Xaraya
Request for Comments: 0029
Category: Experimental

M.R. van der Boom
J Schrage
F. Besler
Xaraya Development Group
December 2002

Automated Unit testing for Xaraya

Status of this Memo

This memo defines an Experimental Protocol for the Xaraya community. It does not specify a Xaraya standard of any kind. Discussion and suggestions for improvement are requested. Distribution of this memo is unlimited.

Copyright Notice

Copyright © The Digital Development Foundation (2002). All Rights Reserved.

Abstract

This RFC contains a design of a (partially) automated test framework. The design was inspired by eXtreme programming, design by contract and the jUnit testing framework. The ideas have been slightly adapted to make it more suitable for the Xaraya System.

Table of Contents

1 Introduction **3**

2 Requirements **4**

3 Design **6**

3.1 General design 6

3.2 Requirements specific design 6

4 Implementation **10**

5 Integration **11**

6 How to write tests **12**

7 Remaining issues **13**

8 Revision history **14**

9 References **15**

Authors' Addresses **16**

A Glossary **17**

Intellectual Property and Copyright Statements **18**

1. Introduction

Any sufficiently large software project with enough people making changes to the same code base has a control problem: "How to make sure that everything keeps working after I push my great new thing?". This RFC deals with this problem from the point of view of providing a mechanism to be able to test whether things which worked in the past still work when a new feature is added to the codebase.

This problem is basically a problem of developer mindset. Every developer needs to really want things to be top-notch for any system to work. The solution presented is no magic bullet which keeps people from doing the wrong thing. What it does provide is a method to be able to check early, and check often whether anything will break when introducing a new code-snippet.

The solution proposed borrows ideas from eXtreme Programming as a foundation for the solution, meaning one writes tests for the new feature before writing the new feature itself and the feature isn't finished untill all existing tests and the new tests run successfully.

The remainder of this RFC follows a rather classic line of describing the requirements to make up a certain design, implemented in a certain way predicting the integration issues while gathering unknowns and postponed items in a remaining issues section.

2. Requirements

We've limited ourselves to specifying at most 10 requirements. A testing framework can be very complex if desired. By limiting ourselves to the ten highest priority requirements we ensure a reasonable implementation time for the system. The listed requirements are not sorted, all 10 requirements are of equal importance.

1. *Tests are conservative, the default result being "failed"*

With the implementation of automated systems in general the most important thing to watch out for is a false sense of security. The fact that an automated system is available doesn't, as such, improve the quality of the code produced. For example it's pretty easy to write useless tests which pass. The writing of tests is so important that a complete section of this document is devoted to it.

The above actually means: it will be hard to make tests pass; (under the assumption that the tests are well written and reflect the intent of the developer as to what the code should do) if a test passes it will be a reliable measure that the code actually does what it was designed for.

2. *The test system can test itself*

To prevent having a test system which fails rather then the code being tested, the system should be as simple as possible. Ideally the system should be bug free, thus giving a reasonable guarantee that when a test fails it's the tested code which needs reviewing, not the test system itself. While this is a good requirement, it's not quantitatively measurable. It's better to have a different perspective on this: "make the test system test itself". These tests are a precondition before the test system can be used. All self-tests should "pass" before the system can be used.

3. *Tests are as close as possible to the code for which they are designed*

Adopting a unit test system *will* create extra work for the developer, but it also yields an extra gain in terms of the quality of their code. The tests should be as close to the code tested as possible, to make both the developer's life as easy as possible and to be able to check the tests with the code and maintain them in the same place.

Ideally there should be native support in the programming language for integrated testing. See EIF [1] for an example of how implicit testing can be built into a programming language . (In this case: design by contract). For PHP this is not (yet?) available as native language constructs so we'll have to come up with something for ourselves.

