Mutation Testing

Piteclipse

CptS 422: Software Engineering Principles II
Sarah Fakhoury

✓Whitebox testing

✓Blackbox testing

How did you assess confidence in your checks?

What criteria did you use to write your test cases?

Code coverage as a proxy for confidence?

```
public someConstructor() {

   foo = DEFAULT_VALUE;

 }
```

```
@Before
 public void setUp() {
   constructor = new someConstructor();
 }
```

Code coverage as a proxy for confidence?

```
public someConstructor() {

   foo = NULL;

 }
```

```
@Before
 public void setUp() {
    constructor = new someConstructor();
 }
```

Code coverage as a proxy for confidence?

```
public someConstructor() {

   foo = NULL;

}
```

```
@Before
 public void setUp() {
   constructor = new someConstructor();
 }
```

Bug is not caught -> Test Passes!

How to catch the bug?

Code coverage as a proxy for confidence?

```
public someConstructor() {

   foo = NULL;

 }
```

```
@Before
 public void setUp() {
    constructor = new someConstructor();
 }
@Test
public void initialState() {
    assertEquals( DEFAULT_VALUE, constructor.getFoo() );
}
```

Bug is caught -> Test Fails!

Using code coverage as a criterion is a good way to start writing tests

→Does not ensure all bugs will be caught

→Is not a good way to asses the quality of your test cases

- We can define the quality criterion of our tests as how many bugs they can catch.
- We want to measure the failure rate of our tests
- Need a proxy for failure

→Mutation Testing

Mutation Testing

- The goal of Mutation Testing is to assess the quality of the test cases which should be robust enough to fail mutated code.

- Mutation testing is a testing strategy similar to data flow and boundary value analysis but instead *coverage* is in terms of number of mutants caught instead of tested lines, branches, statements etc.

- Mutation testing should be used in conjunction with other testing techniques, not on its own.

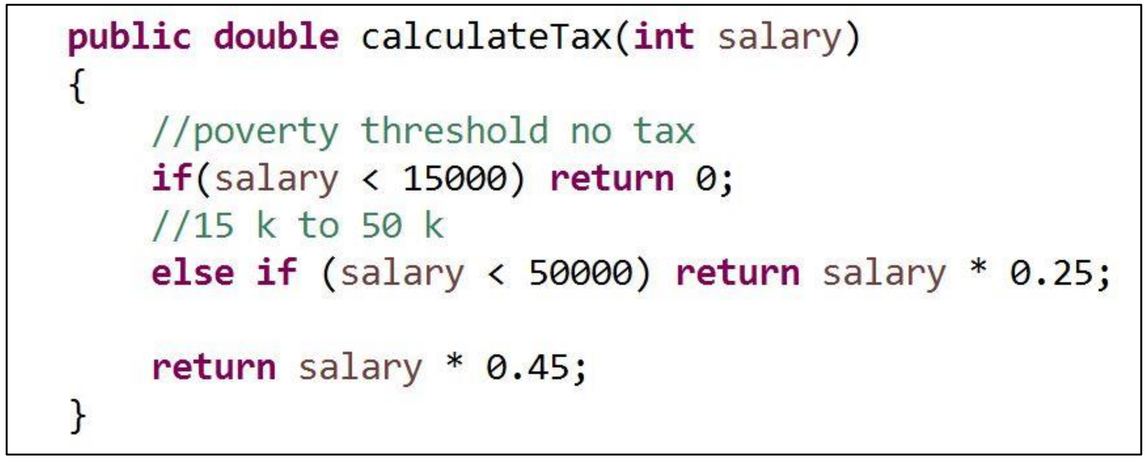
Mutation Testing - Why?

- Traditional test coverage (line, statement, branch etc) measures only which code is **executed** by your tests.
- They do not assess whether tests are actually able to **detect faults** in the executed code
- A suite that only partially tests code can still execute all its branches, so you would have full branch coverage but the code is still only partially tested by the test suite.
- Mutation testing is the **gold standard** against which all other types of coverage are measured

Mutation Testing – How?

- Faults are introduced into the program by mutating code.
- These altered versions of the code are called mutants
- Each mutant contains a single fault
- Ideally, the test cases will fail on the mutated code
- If the test cases do not fail, it may indicate an issue with the test suite.

