Mutation Testing

Software Testing
Mutation Testing

• What is mutation testing?
• Mutation operators
• Steps of mutation testing
• Conclusions
What is Mutation Testing?

- **Mutation Testing** is a type of software testing where we mutate (change) certain statements in the source code and check if the test cases are able to find the errors.

- Each **mutant** is a copy of the program under test, usually with a **small syntactic change**, which is interpreted as a fault:
  - It is a type of **white box testing** which is mainly used for unit testing.
  - The **changes** in mutant program are kept **extremely small**, so it does not affect the overall objective of the program.
The Goal of Mutation Testing

- The goal of Mutation Testing is to assess the quality of the test cases which should be robust enough to fail mutant code.
  - In other words, the goal of mutation testing is to find faults on the system under test
- Thus, a test suite is more or less effective depending on its ability to find faults on the system under test
- This method is also called as Fault based testing strategy as it involves creating fault in the program
Mutation Testing - Example

\[ a := b + c \]

\[ a := c + c \]
\[ a := b - c \]

\[ P + \text{Syntactic change} = m \]

\[ m : \text{syntactically correct} \]
A program and four mutants - Example

<table>
<thead>
<tr>
<th>Version</th>
<th>Code</th>
<th>Test data (a,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P (original)</td>
<td>int sum(int a, int b) { return a + b; }</td>
<td></td>
</tr>
<tr>
<td>Mutant 1</td>
<td>int sum(int a, int b) { return a - b; }</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>Mutant 2</td>
<td>int sum(int a, int b) { return a * b; }</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>Mutant 3</td>
<td>int sum(int a, int b) { return a / b; }</td>
<td>(-1, 0)</td>
</tr>
<tr>
<td>Mutant 4</td>
<td>int sum(int a, int b) { return a + b++; }</td>
<td>(-1, -1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 1)</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(0, 0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Error</td>
<td>0</td>
</tr>
<tr>
<td>(-1, 0)</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>Error</td>
<td>1</td>
</tr>
<tr>
<td>(-1, -1)</td>
<td>-2</td>
<td>0</td>
<td>-1</td>
<td>1</td>
<td>-2</td>
</tr>
</tbody>
</table>
Mutation Coverage

• Given a mutant $m$ of a derivation $d$, a test is said to \textit{kill} the mutant if and only if this test produces a different output on $m$ than on $d$.

• \textbf{Mutation coverage} requires every mutant to be killed by at least one test.
Different types of Mutants

- **Stillborn mutants**: Syntactically incorrect, killed by compiler
  - Example: \( x = a ++ b \)
- **Trivial mutants**: Killed by almost any test case
- **Equivalent mutant**: Always acts in the same behavior as the original program,
  - Example: \( x = a + b \) and \( x = a - (-b) \)

- Those mutants are interesting which behave differently than the original program, and we do not have test cases to identify them (to cover those specific changes)
Example of an Equivalent mutant

Original program

```c
int index=0;
while (...) {
    ...;
    index++;
    if (index==10) 
        break;
}
```

A mutant

```c
int index=0;
while (...) {
    ...;
    index++; 
    if (index>=10) 
        break;
}
```
In Mutation Testing:

1. We take a program and a test suite generated for that program (using other test techniques)

2. We create a number of similar programs (mutants), each differing from the original in one small way, i.e., each possessing a fault
   - E.g., replacing an addition operator by a multiplication operator

3. The original test data are then run on the mutants

4. If test cases detect differences in mutants, then the mutants are said to be dead (killed), and the test set is considered adequate
Basic Ideas (II)

• A mutant remains live either
  – Because it is equivalent to the original program (functionally identical although syntactically different – called an equivalent mutant) or,
  – The test set is inadequate to kill the mutant

• In the latter case, the test data need to be augmented (by adding one or more new test cases) to kill the live mutant

