MTAT.03.159: Software Testing

Lecture 04: Mutation Testing
Static Analysis (Inspection) and Defect Estimation (Textbook Ch. 10 & 12)

Spring 2017

Dietmar Pfahl
email: dietmar.pfahl@ut.ee
Structure of Lecture 04

- Mutation Testing
- Lab 3
- Static Analysis
- Defect Estimation
- Lab 4
- Lab 5
Mutation Testing (Fault-Based Testing)

Original Program → Mutant Program

Test Cases Applied to Both Original & Mutant Program

Output is compared. If results for original and mutant program are same, mutant is KILLED.
Assessing Test Suite Quality

• Idea
  – I make \( n \) copies of my program, each copy with a known number \( m_n \) of (unique) faults
  – Assume introduced faults are exactly like real faults in every way
  – I run my test suite on the programs with seeded faults ...
    • ... and the tests reveal 20% of the introduced faults

• What can I infer about my test suite?
Mutation Testing Procedure

1. Take a program and test data generated for that program
2. Create a number of similar programs (mutants), each differing from the original in a small way
3. The original test data are then run through the mutants
4. If tests detect all changes in mutants, then the mutants are dead and the test suite adequate
   Otherwise: Create more test cases and iterate 2-4 until a sufficiently high number of mutants is killed
Mutation Testing – Terminology

- **Mutant** – new version of the program with a small deviation (=fault) from the original version
- **Killed** mutant – new version detected by the test suite
- **Live** mutant – new version *not* detected by the test suite
Examples of Mutation Operations

- Change relational operator (<, >, …)
- Change logical operator (||, &, …)
- Change arithmetic operator (*, +, -, …)
- Change constant name / value
- Change variable name / initialisation
- Change (or even delete) statement
- …

More examples:
http://pitest.org/quickstart/mutators/
Example Mutants

```
if (a || b)
    c = a + b;
else
    c = 0;
if (a && b)
    c = a + b;
else
    c = 0;
```

```
if (a || b)
    c = a * b;
else
    c = 0;
```
Types of Mutants

Not interesting:

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

Those mutants are interesting which behave differently than the original program, and we do not (yet) have test cases to identify them.
Equivalent Mutants

```java
if (a == 2 && b == 2)
    c = a + b;
else
    c = 0;

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

```java
if (a == 2 && b == 2)
    c = a * b;
else
    c = 0;

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

```java
nbrs = new int[range]
public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i <= range; i++)
        if a[i] > a[imax]
            imax:= i;
return imax;
}
```

**Program returns the index of the array element with the maximum value.**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Program Example

nbrs = new int[range]
public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i <= range; i++)
        if a[i] > a[imax]
            imax := i;
    return imax;
}

Program returns the index of the array element with the maximum value.
Variable Name Mutant

nbrs = new int[range]
public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i <= range; i++)
        if i > a[imax]
            imax := i;
    return imax;
}

Program returns the index of the array element with the maximum value.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Relational Operator Mutant

nbrs = new int[range]

public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i <= range; i++)
        if a[i] >= a[imax]
            imax := i;
    return imax;
}

