EVOSUITE

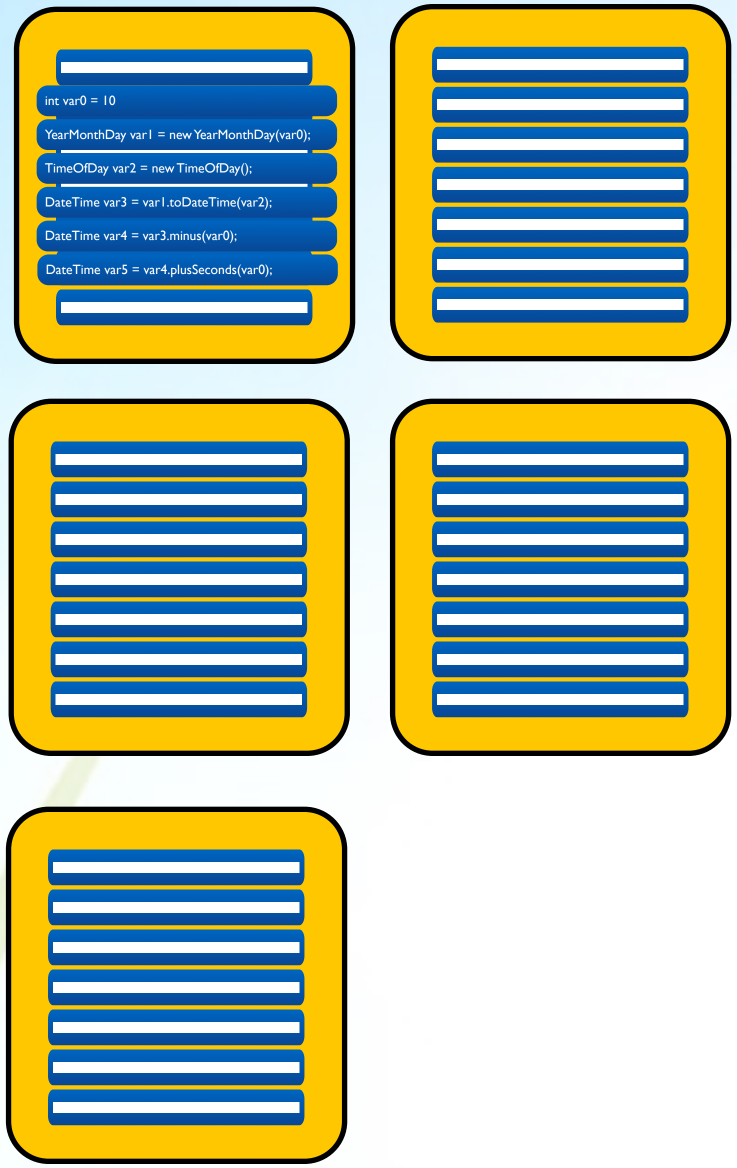
Gordon Fraser, José Miguel Rojas, José Campos, University of Sheffield

Andrea Arcuri, Westerdals Oslo ACT and University of Luxembourg

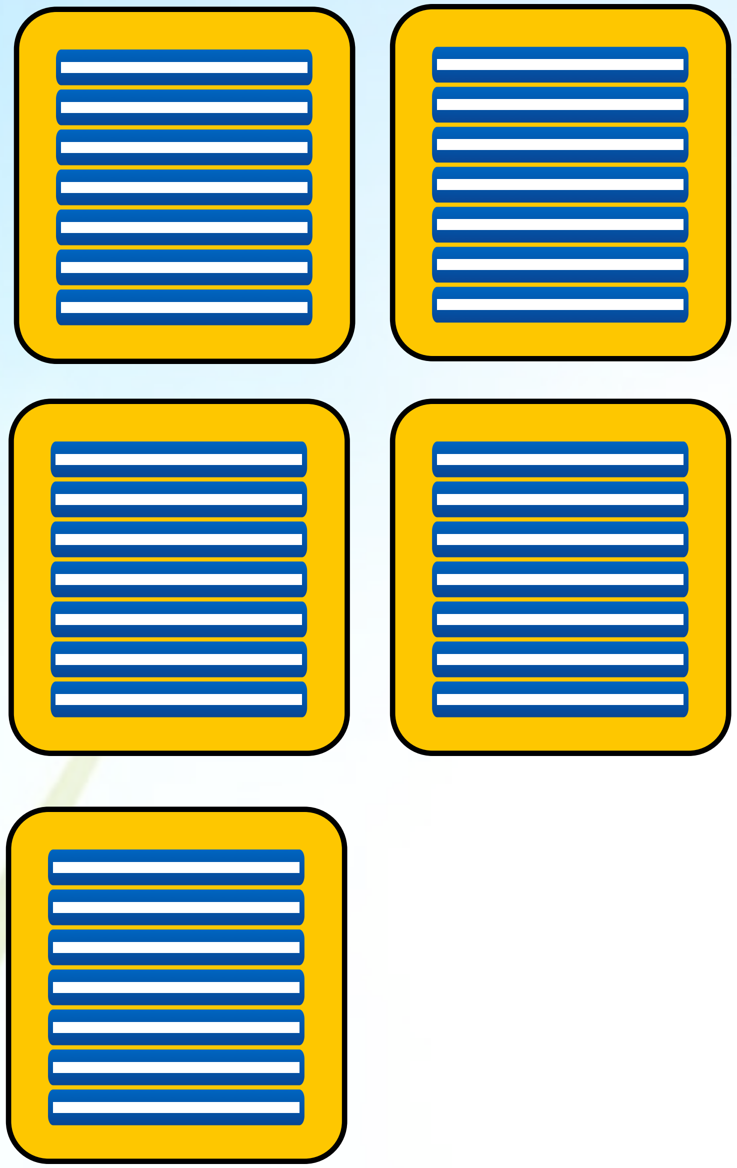
…and many others

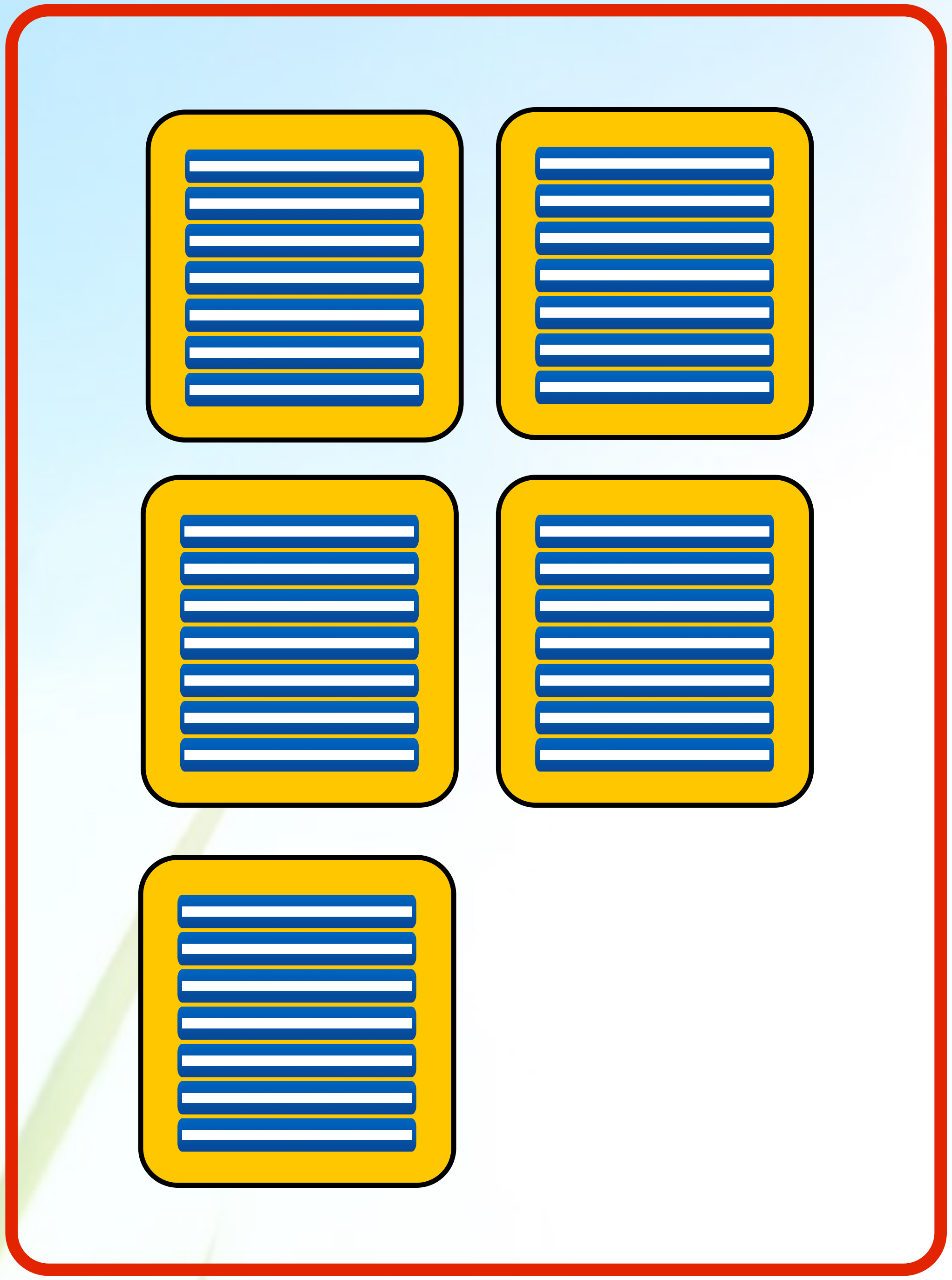
```
@Test

public void test()
{
    int x = 2;
    int y = 2;

    int result = x + y;

    assertEquals(4, result);

}
```

```
@Test
public void test()
{
    int var0 = 10
    YearMonthDay var1 = new YearMonthDay(var0);
    TimeOfDay var2 = new TimeOfDay();
    DateTime var3 = var1.toDateTime(var2);
    DateTime var4 = var3.minus(var0);
    DateTime var5 = var4.plusSeconds(var0);
}
```

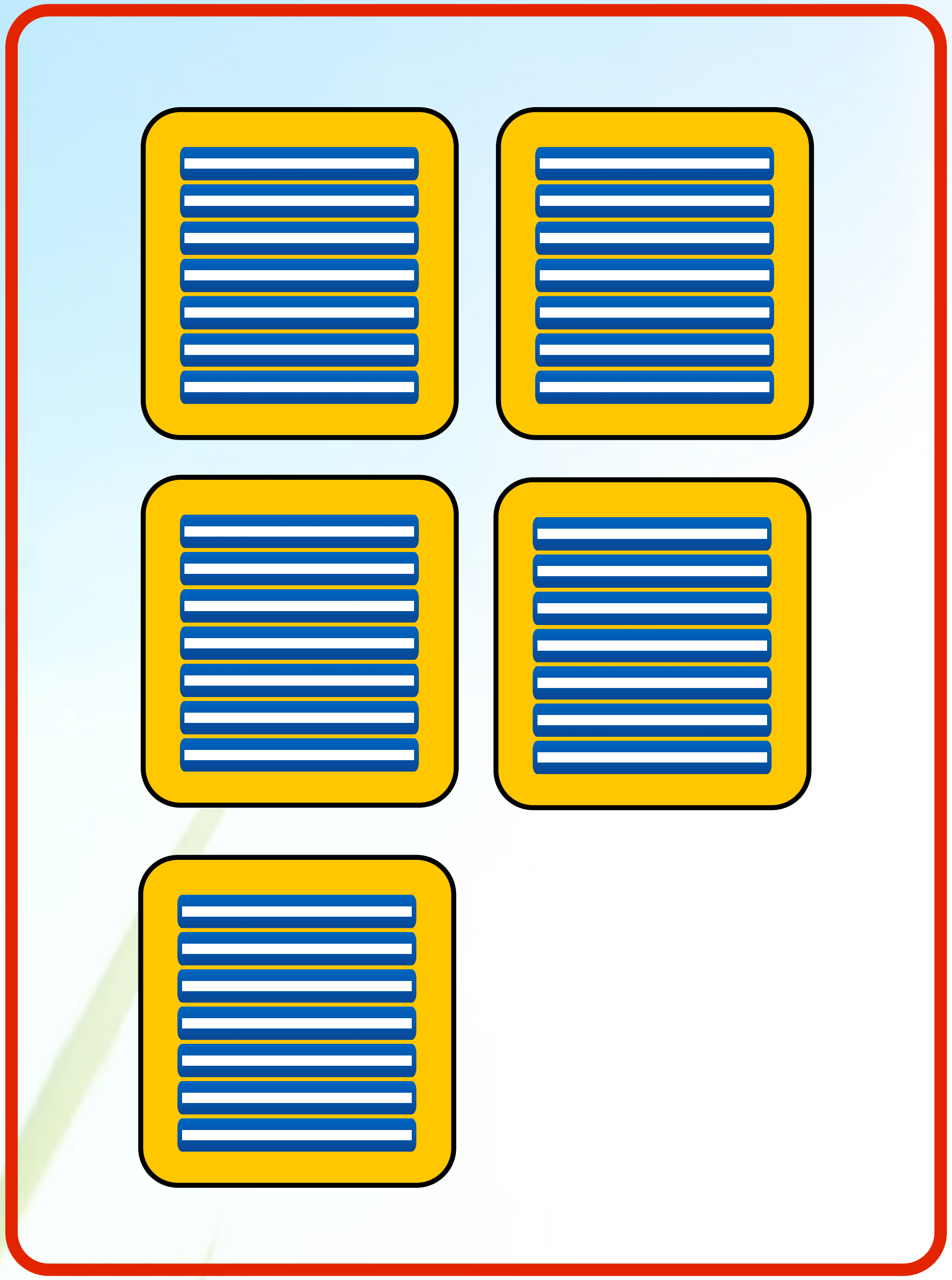


```
int var0 = 10
YearMonthDay var1 = new YearMonthDay(var0);
TimeOfDay var2 = new TimeOfDay();
DateTime var3 = var1.toDateTime(var2);
DateTime var4 = var3.minus(var0);
DateTime var5 = var4.plusSeconds(var0);
```