• For the automated generation of mutants, we use mutation operators, that is predefined program modification rules (i.e., corresponding to a fault model)
Some Mutation Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Substitution of a variable $x$ by $abs(x)$</td>
</tr>
<tr>
<td>ACR</td>
<td>Substitution of a variable array reference by a constant</td>
</tr>
<tr>
<td>AOR</td>
<td>Arithmetic operator replacement ($a+b$ by $a-b$)</td>
</tr>
<tr>
<td>CRP</td>
<td>Substitution of a constant value</td>
</tr>
<tr>
<td>ROR</td>
<td>Relational operator replacement ($A$ and $B$ by $A$ or $B$)</td>
</tr>
<tr>
<td>RSR</td>
<td>Return statement substitution ($return$ $5$ by $return$ $0$)</td>
</tr>
<tr>
<td>SDL</td>
<td>Removal of a sentence</td>
</tr>
<tr>
<td>UOI</td>
<td>Unary operator insertion (instead of $x$, write $-x$)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example of Mutation Operators (I)

- Constant replacement
- Scalar variable replacement
- Scalar variable for constant replacement
- Constant for scalar variable replacement
- Array reference for constant replacement
- Array reference for scalar variable replacement
- Constant for array reference replacement
- Scalar variable for array reference replacement
- Array reference for array reference replacement
- Source constant replacement
- Data statement alteration
- Comparable array name replacement
- Arithmetic operator replacement
- Relational operator replacement
- Logical connector replacement
- Absolute value insertion
- Unary operator insertion
- Statement deletion
- Return statement replacement
Example of Mutation Operators (II)

- Specific to **object-oriented** programming languages:
  - Replacing a type with a compatible subtype (inheritance)
  - Changing the access modifier of an attribute, a method
  - Changing the instance creation expression (inheritance)
  - Changing the order of parameters in the definition of a method
  - Changing the order of parameters in a call
  - Removing an overloading method
  - Reducing the number of parameters
  - Removing an overriding method
  - Removing a hiding Field
  - Adding a hiding field
Absolute Value Insertion

Each arithmetic expression is modified by functions abs(), negAbs(), and failOnZero().

Example:

\[ x = 3 \times a \quad \text{Original} \]

\[ x = 3 \times \text{abs}(a), \]
\[ x = 3 \times -\text{abs}(a), \]
\[ x = 3 \times \text{failOnZero}(a); \]

Mutants
Each occurrence of one of the arithmetic operators +, -, *, /, **, and % is replace by each of the other operators, and special operators leftOp, rightOp, and mod.

Example:
\[
\begin{align*}
x &= a + b \\
x &= a - b \\
x &= a * b \\
x &= a / b \\
x &= a ** b \\
x &= a \\
x &= b \\
x &= a \% b
\end{align*}
\]
Relational Operator Replacement

Each occurrence of one of the relational operators (<, <=, >, >=, =, !=) is replaced by each of the other operators and by falseOp and trueOp.

Example:
if (m > n)
if (m >= n),
if (m < n),
if (m <= n),
if (m == n),
if (m != n),
if (false),
if (true)

Mutants
Conditional Operator Replacement

Each occurrence of each logical operator (&&, ||, &, |, ^) is replaced by each of the other operators, and falseOp, trueOp, leftOp, and rightOp.

Example:

```java
if (a && b)
if (a || b),
if (a & b),
if (a | b),
if (a ^ b),
if (false),
if (true),
if (a),
if (b)
```

Mutants
Shift Operator Replacement

Each occurrence of one of the shift operators (<<, >>, and >>>) is replaced by each of the other operators, and the special operator leftOp.

Example:

\[ x = m << a \]

\[ x = m >> a, \]
\[ x = m >>> a, \]
\[ x = m \]

Mutants
Each occurrence of each bitwise logical operator (&, |, and ^) is replaced by each of the other operators, and leftOp and rightOp.

Example:

\[
\begin{align*}
x &= m \& n \\
x &= m \mid n, \\
x &= m \uparrow n, \\
x &= m, \\
x &= n
\end{align*}
\]

Mutants
Assignment Operator

Each occurrence of one of the assignment operators (+=, -=, *=, /=, %=, &=, !=, *=, <<=, >>=, >>>=) is replaced by each of the other operators.

Example:

x += 3
x -= 3,

x *= 3,

x /= 3,

x %= 3,

...
Unary Operator Insertion

Each unary operator (+, -, !, ~) is inserted before each expression of the correct type.

