LTAT.05.006: Software Testing


Spring 2019
Lab 8 – Student Feedback (n=59)

Q1: The goals of the lab were clearly defined and communicated
Q2: The tasks of the lab were clearly defined and communicated
Q3: The instructions of the lab were appropriate and helpful
Q4: The tools used in the lab were appropriate and useful
Q5: Compared to the previous labs, the homework assignment was more difficult
Q6: Overall, what I learned in the lab is relevant for working in the software industry
Q7: Overall, the lab was interesting and inspiring

Graph showing feedback distribution for each question.
Lectures (J. Liivi 2-111)

- Lecture 1 (14.02) – Introduction to Software Testing
- Lecture 2 (21.02) – Basic Black-Box Testing Techniques
- Lecture 3 (28.02) – BBT advanced: Combinatorial Testing
- Lecture 4 (07.03) – Basic White-Box Testing Techniques
- Lecture 5 (14.03) – Test Lifecycle, Test Tools, Test Automation
- Lecture 7 (28.03) – BBT advanced: State-Transition Testing & Exploratory Testing
- Lecture 8 (04.04) – BBT advanced: Security, Usability and A/B Testing
- Lecture 9 (11.04) – WBT advanced: Data-Flow Testing / Mutation Testing
- Lecture 11 (25.04) – Quality Estimation / Test Documentation, Organisation and Process Improvement (Test Maturity Model)
- 02.05 - no lecture
- Lecture 12 (09.05) – Industry Guest Lecture 1 (90 min)
- Lecture 13 (16.05) – Industry Guest Lecture 2 (60 min) / Exam Preparation
Industry Guest Lectures – in May

Guest Lecture 1 (09 May):
• **Modern Testing Principles** by Risko Ruus, Rush Street Interactive

Guest Lecture 2 (16 May):
• **Security Testing of Mobile Applications** by Kristiina Rahkema, Nestri Solutions OÜ
White-Box Testing Techniques

- Control-Flow Testing
- Data-Flow Testing
- Mutation Testing
- Symbolic Execution
- Static Code Analysis
- Inspections/Reviews
Structure of Lecture 10

- Symbolic Execution
- Static Code Analysis
- Lab 10
- Manual Static Analysis
  - Document Inspections
  - Code Review
Data Flow & Control Flow Criteria

- All c-uses
- All def-use paths
- All paths

- All uses
- All def-use paths
- All paths

- All p-uses, some c-uses
- All p-uses, some p-uses
- All c-uses, some p-uses

Weaker

All branches

Stronger

# tests

All c-uses
All def
All p-uses
White-Box Testing Coverage Criteria – How to find Test Cases?

• Branch coverage: Analyse requirements yielding an outcome at each predicate node contained in a path
• Consider all requirements together
• Guess a value that will satisfy these requirements

---

• Can we improve on this?

Remember Lab 4
How to cover this path?

'Magic' Function $M$

$M(x, y) \rightarrow \text{result}$

with $x, y: \text{int (32 bit)}$

... if $(x - 100 \leq 0)$
  if $(y - 100 \leq 0)$
    if $(x + y - 200 == 0)$
      crash();
    ...
...

...
How to cover this path?

'Magic' Function $M$

$M(x, y) \rightarrow \text{result}$

with $x, y: \text{int (32 bit)}$

1st if = true: $x \leq 100$
2nd if = true: $y \leq 100$
3rd if = true: $x + y = 200$

$(\text{if}_1) \land (\text{if}_2) \land (\text{if}_3) \rightarrow M(x=100, y=100) \rightarrow \text{crash()}$
Symbolic Execution

• How to find test inputs to exercise a path?
  – Need to make a choice at each predicate node
  – Give a symbolic value to each variable
  – Walk the path collecting requirements on symbolic input

• Then have a set of (in)equalities to solve
  – Tool support: Constraint Solver

• Example: Find test cases for each path by symbolic execution \( \rightarrow \) …
Program finds the smallest factor of a prime decomposition of a given number p.

Examples: [8 -> 2, fact] [13 -> 13, prim]

Questions:
- What is the Cyclomatic Complexity?
- How many linearly independent paths to cover?
- How many paths to cover for 100% statement coverage?
- How many paths to cover for 100% branch coverage?
- How many test cases (minimum) needed for statement, branch, linearly independent, multiple-condition, all-path coverage?

```c
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0
        AND
        q < sqrt p)
        do
            q := q+1
        ;
    if (p mod q = 0)
    then
        print(q,'is factor')
    else
        print(p,'is prime')
    ;
} 
```
Program

smallest(int p) (*p>2*)
{
    int q = 2;

    while(p mod q > 0 AND q < sqrt p)
    do
        q := q+1
    ;
    if (p mod q = 0)
    then
        print(q,'is factor')
    else
        print(p,'is prime')
    ;
}

Program finds the smallest factor of a prime decomposition of a given number p.