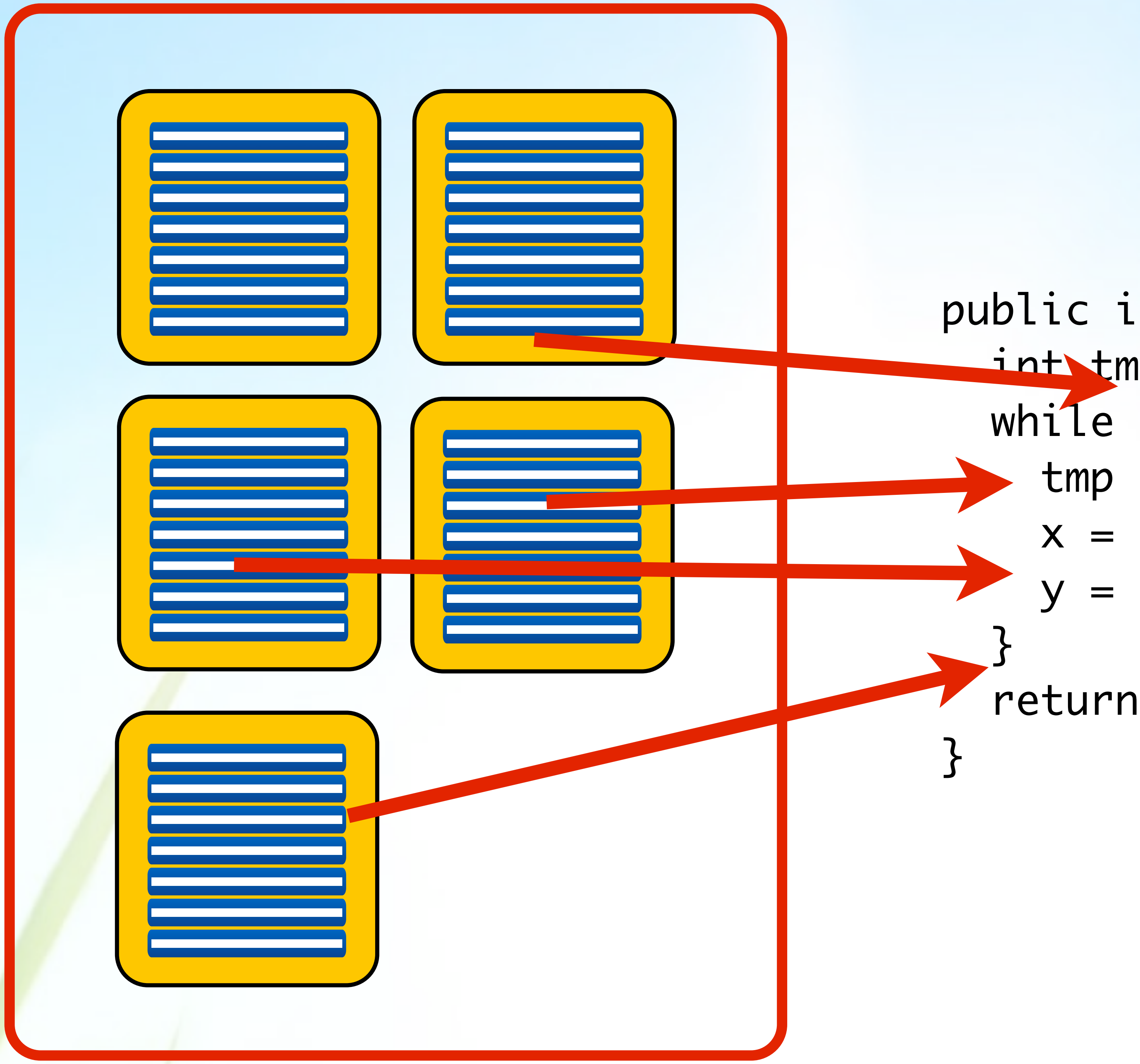


Fitness



```
public int gcd(int x, int y) {
  int tmp;
  while (y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }
  return x;
}
```

Fitness



```
public int gcd(int x, int y) {
  int tmp;
  while (y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }
  return x;
}
```

EVOSUITE

GA vs MOSA

- Why not using MOSA?

“Reformulating Branch Coverage as a Many-Objective Optimization Problem,” Panichella, Kifetew, and Tonella, ICST 2015.

- Various criteria

Line Coverage, Branch Coverage, Exceptions, Mutation testing, Method-Output, Top-Level Methods, No-Exception Top-Level Methods, Context Branch.

“Combining Multiple Coverage Criteria in Search-Based Unit Test Generation,” Rojas, Campos, Vivanti, Fraser and Arcuri, SSBSE 2015.

- DynaMOSA

“Automated Test Case Generation as a Many-Objective Optimization Problem with Dynamic Selection of the Targets,” Panichella, Kifetew and Tonella, TSE 2017.

Contributing Features

- Time budget management

  Search: 50% of time, Remaining phases (initialisation, minimisation, generating assertions, removing flaky tests, writing tests in disk): 10% of time each.

- Dynamic seeding of constant values

  “Seeding strategies in search-based unit test generation,” Rojas, Fraser and Arcuri, STVR, 2016.

- Test archive

  “A detailed investigation of the effectiveness of whole test suite generation,” Rojas, Vivanti, Arcuri, and Fraser, EMSE, 2017.

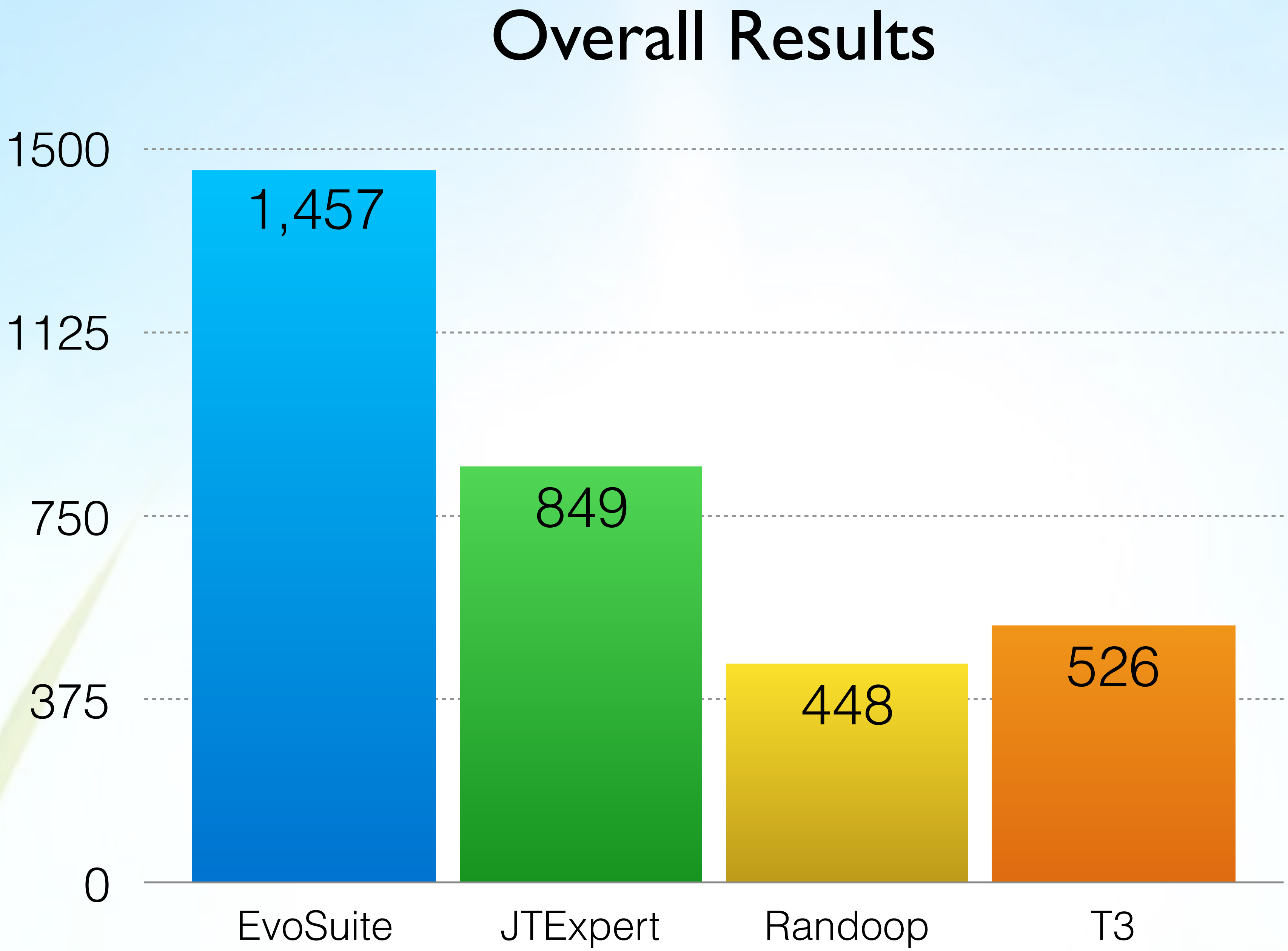
- Mocking objects, private fields access,…

