Let the CI spot the holes in tested code with the Descartes tool

Oscar L. Vera-Pérez

Inria Rennes - Bretagne Atlantique
Rennes, France
oscar.vera-perez@inria.fr

Benoit Baudry

KTH Royal Institute of Technology
Stockholm, Sweden
baudry@kth.se

Vincent Massol

XWiki
Paris, France
vincent@xwiki.com

Abstract

This documents contains complementary notes for the tutorial "Let the CI spot the holes in tested code with the Descartes tool". It explains the main concepts surrounding mutation testing and the recently proposed extreme mutation. It also describes *Descartes* a tool to detect pseudo-tested methods. We welcome all feedback and suggestions to improve this document.

1 Introduction

Test automation is a common practice in software development nowadays. Test artifacts are often as large or even larger than the main codebase. Tests are being written for different scopes and different levels of abstraction and granularity: unit tests, integration tests, system tests, end-to-end tests, performance tests, etc.

Test cases are expected to cover the requirements of the application under development. They are also expected to stress the application and prevent regressions. Overall they are designed and executed to find bugs before the code goes to production.

Most tests are automated with the use of libraries designed for the matter. A typical JUnit test case is shown in Listing 1. It is a method that initializes the program in a specific state , triggers specific behaviors (Line 5 to Line 9) and specifies the expected effects for these behaviors through assertions (Line 7 to Line 9).

```
1       class CharSetTest {
            ...
3           @Test
            public void testConstructor() {
5               CharSet set = CharSet.getInstance("a");
                CharRange[] array = set.getCharRanges();
7               assertEquals("[a]", set.toString());
                assertEquals(1, array.length);
9               assertEquals("a", array[0].toString());
            }
11          ...
        }
```

Listing 1: A test case of the class `CharSetTest` taken from `commons-lang`

The quality of a test case depends on how well the input has been chosen and how strong the assertions are. But, being the test suite a piece of software as well, How can developers and testers be sure, that the test suite does what it is supposed to do? How can they be sure that the test suite is adequate enough to spot bugs in the codebase?

2 Code coverage

One of the most used criterion to assess the quality of a test suite is to compute the percentage of instructions in the codebase that are executed (covered) by the test suite, known as code coverage. Code coverage is widely used because it is relatively easy and cheap to obtain. It just requires to instrument the source code and does not add a great overhead in terms of execution time. There are several tools available for most programing languages. For Java programs it is possible to use JaCoco[1], Cobertura[2], OpenClover[3], just to mention a few. Many IDEs support code coverage computation out of the box (IntelliJ) of via plugins

[1] `https://www.jacoco.org/jacoco/`
[2] `https://cobertura.github.io/cobertura/`
[3] `http://openclover.org/`

(Eclipse). Most of these tools are also available for Continuous Integration Servers such as Jenkins or Travis.

Code coverage is useful to determine the parts of the code that are not tested. Listing 2 shows a method that computes the factorial of a given number and Listing 3 shows a test suite designed to check this method. Code coverage is helpful to notice that the test case in Line 2 executes all instructions of the method body but the return instruction in Line 3, which means that the corner case where the input is 0 is not being tested. Therefore, the test case in Line 8 is needed to achieve a full coverage and execute the corner case.

```
    long fact(int n) {
 2    if(n==0)
        return 1;
 4    long result=1;
      for(int i=2; i<=n; i++)
 6      result=result*i;
      return result;
 8  }
```

Listing 2: A method to compute the factorial of a given number

```
    @Test
 2  factorialWith5Test() {
      long obs = fact(5);
 4    assertTrue(5 < obs);
    }
 6
    @Test
 8  factorialWith0Test() {
      assertEqual(1, fact(0));
10  }
```

Listing 3: A test suite to check the factorial implementation

But a high percentage of code coverage does not necessarily means that the test suite is effective. It can be expected that, in a well tested codebase, the test suite achieves a high coverage but the opposite is not true in general. In an extreme case, all the assertions included in a test suite could be removed and then the test suite would be able to produce the same coverage as before without actually verifying anything. Moreover, achieving a 100% coverage is an unrealistic goal that can lead to an important waste of resources and efforts and actually not needed in the general case.

2.1 The XWiki experience

The XWiki Project[4] builds a Java platform for developing collaborative applications using the wiki paradigm. Its main codebase if composed of 3 Maven multi-module projects with more than 40 submodules each. As an example, the `xwiki-rendering` has 37571 LOCs in the main codebase and 9276 LOCs in their test code, implementing 2247 test cases. They have a very solid testing practice that combines custom JUnit test runners, and build profiles dedicated only to check the quality of their products. The development process is monitored in a Continuous Integration fashion, using a Jenkins instance[5].

