STAMP

Deliverable D1.2

Initial prototype of the unit test amplification tool

Project Number	:	731529
Project Title	:	STAMP
Deliverable Type	:	Report

Deliverable Number	:	D1.2
Title of Deliverable	:	Initial prototype of the unit test amplification tool
Dissemination Level	:	Public
Version	:	1.00
Latest version	:	https://github.com/STAMP-project/docs-forum/blob/master/docs/d12_initial_prototype_unit_test_amplification_tool.pdf
Contractual Delivery Date	:	M12 November, 30 2017
Contributing WPs	:	WP 1
Editor(s)	:	Benoit Baudry, KTH
Author(s)	:	Benoit Baudry, KTH Benjamin Danglot, INRIA Vincent Massol, XWiki Martin Monperrus, KTH Oscar Vera-Perez, INRIA
Reviewer(s)	:	Daniele Gagliardi, Engineering Caroline Landry, INRIA

Abstract

This deliverable reports on the progress towards the construction of a tool-box for unit testing amplification. This tool-box aims both at exploring novel research questions and at providing relevant feedback to developers to improve the quality of industrial open source projects. The core research question we address within workpackage 1 is as follows: Can automatic unit test amplification synthesize actionable hints for developers to improve their test suite?

Chapter 1

Introduction

This deliverable reports on the progress towards the construction of a tool-box for unit testing amplification. This tool-box aims both at exploring novel research questions and at providing relevant feedback to developers to improve the quality of industrial open source projects. The core research question we address within workpackage 1 is as follows:

> Can automatic unit test amplification synthesize actionable hints for developers to improve their test suite?

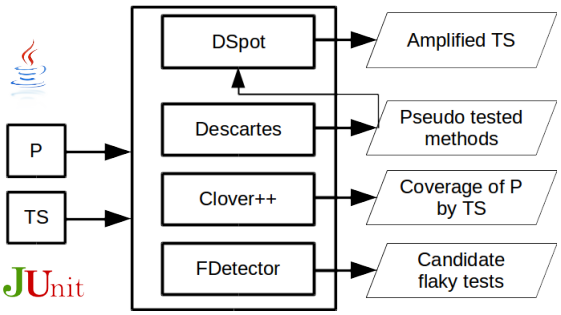


Figure 1.1: Overview of the test amplification tool-box

This deliverable reports on the tool-box that we have built to address this original and hard question. As illustrated in Figure 1.1, we consider a program and a test suite as inputs for the tool chain, and perform a number of analyses, which provide hints in different forms to the developers:

- edits for existing test cases or proposals for new test cases (as discussed in chapter 2)
- methods that are not well tested, according to extreme mutation (chapter 3)
- test cases that are suspected to be flaky (chapter 4)
- areas of large code bases that are not covered by the test suite (chapter 5)

Chapter 2

DSpot

Overview

The ultimate goal of DSpot is to automatically synthesize test improvements, *i.e.*, modifications of existing test cases, which are committed to the main test code repository. Our tool takes as input an object-oriented program and its test suite. From the existing tests, it synthesizes test improvements in the form of diffs that are proposed to the developer. For instance, Figure 2.1 shows a test improvement suggested by DSpot. The diffs are meant to be proposed as pull requests in a collaborative development setting such as Github.

```
146         ByteArrayOutputStream out = new ByteArrayOutputStream();
    -       writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
147 +       final int bytesWritten = writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
148 +       assertEquals(0, bytesWritten);
149         byte[] data = out.toByteArray();
```

Figure 2.1: Example of what DSpot produces: a diff to improve an existing test case.

DSpot aims at being fully automated. Hence, there is a need to approximate the features developers value the most in tests. To do so, DSpot uses the mutation score as a proxy to the developers' assessed value. In essence, developers value changes in test code if they enable them to catch new bugs, or in other words, if the improved test better specifies a piece of code. This is also reflected in the mutation score: if the mutation score increases, it means that a code element, say a statement, is better specified than before. In other words, DSpot uses the mutation score as fitness function to be increased, following the original conclusions of DeMillo, Lipton and Sayward who observed that mutants provide hints to generate test data [4]. More recent works on mutation testing demonstrated that mutation is suitable for testing experiments [1] and that mutants are valid substitutes for real faults [5].

There are two main kinds of improvement that can be done in a test case: assessing the behavior according to a new execution path (for new inputs), and better assessing the correctness the behaviour after one execution (for existing inputs). In DSpot, both improvements are done: DSpot proposes test modifications that improves existing tests both by triggering new execution paths and by strengthening assertions.

Hence, DSpot's work flow is composed of 2 main phases: 1) transformation of test code to create new test inputs, we call this "input space exploration"; 2) addition of new assertions with oracles to verify the execution state for new test inputs, we call this phase "assertion improvement". In DSpot, "amplification" means the combination of input space exploration and assertion improvement. The

following subsections are constructed on those two kinds of exploration.

Concepts

Definitions

We first define the core terminology of DSpot in the context of object-oriented Java programs.

Test suite is a set of test classes.

Test class is a class that contains test methods. A test class is neither deployed nor executed in production.

Test method or **test case** is a method that sets up the system under test into a specific state and checks that the actual state at the end of the method execution is the expected state.

Unit test is a test method that tests a single unit or component of the system. Typically, unit tests execute a small amount of code.

Test Case Structure

We consider traditional test cases for object-oriented programs. They are typically composed of two parts: input setup and assertions.

The input setup part is responsible for driving the program into a specific state. For instance, one creates objects and invokes methods on them to produce a specific state.

The assertion part is responsible for assessing that the actual behavior of the program corresponds to the expected behavior, the latter being called the oracle. To do so, the assertion uses the state of the program, *i.e.*, all the observable values of the program, and compare it to expected values written by developers. If the actual observed values of the program state and the oracle are different (or if an exception is thrown), the test fails and the program is considered as incorrect.

Input Space and Observation Space

We define the "input space" as the set of all possible test inputs under which the program can be exercised. This input space is an idealized concept because in practice it cannot be enumerated: the input space is infinite. The "specified input space" consists of points in the input space that are used in the existing test suite. The remaining points are considered as unspecified. DSpot aims at specifying unspecified points.

We define the "observation space" as the set of all possible observation points on a program under test. Technically, an observation point is a call to a public method, that allows to access a value describing one part of the execution state of the program under test.

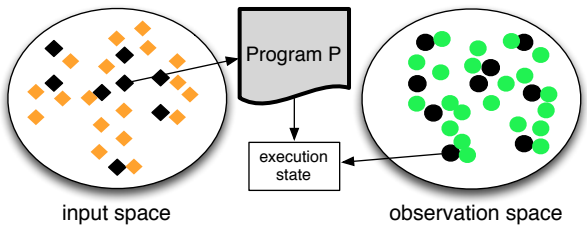


Figure 2.2: on the left the input space is composed by specified input points(orange diamond) and unspecified input points (black diamonds). On the right, the observation space over a given execution state of program. The green circle are values that are already asserted in the existing test suite., the newly added observations (black circles).

Algorithms

Input Space Exploration Algorithm

DSpot aims at exploring the input space so as to set the program in new never explored states. To do so, DSpot applies code transformations to the original manually-written test.

