Electronic Communications of the EASST
Volume 33 (2010)

Proceedings of the
Fourth International Workshop on
Foundations and Techniques for
Open Source Software Certification
(OpenCert 2010)

Testing as a Certification Approach

Alberto Simões, Nuno Carvalho and José João Almeida

10 pages

Guest Editors: Luis S. Barbosa, Antonio Cerone, Siraj A. Shaikh
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Testing as a Certification Approach

Alberto Simões[1], **Nuno Carvalho**[2] **and José João Almeida**[3]

[1] ambs@cpan.org, http://www.eseig.ipp.pt/
Escola Superior de Estudos Industriais e de Gestão, Instituto Politécnico do Porto
[2]smash@cpan.org, [3]jj@di.uminho.pt, http://www.di.uminho.pt/
Departamento de Informática, Universidade do Minho

Abstract: For years, one of the main reasons to buy commercial software instead of adopting open-source applications was the, supposed, guarantee of quality. Unfortunately that was rarely true and, fortunately, open-source projects soon adopted some good practices in their code development that lead to better tested software and therefore higher quality products.

In this article we provide a guided tour of some of the best practices that have been implemented in the Perl community in the recent years, as the pathway to a better community-oriented repository of modules, with automatic distributed testing in different platforms and architectures, and with automatic quality measures calculation.

Keywords: test-driven development, test-coverage, distributed testing, Perl community

1 Introduction

Test-driven development [Max03] is not a new approach on the now widely discussed Extreme Programming Techniques [Bec99]. The idea is simple and effective: before writing code, or even thinking on how it will be implemented, the programmer is invited to analyze how he would like to use the application (or the function or methods being developed), and look at it, as often young scholars do, as a little black box, and decide what gets in and what should get out.

After this first discussion, some tests should be written. These tests will use the application's functions or methods being developed, invoking them with some kind of input, and checking its output against some kind of gold standard. This is also a great opportunity for developers to analyze the code API, if it should have one, because since there is not any code actually written yet, the signature of operations made available by the API can easily change.

Only after a few tests are written the developer should start the implementation. This also gives a chance for the developer to meditate about the expected behavior of the new code without any concerns about implementation details. The behavior should always be chosen outside the scope of implementation, since the expected behavior of a function, or method, should not be tailored by implementation difficulties. This is true for most of the cases, but not always. As soon as a first running code is available, it should be run against the written tests. If any test fails, it means the algorithm is not working properly and, if the test passes, it means the code is supporting the cases described in the tests.

This process iterates. During development it is natural that the developer thinks of some new situation that should be handled. Before coding that portion of code, he should write a new test that tests that specific case.

What test-driven development guarantees is that new code will not break previously working code, as the test suite grows. It does not guarantee that the code handles all situations, as it depends on the written tests and their coverage. Also, it does not guarantee that the developer did not cheat, as he is aware of the input of each test. As any other technique, it depends on the good will of the involved persons.

Another major advantage of using testing frameworks is the ability to easily re-factory. In today's development environments often happens that different teams put together smaller programs to be used by other teams to build more complex applications. If at any given time one of these smaller programs needs to be re-implemented, because of efficiency problems for example, the developer, being the same or a new one, just needs to make sure that the new implementation passes all the tests. This, in most cases, automatically makes complex programs, that use the re-factored code, immediately also work. This scales very well, meaning that if the complex program also passes its own test suite, because the re-factored program respected the old code behavior, then even more complex programs that rely on both of these will probably automatically work with the new code.

The Open-Source community is investing in this approach for code development. Examples are the unit-testing of Java, Ruby or Perl modules, and the number of available frameworks for testing. Even in the corporate world more and more often companies release their applications as open-source projects and many times rely on testing frameworks to make sure their code is not only working, since there is the chance of many more contributions and changes to the original code, but also guarantee that the most recent code still maintains the original behavior.

In this article we would like to give a tour of the initiatives and techniques that are being used by the Perl community to guarantee some minimum quality standard on the modules made available by the well known Comprehensive Perl Archive Network [CPA10] (CPAN).