Need a test case with two identical max entries in a[], e.g., (1, 3, 3)
Variable Operator Mutant

```java
nbrs = new int[range]
public int max(int[] a) {
    int imax := 0;
    for (int i = 0; i < range; i++)
        if a[i] > a[imax]
            imax:= i;
return imax;
}
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Need a test case detecting wrong loop counting
Mutation Testing

Assumptions

• Competent programmer hypothesis:
  – Programs are nearly correct
    • Real faults are small variations from the correct program
    • => Mutants are reasonable models of real faulty programs

• Coupling effect hypothesis:
  – Tests that find simple faults also find more complex faults
    • Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults too
Mutation Testing Tool: PIT

Real world mutation testing
PIT is a state of the art mutation testing system, providing gold standard test coverage for Java and the jvm. It's fast, scalable and integrates with modern test and build tooling.

Get Started
## Default Mutation Operators in PIT

<table>
<thead>
<tr>
<th>Mutation operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditionals Boundary</td>
<td>Replaces relational operators with their boundary counterpart (e.g., <code>&lt;</code> becomes <code>&lt;=</code>, <code>&gt;=</code> becomes <code>&gt;</code>, etc.).</td>
</tr>
<tr>
<td>Negate Conditionals</td>
<td>Replaces all conditionals with their negated counterpart (e.g., <code>==</code> becomes <code>!=</code>, <code>&lt;</code> becomes <code>&gt;=</code>, etc.).</td>
</tr>
<tr>
<td>Math</td>
<td>Replaces binary arithmetic operations from either integer or floating-point arithmetic with another operation (e.g., <code>+</code> becomes <code>-</code>, <code>*</code> becomes <code>/</code>, etc.).</td>
</tr>
<tr>
<td>Increments</td>
<td>Replaces increments of local variables with decrements and vice versa.</td>
</tr>
<tr>
<td>Invert Negatives</td>
<td>Inverts the negation of integer and floating point numbers.</td>
</tr>
<tr>
<td>Return Values</td>
<td>Changes the return value of a method depending on the return type (e.g., non-null return values are replaced with null, integer return values are replaced with 0, etc.).</td>
</tr>
<tr>
<td>Void Method Call</td>
<td>Removes method calls to void methods.</td>
</tr>
</tbody>
</table>
Structure of Lecture 04

- Mutation Testing
- Lab 3
- Static Analysis
- Defect Estimation
- Lab 4
- Lab 5
Lab 3 – Mutation Testing

Lab 3 (week 29: Mar 14 - Mar 15) - Mutation Testing (10%)  

Lab 3 Instructions & Tools

Submission Deadlines:
- Tuesday Labs: Monday, 20 Mar, 23:59
- Wednesday Labs: Tuesday, 21 Mar, 23:59

- Penalties apply for late delivery: 50% penalty, if submitted up to 24 hours late; 100 penalty, if submitted more than 24 hours late

SUT: Minimum Binary Heap (incl. Test code)

Mutation Testing Tool: PIT
Lab 3 – Mutation Testing (cont’d)

Instructions

Mutation Testing Tool: PIT

SUT: Minimum Binary Heap (incl. Test code)

Mutants

Mutation Testing:
Run tests, kill mutants
Add tests, kill more mutants, detect faults

Report:
Detected faults
Mutation coverage
Code coverage

Improved Test Suite
Lab 3 – Mutation Testing (cont’d)

This lab is the result of a BSc thesis by Cornelia Efros.

We need your feedback:

Short questionnaire (online)

Participation is anonymous and voluntary (and highly appreciated)
Structure of Lecture 04

- Mutation Testing
- Lab 3
- Static Analysis
- Defect Estimation
- Lab 4
- Lab 5
Static Analysis

- Document Review (manual)
  - Different types

- Static Code Analysis (automatic)
  - Structural properties / metrics / etc.

Lab 4

Lab 5
Reviews complement testing
Question

What is better?
Review (Inspection) or Test?
Review Metrics

Basic
- Size of review items
  - Pages, LOC
- Review time & effort
- Number of defects found
- Number of slipping defects found later

Derived
- Defects found per review time or effort
  - Efficiency
- Defects found per artifact (review item) size
  - Effectiveness
- Size per time or effort