I-Amplification: I-Amplification, for Input Amplification, is the process of automatically creating new test input points from existing test input points.

DSpot uses three kinds of I-Amplification.

1) Amplification of literals: the new input point is obtained by changing a literal used in the test (numeric, boolean, string). For numeric values, there are five operators: $+1$, -1, $\times 2$, $\div 2$, and replacement by an existing literal of the same type, if such literal exists. For Strings, there are four operators: add a random char, remove a random char, replace a random char and replace the string by a fully random string of the same size. For booleans, there is one operator: negate the value;

2) Amplification of method calls DSpot manipulates method calls as follows: DSpot duplicates an existing method call in the original test; or it removes method calls; or it reuses an existing variable in the test input and generates an invocation for an accessible method.

3) Test objects If a new object is needed as a parameter while amplifying method calls, DSpot creates a new object of the required type using the default constructor if it exists. In the same way, when a new method call needs primitive value parameters, DSpot generates a random value.

DSpot combines the different kinds of I-Amplification iteratively: at each iteration all kinds of I-Amplification are applied, resulting in new tests. From one iteration to another, DSpot reuses the previously amplified tests, and further applies I-Amplification.

In the context, we call the *"parent"* of a test case t', the test case t on which DSpot applied a transformation to synthesize t'. In others words, $t' = amplifier.apply(t)$ in object-oriented difference (ie $t' = amplify(t)$ in imperative notation).

Assertion Improvement Algorithm

To improve existing tests, DSpot adds new assertions as follows.

A-Amplification: A-Amplification, for Assertion Amplification, is the process of automatically creating new assertions.

In DSpot, assertions are added on objects from the original test case. DSpot generates new assertions in existing test cases as follows: 1) it instruments the test cases to collect the state of program after execution (but before assertions), *i.e.*, it creates observation points; 2) it collects the actual values by running the instrumented test; 3) it generates new assertions in place of the observation points, using the collected values as oracle (as expected value in xUnit parlance).

During execution, a new test input may set the program in a state that throws an exception. In this case, DSpot produces a test asserting that the program throws a specific exception.

Implementation

DSpot is implemented in Java. It consists of 8800+ logical lines of code (as measured by cloc). For the sake of open-science, DSpot is made publicly available on Github[1].

Results

We evaluated DSpot on 10 open-source projects from GitHub. We selected those projects according to those following criteria: 1) they are written in Java, and use mostly JUnit as a testing framework.

[1] https://github.com/STAMP-project/dspot

2) they have a large and well developed test suite, because DSpot requires tests. 3) they have an active community on GitHub.

We selected 40 test classes within those 10 projects, 4 test classes for each one. We selected test classes according to their original mutation score, and to have a variety of seed classes, *i.e.*, some of test classes have a very good mutation score, and others have bad mutation score.

The results of the amplification is presented in Table 2.1 as follows: The first column is a numeric identifier. The second column is the name of test class to be amplified. The third column is the number of test methods in the original test class. The fourth column is the mutation score of the original test class. The fifth is the number of test methods generated by DSpot; the sixth and the seventh provide the number of covered (eg executed) mutants before and after amplification. The eight, ninth and tenth are respectively the mutation score of the original test class, the mutation score of its amplified version and the absolute increase obtained with amplification, with a pictogram indicating the presence of improvement; The eleventh and twelfth concerns the mutation score when only A-amplification is used. The thirteenth is the time consumed by DSpot to amplify the considered test class.

From Table 2.1, we observe: DSpot significantly improves the capacity of test classes at killing mutants in 26 out 40 of test classes, even in cases where the original test class is already very strong. The conjunct run of I-Amplification and A-Amplification is the best strategy for DSpot to improve the existing test classes. This experiment has shown that A-Amplification alone performs bad, in particular on tests that are already strong.

We also evaluated DSpot by proposing amplified test cases to developers through pull requests on GitHub. We proposed 9 pull requests; 7 of them have been merged by the developers. That means that the amplification of DSpot is considered as valuable by developers of real software.

This work is currently under review in an international, peer-reviewed journal.

Future Work

As future work, we investigate a new way to explore the input space.

This addresses some of the issues of the current algorithm. A lot of amplified tests have little value, because they trigger the same behavior as the original test, which is not interesting if we want to increase mutation score. Also, the exploration of the input space is done random, using pre-defined operators. That is why, we are currently devising an approach to create inputs that trigger new behavior.

To do this, we use symbolic execution. We transform the execution of the original test into a constraint model in Alloy[2]: classes are transformed into abstract signature; instances are transformed into unique signatures, that extend the abstract signature of the corresponding class; each assignment creates new unique signature with an assignment, as a fact; each conditional is transformed into facts.

The algorithm will negate each conditional (transformed as fact), one by one, to trigger a new behavior. It results with new inputs that trigger new behavior. This will be used has a I-Amplification. Then, DSpot will generate assertions and select amplified test cases.

[2] http://alloy.mit.edu/alloy/index.html

Table 2.1: The effectiveness of test amplification on 40 real test classes from notable open-source projects. A star after a test name means that the test is considered as subject for the case-study of Pull Request Evaluation. The source code of the original test classes as well as the amplified test classes is available on the online appendix.