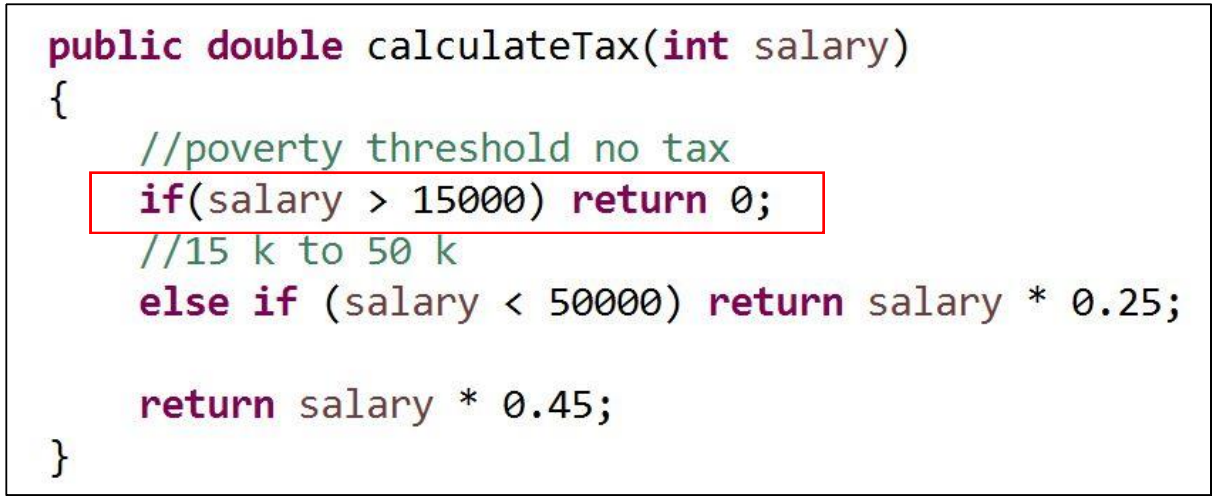
Example Mutation



```java
public double calculateTax(int salary)
{
    //poverty threshold no tax
    if(salary < 15000) return 0;
    //15 k to 50 k
    else if (salary < 50000) return salary * 0.25;

    return salary * 0.45;
}
```

Original code



```java
public double calculateTax(int salary)
{
    //poverty threshold no tax
    if(salary > 15000) return 0;
    //15 k to 50 k
    else if (salary < 50000) return salary * 0.25;

    return salary * 0.45;
}
```

Mutated code

Mutation Testing Algorithm

- Execute each test case against each alive mutant
    - If the output of the mutant differs from the output of the original program, the mutant is considered incorrect and is killed

- Two kinds of mutants survive:
    - **Functionally equivalent** to the original program: Cannot be killed, no test case can catch this
    - Killable: Test cases are insufficient to kill the mutant. New test cases must be created.

Mutation Score

- The mutation score for a set of test cases is the percentage of non-equivalent mutants killed by the test data
- Mutation **Score = 100 * D / (N - E)**
  - D = Dead mutants
  - N = Number of mutants
  - E = Number of equivalent mutants
- A set of test cases is mutation adequate if its mutation score is 100%.

Mutation Testing

- Theoretical and experimental results have shown that mutation testing is an effective approach to measuring the adequacy of test cases.
- The major drawback of mutation testing is the cost of generating the mutants and executing each test case against them.
- Most efforts on mutation testing have focused on reducing its cost by reducing the number of mutants while maintaining the effectiveness of the technique.
- Automated tool needed

Mutation testing for Java

μJava - http://cs.gmu.edu/offutt/mujava/

Oldest tool for java mutation testing

Source code is available on a limited basis to researchers in mutation analysis.

Jester - http://jester.sourceforge.net/

Not actively developed or supported

Jumble - http://jumble.sourceforge.net/

No releases since 2009

Javalanche - http://www.st.cs.uni-saarland.de/mutation/

Piteclipse

- Automated plugin for Eclipse
- Can be used from command line or other IDE’s like InteliJ
- Generates reports per mutation
- Generates reports about code coverage per line
- PIT is currently the only mutation testing system known to work with all of JMock, EasyMock, Mockito, PowerMock and JMockit.

Types of Mutations in PitTest

- Conditionals Boundary Mutator
- Increments Mutator
- Invert Negatives Mutator
- Math Mutator
- Negate Conditionals Mutator
- Return Values Mutator
- Void Method Calls Mutator
- Constructor Calls Mutator
- Inline Constant Mutator
- Non Void Method Calls Mutator
- Remove Conditionals Mutator
- Experimental Member Variable Mutator
- Experimental Switch Mutator

Conditionals Boundary Mutator

Original conditional	Mutated conditional
<	<=
<=	<
>	>=
>=	>

```
if (a < b) { // do something }
```

```
if (a <= b) { // do something }
```

