Software Testing

No issue is meaningful unless it can be put to the test of decisive verification.

C.S. Lewis, 1934

Testing a ballpoint pen

- Does the pen write in the right color, with the right line thickness?
- Is the logo on the pen according to company standards?
- Is it safe to chew on the pen?
- Does the click-mechanism still work after 100 000 clicks?
- Does it still write after a car has run over it?

What is expected from this pen?  
Intended use!!
Goal: develop software to meet its intended use!
But: human beings make mistake!

⇒ Product of any engineering activity must be verified against its requirements throughout its development.

- Verifying bridge = verifying design, construction, process,…

- Software must be verified in much the same spirit. In this lecture, however, we shall learn that verifying software is perhaps more difficult than verifying other engineering products.

We shall try to clarify why this is so.
Outline

- Some notations
- Integration testing
- Component/Unit/Module/Basic testing
- Function testing
- Performance testing
- Acceptance testing
- Installation testing
- Real life examples
Error, Fault, Failure

- Human error
- Can lead to Fault
- Can lead to Failure

Debugging vs Testing

- **Debugging**: to find the bug
- **Testing**: to demonstrate the existence of a fault
  - fault identification
  - fault correction / removal
Types of Faults
(dep. on org. IBM, HP)

- Algorithmic: division by zero
- Computation & Precision: order of op
- Documentation: doc - code
- Stress/Overload: data-str size (dimensions of tables, size of buffers)
- Capacity/Boundary: x devices, y parallel tasks, z interrupts
- Timing/Coordination: real-time systems
- Throughout/Performance: speed in req

Types of Faults

- Recovery: power failure
- Hardware & System Software: modem
- Standards & Procedure: organizational standard; difficult for programmers to follow each other
Unit & Integration Testing

**Objective**: to ensure that code implemented the design properly.

Classes of Integration Testing

- Top-down
- Bottom-up
- Big bang
- Sandwich
Components

- driver
- Component to be tested
- stub
- Boundary conditions
- independent paths
- interface
- ...
- Test cases

January 2006 CUGS, SE, Mariam Kamkar, IDA, LiU
A

B

C

D

E

F

G

Top-down
Bottom-up diagram
Bottom-up

Big-bang
<table>
<thead>
<tr>
<th></th>
<th>Top-down</th>
<th>Bottom-up</th>
<th>Big-bang</th>
<th>Sandwich</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to a basic working program</td>
<td>Early</td>
<td>Late</td>
<td>Late</td>
<td>Early</td>
</tr>
<tr>
<td>Driver needed</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>In part</td>
</tr>
<tr>
<td>Stubs needed</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>In part</td>
</tr>
</tbody>
</table>
Unit Testing

- Code Inspections
- Code Walkthroughs
- Open box testing
- Black box testing
Two Types of Oracles

- **Human**: an expert that can examine an input and its associated output and determine whether the program delivered the correct output for this particular input.

- **Automated**: a system capable of performing the above task.
Balls and Urn

- Testing can be viewed as selecting different colored balls from an urn where:
  - Black ball = input on which program fails.
  - White ball = input on which program succeeds.
- Only when testing is exhaustive is there an “empty” urn.

A program that always fails

A correct program

A typical program
Inspection
(originally introduced by Fagan 1976)

- overview (code, inspection goal)
- preparation (individually)
- reporting
- rework
- follow-up

Inspection (cont)
some classical programming errors

- Use of un-initialized variables
- Jumps into loops
- Non-terminating loops
- Incompatible assignments
- Array indexes out of bounds
- Off-by-one errors
- Improper storage allocation or de-allocation
- Mismatches between actual and formal parameters in procedure calls
Walkthroughs

design, code, chapter of user’s guide,…

• presenter
• coordinator
• secretary
• maintenance oracle
• standards bearer
• user representative

<table>
<thead>
<tr>
<th>Discovery activity</th>
<th>Faults found per thousand lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements review</td>
<td>2.5</td>
</tr>
<tr>
<td>Design review</td>
<td>5.0</td>
</tr>
<tr>
<td>Code inspection</td>
<td>10.0</td>
</tr>
<tr>
<td>Integration test</td>
<td>3.0</td>
</tr>
<tr>
<td>Acceptance test</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Experiments