Defect Containment Measures

Fixing Cost per Defect

Total Fixing Cost
Relative Cost of Faults

Empirical Results: Inspection & Test

Table 3
Average values of effectiveness and efficiency for defect detection

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Code</th>
<th>Inspection effectiveness (%)(^*)(^\dagger)</th>
<th>Inspection efficiency(^*)(^\dagger)</th>
<th>Testing effectiveness(^\ddagger) (%)</th>
<th>Testing efficiency(^\ddagger) (%)</th>
<th>Different faults found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hetze(^\dagger)(^\ddagger)</td>
<td>17</td>
<td>37.3</td>
<td>–</td>
<td>47.7; 46.7</td>
<td>1.62; 2.07</td>
<td>Yes</td>
</tr>
<tr>
<td>Myers(^\ddagger)</td>
<td>18</td>
<td>38.0</td>
<td>0.8</td>
<td>30.0; 36.0</td>
<td>–</td>
<td>Yes</td>
</tr>
<tr>
<td>Basili and Selby(^\ddagger)</td>
<td>2</td>
<td>54.1</td>
<td>Dependent on software type</td>
<td>54.6; 41.2</td>
<td>–</td>
<td>Yes</td>
</tr>
<tr>
<td>Kamsties and Lott(^\dagger)(^\ddagger)</td>
<td>19</td>
<td>43.5</td>
<td>2.11</td>
<td>47.5; 47.4</td>
<td>4.69; 2.92</td>
<td>Partly (for some types)</td>
</tr>
<tr>
<td>Roper et al.(^\ddagger)</td>
<td>20</td>
<td>32.1</td>
<td>1.06</td>
<td>55.2; 57.5</td>
<td>2.47; 2.20</td>
<td>Yes</td>
</tr>
<tr>
<td>Laitenberger(^\dagger)</td>
<td>11</td>
<td>38</td>
<td>–</td>
<td>9#</td>
<td>–</td>
<td>No</td>
</tr>
<tr>
<td>So et al.(^\dagger)</td>
<td>12</td>
<td>17.9, 34.6</td>
<td>0.16; 0.26</td>
<td>43.0</td>
<td>0.034</td>
<td>Yes</td>
</tr>
<tr>
<td>Runeson and Andrews(^\ddagger)</td>
<td>13</td>
<td>27.5</td>
<td>1.49</td>
<td>37.5</td>
<td>1.8</td>
<td>Yes</td>
</tr>
<tr>
<td>Juristo and Vegas(^\ddagger)</td>
<td>14</td>
<td>20.0</td>
<td>–</td>
<td>37.7; 35.5</td>
<td>–</td>
<td>Partly (for some types)</td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Andersson et al.(^\dagger)</td>
<td>15</td>
<td>53.5</td>
<td>5.05</td>
<td>41.8</td>
<td>2.78</td>
<td>Yes for one version, no for the other</td>
</tr>
<tr>
<td>Case studies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conradi et al.(^\dagger)</td>
<td>16</td>
<td>–</td>
<td>0.82</td>
<td>–</td>
<td>0.013</td>
<td>–</td>
</tr>
<tr>
<td>Berling and Thelin(^\dagger)</td>
<td>3</td>
<td>86.5 (estimated)</td>
<td>0.68 (0.13)</td>
<td>80</td>
<td>0.10</td>
<td>Yes</td>
</tr>
</tbody>
</table>

\(^*\) Percent of the artifact’s defects that are detected.

\(^\dagger\) Single entries involve code reading; multiple entries in one cell are reported in this order: code reading, Fagan inspection.

\(^\ddagger\) Detected defects per hour.

\(^\ddagger\) Testing is conducted in sequence after the inspection.

Source:
Empirical Results: Inspection & Test

Table 3

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Code</th>
<th>Study</th>
<th>Inspection effectiveness (%)(^*)(^{+})</th>
<th>Inspection efficiency(^1)(^{+})</th>
<th>Testing effectiveness(s)(^$)</th>
<th>Testing efficiency(^1)(^{$})</th>
<th>Different faults found</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Hetzel(^17)</td>
<td>37.3</td>
<td>–</td>
<td>47.7; 46.7</td>
<td>1.62; 2.07</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Myers(^18)</td>
<td>38.0</td>
<td>0.8</td>
<td>30.0; 36.0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Basili and Selby(^2)</td>
<td>54.1</td>
<td>Dependent on software type</td>
<td>54.6; 41.2</td>
<td>–</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kamsties and Lott(^19)</td>
<td>43.5</td>
<td>2.11</td>
<td>47.5; 47.4</td>
<td>4.69; 2.92</td>
<td>Partly (for some types)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Roper et al.(^20)</td>
<td>32.1</td>
<td>1.06</td>
<td>55.2; 57.5</td>
<td>2.47; 2.20</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Laitenberger(^11)</td>
<td>38</td>
<td>–</td>
<td>9(^#)</td>
<td>–</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>So et al.(^12)</td>
<td>17.9, 34.6</td>
<td>0.16; 0.26</td>
<td>43.0</td>
<td>0.034</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Runeson and Andrews(^13)</td>