[4] http://www.xwiki.org/
[5] https://ci.xwiki.org/

One of the jobs devoted to check the quality of the project monitors code coverage. Each module in the codebase has a predefined threshold and the code coverage can not decrease below this value, otherwise the build will fail. In this way, if a developer adds some code she has to also provide new tests cases so the coverage ratio remains above or equal the predefined value. If a developer achieves a coverage above threshold, then she is given the possibility to raise the value for the module. In this way it is ensured that the code coverage never decreases and this is what they call the *Ratchet Effect*. This strategy has led to an effective use of the code coverage metric. They report an increase from 74.07% to 76.28% of code coverage in little less than 11 months[6].

3 Mutation testing

DeMillo et al. [5] proposed a different criterion to evaluate a test suite known as *mutation testing*, *mutation analysis* or originally *program mutation*. It consists on introducing subtle faults in the program under test in the form of common programming errors and then verify if the test suite is able to detect the planted changes. Each program variant created after introducing a fault is called a *mutant*. A mutant is said to be *live* if it is not detected by the test suite otherwise it is said to be *killed*.

Mutation testing is based on two main assumptions:

- Programmers create programs that are close to being correct. That is, competent programmers make small mistakes. (*The competent programmer hypothesis*)
- A test suite that detects all simple faults can detect most complex faults. That is, complex errors are coupled to simple errors. (*The coupling effect*)

The model of faults that are introduced in a program, often called as *mutation operators*, are designed according to these two assumptions and target the most common mistakes that programmers tend to make. Typical mutation operators would change a comparison operator by other, change an arithmetic operator, slightly alter the result of a method or change a constant value. Listing Listing 4 shows two examples of mutants that could be created for the method in Listing Listing 2. Notice the change of the relational operator in Line 3 and how the returning value in Line 18 is altered by adding 1. Notice that the first mutant is detected by the first test case included in Listing 3 while the second is a live mutant as it is not detected by any of the two test cases in the same Listing. This live mutant in fact, can be useful to see that those test cases are not effective enough, in particular the assertion of the first test case.

```
   //Mutant 1
2  long fact(int n) {
     if(n!=0)
4      return 1;
     long result=1;
```

[6] https://massol.myxwiki.org/xwiki/bin/view/Blog/ComparingCloverReports

```
   Data: P, TS, Ops
   Result: score, live
 1 M ← generateMutants(P, TS, Ops)
 2 foreach m ∈ M do
 3 |  run(TS, m)
 4 |  if one − test − fails then
 5 |  |  killed ← m
 6 |  end
 7 |  else
 8 |  |  live ← m
 9 |  end
10 end
11 score ← killed / |M|
12 return score, live
```

Algorithm 1: Mutation analysis

```java
  for(int i=2; i<=n; i++)
    result=result*i;
  return result;
}

//Mutant 2
long fact(int n) {
  if(n==0)
    return 1;
  long result=1;
  for(int i=2; i<=n; i++)
    result=result*i;
  return result+1;
}
```

Listing 4: Examples of mutants for the *fact* method

The ratio of detected or *killed* mutants to the total number of mutants created for a program is named *mutation score*. This ratio is often used as a quantitative measure to compare test suites and as a proxy to the fault detection capabilities of a test suite. The greater the mutation score, the more faults shall the test suite detect.

Algorithm 1 shows the general implementation of the mutation analysis. Every mutant is analyzed in isolation (line 3). The result of one mutant is not expected to affect the outcome of another. This is not always a guarantee in practice. Both, the live mutants and the mutation score are the expected results (line 12).

There are several tools available that provide effective implementations of the mutation testing process. In the Java world the most popular alternative is PIT. PIT or PITest[7] targets Java projects and implements most traditional mutation operators. The tool performs all transformations over the compiled bytecode. It also implements some strategies for test prioritization, test selection

[7] `http://pitest.org`

and parallel test execution to speed up the process. PIT integrates with major build systems such as Ant, Maven and Gradle. The default functionalities of this tools can be extended via plugins. Other tools available are Javalanche[8] which also manipulates bytecode and Major[9] that operates at the source code level, is integrated with the Java compiler and provides a mechanism to define new mutation operators.

3.1 Limitations of mutation testing

Mutation analysis is a simple, yet effective, idea. However it hasn't been widely adopted by industry practitioners despite the decades of research invested in the subject. The three main reasons often used against mutation testing[11, 10] are:

- The cost of the analysis. The number of mutants that can be created is huge even for simple programs making the analysis time consuming and prohibitively expensive in terms of computation budget in some cases.
- The presence of equivalent mutants. The mutation operators may create program variants which are equivalent to the original code and thus indistinguishable from live mutants. The automatic detection of equivalent mutants is, in general and undecidable problem.
- The lack of integrated and production ready tools. Even when there are several practical alternatives, most of them are created for academic purposes. PIT is one of the few that has been created with an industry exploitation mentality.

