Guide to test artifacts and procedures

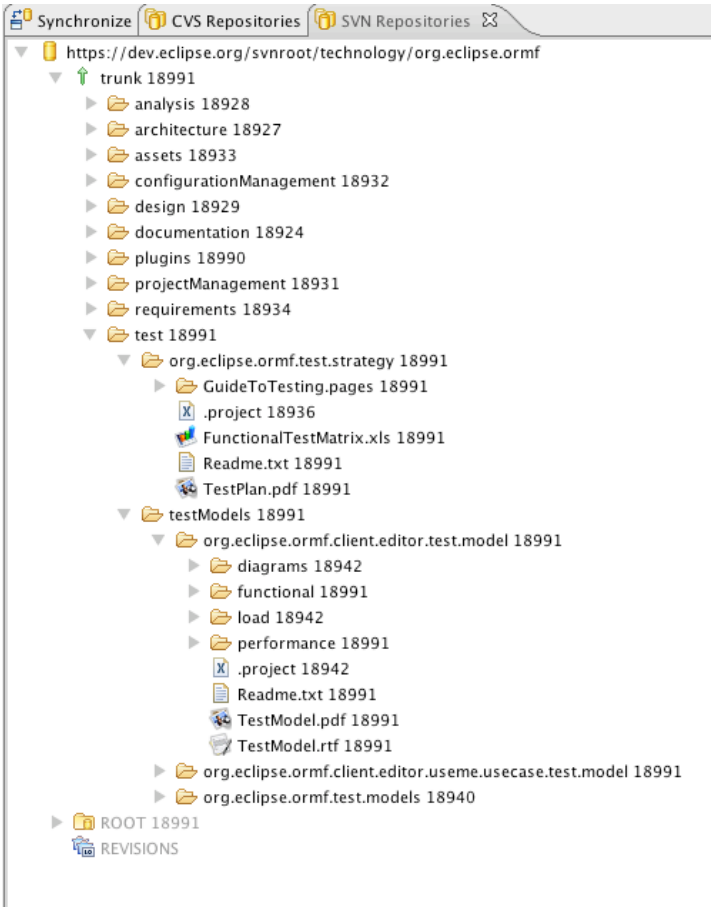
Introduction

This document describes the existing test related artifacts and the processes that have been followed to date, with particular emphasis to functional testing.

The aim of this document is to provide a succinct explanation of the strategy that has been followed in testing Useme so far and a starting point for what will become the test strategy to be applied to ORMF in the foreseeable future. It is one of the goals of the first iteration of ORMF development to either extend or replace this testing strategy to create a testing framework that will serve us for the future development of ORMF.

The location of the artefacts in the ORMF repository

All test related artefacts are stored in the high level **`test`** folder of the ORMF repository. The **`test`** folder is organised as indicated in the figure that follows. It contains two high level childern, namely **`org.eclipse.ormf.test.strategy`** and **`testModels`**.



`org.eclipse.ormf.test.strategy` is an Eclipse project containing all artefacts that establish the overall testing strategy that is followed in ORMF and that control the overall testing evolution. Besides this document, this folder currently contains the Test Plan that was initially devised for Useme and the Functional Test Matrix, a spreadsheet that completely defines the functional tests identified for each Useme use case, as well as their current status of development. Both of these artefacts are described in more detailed in sections The overall picture: the Test Plan and The Functional Test Matrix below.

`testModels` is a folder that contains three Eclipse projects, each of which represents the test model, complete of all test cases, test matrices and ancillary artefacts for a specific ORMF component. So far we have a test model for the overall ORMF framework (**`org.eclipse.ormf.test.models`**), one for the generic ORMF editor client components (**`org.eclipse.ormf.client.editor.test.model`**) and one for the Useme editor client components (**`org.eclipse.ormf.client.editor.useme.test.model`**).

All three projects have the same internal structure, which is shown in the figure above for the **`org.eclipse.ormf.test.models`** project: specific directories are allocated for functional, performance and load tests and we also have a directory for any high level diagrams and a comprehensive TestModel

document, which acts as the aggregator of all tests created for the specific component (i.e. **`org.eclipse.ormf.test.models`** in this case), providing any necessary high level description and links to all specific test artefacts.