<td>27.5</td>
<td>1.49</td>
<td>37.5</td>
<td>1.8</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Juristo and Vegas(^14)</td>
<td>20.0</td>
<td>–</td>
<td>37.7; 35.5</td>
<td>–</td>
<td>Partly (for some types)</td>
</tr>
<tr>
<td>Design</td>
<td>Andresson et al.(^15)</td>
<td>53.5</td>
<td>5.05</td>
<td>41.8</td>
<td>2.78</td>
<td>Yes for one version, no for the other</td>
<td></td>
</tr>
<tr>
<td>Case studies</td>
<td></td>
<td>Conradi et al.(^16)</td>
<td>–</td>
<td>0.82</td>
<td>–</td>
<td>0.013</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Berling and Thelin(^3)</td>
<td>86.5 (estimated)</td>
<td>0.68 (0.13)</td>
<td>80</td>
<td>0.10</td>
<td>Yes</td>
</tr>
</tbody>
</table>

\(^*\) Percent of the artifact's defects that are detected.

\(^{+}\) Single entries involve code reading: multiple entries in one cell are reported in this order: code reading, Fagan inspection.

\(^\$\) Detected defects per hour.

\(^\#\) Single entries involve functional testing: multiple entries in one cell are reported in this order: functional test, structural test.

\(^\$\) Testing is conducted in sequence after the inspection.

Source:
Empirical Results: Inspection & Test

**Table 3**

Average values of effectiveness and efficiency for defect detection

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Code</th>
<th>Effectiveness (%)</th>
<th>Inspection efficiency</th>
<th>Testing effectiveness §</th>
<th>Testing efficiency §</th>
<th>Different faults found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hetzel</td>
<td>37.2</td>
<td>–</td>
<td>37.7; 46.7</td>
<td>1.62; 2.07</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Myers</td>
<td>38.0</td>
<td>–</td>
<td>30.0; 36.0</td>
<td>–</td>
<td>Yes</td>
<td>–</td>
</tr>
<tr>
<td>Basili and Selby</td>
<td>54.1</td>
<td>Dependent on software type</td>
<td>54.0; 41.2</td>
<td>–</td>
<td>–</td>
<td>Yes</td>
</tr>
<tr>
<td>and Lott</td>
<td>50.3</td>
<td>1.52</td>
<td>47.5; 47.4</td>
<td>3.07; 1.92</td>
<td>Partly (for some types)</td>
<td></td>
</tr>
<tr>
<td>Roper et al.</td>
<td>32.1</td>
<td>1.06</td>
<td>55.2; 57.5</td>
<td>2.47; 2.20</td>
<td>Yes</td>
<td>–</td>
</tr>
<tr>
<td>Laitenberger</td>
<td>66</td>
<td>–</td>
<td>9</td>
<td>–</td>
<td>–</td>
<td>No</td>
</tr>
<tr>
<td>So et al.</td>
<td>17.9; 34.6</td>
<td>0.16; 0.26</td>
<td>43.0</td>
<td>0.034</td>
<td>Yes</td>
<td>–</td>
</tr>
<tr>
<td>Jonsson and Andrews</td>
<td>27.5</td>
<td>1.49</td>
<td>37.5</td>
<td>1.8</td>
<td>Yes</td>
<td>–</td>
</tr>
<tr>
<td>Jristo and Vegas</td>
<td>20.0</td>
<td>–</td>
<td>37.7; 35.5</td>
<td>–</td>
<td>Partly (for some types)</td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td>Andersson et al.</td>
<td>53.5</td>
<td>5.05</td>
<td>41.8</td>
<td>2.78</td>
<td>Yes for one version, no for the other</td>
</tr>
<tr>
<td>Case studies</td>
<td>Conradi et al.</td>
<td>–</td>
<td>0.82</td>
<td>–</td>
<td>0.013</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Berling and Thelin</td>
<td>86.5 (estimated)</td>
<td>0.68 (0.13)</td>
<td>80</td>
<td>0.10</td>
<td>Yes</td>
</tr>
</tbody>
</table>

* Percent of the artifact’s defects that are detected.
† Single entries involve code reading; multiple entries in one cell are reported in this order: code reading, Fagan inspection.
‡ Detected defects per hour.
§ Single entries involve functional testing; multiple entries in one cell are reported in this order: functional test, structural test.
# Testing is conducted in sequence after the inspection.

### Empirical Results: Inspection & Test

#### Table 3
Average values of effectiveness and efficiency for defect detection

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Code</th>
<th>Study</th>
<th>Inspected effectiveness (%)</th>
<th>Inspected efficiency</th>
<th>Testing effectiveness</th>
<th>Testing efficiency</th>
<th>Different faults found</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Hetzel(^1)</td>
<td>37.7(^7)</td>
<td>9.6</td>
<td>37.7; 46.7</td>