SBST'17 Competition

Overall Results

EVOSUITE

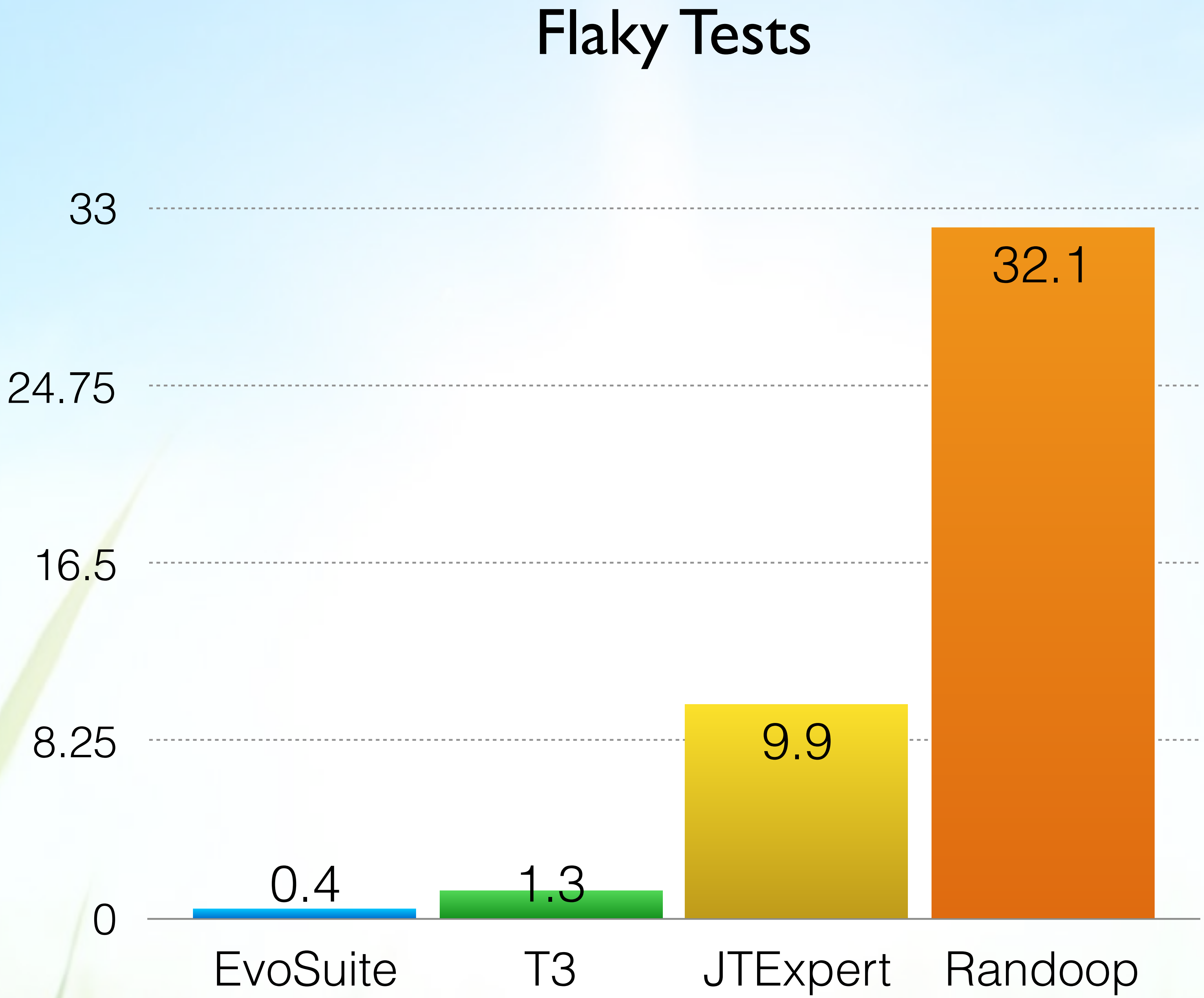
SBST'17 Competition



SBST'17 Competition

Flaky Tests

SBST'17 Competition

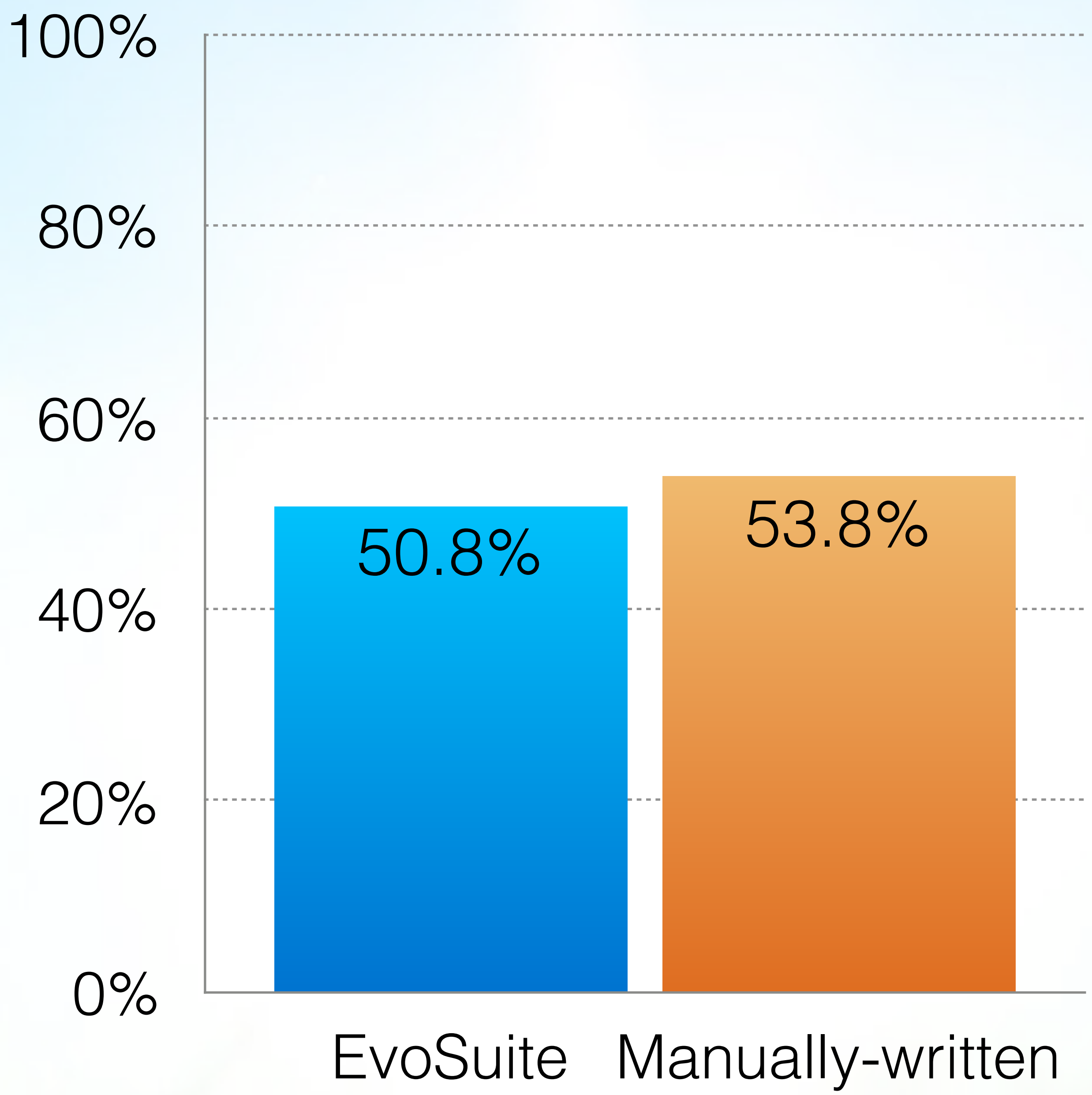


SBST'17 Competition

Branch Coverage vs Manually-written Tests

SBST'17 Competition

Branch Coverage vs Manually-written Tests



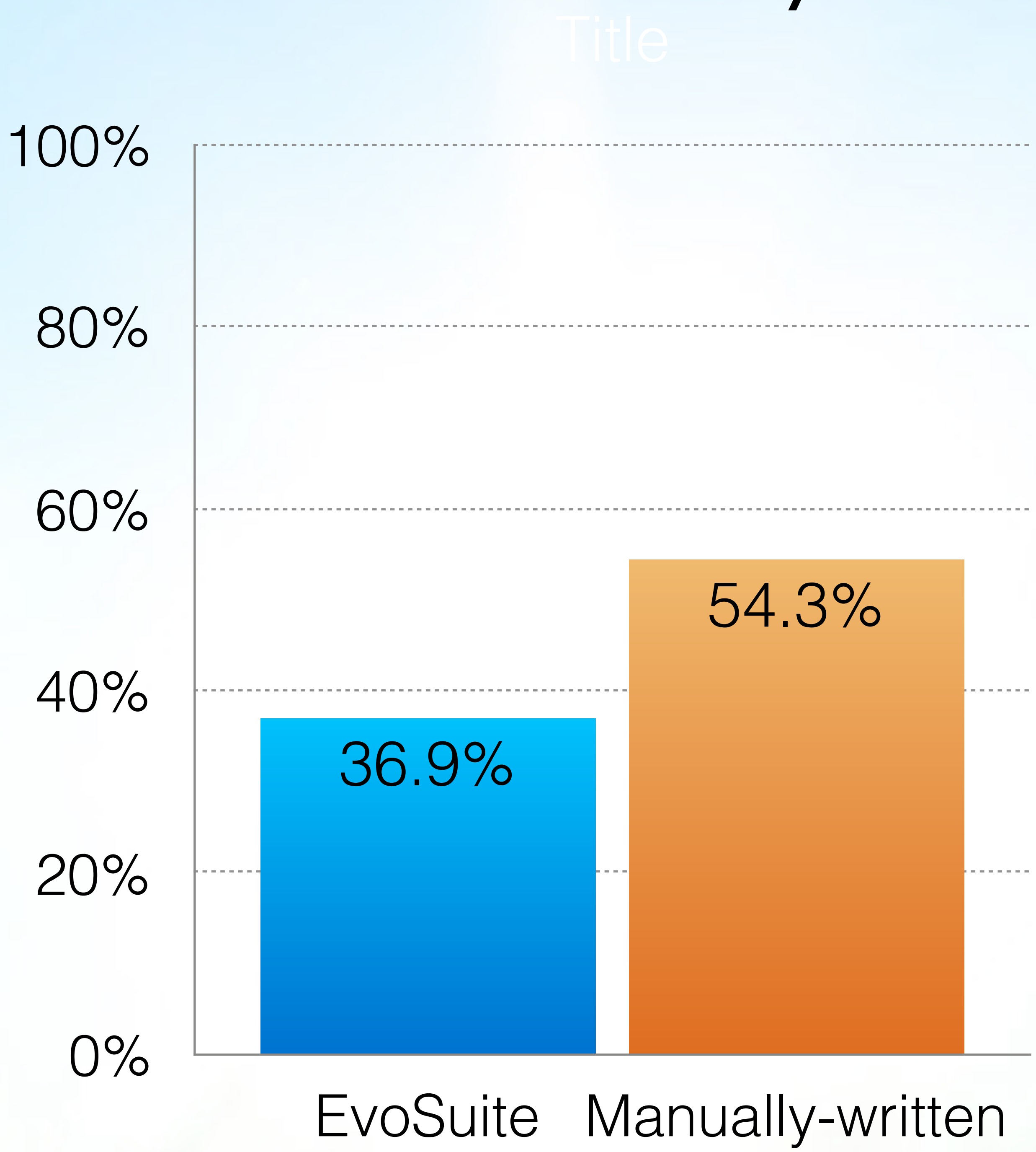
$A_{12} = 0.45$
pv $= 0.37$

SBST’17 Competition

Mutation Scores vs Manually-written Tests

SBST'17 Competition

Mutation Scores vs Manually-written Tests



$A_{12} = 0.31$
pv = <0.001

Test Readability

“Developers read test cases 77% of the total time they spend in them”

—“**When, How, and Why Developers (Do Not) Test in Their IDEs,”** Beller, Gousios, Panichella, and Zaidman, FSE 2015.

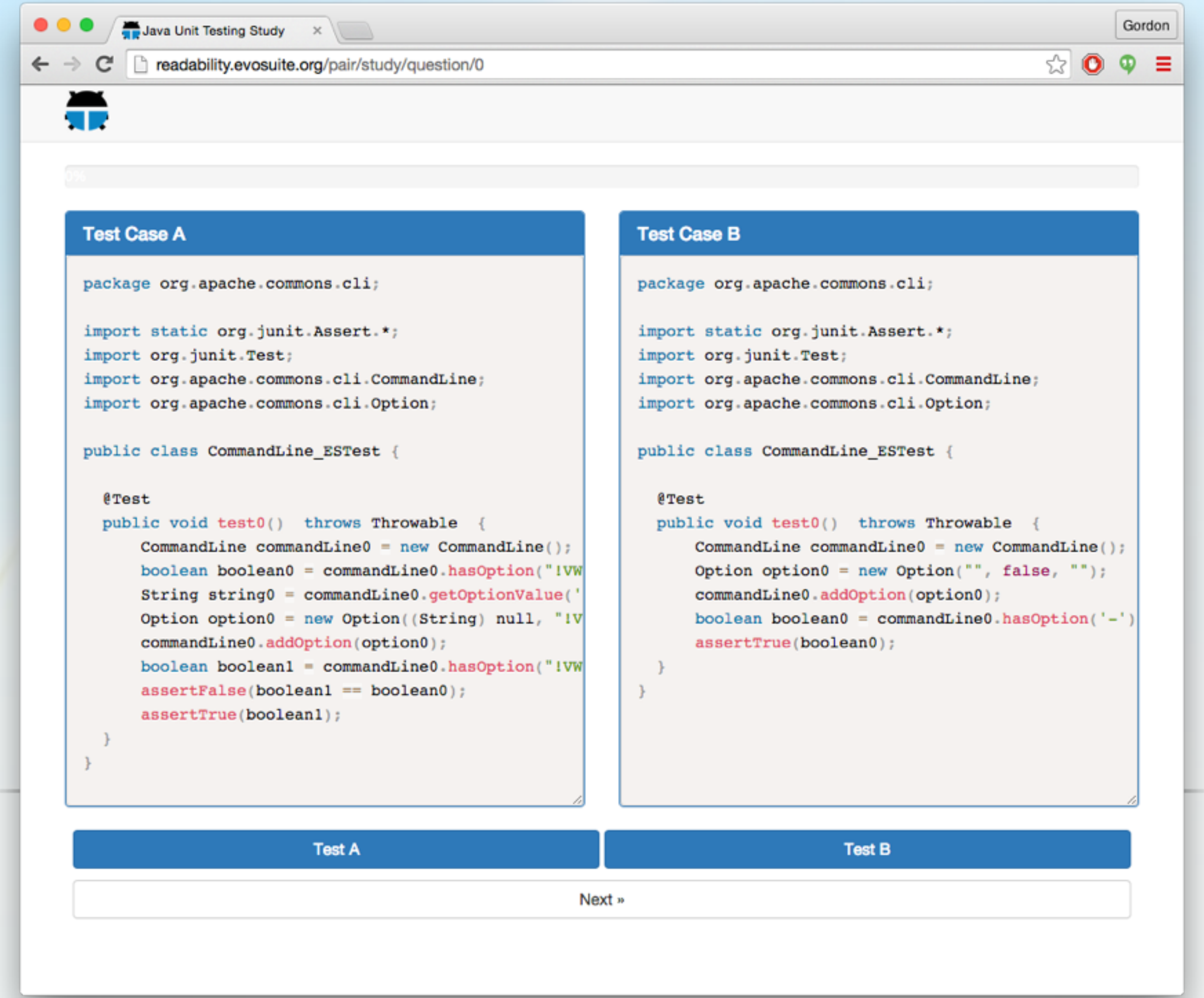
Why Test Readability Matters?

- Less human effort
- More understandable tests
- More maintainable tests
- More maintainable software

Software quality

EVOSUITE

What Makes a Test Readable?



