Softwaretechnik
Lecture 08: Testing and Debugging — Overview

Peter Thiemann

University of Freiburg, Germany

SS 2012
Literature

Essential Reading

Essential Reading

Essential Reading


Further Reading

Cost of Software Errors

$ 60 billion
Cost of Software Errors

$60\ \text{billion}$

yearly cost of software errors for US economy [NIST 2002]
Cost of Software Errors

$ 180 billion
Cost of Software Errors

$ 180 billion

total sales of software in 2000
Cost of Software Errors

$180 billion

total sales of software in 2000

697,000 software engineers & 585,000 computer programmers
Cost of Software Errors

estimated 50%
Cost of Software Errors

estimated

50%

of each software project spent on testing
Cost of Software Errors

estimated

50%

of each software project spent on testing

(spans from 30% to 80%)
Cost of Software Errors

very rough approximation

money spent on testing \approx \text{cost of remaining errors}
Cost of Software Errors

very rough approximation

money spent on testing + remaining errors = cost of size of software industry
Cost of Software Errors

very rough approximation

\[ \text{money spent on testing} + \text{remaining errors} = 66\% \text{ of size of software industry} \]
A Quiz About Testing

A simple program

Input
Read three integer values from the command line. The three values represent the lengths of the sides of a triangle.

Output
Tells whether the triangle is

- **Scalene**: no two sides are equal
- **Isosceles**: exactly two sides are equal
- **Equilateral**: all sides are equal
A simple program

**Input**
Read three integer values from the command line. The three values represent the lengths of the sides of a triangle.

**Output**
Tells whether the triangle is
- **Scalene**: no two sides are equal
- **Isosceles**: exactly two sides are equal
- **Equilateral**: all sides are equal

Task: Create a Set of Test Cases for this Program
Q 1: (4,1,2) a invalid triangle
Q 1: (4,1,2) a invalid triangle

Why not a valid triangle?
Q 1: (4,1,2) a invalid triangle

Why not a valid triangle? \((a,b,c)\) with \(a > b + c\)
Q 1: (4,1,2) a **invalid** triangle

Why not a valid triangle? \((a,b,c)\) with \(a > b + c\)

Define valid triangles: \(a \leq b + c\)
Q 2: some permutations of previous (1,2,4), (2,1,4)
Q 2: some permutations of previous (1,2,4), (2,1,4)

Fulfill above definition, but are still invalid.
Q 2: some permutations of previous (1,2,4), (2,1,4)

Fulfill above definition, but are still invalid.

Patch definition of valid triangles:

\[ a \leq b + c \text{ and } b \leq a + c \text{ and } c \leq a + b \]
Q 3: (4,2,2) a invalid triangle with equal sum
Q 3: (4, 2, 2) a invalid triangle with equal sum

Fulfills above definition, but is invalid (depending on what we want!).
Q 3: (4,2,2) a invalid triangle with equal sum

Fulfills above definition, but is invalid (depending on what we want!).

Patch definition of valid triangles:

\[ a < b + c \text{ and } b < a + c \text{ and } c < a + b \]
Q 4: some permutations of previous (2,2,4), (2,4,2)
Q 5: (3,4,5) a **valid scalene** triangle
Q 6: (3,3,3) an equilateral triangle
Q 7: \((3, 4, 3)\) valid isosceles triangle.
Q 8: all permutations of valid isosceles triangle:

(3,4,3), (3,3,4), (4,3,3)
Q 9: one side with zero value (0,4,3)
Q 10: one side with negative value (-1,4,3)
Q 11: all sides zero (0,0,0)
Q 12: at least one value is non-integer (1,3,2.5)
Q 13: wrong number of arguments (2,4) or (1,2,3,3)
Q 14 (the most important one):

Did you specify the expected output in each case?
About the Quiz

- Q 1–13 correspond to failures that have actually occurred in implementations of the program
- How many questions did you answer?
About the Quiz

- Q 1–13 correspond to failures that have actually occurred in implementations of the program
- How many questions did you answer?
- Highly qualified, experienced programmers score 7.8 on average
First Conclusions

- Finding good and sufficiently many test cases is difficult
- Even a good set of test cases cannot exclude more failures
- A specification is required to identify failures
First Conclusions

- Finding good and sufficiently many test cases is difficult
- Even a good set of test cases cannot exclude more failures
- A specification is required to identify failures

The discipline of Testing is all about Test Cases
First Conclusions

- Finding good and sufficiently many test cases is difficult
- Even a good set of test cases cannot exclude more failures
- A specification is required to identify failures

The discipline of Testing is all about Test Cases

Remark: At Ericsson: 35% of code is test cases!
What is a Bug?