<td>1.62; 2.07</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Myers(^1)</td>
<td>38.0</td>
<td>0.6</td>
<td>30.0; 36.0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Basli and Selby(^2)</td>
<td>54.1</td>
<td>Dependent on software type</td>
<td>54.0; 41.2</td>
<td>–</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and Lott(^1)</td>
<td>43.5</td>
<td>2.14</td>
<td>50.3</td>
<td>1.52</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Roper et al.(^2)</td>
<td>32.1</td>
<td>1.06</td>
<td>80</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Laitenberger(^1)</td>
<td>0.9</td>
<td></td>
<td>17.9, 34.6</td>
<td>0.16; 0.26</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>So et al.(^3)</td>
<td>22.7</td>
<td></td>
<td>39.6</td>
<td>1.40</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>van Essen and Andrews(^1)</td>
<td>56.1</td>
<td></td>
<td>37.5</td>
<td>1.8</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Juristo and Vegas(^1)</td>
<td>56.1</td>
<td></td>
<td>37.7; 35.5</td>
<td>–</td>
<td>Partly (for some types)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Roper et al.(^2)</td>
<td>17.3</td>
<td></td>
<td>75.8; 71.4</td>
<td>–</td>
<td>Partly (for some types)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>So et al.(^3)</td>
<td>41.8</td>
<td>2.78</td>
<td>80</td>
<td>0.10</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Berling and Thelin(^3)</td>
<td>86.5 (estimated)</td>
<td>0.68 (0.13)</td>
<td>80</td>
<td>0.10</td>
<td>Yes</td>
</tr>
</tbody>
</table>

* Percent of the artifact’s defects that are detected.
+ Single entries involve code reading; multiple entries in one cell are reported in this order: code reading, Fagan inspection.
+ Detected defects per hour.
+ Single entries involve functional testing; multiple entries in one cell are reported in this order: functional test, structural test.
+ Testing is conducted in sequence after the inspection.

1st Entry: Functional Testing
2nd Entry: Structural Testing

Source:
Inspection Process

Origin: Michael Fagan (IBM, early 1970’s)

Approach: Checklist-based

Phases
- Overview, Preparation, Meeting, Rework, Follow-up
- Fault searching at meeting! – Synergy

Roles
- Author (designer), reader (coder), tester, moderator

Classification
- Major and minor
# Empirical Results: Inspection & Test

## Table 3

### Average values of effectiveness and efficiency for defect detection

<table>
<thead>
<tr>
<th>Study</th>
<th>Inspection effectiveness (%)</th>
<th>Inspection efficiency</th>
<th>Testing effectiveness</th>
<th>Testing efficiency</th>
<th>Different faults found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experiments</td>
<td>Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hétzel</td>
<td>37.3</td>
<td>-</td>
<td>47.7; 46.7</td>
<td>-</td>
<td>1.62; 2.07</td>
</tr>
<tr>
<td>Myers</td>
<td>38.0</td>
<td>0.8</td>
<td>30.0; 36.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Basili and Selby</td>
<td>54.1</td>
<td>-</td>
<td>54.6; 41.2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Kamsties and Lott</td>
<td>43.5</td>
<td>2.11</td>
<td>47.5; 47.4</td>
<td>4.69; 2.92</td>
<td>3.07; 1.92</td>
</tr>
<tr>
<td>Roper et al.</td>
<td>32.1</td>
<td>1.06</td>
<td>55.2; 57.5</td>
<td>2.47; 2.20</td>
<td>-</td>
</tr>
<tr>
<td>Laitenberger</td>
<td>17.9</td>
<td>0.16; 0.26</td>
<td>43.0</td>
<td>0.034</td>
<td>-</td>
</tr>
<tr>
<td>So et al.</td>
<td>17.9</td>
<td>0.16; 0.26</td>
<td>43.0</td>
<td>0.034</td>
<td>-</td>
</tr>
<tr>
<td>Runeson and Andrews</td>
<td>27.5</td>
<td>1.29</td>
<td>37.5</td>
<td>1.8</td>
<td>-</td>
</tr>
<tr>
<td>Juristo and Vegas</td>
<td>20.0</td>
<td>-</td>
<td>37.7; 35.5</td>
<td>-</td>
<td>Partly (for some types)</td>
</tr>
<tr>
<td>Design</td>
<td>Andersson et al.</td>
<td>53.5</td>
<td>5.05</td>
<td>41.8</td>
<td>2.78</td>
</tr>
<tr>
<td>Case studies</td>
<td>Conradi et al.</td>
<td>-</td>
<td>0.82</td>
<td>-</td>
<td>0.013</td>
</tr>
<tr>
<td></td>
<td>Berling and Thelin</td>
<td>86.5 (estimated)</td>
<td>0.68 (0.13)</td>
<td>80</td>