Cyclomatic Complexity:

\[ v(G) = 9 - 8 + 2 = 2 + 1 = 3 \]

Sets of linearly independent paths:

Example Set 1:
1-2-3-5-6-8 1-1-0-0-1-0-0-1-1
1-2-3-4-3-5-7-8 1-1-1-1-1-1-1-0-0
1-2-3-5-7-8 1-1-0-0-1-1-1-0-0
1-2-3-5-7-8 1-1-0-0-1-1-1-0-0

Example Set 2:
1-2-3-5-6-8 1-1-0-0-1-0-0-1-1
1-2-3-4-3-5-6-8 1-1-1-1-1-1-0-0-1-1
1-2-3-5-7-8 1-1-0-0-1-1-1-0-0-1-1
1-2-3-5-7-8 1-1-0-0-1-1-1-0-0-1-1
Symbolic Execution Tree

Program

```c
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0
        AND
        q < sqrt(p))
        do
            q := q+1;
    if (p mod q = 0)
        then
            print(q, `is factor`)
        else
            print(p, `is prime`)
    
}
```

CFG

PC: path condition

$p: \text{x}$

$\text{PC: x>2}$

Solve path conditions →
Test input!
Symbolic Execution Tree

Program

```c
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0 AND q < sqrt p)
        q := q+1;
    if (p mod q = 0)
        print(q,'is factor');
    else
        print(p,'is prime');
}
```

CFG

Solve path conditions →
Test input!
Symbolic Execution Tree

Program

```c
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0
        AND
        q < sqrt p)
        do
            q := q+1
        if (p mod q = 0)
        then
            print(q, 'is factor')
        else
            print(p, 'is prime')

    }
```
Symbolic Execution Tree

Program

```
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0
        AND
        q < sqrt p)
        do
            q := q+1
        ;
    if (p mod q = 0)
        then
            print(q, 'is factor')
        else
            print(p, 'is prime')
    ;
}
```

CFG

PC: path condition

```
while-predicate = false
(don't enter while-loop)
```

Solve path conditions ➔ Test input!

Explanation of PC:

while-predicate = false can happen if:
- p mod q = 0 -> x mod 2 = 0 -> x = 2 * n
or
- sqrt(p) <= q -> sqrt(x) <= 2 -> x <= 4
Symbolic Execution Tree

Program

```c
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0
        AND
        q < sqrt p)
        do
            q := q+1;
        if (p mod q = 0)
            then
                print(q, 'is factor');
            else
                print(p, 'is prime');
    }
}
```

Solve path conditions → Test input!

PC: path condition

1-2

while-predicate = false
(don’t enter while-loop)

1-2-3

PC: x > 2 and
(2 ≥ sqrt(x) or
x = 2*n)

p: x q: 2

if-predicate = false
(take the left branch)

1-2-3-5

p: x is prime q: 2
PC: x > 2 and
(2 ≥ sqrt(x) and
x = 2*n + 1)
Symbolic Execution Tree

Program

```
smallest(int p) {p>2}{
    int q = 2;
    while(p mod q > 0)
        q := q + sqrt(p);
    if (p mod q = 0)
        print(q, 'is factor');
    else
        print(p, 'is prime');
}
```

Solve path conditions → Test input!

PC: path condition

1-2-3

\[ p: x \]
\[ PC: x > 2 \]

1-2

\[ p: x \]
\[ q: 2 \]

1-2-3

\[ PC: x > 2 \]

1-2-3-5

\[ p: x \]
\[ q: 2 \]

PC: \( x > 2 \) and
\[ 2 \geq \sqrt{x} \) or \( x = 2n \)

1-2-3-5-7-8

Explanation of PC:
if-predicate = false happens if:
- \( p \ mod \ q > 0 \) ->
- \( x \ mod \ 2 > 0 \) ->
- \( x \ mod \ 2 = 1 \) ->
- \( x = 2n + 1 \)
Symbolic Execution Tree

```
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0
        AND
        q < sqrt p)
    do
        q := q+1
    ;
    if (p mod q = 0)
        print(q, 'is factor')
    else
        print(p, 'is prime')
}
```

Solve path conditions → Test input!

- \( p: x \quad q: 2 \)
- PC: \( x > 2 \) and \( (2 \geq \sqrt{x} \) or \( x = 2n) \)

- \( p: x \quad q: 2 \)
- PC: \( x > 2 \) and \( (2 \geq \sqrt{x} \) and \( x = 2n + 1) \)

\[ \sqrt{x} \leq 2 \rightarrow x \leq 4 \rightarrow 2n + 1 \leq 4 \rightarrow n = 1 \rightarrow x = 3 \]
Symbolic Execution Tree

Program

`smallest(int p) (*p>2*)` { `int q = 2;` while(p mod q > 0 `AND` q < sqrt p) do `q := q+1;` if (p mod q = 0) then `print(q,'is factor');` else `print(p,'is prime');` }

Solve path conditions → Test input!

