HAL
open science

Exposing Test Analysis Results with DrTests

Dayne Guerra Calle, Julien Delplanque, Stéphane Ducasse

To cite this version:

Dayne Guerra Calle, Julien Delplanque, Stéphane Ducasse. Exposing Test Analysis Results with DrTests. International Workshop on Smalltalk Technologies, Aug 2019, Cologne, Germany. hal-02404040

HAL Id: hal-02404040

https://inria.hal.science/hal-02404040v1

Submitted on 11 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Exposing Test Analysis Results with DrTests

Dayne Guerra Calle
Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL
France
dayne-lorena.guerra-calle@inria.fr

Julien Delplanque
Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL
France
julien.delplanque@inria.fr

Stéphane Ducasse
Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL
France
stephane.ducasse@inria.fr

Abstract

Tests are getting the cornerstone of continuous development process and software evolution. Tests are the new gold. To improve test quality, a plethora of analyses is proposed such as test smells, mutation testing, test coverage. The problem is that each analysis often needs a particular way to expose its results to the developer. There is a need for an architecture supporting test running and analysis in a modular and extensible way. In this article we present an extensible plugin-based architecture to run and report test results. DrTests is a new test browser that implements such plugin-based architecture. DrTests supports the execution of rotten tests, comments to tests, coverage and profiling tests.

Keywords Unit testing, tests, coverage, analyses, rotten green tests, SUnit

1 Introduction

Testing is an important activity to ensure code quality [Bei90, Bin99]. Unit tests have become an important tenant of software development process [Bec02, BA04, BS07]. Unit Testing is defined as a type of software testing where individual units/components of a software system are tested. The objective of Unit Testing is to isolate a section of code and verify its correctness.

Software development teams grow unit tests as an health insurance to make sure that they are able to evolve their applications [BG98, DDN02]. Pharo is growing regularly its test basis for example with the recent definition of unit tests for UI elements. In industry test bases can be quite large: for example Lifeware reports to have 119,000 tests.

The natural next steps is to ensure that the tests themselves are of good quality [BVR08, BTP+17]. This is why a plethora of analyses have emerged such as test smells [DMBK01, RBD06, RGD07, BQO+12], mutation testing, pseudo-tested methods [VPDMB18], rotten green tests [DAEN18] to name a few.

The problem is that test execution and analyses often require to customize test execution parameters as well as present results in specific ways. For example, the user may want to execute only one column/row of a parametrized matrix or get information about test coverage. This last point is clearly different from just plain test execution. There is a need for a flexible architecture to execute and display test and analysis results.

In this paper we present a plugin-based architecture to support modular and extensible definitions of analyses of tests. DrTests[1] is a new tool implementing such architecture. We show that several analyses can present adequately their results.

The rest of this article is organized as follow. Section 2 starts with an exploration of a sample of test analyses and list problems. Section 3 discusses limitations of the actual UI to handle tests in Pharo. In Section 4, we present an extensible architecture to execute and display test results and test analyses. Section 5 concludes the paper and discuss perspectives.

2 Test analysis result diversity

Tests can be analyzed under multiple aspects. For example, one can run a test and watch its result (passes, fails or raises an error) but it is also possible to analyze the coverage of this test.

Each kind of analysis applied on a test has similar but not totally equal kind of input and can generate various kinds of output. However one would like to threat these analyses uniformly to make the tooling around them smooth. For example, provide a common user interface.

To illustrate this problem, let us review 5 distinct kinds of analysis that are performed on tests: running a test, computing test coverage, profiling the test execution, running executable comments and detecting rotten green tests.

2.1 Running a test

Running a test is the process to execute the test code and reporting its execution. In particular, test runner check whether assertions are violated or not.

In Pharo, the test framework is named SUnit and follows the xUnit frameworks architecture [DDN02]. That is to say, test classes are defined as subclasses of TestCase. This root class contains the assertion primitives.

From the test classes of one or multiple packages, a test suite is built. This suite contains all the test cases to be executed. A test runner runs the test suite and reports execution results.

IWST'19, August 27-29th, 2019, Cologne, Germany
2019.

[1] https://github.com/juliendelplanque/DrTests

The result returned by the test runner contains, for each test, the status of the execution: *pass* means that all assertions in the test were validated, *fail* means that at least one assertion in the test was invalid, *error* which means that an exception occurs during test execution or *skipped* meaning that the whole test or some part of it were not executed.

2.2 Test Coverage

Test coverage is a measure of the degree to which the source code of a program is executed when a particular test suite runs [Bn12].

To execute the coverage we have to identify the test suites and the source code to cover. In Pharo, the test coverage tool allows one to select a single package. However, a tool can propose coverage analysis at finer grain (e.g. coverage of a single class or the abstract syntax tree of a single method). This analysis executes test cases and verifies if they execute the methods inside the package under analysis. The methods that are not executed are reported to the user.

The results provided by this analysis are twofold. First, the percentage of executed methods over the total number of methods inside the package. Second, the list of the uncovered methods.

