Comparing User-Provided Tests to Developer-Provided Tests

René Just, Chris Parnin, Ian Drosos, Michael D. Ernst
ISSTA 2018
User-provided tests

- Found in bug reports
- One small test
- Weak or no assertions
- High code coverage
- Used by programmers

Fault localization
- 5-14% worse
- Automated program repair
- 54-100% worse

Developer-provided tests

- Committed to repository
- More tests, more LOC
- More, stronger assertions
- Focused on the defect
- Used in experiments

User-provided tests should be used in experiments.
Fault localization: where is the defect?

Defective program

def double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}
Fault localization: where is the defect?

Defective program

default avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}

Test suite

Passing tests

Failing tests

Statement ranking

default avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}
Evaluating fault localization

Defective program

```java
double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}
```

Statement ranking

Compare to known location of defect

Test suite

- Passing tests
- Failing tests
Evaluating fault localization

Defective program

double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}

Test suite

Passing tests

Failing tests

Statement ranking

double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}

Fault localization technique 1

Fault localization technique 2

Compare to known location of defect
Evaluating fault localization

Fault localization technique 1

default avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}

Fault localization technique 2

default avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}

Compare to known location of defect

Statement ranking

Defective program
default avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}

Test suite

Passing tests

Failing tests
Evaluating fault localization

Defective program
```java
double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}
```

Test suite
- Passing tests
- Failing tests

Early work
- Artificial defects (“mutants”)
  - Easy to create lots of them
  - Known fault locations
- Pearson et al. [ICSE 2017]
  - 310 real defects (Defects4J)
  - 2995 artificial defects

Fault localization technique 2
```java
double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}
```
Evaluating fault localization

Defective program
```java
double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}
```

Test suite
- Passing tests
- Failing tests

Early work
- Artificial defects ("mutants")
  - Easy to create lots of them
  - Known fault locations

Pearson et al. [ICSE 2017]
- 310 real defects (Defects4J)
- 2995 artificial defects

Early work
- Artificial tests
  - Written by researchers
  - Unrealistically strong

Pearson et al. [ICSE 2017]
- Real tests (Defects4J)
  - Written by developers
  - Committed with the fix
Comparison of fault localization techniques

<table>
<thead>
<tr>
<th>Prior studies</th>
<th>(winner &gt; loser)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ochiai &gt; Tarantula</td>
<td></td>
</tr>
<tr>
<td>Barinel &gt; Ochiai</td>
<td></td>
</tr>
<tr>
<td>Barinel &gt; Tarantula</td>
<td></td>
</tr>
<tr>
<td>Op2 &gt; Ochiai</td>
<td></td>
</tr>
<tr>
<td>Op2 &gt; Tarantula</td>
<td></td>
</tr>
<tr>
<td>DStar &gt; Ochiai</td>
<td></td>
</tr>
<tr>
<td>DStar &gt; Tarantula</td>
<td></td>
</tr>
<tr>
<td>Metallaxis &gt; Ochiai</td>
<td></td>
</tr>
<tr>
<td>MUSE &gt; Op2</td>
<td></td>
</tr>
<tr>
<td>MUSE &gt; Tarantula</td>
<td></td>
</tr>
</tbody>
</table>

SBFL vs. SBFL

SBFL vs. SBFL

MBFL vs. SBFL
## Comparison of fault localization techniques

<table>
<thead>
<tr>
<th>Prior studies (winner &gt; loser)</th>
<th>Ours (artificial faults) Replicated</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBFL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ochiai &gt; Tarantula</td>
<td>yes</td>
<td>small</td>
</tr>
<tr>
<td>Barinel &gt; Ochiai</td>
<td>no</td>
<td>small</td>
</tr>
<tr>
<td>Barinel &gt; Tarantula</td>
<td>yes</td>
<td>negligible</td>
</tr>
<tr>
<td>Op2 &gt; Ochiai</td>
<td>yes</td>
<td>negligible</td>
</tr>
<tr>
<td>Op2 &gt; Tarantula</td>
<td>yes</td>
<td>small</td>
</tr>
<tr>
<td>DStar &gt; Ochiai</td>
<td>yes</td>
<td>negligible</td>
</tr>
<tr>
<td>DStar &gt; Tarantula</td>
<td>yes</td>
<td>small</td>
</tr>
<tr>
<td>MBFL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metallaxis &gt; Ochiai</td>
<td>yes</td>
<td>negligible</td>
</tr>
<tr>
<td>MUSE &gt; Op2</td>
<td>no</td>
<td>negligible</td>
</tr>
<tr>
<td>MUSE &gt; Tarantula</td>
<td>no</td>
<td>negligible</td>
</tr>
</tbody>
</table>