4. *Tests are grouped into testcases in testsuites, which can both be run separately*

The perceived method for using the test system is that the tests are written before the code itself is written and the developer continues to write the code until all tests for that unit pass. This means that it must be easy to run those specific tests repeatedly without running other registered tests. For this we define a concept of "testcases" (checks one specific piece of functionality) and "testsuites" (groups testcases which belong logically together)

5. *Tests can be run directly from the development environment*

While developing code a developer must be able to run the written tests for that piece of functionality. This means that the testruns must be highly integrated with the development environment.

6. *Tests are distributed*

To increase the use of the system the tests should be able to run in each developer's environment, and be replicated to other developers' environments. This allows tests to run in all used environments and to be run more often.

To anticipate inter-unit testcases the testsystem is distributed across all developers. Assuming we have a central place which contains a consolidated version of the codebase (for example the repository on our server) the testcases must also be able to run remotely, or at least the results must be remotely accessible (for example by accessing a web page).

7. *Tests are run before a commit into the repository*

Before committing anything the system should give the developer a warning when one or more test do not pass. In "strict" mode the commit could even be prevented. This setting can for example be applied to a "stable" tree, while a "development" tree might have a more relaxed setting, allowing commits even when tests fail.

8. *A testrun has a unique "point in time" marker*

As the time of running the testcases and the time of presenting the consolidated results may differ, a "marker" of some sort must be attached to the results of each testrun. This means a developer can easily check at which point in time a certain test started to fail and reproduce that situation

In our case this "marker" would probably be a changeset revision number, as this presents exactly one state of the codebase at a certain point in time and allows for creating that specific state at any time.

9. *The testing framework has no impact on runtime requirements*

While using the software, the availability of a testing framework should have no impact at runtime. Although a minimal influence is allowed and expected, no extra requirements for software or hardware can be imposed by the mere fact the testing framework is there. When this influence is necessary, the system must provide a switch to disable it during runtime. It is however allowed that the testing system has an impact during runtime when it is actively being used.

10. *Unattended operation*

The system must actively support a mode in which tests can run unattended, logging the results in a file, database or another storage vault for later examination.

The rationale behind this is that a test may pass when it is run once, twice or multiple times and it will only fail at the 77th run. By allowing unattended operation these bugs can also be caught by the system.

In addition to logging the results a possibility for active signaling must also be available, for example posting an email message saying that a certain test has failed.

The unattended operation must be controllable by a number of configuration parameters specifying how to run the tests and how to create the circumstances to be able to start the testruns (for example creating the codebase at the time of the changeset revision number).

3. Design

Based on the 10 requirements in the first section:

1. Tests are conservative, the result defaulting to "failed"
2. The test system can test itself
3. Tests are as close as possible to the code for which they are designed
4. Tests are grouped into testcases and testsuites, which can both be run separately
5. Tests can be run directly from the development environment
6. Tests are distributed
7. Tests are run before a commit into the repository
8. A testrun has a unique "point in time" marker
9. The testing framework has no impact on runtime requirements
10. Unattended operation

This section will deal with the design implications for all those criteria.

3.1 General design

Instead of reinventing a whole new philosophy for a testing framework, we will basically implement the system according to the jUnit test framework [2].

The test writer will be able to write a special sort of "program" which is targeted specifically for testing. Envisioned is the instantation of a special "testCase" class which holds methods for setting up the test environment, evaluating the precondition(s), running the tests and cleaning up afterwards. After writing the testcases, the writer will be able to group the testcases into testsuites, which logically group tests together. These two activities are the extra work the developer is supposed to do.

Given the written testSuites and testCases a "runner" program can be executed to actually run the tests over the current state of the codebase. The runner executes the tests and reports the result in a specified format. This report will contain the names of the suites and cases and will report passes and failures in a convenient way, so the developer can quickly check whether the tests pass or fail.

For using the test system in a "development mode" a simple text format report will be used. The automated environment will most likely present the results in a web page and store those in a predefined place for later inspection. Other report formats (for example in a dedicated XML format) are planned but won't be available in the first implementation.

3.2 Requirements specific design

This section will focus on the specific design consequences directly related to each separate requirement. This section details the overview given in the previous section.

