Traceability tool

# Introduction

This project aims at simplifying the management of traceability between tests (unit tests (C#, java), cucumber tests, ALM tests) and specification requirements.

# Principles

We consider we are in the following configuration:

* The specification document is a Word document (.docx) where requirements are defined within the first column of tables (see example in Table 1 ). The table can have different number of columns or different column name. The only important thing is that the requirement identifiers are located in the first columns (their style do not matter). Indeed, in order to not prevent using tables for something else in specification documents, the tool will allow defining a regular expression to decide whether what is found within the first column of a table is actually a requirement or not, so try to find a naming convention of requirements that makes it easy (e.g., having all of them starting with the same prefix).
* The requirements are covered through one of the following kinds of test:
  + Unit tests. The traceability to requirements must be handled in the code of the unit tests. This is, for the moment, only handled for tests in Visual Studio for C#.
  + Cucumber tests. The traceability to requirements must be handled in the scenario steps. This is handled in cucumber scenarios with a specific Then clause.
  + ALM tests. The traceability to requirements must be defined in the expected result with a specific syntax. This is handled by analyzing the export to CSV made from HP ALM.
* There is also a possibility to justify why a requirement is not covered, through an Excel file called justification file.

**SD**

REQ-1

REQ-3

REQ-2

REQ-4

REQ-5

REQ-6

…

**ALM**

**extract file**

**XML**

**files**

**Feature**

**files**

**HP ALM**

**Cucumber**

**C# unit tests**

**Java unit tests**

**Java**

**files**

**Justification file**

Figure 1 – General requirement coverage overview

|  |  |  |  |
| --- | --- | --- | --- |
| Requirement | Title | Description | US |
| REQ-1 |  |  |  |
| REQ-2 |  |  |  |

Table 1 – Example of table defining requirements

From the input SD file and all the analysis of the different kind of tests, a traceability matrix can be generated to indicate which requirements of the SD are covered by which tests, and indeed which requirements are not yet covered by a test.

## Project organization

TODO

## Unit tests

### C# unit tests under Visual Studio

To be able to extract the coverage of C# unit tests, the compilation of the test project shall be configured to generate the XML documentation. Figure 1 shows how to configure that in Visual Studio, once the properties of the test project opened.

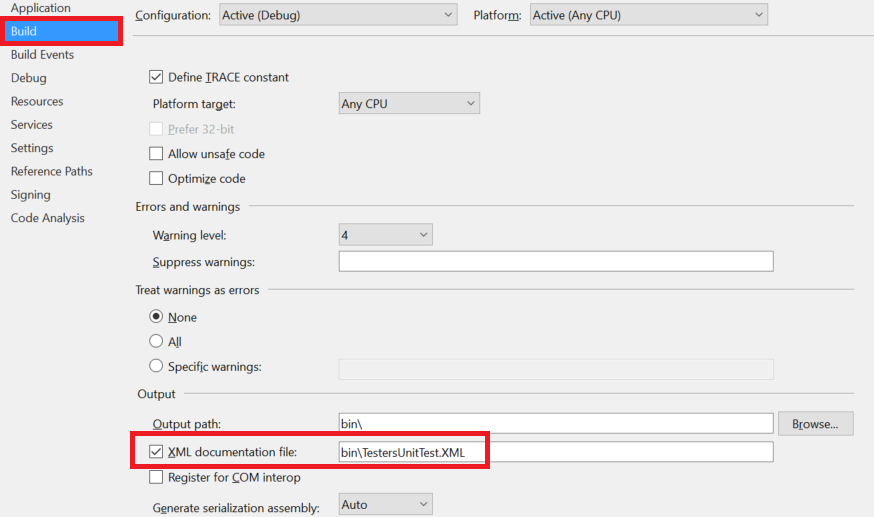


Figure 2 – Configuration of C# test project to generate XML documentation

Then, the following convention needs to be applied in testing functions in the classes:

/// <summary>

/// Description of the test.