PC: path condition

1-2-3

1-2

1-2-3-5

1-2-3-5-7-8

sqrt(x) <= 2 → x <= 4 → 2*n + 1 <= 4 → n = 1 → x = 3
[also satisfies PC: x > 2]
Symbolic Execution Tree

Program
\[
\text{smallest}(\text{int } p) \quad (\ast p \geq 2 \ast) \\
\{ \\
\quad \text{int } q = 2; \\
\quad \textbf{while}(p \mod q > 0) \\
\quad \quad \quad \text{AND} \\
\quad \quad \quad q < \text{sqrt}(p) \\
\quad \quad \textbf{do} \\
\quad \quad \quad q := q+1 \\
\quad \quad \textbf{if} \ (p \mod q = 0) \\
\quad \quad \quad \textbf{then} \\
\quad \quad \quad \quad \text{print}(q, \text{'is factor'}) \\
\quad \quad \quad \textbf{else} \\
\quad \quad \quad \quad \text{print}(p, \text{'is prime'}) \\
\quad \quad \textbf{end} \\
\quad \textbf{end} \\
\}\n\]

while-predicate
\(= \text{false} \)
\((\text{don't enter while-loop})\)

PC: path condition
\(p: x \quad q: 2\)
PC: \(x > 2\)

Solve path conditions \(\rightarrow\)
Test input!
Symbolic Execution Tree

Program

```c
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0
        AND
        q < sqrt(p))
        do
            q := q+1;
        if (p mod q = 0)
            then
                print(q,'is factor');
            else
                print(p,'is prime');
    }
}
```

CFG

1. \( p: x \)  
   - \( PC: x > 2 \)
2. \( p: x \)  
   - \( q: 2 \)
3. \( p: x \)  
   - \( q: 2 \)
   - \( PC: x > 2 \)
4. \( p: x \)  
   - \( q: 2 \)
5. \( p: x \)  
   - \( q: 2 \)
   - \( PC: x > 2 \) and \( x = 2\times n \)
6. \( p: x \)  
   - \( q: 2 \)
   - \( PC: x > 2 \) and \( x = 2\times n \) and \( 2 \geq \sqrt{x} \) or \( x = 2\times n \)
7. \( p: x \)  
   - \( q: 2 \)
   - \( PC: x > 2 \) and \( x = 2\times n \) and \( 2 \geq \sqrt{x} \) or \( x = 2\times n \) and factor

Solve path conditions → Test input!

if-predicate = true (take the right branch)

while-predicate = false (don’t enter while-loop)
Symbolic Execution Tree

Program

```c
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0
        AND
        q < sqrt(p))
        do
            q := q+1
        ;
    if (p mod q = 0)
        then
            print(q, 'is factor')
        else
            print(p, 'is prime')
    }
}
```

CFG

PC: path condition

1-2

1-2-3-5-6-8

1-2

1-2-3

1-2-3-5

1-2-3-5-6

1-2-3-5-6-8

Solve path conditions → Test input!

Explanations of PC:

- if-predicate = true happens if:
  - p mod q = 0 → x mod 2 = 0 → x = 2*n

```c
p: x
PC: x>2

q: 2
PC: x>2

p: x
q: 2
PC: x > 2 and
(2 ≥ sqrt(x) or
x = 2*n)

p: x
q: 2 and factor
PC: x > 2 and
x = 2*n
Symbolic Execution Tree

Program

```c
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0
        AND
        q < sqrt p)
        do
            q := q+1
            ;
    if (p mod q = 0)
        then
            print(q, ´is factor´)
        else
            print(p, ´is prime´)
    }
```

Solve path conditions → Test input!

```
p: x
PC: x>2
```

```
p: x  q: 2
PC: x>2 and
(2 ≥ sqrt(x) or x = 2*n)
```

```
x > 2 and x = 2*n
x in {4, 6, ...}

"2 is factor"
```

```
p: x  q: 2 and factor
PC: x > 2 and
x = 2*n
```

```
x > 2 and x = 2*n  →
1-2-3-5-6-8
```

PC: path condition
Symbolic Execution Tree

Program

```
smallest(int p) (*p>2*)
{
int q = 2;
while(p mod q > 0
    AND
    q < sqrt p)
do
    q := q+1
if (p mod q = 0)
    then
        print(q, 'is factor')
else
        print(p, 'is prime')
}
```

Solve path conditions →
Test input!

while-predicate = true
(enter while-loop)

Continue as voluntary homework ...

\[ p: x > 2 \]
\[ PC: x > 2 \]

\[ p: x \quad q: 2 \]
\[ PC: x > 2 \]

1-2

1-2-3

p: x \quad q: 2
PC: x > 2 and
2 < sqrt(x) and
x <> 2*n
Symbolic Execution Tree

Program

```c
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0
        AND
        q < sqrt p)
    do
        q := q+1
    if (p mod q = 0)
        then
            print(q, 'is factor')
    else
        print(p, 'is prime')
}
```

CFG

PC: path condition

\( p: x \)

PC: \( x > 2 \)

\( p: x \) \( q: 2 \)

PC: \( x > 2 \)

Summary of example ...

Solve path conditions → Test input!
Symbolic Execution Tree

Program