The article is divided in four main sections. First, section 2, presents briefly the CPAN archive, how it works and the available tools for the common Perl programmer to interact with it. Section 3 covers some of the available frameworks for writing tests for Perl Modules. These frameworks will be divided in three blocks: testing code behavior, testing documentation (both syntax and coverage) and testing module distribution. Trying to overcome the usual problem of discussing who certifies certification agencies, or who controls persons responsible for controlling others, section 4 presents an approach to testing tests using the code that was written to pass those tests. Finally, section 5 will focus on two community initiatives: the support for distributed testing on different architectures and operating systems, and the analysis of modules' code with the computation of a quality measure.

In summary, in this article we will present approaches that will help the developer to tell the user *here is my code* and *here is a way to show you that it works properly*. The certification itself is basically given by a positive outcome of the testing framework, obviously assuming that the tests themselves are trusted and were written in good faith.

2 Quick Introduction to CPAN

The Comprehensive Perl Archive Network (CPAN) has its origins (in concept and name) in the Comprehensive TEX Archive Network [CTA10] (CTAN), the archive of TEX classes and packages. The CTAN idea is simple: create a centralized archive of modules, scripts and other tools related to a community of users, where any user can contribute, and the entire community can make use of the entire archive. This same approach is being used by other communities, like R developers with the Comprehensive R Archive Network [CRA10] (CRAN), Ruby Application Archive [RAA10] (RAA) for Ruby developers, or Python Package Index [PyP10] (PyPI) for Python developers.

Every one of these archives shares the same basic principles, but adds different functionalities according to their user's needs. These archives' baseline of functionalities can be described as:

1. any user can contribute any code/package/module (and, normally, the contribution is not reviewed);
2. there are no restrictions on adding contributions that mimic the behavior of other contributions;
3. there is some kind of taxonomy that allows the contributor to classify his contribution (usually contributions are indexed also by contributor name);
4. any user can search the catalog and download any package he wants. Usually these archives also add some text explaining the package so that the user can choose using something more elaborate than just the package name, author information and contribution class.

It is easy to notice that these operating rules are too liberal. In particular, rules 1 and 2 lead to anarchy very easily, as users can contribute bad, buggy or malicious code, and can even contribute code with similar behavior of other already archived. Therefore, the user searching for a module will have to deal with the questions: how to be sure that a module can be downloaded and used safely; and how to choose from a set of possible modules that can be used to perform the same task.

Unfortunately, unless the rules get replaced by new rigid ones, these two problems do not have a simple solution. Of course this contribution flexibility motivates and promotes more developers to make their code available in the archive. Nevertheless, some extra meta-information can be added to the repository, making the task of choosing what modules to download, and use, easier for the user.

With this objective, CPAN includes a few extra meta-information mechanisms:

1. each module can be rated, as if it were a movie, by any user. The user can add comments on it as well. Unfortunately it is not easy to convince users to rate modules. While some perform that task, most CPAN modules are not rated or commented on;
2. each module has a clearly associated author, with e-mail address and picture (when available). As authors get well known and get reputation, users get confident in using their contributions. This is especially true given the number of conferences organized each year by the community, that work well to introduce developers;
3. together with the module description it is possible to visit, automatically, its documentation. Also, as the community suggests a well structured template for documentation, it is relatively easy to compare modules' documentation and their completeness (therefore, making it easier to choose which module to use);
4. the date of the last update is also shown. Usually modules with old dates are not maintained. But it can also mean the module is stable (although this is rarely the case);
5. a detailed matrix of the tests and their status (pass/fail) on different platforms is also shown. Refer to section 5.1 for more information on how this data is computed.

Although not related to tests and software quality, we would like to add a final remark: there are some applications that can be used to install modules from CPAN. The fact that more than one tool exists for this task is an example of the multiplicity of available modules for performing the same task. When installation fails the user has the option to automatically report the failure. This will issue an e-mail that will be sent to the module(s) maintainer(s).

