
Spring 2021
Lectures

• Lecture 1 (11.02) – Introduction to Software Testing
• Lecture 2 (18.02) – Basic Black-Box Testing Techniques: Boundary Value Analysis & Equivalence Class Partitioning
• Lecture 3 (25.02) – BBT advanced: Combinatorial Testing
• Lecture 4 (04.03) – Basic White-Box Testing Techniques: Control-Flow Coverage
• Lecture 5 (11.03) – BBT adv.: State-Transition, Metamorphic, Random Testing
• Lecture 6 (18.03) – Test Levels, Test Tools, Test Automation
• Lecture 7 (25.03) – BBT adv.: Exploratory Testing, Behaviour Testing
• Lecture 8 (01.04) – BBT adv.: GUI / Visual Testing, Usability Testing, A/B Testing
• Lecture 9 (08.04) – Data-Flow Testing / Test-Suite Effectiveness: Mutation Testing
• Lecture 10 (15.04) – WBT adv.: Symbolic Execution, Static Code Analysis, Review
• Lecture 11 (22.04) – Defect Estimation / Test Documentation, Organisation and Process Improvement (Test Maturity Model)
• Lectures 12+13 (29.04 + 06.05) – Industry Guest Lectures + Advanced Topics
• Lecture 14 (13.05) – Exam Preparation
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 9
- Manual Static Analysis
  - Document Inspections
  - Code Review
Data Flow & Control Flow Criteria

All c-uses \rightarrow All defs \rightarrow All p-uses

- All c-uses, some p-uses \rightarrow All uses
  - All def-use paths
  - All paths

All branches

Stronger

Weaker

# tests
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) → result
with x, y: int (32 bit)
How to cover this path?

'Magic' Function M

M (x, y) → result
with x, y: int (32 bit)

1st if = true: x <= 100
2nd if = true: y <= 100
3rd if = true: x + y = 200
(if_1) && (if_2) && (if_3) -> M (x=100, y=100) -> 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 → …
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')
    ;
}
```

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

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

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 1-1-1-1-1-1-0-0
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 1-1-1-1-1-0-0-1-1
1-2-3-5-7-8 1-1-0-0-1-1-1-0-0
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

![CFG Diagram]

$p$: x
PC: $x > 2$

Solve path conditions →
Test input!
Symbolic Execution Tree

Program

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

CFG

1 - Solve path conditions ➔
2 - Test input!
Symbolic Execution Tree

Program

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

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

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')
    ;
}
```

PC: path condition

1-2

PC: x>2

1-2-3

PC: x>2

1

p: x  q: 2

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

2

1-2

p: x  q: 2

PC: x > 2 and

(2 ≥ sqrt(x) or

x = 2*n)

3

4

5

6

7

8

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

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')
    ;
}
```

CFG

Solve path conditions →
Test input!

1-2

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

1-2-3

PC: x > 2

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

1-2-3-5

if-predicate = false
(take the left branch)

while loop path

S: p: x
PC: x > 2

1-2

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

1-2-3

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

1-2-3-5

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

1-2-3-5
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

1-2-3-5-7-8

Solve path conditions → Test input!

PC: path condition

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

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

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

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

Explanation of PC:
if-predicate = false happens if:
- p mod q > 0 ->
- x mod 2 > 0 ->
- x mod 2 = 1 ->
- x = 2*n + 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

1-2-3

1-2-3-5

Solve path conditions →
Test input!

PC: path condition

1-2

p: x
PC: x>2

p: x  q: 2
PC: x>2

p: x  q: 2
PC: x>2

p: x  q: 2
PC: x>2

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

sqrt(x) <= 2 →
    x <= 4 →
2*n + 1 <= 4 →
n = 1 →
x = 3
Symbolic Execution Tree

Program

\begin{verbatim}
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')
    ;
}
\end{verbatim}

CFG

Solve path conditions \rightarrow Test input!

\begin{align*}
p: & x > 2 \\
PC: & x > 2
\end{align*}

1-2

\begin{align*}
p: & x \quad q: 2 \\
PC: & x > 2 \quad (2 \geq \sqrt{x} \text{ or } x = 2n)
\end{align*}

1-2-3

\begin{align*}
p: & x \quad q: 2 \\
PC: & x > 2 \quad 2 \geq \sqrt{x} \quad x = 2n + 1
\end{align*}

1-2-3-5

sqrt(x) \leq 2 \quad 2n + 1 \leq 4 \quad n = 1 \quad x = 3

[also satisfies PC: x > 2]

1-2-3-5-7-8

1-2
Symbolic Execution Tree

Solve path conditions →
Test input!

PC: path condition

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')
    ;
}
```

PC: path condition

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

if-predicate = true
(take the right branch)

Solve path conditions → Test input!
Symbolic Execution Tree

Program

```c
int smallest(int p) {
    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")
    }
}
```

CFG

1 - 2 - 3

1-2-3

1-2-3-5

PC: path condition

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

PC: \( x > 2 \) and

\[ (2 \geq \sqrt{x} \text{ or } x = 2n) \]

Explanation of PC:

if-predicate = true happens if:

- \( p \mod q = 0 \rightarrow \)
- \( x \mod 2 = 0 \rightarrow \)
- \( x = 2n \)

Test input!

