Master Test Plan

for Software Testing Course OHJ-3060/3066,
periods I-II year 2008-2009
Tampere University of Technolgy
Version 3.1.1

Antti Kervinen
ask@cs.tut.fi

18th August 2008

Version	Changes
3.1→3.1.1	• Firefox 3.1 alpha 1 candidate will be tested.
3.0→3.1	• The requirement for testing IPv6 clarified. • RFC 3986 replaces RFCs 2396 and 2732.
2.01→3.0	• Major changes in unit test design. • Firefox 3.0 alpha 7 “minefield” will be tested.
2.0→2.01	• Fixed the explanation of setUp and tearDown.
1.1→2.0	• Tested software changed to Firefox version 2.0b1. • Test environment changed from Solaris to Linux. • Added sample output of the parser.
1.0→1.1	• Lots of small changes for Autumn 2005 course.
0.9→1.0	• An example of relative url system test case added. • The requirements for reporting working hours was clarified. • New fields (created & modified) were introduced in test cases. • Dependencies between test cases are now required to be mentioned in test plans.
0.8→0.9	• Task list for system testing. (Section 8.2) • Outline for system test report. (Section 7.2) • Instructions for setting up system test environment. (Section 9)
0.7→0.8	• Requirements for the submission of the unit test report changed. (Section 15.2) • Outline for test reports added. (Section 14.2)

Contents

References

[CPPUNIT] Feathers M., Lepilleur B.: *CppUnit Cookbook* http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html

[HTML] Korpela, J.: *Documents about WWW written or recommended by Jukka Korpela*. http://www.cs.tut.fi/~jkorpela/www.html

[IEEE829] *IEEE Standard for Software Test Documentation.* IEEE Std 829-1998. September, 1998. Available from TUT addresses in http://www.ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=16010

[Firefox] Firefox web pages. http://www.mozilla.com.

[Mye04] Myers G. J., Sandler C., Badgett T., Thomas T. M.: *The Art of Software Testing*, 2nd edition. John Wiley & Sons, 2004.

[RFC3986] Uniform Resource Identifiers (URI): Generic Syntax. http://www.ietf.org/rfc/rfc3986.txt

Glossary

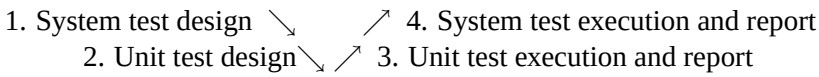
Firefox	An open source graphical web browser.
RFC	*Request for comments*, a series of Internet informational documents and standards
URL	*Unified Resource Locator*, a string that identifies a resource by its location

1 Introduction

This document describes the course project for software testing courses (both English and Finnish) in Tampere University of Technology, periods I-II year 2008-2009.

1.1 Overview

In this project the URL parser of Firefox [Firefox] web browser will be tested. The tests are designed and executed according to the "V" model:



In the system test, the parsers will be tested by running a slightly modified Firefox, giving it some URLs to parse, and by examining the output of the parser. In the unit test, a nsStdURLParser class is tested separately by calling its methods from test drivers written by students. Running unit tests requires implementing stubs, too.

The first two tasks are the most important. Once you have good test plans for both unit and system tests, the tests in the last two tasks will be easy to implement and run. With a pitiful test design, however, testing will be very laborious and in the worst case you have to redesign the tests during the execution tasks.

1.2 Test environment and tools

1.2.1 Shell: Bash

This document contains some shell commands and scripts. They are written for Bash shell. Do not run them in (t)csh shells. If your default shell in Lintula is tcsh, it is recommended that you change it to Bash, see `http://www.cs.tut.fi/lintula/ohjeet/komentotulkki.shtml` for instructions. If you do not want to change the default shell, remember to start Bash with command `bash` before running the commands and scripts.

If you are not sure about your shell, run "`ps | grep $$`".

1.2.2 Operating system: Linux

All the necessary tools, libraries and scripts are installed in Lintula Linux environment. Do not try to use them on Solaris, they will not work on it. If you are unsure about the operating system of a workstation or a server, run "`uname`" (SunOS equals Solaris).

Lintula Linux workstations are located in TC217. In addition to them, Lintula Linux server `haikara.cs.tut.fi` can be used for the project.