```
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0
        AND
        q < sqrt p)
        do
            q := q+1;
        if (p mod q = 0)
            then
                print(q, 'is factor')
            else
                print(p, 'is prime')
    }
```

Solve path conditions → Test input!

PC: path condition

PC: x > 2

PC: x > 2 and (2 ≥ sqrt(x) or x = 2*n)
Symbolic Execution Tree

Program

\[
\text{smallest}(\text{int } p) \quad (*p \geq 2*)
\]
\[
\{ 
\text{int } q = 2;
\text{while}(p \text{ mod } q > 0 \\
\text{AND} \\
q < \sqrt{p})
\text{do} \\
\text{q := q+1} \\
\text{if } (p \text{ mod } q = 0) \\
\text{then} \\
\text{print}(q, \text{`is factor'}) \\
\text{else} \\
\text{print}(p, \text{`is prime'})
\}
\]

CFG

1-2-3

\[p: x \quad q: 2\]
\[\text{PC: } x > 2 \quad \text{and} \quad (2 \geq \sqrt{x} \text{ or } x = 2n)\]

1-2-3-5

\[p: x \quad q: 2\]
\[\text{PC: } x > 2 \quad \text{and} \quad 2 < \sqrt{x} \quad \text{and} \quad x \not< 2n\]

Solve path conditions → Test input!
Symbolic Execution Tree

Program

```c
smallest(int p) {*
p = 2;
while(p mod q > 0 AND q < sqrt p)
do
    q := q + 1;
if (p mod q = 0)
    print(q, 'is factor');
else
    print(p, 'is prime');
}
```

Solve path conditions → Test input!
Symbolic Execution Tree

Program

```c
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0
        AND
        q < sqrt p)
        do
            q := q+1
    ;
    if (p mod q = 0)
    then
        print(q, 'is factor')
    else
        print(p, 'is prime')
    ;
}
```

Solve path conditions → Test input!

PC: path condition

1-2

p: x
PC: x > 2

1-2-3

p: x q: 2
PC: x > 2 and
(2 ≥ sqrt(x) or
x = 2*n)

1-2-3-5

p: x q: 2
PC: x > 2 and
(2 ≥ sqrt(x) and
x = 2*n + 1)

1-2-3-5-7-8

p: x q: 2
PC: x > 2 and
x = 2*n

1-2-3-5-6-8

Factor

p: x q: 2 and factor

4 ≥ 2*n + 1
n = 1
x = 3

p: x q: 2 and factor

4 ≥ 2*n + 1
n = 1
x = 3

PC: path condition
Symbolic Execution Tree

Program

```c
smallest(int p) (*p>2*)
{
    int q = 2;
    while(p mod q > 0
        AND
        q < sqrt p)
        do
            q := q+1
        ;
    if (p mod q = 0)
        then
            print(q, `is factor`)
        else
            print(p, `is prime`)
    }
}
```

Solve path conditions →
Test input!
Symbolic Execution – Challenges

Main Problems:
• Path Explosion
• Interaction with code out of control of the symbolic execution tool
Symbolic Execution – Challenges

Main Problems:
- Path Explosion
- Interaction with code out of control of the symbolic execution tool

Could be addressed by:
1. Parallelizing the analysis of independent paths
2. Using heuristics to speed-up path coverage (e.g., avoiding to try infeasible paths)
3. Merging similar paths
Symbolic Execution – Tool: KeY

Can be used for interactive debugging with SED (Symbolic Execution Debugger)

More info:
• https://www.key-project.org
Symbolic Execution – Tool: JPF

"JPF is an explicit state software model checker for Java™ bytecode"

More tools:
• https://en.wikipedia.org/wiki/Symbolic_execution
Structure of Lecture 10

• Symbolic Execution
• Static Code Analysis
• Lab 10
• Manual Static Analysis
  • Document Inspections
  • Code Review
Introductory Example

1. int foo() {
2.     Integer x = new Integer(6);
3.     Integer y = bar();
4.     int z;
5.     if (y != null)
6.         z = x.intVal() + y.intVal();
7.     else {
8.         z = x.intVal();
9.         y = x;
10.        x = null;
11.    }
12.    return z + x.intVal();
13. }

Are there any possible null pointer exceptions in this code?
Control-Flow Graph / Data Flow Analysis

1. int foo() {
2.     Integer x = new Integer(6);
3.     Integer y = bar();
4.     int z;
5.     if (y != null)
6.         z = x.intVal() + y.intVal();
7.     else {
8.         z = x.intVal();
9.         y = x;
10.        x = null;
11.    }
12.    return z + x.intVal();
13. }

Integer x = new Integer(6);
Integer y = bar();
int z;
if (y != null)
    z = x.intVal() + y.intVal();
else {
    z = x.intVal();
    y = x;
    x = null;
}
return z + x.intVal();
Null Pointer Analysis ...

- Track each variable in the program at all program points.

- Three states for each variable:
  - null, not-null, and maybe-null.

- Then check if, at each dereference, the analysis has identified whether the dereferenced variable is or might be null.
Control-Flow Graph / Data Flow Analysis

1. int foo() {
2.   Integer x = new Integer(6);
3.   Integer y = bar();
4.   int z;
5.   if (y != null)
6.     z = x.intVal() + y.intVal();
7.   else {
8.     z = x.intVal();
9.     y = x;
10.    x = null;
11.  }
12.  return z + x.intVal();
13. }

Warning: may have null pointer on line 12, because x may be null!
Static Code Analysis – Tool Support

**PMD** - https://pmd.github.io/pmd-6.13.0/index.html

**SonarQube** - https://docs.sonarqube.org/latest/
  - For local analysis, integrated in IDE: SonarLint
  - https://www.sonarlint.org

**FindBugs** – IntelliJ plugin:
https://plugins.jetbrains.com/plugin/3847-findbugs-idea
What FindBugs is …

• Result of a research project at the University of Maryland, USA

• Static analysis tool for Java
  • Not concerned with formatting / coding standards
  • Concentrates on detecting potential bugs and performance issues
  • Can detect many types of common, (manually) hard-to-find bugs
How FindBug works ...

- Uses “bug patterns” to detect potential bugs
- Examples

NullPointerException