> N.B.: In order to work with test artefacts, please check out the required Eclipse project(s) in the **`test`** folder and do not check out the entire **`test`** folder.

The overall picture: the Test Plan

This is the document that describes in depth the strategy identified for the whole ORMF project. The current version refers to the original Useme project. Although it has not not been maintained up to date with the evolution of our testing processes, it is a reasonable representation of the fundamental approach we originally chose for Useme. We intend for this document to be revised or replaced by the person that will take overall responsibility for the testing architecture.

The most fundamental concepts that are described in the Test Plan and that are required to understand the artefacts that have been developed so far and that can be found in the repository are described in the sections below. Of the types of tests highlighted in the Test Plan, the ones that have been tacked in practice so far are the following three: Unit (and integration), functional and performance tests. These are described in greater detail in the sections that follow.

Unit (and integration) tests

Unit tests have been performed on Useme, as a means of both testing individual class methods and integration tests. However these have not been maintained up to date with the latest iterations of development and therefore we have not included in the ORMF repository.

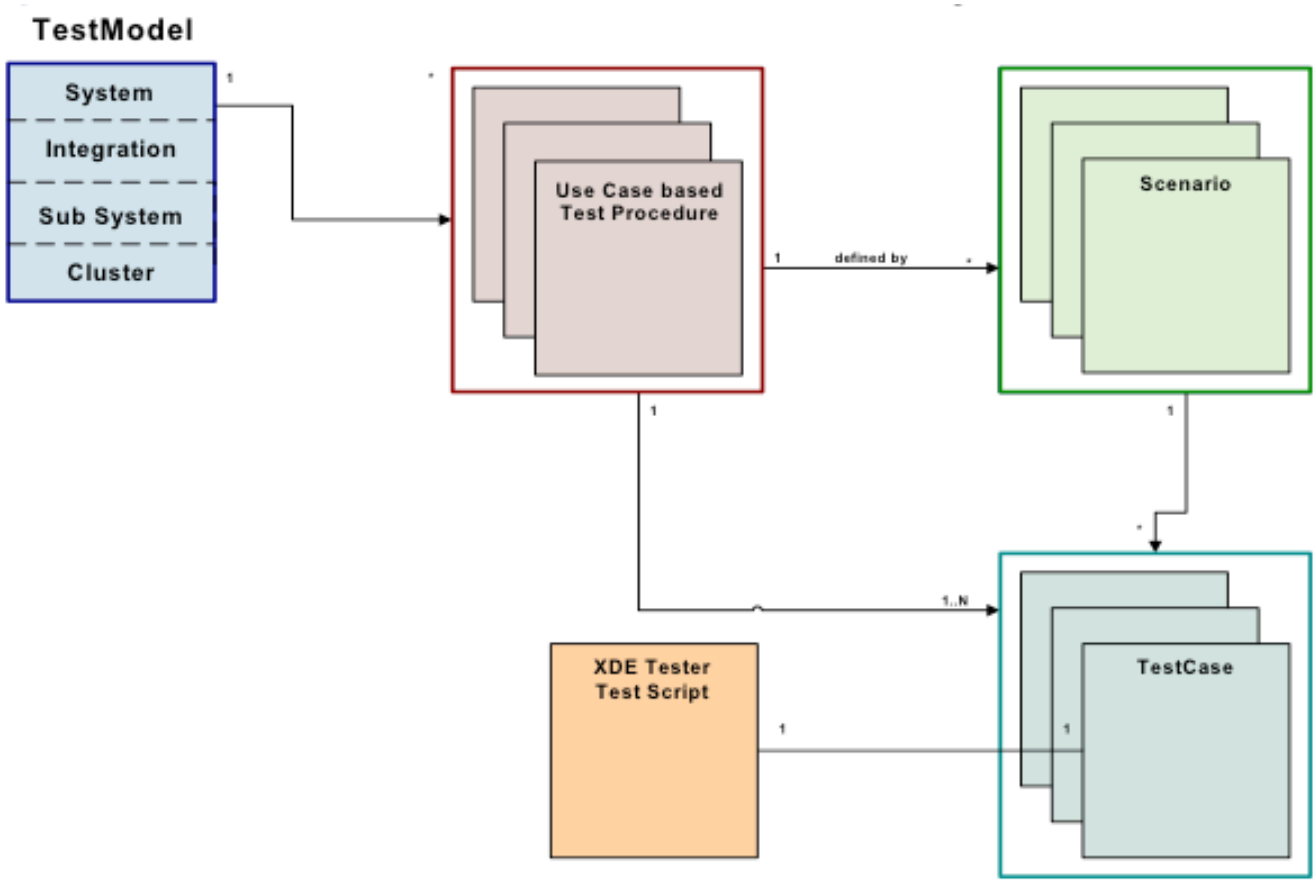
We anticipate that unit testing will be the responsibility of each individual developer (utilising both JUnit and, where appropriate, PDE Plug-in JUnit testing)and that these tests will be collected in appropriate fragments of the plugin projects that they test, as per common Eclipse wisdom. Joel and I will undertake a more detailed organisation of unit testing and how to use it in a regression scenario as part of iteration I1.