<td>0.10</td>
</tr>
</tbody>
</table>

* Percent of the artifact’s defects that are detected.
† Single entries involve code reading; multiple entries in one cell are reported in this order: code reading, Fagan inspection.
‡ Detected defects per hour.
§ Single entries involve functional testing; multiple entries in one cell are reported in this order: functional test, structural test.
# Testing is conducted in sequence after the inspection.

Source:
Inspections – Empirical Results

- **Requirements defects** – no data; but: reviews potentially good since finding defects early is cheaper.
- **Design defects** – reviews are both more efficient and more effective than testing.
- **Code defects** - functional or structural testing is more efficient and effective than reviews.
  - May be complementary regarding types of faults.
- **Generally, reported effectiveness is low**
  - Reviews find 25-50% of an artifact’s defects.
  - Testing finds 30-60% of defects in the code.
Summary: Why Review?

• Main objective
  • Detect faults

• Other objectives
  • Inform
  • Educate
  • Learn from (other’s) mistakes → Improve!

• (Undetected) faults may affect software quality negatively – during all steps of the development process!
Tool 1: Gerrit

- Supports Code Review Process
Tool 1: Gerrit Example (1)

Review Changed Code
Tool 1: Gerrit Example (2)

Vote

Cover Message

Publish
Code
Review

Switching to a Yeast based dough is a good idea, but please recheck the quantities.

Patch Comments:
PizzaDough.txt
Line 5:
1 cup yeast? That seems like a lot. Are you sure you don't mean 1 tbsp?
Tool 1: Gerrit Example (3)

Verify
Change
Vote
Cover Message

Change Id: Ifb5296ebc1b3a7da26057c0d47dd1cc1d0e
Change to a proper, yeast based pizza dough.

Change Id: Ifb5296ebc1b3a7da26057c0d47dd1cc1d0e
Cover Message:
Looks Good, Test Pass.
Tool 2: FindBugs

- Supports Static Code Analysis
Reading Techniques

- Ad hoc
- Checklist-based
- Defect-based
- Usage-based
- Perspective-based
Ad-hoc / Checklist-based / Defect-based Reading

<table>
<thead>
<tr>
<th>Ad Hoc</th>
<th>Checklist</th>
<th>Defect-based Reading</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Omission</strong></td>
<td><strong>Omission</strong></td>
<td><strong>Data Type Inconsistencies</strong></td>
</tr>
<tr>
<td>Functionality</td>
<td>Missing Functionality</td>
<td>1 Identify all data objects mentioned...</td>
</tr>
<tr>
<td>Performance</td>
<td>Missing Performance</td>
<td>1a. Are all data objects mentioned...</td>
</tr>
<tr>
<td>Interface</td>
<td>Missing Environment</td>
<td>...</td>
</tr>
<tr>
<td>Environment</td>
<td>Missing Interface</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Commission</th>
<th><strong>Commission</strong></th>
<th><strong>Incorrect Functionality</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ambiguity</td>
<td>Ambiguous Information</td>
<td>1 For each functional requirement identify...</td>
</tr>
<tr>
<td>Inconsistency</td>
<td>Inconsistent Information</td>
<td>1a. Are all values written to each input...</td>
</tr>
<tr>
<td>Incorrect</td>
<td>Incorrect or Extra Func.</td>
<td>...</td>
</tr>
<tr>
<td>Wrong</td>
<td>Wrong Section</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Ambiguities or Missing Functionality</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Identify the required precision, response...</td>
</tr>
<tr>
<td>1a. Are all required precisions indicated?</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Usage-Based Reading

1 – Prioritize Use Cases (UCs)
2 – Select UC with highest priority
3 – Track UC’s scenario through the document under review
4 – Check whether UC’s goals are fulfilled, needed functionality provided, interfaces are correct, and so on (report issues detected)
5 – Select next UC

Source:
Usage-Based Reading

Table 1

<table>
<thead>
<tr>
<th>Fault class</th>
<th>Mean</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UBR</td>
<td>CBR</td>
</tr>
<tr>
<td>All faults*</td>
<td>5.6</td>
<td>4.1</td>
</tr>
<tr>
<td>Class A faults*</td>