ID	class	# Orig. test methods	Mutation Score	# New test methods	# Exec. mutants orig.	# Exec. mutants ampl.	# Killed mutants orig.	# Killed mutants ampl.	Increase killed		# Killed mutants only A-ampl	Increase killed only A-ampl	
	High Mutation Score ($> 50\%$)												
1	TypeNameTest*	12	57%	19	1037	1193	599	715	19%	↗	599	0%	→
2	NameAllocatorTest	11	87%	0	90	90	79	79	0%	→	79	0%	→
3	MetaClassTest*	7	63%	108	719	783	455	534	17%	↗	455	0%	→
4	ParameterExpressionTest	14	91%	2	177	177	162	164	1%	↗	162	0%	→
5	ObdDecoderTest*	1	83%	9	61	63	51	54	5%	↗	51	0%	→
6	MiscFormatterTest	1	89%	5	47	58	42	47	11%	↗	42	0%	→
7	TestLookup3Hash	2	95%	0	485	485	464	464	0%	→	464	0%	→
8	TestDoublyLinkedList	7	92%	1	112	113	104	105	1.0%	↗	104	0%	→
9	ArraysIndexesTest	1	58%	15	979	1076	576	647	12%	↗	586	1%	↗
10	ClasspathResolverTest	10	67%	0	74	74	50	50	0%	→	50	0%	→
11	RequestTest*	17	89%	4	157	173	141	156	10%	↗	141	0%	→
12	PrefixedCollapsibleMapTest	4	96%	0	56	56	54	54	0%	→	54	0%	→
13	TokenQueueTest	6	72%	18	209	218	152	165	8%	↗	152	0%	→
14	CharacterReaderTest	19	79%	71	387	391	309	336	8%	↗	309	0%	→
15	AttributesTest	5	52%	9	601	604	316	322	1%	↗	316	0%	→
16	TailDelimiterTest*	10	71%	1	530	530	381	384	0.8%	↗	381	0%	→
17	LinkBufferTest	3	75%	12	87	136	66	90	36%	↗	66	0%	→
18	CodedInputTest	1	60%	29	180	400	108	166	53%	↗	108	0%	→
19	FileNamePatternTest*	12	75%	27	757	982	573	686	19%	↗	573	0%	→
20	SyslogAppenderBaseTest	1	99%	1	144	149	143	148	3%	↗	143	0%	→
21	RequestBuilderAndroidTest	2	99%	0	514	514	513	513	0%	→	513	0%	→
22	CallAdapterTest	4	94%	0	58	58	55	55	0%	→	55	0%	→
23	ExecutorCallAdapterFactoryTest	7	62%	0	191	191	119	119	0%	→	119	0%	→
24	CallTest	35	69%	3	920	922	642	644	0.3%	↗	642	0%	→
	Low Mutation Score ($<= 50\%$)												
25	FieldSpecTest	2	32%	12	694	712	223	316	41%	↗	223	0%	→
26	ParameterSpecTest	2	32%	11	653	663	214	293	36%	↗	214	0%	→
27	WrongNamespacesTest	2	9%	6	870	870	78	249	219%	↗	249	219%	↗
28	WrongMapperTest	1	8%	3	1089	1089	97	325	235%	↗	325	235%	↗
29	ProgressProtocolDecoderTest	1	17%	2	105	108	18	27	50%	↗	23	27%	↗
30	IgnitionEventHandlerTest	1	22%	0	59	59	13	13	0%	→	13	0%	→
31	TestICardinality	2	7%	0	248	248	19	19	0%	→	19	0%	→
32	TestMurmurHash*	2	25%	40	202	292	52	275	428%	↗	174	234%	↗
33	ConcurrencyTest	2	28%	2	727	728	210	342	62%	↗	210	0%	→
34	AbstractClassTest*	2	37%	28	1026	1102	383	475	24%	↗	405	5%	↗
35	AllTimeTest	3	42%	0	386	386	163	163	0%	→	163	0%	→
36	DailyTest	3	42%	0	386	386	163	163	0%	→	163	0%	→
37	AttributeTest*	2	40%	33	437	489	178	225	26%	↗	180	1%	↗
38	CodedDataInputTest	1	1%	0	451	451	5	5	0%	→	5	0%	→
39	FileAppenderResilience_AS_ROOT_Test	1	4%	0	87	87	4	4	0%	→	4	0%	→
40	Basic	1	10%	0	57	57	6	6	0%	→	6	0%	→

Chapter 3

Descartes

Mutation testing has been largely recognized as a robust technique to evaluate test suites [5]. It transforms the main code by introducing artificial faults in the form of programming errors. After that, the involved test cases are run to verify if they can detect the planted faults. The number of potential code transformations, or mutations, is huge even for simple programs making this analysis computationally expensive. That is one of the main arguments used to explain why this technique haven't found wide utilization in the software industry [6]. Nevertheless, there are some practical tools that provide mutation testing analysis implementations and the most popular of them, targeting Java programs, is PITest [1] [2].

In 2016, a new type of mutation scheme has been proposed [7]. This new approach implements a different kind of mutation that does not simulate subtle programming errors. The transformations consist in removing all the instructions of a method. If the method is non-void, then its body is left with only a proper return instruction using a predefined value. This modification ensures that all possible side effects are removed and the result after each invocation will always be the same. It is possible to apply more than one mutation to a non-void method if different values of the corresponding type are used. A method is said to be pseudo-tested if no extreme mutation applied to its body is detected by the test suite. As it modifies the code in a more aggressive manner, this scheme has been called **extreme mutation**

The authors of this new approach expose that it has two major advantages: for one thing, the number of possible mutants remains linear with respect to the number of methods to be analyzed and secondly, most equivalent mutants created can be automatically detected. These two features make the approach attractive to be employed in real-life projects.

In their study, the authors provide empirical evidence suggesting that the ratio of pseudo-tested methods varies among different projects, ranging from 6% to 53% within their subjects. They also show that a significant number of those methods contains code with relevant functionalities and are frequently covered only by system tests, that is, tests executing a large portion of the entire project.

To be able to experiment with extreme mutation and expand its use we have created an effective implementation in the form of a plug-in for PITest called Descartes [2].

PITest and Descartes

PIT or Pitest is a state of the art mutation tool that targets Java projects. It can be used with most major build systems such as Ant, Maven and Gradle. A comparison made some years ago on the usability of tools for mutation analysis [3], highlights PITest as the most robust alternative at the time. All

[1] http://pitest.org
[2] https://github.com/STAMP-project/pitest-descartes

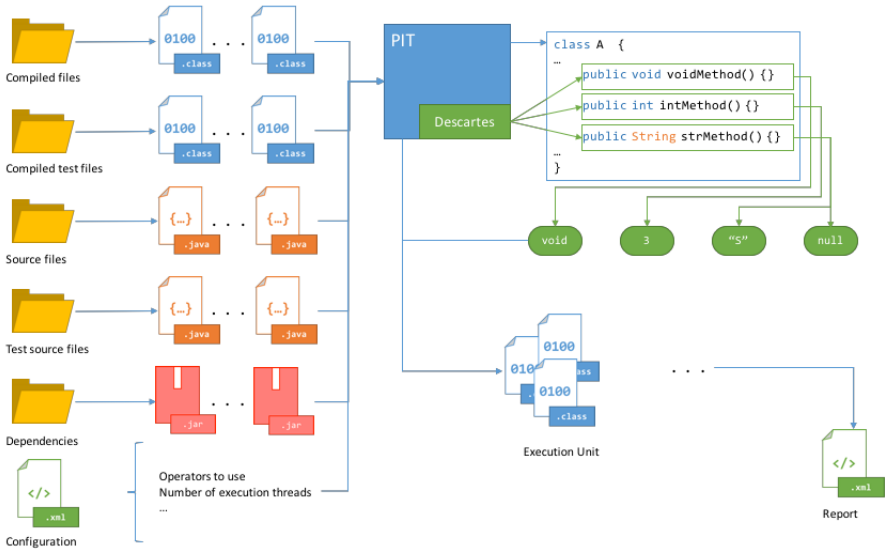


Figure 3.1: Interaction between PITest, Descartes and a target project during mutation analysis.

features that made PITest a good selection for this task still hold. The tool incorporates mechanisms for test selection, ordering and multi-threaded execution to speed up the analysis. It implements many traditional types of mutation and can be extended via plug-ins. This last feature makes PITest a good choice as development framework to build a custom mutation tool.

Descartes is, in fact, a new mutation engine for PITest. In PITest's jargon, a mutation engine handles the available operators, finds the possible mutation points and ultimately applies the transformations. The remaining parts of PITest take care of classes and tests discovery, execution and reporting (See figure 3.1).

In mutation testing, all transformations are performed by units called **mutation operators**. Our extension provides a set of configurable extreme mutation operators. A user can specify the constant values to be returned in methods involving primitive types and $String$. There is also a $null$ operator that targets reference type methods, another that returns an empty array where possible and one that works over $void$ methods. The tool automatically skips from the analysis class constructors, empty $void$ methods, methods returning only a constant value and methods with the structure of a simple getter or setter, as a way to eliminate potential equivalent mutants, that is, mutants that have the same behavior as the original program. The complete list of mutation operators can be found in Descartes' code repository [3].