- 82% of faults discovered during design & code inspection (Fagan)
- 93% of all faults in a 6000-lines application were found by inspections (Ackerman, et al 1986)
- 85% of all faults removed by inspections from examining history of 10 million lines of code (Jones 1977)
- Inspections: finding code faults
- Prototyping: requirements problem

Proving code correct

- Formal proof techniques
- Symbolic execution
- Automated theorem proving
Black box / Closed box testing

- incorrect or missing functions
- interface errors
- performance error

Black box testing

- Equivalence partitioning
- Boundary value analysis
- Exhaustive testing
Equivalence partitioning

Specification: the program accepts four to eight inputs which are 5 digit integers greater than 10000.
Guidelines

If an input condition specifies

- A **range**: one valid and two invalid equivalence classes.
- A **specific value**: one valid and two invalid equivalence classes.
- A **member of a set**: one valid and one invalid equivalence classes.
- A **boolean**: one valid and one invalid class.

Boundary value analysis

<table>
<thead>
<tr>
<th></th>
<th>Less than 10000</th>
<th>Between 10000 and 99999</th>
<th>More than 99999</th>
</tr>
</thead>
</table>
Exhaustive testing

- **Definition**: testing with every member of the input value space.

- Input value space: the set of all possible input values to the program.

Glass box testing!
White box testing!
Open box testing!
Clear box testing!
Glass box testing

- logical decision
- loops
- internal data structure
- paths
- ...

Coverage!!

Statement Coverage

Begin
if ( y >= 0)
then y = 0;
abs = y;
end;

test case-1 (yes):
input: y = ?
expected result: ?
actual result: ?
Branch Coverage

Begin
if ( y >= 0)
    then y = 0;
abs = y;
end;

\[ \text{begin} \]
\[ \text{y >= 0} \]
\[ \text{no} \]
\[ \text{y = 0} \]
\[ \text{yes} \]
\[ \text{abs = y} \]

**test case-1 (yes):**
*input: y = 0*  
*expected result: 0*  
*actual result: 0*

**test case-2 (no):**
*input: y = ?*  
*expected result: ?*  
*actual result: ?*

Begin
if ( x < 10 && y > 20) {
    z = foo (x,y); else z = fie (x,y);
}
end;

\[ \text{begin} \]
\[ \text{x<10} \]
\[ \text{&&} \]
\[ \text{y>20} \]
\[ \text{yes} \]
\[ \text{z=fie (x,y)} \]
\[ \text{z=foo (x,y)} \]
\[ \text{no} \]

**test case-1 (yes):**
*input: x = ?, y = ?*  
*expected result: ?*  
*actual result: ?*

**test case-2 (no):**
*input: x = ?, y = ?*  
*expected result: ?*  
*actual result: ?*
**Condition - Branch Coverage**

Begin

```java
if ( x < 10 && y > 20) {
    z = foo (x,y); else z = fie (x,y);
}
end;
```

<table>
<thead>
<tr>
<th>x&lt;?</th>
<th>y&gt;?</th>
</tr>
</thead>
<tbody>
<tr>
<td>test-case-1:</td>
<td>t</td>
</tr>
<tr>
<td>test-case-2:</td>
<td>t</td>
</tr>
<tr>
<td>test-case-3:</td>
<td>f</td>
</tr>
<tr>
<td>test-case-4:</td>
<td>f</td>
</tr>
</tbody>
</table>

**Path Coverage**

```java
x <> 0
```

<table>
<thead>
<tr>
<th>x&lt;&gt;0</th>
<th>z = z-x</th>
<th>z = sin(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>no</td>
<td></td>
<td>no</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>z &gt; 10</th>
<th>z = 0</th>
<th>z = z / x</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td></td>
<td>no</td>
</tr>
</tbody>
</table>

- (n, y) x = ?, z = ?
- (y, n) x = ?, z = ?
- (n, n) x = ?, z = ?
- (n, y) x = ?, z = ?
- (y, n) x = ?, z = ?
- (y, y) x = ?, z = ?
Path with loops

January 2006  CUGS, SE, Mariam Kamkar, IDA, LiU

Path with loops

January 2006  CUGS, SE, Mariam Kamkar, IDA, LiU
Data Flow Testing

DEF(S) = \{x \mid \text{statement } S \text{ contains a definition of variable } x\}
USE(S) = \{x \mid \text{statement } S \text{ contains a use of variable } x\}
DEF-USE-Chain (du chain) = [x, S, S']