<td>2.6</td>
<td>1.3</td>
</tr>
<tr>
<td>Class B faults</td>
<td>2.1</td>
<td>1.4</td>
</tr>
<tr>
<td>Class C faults</td>
<td>0.9</td>
<td>1.4</td>
</tr>
<tr>
<td>Class A + Class B faults*</td>
<td>4.7</td>
<td>2.8</td>
</tr>
</tbody>
</table>

*Statistically significant at a 95% level

Table 2

<table>
<thead>
<tr>
<th>Fault class</th>
<th>Mean</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UBR</td>
<td>CBR</td>
</tr>
<tr>
<td>All faults (38)</td>
<td>0.31</td>
<td>0.25</td>
</tr>
<tr>
<td>Class A faults (13)*</td>
<td>0.43</td>
<td>0.24</td>
</tr>
<tr>
<td>Class B faults (14)</td>
<td>0.31</td>
<td>0.24</td>
</tr>
<tr>
<td>Class C faults (11)</td>
<td>0.18</td>
<td>0.30</td>
</tr>
<tr>
<td>Class A + Class B faults (27)*</td>
<td>0.37</td>
<td>0.24</td>
</tr>
</tbody>
</table>

*Statistically significant at a 95% level

Comparison of UBR with Checklist-Based Reading (CBR)

Source:
Perspective-based Reading

- Scenarios
- Purpose
  - Decrease overlap (redundancy)
  - Improve effectiveness
Structure of Lecture 04

• Mutation Testing
• Lab 3
• Static Analysis
• Defect Estimation
• Lab 4
• Lab 5
Quality Prediction

- Based on product and process properties
  - Quality = Function(Code Size or Complexity)
  - Quality = Function(Code Changes)
  - Quality = Function(Test Effort)
  - Quality = Function(Detected \#Defects)

- Based on detected defects
  - **Capture-Recapture Models**
  - Reliability Growth Models

Quality defined as: Undetected \#Defects
Capture-Recapture – Defect Estimation
Capture-Recapture – Defect Estimation
Capture-Recapture – Defect Estimation

- Situation: Two inspectors are assigned to inspect the same product
  - $d_1$: #defects detected by Inspector 1
  - $d_2$: #defects detected by Inspector 2
  - $d_{12}$: #defects by both inspectors
  - $N_t$: total #defects (detected and undetected)
  - $N_r$: remaining #defects (undetected)

\[
N_t = \frac{d_1 d_2}{d_{12}} \quad N_r = N_t - (d_1 + d_2 - d_{12})
\]
Capture-Recapture – Example

- Situation: Two inspectors are assigned to inspect the same product
  - \( d_1 \): 50 defects detected by Inspector 1
  - \( d_2 \): 40 defects detected by Inspector 2
  - \( d_{12} \): 20 defects by both inspectors
  - \( N_t \): total defects (detected and undetected)
  - \( N_r \): remaining defects (undetected)

\[
N_t = \frac{d_1 d_2}{d_{12}} = \frac{50 \cdot 40}{20} = 100 \\
N_r = 100 - (50 + 40 - 20) = 30
\]
Advanced Capture-Recapture Models

• Four basic models used for inspections
  • Difference: Degrees of freedom

• Prerequisites for all models
  • All reviewers work independently of each other
  • It is not allowed to inject or remove faults during inspection

Bonus Task in Lab 4
### Advanced Capture-Recapture Models

<table>
<thead>
<tr>
<th>Model</th>
<th>Probability of defect being found is equal across ...</th>
<th>Estimator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Defect</td>
<td>Reviewer</td>
</tr>
<tr>
<td>M0</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mt</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mh</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mth</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Mt Model

Maximum-likelihood:

• Mt = total marked animals (=faults) at the start of the t'th sampling interval
• Ct = total number of animals (=faults) sampled during interval t
• Rt = number of recaptures in the sample Ct
• An approximation of the maximum likelihood estimate of population size (N) is: \( \frac{\sum(Ct \times Mt)}{\sum(Rt)} \)

First resampling:
- M1=50 (first inspector)
- C1=40 (second inspector)
- R1=20 (duplicates)
- N=40*50/20=100

Second resampling:
- M2=70 (first and second inspector)
- C2=40 (third inspector)
- R2=30 (duplicates)
- N=(40*50+40*70)/(20+30)=4800/50=96

Third resampling:
- M3=80 (first, second and third inspector)
- C3=30 (fourth inspector)
- R3=30 (duplicates)
- N=(2000+2800+30*80)/(20+30+30)=7200/80=90
Structure of Lecture 04

- Mutation Testing
- Lab 3
- Static Analysis
- Defect Estimation