The work of Gopinath et al. [8, 7] studies the limits of the two assumptions on which mutation testing is based. These authors investigated a total of 240000 bug fixes across 5000 programs written in four different programming languages [8]. They concluded that a significant number of changes are larger than the ones created by traditional mutation operators, which suggests that, in this sense, real faults are different from mutants. They also observe that there are differences in the patterns of changes among different programming languages and that the mutation analysis also exhibits differences in this aspect. This fact was also observed in practice by Petrovic and Ivankovic [14].

Kurtz et al. state that the mutation score is affected by the presence of equivalent mutants and redundant mutants, that is mutants that are killed by the same test cases that detect others.

3.2 Overcoming the limitations

The specialized literature gathers an important set of works directed to overcome the problems of mutation testing.

[8] `https://www.st.cs.uni-saarland.de/mutation/`
[9] `http://mutation-testing.org/`

Most proposals try to make mutation analysis more efficient. Untch [17] states that these works mainly follow three strategies:

- *do fewer* these approaches "try to run fewer mutated program- s without incurring intolerable loss in effectiveness".
- *do smarter* which "distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs".
- *do faster* these "try to generate and run mutant programs more quickly".

Works following the *do faster* and *do smarter* strategy propose to integrate mutation operators in the compilation process to speed up mutation creation [4], or propose a cloud infrastructure to distribute the analysis and make it faster [3, 16]. Tools like PIT execute for each mutant only the tests that cover the change. PIT also sorts the tests from so the closer to change are run first and provides the possibility to execute tests concurrently.

A notable set of works has been devoted to reduce the number of mutants in the analysis, (the *do fewer* approach). Some authors propose to randomly sample the mutants to be used [2, 1, 18]. Other authors propose to use only a subset of mutation operators [13]. Another subset of works explore the trade-offs of mutant sampling and operator-based selection [19, 21, 20]. The use of higher order mutants, that combine several first order traditional mutants, has been explored as well as a way to reduce the execution time [15] and deal with equivalent mutants [9].

Untch [17] proposed to use statement deletion operators. It shows a drastic decrease on the number of mutants while maintaining the accuracy. The idea was expanded by Deng et al. [6] and Delamaro et al. [**?**] to additional programming languages and the deletion of bloks variables operators and constants.

3.3 The Google experience

Petrovic and Ivankovic [14] have recently described the use of mutation analysis in the Google code base. They explain that the Google repository contains about two billion lines of code and on average, 40000 changes are commited every workday and 60% of them are created by automated systems. In this environment it is not practical to compute a mutation score for the entire codebase and very hard to provide an actionable feedback.

Since most changes pass through a code review process, the authors argue that this is the best location in the workflow to provide feedback about the mutation analysis and eliminate the need for developers to run a separated program and act upon its output. So live mutants are shown as *code findings* in code reviews.

To make the mutation analysis feasible the proposed system creates at most one mutant by covered line. The mutation operator is selected at random from a set of available operators. To further reduce the number of mutants, they classify each node of the Abstract Syntax Tree (AST) as important or non-important (*arid*). To do this, they maintain a curated collection of simple AST nodes classified by experts, that keeps updating with the feedback of the reviewing process. Compound nodes are classified as arid if all their children are arid. Uninteresting nodes may be related to logging, non-functional properties and nodes seen as "axiomatic" for the language and thus the mutants are trivially killed. This selection may suppress relevant live mutants but the authors state that the tradeoff between correctness and usability of the system is good, as the number of potential mutants is always much larger than what can be presented to reviewers.

The system analyses programs written in C++, Java, Python, Go, JavaScript, TypeScript and Common Lisp. It has been validated with more than 1M mutants in more than 70K diffs. 150K live mutants were presented and 11K received feedback. 75% of the findings with feedback were reported to be useful. The authors also observed interesting differences related to the survival ratio of mutants when contrasted with the programming language and mutation operator.

4 Extreme mutation, pseudo-tested methods and Descartes

Niedermayr et al. [12] recently introduced *extreme mutation* analysis, It is an alternative to traditional mutation that performs more coarse-grained transformations by eliminating, at once, all side effects of a method. For a `void` method this approach removes all instructions from its body. If the method is not `void`, then the body is replaced by a single `return` instruction with a predefined value.