```java
Address address = client.getAddress();
if ((address != null) || (address.getPostCode() != null)) {
  ...
}
```

Uninitialized field

```java
public class ShoppingCart {
  private List items;
  public addItem(Item item) {
    items.add(item);
  }
}
```
FindBugs – 200+ Bug Patterns

FindBugs Bug Descriptions

This document lists the standard bug patterns reported by FindBugs version 3.0.1.

Summary

http://findbugs.sourceforge.net/bugDescriptions.html

<table>
<thead>
<tr>
<th>Description</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC: Equals method should not assume anything about the type of its argument</td>
<td>Bad practice</td>
</tr>
<tr>
<td>BIT: Check for sign of bitwise operation</td>
<td>Bad practice</td>
</tr>
<tr>
<td>CN: Class implements Cloneable but does not define or use clone method</td>
<td>Bad practice</td>
</tr>
<tr>
<td>CN: clone method does not call super.clone()</td>
<td>Bad practice</td>
</tr>
<tr>
<td>CN: Class defines clone() but doesn't implement Cloneable</td>
<td>Bad practice</td>
</tr>
<tr>
<td>CNT: Rough value of known constant found</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Co: Abstract class defines covariant compareTo() method</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Co: compareTo()/compareTo() incorrectly handles float or double value</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Co: compareTo()/compareTo() returns Integer.MIN_VALUE</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Co: Covariant compareTo() method defined</td>
<td>Bad practice</td>
</tr>
<tr>
<td>DE: Method might drop exception</td>
<td>Bad practice</td>
</tr>
<tr>
<td>DE: Method might ignore exception</td>
<td>Bad practice</td>
</tr>
<tr>
<td>DMI: Adding elements of an entry set may fail due to reuse of Entry objects</td>
<td>Bad practice</td>
</tr>
<tr>
<td>DMI: Random object created and used only once</td>
<td>Bad practice</td>
</tr>
<tr>
<td>DMI: Don't use removeAll to clear a collection</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Dm: Method invokes System.exit(...)</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Dm: Method invokes dangerous method runFinalizersOnExit</td>
<td>Bad practice</td>
</tr>
<tr>
<td>ES: Comparison of String parameter using == or '=='</td>
<td>Bad practice</td>
</tr>
<tr>
<td>ES: Comparison of String objects using == or '=='</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Eq: Abstract class defines covariant equals() method</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Eq: Equals checks for incompatible operands</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Eq: Class defines compareTo(...) and uses Object.equals()</td>
<td>Bad practice</td>
</tr>
</tbody>
</table>
FindBugs – 200+ Bug Patterns

FindBugs Bug Descriptions

This document lists the standard bug patterns reported by FindBugs version 3.0.1.

Summary

<table>
<thead>
<tr>
<th>Description</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC: Equals method should not assume anything about the type of its argument</td>
<td>Bad practice</td>
</tr>
<tr>
<td>BIT: Check for sign of bitwise operation</td>
<td>Correctness</td>
</tr>
<tr>
<td>CN: Class implements Cloneable but does not define or use clone method</td>
<td>Malicious code vulnerability</td>
</tr>
<tr>
<td>CN: clone method does not call super.clone()</td>
<td>Multithreaded correctness</td>
</tr>
<tr>
<td>CN: Class defines clone() but doesn't implement Cloneable</td>
<td>Performance</td>
</tr>
<tr>
<td>CN1: Rough value of known constant found</td>
<td>Security</td>
</tr>
<tr>
<td>Co: Abstract class defines covariant compareTo() method</td>
<td>Dodgy code</td>
</tr>
<tr>
<td>Co: compareTo()/compare() incorrectly handles float or double value</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Co: compareTo()/compare() returns Integer.MIN_VALUE</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Co: Covariant compareTo() method defined</td>
<td>Bad practice</td>
</tr>
<tr>
<td>DE: Method might drop exception</td>
<td>Bad practice</td>
</tr>
<tr>
<td>DE: Method might ignore exception</td>
<td>Bad practice</td>
</tr>
<tr>
<td>DMI: Adding elements of an entry set may fail due to reuse of Entry objects</td>
<td>Bad practice</td>
</tr>
<tr>
<td>DMI: Random object created and used only once</td>
<td>Bad practice</td>
</tr>
<tr>
<td>DMI: Don't use removeAll to clear a collection</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Dm: Method invokes System.exit(...)</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Dm: Method invokes dangerous method runFinalizersOnExit</td>
<td>Bad practice</td>
</tr>
<tr>
<td>ES: Comparison of String parameter using == or !=</td>
<td>Bad practice</td>
</tr>
<tr>