Results agree with most prior studies on artificial faults but **only 3 effect sizes are not negligible.**
Comparison of fault localization techniques

<table>
<thead>
<tr>
<th></th>
<th>Prior studies (winner &gt; loser)</th>
<th>Ours (artificial faults)</th>
<th>Ours (real faults)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SBFL</strong> vs. <strong>SBFL</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ochiai &gt; Tarantula</td>
<td>yes</td>
<td>small</td>
<td>insignificant negligible</td>
</tr>
<tr>
<td>Barinel &gt; Ochiai</td>
<td>no</td>
<td>small</td>
<td>insignificant negligible</td>
</tr>
<tr>
<td>Barinel &gt; Tarantula</td>
<td>yes</td>
<td>negligible</td>
<td>insignificant negligible</td>
</tr>
<tr>
<td>Op2 &gt; Ochiai</td>
<td>yes</td>
<td>negligible</td>
<td>no negligible</td>
</tr>
<tr>
<td>Op2 &gt; Tarantula</td>
<td>yes</td>
<td>small</td>
<td>insignificant negligible</td>
</tr>
<tr>
<td>DStar &gt; Ochiai</td>
<td>yes</td>
<td>negligible</td>
<td>insignificant negligible</td>
</tr>
<tr>
<td>DStar &gt; Tarantula</td>
<td>yes</td>
<td>small</td>
<td>insignificant negligible</td>
</tr>
<tr>
<td><strong>MBFL</strong> vs. <strong>SBFL</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metallaxis &gt; Ochiai</td>
<td>yes</td>
<td>negligible</td>
<td>no small</td>
</tr>
<tr>
<td>MUSE &gt; Op2</td>
<td>no</td>
<td>negligible</td>
<td>no large</td>
</tr>
<tr>
<td>MUSE &gt; Tarantula</td>
<td>no</td>
<td>negligible</td>
<td>no large</td>
</tr>
</tbody>
</table>

Results disagree with all prior studies on real faults. Design decisions don’t matter: techniques indistinguishable.
Evaluating fault localization

Defective program

```java
double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}
```

Test suite

- Passing tests
- Failing tests

Fault localization technique 1

Statement ranking

Fault localization technique 2

Compare to known location of defect

```java
double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}
```
Evaluating fault localization

Defective program

do double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}

Test suite

Passing tests

Failing tests

New standard methodology: Use real defects from Defects4J (mined from version control repositories)

Fault localization technique 2

do double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}

Compare to known location of defect
Evaluating fault localization

**Defective program**
```java
double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}
```

**Test suite**
- **Passing tests**
- **Failing tests**

**New standard methodology:**
Use real defects from Defects4J (mined from version control repositories)

**Defects4J:** real triggering tests
- Written by developers
- Committed with the fix
Evaluating fault localization

Defective program

doctor avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}

Test suite

- Passing tests
- Failing tests

New standard methodology: Use **real defects** from Defects4J (mined from version control repositories)

Defects4J: **real triggering tests**
- Written by developers
- Committed with the fix

Written before or after the fix?
Evaluating fault localization

Defective program

double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for (int i = 0; i < n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}

Test suite

Passing tests

Failing tests

New standard methodology: Use real defects from Defects4J (mined from version control repositories)

Defects4J: real triggering tests
- Written by developers
- Committed with the fix

Written before or after the fix?

In practice, fault localization is run before the fix, using triggering tests from bug reports.
User-provided test

https://issues.apache.org/jira/browse/LANG-857

class UserTest {
   public void userTest() {
       assertEquals("😰", StringEscapeUtils.escapeCsv("😰");
   }
}
public void testLang857() {
    assertEquals("\uD83D\uDE30", StringEscapeUtils.escapeCsv("\uD83D\uDE30"));
    // Examples from https://en.wikipedia.org/wiki/UTF-16
    assertEquals("\uD800\uDC00", StringEscapeUtils.escapeCsv("\uD800\uDC00"));
    assertEquals("\uD834\uDD1E", StringEscapeUtils.escapeCsv("\uD834\uDD1E"));
    assertEquals("\uDBFF\uDFFD", StringEscapeUtils.escapeCsv("\uDBFF\uDFFD"));
    assertEquals("\uDBFF\uDFFD", StringEscapeUtils.escapeCsv("\uDBFF\uDFFD"));
}

Developers accept 20% of user-provided tests as is.