Test Case A

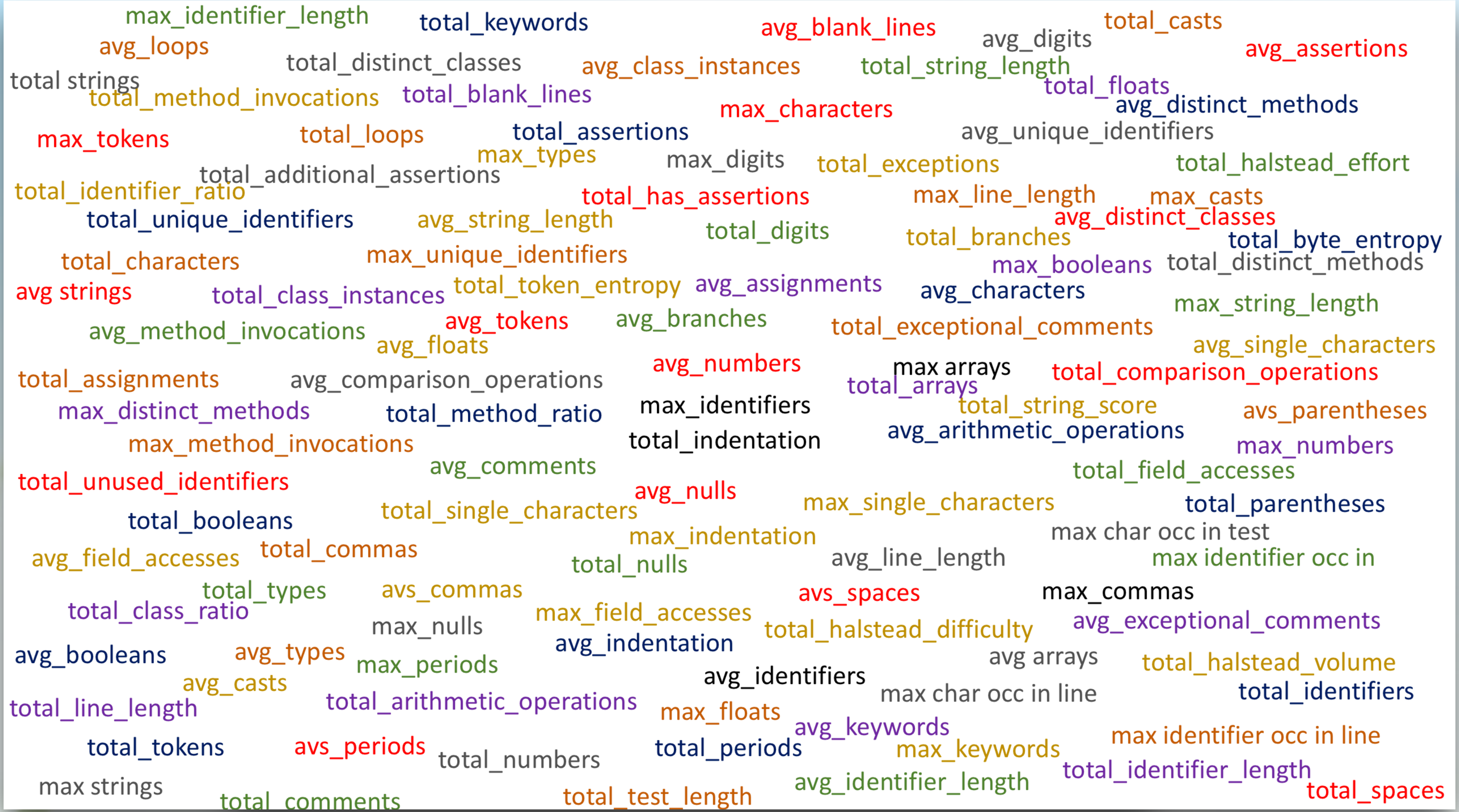
```java
package org.apache.commons.cli;

import static org.junit.Assert.*;
import org.junit.Test;
import org.apache.commons.cli.CommandLine;
import org.apache.commons.cli.Option;

public class CommandLine_ESTest {

  @Test
  public void test0()  throws Throwable  {
      CommandLine commandLine0 = new CommandLine();
      boolean boolean0 = commandLine0.hasOption("!VW
      String string0 = commandLine0.getOptionValue('
      Option option0 = new Option((String) null, "!V
      commandLine0.addOption(option0);
      boolean boolean1 = commandLine0.hasOption("!VW
      assertFalse(boolean1 == boolean0);
      assertTrue(boolean1);
  }
}
```

Test Case B

```java
package org.apache.commons.cli;

import static org.junit.Assert.*;
import org.junit.Test;
import org.apache.commons.cli.CommandLine;
import org.apache.commons.cli.Option;

public class CommandLine_ESTest {

  @Test
  public void test0()  throws Throwable  {
      CommandLine commandLine0 = new CommandLine();
      Option option0 = new Option("", false, "");
      commandLine0.addOption(option0);
      boolean boolean0 = commandLine0.hasOption('-')
      assertTrue(boolean0);
  }
}
```

—“**Modeling Readability to Improve Unit Tests,**” Daka, Campos, Fraser, Dorn, and Weimer, FSE 2015.

What Makes a Test Readable?



—“**Modeling Readability to Improve Unit Tests,”** Daka, Campos, Fraser, Dorn, and Weimer, FSE 2015.

Naming Generated Tests

```
public class ShoppingCart {
   private int total = 0;
   private final static int MAX = 1000;

   public boolean addPrice(int cost)
          throws IllegalArgumentException {
     if (cost <= 0)
        throw new IllegalArgumentException(“Negative cost");
     if (cost < MAX) {
        total += cost;
        return true;
     } else {
        return false;
     }
   }
   public int getTotal() {
     return total;
   }
}
```

Naming Generated Tests

```
public class ShoppingCart {
   private int total = 0;
   private final static int MAX = 1000;

   public boolean addPrice(int cost)
            throws IllegalArgumentException {
      if (cost <= 0)
         throw new IllegalArgumentException(“Negative cost");
      if (cost < MAX) {
         total += cost;
         return true;
      } else {
         return false;
      }
   }
   public int getTotal() {
      return total;
   }
}
```

Naming Generated Tests

```
public class ShoppingCart {
   private int total = 0;
   private final static int MAX = 1000;

   public boolean addPrice(int cost)
           throws IllegalArgumentException {
      if (cost <= 0)
         throw new IllegalArgumentException(“Negative cost");
      if (cost < MAX) {
         total += cost;
         return true;
      } else {
         return false;
      }
   }
   public int getTotal() {
      return total;
   }
}
```

EvoSuite

`test0, test1, test2, test3, test4`

Naming Generated Tests

```
public class ShoppingCart {
   private int total = 0;
   private final static int MAX = 1000;

   public boolean addPrice(int cost)
            throws IllegalArgumentException {
      if (cost <= 0)
         throw new IllegalArgumentException(“Negative cost");
      if (cost < MAX) {
         total += cost;
         return true;
      } else {
         return false;
      }
   }
   public int getTotal() {
      return total;
   }
}
```

EvoSuite

`test0, test1, test2, test3, test4`

jTExpert

`TestCase0, TestCase1, TestCase2`

Naming Generated Tests

```java
public class ShoppingCart {
   private int total = 0;
   private final static int MAX = 1000;

   public boolean addPrice(int cost)
           throws IllegalArgumentException {
      if (cost <= 0)
         throw new IllegalArgumentException(“Negative cost");
      if (cost < MAX) {
         total += cost;
         return true;
      } else {
         return false;
      }
   }
   public int getTotal() {
      return total;
   }
}
```

EvoSuite

`test0, test1, test2, test3, test4`

jTExpert

`TestCase0, TestCase1, TestCase2`

Randoop

`test01, test02, ..., test21`

Naming Generated Tests

```
public class ShoppingCart {
   private int total = 0;
   private final static int MAX = 1000;

   public boolean addPrice(int cost)
         throws IllegalArgumentException {
      if (cost <= 0)
         throw new IllegalArgumentException(“Negative cost");
      if (cost < MAX) {
         total += cost;
         return true;
      } else {
         return false;
      }
   }
   public int getTotal() {
      return total;
   }
}
```

EvoSuite

`test0, test1, test2, test3, test4`

jTExpert

`TestCase0, TestCase1, TestCase2`

Randoop

`test01, test02, ..., test21`

AgitarOne

`testConstructor, testGetTotal,`
`testAddPrice, testAddPrice1`

Descriptive Test Names

```
public class ShoppingCart {
    private int total = 0;
    private final static int MAX = 1000;

    public boolean addPrice(int cost)
                throws IllegalArgumentException {
        if (cost <= 0)
            throw new IllegalArgumentException(“Negative cost");
        if (cost < MAX) {
            total += cost;
            return true;
        } else {
            return false;
        }
    }
    public int getTotal() {
        return total;
    }
}
```

Descriptive Test Names

```
public class ShoppingCart {
    private int total = 0;
    private final static int MAX = 1000;

    public boolean addPrice(int cost)
                throws IllegalArgumentException {
        if (cost <= 0)
            throw new IllegalArgumentException(“Negative cost");
        if (cost < MAX) {
            total += cost;
            return true;
        } else {
            return false;
        }
    }
    public int getTotal() {
        return total;
    }
}
```

Coverage Criterion
Methods
Exceptions
Output
Input

Descriptive Test Names

```
public class ShoppingCart {
    private int total = 0;
    private final static int MAX = 1000;

    public boolean addPrice(int cost)
                throws IllegalArgumentException {
        if (cost <= 0)
            throw new IllegalArgumentException(“Negative cost");
        if (cost < MAX) {
            total += cost;
            return true;
        } else {
            return false;
        }
    }
    public int getTotal() {
        return total;
    }
}
```

Coverage Criterion	**Coverage Goals**
Methods	<init>, addPrice, getTotal
Exceptions	addPrice
Output	getTotal (0, >0,<0)
Input	addPrice (0,>0,<0)

Descriptive Test Names

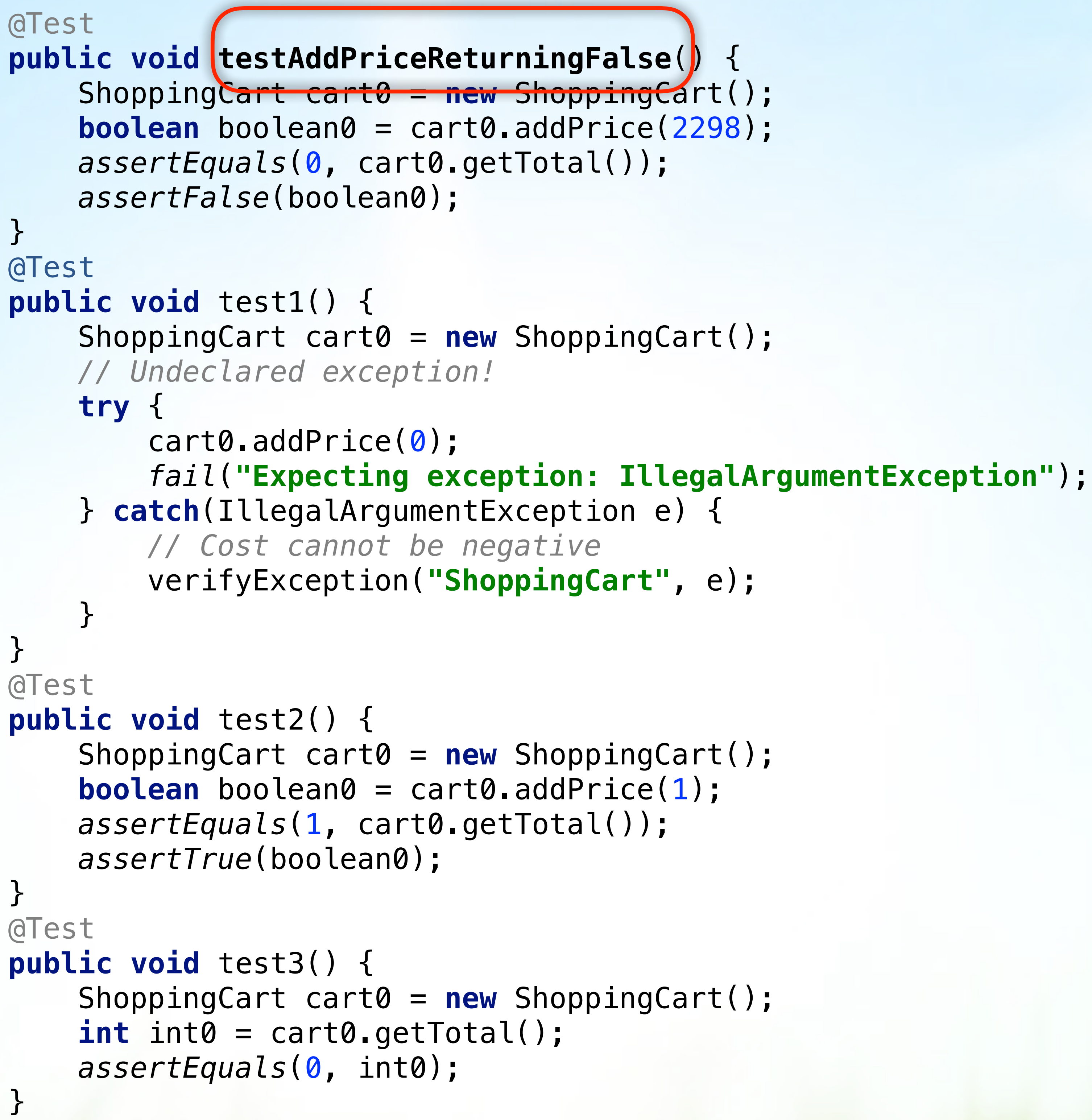
```
public class ShoppingCart {
    private int total = 0;
    private final static int MAX = 1000;

    public boolean addPrice(int cost)
                throws IllegalArgumentException {
        if (cost <= 0)
            throw new IllegalArgumentException(“Negative cost");
        if (cost < MAX) {
            total += cost;
            return true;
        } else {
            return false;
        }
    }
    public int getTotal() {
        return total;
    }
}
```

Coverage Criterion	Coverage Goals	Test Names
Methods	<init>, addPrice, getTotal	testCreateShoppingCart, testAddPrice, testGetTotal
Exceptions	addPrice	testAddPriceThrowsIAE
Output	getTotal (0, >0,<0)	testGetTotalReturnsZero, testGetTotalReturnsPositive, testGetTotalReturnsNegative
Input	addPrice (0,>0,<0)	testAddPriceWithZero, testAddPriceWithPositive, testAddPriceWithNegative