<td>ES: Comparison of String objects using == or !=</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Eq: Abstract class defines covariant equals() method</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Eq: Equals checks for incompatible operand</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Eq: Class defines compareTo(...) and uses Object.equals()</td>
<td>Bad practice</td>
</tr>
<tr>
<td>Categories:</td>
<td></td>
</tr>
<tr>
<td>- Bad practice</td>
<td></td>
</tr>
<tr>
<td>- Correctness</td>
<td></td>
</tr>
<tr>
<td>- Malicious code vulnerability</td>
<td></td>
</tr>
<tr>
<td>- Multithreaded correctness</td>
<td></td>
</tr>
<tr>
<td>- Performance</td>
<td></td>
</tr>
<tr>
<td>- Security</td>
<td></td>
</tr>
<tr>
<td>- Dodgy code</td>
<td></td>
</tr>
</tbody>
</table>
FindBugs

Bug Tree

= hierarchical representation of all potential bugs detected in the analyzed Jar files.

When you select a particular bug instance in the top pane, you will see a description of the bug in the "Details" tab of the bottom pane. In addition, the source code pane on the upper-right will show the program source code where the potential bug occurs.

In this example, the bug is a stream object that is not closed. The source code window highlights the line where the stream object is created.

http://findbugs.sourceforge.net/
FindBugs Filters

- `<Bug>`
  - filter out bugs of special type (see previous slide)
- `<Confidence>` (was `<Priority>` up to Version 2.x)
  - High / Normal / Low confidence warning (-> false positive?)
- `<Rank>`
  - Scariest (1-4) / Scary (5-9) / Troubling (10-14) / Concern (15-20)

How to use FindBugs?

• Standalone application
• Eclipse / IntelliJ plug-in
• Integrated into the build process (Ant, Gradle, Maven)
Biggest Challenge when using FindBugs (or any other static code analyzer)?

- Potentially high number of “False Positives”
- Definitions:
  - True Positive: what is marked as potential bug is a true bug
  - False Positive: what is marked as potential bug is not a bug
  - True Negative: what is not marked as a potential bug is indeed not a bug
  - False Negative: what is not marked as a potential bug is actually a bug

Sometimes hard to decide what is actually the case
Structure of Lecture 10

• Symbolic Execution
• Static Code Analysis
• Lab 10
• Manual Static Analysis
  • Document Inspections
  • Code Review
Lab 10 – Static Code Analysis

Lab 10 (week 34: Apr 23 & 24) - Static Code Analysis (10 points)

Lab 10 Instructions & Tools

Submission Deadlines:
- Tuesday Labs: Monday, 29 Apr, 23:59
- Wednesday Labs: Tuesday, 30 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
SUT: HospitalSystem
Static Code Analyzer: FindBugs
Lab 10 – Static Code Analysis (cont’d)

Instructions

Static Code Analyzer: FindBugs (Eclipse/IntelliJ plugin)

List of Defects/Issues (categorized)

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

SUT: HospitalSystem

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.
Lab 10 – Static Code Analysis (cont’d)

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).

Structure of Lecture 10

• Symbolic Execution
• Static Code Analysis
• Lab 10
• Manual Static Analysis
  • Document Inspections
  • Code Review
Static Analysis

- Static Code Analysis (automatic)
  - Structural properties / metrics / etc.
- Document Review (manual)
  - Different types
Reviews complement testing
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
Defect Causal Analysis (DCA)

Organizational Processes → define

Software Constr. (Analyse / Design / Code / Rework)

Artifact → find defects

Defect Database

extract sample of defects

Defect (Fault) Detection (Review / Test)

fix defects

Defect Causal Analysis (DCA)

prioritize & implement actions

Action Team Meeting

propose actions

Causal Analysis Meeting
Question

What is better?

Review/Inspection or Test?
Review/Insp. 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
Defect Containment Measures

What is the effectiveness of Requirements Reviews in this example?
Relative Cost of Faults

Defect Containment Measures

Fixing Cost per Defect

Total Fixing Cost
### Table 3

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

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Study</th>
<th>Inspection effectiveness (%)(^{1,7})</th>
<th>Inspection efficiency (^{1,7})</th>
<th>Testing effectiveness (^{1,8})</th>
<th>Testing efficiency (^{1,8})</th>
<th>Different faults found</th>
</tr>
</thead>
<tbody>
<tr>
<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>Yes</td>
</tr>
<tr>
<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>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>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>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>Laitenberger(^{11})</td>
<td>38.0</td>
<td>–</td>
<td>9(^a)</td>
<td>–</td>
<td>No</td>
</tr>
<tr>
<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>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>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>Andersson 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>