3 Pure Test-Driven Development

Every Perl module available on CPAN includes a test suite (of course, there are a couple of exceptions that prove the rule), from simple and incomplete to fully featured test suites.

These test suites' appearance was not guided by any rule or requirement imposed by CPAN. That would not work! Instead, an initial framework for testing was born and, at that time, the only tool available to bootstrap empty Perl modules from a skeleton template incorporated one or two simple tests of module usability (for instance, checking the module loads).

It was this initiative that resulted in a greater number of people writing tests, not because they were a requirement. Nobody really cares to complain if a module does not include a test suite, but the author, when creating the module, and noticing there is already a basic framework for writing tests, tries to maintain it, adding new tests.

Three different aspects of Perl modules began to be tested (there are some other aspects that could be included here but that we decided to ignore them, as they can be considered part of one of the categories we present here):

- tests started with simple *code testing* (section 3.1), just like any other programming language unit-testing framework;
- then followed the addition of *tests for documentation* and documentation coverage (section 3.2), checking the syntax of the documentation and its completeness;
- finally, tests for *checking distribution contents* (section 3.3) are arising, to ensure every file required is being shipped in the module tarball.

These same tests can be divided in two categories:

- **developer tests** should be checked only by the programmer, locally, before distribution. They normally check that all files are present in the distribution, that the documentation is complete and with the correct syntax;
- **user tests** should be shipped with the module and should be run by every user that wants to install the module. They usually test the algorithm of the application. These tests will guarantee that the relevant code works independently of the architecture, operating system or Perl version, as developers might have some difficulty in having different machines for testing purposes (check section 5.1 for more initiatives on multi-architecture testing).

Note that while we focus primarily the testing frameworks for Perl module development, the Perl core itself has a complete test suite.

3.1 Testing Code

Testing code is the more usual paradigm of testing. As described in the introduction of this article, the developer is invited to declare the behavior of its method or applications, writing a set of typical inputs (hopefully including some edge cases), and the corresponding correct results (outputs).

Depending on the complexity of the method or application being tested, the complexity of the test can grow. A common good practice is to start writing tests for small auxiliary functions at the beginning of the project, in such a way that every new function has all its dependencies well tested.

The more usual tests for code can be divided in the following categories [Lc05]:

- **Comparing the return value with the correct answer:**
  Most tests receive an input and check the output against a gold standard, the correct answer. This verification can be as simple as checking if the return value is the same as a specific integer or string, or checking if the return value is inside the expected range of possible answers.

  Perl frameworks implement a set of functions to help implement these tests. Each test includes the code to be tested but also a small description of what is being tested. This is useful, as it makes the process of reading test reports easier.

  ```
  is( add(2,3) , 5 , 'Simple test for add' )
  ```

  Modern test frameworks provide more flexible testing mechanisms, so that strings can be matched against regular expressions, or full complex data structures matched against other sample structures.

  ```
  is_deeply( parse('2+3') , ['+', 2, 3] , 'Parse sum op' )
  ```

  Also, there are other modules that allow checking for other kind of output, like text document generation, analyzing XML structures (matching it against a schema or simply analyzing the contents of some XPath expressions), or checking the values present on a database.

  ```
  my $snoopy = Dog->new("Snoopy");
  isa_ok( $snoopy , 'Animal');    # Snoopy is an animal
  can_ok( $snoopy , 'bark');      # Snoopy is able to bark
  ```

  All these tests are of the same kind: with some input, the function, method or application delivers the correct output.

- **Checking that the module is loadable without errors:**
  A fundamental test for any program written in any language is that it compiles or gets interpreted correctly without syntax errors. This test is automatically generated for any new module created by the common Perl module generators, ensuring that each new module that gets in CPAN ships with this basic guarantee.

- **Analyzing an objects' hierarchy and available methods:**
  Object oriented programs can create classes and objects at run time. These classes need to be tested, for instance, checking their parent information (*isa* relationship) and checking that they can handle some specific methods.