Descriptive Test Names

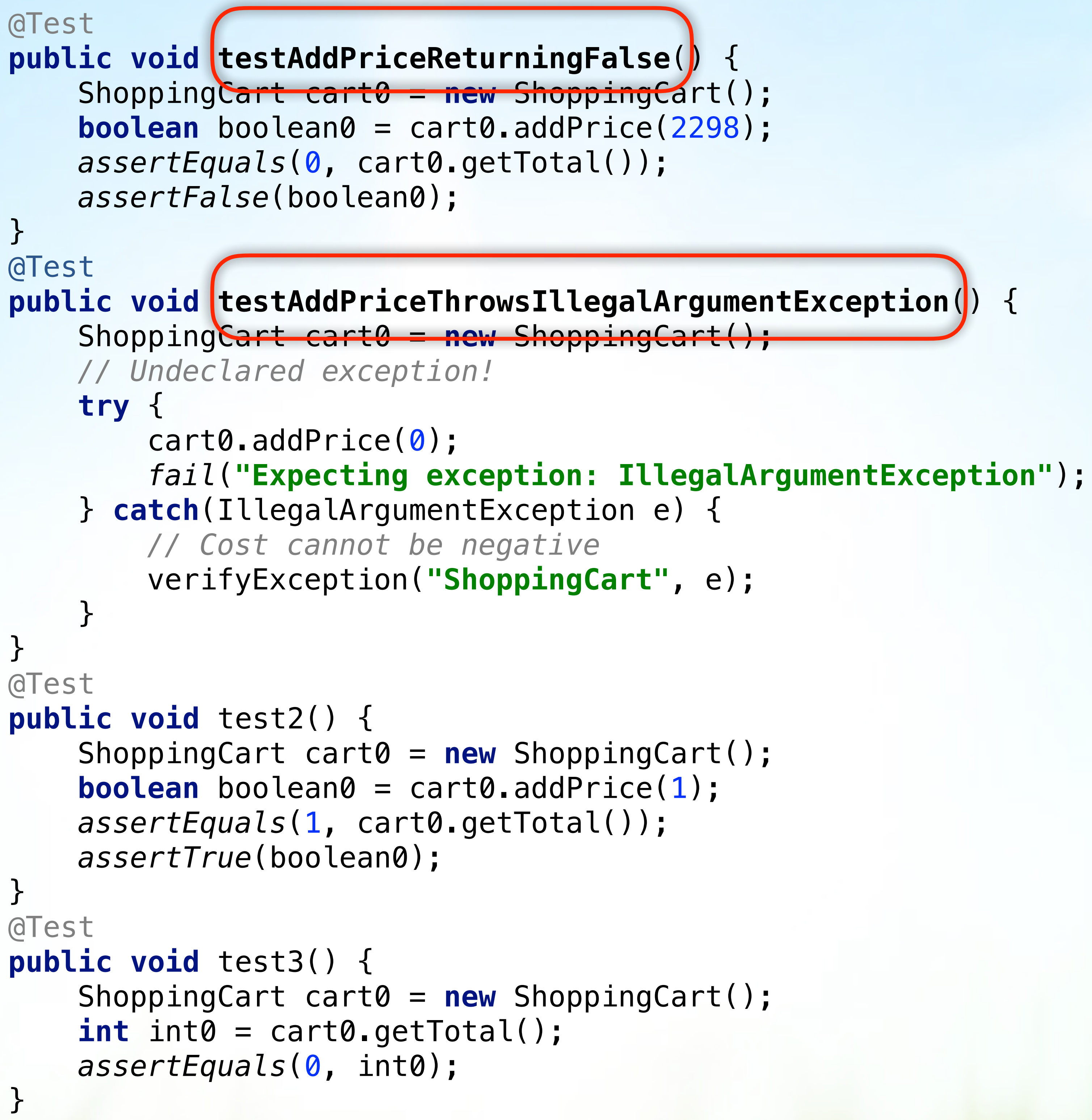
```
@Test
public void test0() {
    ShoppingCart cart0 = new ShoppingCart();
    boolean boolean0 = cart0.addPrice(2298);
    assertEquals(0, cart0.getTotal());
    assertFalse(boolean0);
}
@Test
public void test1() {
    ShoppingCart cart0 = new ShoppingCart();
    // Undeclared exception!
    try {
        cart0.addPrice(0);
        fail("Expecting exception: IllegalArgumentException");
    } catch(IllegalArgumentException e) {
        // Cost cannot be negative
        verifyException("ShoppingCart", e);
    }
}
@Test
public void test2() {
    ShoppingCart cart0 = new ShoppingCart();
    boolean boolean0 = cart0.addPrice(1);
    assertEquals(1, cart0.getTotal());
    assertTrue(boolean0);
}
@Test
public void test3() {
    ShoppingCart cart0 = new ShoppingCart();
    int int0 = cart0.getTotal();
    assertEquals(0, int0);
}
```

Descriptive Test Names



```
@Test
public void testAddPriceReturningFalse() {
    ShoppingCart cart0 = new ShoppingCart();
    boolean boolean0 = cart0.addPrice(2298);
    assertEquals(0, cart0.getTotal());
    assertFalse(boolean0);
}
@Test
public void test1() {
    ShoppingCart cart0 = new ShoppingCart();
    // Undeclared exception!
    try {
        cart0.addPrice(0);
        fail("Expecting exception: IllegalArgumentException");
    } catch(IllegalArgumentException e) {
        // Cost cannot be negative
        verifyException("ShoppingCart", e);
    }
}
@Test
public void test2() {
    ShoppingCart cart0 = new ShoppingCart();
    boolean boolean0 = cart0.addPrice(1);
    assertEquals(1, cart0.getTotal());
    assertTrue(boolean0);
}
@Test
public void test3() {
    ShoppingCart cart0 = new ShoppingCart();
    int int0 = cart0.getTotal();
    assertEquals(0, int0);
}
```

Descriptive Test Names



```
@Test
public void testAddPriceReturningFalse() {
    ShoppingCart cart0 = new ShoppingCart();
    boolean boolean0 = cart0.addPrice(2298);
    assertEquals(0, cart0.getTotal());
    assertFalse(boolean0);
}
@Test
public void testAddPriceThrowsIllegalArgumentException() {
    ShoppingCart cart0 = new ShoppingCart();
    // Undeclared exception!
    try {
        cart0.addPrice(0);
        fail("Expecting exception: IllegalArgumentException");
    } catch(IllegalArgumentException e) {
        // Cost cannot be negative
        verifyException("ShoppingCart", e);
    }
}
@Test
public void test2() {
    ShoppingCart cart0 = new ShoppingCart();
    boolean boolean0 = cart0.addPrice(1);
    assertEquals(1, cart0.getTotal());
    assertTrue(boolean0);
}
@Test
public void test3() {
    ShoppingCart cart0 = new ShoppingCart();
    int int0 = cart0.getTotal();
    assertEquals(0, int0);
}
```

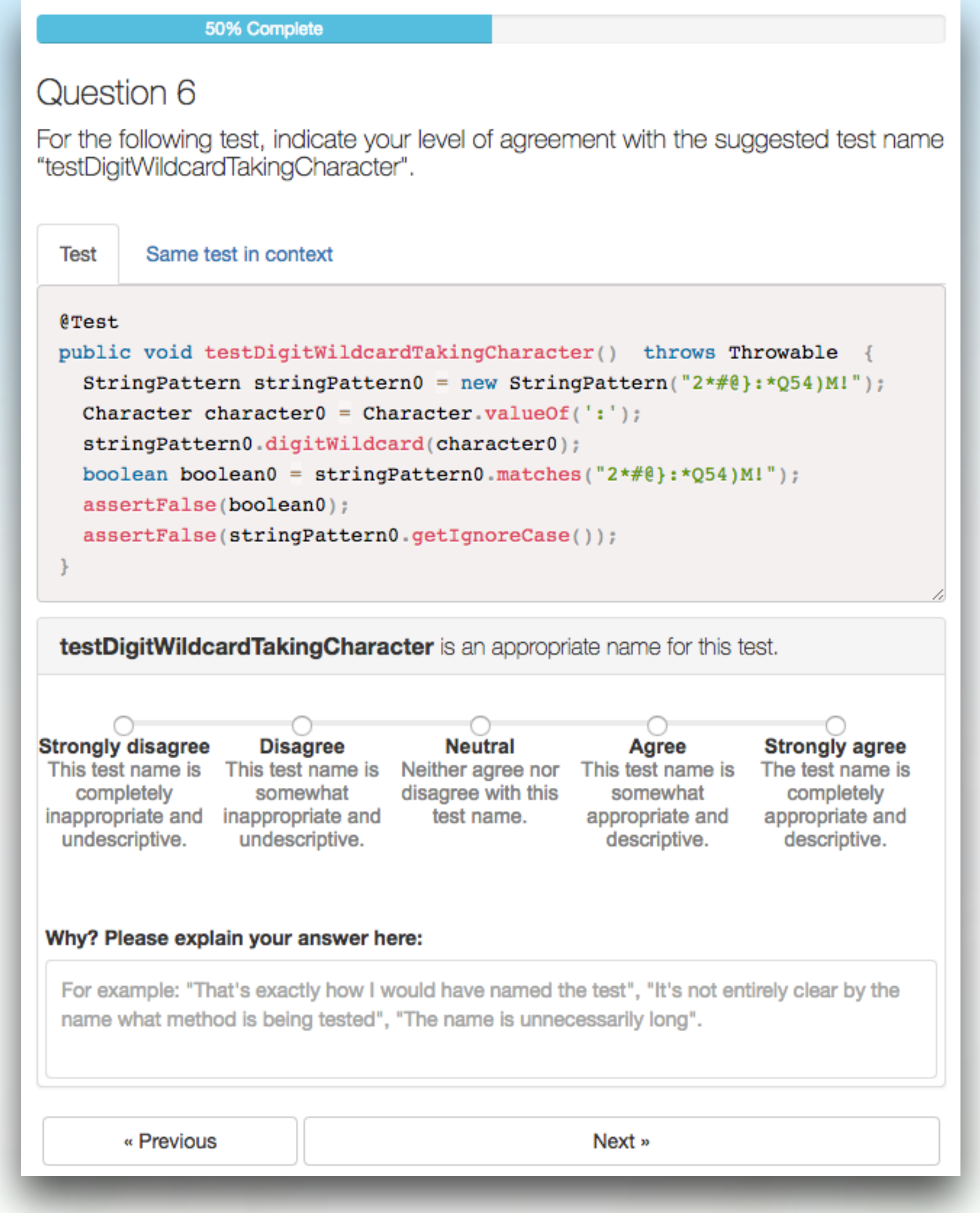
Descriptive Test Names

```
@Test
public void testAddPriceReturningFalse() {
    ShoppingCart cart0 = new ShoppingCart();
    boolean boolean0 = cart0.addPrice(2298);
    assertEquals(0, cart0.getTotal());
    assertFalse(boolean0);
}
@Test
public void testAddPriceThrowsIllegalArgumentException() {
    ShoppingCart cart0 = new ShoppingCart();
    // Undeclared exception!
    try {
        cart0.addPrice(0);
        fail("Expecting exception: IllegalArgumentException");
    } catch(IllegalArgumentException e) {
        // Cost cannot be negative
        verifyException("ShoppingCart", e);
    }
}
@Test
public void testAddPriceReturningTrue() {
    ShoppingCart cart0 = new ShoppingCart();
    boolean boolean0 = cart0.addPrice(1);
    assertEquals(1, cart0.getTotal());
    assertTrue(boolean0);
}
@Test
public void test3() {
    ShoppingCart cart0 = new ShoppingCart();
    int int0 = cart0.getTotal();
    assertEquals(0, int0);
}
```

Descriptive Test Names

```java
@Test
public void testAddPriceReturningFalse() {
    ShoppingCart cart0 = new ShoppingCart();
    boolean boolean0 = cart0.addPrice(2298);
    assertEquals(0, cart0.getTotal());
    assertFalse(boolean0);
}
@Test
public void testAddPriceThrowsIllegalArgumentException() {
    ShoppingCart cart0 = new ShoppingCart();
    // Undeclared exception!
    try {
        cart0.addPrice(0);
        fail("Expecting exception: IllegalArgumentException");
    } catch(IllegalArgumentException e) {
        // Cost cannot be negative
        verifyException("ShoppingCart", e);
    }
}
@Test
public void testAddPriceReturningTrue() {
    ShoppingCart cart0 = new ShoppingCart();
    boolean boolean0 = cart0.addPrice(1);
    assertEquals(1, cart0.getTotal());
    assertTrue(boolean0);
}
@Test
public void testGetTotal() {
    ShoppingCart cart0 = new ShoppingCart();
    int int0 = cart0.getTotal();
    assertEquals(0, int0);
}
```

Descriptive Test Names



Question 6

For the following test, indicate your level of agreement with the suggested test name "testDigitWildcardTakingCharacter".

```
@Test
public void testDigitWildcardTakingCharacter()  throws Throwable  {
  StringPattern stringPattern0 = new StringPattern("2*#@}:*Q54)M!");
  Character character0 = Character.valueOf(':');
  stringPattern0.digitWildcard(character0);
  boolean boolean0 = stringPattern0.matches("2*#@}:*Q54)M!");
  assertFalse(boolean0);
  assertFalse(stringPattern0.getIgnoreCase());
}
```

testDigitWildcardTakingCharacter is an appropriate name for this test.