</tr>
<tr>
<td>Case studies</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>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.
\(^a\) Testing is conducted in sequence after the inspection.

Source:
### Empirical Results: Rev./Insp. & Test

#### Table 3

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

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Code</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><strong>Experiments</strong></td>
<td></td>
<td><strong>Study</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hetzel(^1)</td>
<td>37.3</td>
<td>–</td>
<td>47.7; 46.7</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Myers(^2)</td>
<td>38.0</td>
<td>0.8</td>
<td>30.0; 36.0</td>
<td>1.62; 2.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Basili and Selby(^3)</td>
<td>54.1</td>
<td>Dependent on software type</td>
<td>54.6; 41.2</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kamsties and Lott(^4)</td>
<td>43.5</td>
<td>2.11</td>
<td>47.5; 47.4</td>
<td>4.69; 2.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Roper et al.(^5)</td>
<td>32.1</td>
<td>1.06</td>
<td>55.2; 57.5</td>
<td>2.47; 2.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Laitenberger(^6)</td>
<td>38</td>
<td>–</td>
<td>9*</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>So et al.(^7)</td>
<td>17.9, 34.6</td>
<td>0.16; 0.26</td>
<td>43.0</td>
<td>0.034</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Runeson and Andrews(^8)</td>
<td>27.5</td>
<td>1.49</td>
<td>37.5</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Juristo and Vegas(^9)</td>
<td>20.0</td>
<td>–</td>
<td>37.7; 35.5</td>
<td>–</td>
</tr>
<tr>
<td><strong>Design</strong></td>
<td></td>
<td>Andersson et al.(^10)</td>
<td>53.5</td>
<td>5.05</td>
<td>41.8</td>
<td>2.78</td>
</tr>
<tr>
<td><strong>Case studies</strong></td>
<td></td>
<td>Conradi et al.(^11)</td>
<td>–</td>
<td>0.82</td>
<td>–</td>
<td>0.013</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Berling and Thelin(^12)</td>
<td>86.5 (estimated)</td>
<td>0.68 (0.13)</td>
<td>80</td>
<td>0.10</td>
</tr>
</tbody>
</table>

---

**Source:**

---

* Percent of the artifact's defects that are detected.

\(^1\) Single entries involve code reading; multiple entries in one cell are reported in this order: code reading, Fagan inspection.

\(^2\) Detected defects per hour.

\(^3\) 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: Rev./Insp. & Test

## Table 3

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

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Code</th>
<th>Inspect. effectiveness (%)</th>
<th>Inspect. efficiency</th>
<th>Testing effectiveness</th>
<th>Testing efficiency</th>
<th>Different faults found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hetzel</td>
<td>-</td>
<td>37.2</td>
<td>0.6</td>
<td>41.7; 46.7</td>
<td>1.62; 2.07</td>
<td>-</td>
</tr>
<tr>
<td>Myers</td>
<td></td>
<td>38.0</td>
<td></td>
<td>30.0; 36.0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Basili and Selby</td>
<td></td>
<td>54.1</td>
<td>Dependent on software type</td>
<td>54.6; 41.2</td>
<td>3.07; 1.92</td>
<td>Yes</td>
</tr>
<tr>
<td>Roper et al.</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</td>
<td></td>
<td>26</td>
<td>-</td>
<td>9</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td>So et al.</td>
<td></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</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</td>
<td></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>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>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.
§ Testing is conducted in sequence after the inspection.

Source:
Empirical Results: Rev./Insp. & 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>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></td>
<td></td>
<td>Hetzel(^\text{17})</td>
<td>37.7</td>
<td>–</td>
<td>37.7; 74.6</td>
<td>–</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Myers(^\text{18})</td>
<td>38.0</td>
<td>–</td>
<td>30.0; 36.0</td>
<td>1.62; 2.07</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Basli and Selby(^\text{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>Roper et al.(^\text{20})</td>
<td>32.1</td>
<td>1.52</td>
<td>37.5; 1.8</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Laitenberger(^\text{11})</td>
<td>68</td>
<td>–</td>
<td>37.5; 1.8</td>
<td>Partly (for some types)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>So et al.(^\text{13})</td>
<td>17.9, 34.6</td>
<td>0.16; 0.26</td>
<td>37.7; 35.5</td>
<td>Yes for one version, no for the other</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unisson and Andrews(^\text{13})</td>
<td>69.5</td>
<td>1</td>
<td>37.5; 1.8</td>
<td>Partly (for some types)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Juristo and Vegas(^\text{14})</td>