More complicated testing mechanisms are also supported in Perl. For instance, there are modules for testing regular expressions (checking that they match the required string and that they will not match false positives), XML (that the document is well formed, or valid against a specific schema), XPath expressions (that the expressions are correct and that they yield the correct value when matched against an XML document), images (checking their size, checking specific pixel colors, etc), web applications (simulating an user, interacting with the web application and analyzing the resulting web pages) and many more.

Last, but not least, testing of coding standards (or best practices [Con05]), such as indentation, or function or variable name capitalization, is also contemplated.

3.2 Testing Documentation

Perl has a great advantage with documentation over some other languages. While Java or C support JavaDoc [Ora10] and Doxygen [vH10] respectively, they were never seen as a real standard for writing documentation (probably more with JavaDoc than Doxygen), Perl has a *de facto* standard, named POD[1], that is broadly used by all Perl modules.

It can be used in a literate programming approach, where the programmer can interleave code with documentation. Unlike JavaDoc or Doxygen, Perl is very flexible on the POD usage. There is no requirement to write the documentation near the respective functions, for example (JavaDoc or Doxygen work as code annotation).

POD has a simple textual format. It supports a few headings, some lists, basic word highlighting, and verbatim sections.

This documentation should also be tested and, on newly created modules, two kinds of tests are automatically created:

- **Checking documentation syntax correctness:**
  Given that Perl uses a specific syntax for writing documentation, and that that documentation is interpreted to generate the documentation in different formats (Unix man-page, HTML, PDF, LaTeX), it is important that the documentation syntax is correct. For that purpose, a syntax checker exists that is able to search for all documentation present on a Perl module directory and complain about syntax errors.

- **Checking documentation coverage:**
  The second level of quality assurance for documentation is its coverage. It is not enough that the documentation has a valid syntax, it is also required to cover all methods implemented. This framework parses the documentation and ensures that each function or method defined has a corresponding documentation section. As some methods might be irrelevant for documentation (maybe because you just do not want to make users aware of it), their names can be prefixed with an underscore and the testing mechanism will ignore them.

Once more, these testing approaches do not guarantee any documentation quality, but they assist the developer who is interested in writing and maintaining complete documentation.

3.3 Testing Distribution

When creating a tarball with the module files and uploading it to CPAN servers, the developer needs to ensure the tarball is complete, and that all files are edited accordingly. In this area, the Perl community also offers some frameworks for testing purposes:

- **Checking distribution tarball completeness:**
  When developing a program or module, there are a bunch of files that are created with small debug programs, small test cases and other information (such as version control software files). These files are not part of the distribution that should be released. Therefore, Perl adopted the concept of a manifest file with the list of files to be included in the release tarball.

  While this solution is great, it is also annoying. Every time a new file that should be included in the distribution is created the developer needs to edit the manifest file and add the new file. If he forgets to do so, the distribution will be incomplete and unusable. Therefore, a mechanism to ensure that all files that are listed in the manifest file exist is required. For that to work, this test uses another manifest file, with regular expressions that match the files that should be ignored and not included in the distribution. Then, if a file appears that is not listed in any of the two manifest files, the test will fail.

[1] Stands for Plain Old Documentation (whilst old, it is not dead, and has been evolving in the last year).

- **Checking that generated files were properly edited:**
  Another kind of test that should be performed prior to the module distribution is ensuring that all files that were generated by the common module generation tools were edited. To explain the relevance of this test I should explain that about 8 years ago, many modules in CPAN had as author “A. U. Thor”, the name used by one of the modules generation tools.

  These tests, named *boilerplate*, ensure the programmer edited the generated code, installation and other documentation files.

Other examples of distribution testing include the analysis of modules and sub-modules, change log and read me files, ensuring all refer to the same version.

4 Testing Test Coverage

The main problem when writing tests is the question about how to test the quality of the tests. In fact, testing tests it not really possible. But we can assess how much of our code is covered by currently written tests.