Solve path conditions →
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

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

x > 2 and x = 2*n →
x in {4, 6, ...}

"2 is factor"

1-2-3-5-6-8

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

Solve path conditions → Test input!

while-predicate = true (enter while-loop)

Continue as voluntary homework ...
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

1

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

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

CFG

```
1 -> 2
2 -> 3
3 -> 4
4 -> 3
5 -> 6
6 -> 7
7 -> 8
8
```

Solve path conditions ➔

Test input!

PC: path condition

\( p: x \)  
PC: \( x > 2 \)

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

\( p: x \ q: 2 \)  
PC: \( x > 2 \)  
\( 2 < \sqrt{x} \) and \( x \neq 2 \times n \)
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!
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

Solve path conditions → Test input!

PC: path condition

1-2

PC: x>2

1-2-3

PC: x>2

1-2-3

p: x q: 2

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

p: x q: 2

PC: x > 2 and
    2 < sqrt(x) and
    x <> 2*n

p: x is prime q: 2

PC: x > 2 and
    (2 ≥ sqrt(x) and
     x = 2*n + 1)
    ⇒ 4 ≥ 2*n + 1
    ⇒ n = 1 ⇒ x = 3

1-2-3-5-7-8

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!
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

Solve path conditions →
Test input!

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

```
p: x  q: 2
PC:  x > 2 and
(2 ≥ sqrt(x) and
 x = 2*n + 1)
⇒ 4 ≥ 2*n + 1
⇒ n = 1 ⇒ x = 3
```

```
p: x  q: 2
PC:  x > 2 and
 x <> 2*n
```

```
p: x  q: 2
PC:  x > 2 and
2 < sqrt(x) and
x <> 2*n
```

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

```
p: x  q: 2
PC:  x > 2 and
2 < sqrt(x) and
x <> 2*n
```

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

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

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

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

PC: path condition
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

More info:
- https://www.key-project.org

Can be used for interactive debugging with SED (Symbolic Execution Debugger)
Symbolic Execution – Tool: JPF

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

More tools:
Structure of Lecture 10

• Symbolic Execution
• Static Code Analysis
• Lab 9
• 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 FindBugs works ...