2.3 Test Profiler

Profiling is a form of dynamic program analysis. It can measures: the space or time complexity of a program, the number of time of particular instructions are executed, or the frequency and duration of function calls.

This analysis executes and benchmarks test cases. Additionally to the test runner process, it keeps track of the time taken by each test.

The results of the test profiler are twofold. First, results of each test run (as described previously in Section 2.1) Second, the duration of the execution for each test case.

2.4 Executable comments

Pharo offers executable examples in comment using the message >>>. This message acts as an assertion: the left side has a piece of code to run and the right side has the expected result. See for example Listing 1.

```
copyReplaceAll: oldSubstring with: newSubstring
  "Answer a copy of the receiver in which all occurrences
   of oldSubstring have been replaced by newSubstring.

  'ab cd ab ef ab' copyReplaceAll: 'ab' with: 'zk'
    >>> 'zk cd zk ef zk'"
  [...]
```

Listing 1. Example of runnable comment in the comment of a method.

The objective of executable comments is to keep documentation synchronized with the implementation. Furthermore, it provides an easy way to understand the API of a method.

Since executable comments are just assertion, they can be handled by test tools. To do that, they are converted to regular test suites and run by the test runner. Thus, while the input of this analysis is different, its results are similar to the one of the test runner.

2.5 Rotten Green Tests

A rotten green test is a test that pass and contain assertions, but whose assertions are not executed [DAEN18]. As stated by Deplanque et. al., such tests are worse than no tests at all because they give developers false confidence in the system under tests.

RottenTestsFinder[2] is a tool revealing rotten green tests in a test suite. The input provided to this plugin is a test suite. The plugin then combines static analysis of tests' source code with code instrumentation to spot call sites to assertion primitive that are not executed. As result, a list of rotten tests is returned to the developer who need to fix them. Additionally, for each rotten test the developer can access rotten call-sites.

3 Actual test runner limitations

Figure 1 shows the test runner. It is the UI currently used in Pharo for executing tests and displaying their results. It allows one to select the packages that contain tests (left list) and some or all test cases they hold (middle list).

Test Runner provides 6 buttons performing predefined actions. The three first starting from the left allow one to run tests, run tests while profiling them and compute coverage.

Results are split across 3 panels on the right of the window: summary result, failed tests and errors. A user can re-run either failed or error tests by clicking on them. The three last buttons are used to re-run all failed tests, re-run all error tests and export the last results created by the test runner into a file.

If we take a look at the integration of coverage computation of test profiler, we can observe the following. Running coverage requires an extra input for the package under analysis and it opens a new window for that. The result of the computation are displayed in yet other window. The test profiler runs the test while watching their execution time and opens a new window for showing the result.

However, with the new analyses available for unit tests, the Test runner does not scale. In fact, the integration of the coverage analysis and the time profiler is already not nice because it relies on additional windows and widgets. As a reminder, Table 1 summarize input and output of the 4 analyses presented previously. From this table, we can observe that to plug additional analyses, the test runner UI needs to be modified. This is problematic because we do not want to change the UI each time a new analysis arise.

[2]https://github.com/juliendelplanque/RottenTestsFinder

Table 1. Summary of the different analyses input/output.

Analysis	Tests Runner	Test Coverage	Tests Profiler	Comment to test	Rotten Tests Finder
Input	Test cases	Test cases + Package under analysis	Test cases	Executable comments	Test cases
Output	Test results by groups (i.e. errors, failures, skipped and passed tests)	Percent of code coverage + List of uncovered methods	Test results by group (i.e. errors, failures, skipped and passed tests) + Time by group and time for each test	Test results by group (i.e. errors, failures, skipped and passed tests)	List of rotten green tests

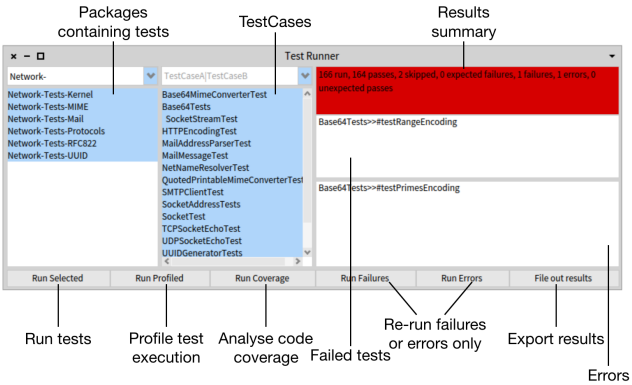

Figure 1. UI of the actual "Test Runner" in Pharo.

4 DrTests Architecture

In this section, we describe DrTests architecture. It is made of 3 main components: configuration, plugin and result. Figure 2 provides a high-level illustration of this architecture.