Preliminary results

We have used Descartes to analyze a several real-life and open source projects from Github. The list of projects and their size measured in the amount of lines of code can be seen in table 3.1. We also used PITest to analyze each project and compare the outcome given by Descartes with the results reported by the default mutation engine shipped with PITests.

Table 3.2 shows the number of mutants created by both engines, as well as the time that took each analysis and the mutation score, that is, the percentage of introduced faults that were detected. It can be seen that the number of mutants created by Descartes is lower in all cases, fact that has a strong

[3] https://github.com/STAMP-project/pitest-descartes/blob/d2eb86ce3d59e83b075506d60833a83c20e09772/README.md

Table 3.1: Projects used as study subjects and their size in number of lines of code.

Project	LOC	Project	LOC
Amazon Web Services SDK	1703892	SAT4J	26415
Apache Commons Math	180951	Apache Commons IO	24334
Apache PdfBox	137099	ImageJ Common	23868
JFreeChart	134353	Jaxen XPath Engine	21204
Java Git	128874	Google Gson	20068
XWiki Rendering Engine	104727	jsoup	17839
Joda-Time	85911	Apache Commons Codec	17267
Apache Commons Lang	60733	AuthZForce PDP Core	16602
SCIFIO	55347	JOpt Simple	9214
Apache Commons Collections	51632	Apache Commons CLI	7005
Urban Airship Client Library	40885		

Table 3.2: Projects used as study subjects, their size in amount of lines of code and the outcome from PITest and Descartes.

	Descartes			PITest		
Project	Mutants	Score	Time	Mutants	Score	Time
Amazon Web Services SDK	161758	82.67	01:27:30	2141690	76.28	04:25:35
Apache Commons Math	7150	96.06	00:08:30	104786	83.81	03:22:18
Apache PdfBox	7559	78.95	00:42:11	79763	58.89	06:20:25
JFreeChart	7210	79.95	00:05:26	89592	58.04	00:41:28
Java Git	7152	84.28	00:56:07	78316	73.86	16:02:03
XWiki Rendering Engine	5534	81.61	00:10:50	112609	50.89	01:59:55
Joda-Time	4525	95.60	00:04:13	31233	81.65	00:16:32
Apache Commons Lang	3872	95.62	00:02:18	30361	86.17	00:21:02
SCIFIO	3627	79.55	00:15:26	62768	45.92	03:12:11
Apache Commons Collections	3558	93.63	00:01:48	20394	85.94	00:05:41
Urban Airship Client Library	3082	93.14	00:09:38	17345	82.26	00:11:31
SAT4J	2296	74.63	00:56:42	17067	68.58	11:11:48
Apache Commons IO	1164	93.59	00:02:18	8809	84.73	00:12:48
ImageJ Common	1947	72.64	00:04:08	15592	54.77	00:29:09
Jaxen XPath Engine	1252	96.51	00:01:34	12210	67.13	00:24:40
Google Gson	848	92.86	00:01:11	7353	81.76	00:05:34
jsoup	1566	94.38	00:02:45	14054	78.34	00:12:49
Apache Commons Codec	979	95.47	00:02:02	9233	87.82	00:07:57
AuthZForce PDP Core	626	92.86	00:08:45	7296	88.18	01:23:50
JOpt Simple	412	94.72	00:00:25	2271	93.52	00:01:36
Apache Commons CLI	271	96.14	00:00:09	2560	88.71	00:01:26

impact in the running time of the analysis. Two cases to note come from comparing the running time in the analysis of SAT4J and Java Git. For those projects PITest took more that 10 hours to complete the process, while Descartes employed a little less than one hour. As for the scores, they are correlated to a certain point. In most cases, projects with high mutation score for one tool will exhibit a similar result for the other. With this, Descartes could be used as a first step analysis while checking the test suite of a project.

Experiment with XWiki

XWiki has participated in several discussions on the topic of Descartes and PITest, with the goal of identifying how they can be used on a real world project such as XWiki. The main goal was to identify what value could be extracted from applying them and the limitations that currently exist and that would need to be improved.

Some identified areas are:

- It's interesting to classify the results in 3 categories:
  - *strong pseudo-tested methods*: no matter the return values of a method, the tests still passes. This is the worst offender since it means the tests really needs to be improved.
  - *weak pseudo-tested methods*: the tests passes with at least one modified value. Not as bad as strong pseudo-tested but you may want still want to check it out.
  - *fully tested methods*: the tests fail for all mutations and thus can be considered as tested.
- The generated report should provide this classification to help analyze the results and focus on important problems.
- It would be nice if the Maven plugin was improved and be able to fail if the mutation score was below a certain threshold (as XWiki does for test coverage). This is a functionality that PITest already provides for single-module projects.
- Performance: It's quite slow compared to Jacoco execution time for example. Overall, one of the main issues with mutation testing is precisely the execution time. On the other hand, code coverage is easy and cheap to compute.
- It would be nice to have a Sonar integration so that PIT/Descartes could provide some stats on the Sonar dashboard.
- Big limitation: ATM there's a big limitation: PIT (and/or Descartes) doesn't support being executed on a multi-module project. This means that right now you need to compute the full classpath for all modules and run all sources and tests as if it was a single module. This causes problems for all tests that depend on the filesystem and expect a given directory structure. It's also tedious and a error-prone problem since the classpath order can have side effects.

XWiki is also currently experimenting with DSpot. DSpot uses PIT and Descartes but in addition it uses the results to generate new tests automatically. The Descartes project could also use the information provided by line coverage to automatically generate tests to cover the spotted issues.

The image below shows an example of running Descartes on XWiki and finding a bug in the equals() method of a Class:

```
106     @Override
107     public boolean equals(Object object)
108     {
109         boolean result;
110
111         // See http://www.technofundo.com/tech/java/equalhash.html for the detail of this algorithm.
112 2       1. equals : All method body replaced by: return true → SURVIVED
113         2. equals : All method body replaced by: return false → KILLED
114         } else {
115             if ((object == null) || (object.getClass() != this.getClass())) {
116                 result = false;
117             } else {
118                 MacroId macroId = (MacroId) object;
119                 result =
120                     (getId() == macroId.getId() || (getId() != null && getId().equals(macroId.getId())))
121                         && (getSyntax() == macroId.getSyntax() || (getSyntax() != null && getSyntax().equals(
122                             macroId.getSyntax())));
123             }
124         }
125         return result;
126     }
127 }
```

Future work

A big Maven project would probably have a complex structure involving many submodules referencing each other. That is the case, for example, of Amazon Web Services SDK and XWiki Rendering Engine, both included in table 3.1. In this kind of project is frequent to find integration tests between submodules and even submodules devoted only to contain unit tests the others. PITest is not able to tackle such a scenario as it will handle each submodule in isolation. The possibility of handling projects with submodules is a popular request in the PITest user community. It has been the source of several issues posted in the Github page and it is also the main subject of some discussions in the user forums [4]. During the experiments performed we have found a workaround for projects with multiple-modules. Nevertheless, a formal and reusable solution is desirable and we have already started its development [5].