Tests are conservative, the result defaulting to "failed"

This means that tests will only pass when a very specific test condition has been met, in all other cases the test will fail (regardless of the cause, which could actually also be a fault in the test system itself). In other words, tests will only pass if we've made sure that the exact testcondition evaluates to true, in all other cases tests will fail.

In the preliminary design, we've taken into account that it might be convenient to have a "tri-state" result. PASS, FAIL, EXCEPTION, the latter being that the test didn't even start for some strange reason. For practical purposes exceptions should be viewed as failures nonetheless.

The test system can test itself

This is kind of a tricky requirement. If worked out in full it creates an endless loop of tests. We are going to make an effort though to use the mechanisms described in this RFC for the unit testing code itself.

The results of that 'special' testing should be interpreted a bit differently than normal tests. While in normal tests we assume that the test system is running perfectly, in the tests for the test system itself both the test code as well as the code tested use the same functions. If a test fails this can both mean that the test code or the code tested is broken. In any case there is something wrong which needs to be fixed.

Tests are as close as possible to the code for which they are designed

For storing the tests we have roughly the following choices, assuming we store the tests in the repository.

1. fixed place in the repository (for example <workdir-root>/tests/*) in the repository
2. a subdirectory below the directory which contains the original code (for example /includes/xartests/* files in the includes directory)
3. directly in the code
4. for each file there is a file which contains the test for the code in that file

The first option would be convenient for test writers, but it will be hard to see which tests belong to which part of the code. The second option satisfies the "closeness" criterium better, but will scatter the tests all over the repository, albeit in a predefined place for each directory. Embedding tests directly into the code would be the ideal situation theoretically, but frankly, we don't have a clue how to do that in PHP. Someday maybe PHP will include an *ensure()* operation which allows tests to be embedded into the code. The last option is somewhat like the second with the advantage that it's very clear for which file the tests are written. This advantage is not so great that we are prepared to double the number of files and have no predefined place for the tests.

Weighing the above, we chose option number 2.

Tests are grouped into testcases and testsuites, which can both be run separately

A testcase tests a number of features for a piece of functionality, typically one class or one function. The testcase may consist of a number of tests which logically belong into one group.

By default all testcases fall under the testsuite: "default". It is however possible to create new testsuites and assign new testcases to these testsuites.

Tests can be run directly from the development environment

Effectively this means that a bk command is defined to run the tests. The proposed syntax for this command is:

```
            bk [-r] tests [-s<suitelist>] [-c<caselist>] [-o[html|text]] <dirs>

            Description:
            The 'bk tests' command runs the registered unittests for a certain
            part of code. By default it looks for testsuites and testcases from
            the current directory downwards in directories named 'xartests'.
            If -r is specified the command runs from the root of the repository
            and traverses all directories for xartests directories to look for
tests.

            Options:
            -s     : only run the tests registered in the list of testsuites
specified
                     The <suitelist> is a comma separated list of suitenames.
                     Default: run all testsuites.
            -c     : only run the testcases specified in the list on the command
line.
                     The actual list of testcases is determined as follows:
                     - if the -s option is not specified only the specified
testcases
                       are run (for all testsuites)
                     - if the -s option is specified the testcases are first
```

```
filtered
                       by the specified testsuites. From that list only the
specified
                       cases are executed.
                     Default: run all testcases
            -o     : specify the output format to be generated
                     - html : generate html
                     - text : generate plain text
                     Default: text
            <dirs> : Process the specified directories.
                     Default: current directory
```

Tests are distributed

This requirement should be interpreted such that the testcode is distributed, allowing it to run everywhere the original code can. In our current infrastructure this translates to the tests being registered and maintained within the repository. We make a habit of maintaining all sources in bk repositories so this shouldn't be a problem.