1.2.3 Code coverage: CTC++

CTC++ is a test coverage analyser from Testwell. The tool is used for measuring multicondition coverage in unit testing. CTC++ is installed in Lintula and can be used in both Solaris and Linux operating systems. For more information, run (in Bash):

```
~$ . /share/testwell/bin/testwell_environment
~$ man ctc
```

1.2.4 Unit test framework: CPPUnit

CPPUnit is a C++ port of JUnit unit testing framework. In task 3, tests are implemented and run with CPPUnit.

1.3 Structure of this document

This document contains two parts: the test plan for system testing and the test plan for unit testing. Both parts follow the template for test plans in IEEE standard for software test documentation [IEEE829].

Due to the “V” model, the first part includes the requirements for tasks 1 and 4 , the second part for tasks 2 and 3.

Part I

Test plan for system testing

2 Test items

The URL parser of Firefox 3.1 alpha 1 CE will be tested in the system level. "CE" stands for "Course Edition", meaning a slightly modified version of the original Firefox 3.1 alpha 1. It is quite likely that the slight modifications have introduced some new faults in the parser.

The syntax of URLs is specified in [RFC3986]. The document must be used as a specification in the test case design.

3 Features to be tested

Test how Firefox parses valid URLs and what it does to the invalid ones. The objective is to find valid URLs that are parsed incorrectly (separation to various fields fails) and incorrect URLs that are considered valid. Finding a URL that causes a crash or some other unexpected behaviour is even more desired. You may even get rewarded for that (see `http://www.mozilla.org/security/bug-bounty.html`).

We limit the testing to the URLs in http scheme, that is,

- *absolute* URLs that begin with `http:` (including URLs that contain IPv6 addresses),
  for example "`http://www.cs.tut.fi/~testaus/`"
- *relative* URLs in the cases where the base URL is in http scheme,
  for example "`../../images/foobar.jpg`"

4 Features not to be tested

Others than URLs in http scheme (such as `ftp://ftp.funet.fi/README`, `mailto:testaus@cs.tut.fi` etc.) will not be tested.

Other functionality than the URL parsing will not be tested.

5 Approach

Firefox (in `/share/tmp/testaus/firefox3.1a1`) has been altered to support system testing in this project. Every time a URL in http-scheme is parsed, the

results of the parsing are printed to standard output. That is, either “valid URL” or “invalid URL” is printed. If the URL was considered valid, the URL parser prints the contents of every field in the URL, such as port number, scheme, authority and so on. It should be verified that the separation is correct.

In the system testing phase you should run Firefox so that it parses the URLs in your test cases. All the output, that is, results of the parsing should be recorded.

You may decide how you give the URLs in your test cases to Firefox. There are several ways to do this. One possibility is to write an HTML [HTML] file containing all the URLs to be tested. Here is one example of the file contents.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=iso-8859-1">
    <base href="http://u:p@foo.foobar.fi:8080/dir1/dir2/file1.ext;p?q#r">
    <title>
    </title>
</head>

<body>

<p>Valid relative paths<br>
<a href="*foo.txt">starts with *</a><br>
</p>

<p>Invalid relative paths<br>
</p>

<p>Valid absolute paths<br>
<a href="http://www.cs.tut.fi/~testaus/">ends with directory</a>
</p>

<p>Invalid absolute paths<br>
<a href="http://space  in.host.fi/illegal">space in hostname</a>
</p>

</body>
```

If `testurls.html` contains the URLs, you run the test as follows:

```
/share/tmp/testaus/firefox3.1a1/run.sh \
    file:testurls.html 2>/dev/null | tee output.txt
```

`tee` program reads standard input and copies it to the given files (here `output.txt` and standard output. `2>/dev/null` hides the stuff Firefox writes to standard error.

When Firefox reads the file, all URLs of the links are parsed and thus the results can be found in the `output.txt` file. You can then check the parsing results in the file. The output caused by parsing the above example file is:

```
§ parsing 'http://u:p@foo.foobar.fi:8080/dir1/dir2/file1.ext;p?q#r'
§ result 'valid URL'
```

