

# Softwaretechnik

## Lecture 12: Testing and Debugging — Debugging

Peter Thiemann

University of Freiburg, Germany

21.06.2012

# Today's Topic

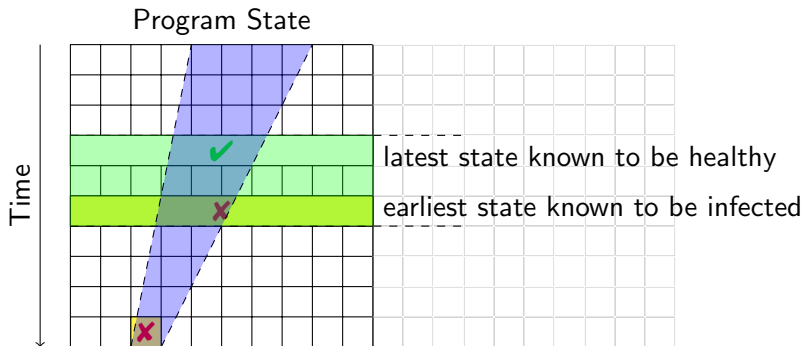
— Last Lecture —

- ✓ Bug tracking
- ✓ Program control — Design for Debugging
- ✓ Input simplification

— This Lecture —

- ▶ Execution observation
  - ▶ With logging
  - ▶ Using debuggers
- ▶ Tracking causes and effects

# The Main Steps in Systematic Debugging



- ▶ Separate healthy from infected states
- ▶ Separate relevant from irrelevant states

# Central Problem

How can we observe a program run?

## Challenges/Obstacles

- ▶ Observation of intermediate state not part of functionality
- ▶ Observation can change the behavior
- ▶ Narrowing down to relevant time/state sections

# The Naive Approach: Print Logging

## Println Debugging

Manually add print statements at code locations to be observed

```
System.out.println("size_□=□" + size);
```

- ✓ Simple and easy
- ✓ Can use any output channel
- ✓ No tools or infrastructure needed, works on any platform
- ✗ Code cluttering
- ✗ Output cluttering (at least need to use debug channel)
- ✗ Performance penalty, possibly changed behavior (timing, ...)
- ✗ Buffered output lost on crash
- ✗ Source code required, recompilation necessary

# Logging Frameworks

## Example (Logging Framework for JAVA)

```
java.util.logging
```

### Main principles of Java logging

- ▶ Each class can have its own `Logger` object
- ▶ Each logger is associated with a level and a handler
- ▶ Levels: `FINEST < FINER < FINE < CONFIG < INFO < WARNING < SEVERE`
- ▶ Handlers: `j.u.l.ConsoleHandler`, `j.u.l.FileHandler`
- ▶ **Example:** log message with `myLogger` and level `INFO`:  
`myLogger.info(Object message);`
- ▶ Logging can be controlled by program or properties file: which logger, level, filter, formatting, handler, etc.
- ▶ No recompilation necessary for reconfiguration

# Logging Demo

# Evaluation of Logging Frameworks

- ✓ Output cluttering can be mastered
- ✓ Small performance overhead
- ✓ Exceptions are loggable
- ✓ Log complete up to crash
- ✓ Instrumented source code reconfigurable w/o recompilation
- ✗ Code cluttering — don't try to log everything!

Code cluttering avoidable with aspects, but also with **Debuggers**



# What is a Debugger?

## Basic Functionality of a Debugger

**Execution Control** Stop execution on specified conditions:  
**breakpoints**

**Interpretation** **Step-wise** execution of code

**State Inspection** **Observe** value of variables and stack

**State Change** **Change** state of stopped program

Historical term **Debugger** is misnomer as there are many debugging tools

- ▶ Traditional debuggers (gdb for C) based on command line I/F
- ▶ We use the built-in GUI-based debugger of the ECLIPSE framework
  - ▶ Feel free to experiment with other debuggers!

## Running Example

```
1 public static int search( int[] array,
2                           int target ) {
3
4     int low = 0;
5     int high = array.length;
6     int mid;
7     while ( low <= high ) {
8         mid = (low + high)/2;
9         if ( target < array[ mid ] ) {
10             high = mid - 1;
11         } else if ( target > array[ mid ] ) {
12             low = mid + 1;
13         } else {
14             return mid;
15         }
16     }
17     return -1;
18 }
```

# Eclipse Debugger

- ▶ Open directory BinSearch, create project Search
- ▶ Create/show run configuration testBin1
- ▶ Run testBin1
- ▶ Open Debugging view of project Search