Listing Listing 5 shows two extreme mutants that could be created for the method in Listing 2.

```
//Extreme mutant 1
long factorial(int n) {
    return 0;
}

//Extreme mutant 2
long factorial(int n) {
    return 1;
}
```

Listing 5: Two mutans created with extreme mutation.

Extreme mutation creates much less mutants than the traditional approach and can automatically avoid most transformations that could be equivalent to the original code. Another benefit of this technique comes from operating at the method level. This which eases the understanding of the underlying testing problem.

In addition to the mutation score, extreme mutation pinpoints a list of worst tested methods. In particular, the technique higligths methods executed by the test suite but where no extreme mutant is detected while running the tests. These methods are labeled as *pseudo-tested* in the work of Niedermayr et.

al.[12]. These authors report having found pseudo-tested methods in all the 14 projects they have studied, as result we have replicated with our own tooling.

Listing 6 shows a class and a test class with one test case. In the absence of more test cases, the `incrementVersion` method on Line 9 is pseudo-tested, as its effects are never assessed. This is a common scenario in which this type of methods appear.

```
1   class VList {
        private List elements;
3       private int version;
        public void add(Object item) {
5           elements.add(item);
            incrementVersion();
7       }

9       private void incrementVersion() {
            version++;
11      }

13      public int size() {
            return elements.size();
15      }
    }
17
    class VListTest {
19      @Test
        public void testAdd() {
21          VList l = new VList();
            l.add(1);
23          assertEquals(l.size(), 1);
        }
25  }
```

Listing 6: Example of a pseudo-tested method

4.1 Descartes

Descartes is a tool that implements extreme mutation and automatically detects pseudo-tested methods. It has been conceived as a *mutation engine* plugin for PIT. In PIT's jargon, a mutation engine is a plugin that handles the discovery and creation of mutants. The rest of the tool's framework deals with the project structure, test discovery and execution. Figure 1 captures the interrelation between PIT and Descartes. By piggybacking on PIT, the tool is able to target Java programs being built with Ant, Gradle or Maven and using JUnit or TestNG.

The tool can target most Java methods except constructors. It can be configured with the constant literals that will be used as return values for the methods analyzed. All Java primitive types and `String` are supported. Reference types are targeted using the `null` value and there is a special operator to return an empty array where possible. Descartes also includes mechanisms to avoid methods that could be not interesting based on their structure. It can skip for example, simple getters and setters, receiving methods in delegation patterns, deprecated methods and empty `void` methods. By default the tool uses the values and operators shown in Table 1.

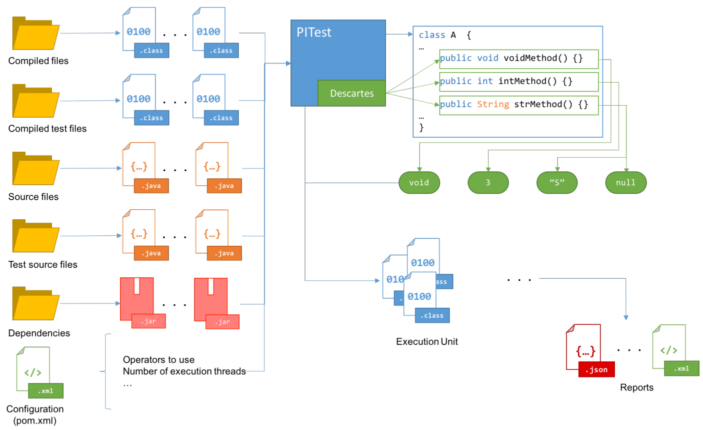

Figure 1: Interconnection between PIT and Descartes.

Table 1: Extreme mutation operators used in the comparison.

Method type	Transformations
void	Empties the method
Reference types	Returns null
boolean	Returns true or false
byte,short,int,long	Returns 0 or 1
float,double	Returns 0.0 or 0.1
char	Returns ‘ ’ or ‘A’
String	Returns “” or “A”
T[]	Returns new T[]{}

Table 2: List of projects used to compare both engines, the execution time for the analysis, the number of mutants created, mutants covered and placed in methods targeted by both tools, mutants killed and the mutation score.