```
§ port ’8080’
§ scheme ’http’
§ authority ’u:p@foo.foobar.fi:8080’
§ username ’u’
§ password ’p’
§ hostname ’foo.foobar.fi’
§ path ’/dir1/dir2/file1.ext;p?q#r’
§ filepath ’/dir1/dir2/file1.ext’
§ directory ’/dir1/dir2/’
§ basename ’file1’
§ extension ’ext’
§ param ’p’
§ query ’q’
§ ref ’r’

§ parsing ’http://u:p@foo.foobar.fi:8080/dir1/dir2/*foo.txt’
§ result ’valid URL’
§ port ’8080’
§ scheme ’http’
§ authority ’u:p@foo.foobar.fi:8080’
§ username ’u’
§ password ’p’
§ hostname ’foo.foobar.fi’
§ path ’/dir1/dir2/*foo.txt’
§ filepath ’/dir1/dir2/*foo.txt’
§ directory ’/dir1/dir2/’
§ basename ’*foo’
§ extension ’txt’
§ param ’’
§ query ’’
§ ref ’’

§ parsing ’http://www.cs.tut.fi/~testaus/’
§ result ’valid URL’
§ port ’-1’
§ scheme ’http’
§ authority ’www.cs.tut.fi’
§ username ’’
§ password ’’
§ hostname ’www.cs.tut.fi’
§ path ’/~testaus/’
§ filepath ’/~testaus/’
§ directory ’/~testaus/’
§ basename ’’
§ extension ’’
§ param ’’
§ query ’’
§ ref ’’