○ **Strongly disagree** This test name is completely inappropriate and undescriptive.

○ **Disagree** This test name is somewhat inappropriate and undescriptive.

○ **Neutral** Neither agree nor disagree with this test name.

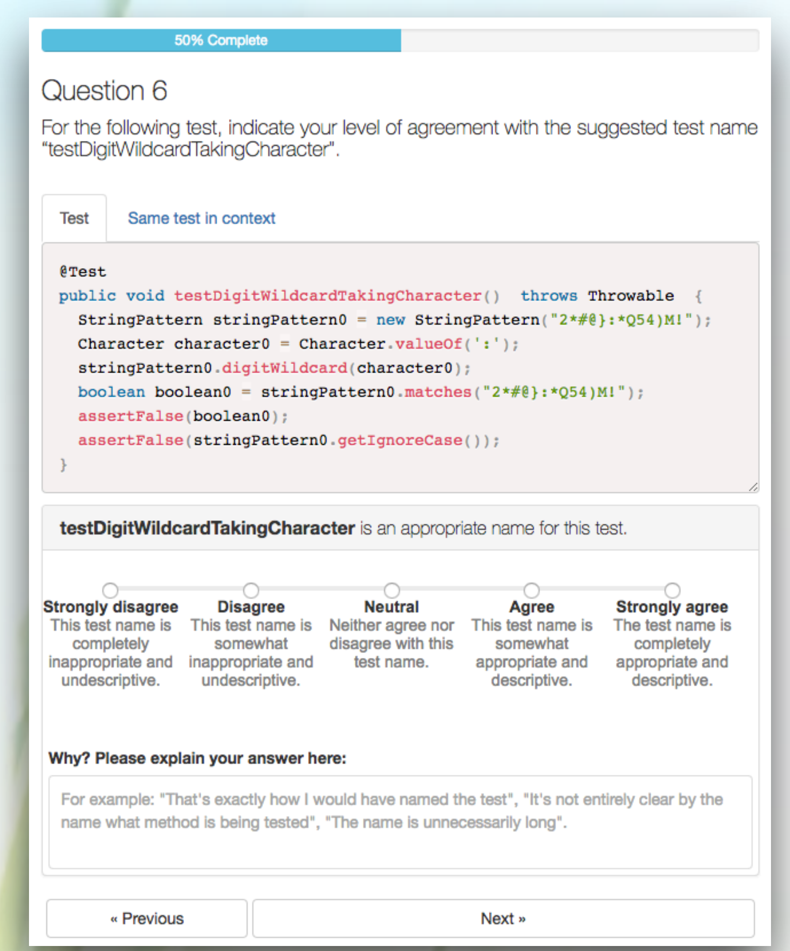
○ **Agree** This test name is somewhat appropriate and descriptive.

○ **Strongly agree** The test name is completely appropriate and descriptive.

Why? Please explain your answer here:

For example: "That's exactly how I would have named the test", "It's not entirely clear by the name what method is being tested", "The name is unnecessarily long".

Descriptive Test Names



Question 6

For the following test, indicate your level of agreement with the suggested test name "testDigitWildcardTakingCharacter".

```
@Test
public void testDigitWildcardTakingCharacter()  throws Throwable  {
  StringPattern stringPattern0 = new StringPattern("2*#@}:*Q54)M!");
  Character character0 = Character.valueOf(':');
  stringPattern0.digitWildcard(character0);
  boolean boolean0 = stringPattern0.matches("2*#@}:*Q54)M!");
  assertFalse(boolean0);
  assertFalse(stringPattern0.getIgnoreCase());
}
```

testDigitWildcardTakingCharacter is an appropriate name for this test.

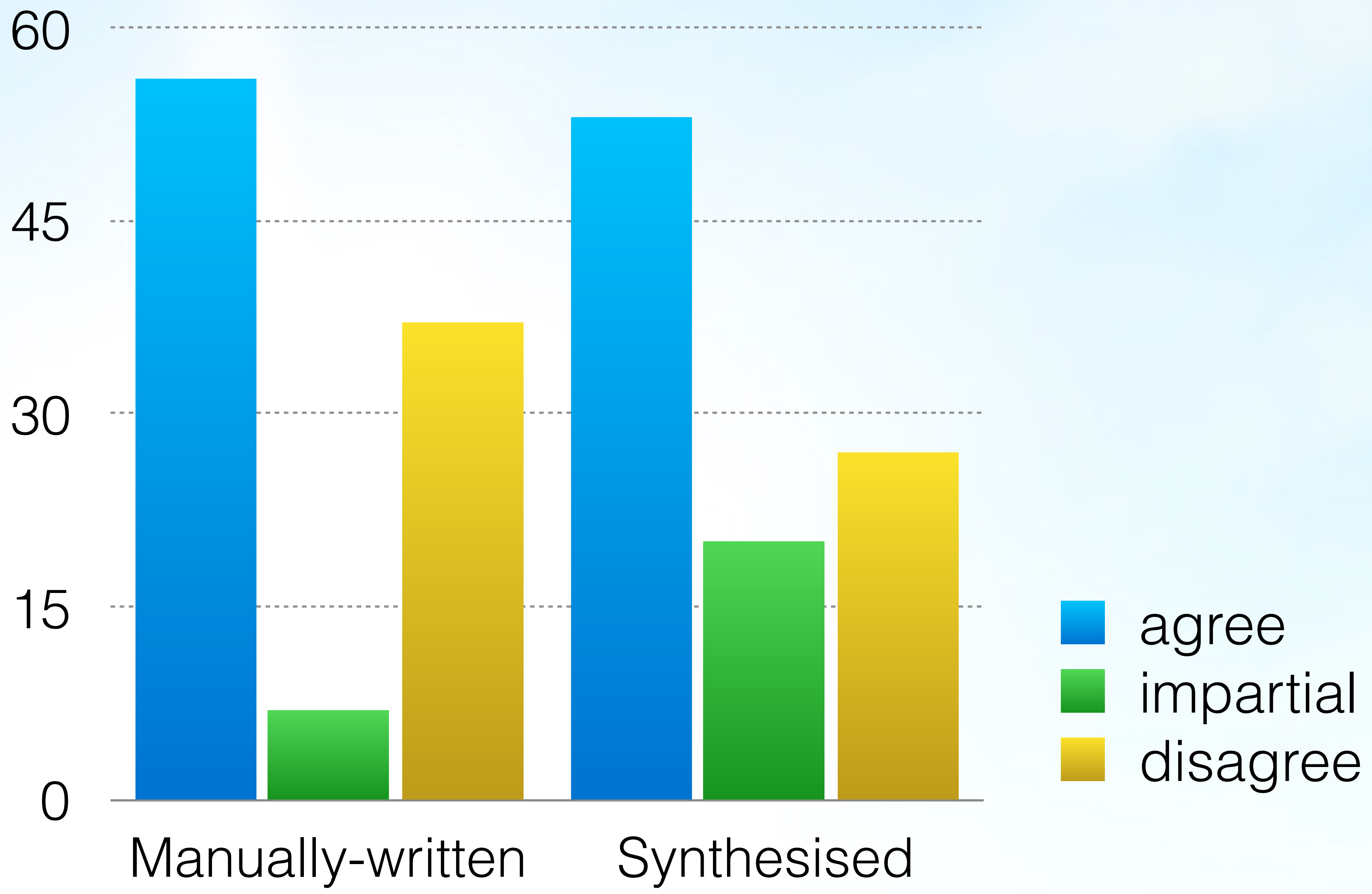
○	○	○	○	○
Strongly disagree This test name is completely inappropriate and undescriptive.	**Disagree** This test name is somewhat inappropriate and undescriptive.	**Neutral** Neither agree nor disagree with this test name.	**Agree** This test name is somewhat appropriate and descriptive.	**Strongly agree** The test name is completely appropriate and descriptive.

Why? Please explain your answer here:

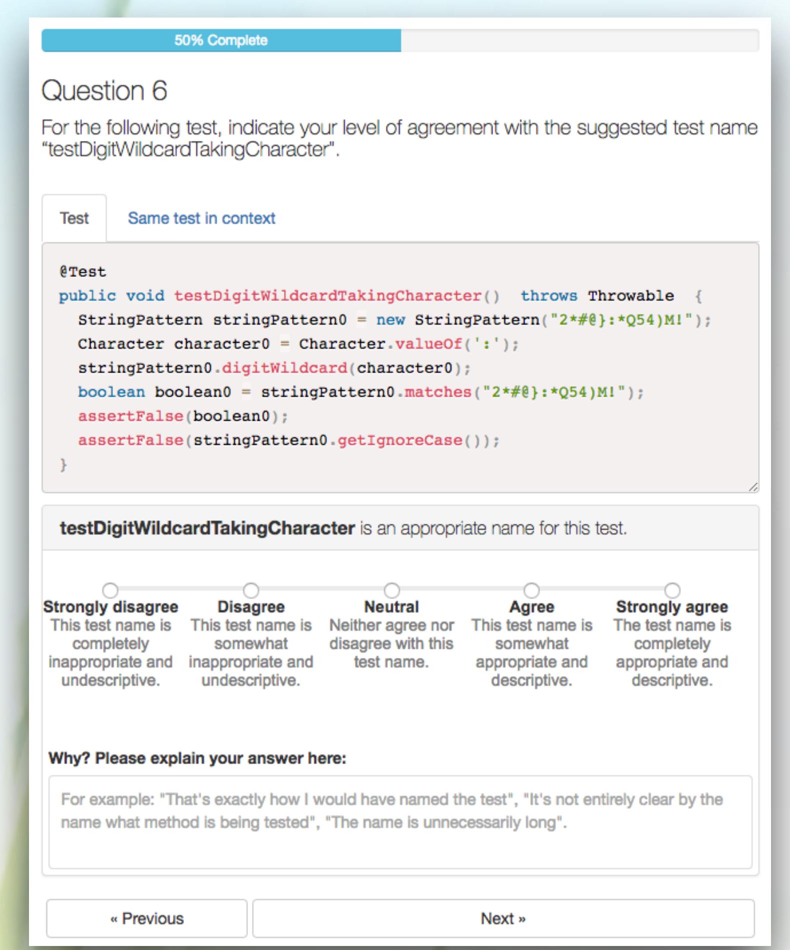
For example: "That's exactly how I would have named the test", "It's not entirely clear by the name what method is being tested", "The name is unnecessarily long".

Descriptive Test Names

“Do you agree with the name of the following test?”



Developers agreed similarly—and disagreed less—with synthesized test names than with manually given names.



Question 6

For the following test, indicate your level of agreement with the suggested test name “testDigitWildcardTakingCharacter”.

```
@Test
public void testDigitWildcardTakingCharacter()  throws Throwable  {
  StringPattern stringPattern0 = new StringPattern("2*#@}:*Q54)M!");
  Character character0 = Character.valueOf(':');
  stringPattern0.digitWildcard(character0);
  boolean boolean0 = stringPattern0.matches("2*#@}:*Q54)M!");
  assertFalse(boolean0);
  assertFalse(stringPattern0.getIgnoreCase());
}
```

testDigitWildcardTakingCharacter is an appropriate name for this test.

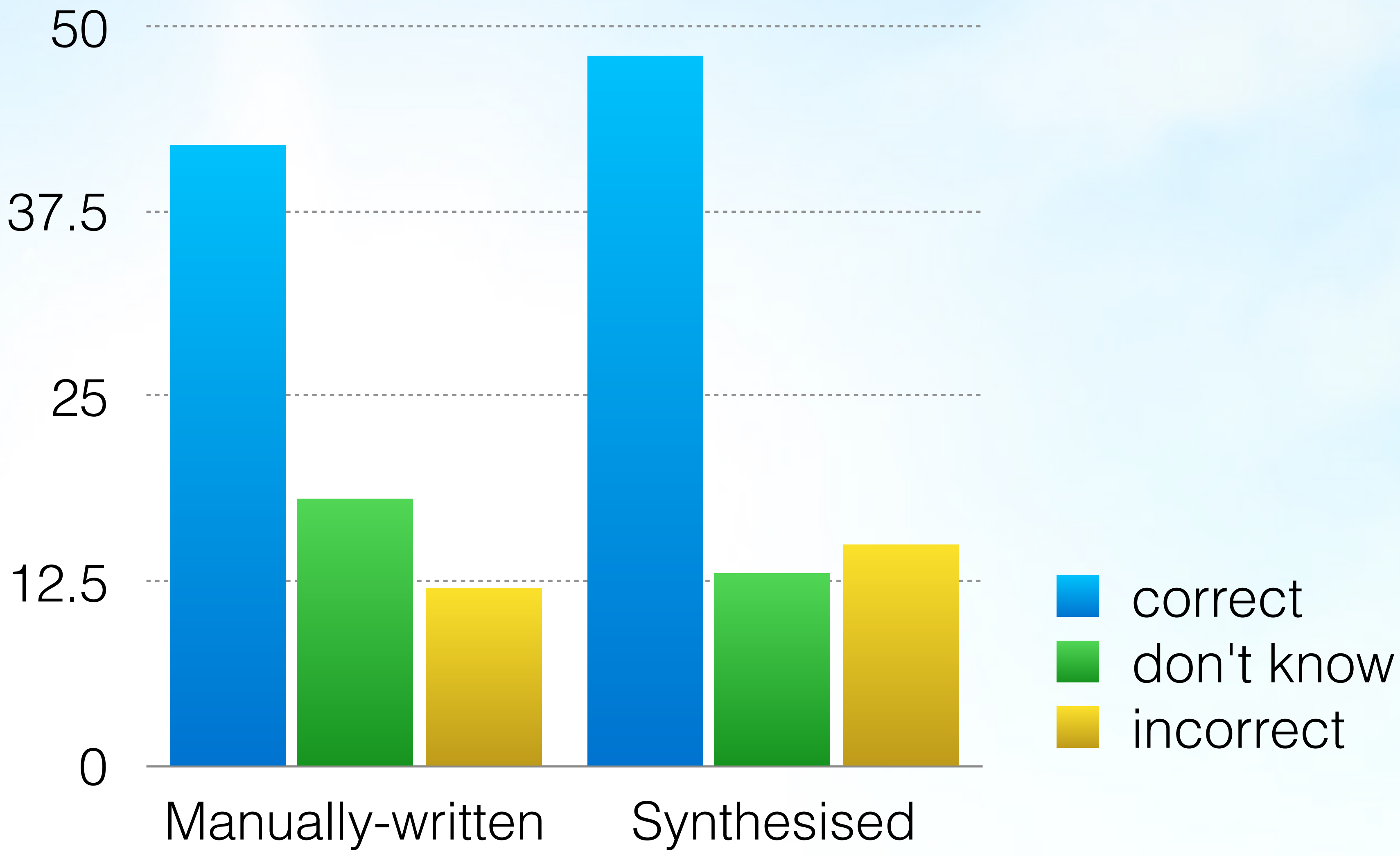
○	○	○	○	○
Strongly disagree This test name is completely inappropriate and undescriptive.	**Disagree** This test name is somewhat inappropriate and undescriptive.	**Neutral** Neither agree nor disagree with this test name.	**Agree** This test name is somewhat appropriate and descriptive.	**Strongly agree** The test name is completely appropriate and descriptive.