Example:

\[ x = 3 \times a \]

\[ x = 3 \times +a, \quad x = 3 \times -a, \quad x = +3 \times a, \quad x = -3 \times a \]

Mutants
Unary Operator Deletion

Each unary operator (+, -, !, ~) is deleted.

Example:
if !(a > - b)

\[
\begin{align*}
&\text{if } (a > -b), \\
&\text{if } !(a > b)
\end{align*}
\]

Mutants
Scalar Variable Replacement

Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.

Example:

\[ x = a \times b \]
\[ x = a \times a, \]
\[ a = a \times b, \]
\[ x = x \times b, \]
\[ x = a \times x, \]
\[ x = b \times b, \]
\[ b = a \times b \]
Each statement is replaced by a special `Bomb()` function

**Example:**

```plaintext
x = a * b
```

```
Bomb()
```

```
Mutant
```
Delta’s represent syntactic modifications. In fact, each of them will be embedded in a different program version, a mutant.
Discussion of the Example

• Mutant 3 is equivalent as, at this point, \( \text{minVal} \) and \( A \) have the same value

• Mutant 1: In order to find an appropriate test case to kill it, we must
  1. Reach the fault seeded during execution reachability)
     1. Always true (i.e., we can always reach the seeded fault)
  2. Cause the program state to be incorrect (Infection) • \( A <> B \)
  3. Cause the program output and/or behavior to be Incorrect (Propagation) (\( B<A \) = false)
Strong Mutation Coverage

• Requires each mutant be **strongly killed** by at least one test.

• Given a mutant m for a program P and a test t, t is said to **strongly kill** m if and only if the output of t on P is different from the output of t on m.
Assumptions

• What about more complex errors, involving several statements?

• Let’s discuss two assumptions:
  – **Competent programmer assumption**: They write programs that are nearly correct
  – **Coupling effect assumption**: Test cases that distinguish all programs differing from a correct one by only simple errors is so sensitive that they also implicitly distinguish more complex errors
Another Example

• Specification:
  – The program should prompt the user for a positive integer in the range 1 to 20 and then for a string of that length.
  – The program then prompts for a character and returns the position in the string at which the character was first found or a message indicating that the character was not present in the string.
... 
found := FALSE;
i := 1;
while(not(found)) and (i <= x) do begin // x is the length
    if a[i] = c then
        found := TRUE
    else
        i := i + 1
end
if (found)
    print("Character %c appears at position %i");
else
    print("Character is not present in the string");
end
...
### Mutation Testing Example: Test Set 1

<table>
<thead>
<tr>
<th>Input</th>
<th>Expected Output (oracle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>The input integer should be between 1 and 20</td>
</tr>
<tr>
<td>a[ ]</td>
<td>Character x appears at position 1</td>
</tr>
<tr>
<td>c</td>
<td>Character is not present in the string</td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>1 x x</td>
<td>found</td>
</tr>
<tr>
<td>1 x a</td>
<td>not found</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mutation Testing Example: Mutant 1
(for Test Set 1)

- Replace `Found := FALSE;` with `Found := TRUE;`
- Re-run original test data set
- Note: It is better in Mutation Testing to make only one small change at a time to avoid the danger of introduced faults with interfering effects (masking)
- Failure: “character a appears at position 1” instead of saying “character is not present in the string”
- Mutant 1 is killed (since Output <> Oracle)

```plaintext
found := FALSE; TRUE;
i := 1;
while(not(found)) and (i <= x) do begin
  if a[i] = c then
    found := TRUE
  else
    i := i + 1
end
if (found)
  print(“Character %c appears at position %i”);
else
  print(“Character is not present in the string”);
end
```
Mutation Testing Example: Mutant 2 (for Test Set 1)

- **Replace** `i:=1;` with `x:=1;`
- **Will our original test data (test set 1) reveal the fault?**
  - No, our original test data set fails to reveal the fault (because the `x` value was 1 in the second test case of test set 1)
- **As a result of the fault, only position 1 in string will be searched for. So what should we do?**
- **In our test set, we need to increase our input string length and search for a character further along it**
- **We modify the test set 1 and create a new test set 2 (next) so as**
  - To preserve the effect of earlier tests
  - To make sure the live mutant (#2) is killed

```plaintext
Int i=1;
...
found := FALSE;
\textcolor{red}{i := 1; x := 1;}
while(not(found)) and (i <= x) do begin
  if a[i] = c then
    found := TRUE
  else
    i := i + 1
end
if (found)
  print("Character %c appears at position %i");
else
  print("Character is not present in the string");
end
```
## Mutation Testing Example: Test Set 2

