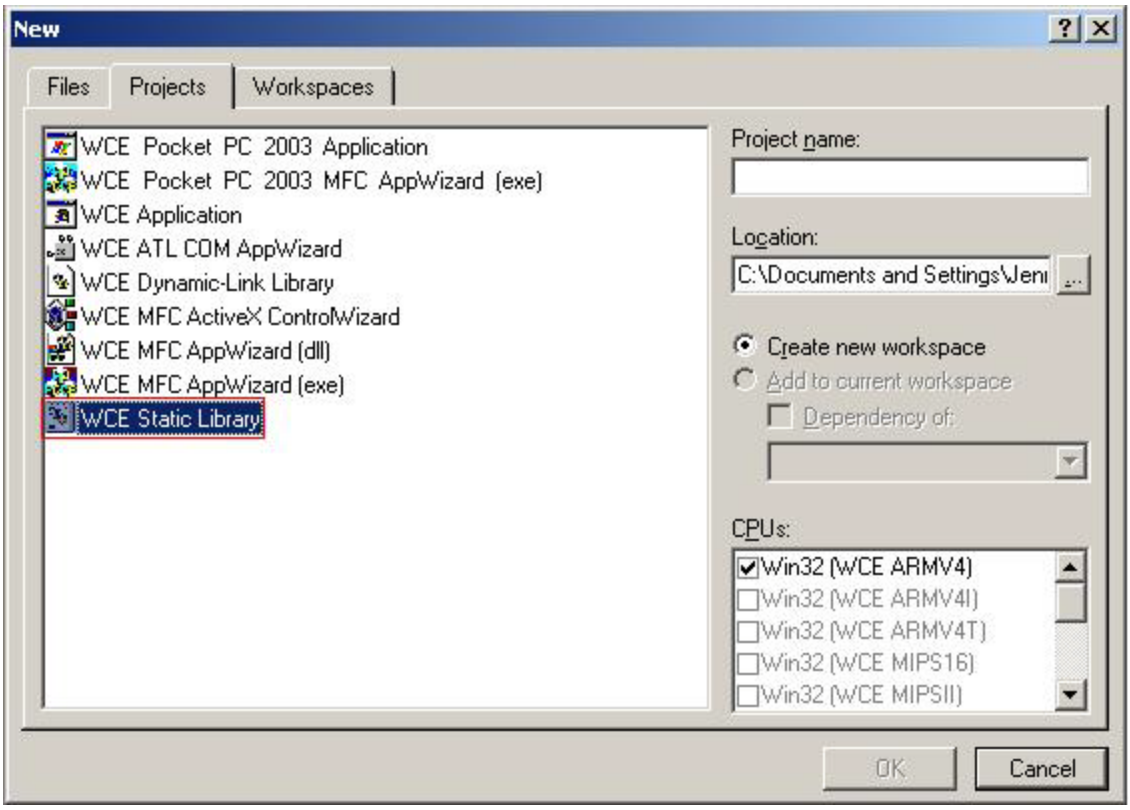
Name	Size	Type	Date Modified
.svn		File Folder	12/1/2009 7:22 PM
AdditionalMessage.cpp	1 KB	C++ Source file	11/30/2009 10:35 PM
Asserter.cpp	3 KB	C++ Source file	11/30/2009 10:35 PM
BeOsDynamicLibraryManager....	1 KB	C++ Source file	11/30/2009 10:35 PM
BriefTestProgressListener.cpp	1 KB	C++ Source file	11/30/2009 10:35 PM
CompilerOutputter.cpp	5 KB	C++ Source file	11/30/2009 10:35 PM
cppunit.dsp	16 KB	VC++ 6 Project	11/30/2009 10:35 PM
cppunit_dll.dsp	17 KB	VC++ 6 Project	11/30/2009 10:35 PM
cppunit_dll.vcp	17 KB	Project File	11/30/2009 10:35 PM
DefaultProtector.cpp	1 KB	C++ Source file	11/30/2009 10:35 PM
DefaultProtector.h	1 KB	C Header file	11/30/2009 10:35 PM
DllMain.cpp	1 KB	C++ Source file	11/30/2009 10:35 PM
DynamicLibraryManager.cpp	2 KB	C++ Source file	11/30/2009 10:35 PM
DynamicLibraryManagerExcep...	1 KB	C++ Source file	11/30/2009 10:35 PM
Exception.cpp	3 KB	C++ Source file	11/30/2009 10:35 PM
FUDGProgressBarListener.cpp	1 KB	C++ Source file	11/30/2009 10:35 PM
FUDGTestListener.cpp	1 KB	C++ Source file	11/30/2009 10:35 PM
FUDGTestTime.cpp	2 KB	C++ Source file	11/30/2009 10:35 PM
FUDGTextTestRunner.cpp	4 KB	C++ Source file	11/30/2009 10:35 PM
FUDGXmlHook.cpp	2 KB	C++ Source file	11/30/2009 10:35 PM
Makefile.am	2 KB	AM File	11/30/2009 10:35 PM
Makefile.in	23 KB	IN File	11/30/2009 10:35 PM
Message.cpp	3 KB	C++ Source file	11/30/2009 10:35 PM
PlugInManager.cpp	3 KB	C++ Source file	11/30/2009 10:35 PM
PlugInParameters.cpp	1 KB	C++ Source file	11/30/2009 10:35 PM
Protector.cpp	2 KB	C++ Source file	11/30/2009 10:35 PM
ProtectorChain.cpp	2 KB	C++ Source file	11/30/2009 10:35 PM
ProtectorChain.h	1 KB	C Header file	11/30/2009 10:35 PM
ProtectorContext.h	1 KB	C Header file	11/30/2009 10:35 PM