§ parsing ’http://space  in.host.fi/illegal’
§ result ’invalid URL’
```

This is just one possibility how a test could be run. You are allowed to separate your test cases in many HTML files or feed the URLs one by one in the location text box (the latter works only with absolute URLs). If it makes your life easier, you may write a program that, for example, reads a list of URLs from your system test design document, generates an HTML file that contains the URLs, runs Firefox and checks that the results of parsing are what you required in the document.

After all, all you are required to do is to test how Firefox parses URLs in the system level. No matter how you get there.

6 Item pass/fail criteria

The system test fails if and only if any (valid or invalid) URL causes Firefox to crash or stop responding, or if a valid URL is incorrectly parsed.

7 Test deliverables

7.1 Test design and test case specifications (the first task)

The information required in the test design document is listed below. It is recommended that you use the list as an outline for your document. However, if you think that an item in the list is irrelevant, redundant or otherwise unnecessary, it is better that you ignore it than write something stupid under the heading. But if you ignore an item, an explanation is required!

1. **Cover page**
   - Provide the names and student numbers of the authors.
   - Give a unique identifier to this document so that you can refer to it in the report, for instance.
2. **Features to be tested**
   - Identify test items and describe the features and combinations of features that is going to be tested.
     - *Numerical IPv4 addresses*
     - *User info (names and passwords)*
     - ...
3. **Approach refinement**
   - Tell how the tests will be run and what will be needed to do that. To reach the right level of detail, imagine that you are writing instructions for unskilled testers.
   - For example, list the files that should be created or modified for test runs and the files that will be created during the test run. Tell what the files are for and describe their contents. List the commands that will be executed and why. What should the tester do during a test run? How is the data going to be gathered and analysed? Is it the same for all test cases or just for some sets of test cases?
4. **Test identification**
   - Here you present what features are tested by which test case.

- Give an identifier and a brief description of each test case: which features (item 2) are tested by each test case. Note that the same test case can test many features.

  For example, you can present identified features and test cases in a table like this:

	Username	Password	Hostname
Empty	`TC1`	`TC1`	–
Very long	`TC2`	`TC2`	`TC2`
Illegal	–	`TC3`	–

  In the table it is easy to see that empty username and empty password are tested in test case `TC1`. It also shows that there is no test case that would test a URL with an illegal hostname. However, do not forget features like "handling too many '`../../`' in relative URLs", which might not fit in the table. You may use more tables, trees or some other ways to systematically derive and present test cases and the features they test.

5. **Priorisation**

   - There might not be enough time to execute all the test cases you have designed. To prepare for this situation, divide your test cases in three classes based on their importance:
     - *Critical.* At least these test cases should be executed and passed before the new browser can be released.
     - *Important.* These test cases should be run next, if there is still time left.
     - *Low.* The rest of the test cases. Important, of course, but not necessarily as important as the others.
   - Explain why you divided the test cases like you did. (A risk analysis with impact and probability estimates is required.)

6. **Dependencies between test cases**

   - Describe the dependencies between test cases and their effects on test runs. What causes the dependencies? Which test cases (or test suites) should be executed before the other? If some test cases fail, should the tester skip some other test cases? Explain why.

7. **Test cases** At least the following information should be included in every test case.

   - Test case identifier
   - Time when the test case was created
   - Time of last modification
   - Priority
   - Input specifications
   - Output specifications

In addition to the information above, you may provide more detailed explanation and/or appropriate references to the RFC. From the references, it is possible to verify why the password in test case `TC3` is illegal. If you have to check the correctness of your test cases later on (was the error in Firefox or in the test case?), you may be in trouble without any explanations and references to specifications.

If necessary, divide test cases to test suites. It helps the reader and it helps you to document the common properties of test cases without having to repeat them over and over again.

In the following example of test cases the first test case `ok-allfields` includes all the fields that the modified Firefox parser outputs when it is given a valid URL. In the output specifications of test cases you need to specify *only the values of the fields that should be checked* in the test case (see the second test case for example). In the output specifications of a test case `err-IPv4` shows the output in case of an invalid URL.

Test case id	`ok-allfields`	
Times (c/m)	*2005-07-14 13:30 / 2005-07-15 16:00*	
Priority	*Critical*	
Input	`http://usr:pwd@tut.fi/d1/f.txt;p1?q#r`	
Output	*result*	`valid URL`
	port	`-1`
	scheme	`http`
	authority	`usr:pwd@tut.fi`
	username	`usr`
	password	`pwd`
	hostname	`tut.fi`
	path	`/d1/f.txt;p1?q#r`
	filepath	`/d1/f.txt`
	directory	`/d1/`
	basename	`f`
	extension	`txt`
	param	`p1`
	query	`q`
	ref	`r`
Test case id	`relative-parentdir`	
Times (c/m)	*2005-07-14 13:00 / 2005-07-14 13:00*	
Priority	*Important*	
Input	*base URL:* `http://www.cs.tut.fi/tos/su/`	
	relative part: `../../../tos/tausta.jpg`	
Output	*result*	`valid URL`
	path	`/tos/tausta.jpg`
Test case id	`err-IPv4`	
Times (c/m)	*2005-07-14 13:00 / 2005-07-14 13:00*	
Priority	*Low*	
Input	`http://www.  foobar.com/999/foo.txt`	
Output	*result* `invalid URL`	
Special notes	*Host name must not include white space [RFC3986 3.2.2].*	

7.2 Test summary report (the fourth task)

The outline for the test summary report is similar to the one in Section 14.2 on page 18.

8 Testing tasks

8.1 System test design (the first task)

1. Familiarise yourself with the URL specifications [RFC3986].
2. Based on the RFC document, identify the features to be tested (empty username, extremely long password, invalid IPv6 address...).
3. Design test cases that test the features you identified. Specify inputs (URLs), expected outputs (the results of the parsing).
4. Decide how you run the test cases. (Approach.)
5. Prioritise the test cases. (Risk analysis.)
6. Write system test design document, outlined in Section 7.1. Students are adviced to check out Sections 5 and 6 in [IEEE829].

8.2 System test (the fourth task)

1. Set up environment for system testing.
2. Create the files you need for test runs.
3. Run your test cases.
4. Check the results carefully. Did you find errors? Were errors in the test cases (fix them!) or in the browser?
5. Write test summary report outlined in Section 14.2.

9 Environmental needs

You need access to Lintula network (`cs.tut.fi`). There you can test Firefox on any Linux workstation (see `http://www.cs.tut.fi/lintula/koneet/`) or, in a case of emergancy, on Lintula student server Haikara. If you really have to do this, at least use compression (option `-C`) in your ssh connection:

```
ssh -C haikara.cs.tut.fi
```

Before starting Firefox, make sure that you do not already have any versions of Firefox running. To make sure that the Firefox does not mess up your current

Firefox plugins, bookmarks etc., move your .mozilla directory under some other name. For example:

```
~$ mv .mozilla mozilla-backup
```

Start Firefox CE by running `/share/tmp/testaus/firefox3.1a1/run.sh`. After the test run you can remove `.mozilla` directory created by the Firefox CE and restore your old settings from the backup.