## Running a few test cases ...

`search( {1,2}, 1 ) == 0` ✓

`search( {1,2}, 2 ) == 1` ✓

`search( {1,2}, 4 )` throws

`ArrayIndexOutOfBoundsException: 3` ✗

Example taken from a published JAVA text book :- (

# Halting Program Execution

## Breakpoint

A program location that, when it is reached, halts execution

## Example (Setting Breakpoint)

In `search()` at loop, right-click, toggle breakpoint

## Some remarks on breakpoints

- ▶ Set breakpoint at **last statement where state is known to be healthy**
- ▶ Formulate healthiness as an explicit **hypothesis**
- ▶ In `ECLIPSE`, not all lines can be breakpoints, because these are actually inserted into bytecode
- ▶ Remove breakpoints when no longer needed

# Resuming Program Execution

## Example (Execution Control Commands)

- ▶ **Start** debugging of run configuration testBin1
- ▶ **Resume** halts when breakpoint is reached in next loop execution
- ▶ **Disable** breakpoint **for this session**
- ▶ **Resume** executes now until end
- ▶ **Remove** from debug log (Remove All Terminated)
- ▶ **Enable** breakpoint again in Breakpoints window
- ▶ **Close** debugging perspective

# Step-Wise Execution of Programs

## Step-Wise Execution Commands

**Step Into** Execute next statement, then halt

**Step Over** Consider method call as **one** statement

## Some remarks on step-wise execution

- ▶ Usually `JAVA` library methods stepped over
  - ▶ They should not contain defects
  - ▶ You probably don't have the source code
- ▶ To step over bigger chunks, change breakpoints, then resume

# Inspecting the Program State

## Inspection of state while program is halted

- ▶ Variables window
  - ▶ Unfold reference types
  - ▶ Pretty-printed in lower half of window
- ▶ Tooltips for variables in focus in editor window
- ▶ Recently changed variables are highlighted

## Example (Tracking `search()`)

- ▶ Start debugging at beginning of loop (`testBin2`)
- ▶ Step through one execution of loop body
- ▶ After first execution of loop body `low==high==2`
- ▶ Therefore, `mid==2`, but `array[2]` doesn't exist!
- ▶ If `target` is greater than all array elements, eventually `low==mid==array.length`



# Changing the Program State

## Hypothesis for Correct Value

Variable `high` should have value `array.length-1`

Changing state while program is halted

- ▶ Right-click on identifier in Variables window, **Change Value**

## Example (Fixing the defect in the current run)

At start of second round of loop, set `high` to correct value 1

Resuming execution now yields correct result

# Watching States with Debuggers

## Halting Execution upon Specific Conditions

Use Boolean **Watch** expression in **conditional breakpoint**

### Example (Halting just before exception is thrown)

- ▶ From test run: argument `mid` of array is 2 at this point
- ▶ Create breakpoint at code position where evaluation takes place
- ▶ Add watch expression `mid==2` to breakpoint properties
- ▶ Disable breakpoint at start of loop
- ▶ Execution halts exactly when `mid==2` becomes true

### Hints on watch expressions

- ▶ Make sure scope of variables in watch expressions is big enough

# Evaluation of Debuggers

- ✓ Code cluttering completely avoided
- ✓ Prudent usage of breakpoints/watches reduces states to be inspected
- ✓ Full control over all execution aspects
- ✗ Debuggers are **interactive** tools, re-use difficult
- ✗ Performance can degrade, disable unused watches
- ✗ Inspection of reference types (lists, etc.) is tedious

## Important Lessons

- ▶ Both, logging **and** debuggers are necessary and **complementary**
- ▶ Need **visualization** tools to render complex data structures
- ▶ Minimal/small input, localisation of unit is important

# Tracking Causes and Effects

Determine defect that is **origin** of failure

## Fundamental problem

Program executes **forward**, but need to reason **backwards** from failure

## Example

In `search()` the failure was caused by wrong value `mid`, but the real culprit was `high`

# Effects of Statements

Fundamental ways how statements may affect each other

**Write** Change the program state  
Assign a new value to a variable read by another statement

**Control** Change the program counter  
Determine which statement is executed next

Statements with **Control** Effect (in JAVA)

- ▶ **Conditionals, switches**
- ▶ **Loops**: determine whether their body is executed
- ▶ **Dynamic method calls**: implicit case distinction on implementations
- ▶ **Abrupt termination** statements: `break`, `return`
- ▶ **Exceptions**: potentially at each object or array access!