/// <para>Input: Description of used inputs</para>

/// <para>Expected Results: description of expected results</para>

/// <para>TestID: Test\_Identifier</para>

/// <para> Covered REQ: REQ-01, REQ-02, REQ-03</para>

/// </summary>

[TestMethod]

public void VTP\_TestMethodName()

{

// Test method code

}

* For the method comment:
  + Add a para markup starting with TestID: followed by the test identifier.
  + Add a para markup starting with Covered REQ: followed by a comma separated list of requirement identifiers, for the requirements being covered by the test.
* The test method name shall be prefixed with VTP\_.

With that:

* the contents of the summary markup (except for the para with TestID: and the para with Covered REQ:) will constitute de description of the test,
* the method name will constitute the test name,
* the test identifier mentioned in the para with TestID: will be used as tests identifier,
* the requirements extracted from the para with Covered REQ: will indicate which requirements are covered by the tests.

### Java unit tests

To be able to extract the coverage of java unit tests, the following convention shall be followed.

The testing method shall have an associated Javadoc organized this way:

/\*\*

\* Description of the test.

\*

\* @testId "MyTestId"

\* @expectedResult "I expect this result"

\* @coveredReqs "COVERED-REQ-1, COVERED-REQ-2"

\*/

@Test

public void VTP\_testMethodName() {

…

}

With that:

* the contents of the Javadoc (except for annotations) will constitute de description of the test,
* the method name will constitute the test name,
* the test identifier mentioned in annotation @testId will be used as tests identifier,
* the requirements extracted from the annotation @coveredReqs: will indicate which requirements are covered by the tests.

**Warning** : A constraint, for the tool to work, is that there is no / inside the Javadoc section.

## Cucumber tests

In order to manage the traceability between cucumber tests and the requirements in SD, we need to mention SD requirements directly in the scenarios.

The basic is that those requirement references are to be placed in the final Then clause after a series of Given, When or And:

Given ...

Given ...

When ...

When ...

Then test result

Then Reference SD [Req 1],[Req 2]

Given ...

Given ...

Then test result2

Then Reference SD [Req 3],[Req 4]

The clause Then followed by Reference SD, followed by a comma-separated list of requirements placed between square brackets indicates that all the previous steps (i.e., all Given, When, Then preceding this Then) are validated, and hence the requirements are covered if the scenario reaches this step.

## ALM tests

The tests created in HP ALM shall have the covered requirements of the SD mentioned in the expected result between square brackets.

When exported in an Excel file, we expect to have the following columns:

1. **Test Suite**: Name of the test suite,
2. **TC No**: number of the test case,
3. **Test Case Title**: the title of the test case,
4. **Step Name**: the name of the step,
5. **Action**: the description of the action to perform for this step,
6. **Expected Result**: the description of what shall happen after the step is performed. This is where the requirements shall be mentioned.

## Generic tests

As we have heterogeneous test to manage, but only one target VTP to write, we need to define a generic way to represent any test.

Here is what has been chosen:

* **Test set name**: this is the name of a set of tests, allowing to group tests dealing with the same topic together,
* **Test case identifier**: this is a unique identifier for a test case,
* **Test case name**: this is a name for a test case that indicates in a concise way what the test case is about,
* **Step description**: description of a step of a test case, indicating what is done and whose consequence will be tested,
* **Expected result**: description of what is checked after the step has been performed,
* **Covered requirements**: this is the list of requirements from the SD that are covered by this step.

The composition of test elements is represented in Figure 2.