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

```java
NullPointerException

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

Uninitialized field

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()/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>
</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>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()/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>
</tbody>
</table>

Categories:
- Bad practice
- Correctness
- Malicious code vulnerability
- Multithreaded correctness
- Performance
- Security
- Dodgy code
- …
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.

Method may fail to close stream

The method creates an IO stream object, does not assign it to any fields, pass it to other methods that might close it, or return it, and does not appear to close the stream on all paths out of the method. This may result in a file descriptor leak. It is generally a good idea to use a finally block to ensure that streams are closed.

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 9
• Manual Static Analysis
  • Document Inspections
  • Code Review
Lab 9 – Static Code Analysis

Lab 9 (week 34: Apr 20 & 21) - Static Code Analysis (9 points)

Lab 9 Instructions & Tools

Submission Deadlines:
- Tuesday Labs: Monday, 26 Apr, 23:59
- Wednesday Labs: Tuesday, 27 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

SUT: HospitalSystem

Static Code Analyzer: FindBugs

Instructions
Lab 9 – Static Code Analysis (cont’d)

Instructions

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

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

List of Defects/Issues (categorized)

SUT: HospitalSystem

Static Code Analyzer: FindBugs (Eclipse/IntelliJ plugin)

Issue detected:
- True Positive
- False Positive
- Undecidable?
Lab 9 – 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 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).

Structure of Lecture 10

• Symbolic Execution
• Static Code Analysis
• Lab 9
• 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)

1. Organizational Processes
2. Define action
4. Find defects
5. Defect (Fault) Detection (Review / Test)
6. Extract sample of defects
7. Defect Database
8. Fix defects
9. Artifacts
10. Organizational Processes
11. Prioritize & implement actions
12. Action Team Meeting
13. Propose actions
14. Causal Analysis Meeting
15. Propose actions
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 or review item size
  -> Efficiency
- Defects found out of all defects contained in review item
  -> Effectiveness
# Defect Containment Measures

## Defect Source Distribution

<table>
<thead>
<tr>
<th>Code size</th>
<th>Size units</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>FP</td>
<td></td>
</tr>
</tbody>
</table>

## Origin Activity

<table>
<thead>
<tr>
<th>Activity</th>
<th>%</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>100%</td>
<td>1,000</td>
</tr>
<tr>
<td>Reqs</td>
<td>22%</td>
<td>222</td>
</tr>
<tr>
<td>Design</td>
<td>35%</td>
<td>435</td>
</tr>
<tr>
<td>Code</td>
<td>43%</td>
<td>435</td>
</tr>
<tr>
<td>Test</td>
<td></td>
<td>222</td>
</tr>
</tbody>
</table>

## Found-In Activity

<table>
<thead>
<tr>
<th>Activity</th>
<th>Reviews</th>
<th>Test</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reqs</td>
<td>73</td>
<td>52</td>
<td>24</td>
</tr>
<tr>
<td>Design</td>
<td>116</td>
<td>59</td>
<td>106</td>
</tr>
<tr>
<td>Code</td>
<td>194</td>
<td>142</td>
<td>95</td>
</tr>
<tr>
<td>Test</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Reviews**: 29
- **Test**: 70
- **Use**: 194
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

Fixing Cost per Defect

Total Fixing Cost
## Empirical Results: Rev./Insp. & Test

### Table 3

<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>Hetzel(^1)</td>
<td>17</td>
<td>37.3</td>
<td>–</td>
<td>47.7; 46.7</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Myers(^2)</td>
<td>18</td>
<td>38.0</td>
<td>0.8</td>
<td>30.0; 36.0</td>
<td>1.62; 2.07</td>
<td>Yes</td>
</tr>
<tr>
<td>Basili and Selby(^3)</td>
<td>2</td>
<td>54.1</td>
<td>–</td>
<td>54.6; 41.2</td>
<td>–</td>
<td>Yes</td>
</tr>
<tr>
<td>Kamsties and Lott(^4)</td>
<td>10</td>
<td>43.5; 50.3</td>
<td>2.11; 1.52</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>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(^6)</td>
<td>11</td>
<td>38</td>
<td>–</td>
<td>9(^*)</td>
<td>–</td>
<td>No</td>
</tr>
<tr>
<td>So et al.(^7)</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(^8)</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(^9)</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>Andersson et al.(^10)</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>
</tbody>
</table>

**Design**

<table>
<thead>
<tr>
<th>Case studies</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>Conradi et al.(^11)</td>
<td>–</td>
<td>0.82</td>
<td>–</td>
<td>–</td>
<td>–</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.  
\(^1\) Single entries involve code reading; multiple entries in one cell are reported in this order: code reading, Fagan inspection.  
\(^2\) Defected defects per hour.  
\(^3\) Single entries involve functional testing; multiple entries in one cell are reported in this order: functional test, structural test.  
\(^4\) 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>Inspection effectiveness (%)*</th>
<th>Inspection efficiency</th>
<th>Testing effectiveness*§</th>
<th>Testing efficiency 1, §</th>
<th>Different faults found</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Study</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hetzel17</td>
<td>37.3</td>
<td>–</td>
<td>47.7; 46.7</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Myers18</td>
<td>38.0</td>
<td>0.8</td>
<td>30.0; 36.0</td>
<td>1.62; 2.07</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Basili and Selby2</td>
<td>54.1</td>
<td>Dependent on software type</td>
<td>54.6; 41.2</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Karmstes and Lott19</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>Laitenberger11</td>
<td>38.0</td>
<td>–</td>
<td>9#</td>
<td>–</td>
<td>–</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 Andrews13</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 Vegas14</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 Thelin3</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: 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>Hetzel 17</td>
<td></td>
<td>37.5</td>
<td>8.8</td>