<td>43.5</td>
<td>2.11</td>
<td>37.5; 1.8</td>
<td>Partly (for some types)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Roper et al.(^\text{20})</td>
<td>32.1</td>
<td>1.06</td>
<td>37.5; 1.8</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Laitenberger(^\text{11})</td>
<td>68</td>
<td>–</td>
<td>37.5; 1.8</td>
<td>Partly (for some types)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>So et al.(^\text{13})</td>
<td>17.9, 34.6</td>
<td>0.16; 0.26</td>
<td>37.7; 35.5</td>
<td>Yes for one version, no for the other</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unisson and Andrews(^\text{13})</td>
<td>69.5</td>
<td>1</td>
<td>37.5; 1.8</td>
<td>Partly (for some types)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Juristo and Vegas(^\text{14})</td>
<td>43.5</td>
<td>2.11</td>
<td>37.5; 1.8</td>
<td>Partly (for some types)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Roper et al.(^\text{20})</td>
<td>32.1</td>
<td>1.06</td>
<td>37.5; 1.8</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Effectiveness**

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

**Efficiency**

1st Entry: Doc Reading
2nd Entry: Formal Inspection

Source:
## Empirical Results: Rev./Insp. & Test

### Table 3

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hetze(^1)</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(^2)</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(^3)</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(^4)</td>
<td>43.5, 50.3</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.(^5)</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(^6)</td>
<td>38</td>
<td></td>
<td>9³</td>
<td>–</td>
<td>No</td>
</tr>
<tr>
<td>So et al.(^7)</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(^8)</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(^9)</td>
<td>20.0</td>
<td></td>
<td>37.7; 35.5</td>
<td>37.5; 71.4</td>
<td>Partly (for some types)</td>
</tr>
<tr>
<td>Design</td>
<td>Andersson et al.(^10)</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.(^11)</td>
<td>–</td>
<td>0.82</td>
<td>–</td>
<td>0.013</td>
</tr>
<tr>
<td>Berling and Thelin(^12)</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.
\(^*\) Testing is conducted independently of functional testing; multiple entries in one cell are reported in this order: functional test, structural test.

Source:
Reviews/Insp. – 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
Code Review Tool: Gerrit

- Supports Code Review Process

Standard workflow triggers two steps:
- Code Review
- Verification (of change triggered by review)

Tool integration with Jenkins for automation.
Gerrit Example (1)

Review Changed Code
Gerrit Example (2)

Vote

Cover Message

Publish Code Review
Gerrit Example (3)

Verify
Change
Vote
(after
Rework)

Cover Message
Reading Techniques

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

<table>
<thead>
<tr>
<th>Omission</th>
<th>Omission</th>
<th>Data Type Inconsistencies</th>
</tr>
</thead>
<tbody>
<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>Commission</th>
<th>Incorrect Functionality</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>Ad Hoc</th>
<th>Checklist</th>
<th>Defect-based Reading</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. Identify the required precision, response...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1a. Are all required precisions indicated?</td>
<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>UBR</th>
<th>CBR</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>All faults*</td>
<td>5.6</td>
<td>4.1</td>
<td></td>
</tr>
<tr>
<td>Class A faults*</td>
<td>2.6</td>
<td>1.3</td>
<td>1.0</td>
</tr>
<tr>
<td>Class B faults</td>
<td>2.1</td>
<td>1.4</td>
<td>1.2</td>
</tr>
<tr>
<td>Class C faults</td>
<td>0.9</td>
<td>1.4</td>
<td>0.4</td>
</tr>
<tr>
<td>Class A + Class B faults*</td>
<td>4.7</td>
<td>2.8</td>
<td>1.8</td>
</tr>
</tbody>
</table>

*Statistically significant at a 95% level

**Table 2**

<table>
<thead>
<tr>
<th>Fault class</th>
<th>UBR</th>
<th>CBR</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>All faults (38)</td>
<td>0.31</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>Class A faults (13)*</td>
<td>0.43</td>
<td>0.24</td>
<td>0.17</td>
</tr>
<tr>
<td>Class B faults (14)</td>
<td>0.31</td>
<td>0.24</td>
<td>0.15</td>
</tr>
<tr>
<td>Class C faults (11)</td>
<td>0.18</td>
<td>0.30</td>
<td>0.08</td>
</tr>
<tr>
<td>Class A + Class B faults (27)*</td>
<td>0.37</td>
<td>0.24</td>
<td>0.12</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
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!
Next Week

• Quiz 10:
  • Symbolic Execution, Static Code Analysis, Document Inspections, Code Review

• Lab 10:
  – Static Code Analysis with FindBugs

• Lecture 11:
  – Quality Estimation
  – Test Documentation, Organisation and Process Improvement (Test Maturity Model)