	Descartes					Gregor				
Project	Time	Created	Covered	Killed	Score	Time	Created	Covered	Killed	Score
AuthZForce PDP Core	0:08:00	626	378	358	94.71	1:23:50	7296	3536	3188	90.16
Amazon Web Services SDK	1:32:23	161758	3090	2732	88.41	6:11:22	2141689	17406	13536	77.77
Apache Commons CLI	0:00:13	271	256	246	96.09	0:01:26	2560	2455	2183	88.92
Apache Commons Codec	0:02:02	979	912	875	95.94	0:07:57	9233	8687	7765	89.39
Apache Commons Collections	0:01:41	3558	1556	1463	94.02	0:05:41	20394	8144	7073	86.85
Apache Commons IO	0:02:16	1164	1035	968	93.53	0:12:48	8809	7633	6500	85.16
Apache Commons Lang	0:02:07	3872	3261	3135	96.14	0:21:02	30361	25431	22120	86.98
Apache Flink	0:14:04	4935	2781	2373	85.33	2:29:45	43619	21350	16647	77.97
Google Gson	0:01:08	848	657	617	93.91	0:05:34	7353	6179	5079	82.20
Jaxen XPath Engine	0:01:31	1252	953	921	96.64	0:24:40	12210	9002	6041	67.11
JFreeChart	0:05:48	7210	4686	3775	80.56	0:41:28	89592	47305	28080	59.36
Java Git	1:30:08	7152	5007	4507	90.01	16:02:03	78316	54441	40756	74.86
Joda-Time	0:03:39	4525	3996	3827	95.77	0:16:32	31233	26443	21911	82.86
JOpt Simple	0:00:37	412	397	379	95.47	0:01:36	2271	2136	2000	93.63
jsoup	0:02:43	1566	1248	1197	95.91	0:12:49	14054	11092	8771	79.08
SAT4J Core	0:53:09	2304	804	617	76.74	10:55:50	17163	7945	5489	69.09
Apache PdfBox	0:44:07	7559	3185	2548	80.00	6:20:25	79763	32753	20226	61.75
SCIFIO	0:24:14	3627	1235	1158	93.77	3:12:11	62768	19615	9496	48.41
Spoon	2:24:55	4713	3452	3171	91.86	56:47:57	43916	34694	27519	79.32
Urban Airship Client Library	0:07:25	3082	2362	2242	94.92	0:11:31	17345	11015	8956	81.31
XWiki Rendering Engine	0:10:56	5534	3099	2594	83.70	2:07:19	112605	50472	26292	52.09

4.2 Experiments

We have compared Descartes with Gregor, the default mutation engine for PIT in a set of 21 open source projects. These are all projects that use Maven as main build system, JUnit as main testing framework and are available form a version control hosting service, mostly Github. In this comparison Gregor and Descartes both used their set of default mutation operators.

Table 2 shows, for each project and mutation engine, the number fo mutants created, covered by the test suite and killed. It also shows the raw mutation score and the time required to complete the analysis. Figure 2 shows the relative proportion of mutants created by Descartes with respect to the mutants created by Gregor. The same relative relation is shown with respect to time in Figure 3. It can be seen that Descartes creates less than 20% of the number of mutants created by Gregor and takes less than 40% is all projects but one. In Figure 4 it can be noticed that the raw scores are somehow correlated. In fact Spearman correlation coefficient results in 0.6 for the projects studied with a *p-value* of 0.003, which indeed indicates that there is a moderate positive correlation.

4.3 Real examples of pseudo-tested methods

During the experiments we have conducted we have also found that all inspected projects have pseudo-tested which confirms the findings of Niedermayr et al.. Table 3 shows the number for each project.

Now we present four cases of pseudo-tested methods found in four different projects with the help of Descartes.

Listing 7 shows an example found in *Apache Commons Codec*. The test case

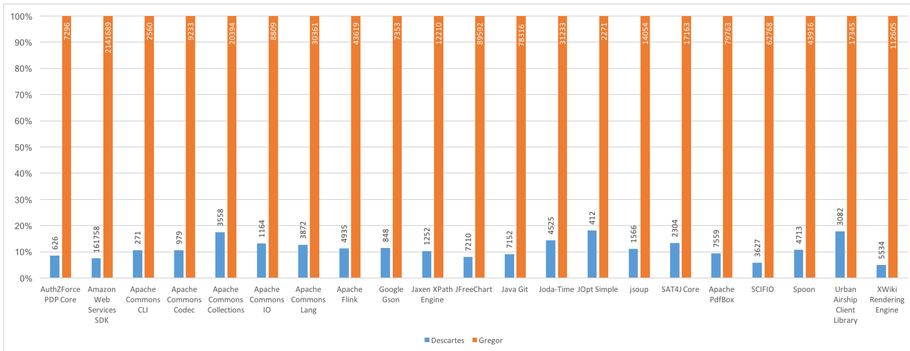

Figure 2: Relative gain in the number of mutants of Descartes with respect to Gregor.