Bug-Related Terminology

1. Defect (aka bug, fault) introduced to code by programmer (not always programmer’s fault, if, e.g., requirements changed)

2. Defect may cause infection of program state during execution (not all defects cause infection)

3. Infected state propagates during execution (infected parts of states may be overwritten or corrected)

4. Infection may cause a failure: an externally observable error (including, e.g., non-termination)

Harvard University, Mark II Aiken Relay Calculator
What is a Bug? Basic Terminology

Bug-Related Terminology

1. **Defect** (aka bug, fault) introduced to code by programmer (not always programmer’s fault, if, e.g., requirements changed)
What is a Bug? Basic Terminology

Bug-Related Terminology

1. **Defect** (aka bug, fault) introduced to code by programmer (not always programmer’s fault, if, e.g., requirements changed)
2. Defect may cause **infection** of program state during execution (not all defects cause infection)
What is a Bug? Basic Terminology

Bug-Related Terminology

1. Defect (aka bug, fault) introduced to code by programmer (not always programmer’s fault, if, e.g., requirements changed)

2. Defect may cause infection of program state during execution (not all defects cause infection)

3. Infected state propagates during execution (infected parts of states may be overwritten or corrected)
## What is a Bug? Basic Terminology

### Bug-Related Terminology

1. **Defect** (aka bug, fault) introduced to code by programmer (not always programmer’s fault, if, e.g., requirements changed)

2. Defect may cause **infection** of program state during execution (not all defects cause infection)

3. Infected state **propagates** during execution (infected parts of states may be overwritten or corrected)

4. Infection may cause a **failure**: an externally observable error (including, e.g., non-termination)

---

**Defect — Infection — Propagation — Failure**
Failure and Specification

Some failures are obvious

- obviously wrong output/behaviour
- non-termination
- crash
- freeze

...but most are not!
Some failures are obvious

- obviously wrong output/behaviour
- non-termination
- crash
- freeze

... but most are not!

In general, what constitutes a failure, is defined by
Failure and Specification

Some failures are obvious

- obviously wrong output/behaviour
- non-termination
- crash
- freeze

...but most are not!

In general, what constitutes a failure, is defined by a specification!
Failure and Specification

Some failures are obvious

- obviously wrong output/behaviour
- non-termination
- crash
- freeze

... but most are not!

In general, what constitutes a failure, is defined by a specification!

Correctness is a relative notion

— B. Meyer, 1997
### Failure and Specification

Some failures are obvious

- obviously wrong output/behaviour
- non-termination
- crash
- freeze

... but most are not!

In general, what constitutes a failure, is defined by a specification!

Correctness is a relative notion

— B. Meyer, 1997

Every program is correct with respect to SOME specification

— myself, today
Economist: The cows in Scotland are brown.

Logician: No, there are cows in Scotland of which at least one is brown!

Computer Scientist: No, there is at least one cow in Scotland, which is brown on one side!
Economist: The cows in Scotland are brown
Economist: The cows in Scotland are brown

Logician: No, there are cows in Scotland of which one at least is brown!
Economist:
The cows in Scotland are brown

Logician:
No, there are cows in Scotland of which one at least is brown!

Computer Scientist:
No, there is at least one cow in Scotland, which is brown on one side!!
Example

A Sorting Program:

```java
public static Integer[] sort(Integer[] a) { ... }
```
Example

A Sorting Program:

```java
public static Integer[] sort(Integer[] a) {
    ...
}
```

Testing `sort()`:

```java
sort({3, 2, 5}) == {2, 3, 5}

sort({}) == {}

sort({17}) == {17}
```
Example

A Sorting Program:

```java
public static Integer[] sort(Integer[] a) {
    // ...
}
```

Testing `sort()`:

- `sort({3, 2, 5}) == {2, 3, 5}` ✔️
Example

A Sorting Program:

```java
public static Integer[] sort(Integer[] a) {
    // ...
}
```

Testing `sort()`:

- `sort({3, 2, 5}) == {2, 3, 5}` ✓
- `sort({}) == {}` ✓
Example

A Sorting Program:

```java
public static Integer[] sort(Integer[] a) { ... }
```

Testing `sort()`:

- `sort({3, 2, 5}) == {2, 3, 5}` ✓
- `sort({}) == {}` ✓
- `sort({17}) == {17}` ✓
Example

A Sorting Program:

```java
public static Integer[] sort(Integer[] a) {
    ...
}
```

Testing `sort()`:

- `sort({3, 2, 5}) == {2, 3, 5}`
- `sort({}) == {}`
- `sort({17}) == {17}`
Example

A Sorting Program:

```java
public static Integer[] sort(Integer[] a) {
    ...}
```

Testing `sort()`:

- `sort({3, 2, 5}) == {2, 3, 5}`
- `sort({}) == {}`
- `sort({17}) == {17}`

Specification?
Example
A Sorting Program:

```java
public static Integer[] sort(Integer[] a) { ... }
```

Testing `sort()`:

- `sort({3, 2, 5}) == {2, 3, 5}` ✓
- `sort({}) == {}` ✓
- `sort({17}) == {17}` ✓

Specification

*Requires*: `a` is an array of integers

*Ensures*: returns the sorted argument array `a`
Example

```java
public static Integer[] sort(Integer[] a) {
    ...
}
```

Specification

*Requires:* \( a \) is an array of integers

*Ensures:* returns the sorted argument array \( a \)

*Is this a good specification?*
Example

```java
public static Integer[] sort(Integer[] a) {
    ...
}
```

Specification

- **Requires**: a is an array of integers
- **Ensures**: returns the sorted argument array a

Is this a good specification?

```java
sort({2, 1, 2}) == {1, 2, 2, 17}  
```
Example Cont’d

Example

```java
public static Integer[] sort(Integer[] a) {
    ...
}
```

Specification

*Requires:* a is an array of integers

*Ensures:* returns a sorted array with only elements from a
Example Cont’d

Example

```java
public static Integer[] sort(Integer[] a) {
    ...
}
```

Specification

**Requires:** a is an array of integers

**Ensures:** returns a sorted array with only elements from a

```
sort({2, 1, 2}) == {1, 1, 2} ✗
```
Example Cont’d

Example

```java
public static Integer[] sort(Integer[] a) {
    ...
}
```

Specification

*Requires:* a is an array of integers

*Ensures:* returns a permutation of a that is sorted
Example Cont’d

Example

```java
public static Integer[] sort(Integer[] a) {
    // ...
}
```

Specification

**Requires:** a is an array of integers

**Ensures:** returns a permutation of a that is sorted

sort(null) throws NullPointerException ✗
Example Cont’d

Example

```java
public static Integer[] sort(Integer[] a) {
    // ...
}
```

Specification

- **Requires:** `a` is a *non-null* array of integers
- **Ensures:** returns a permutation of `a` that is sorted
Example Cont’d

Example

```java
public static Integer[] sort(Integer[] a) {
    ...
}
```

Specification

- **Requires:** a is a *non-null* array of integers
- **Ensures:** returns the *unchanged* reference a containing a permutation of the *old* contents of a that is sorted
The Contract Metaphor

**Contract** is preferred specification metaphor for procedural and OO PLs

first propagated by B. Meyer, *Computer* 25(10)40–51, 1992

Same Principles as Legal Contract between a Client and Supplier

<table>
<thead>
<tr>
<th>Supplier (aka Implementer)</th>
<th>in <strong>JAVA</strong>, a class or method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>Mostly a caller object, or human user for <strong>main</strong>()</td>
</tr>
<tr>
<td>Contract</td>
<td>One or more pairs of <strong>ensures</strong>/<strong>requires</strong> clauses defining mutual benefits and obligations of client and implementer</td>
</tr>
</tbody>
</table>
The Meaning of a Contract

Specification (of method \texttt{C::m()})

\textbf{Requires:} Precondition
\textbf{Ensures:} Postcondition

“If a caller of \texttt{C::m()} fulfills the \textit{required} Precondition, then the class \texttt{C} ensures that the Postcondition holds after \texttt{m()} finishes.”
The Meaning of a Contract

Specification (of method \( C::m() \))

**Requires:** Precondition

**Ensures:** Postcondition

“If a caller of \( C::m() \) fulfills the required Precondition, then the class \( C \) ensures that the Postcondition holds after \( m() \) finishes.”

Often the following **wrong** interpretations of contracts are seen:

**Wrong!**

“Any caller of \( C::m() \) must fulfill the required Precondition.”