Functional tests

Functional tests have been the subject of a great deal of effort on Useme. Unfortunately both tools that have been utilised for this purpose (Rational's XDE Tester followed by TPTP's AGR) have let us down for one reason or another and therefore we do not currently have a working regression suite of test scripts. However we do have a reasonable regression suite of test case documents that describe in detail the intent of each test case, their setup, teardown and detailed steps, along with the required verification points.

In general, functional testing is organised as indicated in the figure that follows.

The fundamental unit of functional testing is the use case. All functional tests are identified and grouped on a use case basis. Each use case has a **Test Procedure** associated with it, which collects all test cases that pertain to the functionality described in the use case. In the current file system, all test procedures are named **`TP.rtf`** (with its corresponding **`TP.pdf`** document). Each one of them is contained in a folder that is located under the **`functional`** folder and that takes the name of the use case that the procedure is meant to test (for example: **`org.eclipse.ormf.client.editor.test.model/functional/`**



`CloseDocuments/TP.rtf`). Any given Test Procedure briefly describes the generalities of the tests associated with the specific use case and then points to a scenario matrix, which is identified as **`Matrices.xls`** in the file system (in the same folder as the Test Procedure).

The scenario matrix is in reality composed of two spreadsheets.

The first one lists all scenarios identified by the use case being tested. These are obtained by listing all flows described in the use case, along with, possibly, the flows of included use cases that are being exercised and tested as part of the primary use case and then constructing scenarios of behaviour in such a way that each use case flow is exercised at least once. The use case's preconditions are also tested as these typically result in the disablement of the visual widgets that trigger the action described by the use case.

The second spreadsheet in the scenario matrix summarises the test cases that have been identified in order to exercise a specific scenario. For each test case, we show in a synthetic tabular format the corresponding scenario, whether or not the identified test case is going to be implemented (and, if not, the reason why it is not implemented), the specific conditions that have to be met for the test case to run and all expected results.

Each test case identified in this second spreadsheet has a hyperlink to the test case's textual document, which is typically named **`TCnnn.rtf`** in the file system. The textual document contains a detailed description of the setup and teardown requirements for the test case, of the detailed steps required to exercise the specific scenario and of the required verification points.

For a few of the test cases developed early on, we also had a test matrix that specifically defined the conditions, results and, if necessary, data used in that specific test case. These can be found in the repository as files named **`tim_nnn.xsl`**. However, these matrices were later on deprecated because too time consuming to create and maintain for little value in return.

All the identified Test Procedures are listed and linked from the above mentioned TestModel document (**`TestModel.rtf`**), in its Functional Testing section (as mentioned, the TestModel document also contains pointer to the other types of testing beyond functional testing).

Finally, setups and teardowns are themselves represented as test cases (and corresponding executable test scripts) and stored in specific **`setups`** and **`teardowns`** directories located inside of the **`functional`** folder. The location of these two directories recognises the fact that setups and teardowns are reusable across testing of multiple use cases.