Developer-provided tests have:
- **More tests**, more LOC
- More, **stronger assertions** (higher mutation score)
- Less code coverage (**more focused**)
Experimental comparison

Developer-provided tests: from Defects4J

User-provided tests: manually extracted from bug reports

Research question: Is experimental setup (dev-provided tests) characteristic of real-world use (user-provided tests)?

● Fault localization
● Program repair
Fault localization applied to user- vs. dev-tests

Top-N metric:
Does the defective statement appear within the first N reports?
Automated program repair (395 bugs, 2 repair tools)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Dev-provided tests</th>
<th>User-provided tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>jGenProg/astor</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ACS</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

Partly due to worse fault localization
Automated program repair (395 bugs, 2 repair tools)

<table>
<thead>
<tr>
<th></th>
<th>Dev-provided tests</th>
<th>User-provided tests</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>jGenProg/astor</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Correct patches</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Generated patches</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td><strong>ACS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Correct patches</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>Generated patches</td>
<td>12</td>
<td>6</td>
</tr>
</tbody>
</table>
Existing developer-written test for Commons Lang #746:

```java
@Test
public void testCreateNumber() {
    // a lot of things can go wrong
    ...
    assertTrue("9 failed", 0xFADE == createNumber("0xFADE").intValue());

    assertTrue("10 failed", -0xFADE == createNumber("-0xFADE").intValue());

    assertEquals("11 failed", Double.valueOf("1.1E20"), createNumber("1.1E20"));
    ...
}
```

More than 20 passing assertions in testCreateNumber!
Test separation

Augmented developer-written test for Commons Lang #746:

```java
@Test
public void testCreateNumber() {
    // a lot of things can go wrong
    ...
    assertTrue("9 failed", 0xFADE == createNumber("0xFADE").intValue());
    assertTrue("9b failed", 0xFADE == createNumber("0Xfade").intValue());
    assertTrue("10 failed", -0xFADE == createNumber("-0xFADE").intValue());
    assertTrue("10b failed", -0xFADE == createNumber("-0Xfade").intValue());
    assertEquals("11 failed", Double.valueOf("1.1E20"), createNumber("1.1E20"));
    ...
}
```
Test separation

Augmented developer-written test for Commons Lang #746:

```java
@Test
class developer-written test for Commons Lang #746:

public void testCreateNumber() {
    // a lot of things can go wrong
    ...
    assertTrue("9 failed", 0xFADE == createNumber("0xFADE").intValue());
    assertTrue("9b failed", 0xFADE == createNumber("0Xfade").intValue());
    assertTrue("10 failed", -0xFADE == createNumber("-0xFADE").intValue());
    assertTrue("10b failed", -0xFADE == createNumber("-0Xfade").intValue());
    assertEquals("11 failed", Double.valueOf("1.1E20"), createNumber("1.1E20"));
    ...
}
```
Test separation

Alternate formulation of the developer-written test:

...  
public void testCreateNumber9() {  
    assertTrue("9 failed", 0xFADE == createNumber("0xFADE").intValue());  
}

public void testCreateNumber9b() {  
    assertTrue("9b failed", 0xFADE == createNumber("0Xfade").intValue());  
}

public void testCreateNumber10() {  
    assertTrue("10 failed", -0xFADE == createNumber("-0xFADE").intValue());  
}

public void testCreateNumber10b() {  
    assertTrue("10b failed", -0xFADE == createNumber("-0Xfade").intValue());  
}

...
Separated tests are better for tools

Developer commits:

- Added only new tests 78% of the time
- Augmented an existing test 22% of the time

What if developers never augmented tests, only added new tests?

Tools should separate tests prior to debugging (see also [Xuan 2014]).
User-provided vs. developer-provided tests

In real-world use, only user-provided tests are available.

User-provided tests:
- Smaller; weaker assertions; less focused
- Fault localization: 5-14% worse
- Automated program repair: 54-100% worse

Experiments should use real artifacts in end-user context.