Include:



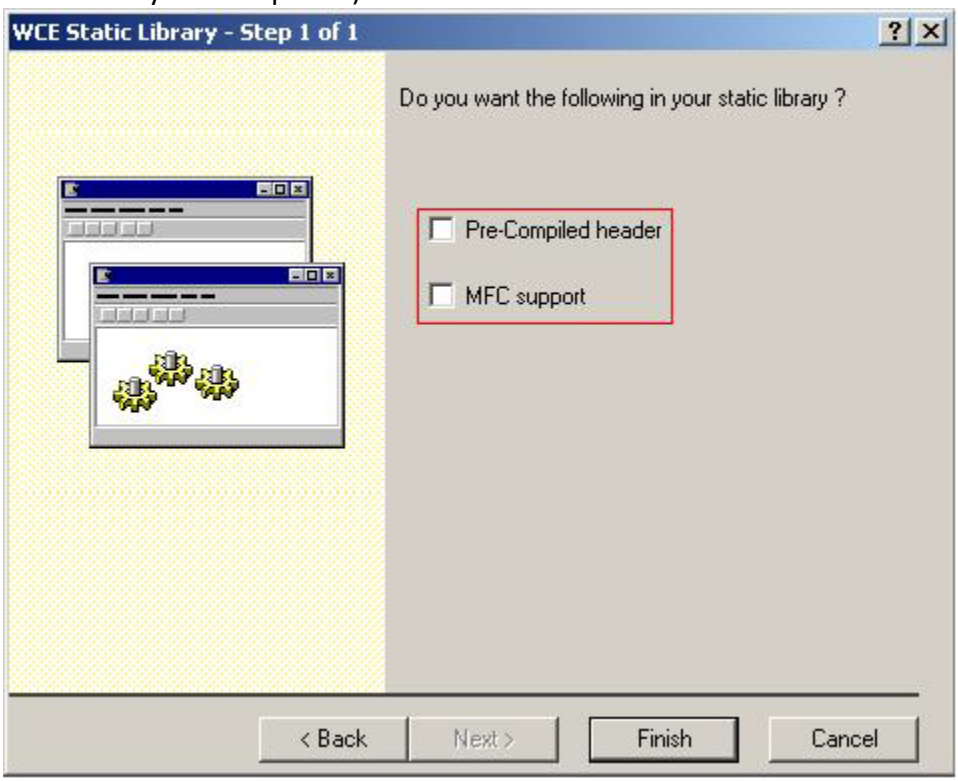
Name	Size	Type	Date Modified
.svn		File Folder	12/1/2009 7:22 PM
config		File Folder	11/30/2009 10:35 PM
extensions		File Folder	11/30/2009 10:35 PM
plugin		File Folder	11/30/2009 10:35 PM
portability		File Folder	11/30/2009 10:35 PM
tools		File Folder	11/30/2009 10:35 PM
ui		File Folder	11/30/2009 10:35 PM
AdditionalMessage.h	3 KB	C Header file	11/30/2009 10:35 PM
Asserter.h	6 KB	C Header file	11/30/2009 10:35 PM
BriefTestProgressListener.h	1 KB	C Header file	11/30/2009 10:35 PM
CompilerOutputter.h	6 KB	C Header file	11/30/2009 10:35 PM
Exception.h	3 KB	C Header file	11/30/2009 10:35 PM
FUDGProgressBarListener.h	1 KB	C Header file	11/30/2009 10:35 PM
FUDGTestListener.h	1 KB	C Header file	11/30/2009 10:35 PM
FUDGTestTime.h	1 KB	C Header file	11/30/2009 10:35 PM
FUDGTextTestRunner.h	3 KB	C Header file	11/30/2009 10:35 PM
FUDGXmlHook.h	2 KB	C Header file	11/30/2009 10:35 PM
Makefile.am	1 KB	AM File	11/30/2009 10:35 PM
Makefile.in	18 KB	IN File	11/30/2009 10:35 PM
Message.h	5 KB	C Header file	11/30/2009 10:35 PM
Outputter.h	1 KB	C Header file	11/30/2009 10:35 PM
Portability.h	6 KB	C Header file	11/30/2009 10:35 PM
Protector.h	3 KB	C Header file	11/30/2009 10:35 PM
SourceLine.h	2 KB	C Header file	11/30/2009 10:35 PM
SynchronizedObject.h	2 KB	C Header file	11/30/2009 10:35 PM
Test.h	4 KB	C Header file	11/30/2009 10:35 PM
TestAssert.h	17 KB	C Header file	11/30/2009 10:35 PM
TestCaller.h	5 KB	C Header file	11/30/2009 10:35 PM
TestCase.h	2 KB	C Header file	11/30/2009 10:35 PM