Please note that the many test cases, scenarios and procedure documents are in different stages of evolution. This is due to the way in which, historically, the testing effort evolved in Useme and to the fact that, to date, a final satisfactory solution to the tool problem has not been found, and therefore there has not been an opportunity to review, rationalise and bring all test cases to the same level of maturity. This will be the task of the team member that takes on board functional testing during iteration I1. After an initial regression suite and all corresponding documentation is complete, we will then discuss who will be responsible for functional testing for functionality that will be added in future iterations.

Finally, note also that no executable test scripts have been placed in the ORMF repository as they depend on testing tools that are no longer being used.

The Functional Test Matrix

The Functional Test Matrix is a set of spreadsheet that acts as an overall control of all functional testing in the system. It currently contains two relevant spreadsheets (a third one is for future development and is not important for this discussion).

The first spreadsheet expresses all use case relationships and identifies which primary use cases are responsible for testing, either partially or fully, the secondary use cases that they include/specialise/provide extension for. The rows represent the primary use cases (i.e. use cases that are initiated by an actor) and the columns represent secondary use cases, not initiated by an actor but utilised by primary use cases for segments of their flows. The convention used to indicate a relationship is as follows: the relevant cell contains a two letters code, with the two letters separated by a "**`/`**" character. The first letter indicates whether the secondary use case is tested or not and, if it is, whether it is tested in full or partially. "**`N`**" indicates that it is not tested, "**`P`**" indicates that it is partially tested and "**`F`**" indicates that is it fully tested. The second letter after the slash indicates the type of relationship, whether it is an inclusion "**`I`**", an extension, "**`E`**" or a specialisation, "**`S`**". The goal is for all columns to have exactly one cell which contains an "**`F`**" and one or more cells with a "**`P`**".

The second spreadsheet is an audit spreadsheet listing all test cases that have been identified for any given primary use case and, for each test case, its exact status of development. The legenda provided in the spreadsheet clearly states how to read the status of a test case depending upon the colour code of the cell's background.

We have found this matrix to be extremely useful in providing a snapshot of the testing effort and we would like to see it maintained in the future, unless better controls at this overall level are suggested.

Performance tests

The performance testing effort was initiated quite a long time back in order to test the length of operations that were deemed to be a little too slow and the purpose of the tests was of course to measure improvement as algorithms were modified or replaced to make execution faster. Currently all editor related operations are quite responsive and timely and we do not feel that the performance tests that we designed and implemented are terribly crucial or relevant at present. However it may be useful to keep them in place for the time being, especially in view of the fundamental changes that we are planning for release 1.0.0 of ORMF/Useme.

Performance tests are captured via textual documents that are similar to the ones for functional test cases in nature and format. They are named according to the function that they test and they are located in the **`performance`** folder of the appropriate test model folder for the system's component that contains that function being tested. They are listed and linked from the TestModel document for the appropriate software component. The ORMF/Useme code is equipped with a timer that, when utilised, automatically calculate the time it takes to perform a given operation. This can be placed at the appropriate locations in the code and activated or deactivated using a command line argument during execution of the client.

Tests are typically performed manually and repeated five times. The resulting times of execution, as reported by the timer, are averaged and the average is recorded in a spreadsheet that takes the name of the corresponding test case, with the word "**`Results`**" appended to the test case name. This spreadsheet contains a cumulative history of the test results as performed during any given iteration.

Conclusion

This document was aimed at briefly describing the status of our testing effort that we developed as part of Useme and that ORMF is inheriting as a starting point. One of our goals during iteration I1 of ORMF is to readdress testing, especially functional testing, in a comprehensive way and to produce a substantial functional regression test suite that may be exercised with the current version of Useme and then utilised in the following iterations, as we replace many of the underlying components of the Useme architecture.

Although many of the features described in this document have proven useful to us during the course of development of Useme, we are not professional test analysts, test architects or tester and we are therefore very open to any suggestion for improvement and/or change in either testing artefacts or processes.