Why? Please explain your answer here:

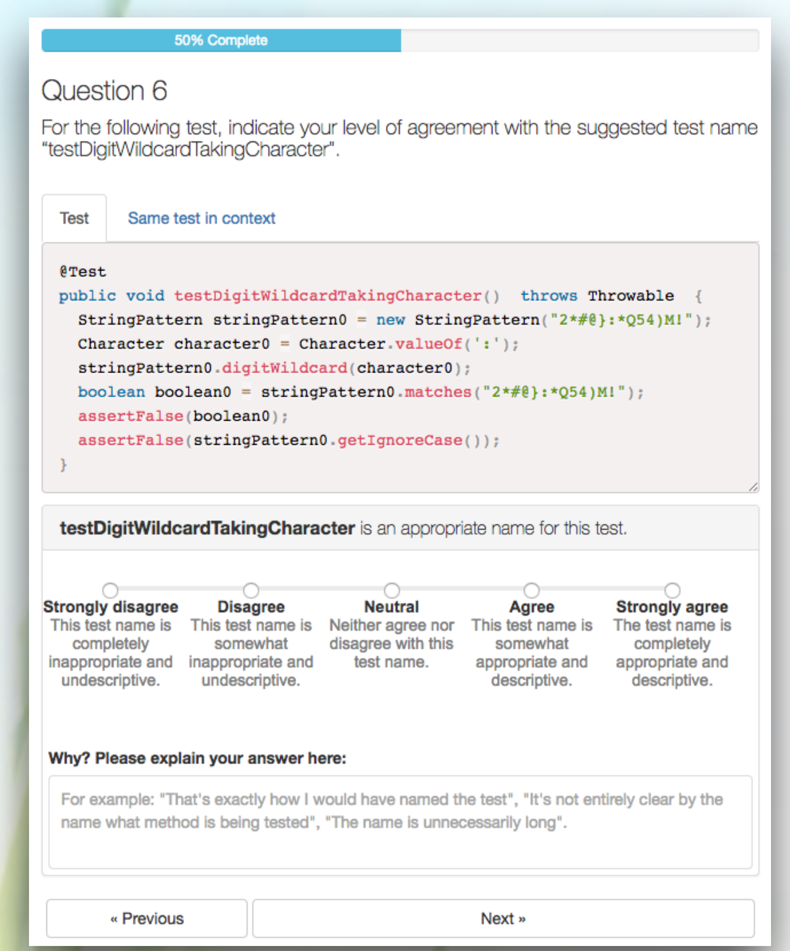
For example: "That's exactly how I would have named the test", "It's not entirely clear by the name what method is being tested", "The name is unnecessarily long".

Descriptive Test Names

“For this test, select the most descriptive name from the list.”



Developers were slightly more accurate and faster at matching tests with synthesized names.



Question 6

For the following test, indicate your level of agreement with the suggested test name “testDigitWildcardTakingCharacter”.

```
@Test
public void testDigitWildcardTakingCharacter()  throws Throwable  {
  StringPattern stringPattern0 = new StringPattern("2*#@}:*Q54)M!");
  Character character0 = Character.valueOf(':');
  stringPattern0.digitWildcard(character0);
  boolean boolean0 = stringPattern0.matches("2*#@}:*Q54)M!");
  assertFalse(boolean0);
  assertFalse(stringPattern0.getIgnoreCase());
}
```

testDigitWildcardTakingCharacter is an appropriate name for this test.

○ **Strongly disagree** This test name is completely inappropriate and undescriptive.

○ **Disagree** This test name is somewhat inappropriate and undescriptive.

○ **Neutral** Neither agree nor disagree with this test name.

○ **Agree** This test name is somewhat appropriate and descriptive.

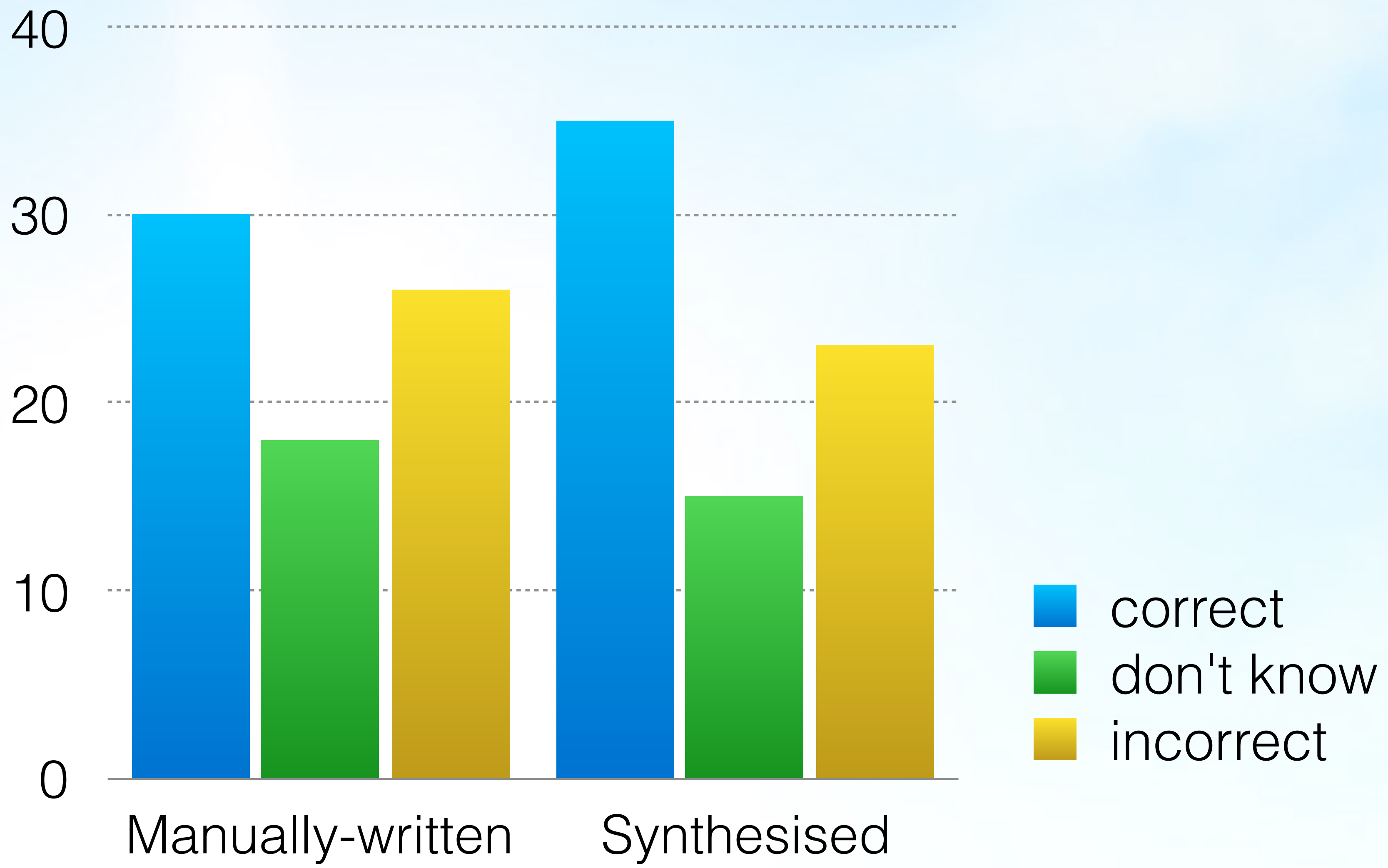
○ **Strongly agree** The test name is completely appropriate and descriptive.

Why? Please explain your answer here:

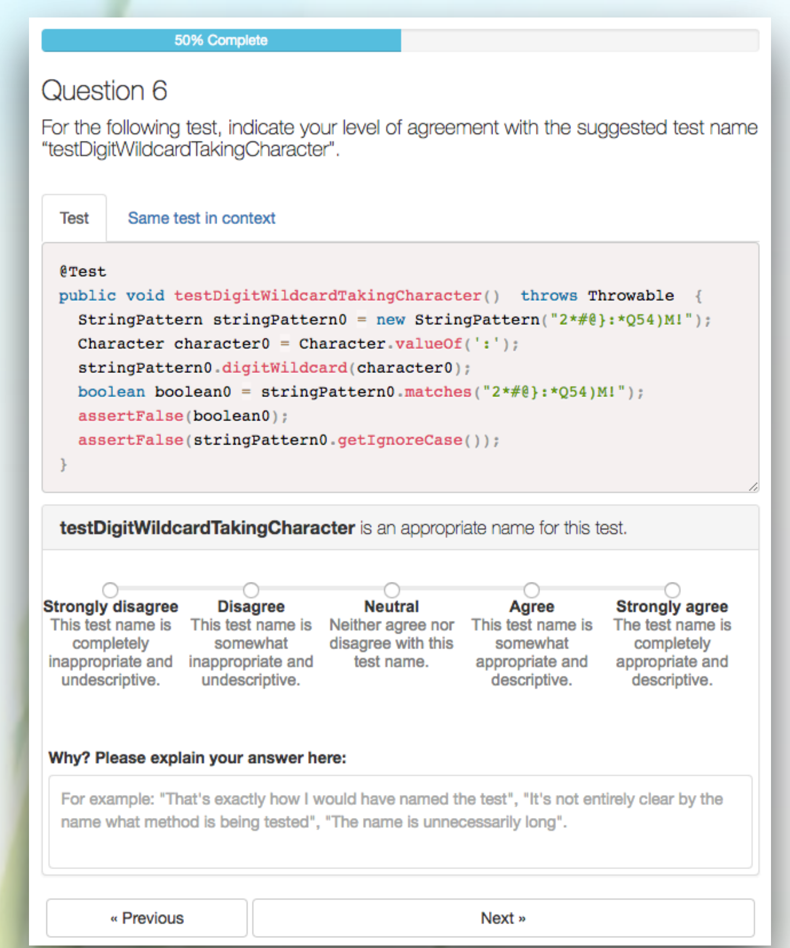
For example: "That's exactly how I would have named the test", "It's not entirely clear by the name what method is being tested", "The name is unnecessarily long".

Descriptive Test Names

"There is a bug in line X, which test would you choose?"



Developers were more accurate at identifying relevant tests for given pieces of code using synthesized test names.



Question 6

For the following test, indicate your level of agreement with the suggested test name "testDigitWildcardTakingCharacter".

```
@Test
public void testDigitWildcardTakingCharacter()  throws Throwable  {
  StringPattern stringPattern0 = new StringPattern("2*#@}:*Q54)M!");
  Character character0 = Character.valueOf(':');
  stringPattern0.digitWildcard(character0);
  boolean boolean0 = stringPattern0.matches("2*#@}:*Q54)M!");
  assertFalse(boolean0);
  assertFalse(stringPattern0.getIgnoreCase());
}
```

testDigitWildcardTakingCharacter is an appropriate name for this test.