Perl offers a framework for this purpose. For each test, each line of code that gets executed it counted. This results in a table that, for each line of code, shows the number of times it was executed. This kind of coverage testing is great when writing tests. If the written code has some conditional structure, for example, there should be a test to exercise each of the possible branches [Joh05]. All this information is presented to the user in HTML format, with all the code annotated with information about how many times each line was executed. Moreover, it also presents some basic statistics, showing the percentage of subroutines or branches that have been tested.

With all this information it becomes easier for the programmer to find out what areas of the code need extra testing.

5 Automatic Distributed Testing

As already stated, some developers do not have access to all platforms (CPU, architectures or operating systems) where Perl can run. This leads to a problem: how can you know if a specific module works correctly on a specific platform? To overcome this problem the Perl community created a distributed testing service (see section 5.1).

All these initiatives (those already discussed and this distributed testing service) assume the good will of the developer, who wants to make his code better. As an independent initiative, another project named CPANTS [Kla10] (The CPAN Testing Service) was created. The goal of this project is to provide some sort of quality measure (called Kwalittee) by code analysis (see section 5.2).

5.1 Distributed Testing Service

The Perl community has a CPAN Testers framework [Bar10]. Community volunteers can join the initiative, registering one or more machines (together with its architecture, operating system and Perl version) and offering to test module distributions uploaded to the archive. There are some tools to help in finding the latest uploaded modules so that CPAN Testers can know what to test. This process can be completely automatic or semi-manual, depending on the testing tool chosen by the tester.

Currently there are testers running Perl versions from 5.004_05 (the maintenance branch of a Perl version with more than ten years old) to the most recent, development branch 5.13.2 (about two weeks old). Operating systems available for testing include OS/2, SunOS/Solaris, IRIX, Mac OS X, OpenBSD, VMS, Windows, Linux, AIX, etc[2].

If the configuration, compilation and installation succeeds, a success report is inserted in a database that can be queried by any user (therefore, knowing if that module is stable for a specific platform). If some error occurs, an e-mail with a full report is generated (with the full output of the compilation process, and details on the platform and Perl configuration variables) and sent to the module maintainers. This same report is stored in the database, so that any user can query it.

5.2 Automatic Quality Measuring by Code Analysis

Being a CPAN Tester is a risky task. While many CPAN Testers test modules in a virtual machine or some kind of sand box, they are risking their machine or installation to malicious code. As far as the authors are aware, no real malicious code was found yet on CPAN, but it is possible (although, if detected, user would be banned and modules deleted).

CPANTS is another project that aims to evaluate Perl modules. Instead of trying to compile, test and install modules, this approach grabs modules and inspects their code (not executing it).

The module code is checked against a list of Kwalitee[3] metrics. Unfortunately, as the basic idea rejects the interpretation of code, the amount of analysis possible to be performed is reduced.

Nevertheless, CPANTS tests are relevant. To mention some examples, CPANTS checks if all module dependencies are listed correctly in the package meta-data file, if any file mentioned in the manifest file is missing, if every module file has a version number, if there is a clear license in the documentation, if a read me and a change log files are present, etc.

Given the automatic behavior of CPANTS, and its objectiveness, it can be almost considered a game, where modules with better module distributions get points for their kwalitee. Therefore, the web site can show a sorted list of authors, that can *play* or *fight*, trying to climb up the table. This playful approach can motivate developers to improve their distributions.

6 Conclusions

In this article we provided a brief tour to the mechanisms implemented by the Perl community to help the development of test suites. Helping to guarantee that modules include tests, delivering methods to harness results of running test suites, measuring testing coverage of the source code, and also trying to make sure that modules run correctly on heterogeneous systems by providing distributed approaches to test code in different contexts, architectures and platforms.

While the testing techniques described in this article can not be seen as a formal certification approach, they can easily motivate open-source developers to include quality assurance tests. Also, a test suite can be used to help demonstrate the end user that the code (implemented by the developer or developers) actually does what it advertises, without the need for the user to browse thousands of lines of source code. Assuming the good faith of the tests writers, we can admit that a positive outcome of running the test suite certifies that the module is working properly, at least for the cases tested. And luckily, enough edge cases and gray areas tests were included to certify that the module or application is working as expected for those particular cases too. This can be seen as a way for

[2] Unfortunately not all platforms have all Perl versions available for testing. Nevertheless, more common operating systems have most Perl versions available. You can check a detailed matrix of what Perl version are available in what Operating System at http://stats.cpantesters.org/osmatrix-full.html

[3] Kwalitee is the name chosen to represent this pseudo-quality information.

a developer, by himself, without the need of any outside entity, to give some assurance (a small non formal certification) that the code written works as expected.