```
~$ rm -rf .mozilla
~$ mv mozilla-backup .mozilla
```

Part II

Test plan for unit testing

10 Test items

A slice of the class hierarchy that implements the URL parser of Firefox 3.1 alpha 1 will be tested. Note, that here we are testing the original URL parser library, not a modified CE version as in the system test. All errors you can find are real.

11 Features to be tested

Four methods of the class nsStdURLParser will be tested. Three of the methods are implemented in its base classes. The methods are

- ParseURL (85 %)
- ParseAfterScheme (85 %)
- ParsePath (85 %)
- ParseServerInfo (92 %)

The required **multicondition coverage** [Mye04] ("moniehtokattavuus" in Finnish) percentages are given in the parentheses for each method.

12 Features not to be tested

No other than the above mentioned methods is to be tested. Be sure to test the right implementations of the methods. There are several implementations under the same names, but only one of them is called when the call comes from a nsStdURLParser object. Non-functional features of the methods are not tested.

13 Approach

13.1 Class hierarchy

The class hierarchy to be tested is located in `nsURLParsers` module. The code inside `nsURLParsers` has not been modified, but the module has been extracted from the Firefox source tree and relocated to

```
/share/tmp/testaus/unittest-READONLY
```

The directory includes all the necessary header files (most of them are empty) needed for compiling the module. To run the tests, drivers and stubs have to be implemented.

13.2 Stubs

`nsIURLParser` module includes the skeletons of two functions, (`net_isValidScheme`, `IsAsciiAlpha`), one method (`ToInteger`) and one class constructor (`nsCAutoString`). Some of them are necessary for your test runs.

The necessary stubs should pass the cppunit tests defined in `testCAutoString` and `testUtilityFunctions` modules. Do not care about the unnecessary ones. You can run the stub tests by setting up the unit test environment, explained in Section 15.3 on page 20, and running in there

```
% make run_stubtest
```

To implement the stubs, you are allowed to extract code from the Firefox source tree (`/share/tmp/testaus/firefox3.1a1`), or you can write your own implementations. The latter is probably much easier because the needed functionality is very limited.

13.3 Driver

`cppunit` will be used to run tests, so you should write the test cases in cppunit classes. To find out how to do that, you can use modules `testCAutoString` and `testUtilityFunctions` as examples, and refer to cppunit documentation [CPPUNIT].

Note that the same test cases should be immediately runnable also with the new versions of the unit under test. Therefore, **do not modify the code of the unit under test.**

14 Test deliverables

14.1 Test design (the second task)

The outline of the unit test design document is quite similar to the system test design, but there are some changes.

1. **Features to be tested**

To find out which truth value combinations you should cover, use CTC++. A quick guide for that:

```
$ . /share/testwell/bin/testwell_environment
$ /share/tmp/testaus/scripts/setup.unittest.sh
$ cd unittest
$ make clean
$ make run_stubtest
$ ctcpost MON.dat | less
```

Consider each truth value combination as a separate feature. For example, true, true, false, "don't care" in line 97 in ParseURL should be covered by some test case. Do not include this section to the unit test design document, because everyone can reproduce the list of these features with CTC++.

2. **Approach refinement**

   - Tell how the tests will be run and what will be needed to do that. Again, to reach the right level of detail, imagine you are writing instructions for unskilled testers.
   - For example, list the files that should be created or modified for test runs and the files that will be created during the test run. Tell what the files are for and what they contain. List the commands that will be executed and why. What should the tester do during a test run? How is the data going to be gathered and analysed? Is it the same for all test cases or just some sets of test cases?

3. **Test case identification**

   - Explain how you have divided test cases to separate suites. Which sets of features are tested by which suite?
   - You do not need to identify individual test cases and the features they test.

4. **Pass/fail criteria**

   - Specify the criteria to determine whether tested functions and the class structure as a whole have passed or failed the test.

5. **Dependencies between test cases**

   - Describe the dependencies between test cases and their effects on test runs. What causes the dependencies? Which test cases (or test suites) should be executed before the other? If some test cases fail, should the tester skip some other test cases? Reason why.

6. **(C++) Test cases** At least the following information should be included in every test case. The information is given in the CPPUnit code.

   - Test case identifier. This is the name of the method that runs the test case.

- Time when the test case was created. Give this as a comment in the implementation of the method. For example:
  `/* Created:  2008-09-22 */`
- Time of the last modification. Also as a comment. For example:
  `/* Modified:  2008-09-25 */`
- Input specifications (which method is called and what are the arguments). This should be easy to read from the code.
- Output specifications (expected return value, expected values in call arguments given by reference, what else has changed its value). This should be easy to read from the code.
- Environmental needs (which stubs are required to execute this test case). Give this as a comment in the method. For example:
  `/* Requires stubs:  stubxyz */`

Include the test cases, that is the CPPUnit methods you have written with the above mentioned comments, in the PDF. Readability should be considered very important here. Do not include headers or other code that does not describe functionality of the test cases.

14.2 Test summary report (the third task)

Outline of test summary report is the following.

1. **Cover page** — provide the names and student numbers of the testers and give a unique identifier to this document
2. **Summary** — answer to the following questions
   - Exactly what was being tested? Specify version/revision level.
   - In what environment the testing activities took place? List everything that may affect test results. For instance, provide information on the computers (hostname, hardware, software) that were used in the test runs. Version numbers of the testing tools and other relevant software are important.
   - Which test design documents and/or test plans was the test based on? Give references to the correct versions of appropriate documents.
3. **Variances** — report how test execution differed from the test plan or test design. Specify reasons for variances.
4. **Comprehensiveness assessment** — evaluate how well the tests covered what was required in the test plan. Identify features that were not sufficiently tested and explain the reasons.
5. **Summary of results — This is one of the most important parts in the test report**: everything suspicious should be mentioned and all errors should be documented so well that they can be easily reproduced by developers.

Identify all test incidents, that is situations where a test case failed or other anomalies arose. Provide the following information on each incident: inputs, expected results, actual results, anomalies (any other suspicious behaviour?), date and time, environment, attempts to repeat the error, testers, observers. Furthermore, what **caused** the incident: was there an error in the test case (was the specification misinterpreted?), in the test execution, in the system or in somewhere else? What **impact** the incident had on the test plan, the test design, the test case specifications or the test execution?

6. **Evaluation** — Provide an overall evaluation based on the test results and pass/fail criteria in test plan and design documents.

7. **Summary of activities** — Summarise the major testing activities and events. Estimate total elapsed time used for the activities. Report the sum of the working hours of all students in the group.

8. **Approvals** — Specify the names of persons (your course assistant) who must approve this report. Provide space for signatures and date.

15 Testing tasks

What you should do in the second and the third tasks is described in the first two subsections. The tools needed are shortly introduced in the following subsections.

15.1 Unit test design (the second task)

1. Read the code of the methods that will be tested.

2. For each method, identify the "features" to be tested as described in Section 14.1 item 1.

3. For each method, design test cases so that every identified feature will be tested. Specify the parameters that are given to the method and what output is expected (what is returned by the function and what is returned in the parameters). Write the test cases in CPPUnit classes.

4. Find out how the test cases will be run.

5. Write the unit test design document.

15.2 Unit test (the third task)

1. Set up your unit testing environment.

2. Implement the necessary stubs in `nsIURLParser.cpp` file.

3. Modify the Makefile so that command `make run_unittest` runs the cppunit test suites and measures the multicondition coverage of the code in `nsURLParsers` (`MON.sym` and `MON.dat` files are generated when the command is run).

4. Write the test report outlined in 14.2.

15.3 Setting up the unit testing environment

Execute

```
/share/tmp/testaus/scripts/setup.unittest.sh
```

in a directory under which you want the unittest environment to be set up.

Less than 300 kB of space will be required.

15.4 Setting up CTC++

CTC++ is used to instrument the code during the compilation. When the instrumented code is run, `MON.dat` and `MON.sym` files will be created. They contain the coverage information of the instrumented code.

You can examine the results by executing `ctcpost MON.dat`. Whenever the CTC++ tools are used or the instrumented code executed, you must have the CTC++ environment variables set. It can be done by running