2. Start new WCE Static Library with a project name like "CPPUnit". On page 2 don't check any options.
Page 1:



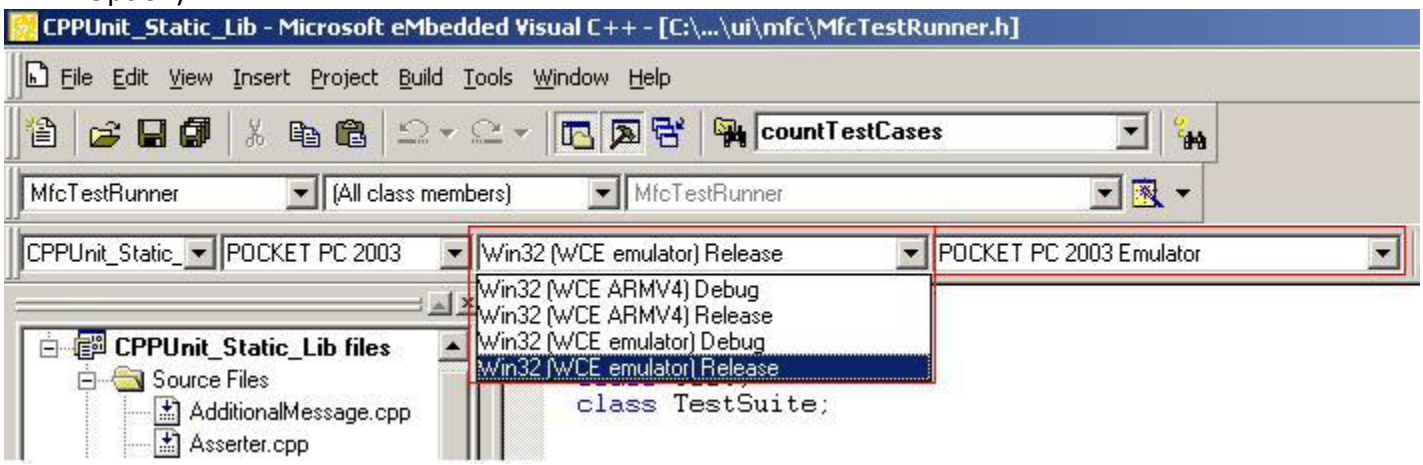
Click "OK"

Page 2 (don't check any of the options):