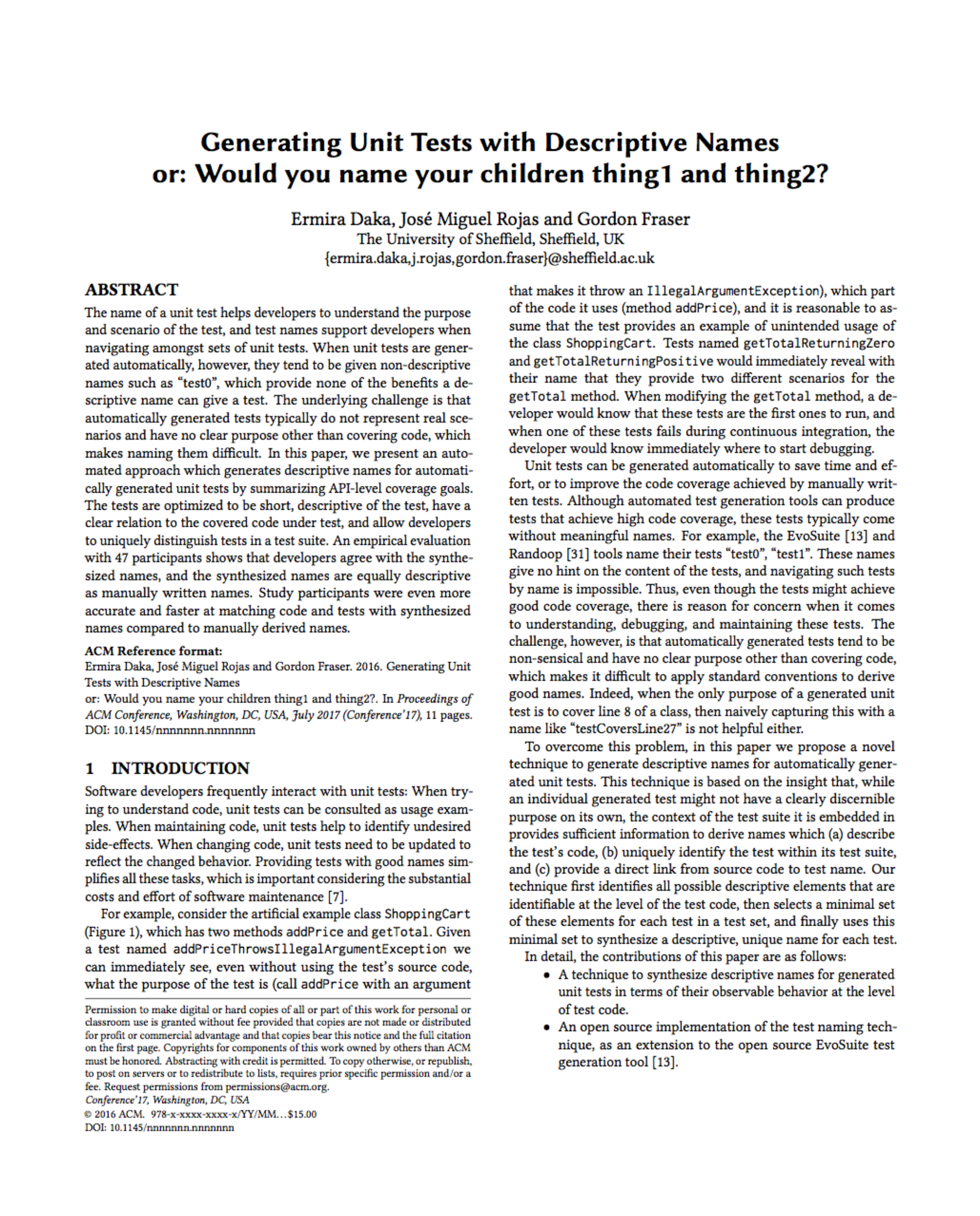
○	○	○	○	○
Strongly disagree This test name is completely inappropriate and undescriptive.	**Disagree** This test name is somewhat inappropriate and undescriptive.	**Neutral** Neither agree nor disagree with this test name.	**Agree** This test name is somewhat appropriate and descriptive.	**Strongly agree** The test name is completely appropriate and descriptive.

Why? Please explain your answer here:

For example: "That's exactly how I would have named the test", "It's not entirely clear by the name what method is being tested", "The name is unnecessarily long".

Descriptive Test Names

ISSTA 2017
Santa Barbara



Generating Unit Tests with Descriptive Names or: Would you name your children thing1 and thing2?

Ermira Daka, José Miguel Rojas and Gordon Fraser
The University of Sheffield, Sheffield, UK
{ermira.daka,j.rojas,gordon.fraser}@sheffield.ac.uk

ABSTRACT

The name of a unit test helps developers to understand the purpose and scenario of the test, and test names support developers when navigating amongst sets of unit tests. When unit tests are generated automatically, however, they tend to be given non-descriptive names such as "test0", which provide none of the benefits a descriptive name can give a test. The underlying challenge is that automatically generated tests typically do not represent real scenarios and have no clear purpose other than covering code, which makes naming them difficult. In this paper, we present an automated approach which generates descriptive names for automatically generated unit tests by summarizing API-level coverage goals. The tests are optimized to be short, descriptive of the test, have a clear relation to the covered code under test, and allow developers to uniquely distinguish tests in a test suite. An empirical evaluation with 47 participants shows that developers agree with the synthesized names, and the synthesized names are equally descriptive as manually written names. Study participants were even more accurate and faster at matching code and tests with synthesized names compared to manually derived names.

ACM Reference format:
Ermira Daka, José Miguel Rojas and Gordon Fraser. 2016. Generating Unit Tests with Descriptive Names
or: Would you name your children thing1 and thing2?. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software developers frequently interact with unit tests: When trying to understand code, unit tests can be consulted as usage examples. When maintaining code, unit tests help to identify undesired side-effects. When changing code, unit tests need to be updated to reflect the changed behavior. Providing tests with good names simplifies all these tasks, which is important considering the substantial costs and effort of software maintenance [7].

For example, consider the artificial example class `ShoppingCart` (Figure 1), which has two methods `addPrice` and `getTotal`. Given a test named `addPriceThrowsIllegalArgumentException` we can immediately see, even without using the test's source code, what the purpose of the test is (call `addPrice` with an argument that makes it throw an `IllegalArgumentException`), which part of the code it uses (method `addPrice`), and it is reasonable to assume that the test provides an example of unintended usage of the class `ShoppingCart`. Tests named `getTotalReturningZero` and `getTotalReturningPositive` would immediately reveal with their name that they provide two different scenarios for the `getTotal` method. When modifying the `getTotal` method, a developer would know that these tests are the first ones to run, and when one of these tests fails during continuous integration, the developer would know immediately where to start debugging.

Unit tests can be generated automatically to save time and effort, or to improve the code coverage achieved by manually written tests. Although automated test generation tools can produce tests that achieve high code coverage, these tests typically come without meaningful names. For example, the EvoSuite [13] and Randoop [31] tools name their tests "test0", "test1". These names give no hint on the content of the tests, and navigating such tests by name is impossible. Thus, even though the tests might achieve good code coverage, there is reason for concern when it comes to understanding, debugging, and maintaining these tests. The challenge, however, is that automatically generated tests tend to be non-sensical and have no clear purpose other than covering code, which makes it difficult to apply standard conventions to derive good names. Indeed, when the only purpose of a generated unit test is to cover line 8 of a class, then naively capturing this with a name like "testCoversLine27" is not helpful either.

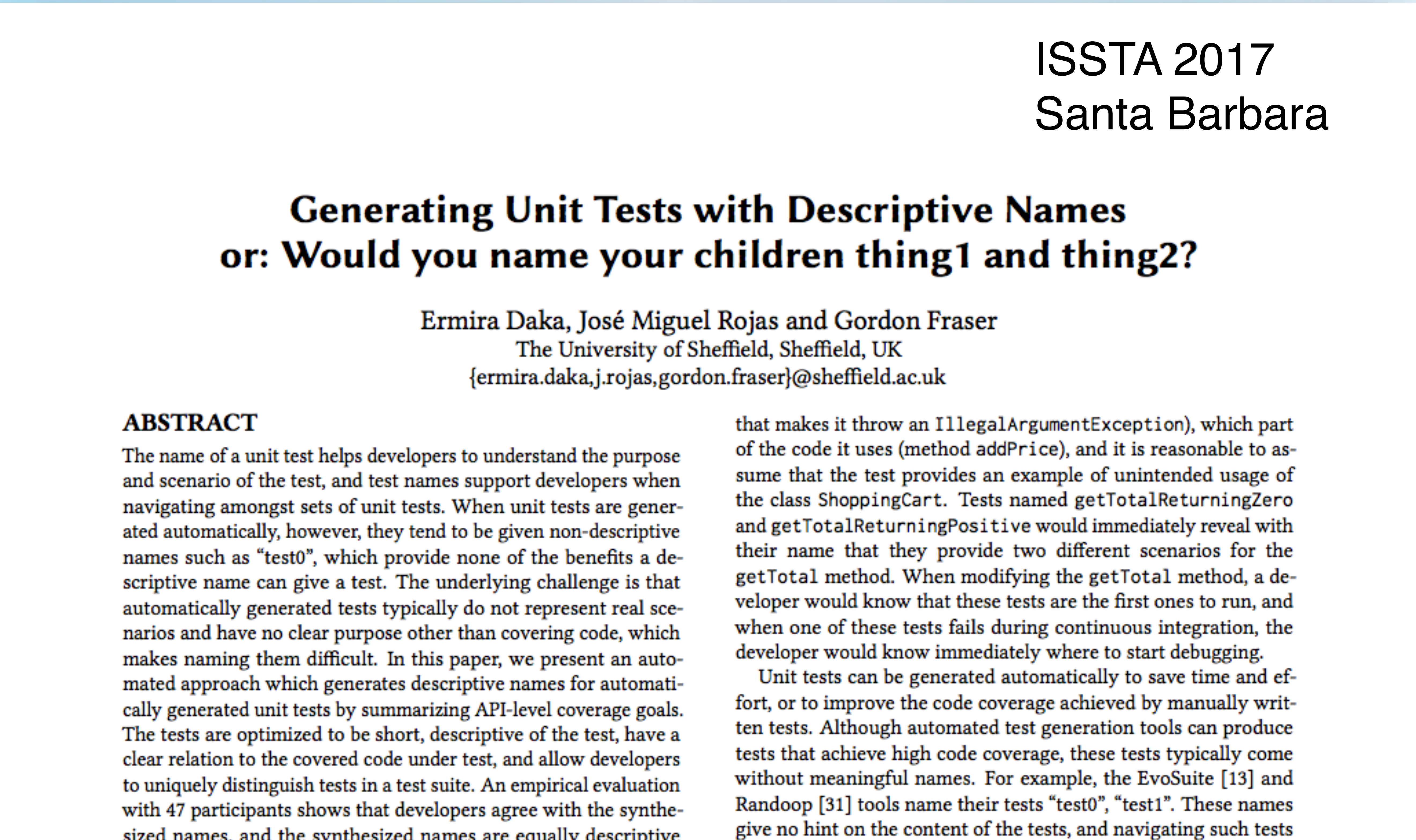
To overcome this problem, in this paper we propose a novel technique to generate descriptive names for automatically generated unit tests. This technique is based on the insight that, while an individual generated test might not have a clearly discernible purpose on its own, the context of the test suite it is embedded in provides sufficient information to derive names which (a) describe the test's code, (b) uniquely identify the test within its test suite, and (c) provide a direct link from source code to test name. Our technique first identifies all possible descriptive elements that are identifiable at the level of the test code, then selects a minimal set of these elements for each test in a test set, and finally uses this minimal set to synthesize a descriptive, unique name for each test.

In detail, the contributions of this paper are as follows:

- A technique to synthesize descriptive names for generated unit tests in terms of their observable behavior at the level of test code.
- An open source implementation of the test naming technique, as an extension to the open source EvoSuite test generation tool [13].

Conference'17, Washington, DC, USA
© 2016 ACM. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Descriptive Test Names



ISSTA 2017
Santa Barbara

Generating Unit Tests with Descriptive Names or: Would you name your children thing1 and thing2?

Ermira Daka, José Miguel Rojas and Gordon Fraser
The University of Sheffield, Sheffield, UK
{ermira.daka,j.rojas,gordon.fraser}@sheffield.ac.uk

ABSTRACT

The name of a unit test helps developers to understand the purpose and scenario of the test, and test names support developers when navigating amongst sets of unit tests. When unit tests are generated automatically, however, they tend to be given non-descriptive names such as "test0", which provide none of the benefits a descriptive name can give a test. The underlying challenge is that automatically generated tests typically do not represent real scenarios and have no clear purpose other than covering code, which makes naming them difficult. In this paper, we present an automated approach which generates descriptive names for automatically generated unit tests by summarizing API-level coverage goals. The tests are optimized to be short, descriptive of the test, have a clear relation to the covered code under test, and allow developers to uniquely distinguish tests in a test suite. An empirical evaluation with 47 participants shows that developers agree with the synthesized names, and the synthesized names are equally descriptive

that makes it throw an IllegalArgumentException), which part of the code it uses (method addPrice), and it is reasonable to assume that the test provides an example of unintended usage of the class ShoppingCart. Tests named getTotalReturningZero and getTotalReturningPositive would immediately reveal with their name that they provide two different scenarios for the getTotal method. When modifying the getTotal method, a developer would know that these tests are the first ones to run, and when one of these tests fails during continuous integration, the developer would know immediately where to start debugging.

Unit tests can be generated automatically to save time and effort, or to improve the code coverage achieved by manually written tests. Although automated test generation tools can produce tests that achieve high code coverage, these tests typically come without meaningful names. For example, the EvoSuite [13] and Randoop [31] tools name their tests "test0", "test1". These names give no hint on the content of the tests, and navigating such tests

Descriptive Test Names

DEMO

Readability in future SBST Contests?

Or some other quality metric beyond coverage/ mutation score

Readability in future SBST Contests?

www.evosuite.org

“EvoSuite failed to produce any test suites for bench- marks JXPATH-7 and OKHTTP-8, and also struggled often for benchmarks LA4J-3, LA4J-7, BCEL-9 (highlighted in Table II). All executions of EVOSUITE for JXPATH-7 failed in the instrumentation phase, where Java’s 64k limit on the size of methods was exceeded.”