- Lab 4
- Lab 5
Lab 4 – Document Inspection & Defect Prediction

Lab 4 (week 31: Mar 28 - Mar 29) - Document Inspection and Defect Prediction (10%)

Lab 4 Instructions
Lab 4 & Sample Documentation

Submission Deadlines:
- Tuesday Labs: Monday, 03,Apr, 23:59
- Wednesday Labs: Tuesday, 04 Apr, 23:59

• Penalties apply for late delivery: 50% penalty, if submitted up to 24 hours late; 100 penalty, if submitted more than 24 hours late

Instructions

Documentation:

Requirements List (User Stories)

Specification
- 2 Screens
- 1 Text
Lab 4 – Document Inspection & Defect Prediction (cont’d)

**Phase A: Individual student work**

- **Instructions**
- **Requirements** (6 User Stories)
- **Inspection of Specification against Requirements**
  - **Issue List** (at least 8 defects)
  - **Specification** (excerpt)
    - 2 screens
    - Text
  - **1 Student**

- **Table columns:**
  - ID, Description, Location, Type, Severity

**Phase B: Pair work**

- **Issue List Student 1**
- **Issue List Student 2**
- **Student Pair**
- **Consolidated Issue List**
- **Remaining Defects Estimation**
Lab 4 – Document Inspection & Defect Prediction (cont’d)

Lab 4 (week 31: Mar 28 - Mar 29) – Defect Estimation (2.5% Bonus)

Lab 4 Bonus Task Instructions
Lab 4 & SciLab Scripts

Phase C:
SciLab scripts

Consolidated Issue List (from Phase B)

Input File (Defects)

Script 1 (M0)

Script 2 (Mt)

Script 3 (Mh)

Report:
- Input Data
- 3 Estimates
- Discussion of Assumptions
Lab 4: Must work in pairs to get full marks!
Structure of Lecture 04

- Mutation Testing
- Lab 3
- Static Analysis
- Defect Estimation
- Lab 4
- Lab 5
Lab 5 – Static Code Analysis

Lab 5 (week 33: Apr 11 - Apr 12) - Static Code Analysis (10%)

Lab 5 Instructions & Tools

Submission Deadlines:
- Tuesday Labs: Monday, 17 Apr, 23:59
- Wednesday Labs: Tuesday, 18 Apr, 23:59
- Penalties apply for late delivery: 50% penalty, if submitted up to 24 hours late; 100 penalty, if submitted more than 24 hours late
Lab 5 – Static Code Analysis (cont’d)

Instructions

Static Code Analysis:
Find issues and determine whether or not it is a false positive

SUT:
HospitalSystem

Static Code Analyzer:
FindBugs (Eclipse plugin)

Report:
10 Issues (Bugs?) with Analysis and discussion

Issue detected:
- True Positive
- False Positive
- Undecidable?
Bug finding is about pointing out programming mistakes: bad practice, coding errors, unexpected behavior. One interesting example of bugs that static analysis can find is null pointer dereference.

```java
int i=0;
String s = null;
if (i>0) {
    s = "positive";
}
if (s.contains("pos")) {
    System.out.println(s);
}
```

This code will compile but at runtime a null pointer exception will be thrown, the String `s` being null and calling the method "contains". Static analysis tools, for instance FindBugs can find this bug and report it.
Bug finding is about pointing out programming mistakes: bad practice, coding errors, unexpected behavior. One interesting example of bugs that static analysis can find is security vulnerabilities.

Static analysis can be applied to find security flaws in code. With dataflow analysis, it is possible to follow the propagation of input data, and thus detect possible code injection.

Example:

```java
public static void main(String args[]) {
    File f = new File(args[0]);
    f.open();
    //...
}
```

This program opens a file with an argument entered in command line. The fact that we use this argument to open a file is just an example, the important fact is that we use directly an input without validation, which constitutes a serious security vulnerability. Dataflow analysis can detect this kind of flaw by finding the source of data inputs, following their propagation until their use in a sensitive instruction (like creating a File object).

Next Week

• Lab 2:
  – Submit assignment (either Monday or Tuesday by 23:59)

• Lecture 5:
  – Test Lifecycle, Test Levels, Test Tools

• In addition to do:
  – Read textbook chapters 10 and 12 (available via OIS)