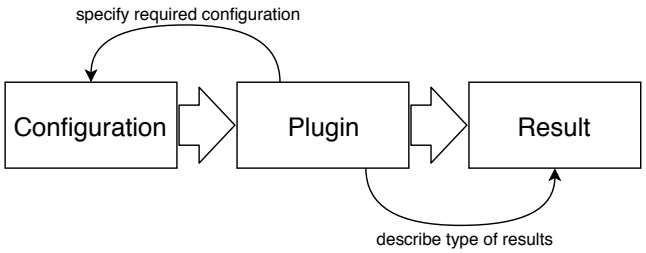

Figure 2. High-level view of DrTests architecture.

4.1 Configuration

The configuration of a plugin contains the required information to run it. In DrTests, it is reified by a class which instances are built by the *plugin* according to developer's preferences. For example, the cases that should be run.

4.2 Plugin

A plugin defines how to execute a specific analysis. For example, running the tests contained in a list of test cases. It is parameterized by the *configuration* it gets as input. During its run, it provides updates to potential observers using announcements. Once the plugin's task is finished, a *result* is returned.

4.3 Result

Finally, a plugin creates a *result*. This object contains all artifacts resulting from the analysis. Each plugin is free to choose the kind of result returned. Because of that, there is no limitation in the kind of data an analysis can return.

4.4 DrTests UI

Figure 3 shows the UI of DrTests. It is the main UI written in Spec 2.0[3] and provides us an interface within we can choose between different plugins.

One can select the plugin to use via the drop-list in the top menu. The UI displays potential inputs for analysis as defined by the currently selected plugin. Users can select a subset of proposed inputs and run the plugin on it. The plugin provides information about event happening during the execution. The logging label at bottom of the UI shows these updates.

Results of the plugin are be displayed using a tree widget. Tree widgets provide a high flexibility when it comes to display various kind of structured data. In case of plugin results, they might be organized hierarchically.

5 Conclusion

In this article, we reviewed the issues of Pharo's current test runner in the context of a growing ecosystem of test analysis. This review led us to the design of DrTest: a plugin-based architecture to deal with tests in Pharo. This architecture allows one to plug any analysis related to tests in Pharo IDE.

[3]https://github.com/pharo-spec/Spec

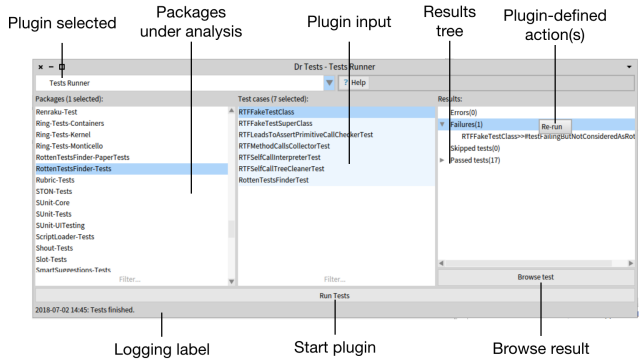

Figure 3. DrtTests UI.

6 Future work

In the future, we will implement more plugins for DrTests. Two possibilities to explore are parameterized tests and mutation testing.

Additionally, on the UI side, we want to allow users to have different views on the result of a plugin. That is to say, allowing plugins to provide multiple ways to build the tree to be shown in the UI.

References

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman, 2002.

[Bei90] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[BG98] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[Bin99] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.

[Bn12] Alexandre Bergel and Vanessa Pe na. Increasing test coverage with hapao. *Science of Computer Programming*, 79(1):86–100, 2012.

[BQO+12] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *International Conference on Software Maintenance (ICSM)*, pages 56–65. IEEE, sep 2012.

[BS07] Cédric Beust and Hani Suleiman. *Next generation Java testing: TestNG and advanced concepts*. Addison-Wesley Professional, 2007.

[BTP+17] David Bowes, Hall Tracy, Jean Petrié, Thomas Shippey, and Burak Turhan. How good are my tests? In *Workshop on Emerging Trends in Software Metrics (WETSoM)*. IEEE/ACM, 2017.

[BVR08] M. Breugelmans and B. Van Rompaey. TestQ: Exploring structural and maintenance characteristics of unit test suites. In *International Workshop on Advanced Software Development Tools and Techniques (WASDeTT)*, 2008.

[DAEN18] Julien Delplanque, Olivier Auverlot, Anne Etien, and Anquetil Nicolas. Définition et identification des tables de nomenclatures. In *36 ème édition d'INFormatique des ORganisations et Systèmes d'Information et de Décision (Inforsid 2018)*, 2018.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[DMBK01] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.

[RBD06] Bart Van Rompaey, Bart Du Bois, and Serge Demeyer. Characterizing the relative significance of a test smell. *icsm*, 0:391–400, 2006.

[RGD07] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based assessment of test quality. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251, October 2007. Special Issue. Proceedings of TOOLS Europe 2007.

[VPDMB18] Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry. A comprehensive study of pseudo-tested methods. *Empirical Software Engineering*, pages 1–33, 2018.