# Statement Dependencies

## Definition (Data Dependency)

Statement B is **data dependent** on statement A iff

1. A writes to a variable  $v$  that is read by B **and**
2. There is at least one execution path between A and B in which  $v$  is not written to

“The outcome of A can directly influence a variable read in B”

# Statement Dependencies

## Definition (Control Dependency)

Statement B is **control dependent** on statement A iff

- ▶ There is an execution path from A to B such that:  
For all statements  $S \neq A$  on the path, all execution paths from S to the method exit pass through B  
**and**
- ▶ There is an execution path from A to the method exit that does **not** pass through B

“The outcome of A can influence whether B is executed”

## Example

```
1 int low = 0;
2 int high = array.length;
3 int mid;
4 while ( low <= high ) {
5     mid = (low + high)/2;
6     if ( target < array[ mid ] ) {
7         high = mid - 1;
8     } else if ( target > array[ mid ] ) {
9         low = mid + 1;
10    } else {
11        return mid;
12    }
13 }
14 return -1;
```

`mid` is control-dependent on the `while` statement



# Computing Backward Dependencies

## Definition (Backward Dependency)

Statement B is **backward dependent** on statement A iff

There is a sequence of statements  $A = A_1, A_2, \dots, A_n = B$  such that:

1. for all  $i$ ,  $A_{i+1}$  is either control dependent or data dependent on  $A_i$
2. there is at least one  $i$  with  $A_{i+1}$  being data dependent on  $A_i$

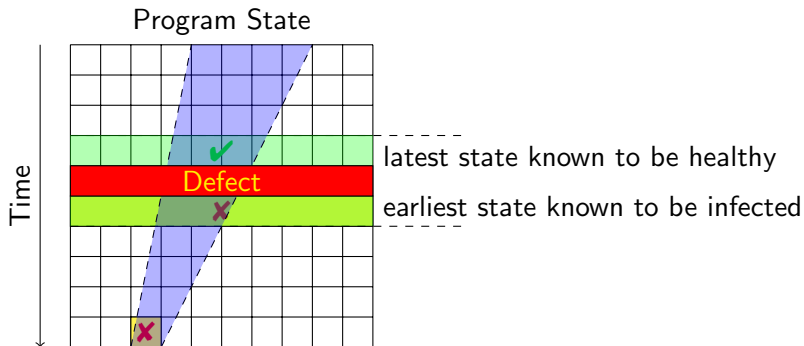
“The outcome of A can influence the program state in B”

## Example

```
1 int low = 0;
2 int high = array.length;
3 int mid;
4 while ( low <= high ) {
5     mid = (low + high)/2;
6     if ( target < array[ mid ] ) {
7         high = mid - 1;
8     } else if ( target > array[ mid ] ) {
9         low = mid + 1;
10    } else {
11        return mid;
12    }
13 }
14 return -1;
```

Backward-dependent statements for **repeated** execution of loop body

# Systematic Location of Defects



- ▶ Separate healthy from infected states
- ▶ Separate relevant from irrelevant states
- ▶ Compute backward-dependent statements from infected locations

# Tracking Down Infections

## Algorithm for systematic location of defects

Let  $\mathcal{I}$  be a set of infected locations (variable+program counter)

1. Initialize  $\mathcal{I} := \{\text{infected location reported by failure}\}$
2. Choose a current, infected location  $L \in \mathcal{I}$
3. Compute statements  $\mathcal{S}$  that may contain origin of defect:  
one level of backward dependency from  $L$  in execution path
4. Inspect variables  $v_1, \dots, v_n$  written to in  $\mathcal{S}$ :  
let  $\mathcal{M} \subseteq \{v_1, \dots, v_n\}$  be the infected variables
5. If  $\mathcal{M} \neq \emptyset$  (there is an infected variable):
  - 5.1 Let  $\mathcal{I} := (\mathcal{I} \setminus \{L\}) \cup \mathcal{M} \times \mathcal{S}$   
(replace  $L$  with new candidates in  $\mathcal{M}$ )
  - 5.2 Goto 2.
6.  $L$  depends only on healthy locations, it must be the infection site!

## Example

```
1 int low = 0;
2 int high = array.length - 1;
3 int mid;
4 while ( low <= high ) {
5     mid = (low + high)/2;
6     if ( target < array[ mid ] ) {
7         high = mid - 1;
8     } else if ( target > array[ mid ] ) {
9         low = mid + 1;
10    } else {
11        return mid;
12    }
13 }
14 return -1;
```

Fixed defect

## After Fixing the Defect

- ▶ Failures that exhibited a defect become **new** test cases after the fix
  - ▶ used for **regression testing**
- ▶ Use **existing** unit test cases to
  - ▶ test a suspected method in isolation
  - ▶ make sure that your bug fix did not introduce new bugs
  - ▶ exclude wrong hypotheses about the defect

# Open Questions

1. How is evaluation of test runs related to specification?  
So far: wrote oracle program or evaluated interactively  
How to check automatically whether test outcome conforms to spec?
2. It is tedious to write test cases by hand  
Easy to forget cases  
JAVA: aliasing, run-time exceptions
3. When does a program have no more bugs?  
How to prove correctness without executing  $\infty$  many paths?

## Literature for this Lecture

### Essential

[Zeller](#) Why Programs Fail: A Guide to Systematic Debugging, Morgan Kaufmann, 2005  
Chapters 7, 8, 9