Even when Descartes provides a faster mutation analysis, the execution process takes a considerable amount of time. For complex projects it is not possible to perform this kind of operation in a regular manner, for example, after every commit. Given this, one of the development directions to expand the usability of the mutation tool is to include it in a Continuous Integration pipeline using well established systems such as Jenkins or Travis. It could be also useful to create a Github utility that automatically analyses the code of a project after a commit or merging a pull request.

The mutation score is often used as a quantitative characterization of the quality of a test suite. However, the number might be affected by several factors and is more valuable for developers to find concrete testing issues and automatically propose a viable solution. In our experiments, we have found that an undetected mutants reported by Descartes may be related to code coverage problems, weak assertions, or code difficult to reach and control from the implemented tests. Also, pseudo-tested methods are executed by the test suite but are not properly verified and could be perfect targets for test amplification. Next research steps will be directed to identify scenarios uncovered by Descartes for which a solution can be achieved by generating new tests or modifying the existing test cases.

[4] For example, issues labeled 41, 224, 307, 376 (https://github.com/hcoles/pitest/issues)
[5] https://github.com/STAMP-project/pitmp-maven-plugin

Chapter 4

Flaky test detector

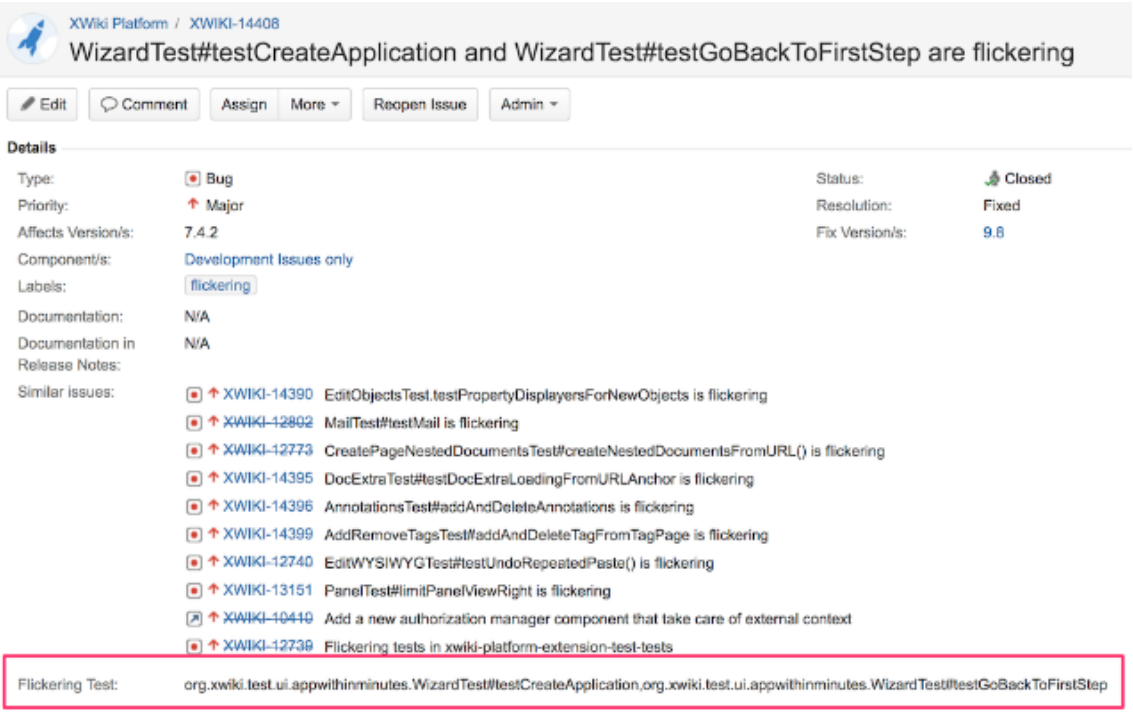
Flaky tests generate false positives and reduce the trust in the Continuous Integration (CI) system in place. Identifying and handling Flaky tests is thus an important priority. In addition to Flaky tests there is another category of failures that cause the same problems: environment related failures (connection to SCM failing, missing X display, JVM crash, etc).

As part of STAMP we defined the following KPI "Decrease by 20% the number of tests which fail once but not again if run several times".

Strategy

We propose the following strategy:

- Execute the tests in a CI. We use Jenkins.
- Identify Flaky tests. This part has to be done manually or semi-manually because it is really hard to identify with accuracy a flaky test and more importantly it takes time. A strategy that would execute each failing test a number of times would not work properly since that would penalize the speed of execution of all the CI jobs leading to unacceptable delays to notice real failures and as importantly, there's no magic number for the number of times a failing test has to be repeated before passing. In our experience some Flaky tests we noticed were working 99 times out of 100 for example.
- We use JIRA to register Flaky test by introducing some custom field:



- Record any Flaky tests in an issue tracker. This serves two purpose:
  - It lots a task to be done to fix the test in the future. Thus this cannot be fogotten and can be planned in the roadmap
  - It identifies existing Flaky tests and allows us to develop a tool so that when a test fails on the CI we can check if it's a known Flaky tests and not send a failure email in this case.
- On the CI recognize environment-related job failures and don't send email in this case to avoid false positives.

From the point of view of STAMP the idea will be to count the number flagged flaky tests in JIRA at the beginning of the project using a JQL query such as: `"Flickering Test" is not empty and resolution = Unresolved`.

Environment-related problems

On the XWiki project's CI build we have identified the following environment-related problems. Most of them are generic and can apply to any Java project.

We identify them by looking for String matches in the Maven build executed by the CI job:

String Match	Reason
A fatal error has been detected by the Java Runtime Environment	JVM crash
Error: cannot open display: :1.0	VNC not running
java.lang.NoClassDefFoundError: Could not initialize class sun.awt.X11GraphicsEnvironment	VNC issue
hudson.plugins.git.GitException: Could not fetch from any repository	Git issue
Error communicating with the remote browser. It may have died.	Browser issue
Failed to start XWiki in .* seconds	XWiki starting too slow
Failed to transfer file.*nexus.*Return code is:.*ReasonPhrase:Service Temporarily Unavailable	Nexus down
com.jcraft.jsch.JSchException: java.net.UnknownHostException: maven.xwiki.or	Maven.xwiki.org unavailable
Fatal Error: Unable to find package java.lang in classpath or bootclasspath	Compilation issue
hudson.plugins.git.GitException: Command.	Git issue
Caused by: org.openqa.selenium.WebDriverException: Failed to connect to binary FirefoxBinary.	Browser issue: Firefox not there
java.net.SocketTimeoutException: Read timed out	Unknown connection issue
Can't connect to X11 window server.	VNC not running
The forked VM terminated without saying properly goodbye.	Maven surefire fork VM crash
java.lang.RuntimeException: Unexpected exception deserializing from disk cache	GWT building issue
Unable to bind to locking port 7054	Selenium bind issue with browser
Error binding monitor port 8079: java.net.BindException: Cannot assign requested address	XWiki instance already running
Caused by: java.util.zip.ZipException: invalid block type	Maven build error
java.lang.ClassNotFoundException: org.jvnet.hudson.maven3.listeners.MavenProjectInfo	Jenkins error
Failed to execute goal org.codehaus.mojo:sonar-maven-plugin.*No route to host	Sonar connexion error

Jenkins Pipeline Tool

The tool we have developed is a Jenkins pipeline to detect flaky tests and environment-related false positives and to not send emails in this case.