**Wrong!**

“Whenever the required Precondition holds, then \( C::m() \) is executed.”
Definition: **failure**

A method **fails** if it is called in a state fulfilling the required precondition of its contract and does not terminate in a state fulfilling the postcondition.

Non-termination, abnormal termination considered as failures here
**Definition: partial correctness**
A method is partially correct if whenever it is started in a state fulfilling the required precondition and it terminates, then its final state fulfills the postcondition.

This amounts to proving **Absence of Failures**!
Notions of Correctness

Definition: **partial correctness**
A method is partially correct if whenever it is started in a state fulfilling the required precondition and it terminates, then its final state fulfills the postcondition.

This amounts to proving **Absence of Failures**!

Definition: **total correctness**
A method is totally correct if whenever it is started in a state fulfilling the required precondition, then it terminates and its final state fulfills the postcondition.

**Total correctness implies termination!**
Invariant

Objects with non-trivial state

often maintain a class invariant.

Example: a class for dates

public class Date {
    public int day;
    public int month;
    public int year;
}

Invariant:

1 <= day <= 31 \ 1 <= month <= 12 \n(month in \{4, 6, 9, 11\} => day <= 30) \n(month == 2 => day <= 29) \n(month == 2 \ (year % 4 != 0 \ (year % 100 == 0 \ (year % 400 != 0
    => day <= 28)
All public methods of a class must preserve the class invariant.
Class invariants can be incorporated into pre- and postconditions.

**Specification (of a method)**

- **Requires**: Precondition and Invariant
- **Ensures**: Postcondition and Invariant
Invariant II

- All public methods of a class must preserve the class invariant.
- Class invariants can be incorporated into pre- and postconditions.

Specification (of a method)

Requires: Precondition and Invariant
Ensures: Postcondition and Invariant

Specification (of a constructor)

Requires: Precondition
Ensures: Invariant
Further Elements of a Contract

Type signature (minimal contract)

Exceptions raised

Temporal properties

- the capacity of the table does not change over time
- a set that is only supposed to grow
Testing vs. Verification

**TESTING**

Goal: find evidence for presence of failures

Testing: execute a program with the intent of detecting failure

Testing cannot guarantee correctness, i.e., absence of failures

Related techniques: code reviews, program inspections
Testing vs. Verification

**TESTING**
Goal: find evidence for presence of failures

Testing: execute a program with the intent of detecting failure

Testing cannot guarantee correctness, i.e., absence of failures

Related techniques: code reviews, program inspections

**VERIFICATION**
Goal: find evidence for absence of failures

Verification guarantees correctness

Related techniques: code generation, program synthesis (from spec)
Both, testing and verification attempts exhibit new failures.

Debugging is a systematic process that finds and eliminates the defect that led to an observed failure.

Programs without known failures may still contain defects:

- If they have not been verified
- If they have been verified, but the failure is not covered by the specification
Where Formalization Comes In

Testing is very expensive, even with tool support

30–80% of development time goes into testing
Where Formalization Comes In

- Testing is very expensive, even with tool support
- 30–80% of development time goes into testing

Diagram:
- Test cases
- Code under test
- Code checking success
Where Formalization Comes In

Testing is very expensive, even with tool support

30–80% of development time goes into testing

- Test cases
- Code under test
- Test oracle
- Test case generator
- Code checking success
Where Formalization Comes In

Testing is very expensive, even with tool support

30–80% of development time goes into testing

Test cases

Code under test

Code checking success

Test case generator

Formal specification

Test oracle
Formal Verification of Program Correctness

Java Code

Formal specification
Formal Verification of Program Correctness

Java Code \[\rightarrow\] Formal specification correct?
Formal Verification of Program Correctness

Java Code → Formal specification

correct?

Program Verification System
Formal Verification of Program Correctness

Java Code \rightarrow \text{Formal specification}

Program Verification System
Formal Verification of Program Correctness

Java Code $\rightarrow$ Formal specification

Program Verification System

- synchronised java.lang.StringBuffer append(char c)
  - ca. 15,000 proof steps
  - ca. 200 case distinctions
  - Two human interactions, ca. 1 minute computing time
Tool Support is Essential

Some Reasons for Using Tools

- Automate repetitive tasks
- Avoid typos, etc.
- Cope with large programs
Tool Support is Essential

Some Reasons for Using Tools

- Automate repetitive tasks
- Avoid typos, etc.
- Cope with large programs

Tools Used

- Automated running of tests: JUNIT
- Debugging: ECLIPSE debugger