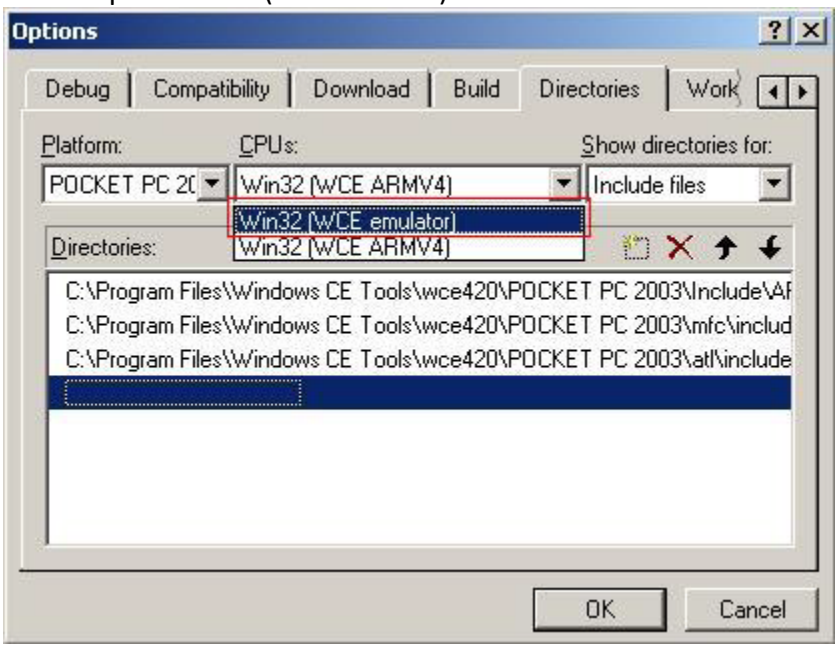


Click "Finish"

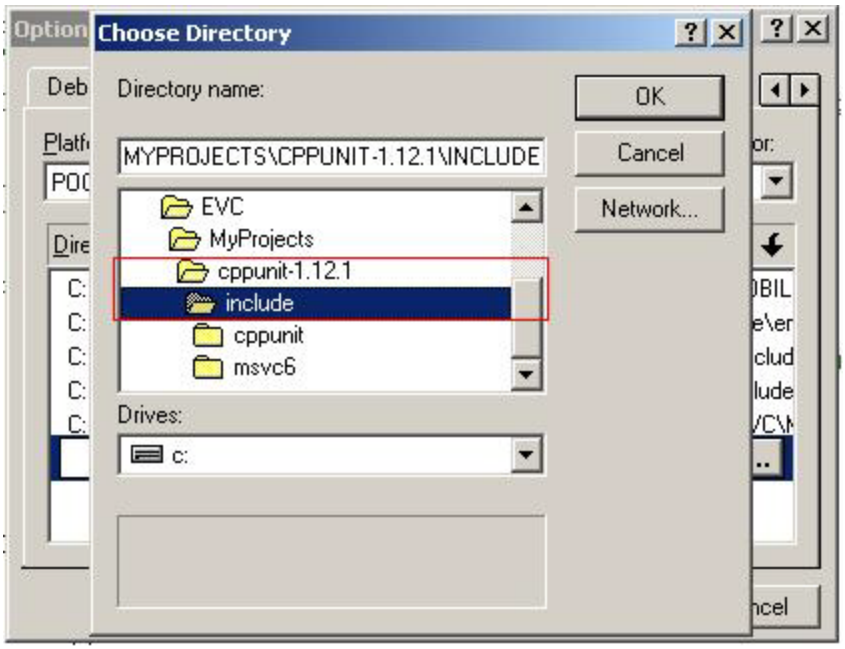
3. Change compile option to "Win32 (WCE emulator) Release" and make sure Pocket PC 2003 Emulator is selected (If using a Pocket PC 2003 device, use the Win32 (WCE ARMV4) Release Option).



4. Tools->Options->Directories->New (looks like a dotted line box with a spark, first on the left of the icons) (Do this step for Win32(WCE ARMV4) if this is for a Pocket PC 2003 option).

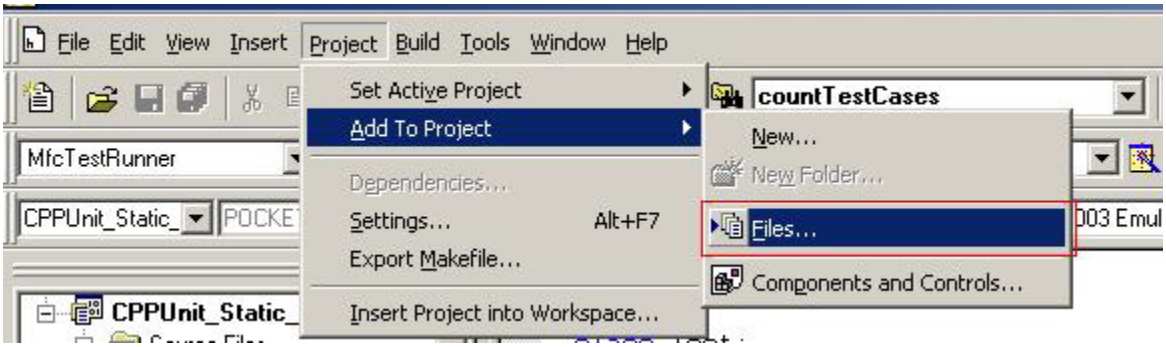


Browse for CPPUnit include folder.