<table>
<thead>
<tr>
<th>Input</th>
<th>Actual Output Response</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>a</td>
<td>Input Integer between 1 and 20</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>Character x appears at position 1</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>Character does not occur in string</td>
</tr>
<tr>
<td>3</td>
<td>xCv</td>
<td>Character v appears at position 3</td>
</tr>
<tr>
<td></td>
<td>v</td>
<td>(this test case will kill the mutant in the previous slide)</td>
</tr>
</tbody>
</table>
Mutation Testing Example: Mutant 3
(for Test Set 2)

- \(i := i + 1\); is replaced with \(i := i + 2\);
- Again, our test data (test set 2) fails to kill the mutant
- We must augment the test set 2 and create a new test set 3 (next) to search for a character in the middle of the string
- With the new test set, mutant 3 can be killed
- Many other changes could be made on this short piece of code, e.g., changing array reference, changing the \(\leq\) relational operator

```plaintext
found := FALSE;
i := 1;
while(not(found)) and (i <= x) do begin
  if a[i] = c then
    found := TRUE
  else
    i := i + 1 - 2
end
if (found)
  print(“Character \%c appears at position \%i”);
else
  print(“Character is not present in the string”);
end
```
### Mutation Testing Example: Test Set 3

<table>
<thead>
<tr>
<th>Input</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Response</td>
</tr>
<tr>
<td>x</td>
<td>a</td>
</tr>
<tr>
<td>25</td>
<td>Input Integer between 1 and 20</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>xCv</td>
</tr>
<tr>
<td>3</td>
<td>xCv</td>
</tr>
<tr>
<td></td>
<td>Character x appears at position 1</td>
</tr>
<tr>
<td></td>
<td>Character does not occur in string</td>
</tr>
<tr>
<td></td>
<td>Character v appears at position 3</td>
</tr>
<tr>
<td></td>
<td>Character C appears at position 2 (this test case will kill the mutant in the previous slide)</td>
</tr>
</tbody>
</table>
boolean isEven (int x) {
    if (x < 0)
        x = 0 - x; // change to x = 0;
    if (float) (x / 2) == ((float) x) / 2.0
        return true;
    else
        return false;
}

Reachability: x < 0
Infection: x != 0
Propagation: x must be odd
Mutation Testing Process

Program P

Input test program → Create mutants → Run equivalence heuristic → Generate test cases → Run tests on P and mutants

Threshold reached?

No → Eliminate ineffective TCs
Yes → P correct?

No → Fix P
Yes → Yes

Define threshold
Kinds of Mutation

• **Value Mutations**: These mutations involve changing the values of constants or parameters (by adding or subtracting values etc), e.g. loop bounds – being one out on the start or finish is a very common error.

• **Decision Mutations**: These involves modifying conditions to reflect potential slips and errors in the coding of conditions in programs. E.g. a typical mutation might be replacing a $>$ by a $<$ in a comparison.

• **Statement Mutations**: These might involve deleting certain lines to reflect omissions in coding or swapping the order of lines of code. There are other operations, e.g. changing operations in arithmetic expressions. A typical omission might be to omit the increment on some variable in a while loop.

• A wide range of mutation operators is possible…
public int Segment(int t[], int l, int u) {
    // Assumes t is in ascending order, and l<u,
    // counts the length of the segment
    // of t with each element l<t[i]<u
    int k = 0;

    for(int i=0; i<t.length && t[i]<u; i++) {
        if(t[i]>l) {
            k++;
        }
    }

    return (k);
}

Mutating to k=1 causes miscounting

Here we might mutate the code to read i=1, a test that would kill this would have t
length 1 and have l < t[0] < u, then the
program would fail to count t[0] and return
0 rather than 1 as a result
public int Segment(int t[], int l, int u) {
    // Assumes t is in ascending order, and l<u,
    // counts the length of the segment
    // of t with each element l<t[i]<u
    int k = 0;
    Mutating to t[i]>u will cause miscounting