```
. /share/testwell/bin/testwell_environment
```

(The dot and the space in the front of the line are important!) When the environment is set, you can see the manual page of CTC++ with command `man ctc`. For the purposes of this project it should be enough to understand the ctc commands that are explained in Section 15.5.

15.5 Using cppunit and CTC++

Cppunit provides means of dividing test cases into test suites. It provides macros for implementing test suites and stating assertions in the test cases. What you need is

- the unit under test (object code is enough)
- the main program (object code is again enough) and

- a class for each test suite (a test case in a test suite is a method in the corresponding class. The classes and the methods are what you need to implement).

Given that you have the unit test environment ready (see 15.3) and you also have CTC++ environment variables set (see 15.4), run

```
make clean
make stubtest
```

The commands exececuted by GNU Make are explained in the following.

```
ctc -i m g++ -o stubs.o -c nsIURLParser.cpp
```

This command compiles `nsIURLParser.cpp` to `stubs.o` object file. `nsIURLParser.cpp` contains the stubs that you will implement in the third task. The code is instrumented with CTC++, because in "stubtest" the stubs are the unit under test. (When you test `nsURLParsers.cpp`, it will be instrumented instead of the stubs.) `-i m` switch tells CTC++ to instrument the code for measuring the multicondition coverage.

```
g++ -I. -Wall -pedantic -o testRunner.o \
-c testRunner.cpp
```

This compiles the main program. The main runs all the test cases that are written in cppunit classes in the other `test*.o` files. You do not need to edit `testRunner.cpp` in any task.

```
g++ -I. -Wall -pedantic -o testCAutoString.o \
-c testCAutoString.cpp
```

This compiles the cppunit test class that will test the implementation of CAutoString stub class.

```
g++ -I. -Wall -pedantic -o testUtilityFunctions.o \
-c testUtilityFunctions.cpp
```

Similarly to the previous command this compiles again a class containing test cases. These test cases will test the stubs of utility functions.

```
ctc -i m g++ -o stubtest stubs.o testRunner.o \
testCAutoString.o testUtilityFunctions.o \
-L. -lcppunit
```

This links the object files to an executable binary. Note that the unit under test (in this case `stubs.o`) and the main program (testRunner) do not need changes nor recompiliation when new test cases are created and added. Just relink the two object files with whatever test case object files you have, and you will get a new binary which executes all linked test cases.

Test cases are implemented in cppunit test classes (test suites) as separate methods. In addition to test cases, test suites include also `setUp` and `tearDown` methods. They are automatically called every time a test case is executed (the first one before and the second after the execution).

Test cases do the testing and then check the results with assertion macros, for example with `CPPUNIT_ASSERT_MESSAGE(assertion,message)` macro. This macro checks the given assertion and, if it fails, causes the execution of the test case to return with verdict "fail" and the given message to be printed. The test case is passed if the execution of a test case reaches the end of the method.

Note that CTC++ collects coverage data in `MON.dat` cumulatively if the file already exists. Therefore, to see the real coverage of your test suite, first delete the file, then run the test and finally examine the coverage with `ctcpost`.

16 Environmental needs

Unit tests have to be run in Lintula Linux workstations or servers. Only the Linux version of cppunit is available in the unit test environment.

Use bash shell rather than tcsh.

Troubleshooting

I cannot create ssh connection to haikara.cs.tut.fi.

A firewall blocks ssh connections from the wilderness (outside the TUT domain) to haikara. You need to log in to some other computer in TUT (for example, `ssh2.cs.tut.fi`) and then continue to `haikara`.

I cannot execute command...

The problem is most likely some of the following: wrong shell (use Bash), wrong OS (use Linux) or lacking environment variables (run `. /share/testwell/bin/testwell_environment`).

g++ gives warnings

`g++` does not like non-virtual destructors in virtual classes inside CPPUnit code. You can ignore those warnings. CTC++ instrumentation may also produce some code which generates warnings.