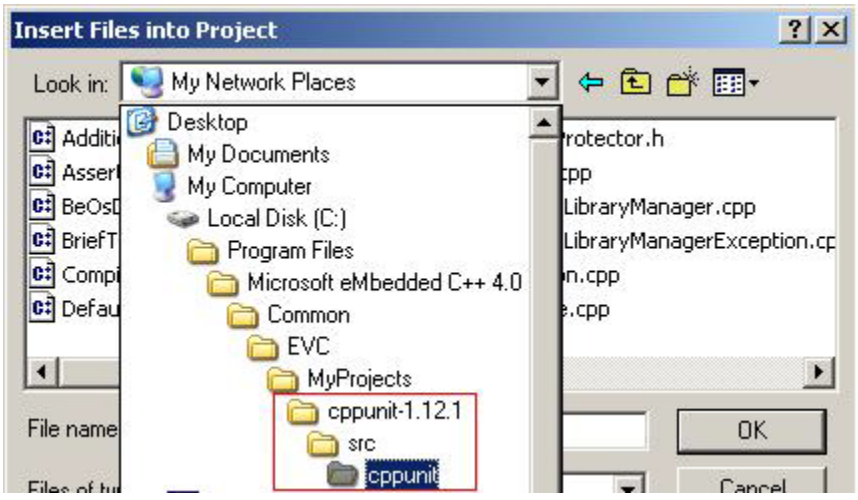


5. Add files by following Project->Add to Project->Files... Add all of the files in cppunit-1.12.1/src/cppunit and cppunit-1.12.1/include/cppunit except for Win32DynamicLibraryManager.cpp.

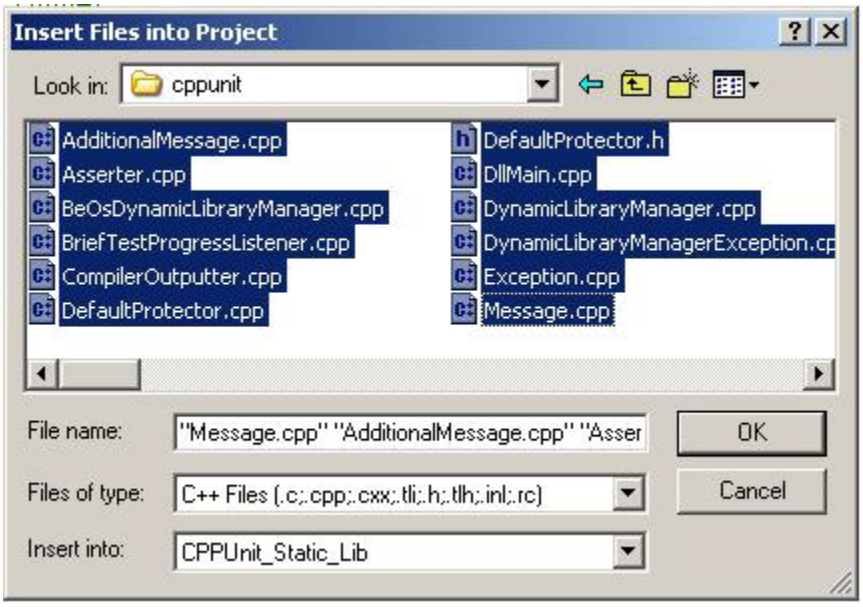
Menu Selection:



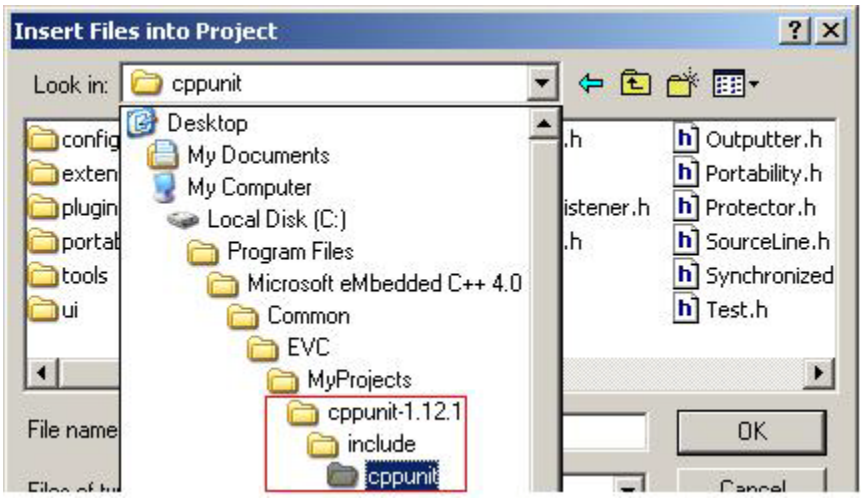
Source File location:



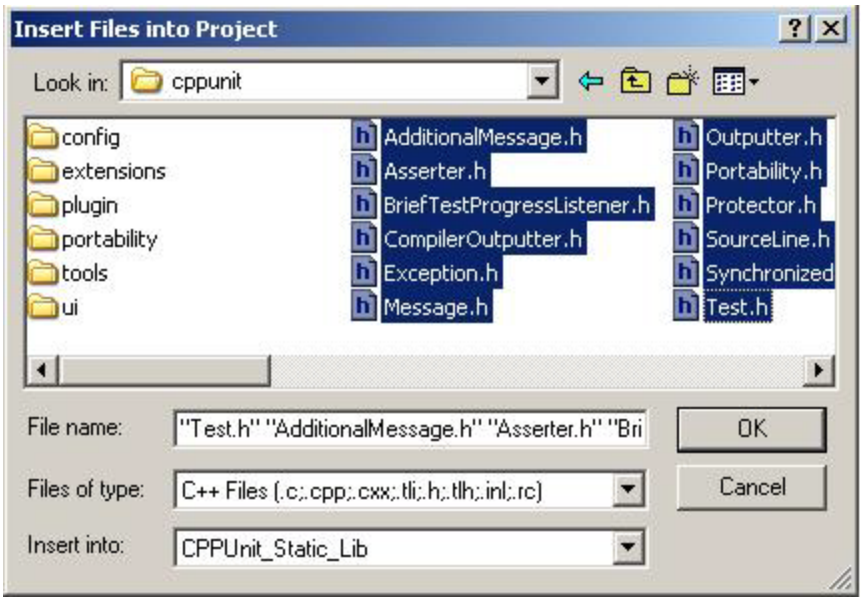
- Select all of these files except for Win32DynamicLibraryManager.cpp:



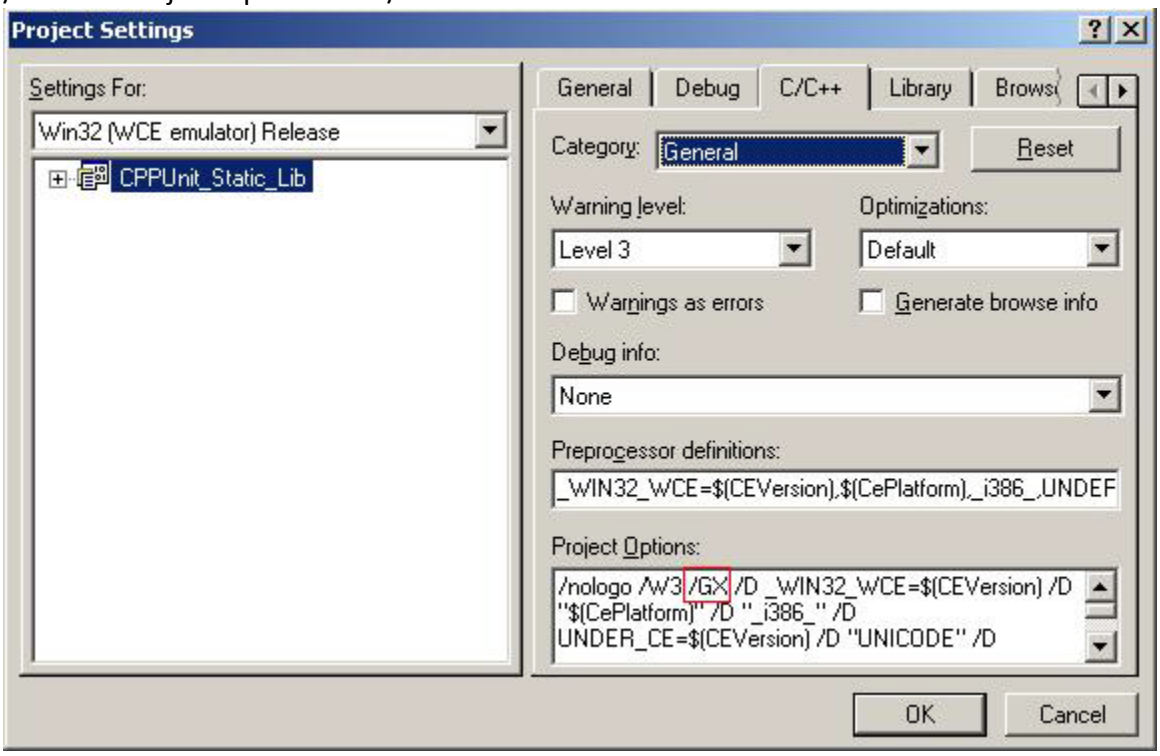
Included File location:



- Select all of these these files (add all files in folders as well. You will have to go into each folder and select the files):



6. C/C++ tab Project Options add /GX

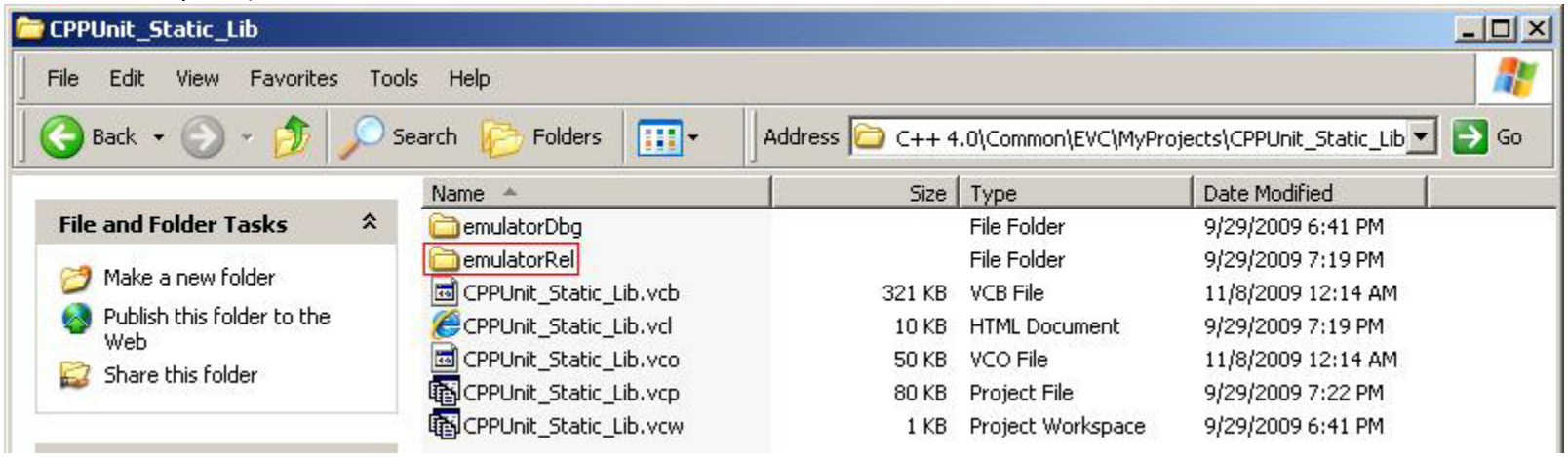


7. In Portability.h remove all but evc4 config file from the platform specific config section. This is the only line you need:

```
#ifndef CPPUNIT_PORTABILITY_H
#define CPPUNIT_PORTABILITY_H

#if defined(_WIN32) && !defined(WIN32)
# define WIN32 1
#endif

/* include platform specific config */

#    include <cppunit/config/config-evc4.h>

// Version number of package
#ifndef CPPUNIT_VERSION
#define CPPUNIT_VERSION  "1.12.0"
#endif

#include <cppunit/config/CppUnitApi.h>     // define CPPUNIT_API & CPPUNIT_NEED_DLL_DECL
#include <cppunit/config/SelectDllLoader.h>
```

To compile the project click the Rebuild Button.

9. After successfully compiling the Library file is now available in the emulatorRel file that is created in your project file (this will be in ARMV4Rel if compiled with the Win32(WCE ARMV4) option).



Appendix B: Glossary

Appendix C: Test Validation Protocol and Report

(to be edited in future drafts)

Parameters
Parameters selected for validation were chosen for their ability to cause errors or to reveal errors. The parameters selected were: testcase tree expansion, testcases selected, testcases saved, testcases loaded, clear selection, immediate results viewing, save file location, load file location, output file location.