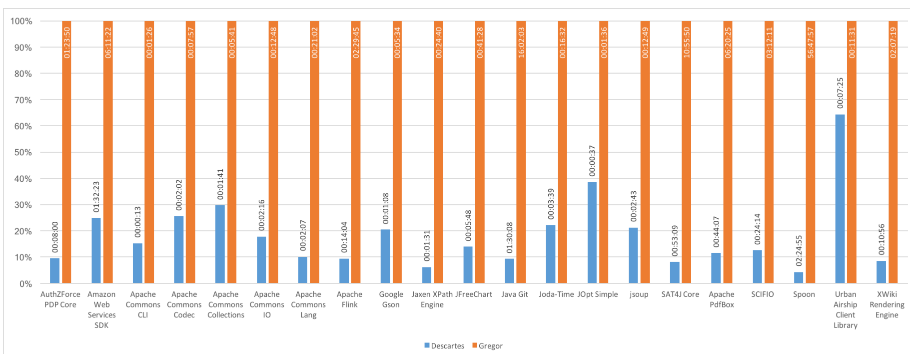

Figure 3: Relative gain in time of mutants of Descartes with respect to Gregor.

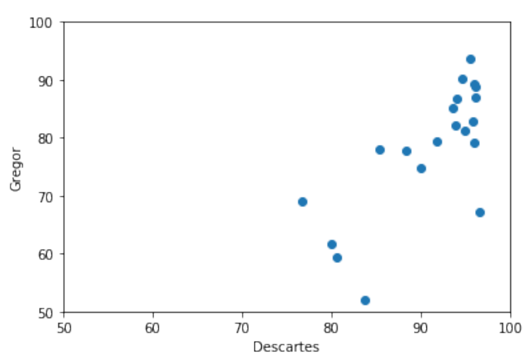

Figure 4: Visual correlation between the raw scores produced by both tools.

Table 3: Number of pseudo-tested methods found on each project studied.

Project	Pseudo-tested methods
AuthZForce PDP Core	13
Amazon Web Services SDK	224
Apache Commons CLI	2
Apache Commons Codec	12
Apache Commons Collections	40
Apache Commons IO	29
Apache Commons Lang	47
Apache Flink	100
Google Gson	10
Jaxen XPath Engine	11
JFreeChart	476
Java Git	296
Joda-Time	82
JOpt Simple	2
jsoup	28
SAT4J Core	143
Apache PdfBox	473
SCIFIO	72
Spoon	213
Urban Airship Client Library	28
XWiki Rendering Engine	239

actually has no assertion so `isEncodeEqual` is pseudo-tested. After placing the assertion, it was discovered that the input in Line 3 was wrong.

```
1       public void testIsEncodeEquals() {
            final String[][] data = {
3               {"Meyer", "M\u00fcller"},
                {"Meyer", "Mayr"},
5               ...
                {"Miyagi", "Miyako"}
7           };
            for (final String[] element : data) {
9                   final boolean encodeEqual =
                this.getStringEncoder().isEncodeEqual(element[1], element
                    [0]);
11       }
        }
```

Listing 7: Covering test case with no assertion.

Listing 8 shows an example found in *Apache Commons IO*. The `write` method invoked in Line 6 is pseudo-tested. If this method is emptied, the output of both streams `baos1` and `baos2` is empty, and therefore the same, so the test case does not fail.

```
        public void testTee() {
2           ByteArrayOutputStream baos1 = new ByteArrayOutputStream();
            ByteArrayOutputStream baos2 = new ByteArrayOutputStream();
4           TeeOutputStream tos = new TeeOutputStream(baos1, baos2);
            ...
6           tos.write(array);
            assertByteArrayEquals(baos1.toByteArray(), baos2.toByteArray());

8       }
```

Listing 8: Test case verifying TeeOutputStream write methods.

Listing 9 shows an example found in *Apache Commons Collections*. The `add` method represents a non-supported operation for `SingletonListIterator` instances. But, if the body of the method is removed, the test case in Line 14 passes anyways. A `fail` invocation is needed after Line 20 to solve the situation.

```
        class SingletonListIterator
2         implements Iterator<Node> {
          ...
4         void add() {
            //This method was found to be pseudo-tested
6           throw new UnsupportedOperationException();
          }
8         ...
        }
10
        class SingletonListIteratorTest {
12        ...
          @Test
14        void testAdd() {
            SingletonListIterator it = ...;
16          ...
            try {
18            //If the method is emptied, then nothing happens
              //and the test passes.
20            it.add(value);
            } catch(Exception ex) {}
22        ...
```

```
}
```

Listing 9: Class containing the pseudo-tested method and the covering test class.