S1: \quad i = 1;

S2: \quad \text{while } (i \leq n)

Data Flow testing

s = 0;
i = 1;
s = 1;
while (i \leq n)
{ s += i; i ++}
print (s);
print (i);
print (n);

du: def-use
dk: def-kill
...
Program Slicing

```c
s = 0;
i = 1;
while (i <= n)
{
s += i;
i ++
}
print (s);
print (i);
print (n);
```

i = 1;
while (i <= n)
{
i ++
}
print (i);

Relative strengths of test strategies (B. Beizer 1990)

```
All paths
  ↓
All definition-use paths
  ↓
All uses
  ↓
All computational/ Some predicate uses
  ↓
All computational uses
  ↓
All definition
```
```
All predicate/ Some computational uses
  ↓
All predicate uses
  ↓
Branch
  ↓
Statement
```
Outline

- Function testing
- Performance testing
- Acceptance testing
- Installation testing

Objective: to ensure that the system does what the customer wants it to do.
Function testing
(testing one function at a time)

functional requirements

- have a high probability of detecting a fault
- use a test team independent of the designers and programmers
- know the expected actions and output
- test both valid and invalid input
- never modify the system just to make testing easier
- have stopping criteria

Cause-Effect
(test case generation from req.)

<table>
<thead>
<tr>
<th>Causes</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1: command is credit</td>
<td>E1: print “invalid command”</td>
</tr>
<tr>
<td>C2: command is debit</td>
<td>E2: print “invalid account number”</td>
</tr>
<tr>
<td>C3: account number is valid</td>
<td>E3: print “debit amount not valid”</td>
</tr>
<tr>
<td>C4: transaction amount is valid</td>
<td>E4: debit account print</td>
</tr>
<tr>
<td></td>
<td>E5: credit account print</td>
</tr>
</tbody>
</table>

January 2006 CUGS, SE, Mariam Kamkar, IDA, LiU
Performance testing
nonfunctional requirements

- Security
- Accuracy
- Speed
- Recovery
- Stress test
- Volume test
- ...

January 2006 CUGS, SE, Mariam Kamkar, IDA, LiU
Acceptance testing
customers, users need

- Benchmark test: a set of special test cases
- Pilot test: everyday working
  - Alpha test: at the developer’s site, controlled environment
  - Beta test: at one or more customer site.
- Parallel test: new system in parallel with previous one

Installation testing
users site

Acceptance test at developers site
→ installation test at users site, otherwise may not be needed!!
Test Planing

- Establishing test objectives
- Designing test cases
- Writing test cases
- Testing test cases
- Executing tests
- Evaluating test results

Automated Testing Tools

- Code Analysis tools
  - Static, Dynamic
- Test execution tools
  - Capture-and-Replay
  - Stubs & Drivers
- Test case generator
Termination Problem
How decide when to stop testing

- The main problem for managers!
- Termination takes place when
  - resources (time & budget) are over
  - found the seeded faults
  - some coverage is reached

Oracle
Scaffolding

What can be automated?

Test case generation
Termination
Empirical studies:

- difficult to locate and gain access to large systems.
- very time consuming, and therefore expensive, to collect and analyze the necessary data.
- difficult to find personnel with the appropriate skills to perform the empirical studies.
Questions

- How faults are distributed over the different files
  - Between release
  - Lifecycle stage
  - Severity
- How the size of modules affected their fault density
- Whether files that contained large numbers of faults during early stages of development, also had larger numbers of faults during later stages, and whether faultiness persisted from release to release.
- Whether newly written files were more fault-prone than ones that were written for earlier releases of the product.

Goal: identify characteristics of files that can be used as predictors of fault-proneness, thereby helping organizations determine how best to use their testing resources.

