# Testing Guidelines for the Eclipse ORMF project

| | |
|---|---|
| Version: | 0.1 |
| Status: | Draft |
| Date: | 2008-10-21 |
| Author(s): | Achim Lörke |

# What's this document about?

One of the main goals of every software project is to release a product with as few bugs as possible. Since even software developers are merely human they tent do make mistakes. One way to find these mistakes is of course testing the software. By testing I mean using the graphical user interface to access all functions of an application. If you do this during the development cycle it's a tedious and unthankful task: test the software, find a problem, get criticized by the developers (this is optional), wait for a bug fix and start again. The "start again" is the tedious part. After a few internal releases even the most patient tester will become bored (and probably sloppy).

The simple solution to this automated software testing. You just capture your test and repeat it time after time. When a bug is fixed you will just rerun your test and instantly know that all is well. Well, actually it isn't. Invariably someone will change the application you're testing in some ingenious but incompatible way. This will be as a consequence of a changed requirement, a major refactoring of the source code or some changes in the new version of the used GUI framework. This will all break your perfectly good test.

What you have to do is **design** a set of tests which are robust to changes in the application under test (or short AUT). They must also be easy to maintain if the requirements of the application get enhanced or (as ridiculous as it sounds) there is an error in your test suites.

So the goal of this document is to provide some guidelines to design test which

- Are robust to minor changes in the AUT
- Are easy to maintain if the AUT business logic or workflow  changes
- Can recover from errors during test execution to continue in other test scenarios

**This document is a work in progress. It is accompanied by a GUIdancer® project which will be used as a base for the actual test project.**

## Scope

I will only cover a limited scope of GUI testing. This document is intended for the Eclipse ORMF project. In this project the GUI testing tool GUIdancer® is used[1]. Therefore I'll focus on designing test for the ORMF project using GUIdancer®.

I'll assume that you are familiar with using GUIdancer® as a tool and that the terms test suite, test case and test step are meaningful to you.

The test scenarios we are talking about are black box test which will mimic end user tests. All tests are therefore GUI based functional tests. White box tests using jUnit are covered in another document. Performance/load test are also not covered here.

*There is an ongoing discussion on how to document test result and test coverage among the development team members. Output of this discussion will be merged into the next release of this document.*

## General Topics

Test scenarios are usually oriented along use case or requirements specifications. For ORMF we will start using the test scenarios available in the SVN repository[2] at eclipse.org. These scenarios are described in the *org.eclipse.ormf.test.strategy* module in the *guideToTesting/GuideToTesting.pdf* document.

## Defining Tests

How is the overall structure of a well designed test? Well, one proven possibility is the following[3]:

- Use Test cases only rarely to specify your testing environment.
- Define your scenarios in (nested) test cases and use a top down approach.
- Design often used tests as reusable test cases.

A proposal for a GUIdancer® project is available as an exported project file in the SVN repository at TBD.

I'll go into more details in the following chapters.

[1] GUIdancer is a commercial tool for automated GUI testing. It is available free of charge for Eclipse projects. For more information have a look at the GUIdancer® web site.
[2] https://dev.eclipse.org/svnroot/technology/org.eclipse.ormf
[3] This is GUIdancer® specific and may not work for other tools. It also require a working knowledge of the GUIdancer® tool and concepts.

## Test Suites

Test suites are the executable parts of a GUIdancer® project. A test suite is bound to an AUT and contains all the test cases to run for this scenario. For ORMF I suggest using the following naming scheme:

| Test suite | Purpose |
|---|---|
| FULLTEST | This test suite contains all tests for this AUT. It's the final challenge, i.e. the AUT can only be correct if this test suite runs without errors. |
| WORK_<userid>_<whatever> | These are test cases used to try something new or to look into a problem. They user-id is only useful in a shared database. |
| Ticket_<bugzilla-id> | This is a test suite which contains test cases to verify a specific problem. |

Remember that the test specified in a test suite relies on certain start conditions the AUT has to honor. If you want to execute a sequence of test suites you must make sure that this start conditions are met. This can be done by resetting the AUT after each single run or by setting up the AUT before the actual test start. From my experience it's easier to initialize the AUT when starting a test suite. To verify the start conditions you can use "retry event handlers" which can correct the state of the AUT by performing the needed actions. An example of this can be found at TBD.

## Test Cases

Test cases can be divided into two categories:

1. Test scenarios derived from use cases or requirements
2. Building blocks for reoccurring tasks during test execution. This might be login procedures, table entries or some sort of error handling.

### High Level Test Cases

These are the nested test cases derived from the test case scenarios. The root test cases define test for a specific area (linked to the projects in the *test/testModels* module in SVN).These test cases refer to more specific test cases for functional requirements. To add a little bit of structure the test cases are organized in categories which are named after the projects and directories in SVN.

### Low Level Test Cases

The low level test cases are performing generally helpful or needed tasks. This can be

- Some sort of login procedure, including connecting over a network
- Initialization of test suites
- Error recovery
- Logging of the AUT state during an error (including screen dumps)

Examples of these are found in the initial project.

## Workflow

To design a set of test cases we will use the following workflow:

1. Choose a target area from the descriptions in SVN.
2. Define the needed nodes in GUIdancer® (if not already done). The names are taken from the appropriate filenames:
    a. Define a category for the SVN project (Example: org.eclipse.ormf.client.editor.test.model)
    b. Define a test case as root for this category (name it like the category).
    c. Define a test case for the functional requirement (Example: CloseDocument)
    d. Define test cases for every test case in the requirements directory (Example: TC001)
3. Add test steps to the test cases defined in 2.d. Try to use only low level test cases or unbound modules.
4. Add the test cases from 2.c to your root test case.
5. Add the root test case to the FULLTEST test suite.

Picture/Example TBD

To test or retest an error we will use the following approach:

1. Define a test suite named “Ticket_<bugzilla-id>” (Example: Ticket_250648)
2. Define a test case named like the test suite and add it to the test suite.
3. At test steps to reproduce the bug’s behavior. Try to use only low level test cases or unbound modules.
4. Add a comment to the ticket once the test is ready to enable the developer to reproduce the error.
5. Once the error is fixed move the ticket to the appropriate category for normal test cases and add it to the root test case. **Do not delete the ticket test case!** It’s a valuable enhancement of the test suite since it covers a behavior which can obviously be implemented wrong.

Picture/Example TBD