    for(int i=0; i<t.length && t[i]<u; i++) {
        if(t[i]>l) {
            k++;
        }
    }

    return(k);
}

We can model “one-off” errors in the loop bound by changing this condition to i<=t.length - provided array bounds are checked exactly this will provoke an error on every execution.
```java
public int Segment(int t[], int l, int u) {
    // Assumes t is in ascending order, and l<u, 
    // counts the length of the segment 
    // of t with each element l<t[i]<u 
    int k = 0;

    for (int i = 0; i < t.length && t[i] < u; i++) {
        if (t[i] > l) {
            k++;
        }
    }

    return k;
}
```

Here we might consider deleting this statement (then count would be zero for all inputs, we might also duplicate this line in which case all counts would be doubled.
Steps of mutation testing

• Mutation testing has three main steps:

1. Mutant generation
2. Mutant execution
3. Result analysis
Mutant generation

• Almost each executable instruction of the original program can be mutated with several mutation operators
• Therefore, the number of mutants generated for a normal program may be huge
• The cost of compilation of all mutants may be also significant
Mutant generation: the MuJava tool
Mutant generation: the MuJava tool

- In general, a parser is required to generate mutants:
  - $a+b$ is translated into $a-b$, $a\times b$, $a/b$
  - Then, these program versions are compiled

- MuJava uses “Mutant Schemata Generation”
  - With some operators, it substitutes (at bytecode level) $a+b$ by a OPERATOR $b$
  - Then, all the program versions are directly generated with no need of compiling
Mutant execution

- In this case, the problem is the huge number of test cases that must be executed: each case is executed against the original program and the mutants.

- For testing a simple BankingAccount class, with 96 mutants and 300 test cases, 96*300=28,800 executions are required (with at least 28,800 accesses to the database, etc.)
Mutant execution

- All the outputs must be compared to detect which mutants are killed:
  
  - In the BankingAccount example, the outputs of the 300 test cases with the original and the 96 mutants
  - Actually, killed mutants can be removed for further comparisons
Mutant execution: MuJava
Mutant execution: testooj

• testooj is a relatively user-friendly research tool

• Generates test cases in several formats and according to several generation strategies

• Executes test cases against versions and gives some additional results
Result analysis

- The major difficulties appear with the detection of functionally equivalent mutants

---

**A program and four mutants**

<table>
<thead>
<tr>
<th>Version</th>
<th>Code</th>
<th>Test data $(a,b)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$(1, 1)$</td>
</tr>
<tr>
<td>P (original)</td>
<td><code>int sum(int a, int b) { return a + b; }</code></td>
<td>2</td>
</tr>
<tr>
<td>Mutant 1</td>
<td><code>int sum(int a, int b) { return a - b; }</code></td>
<td>0</td>
</tr>
<tr>
<td>Mutant 2</td>
<td><code>int sum(int a, int b) { return a * b; }</code></td>
<td>1</td>
</tr>
<tr>
<td>Mutant 3</td>
<td><code>int sum(int a, int b) { return a / b; }</code></td>
<td>1</td>
</tr>
<tr>
<td>Mutant 4</td>
<td><code>int sum(int a, int b) { return a + b++; }</code></td>
<td>2</td>
</tr>
</tbody>
</table>
Result analysis

- A functionally equivalent mutant is a mutant which never will be killed

- Actually, the “fault” introduced is not a fault, but a code de-optimization

<table>
<thead>
<tr>
<th>Version</th>
<th>Code</th>
<th>Test data (a, b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P (original)</td>
<td><code>int sum(int a, int b) { return a + b; }</code></td>
<td>(1, 1) 0 -1 -2</td>
</tr>
<tr>
<td>Mutant 4</td>
<td><code>int sum(int a, int b) { return a + b++; }</code></td>
<td>2 0 -1 -2</td>
</tr>
</tbody>
</table>
Result analysis