System description

- 13 successive releases
- Fault data was collected during:
  - Requirements
  - Design
  - Development
  - Unit testing
  - Integration testing
  - System testing
  - Beta release
  - Limited release:
    - Controlled release
    - General release
- Current version: 1,974 files, 500,000 lines of code, most of the system written in Java (1,412 files)
### Distribution of Faults

<table>
<thead>
<tr>
<th>Release</th>
<th>Files</th>
<th>KLOC</th>
<th>Early-Pre-Release</th>
<th>Late-Pre-Release</th>
<th>Post-Release</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Dev</td>
<td>Unit</td>
<td>Int</td>
<td>Sys</td>
</tr>
<tr>
<td>1</td>
<td>584</td>
<td>146</td>
<td>7</td>
<td>763</td>
<td>2</td>
<td>218</td>
</tr>
<tr>
<td>2</td>
<td>567</td>
<td>154</td>
<td>2</td>
<td>171</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>706</td>
<td>191</td>
<td>15</td>
<td>387</td>
<td>0</td>
<td>85</td>
</tr>
<tr>
<td>4</td>
<td>743</td>
<td>203</td>
<td>0</td>
<td>293</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>804</td>
<td>232</td>
<td>2</td>
<td>282</td>
<td>13</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>867</td>
<td>254</td>
<td>1</td>
<td>287</td>
<td>5</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td>993</td>
<td>292</td>
<td>14</td>
<td>156</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>1197</td>
<td>339</td>
<td>14</td>
<td>363</td>
<td>28</td>
<td>77</td>
</tr>
<tr>
<td>9</td>
<td>1321</td>
<td>377</td>
<td>50</td>
<td>298</td>
<td>28</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>1372</td>
<td>396</td>
<td>84</td>
<td>119</td>
<td>7</td>
<td>24</td>
</tr>
<tr>
<td>11</td>
<td>1607</td>
<td>427</td>
<td>17</td>
<td>158</td>
<td>17</td>
<td>71</td>
</tr>
<tr>
<td>12</td>
<td>1740</td>
<td>476</td>
<td>62</td>
<td>130</td>
<td>8</td>
<td>53</td>
</tr>
<tr>
<td>13</td>
<td>1772</td>
<td>471</td>
<td>35</td>
<td>52</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>Total</td>
<td>303</td>
<td>349</td>
<td>119</td>
<td>734</td>
<td>15</td>
<td>113</td>
</tr>
</tbody>
</table>

January 2006  
CUGS, SE, Mariam Kamkar, IDA, LiU

### Classification of 4,743 faults

- **Severity-1**: 78 faults (1.6%)  
- **Severity-2**: 687 faults (14.5%)  
- **Severity-3**: 3,847 faults (81%)  
- **Severity-4**: 131 faults (2.8%)
Fault concentration by release

- For each release, the faults were heavily concentrated in a relatively small number of files.
- For all of the releases, the percentage of the code mass contained in the files containing faults exceeded the percentage of the files.

### Overall Pareto Distribution by Release

<table>
<thead>
<tr>
<th>Release</th>
<th>10% Files Contain</th>
<th>100% Faults Contained In</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% Faults</td>
<td>% LOC</td>
</tr>
<tr>
<td>1</td>
<td>68</td>
<td>35</td>
</tr>
<tr>
<td>2</td>
<td>85</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>83</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>88</td>
<td>37</td>
</tr>
<tr>
<td>5</td>
<td>85</td>
<td>41</td>
</tr>
<tr>
<td>6</td>
<td>92</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td>97</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>94</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>96</td>
<td>30</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>100</td>
<td>-</td>
</tr>
</tbody>
</table>
Distribution of Faults by Lifecycle Stage

<table>
<thead>
<tr>
<th>Release</th>
<th>Early-Pre-Release</th>
<th>Late-Pre-Release</th>
<th>Post-Release</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% Files</td>
<td>% LOC</td>
<td>% Files</td>
</tr>
<tr>
<td>1</td>
<td>36</td>
<td>67</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>35</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>44</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>39</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>39</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>27</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

January 2006 CUGS, SE, Mariam Kamkar, IDA, LiU

Fault Concentration by Severity

- **Severity-1**: 78 faults (1.6%) -- in 3% files in release-1 to 0% in release 12
- **Severity-4**: 131 faults (2.8%) -- in 4% files in release-1 to 0.3% in release 12
- **Severity-2**: 687 faults (14.5%) -- in small percentage of files
- **Severity-3**: 3,847 faults (81%)

<table>
<thead>
<tr>
<th>Release</th>
<th>Severity-3 (80% faults)</th>
<th>100% Faults Contained In</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% Files</td>
<td>% LOC</td>
</tr>
<tr>
<td>1</td>
<td>36</td>
<td>68</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>35</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>22</td>
</tr>
</tbody>
</table>

January 2006 CUGS, SE, Mariam Kamkar, IDA, LiU
Effects of Module size on Fault-proneness