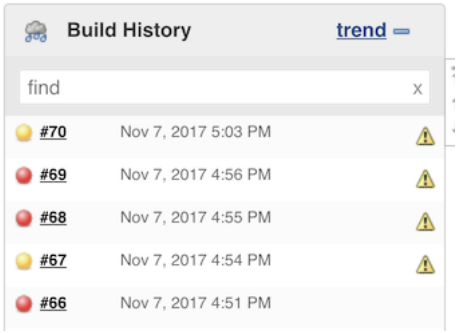
Listing 4.1: Pipeline for flaky test detection

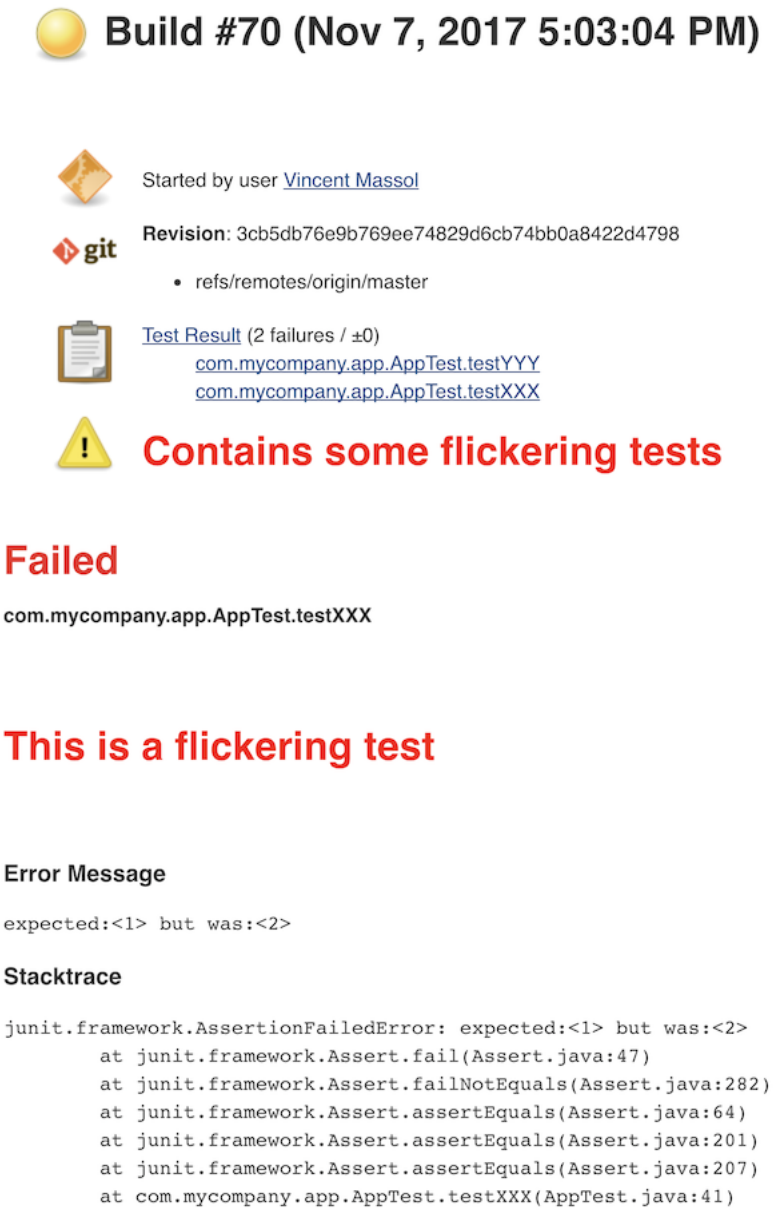
```
1
  stage('Post Build')
3 // Check for false positives \& Flickers.
  echoXWiki {'Checking for false positives and flickers in build results...'}
5 echoXWiki {'Checking for false positives and flickersin build results...'}
  def containsFalsePositivesOrOnlyFlickers =
7 checkForFalsePositivesAndFlickers()
  // Also send a mail notification when the job has failed tests.
```

```
//The JUnit archiver above will mark the build as UNSTABLE when there are failed tests

if (currentBuild.result == 'UNSTABLE' && !containsFalsePositivesOrOnlyFlickers)
notifyByMail(currentBuild.result)
}}

...
def boolean checkForFalsePositivesAndFlickers(){
// Step 1: Check for false positives}
def containsFalsePositives = checkForFalsePositives()
// Step 2: Check for flickers
def containsOnlyFlickers = checkForFlickers()
return containsFalsePositives || containsOnlyFlickers
}

/* Check for false positives for known cases of failures unrelated to
code.
* @return true if false positives have been detected} */
def boolean checkForFalsePositives() {
def messages = [['.*A fatal error has been detected by the Java Runtime Environment.*', 'JVM Crash', 'A JVM crash happened!'], ... ]
messages.each { message ->
if (manager.logContains(message.get(0))) {
manager.addWarningBadge(message.get(1))
manager.createSummary("warning.gif").appendText("<h1>${message.get(2)}</h1>", false, false, false, "red")
manager.buildUnstable()
echoXWiki "False positive detected [${message.get(2)}] ..."
return true}}}


 /** Check for test flickers, and modify test result descriptions for tests that are identified as flicker. A test is
 * a flicker if there's a JIRA issue having the "Flickering Test" custom field containing the FQN of the test in the
 * format {@code <java package name>#<test name>}.
 * @return true if the failing tests only contain flickering tests*/
def boolean checkForFlickers(){
  boolean containsOnlyFlickers = false
  AbstractTestResultAction testResultAction = currentBuild.rawBuild.getAction(AbstractTestResultAction.class)
  if (testResultAction != null) {
    // Find all failed tests
    def failedTests = testResultAction.getResult().getFailedTests()
    if (failedTests.size() > 0) {
      // Get all false positives from JIRA
      def url = "https://jira.xwiki.org/sr/jira.issueviews:searchrequest-xml/temp/SearchRequest.xml?".concat(
      "jqlQuery=%22Flickering%20Test%22%20is%20not%20empty%20and%20resolution%20=%20Unresolved")
      def root = new XmlSlurper().parseText(url.toURL().text)
      def knownFlickers = []
      root.channel.item.customfields.customfield.each() { customfield ->
      if (customfield.customfieldname == 'Flickering Test') {
```

```
        customfield.customfieldvalues.customfieldvalue.text().split(',').each() {
 knownFlickers.add(it)
   }}}
   echoXWiki "Known flickering tests: ${knownFlickers}"
   // For each failed test, check if it's in the known flicker list.
   // If all failed tests are flickers then don't send notification email
   def containsAtLeastOneFlicker = false
   containsOnlyFlickers = true
   failedTests.each() { testResult ->
   // Format of a Test Result id is "junit/<package name>/<test class name>/<test
       method name>"
   def parts = testResult.getId().split('/')
   def testName = "${parts[1]}.${parts[2]}#${parts[3]}"
   if (knownFlickers.contains(testName)) {
     // Add the information that the test is a flicker to the test's description
     testResult.setDescription("<h1 style='color:red'>This is a flickering test</
         h1>${testResult.getDescription() ?: ''}")
 echoXWiki "Found flickering test: [${testName}]"
 containsAtLeastOneFlicker = true
     } else {
       // This is a real failing test, thus we'll need to send athe notification
           email...
       containsOnlyFlickers = false}}
     if (containsAtLeastOneFlicker) {
       manager.addWarningBadge("Contains some flickering tests")
       manager.createSummary("warning.gif").appendText("<h1>Contains some
           flickering tests</h1>", false, false, false, "red")}}}
   return containsOnlyFlickers
}
```

This tool will not only prevent sending false positives email but it'll also modify the Jenkins UI to indicate flaky test very visibly:





This work has been communicated on the XWiki blog [1] [2] .