From these different initiatives we would like to stress that it is important that every kind of software packaging approach includes mechanisms to introduce software tests and that, when they are created by some kind of automatic generator tool, some simple tests are automatically generated. This will motivate the developer to keep the test suite up-to-date. On the other hand, if the task of adding a testing framework is a developer task, it is natural that this framework will often never be used. Moreover, if the testing framework includes any tool to test the tests' coverage, it will help interested programmers in detecting what sections of code are lacking testing. Modern frameworks should also include tests for doing some documentation validation, namely documentation syntax and coverage, this can help the developer to improve the source code documentation. This introduces another interesting, and positive, side effect of writing a more complete test suite which is to have more code examples. And as tests cover more and more features of the code, more examples the author has, that can be used to enhance the module documentation.

Finally, knowing that these test suites will be run in different architectures and platforms automatically, without the need to ask for it, can lead the developers to have a greater interest in writing complete tests.

Clearly there are plenty of advantages on creating and maintaining a test suite, this is so obvious that Perl itself has one. Once you build the Perl interpreter you can run this test suite to validate that the binary files were built correctly, and that Perl behaves as expected. The core modules shipped with the Perl distribution test suites are also executed in the process to make sure that everything works as intended.

All these advantages shared by all contributions can help to promote the use of public code repositories analogous to CPAN. Since individual people, and companies, can clearly state that there is a continuous effort to improve the overall software quality. Deploying better software also improves the level of confidence in this code repositories which helps to promote the language itself, and the adoption of open-source solutions in general.

Acknowledgments

This work was partly sponsored by project grant PTDC/EIA-CCO/108995/2008 (An Infrastructure for Certification and Re-engineering of Open Source Software), from Science and Technology Foundation of Portugal.

An extra acknowledgment to Paul Johnson, author of `Devel::Cover` module (that tests test coverage) for comments and suggestions.

References

[Bar10] Barbie et al. CPAN Testers. 2010. http://www.cpantesters.org/.

[Bec99] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[Con05] D. Conway. *Perl Pest Practices* . O'Reilly, Sebastopol, CA, 1st ed. edition, 2005.

[CPA10] CPAN. Comprehensive Perl Archive Network. 2010. http://www.cpan.org, http://search.cpan.org.

[CRA10] CRAN. Comprehensive R Archive Network. 2010. http://cran.r-project.org/.

[CTA10] CTAN. Comprehensive TEX Archive Network. 2010. http://www.ctan.org.

[vH10] D. van Heesch. Doxygen: Generate documentation from source code. 2010. http://www.stack.nl/~dimitri/doxygen/.

[Joh05] P. Johnson. Devel::Cover – an introduction. In Simões and Castro (eds.), *Yet Another Perl Conference, Europe (YAPC::EU)*. Pp. 85–90. Braga, Portugal, August 2005.

[Kla10] T. Klausner. CPAN Testing Service. 2010. http://cpants.perl.org/.

[Lc05] I. Langworth, chromatic. *Perl Testing: A Developer's Notebook*. O'Reilly, Beijing, 2005.

[Max03] E. M. Maximilien. Assessing Test-Driven Development at IBM. In *In Proceedings of the 25th International Conference on Software Engineering (ICSE-03*. Pp. 564–569. IEEE Computer Society, 2003.

[Ora10] Oracle. Javadoc Tool. 2010. http://java.sun.com/j2se/javadoc/.

[PyP10] PyPI. Python Package Index. 2010. http://pypi.python.org/pypi.

[RAA10] RAA. Ruby Application Archive. 2010. http://raa.ruby-lang.org/.