**Test bundle**

**Type** : Automatic/Manual/Unit

**Test set**

**Name** :

1..\*

**Test case**

**Identifier** :

**Name** :

1..\*

**Test step**

**Step description** :

**Expected results** :

1..\*

**Requirement**

**Identifier** :

0..\*

Figure 3 – Composition of test elements

### Mapping

Table 1 describes the mapping between the different kinds of tests and the generic tests attributes.

| Generic tests | Unit tests  C# | java | Automatic tests | Manual tests |
| --- | --- | --- | --- | --- |
| Test bundle | Unit | | Automatic | Manual |
| Test set name | The assembly name. | The java class name. | The string found after the Feature: keyword. | Name of the test suite. |
| Test case identifier | Identifier found in the para markup starting with TestID: . | Identifier found as value of annotation @testId. | The string found after the Rule: or Scenario Outline: keyword. | Test case number. |
| Test case name | Method name. | Method name. | The string found after the Scenario: keyword. | Test case title. |
| Action description | The contents of the summary markup with the internal para markups contents except for Expected result: and Covered REQ:. | The testing method description. | The scenario steps including all the Given, But, And, \* or When that precede a Then. A step is thus a series of Given, When followed by one or more contiguous Then. | Action. |
| Expected result | The text after the para markup starting with Expected Result:. | The description found after annotation @expectedResult. | The scenario steps including all the contiguous Then except those followed by Reference SD. | Expected result without the requirement part. |
| Covered requirements | The requirements extracted after the para markup starting with Covered REQ:. | The requirements (separated with commas) found after annotation @coveredReqs. | The requirements extracted after the Then followed by Reference SD. | The requirements contained in the expected result part. |

Table 2 – Generic tests mapping from specific tests

# Design

This is a java project managed with ANT.

Table 3 shows the high-level organization of the code, under the root package org.tools.doc.traceability.

| **Path** | | | | **Description** |
| --- | --- | --- | --- | --- |
| common |  |  |  | Common classes |
|  | cmdline |  |  | Classes to deal with command line tools. |
|  | exceptions |  |  | Exceptions used within the applications. |
|  | executor |  |  | Classes for executors, which are classes that perform a process that updates a status for the caller to get the result of the execution. |
|  | gherkin |  |  | Classes used to extract the contents of a Gherkin scenario file. |
|  | model |  |  | Common model classes. |
|  | ui |  |  | Common GUI classes (generic panels and action) |
|  | worksheet |  |  | Common classes to deal with worksheets (styling, format). |
|  | xml |  |  | Utility class to validate XML file against XSD. |
| covmatrixgen |  |  |  | Tool for generating the traceability matrix between requirements from SD and the manual and automatic tests. |
|  | model |  |  | Model classes for this application |
| gui |  |  |  | The GUI part of the Traceability tool. |
|  | components |  |  | Widget components |
|  |  | checkbox |  | Check boxes |
|  |  | inputfield |  | Input fields |
|  |  | radiobutton |  | Radio buttons |
|  | configuration |  |  | Classes about tool configuration |
|  |  | appmodel |  | Model classes to make the link between java and XML. |
|  | controller |  |  | The application controller. |
|  | data |  |  | Classes representing the managed data. |
|  |  | app |  | Data about applications (WMT, BMT, AIM ...) |
|  |  | tools |  | Data about the tools (Coverage matrix generator, VTP Updater) |
|  |  |  | config | Class modeling the configuration of the tool. |
|  |  |  | data | Data about the tool. |
|  |  |  | execution | Tool execution statuses classes |
|  | frame |  |  | Frame classes |
|  | panels |  |  | Panel classes |
|  |  | app |  | Class about the application choice. |
|  |  | config |  | Class about the chosen application configuration. |
|  |  | control |  | Control panel |
|  |  | execution |  | Execution panel classes (includes result panel). |
|  |  | tools |  | Panels corresponding to the selected tool. |
| reqextraction |  |  |  | Tool for extracting requirements from a docx (SD) file. |
| unittestcoverage |  |  |  | A tool parsing the xml files generated while executing unit tests with Visual Studio, and extracting the contained data. |
|  | model |  |  | The class representing the data used by the tool. |
| vtpupdater |  |  |  | A tool that reads the test extract from HP ALM that updates the relevant sheet of the destination VTP with its contents. |
|  | model |  |  | The class representing the data used by the tool. |

Table 3 – High-level description of the project's organization