Roadmap

In the future, we will try to look at the Jenkins recorded history for a past Test and try to automatically infer whether it is a false positive or not. However, as explained above, this can never be guaranteed and thus a human would need to look at the flagged test anyway. Thus it could serve as a semi-automated way of discovering flaky tests.

[1] http://massol.myxwiki.org/xwiki/bin/view/Blog/FlakyTestTool
[2] http://massol.myxwiki.org/xwiki/bin/view/Blog/JenkinsDontSendEmailForFalsePositives

Chapter 5

Enhanced code coverage

An important part of test amplification is the ability to generate additional tests from existing test suites. These additional tests will increase the test coverage, i.e. the amount of code traversed by tests during their executions, leading to identify more easily bugs and preventing future regressions.

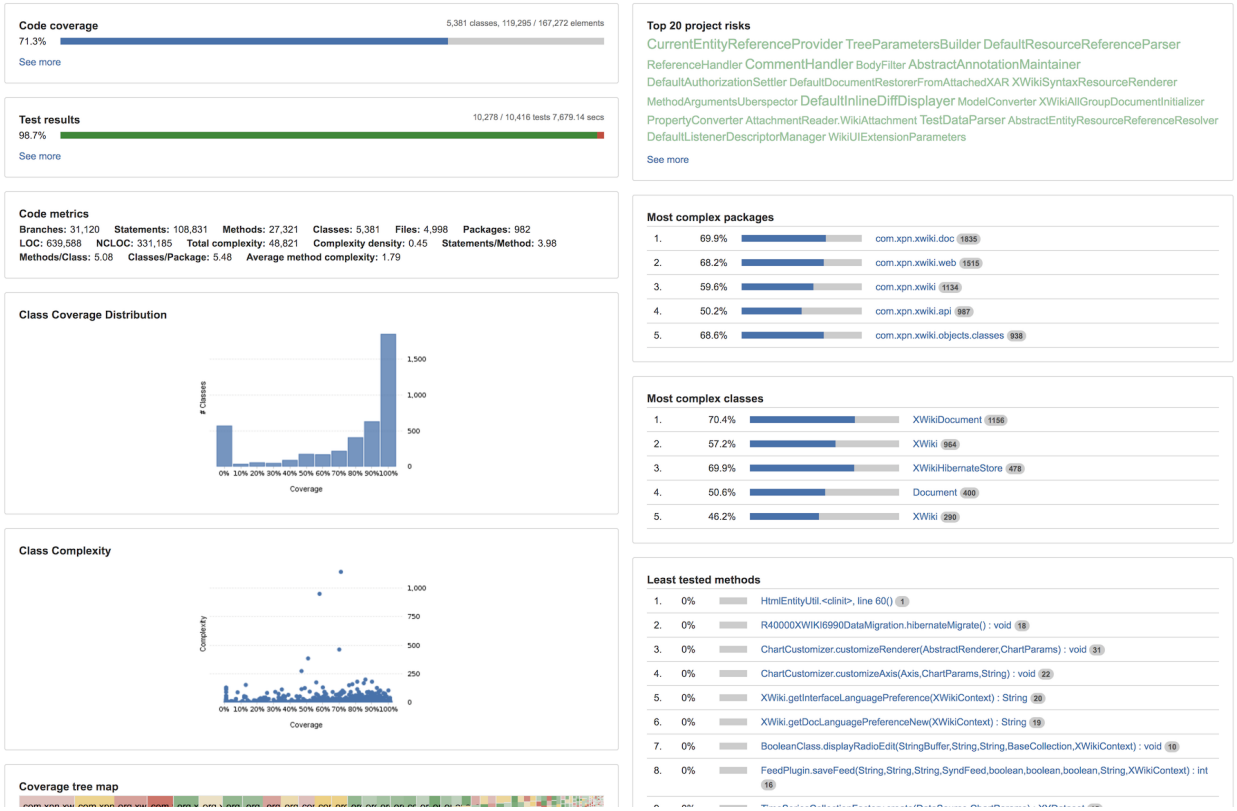
Thus it's important to be able to compute each projet's test coverage at the start of the STAMP research project and at the end, in order to find out the increase in test coverage.

This is actually required to achieve KPI : "Increase the diversity of execution paths covered by 40%".

We're proposing to use Clover for Maven [1] as the main tool to perform this measure. The reasons for choosing Clover are several:

- It's open source [2]
- Works with Maven and other build tools (Ant, Gradle, etc)
- Includes coverage generated by functional and system tests
- It supports generating test coverage from tests executed in multiple builds coming from multiple source repositories
- It generates interesting metrics in addition to test coverage, such as total number of tests, complexity of code, and more. In addition it identifies project risks, least tested methods, and generally provide a very nice dashboard. For example[3]:

[1] https://confluence.atlassian.com/clover/clover-for-maven-2-and-3-96567412.html
[2] https://www.atlassian.com/blog/announcements/atlassian-clover-open-source
[3] http://maven.xwiki.org/site/clover/20171109/clover-commons+rendering+platform-20171109-1920/dashboard.html



Terminology

In order to have a shared understanding on what test coverage means, we're proposing to use the concept of **TPC** which is proposed by Clover and computed as follows[4]:

```
TPC = (CT + CF + SC + MC)/(2*C + S + M) * 100
```

where

- CT - conditionals that evaluated to 'true' at least once
- CF - conditionals that evaluated to 'false' at least once
- SC - statements covered
- MC - methods entered
- C - total number of conditionals
- S - total number of statements
- M - total number of methods

Strategy

The general strategy to use Clover would be:

[4] https://confluence.atlassian.com/clover/how-are-the-clover-coverage-percentages-calculated-79986990.html

- Execute it on each project on the source before any amplification and more generally at the beginning of the STAMP project
- Execute it again on each project after amplification at various points during the STAMP project lifetime
- Compute the TPC difference