Tests are run before a commit into the repository

Usually a version control system support something like triggers which can run at certain defined events using the repository. We want to run the tests which are related to the changes being committed. The events which we could use are (BitKeeper examples, other VCS-es use others):

- pre-commit: called before a changeset is committed
- pre-delta : called before a change to a file is registered

The difference between these two is that in the first case the changes have already been committed to a changeset, leading to a changeset for which the tests might fail, without the direct possibility to undo it. So, at first glance we need the pre-delta trigger.

There are a couple of issues which need further investigation:

1. Deltas are very numerous, so tests will be run often. This is a good thing, but might be a performance killer. Also, not all tests are suitable to run at the delta level because they involve changes from multiple files. In theory this shouldn't matter because tests are written before the code is written ;-) and just keep failing until the code is right.
2. If we run tests before or while committing code, how do we know whether a test is committed or a piece of code is committed. Does it matter?
3. When pulling changes from a remote repository new changeset(s) are created when doing conflict resolving. This leads to two questions:
    1. does the trigger run in this case, or do both triggers not run in the RESYNC dir. What if the RESYNC dir contains changes to the triggers?
    2. do we want them to run while merging?, it is the critical point and tests are more likely to fail during merges than during normal development so they are more valuable at that point.
4. When we run tests at the "delta-boundaries" instead of the "changeset-boundaries" we might have two conflicting requirements (this one and the unique point in time marker)

Obviously the above is a bit complex, especially if the complex details of the version control system come into play. The current plan is to move forward slowly. Implement the basic framework, use it, get feedback and decide whether further integration, or more strict integration, is warranted.

A testrun has a unique "point in time" marker

By specifying a unique point in time marker on each testresult produced testresults can be linked to a certain state of the codebase. The options we have for this marker are:

1. the latest changeset revision number (TOT Top Of Trunk) in the repository
2. a timestamp

The latest changeset revision number is nice, in that we directly have the information to restore a repository to a certain state to run the tests in, if we want. The disadvantage is that it is bitkeeper specific and might not be

supported in other repository systems. We chose to use the bitkeeper specific implementation first and optionally generalize later on. (ADDITION: we switched to monotone and this RFC should actually be rewritten. Some parts have already been made more generally applicable, but a total review is warranted)

The testing framework has no impact on runtime requirements

The current idea of implementation implies that the test framework *pulls in* the necessary code which needs to be tested. As such it can be seen as a separate system in the code base. No extra code will be added to the actual product. The first estimate for the runtime impact is therefore: zero!