- Standard wisdom: large modules – much more fault-prone
- Basili (1984), Moller (1993): contrary, opposite was true.
- In the study here
  - Fault densities between 10 and 75 faults/KLOC for smallest files (under 100 lines)
  - 2-3 faults/KLOC for larger than 1000 lines
- Hatton (1997): fault density was high for smallest components, decreased to minimum for medium-size components, and then started increasing again as components size grew.
- Fenton (2000): don’t agree with any of the above, found no trend at all
- Various other factors: new file, amount of changed code, amount of testing performed, experience of the programmer…

=> needs more investigation

Persistence of High-Fault Files

<table>
<thead>
<tr>
<th>Release</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rel (n-1)</td>
<td>-</td>
<td>27</td>
<td>54</td>
<td>21</td>
<td>45</td>
<td>42</td>
<td>52</td>
<td>34</td>
<td>21</td>
<td>17</td>
<td>39</td>
<td>24</td>
</tr>
<tr>
<td>Rel (n+1)</td>
<td>63</td>
<td>46</td>
<td>27</td>
<td>30</td>
<td>56</td>
<td>34</td>
<td>34</td>
<td>27</td>
<td>22</td>
<td>36</td>
<td>22</td>
<td>-</td>
</tr>
</tbody>
</table>

- High-Fault Files: top 20% of files ordered by decreasing number of faults, plus all other files that have as many faults as the least number among the top 20%
- Rel (n-1): shows the percent of high-fault files in Release (n-1) that remained high-fault files in Release n.
- Rel (n+1): shows the percent of high-fault files in Release (n+1) that had been high-fault files in Release n.

=> file containing high numbers of faults in one release, remain high-fault files in later release
Comparison of Faults For Old and New Files

<table>
<thead>
<tr>
<th>Release</th>
<th>% Faulty Files</th>
<th>Fault /KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OLD</td>
<td>NEW</td>
</tr>
<tr>
<td>2</td>
<td>15.4</td>
<td>16.3</td>
</tr>
<tr>
<td>3</td>
<td>18.5</td>
<td>24.8</td>
</tr>
<tr>
<td>4</td>
<td>13.8</td>
<td>39.1</td>
</tr>
<tr>
<td>5</td>
<td>13.8</td>
<td>31.0</td>
</tr>
<tr>
<td>6</td>
<td>11.0</td>
<td>36.8</td>
</tr>
<tr>
<td>7</td>
<td>10.2</td>
<td>13.3</td>
</tr>
<tr>
<td>8</td>
<td>12.2</td>
<td>12.8</td>
</tr>
<tr>
<td>9</td>
<td>8.9</td>
<td>36.3</td>
</tr>
<tr>
<td>10</td>
<td>7.6</td>
<td>20.7</td>
</tr>
<tr>
<td>11</td>
<td>6.9</td>
<td>7.9</td>
</tr>
<tr>
<td>12</td>
<td>5.8</td>
<td>14.4</td>
</tr>
</tbody>
</table>

- Percentage of faulty new files is larger than the percentage of faulty pre-existing files
- The fault density is higher for new files than for pre-existing ones

=> More resources for testing new files

Conclusions

- Fault concentrate in small numbers of files and small percentages of the code mass.
- For each release, the early-pre-release faults accounted for a clearly majority of the faults.
- Percentage of lines of code contained in files that contained faults exceeded the percentage of files that contained faults.
- Across successive releases, high-fault files of one release tend to remain high-fault in later release.
Real life examples

- First U.S. space mission to Venus failed. (reason: missing comma in a Fortran do loop)
- December 1995: AA, Boeing 575, mountain crash in Colombia, 159 killed. Incorrect one-letter computer command (Cali, Bogota 132 miles in opposite direction, have same coordinate code)
- June 1996: Ariane-5 space rocket, self-destruction, $500 million. (reason: reuse of software from Ariane-4 without recommended testing).

Real life examples

- Australia: Man jailed because of computer glitch. He was jailed for traffic fine although he had actually paid it 5 years ago.
- Dallas Prisoner released due to program design flaw: He was temporary transferred from one prison to another (witness). Computer gave him “temporary assignment”.
Goals of software testing: Historical Evolution

And …

Testing can show the presence, but never the absence of errors in software.

E. Dijkstra, 1969