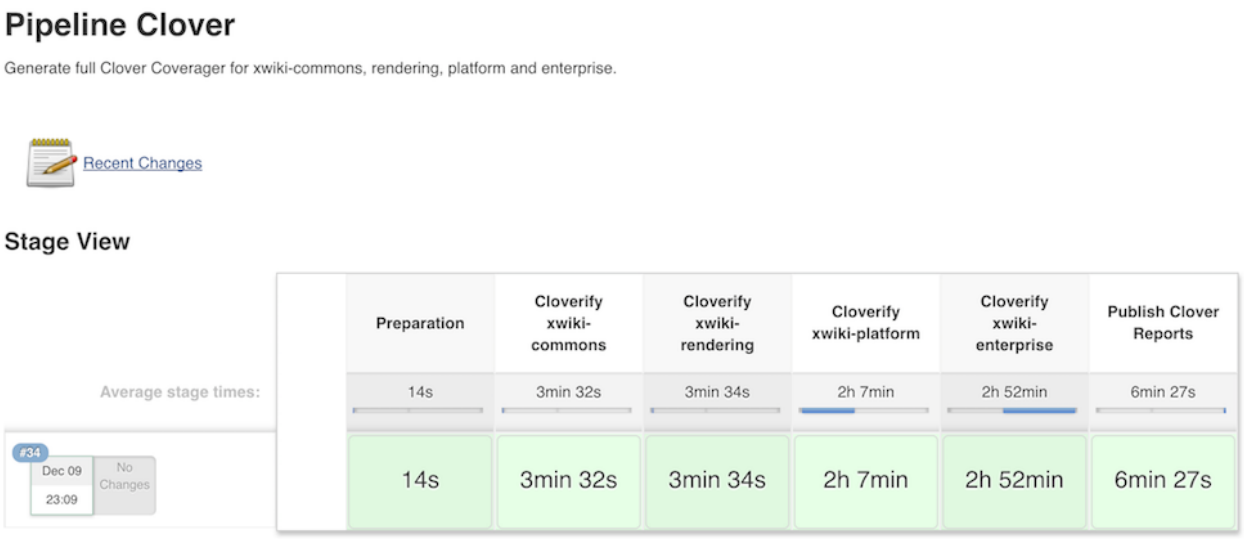
Those 3 steps require developing some tools:

- One for running Clover in Maven
- One for running Clover on all the modules, including modules located in different repositories
- One for computing the differences in the various generated reports

On a large real life project, source code may be located in various repositories (e.g. various Git repositories) and we need to configure Clover to generate Clover data in a single location across builds.

In order to fully automate this, we have developed a Jenkins Pipeline Job [5], which can be configured for each project.

- Prepare a custom Maven repository for storing the result of JAR instrumentation so that the instrumented JAR do not get mixed up with other standard and non instrumented JAR (and so that they don't go by mistake in production too!).
- Prepare a location where Clover data will be output
- For each Git repository (xwiki-commons, xwiki-rendering and xwiki-platform in this example), run Maven with Clover, passing the proper parameters to Maven so that it uses our custom Maven repository and the common Clover location. Also make sure that test failures don't stop the generation.
- Last, generate a Clover report from the full test coverage data. We generate two reports: an HTML one and an XML one. The XML one is useful to extract data and make additional computations as shown below.



[5] https://github.com/STAMP-project/xwiki-jenkins-pipeline/blob/master/scripts/clover.groovy

You can see an example of such a report at [6], and publication [7].

TPC Report Difference

Once we have generated the various Clover reports we need to compute the difference and check how the TPC has improved.

At first sight, it would seem it would be as simple as looking at the TPC value in the report. However it's not that easy. Clover differentiates Application Code from Test Code and depending on the generated reports, this separation does not happen in the same way. And the TPC value computed is computed only on Application Code. Thus for example when you look at the reports [8], one may think that there hss been some regression in TPC, going from 73.2% down to 71.3%. But that is not correct.

Since we asked Clover to generate an XML report too, we can use it compute a proper TPC tha is the same across reports [9]

We have setup a tool for that [10].

Differences

Package	TPC Old	TPC New
ALL	74.07000000000001	76.28
com.xpn.xwiki	60.57	64.34
com.xpn.xwiki.api	48.3	53.81
com.xpn.xwiki.criteria.impl	63.93	64.570000
com.xpn.xwiki.doc	74.68	76.3
com.xpn.xwiki.internal	86.28	87.59
com.xpn.xwiki.internal.cache	81.55	86.4
com.xpn.xwiki.internal.cache.rendering	86.61	87.4
com.xpn.xwiki.internal.event	91.82000000000001	92.27
com.xpn.xwiki.internal.filter	70.19	74.9
com.xpn.xwiki.internal.filter.input	84.56	84.48
com.xpn.xwiki.internal.filter.output	89.62	89.92
com.xpn.xwiki.internal.job	84.37	77.58

[6] http://maven.xwiki.org/site/clover/20171109/clover-commonsrenderingplatform-20171109-1920/dashboard.html

[7] http://massol.myxwiki.org/xwiki/bin/view/Blog/FullCoverageClover

[8] http://maven.xwiki.org/site/clover/20171109/clover-commons+rendering+platform-20171109-1920/dashboard.html,http://maven.xwiki.org/site/clover/20171109/clover-commons+rendering+platform-20171109-1920/pkg-summary.html

[9] http://massol.myxwiki.org/xwiki/bin/view/Blog/ComparingCloverReports

[10] http://dev.xwiki.org/xwiki/bin/view/Community/Testing#HComputeDifferencesbetweenreports

Chapter 6

Conclusion

This deliverable reports on the development activities related to the unit test amplification tool box. The key components are in place to transform existing test cases, analyze executions and assess the quality of test suites.

Future work will focus on:

- Research on DSpot and Descartes. For the former, we focus on the exhaustive exploration of amplification spaces; while for the latter we focus on ranking strategies to identify the most meaningful hints to improve test suites.
- Embed the tools in a DevOps pipeline: integrate Descartes and DSpot in the continuous integration engine and evaluate strategies to manage the mutants and the amplified test cases.
- Enhance the integration with WP2.

Bibliography

[1] ANDREWS, J. H., BRIAND, L. C., AND LABICHE, Y. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering* (2005), ACM, pp. 402–411.

[2] COLES, H., LAURENT, T., HENARD, C., PAPADAKIS, M., AND VENTRESQUE, A. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (New York, NY, USA, 2016), ISSTA 2016, ACM, pp. 449–452.

[3] DELAHAYE, M., AND BOUSQUET, L. D. A Comparison of Mutation Analysis Tools for Java. In *2013 13th International Conference on Quality Software* (July 2013), pp. 187–195.

[4] DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *Computer 11*, 4 (1978), 34–41.

[5] JUST, R., JALALI, D., INOZEMTSEVA, L., ERNST, M. D., HOLMES, R., AND FRASER, G. Are Mutants a Valid Substitute for Real Faults in Software Testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, ACM, pp. 654–665.

[6] MOŽUCHA, J., AND ROSSI, B. Is Mutation Testing Ready to Be Adopted Industry-Wide? In *Product-Focused Software Process Improvement* (Nov. 2016), Lecture Notes in Computer Science, Springer, Cham, pp. 217–232.

[7] NIEDERMAYR, R., JUERGENS, E., AND WAGNER, S. Will my tests tell me if I break this code? In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery* (New York, NY, USA, 2016), ACM Press, pp. 23–29.