<td>47.7; 46.7</td>
<td>–</td>
<td>Yes</td>
</tr>
<tr>
<td>Myers 18</td>
<td></td>
<td>38.0</td>
<td>8.8</td>
<td>30.0; 36.0</td>
<td>1.62; 2.07</td>
<td>Yes</td>
</tr>
<tr>
<td>Basili and Selby 2</td>
<td></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>Roper et al. 20</td>
<td></td>
<td>43.3</td>
<td>2.41</td>
<td>47.5; 47.4</td>
<td>4.69; 2.92</td>
<td>Partly (for some types)</td>
</tr>
<tr>
<td>Laitenberger 11</td>
<td></td>
<td>50.3</td>
<td>1.52</td>
<td>60.7; 52.8</td>
<td>3.07; 1.92</td>
<td>Partly (for some types)</td>
</tr>
<tr>
<td>So et al. 13</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>Lindgren and Andrews 13</td>
<td></td>
<td>27.5</td>
<td>1.49</td>
<td>37.5</td>
<td>1.8</td>
<td>Yes</td>
</tr>
<tr>
<td>Bristo and Vegas 14</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>Andresson et al. 15</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>Conradi et al. 16</td>
<td></td>
<td>–</td>
<td>0.82</td>
<td>41.8</td>
<td>0.013</td>
<td>–</td>
</tr>
<tr>
<td>Berling and Thelin 3</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.
* Defects detected 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: 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\textsuperscript{1}\textsuperscript{7}</td>
<td>37.5</td>
<td>0.8</td>
<td>47.7; 46.7</td>
<td>–</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Myers\textsuperscript{18}</td>
<td>38.0</td>
<td>0.6</td>
<td>30.0; 36.0</td>
<td>1.62; 2.07</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Basili and Selby\textsuperscript{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.\textsuperscript{20}</td>
<td>43.5</td>
<td>2.11</td>
<td>Partly (for some types)</td>
<td>–</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Laitenberger\textsuperscript{11}</td>
<td>50.3</td>
<td>1.52</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>So et al.\textsuperscript{13}</td>
<td>17.9, 34.6</td>
<td>0.16; 0.26</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Jonsson and Andrews\textsuperscript{13}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Giusto and Vega\textsuperscript{14}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Andersson et al.\textsuperscript{15}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td>Code</td>
<td>Study</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conradi et al.\textsuperscript{16}</td>
<td>37.5</td>
<td>1.8</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Berling and Thelin\textsuperscript{3}</td>
<td>37.7; 35.5</td>
<td>–</td>
<td>Partly (for some types)</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>75.5; 71.4</td>
<td>75.8; 71.4</td>
<td>–</td>
<td>Yes for one version, no for the other</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>41.8</td>
<td>2.78</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>37.5</td>
<td>0.9</td>
<td>0.013</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>80</td>
<td>0.10</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source:

---

Effectiveness

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

Efficiency

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

Code

Design
Empirical Results: Rev./Insp. & Test

### Table 3

Average values of effectiveness and efficiency for defect detection

<table>
<thead>
<tr>
<th>Experiments Code</th>
<th>Study</th>
<th>Inspection effectiveness (%)(^\d) (^\d)</th>
<th>Inspection efficiency (^\d) (^\d)</th>
<th>Testing effectiveness (^\d) (^\d)</th>
<th>Testing efficiency (^\d) (^\d)</th>
<th>Different faults found</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hetzel(^\d)</td>
<td>37.3</td>
<td>-</td>
<td>47.7; 46.7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Myers(^\d)</td>
<td>38.0</td>
<td>0.8</td>
<td>30.0; 36.0</td>
<td>1.62; 2.07</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Basili and Selby(^\d)</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(^\d)</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>50.3</td>
<td>1.52</td>
<td>60.7; 52.8</td>
<td>3.07; 1.92</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Roper et al.(^\d)</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(^\d)</td>
<td>38</td>
<td>-</td>
<td>9(^#)</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>So et al.(^\d)</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(^\d)</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(^\d)</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.(^\d)</td>
<td>53.5</td>
<td>5.05</td>
<td>75.8; 71.4</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.(^\d)</td>
<td>-</td>
<td>0.82</td>
<td>-</td>
<td>0.013</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Berling and Thelin(^\d)</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.
\(^\d\) Single entries involve code reading; multiple entries in one cell are reported in this order: code reading, Fagan inspection.
\(^\d\) Detected defects per hour.
\(^\d\) Single entries involve functional testing; multiple entries in one cell are reported in this order: functional test, structural test.
\(^\d\) Testing is conducted in sequence after the inspection.

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)

Publish
Code Review

Vote

Cover Message

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?

Edit

Publish Comments  Cancel
Gerrit Example (3)

Verify

Change

(after Rework)

Vote

Cover Message
Reading Techniques

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

### Omission
- Functionality
  - Missing Functionality
- Performance
  - Missing Performance
- Interface
  - Missing Environment
- Environment
  - Missing Interface

### Commission
- Ambiguity
  - Ambiguous Information
- Inconsistency
  - Inconsistent Information
- Incorrect
  - Incorrect or Extra Func.
- Wrong
  - Wrong Section

### Data Type Inconsistencies
1. Identify all data objects mentioned...
2a. Are all data objects mentioned...
...

### Incorrect Functionality
1. For each functional requirement identify...
2a. Are all values written to each input...
...

### Ambiguities or Missing Functionality
1. Identify the required precision, response...
2a. Are all required precisions indicated?
...

---

**Ad Hoc**

**Checklist**

**Defect-based Reading**
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
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 ➔ Moodle
  • Symbolic Execution, Static Code Analysis, Document Inspections, Code Review

• Lab 9:
  – Static Code Analysis with FindBugs

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