- The example is an occurrence of the AOIS operator
Result analysis

• Other strategies rely on weak mutation:
  – “Strong” mutation has three conditions:
    • **Reachability** (the instruction must be reached)
    • **Necessity** (once the sentences has been reached, the test case must cause an erroneous state on the mutant)
    • **Sufficiency** (the erroneous state must be propagated to the output)
  • Instead of observing the output of each test case, the idea of weak mutation is to detect changes in intermediate states (reachability + necessity)
Mutation Testing: Discussion

• It measures the quality of test cases
• A tool’s slogan: “Jester - the JUnit test tester”.
• It provides the tester with a clear target (mutants to kill)
• Mutation testing can also show that certain kinds of faults are unlikely (those specified by the fault model), since the corresponding test case will not fail
• It does force the programmer to inspect the code and think of the test data that will expose certain kinds of faults
• It is computationally intensive, a possibly very large number of mutants is generated: random sampling, selective mutation operators (Offutt)
• Equivalent mutants are a practical problem: It is in general an undecidable problem
• Probably most useful at unit testing level
Mutation Testing Tools

- Tools – MuClipse: perhaps the best tool out there…
- Jester: A Mutation Testing tool in Java (Open Source)
- Pester: A Mutation Testing tool in Python (Open Source)
- Nester: A Mutation Testing tool in C# (Open Source)
## Offutt’s Mutations for Inter-Class Testing

<table>
<thead>
<tr>
<th>Language Feature</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Control</td>
<td>AMC</td>
<td>Access modifier change</td>
</tr>
<tr>
<td></td>
<td>IHD</td>
<td>Hiding variable deletion</td>
</tr>
<tr>
<td></td>
<td>IHI</td>
<td>Hiding variable insertion</td>
</tr>
<tr>
<td></td>
<td>IOD</td>
<td>Overriding method deletion</td>
</tr>
<tr>
<td></td>
<td>IOP</td>
<td>overriding method calling position change</td>
</tr>
<tr>
<td></td>
<td>IOR</td>
<td>Overriding method rename</td>
</tr>
<tr>
<td></td>
<td>ISK</td>
<td>super keyword deletion</td>
</tr>
<tr>
<td></td>
<td>IPC</td>
<td>Explicit call of a parent’s constructor deletion</td>
</tr>
<tr>
<td>Inheritance</td>
<td>PNC</td>
<td>new method call with child class type</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>PMD</td>
<td>Instance variable declaration with parent class type</td>
</tr>
<tr>
<td></td>
<td>PPD</td>
<td>Parameter variable declaration with child class type</td>
</tr>
<tr>
<td></td>
<td>PRV</td>
<td>Reference assignment with other comparable type</td>
</tr>
<tr>
<td>Overloading</td>
<td>OMR</td>
<td>Overloading method contents change</td>
</tr>
<tr>
<td></td>
<td>OMD</td>
<td>Overloading method deletion</td>
</tr>
<tr>
<td></td>
<td>OAO</td>
<td>Argument order change</td>
</tr>
<tr>
<td></td>
<td>OAN</td>
<td>Argument number change</td>
</tr>
<tr>
<td>Java-Specific Features</td>
<td>JTD</td>
<td>this keyword deletion</td>
</tr>
<tr>
<td></td>
<td>JSC</td>
<td>static modifier change</td>
</tr>
<tr>
<td></td>
<td>JID</td>
<td>Member variable initialization deletion</td>
</tr>
<tr>
<td></td>
<td>JDC</td>
<td>Java-supported default constructor creation</td>
</tr>
<tr>
<td>Common Programming Mistakes</td>
<td>EOA</td>
<td>Reference assignment and content assignment replacement</td>
</tr>
<tr>
<td></td>
<td>EOC</td>
<td>Reference comparison and content comparison replacement</td>
</tr>
<tr>
<td></td>
<td>EAM</td>
<td>Accessor method change</td>
</tr>
<tr>
<td></td>
<td>EMM</td>
<td>Modifier method change</td>
</tr>
</tbody>
</table>
Conclusions

• Mutation is an excellent testing technique
• Mutation testing can be a useful addition to the test process.
• From the point of view of research, it is mature
• From the industry point of view, user friendly tools are required
• Mutation is also applied at other levels: black-box, components, web services, models…