Listing 10 shows an example found in *Amazon Web Services Java SDK*. The `prepareSocket` method is pseudo-tested, as it calls `setEnabledProtocols` (Line 4) and the assertion is placed inside this second method (Line 17), then when `prepareSocket` is emptied, the condition is never verified and the test passes.

```
1  class SdkTLSSocketFactory {
     protected void prepareSocket(SSLSocket s) {
3        ...
         s.setEnabledProtocols(protocols);
5        ...
     }
7  }

9  @Test
   void typical() {
11     SdkTLSSocketFactory f = ...;
       //prepareSocket was found to be pseudo-tested
13     f.prepareSocket(new TestSSLSocket() {
           ...
15         @Override
           public void setEnabledProtocols(String[] protocols) {
17           assertTrue(Arrays.equals(protocols, expected));
           }
19     ...
       });
21 }
```

Listing 10: A weak test case for method prepareSocket.

The nature of these methods can be understood even for outsiders to these projects. To the point that we were able to explain the issue to their development teams and propose pull request which were all accepted [10]

4.4 Partially-tested methods

In our experiments we have also seen that methods with mixed results, that is, with live and killed extreme mutants at the same time, often point to testing issues. We call these methods partially-tested methods. Listing 11 shows a simplified extract of a real case we have found. The `equals` method in Line 8 is partially-tested. If the body is changed by `return false`, the change is detected but `return true` passes. The condition of the assertion in Line 20 is always `false` in Java, so it is a mistake made by the developer whose intention was to test the inequality.

```
1  public class AClass {
     private int aField = 0;
3
     public AClass(int field) {
```

[10] https://github.com/apache/commons-codec/pull/13, https://github.com/apache/commons-io/pull/61, https://github.com/apache/commons-collections/pull/36, https://github.com/aws/aws-sdk-java/pull/1437

```
    aField = field;
    }

    public bool equals(object other) {
      return other instanceof AClass && ((AClass) other).aField == aField;
     }
    }

    public class ACLassTest {
      @Test
      public void test() {
        AClass a = new AClass(3);
        AClass b = new AClass(3);
        AClass c = new AClass(4);
        assertTrue(a.equals(b));
        assertFalse(a == b);
    }
  }
```

Listing 11: Example of a partially-tested method.

Descartes also reports these methods.

4.5 Taking Descartes to the CI

There are several alternatives to bring Descartes to a Continuous Integration environment. The XWiki project, for example, has implemented a strategy similar to the one described in Section 2.1. The threshold is set for the mutation score computed by Descartes for each module. By inspecting the methods reported by the tools, the developers have been able to improve their test code impacting more than 20 test classes between modifications and additions. They also report increments in code coverage between 1% to 3% and between 1% and 7% in mutation score [11]. In a peculiar case, they were able to spot a functionality that could be simplified. Figure 5 shows an example of the output of this plugin.

Descartes could be used in the CI to monitor the methods in the project. A Jenkins plugin[12] has been created. This plugin reports the number of methods that are covered by the test suite and the number of pseudo-tested method in the codebase. It can be used to monitor both numbers as the project evolves.

A third alternative focuses on analyzing pull requests in Github. We have built a Github App[13] that leverages the Github check run API[14]. When a pull request is made to a repository where the application is installed, Github will send a notification to our CI server with the required information to execute the analysis. The main goal is to plant mutants only in the changed code so the analysis can be faster. The pseudo-tested methods are shown inline in the pull request report. Figure 6 shows examples of the output of this application as presented to developers in the Github website.

[11] `https://github.com/STAMP-project/descartes-usecases-output/tree/master/xwiki`
[12] https://github.com/STAMP-project/jenkins-stamp-report-plugin
[13]
[14] `https://developer.github.com/v3/checks/runs/`

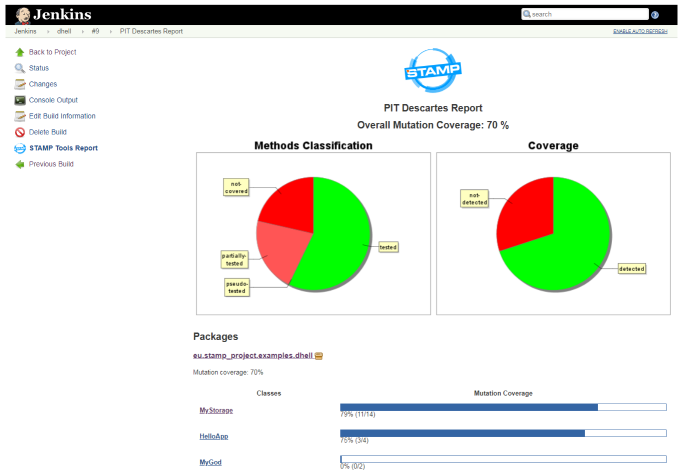

Figure 5: Output of the Jenkins plugin that uses Descartes to monitor pseudo-tested methods.