Unattended operation

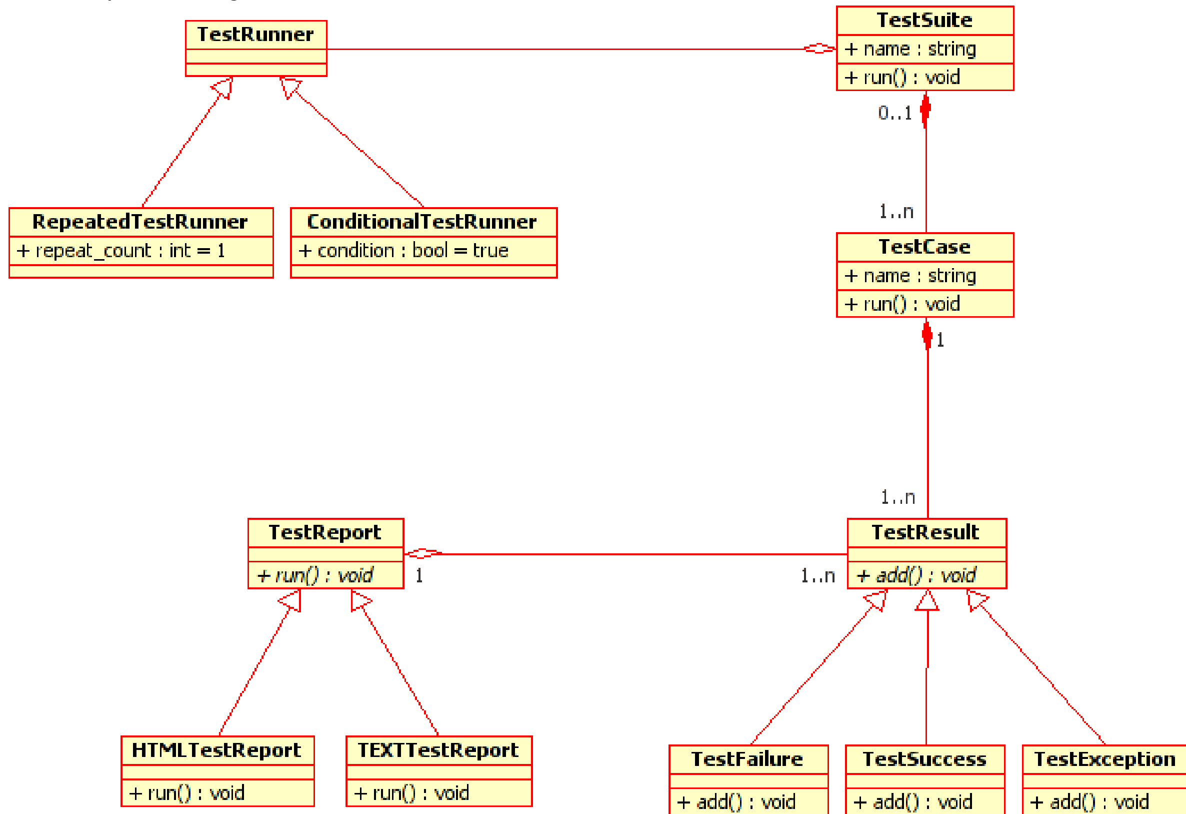
The implemented system must support unattended operation. This means that the tests must be able to run from an automated task scheduler (like cron for example) and allow fairly failsafe operation in that mode. In particular, attention should be paid to:

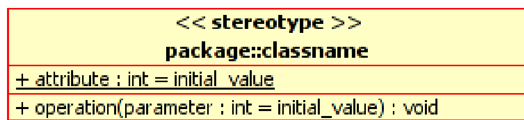
- avoiding interactivity as much as possible
- allowing tests to be skipped, and be marked as such if a certain precondition is not met
- some sort of monitoring, possibly provided by the operating system, that the test system doesn't lock up

The main reason we want unattended operation is that we are able to run tests indefinitely and also catch "repeat causes error"-type of bugs. A test might run 1,2, 3 or even 100 times. The 101st time the test might fail. Running the tests manually will hardly ever catch those type of bugs. Automated running of tests will.

4. Implementation

Preliminary class design:





5. Integration

Describe what the consequences are in starting to use the system. At first glance no integration issues, only some requirements for being able to run it (bk, php, sh, bash that sort of stuff)

6. How to write tests

rather verbose description of a testfile, including code snippets, also a section on what to test for, maybe cite from document from jan

7. Remaining issues

8. Revision history

2002-03-17: MrB : Requirements basically complete

2002-12-28: MrB : initial revision, just a reservation for the RFC

2003-03-28: Miko: typos, grammar, readability

9 References

[1] Eiffel Software, "Design by contract and assertions in the Eiffel programming language", 2003.

[2] junit.org, "The jUnit test framework", 2003.

Authors' Addresses

Marcel van der Boom
Xaraya Development Group
EMail: marcel@hsdev.com
URI: http://www.xaraya.com

Jan Schrage
Xaraya Development Group
EMail: jan@xaraya.com
URI: http://www.xaraya.com

Frank Besler
Xaraya Development Group
EMail: besfred@xaraya.com

A. Glossary

expected result: a defined outcome of a test. (note that it can be possible to expect negative results from a test. The result fails, the test passes

failed: The result of a test does not conform to the expected result

passed: The result of a test does conform to the expected result

Intellectual Property Statement

The DDF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the DDF's procedures with respect to rights in standards-track and standards-related documentation can be found in RFC-0.

The DDF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the DDF Board of Directors.

Acknowledgement

Funding for the RFC Editor function is provided by the DDF