Constraints
Constraints were selected based on Boundary Value Analysis. For each parameter these are the constraints for test development:

Testcase Tree Expansion
1. Condensed
2. Partially Expanded
3. Fully Expanded

Testcases Loaded
1. None Selected
2. One Selected
3. Many Selected
4. All Selected

Save File Location
1. Leave Location Alone
2. Move One Directory
3. Move Many Directories

Testcases Selected
1. None Selected
2. One Selected
3. Many Selected
4. All Selected

Clear Selection
1. None Selected
2. Some Selected
3. All Selected

Load File Location
1. Leave Location Alone
2. Move One Directory
3. Move Many Directories

Output File Location
1. Leave Location Alone
2. Move One Directory
3. Move Many Directories

Immediate Results Viewing
1. None Failed
2. Some Failed
3. All Failed
4. None Passed
5. Some Passed
6. All Passed
7. No Testcases Ran
8. Some Testcases Ran
9. All Testcases Ran

Testcases Saved
1. None Selected
2. One Selected
3. Many Selected
4. All Selected

Constraint Test Set

The following set of constraints was selected initially to cover all constraints for each parameter. Next, after all constraints were covered once, constraint selections were made to attempt to cover as many constraint pairs as possible. Pairings were also selected to maximize the effectiveness of the pairing (for example, pairing Testcases Saved with Testcases Loaded). Finally, invalid pairs were eliminated and replaced with valid pairs to cover more pairs.

These parameters are numbered, as follows:

1. Testcase Tree Expansion
2. Testcases Selected
3. Testcases Saved
4. Testcases Loaded
5. Clear Selection
6. Immediate Results Viewing
7. Save File Location
8. Load File Location
9. Output File Location

The constraints for each parameter follow the constraint lists that were in the Constraints section.

Constraint Set:

	Parameters								
Constraint Set	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	7	1	1	1
2	2	2	2	2	2	9	2	2	2
3	3	3	3	3	2	8	3	3	3
4	1	4	4	4	3	2	1	2	3
5	2	1	3	4	3	1	2	1	3
6	3	2	1	4	2	3	3	2	1
7	2	3	4	1	2	4	1	3	2
8	3	4	2	1	3	5	2	3	1
9	2	4	1	3	2	6	3	1	2

The constraints for all of the parameters except for Immediate Results Viewing (parameter 6) are covered by constraint set four. Constraint sets five through nine are used to fulfill the rest of parameter six's constraints and then to cover more constraint pairs in the other parameters.

Test Setup

Test Results

Appendix D: Administration Information

This section enumerates tasks performed and denotes which team members were responsible for each.
(will be edited in future drafts)

1. *Presentations*

		Lauren	Nate	Mike	Jenni	Brad
4-Sep	create	x		x		x
	present	x		x		x
18-Sep	create		x	x		x
	present		x			x
2-Oct	create			x	x	
	present			x	x	
16-Oct	create	x				
	present	x				
30-Oct	create	x			x	x
	present		x			x
13-Nov	create	x				
	present			x	x	
4-Dec	create					x
	present				x	x
11-Dec	create					
	present					

2. *Documentation*

		Lauren	Nate	Mike	Jenni	Brad
Requirements/Interim 1						
	Content	x	x	x	x	x
	Assembly		x		x	
Design Document/Interim 2						
	Content					
	Assembly					
Interim 2 version 2		x				
	Figures	x				x
Project Report Outline		x	x	x	x	x
Final Report Rough Draft	Content	x	x	x	x	
	Assembly	x				
	Figures	x				
Final Report Draft 2		x				
User Manual					x	
Validation Protocol & Report					x	
Final Report Draft 3		x				

3. *Source*

	Lauren	Nate	Mike	Jenni	Brad
CPPUnit				x	
Timing					x
Checkbox functionality		x			
Merge		x			x
Load/Save			x		
XML/XSL	x				

4. *Research*

		Lauren	Nate	Mike	Jenni	Brad
Frameworks						
	CPPUnit		x			x
	Google Test			x	x	
	jUtAsserter	x		x		
	WCEUnit	x	x			
eVC4/Pocket PC resources		x	x	x	x	x

5. *Administrative / Meta / Miscellaneous*

	Lauren	Nate	Mike	Jenni	Brad
Wiki setup	x				
Cyberonics contact	x				
Facilities interface	x				
Meeting minutes	x	x		x	
Assembla setup		x			