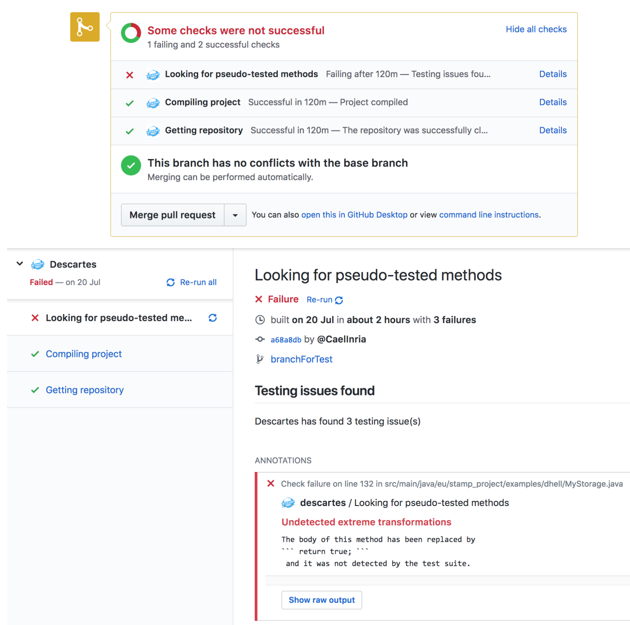

Figure 6: Descartes Github Application using the check run API.

References

[1] Allen Troy Acree, Jr. *On Mutation*. PhD Thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1980.

[2] Timothy Alan Budd. *Mutation Analysis of Program Test Data*. PhD Thesis, Yale University, New Haven, CT, USA, 1980.

[3] Pablo C. Cañizares, Alberto Núñez, and Juan de Lara. OUTRIDER: Optimizing the mUtation Testing pRocess In Distributed EnviRonments. *Procedia Computer Science*, 108:505–514, 2017.

[4] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-integrated program mutation. In *[1991] Proceedings The Fifteenth Annual International Computer Software Applications Conference*, pages 351–356, September 1991.

[5] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer Magazine*, 11(4):34–41, April 1978.

[6] L. Deng, J. Offutt, and N. Li. Empirical Evaluation of the Statement Deletion Mutation Operator. In *Verification and Validation 2013 IEEE Sixth International Conference on Software Testing*, pages 84–93, March 2013.

[7] R. Gopinath, C. Jensen, and A. Groce. The Theory of Composite Faults. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 47–57, March 2017.

[8] Rahul Gopinath, Carlos Jensen, and Alex Groce. Mutations: How close are they to real faults? In *Software reliability engineering (ISSRE), 2014 IEEE 25th international symposium on*, pages 189–200. IEEE, 2014.

[9] Marinos Kintis, Mike Papadakis, and Nicos Malevris. Employing second-order mutation for isolating first-order equivalent mutants. *Software Testing, Verification and Reliability*, 25(5-7):508–535, August 2015.

[10] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, January 2014.

[11] Jakub Možucha and Bruno Rossi. Is Mutation Testing Ready to Be Adopted Industry-Wide? In *Product-Focused Software Process Improvement*, Lecture Notes in Computer Science, pages 217–232. Springer, Cham, November 2016.

[12] Rainer Niedermayr, Elmar Juergens, and Stefan Wagner. Will my tests tell me if I break this code? In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, pages 23–29, New York, NY, USA, 2016. ACM Press.

[13] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An Experimental Evaluation of Selective Mutation. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 100–107, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[14] Goran Petrovic and Marko Ivankovic. State of Mutation Testing at Google. page 9, 2018.

[15] Macario Polo, Mario Piattini, and Ignacio García-Rodríguez. Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification and Reliability*, 19(2):111–131, June 2009.

[16] Iman Saleh and Khaled Nagi. HadoopMutator: A Cloud-Based Mutation Testing Framework. In Ina Schaefer and Ioannis Stamelos, editors, *Software Reuse for Dynamic Systems in the Cloud and Beyond*, Lecture Notes in Computer Science, pages 172–187. Springer International Publishing, 2014.

[17] Roland H. Untch. On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator. In *Proceedings of the 47th Annual Southeast Regional Conference*, ACM-SE 47, pages 71:1–71:4, New York, NY, USA, 2009. ACM.

[18] Weichen Eric Wong. *On Mutation and Data Flow*. PhD Thesis, Purdue University, West Lafayette, IN, USA, 1993.

[19] Weichen Eric Wong and Aditya P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, December 1995.

[20] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 92–102, November 2013.

[21] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. Is Operator-based Mutant Selection Superior to Random Mutant Selection? In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 435–444, New York, NY, USA, 2010. ACM.