Negate Conditionals Mutator

Original conditional	Mutated conditional
==	!=
!=	==
<=	>
>=	<
<	>=
>	<=

```
if (a == b) { // do something }
```

```
if (a != b) { // do something }
```

Math Mutator

Original conditional	Mutated conditional
+	-
-	+
*	/
/	*
%	*
&	|
|	&
^	&
<<	>>
>>	<<
>>>	<<

```
int a = b + c;
```

```
int a = b - c;
```

Inline Constant Mutator

Constant Type	Mutation
boolean	replace the unmutated value true with false and replace the unmutated value false with true
integer	replace the unmutated value 1 with 0, -1 with 1, 5 with -1 or otherwise increment the unmutated value by one.
long	replace the unmutated value 1 with 0, otherwise increment the unmutated value by one.
float	replace the unmutated values 1.0 and 2.0 with 0.0 and replace any other value with 1.0
double	replace the unmutated value 1.0 with 0.0 and replace any other value with 1.0

```
public int foo() {
    int i = 42;
    return i;
}
```

```
public int foo() {
    int i = 43;
    return i;
}
```

Remove Conditionals Mutator

- The remove conditionals mutator will remove all conditionals statements such that the guarded statements always execute

```
if (a == b) { // do something }
```

```
if (true) { // do something }
```

```
if (false) { // do something }
```

```
if (a == b) { // do something }
else { // do something else }
```

```
if (false) { // do something }
else { // do something else }
```

Return Values Mutator

- The return values mutator mutates the return values of method calls. Depending on the return type of the method another mutation is used.

Return Type	Mutation
boolean	replace the unmutated return value true with false and replace the unmutated return value false with true
int	if the unmutated return value is 0 return 1, otherwise mutate to return value 0
long	replace the unmutated return value x with the result of x+1
Float / double	replace the unmutated return value x with the result of -(x+1.0) if x is not NAN and replace NAN with 0
Object	replace non-null return values with null and throw a java.lang.RuntimeException if the unmutated method would return null

```
public Object foo() {
   return new Object();
}
```

```
public Object foo() {
    new Object();
    return null;
}
```

Void Method Call Mutator

- The void method call mutator removes method calls to void methods

```
public void someVoidMethod(int i) {
        // does something
}

 public int foo() {
    int i = 5;
    doSomething(i);
    return i;
}
```

```
public void someVoidMethod(int i) {
        // does something
}

public int foo() {

    int i = 5;
    return i;
 }
```

Non Void Method Call Mutator

- The non void method call mutator removes method calls to non void methods. Their return value is replaced by the Java Default Value for that specific type.

Type	Default Value
boolean	false
int byte short long	0
float double	0.0
char	'\u0000'
Object	null

```
public int someNonVoidMethod() {
   return 5;
}
public void foo() {
   int i = someNonVoidMethod();
   // do more stuff with I
 }
```

```
public int someNonVoidMethod() {
   return 5;
}
public void foo() {
  int i = 0;
  // do more stuff with i
}
```

Constructor Call Mutator

- The constructor call mutator replaces constructor calls with null values.

```
public Object foo() {
    Object o = new Object();
    return o;
}
```

```
public Object foo() {
    Object o = null;
    return o;
}
```

Reports

- **Killed**
- **Lived**
- **No coverage**

  No coverage is the same as **Lived** except there were no tests that exercised the line of code where the mutation was created.
- **Non viable**

  A **non viable** mutation is one that could not be loaded by the JVM as the bytecode was in some way invalid. PIT tries to minimize the number of non-viable mutations that it creates.
- **Timed Out**

  A mutation may **time out** if it causes an infinite loop, such as removing the increment from a counter in a for loop
- **Memory error**

  A **memory error** might occur as a result of a mutation that increases the amount of memory used by the system, or may be the result of the additional memory overhead required to repeatedly run your tests in the presence of mutations. If you see a large number of memory errors consider configuring more heap space for the tests.
- **Run error**

  A **run error** means something went wrong when trying to test the mutation. Certain types of non viable mutation can currently result in an run error. If you see a large number of run errors this is probably be an indication that something went wrong.

Incremental analysis

- PIT can track changes to code and tests and the results from previous runs. It can then use this information to avoid re-running analysis when the results can be logically inferred.

- Good for huge codebases

Using Piteclipse

1. Open Eclipse
2. Search for ‘PIT’ in the eclipse marketplace
3. Install
4. Go to **Window -> show view**
5. Search for PIT and enable all views
6. **Window -> Preferences -> PitTest** for options
7. Right click on project of choice **Run As -> PIT Mutation Test**
8. View results in the PIT Summary and Pit Mutations view