Software Engineering

Lecture 16: Testing and Debugging — Debugging

Peter Thiemann

University of Freiburg, Germany

SS 2014

Today's Topic

— This Lecture —

- Execution observation
- Tracking causes and effects

Central Problem

How can we observe a program run?

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);
```

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

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

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

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

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 and automation difficult
- ✘ Performance can degrade, disable unused watches
- ✘ Inspection of reference types (lists, etc.) is tedious

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 and automation difficult
- ✘ Performance can degrade, disable unused watches
- ✘ Inspection of reference types (lists, etc.) is tedious

Conclusions

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

Running Example

```
public static int search( int[] array,
                          int target ) {

  int low = 0;
  int high = array.length;
  int mid;
  while ( low <= high ) {
    mid = (low + high)/2;
    if ( target < array[ mid ] ) {
        high = mid - 1;
    } else if ( target > array[ mid ] ) {
        low = mid + 1;
    } else {
        return mid;
    }
  }
  return -1;
}
```

Testing

Running a few test cases . . .

`search( {1,2}, 1 ) == 0` ✔

Testing

Running a few test cases ...

```
search( {1,2}, 1 ) == 0 ✔
search( {1,2}, 2 ) == 1 ✔
```

Testing

Running a few test cases …

`search( {1,2}, 1 ) == 0` ✔

`search( {1,2}, 2 ) == 1` ✔

`search( {1,2}, 4 )` throws
`ArrayIndexOutOfBoundsException: 3` ✘

Testing

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 :-(

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

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 Write Effect (in JAVA)

- Assignments
- I/O, because it affects buffer content
- new(), because object initialisation writes to fields

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 (Control Flow Graph (CFG))

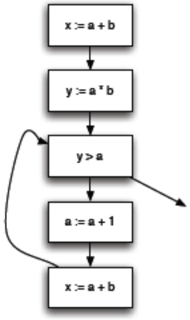
The control flow graph of a method M is a directed graph (V, E) where

- the set V of vertices is the set of program points of M. It contains
  - a designated entry vertex
  - a designated exit vertex corresponding to the `return` statement
  - a vertex for each primitive statement (e.g., an assignment)
  - a vertex for each condition from an `if` or `while` statement
- the set E of edges contains
  - an edge (v_1, v_2) iff v_2 may execute directly after v_1

Statement Dependencies

CFG Example

```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```



- (potential) execution path = path in CFG

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 a path from `A` to `B` in the CFG in which `v` is not written to

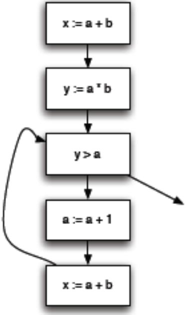
"The outcome of `A` can directly influence a variable read in `B`"

Reaching definitions: the definition of `v` in `A` reaches `B`

Statement Dependencies

Reaching Definitions Example

```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```




```
1: (x,?) (y,?) (a,?) (b,?)
2: (x,1) (y,?) (a,?) (b,?)
3: (x,1) (x,5) (y,2) (a,?) (a,4) (b,?)
4: (x,1) (x,5) (y,2) (a,?) (a,4) (b,?)
5: (x,1) (x,5) (y,2) (a,4) (b,?)
```

- Reaching definitions on entry to the respective node

Excursion: Computing Reaching Definitions

Instance of static program analysis

- Given a method by CFG (V, E)
- Analysis domain: $RD = Var \times (\{?\} \cup V)$
- Analysis result: $RDin, RDout : V \to RD$

Dataflow Equations

- $RDin(entry) = Var \times \{?\}$
- For each vertex $v \in V \setminus \{entry\}$:
  - $RDin(v) = \bigcup_{(v',v) \in E} RDout(v')$
- For each vertex $v \in V$:
  - $RDout(v) = (RDin(v) \setminus kill(v)) \cup gen(v)$
  - $kill(v) = \{x\} \times (\{?\} \cup V)$ if $v = [x := t]$
  - $kill(v) = \emptyset$ otherwise
  - $gen(v) = \{(x, v)\}$ if $v = [x := t]$
  - $gen(v) = \emptyset$ otherwise

Excursion: Computing Reaching Definitions

Solving data flow equations

Initialize

For each $v \in V$

- $RDin(v) = \emptyset$
- $RDout(v) = \emptyset$

Iterate

For each $v \in V$

- apply the dataflow equation for *RDin*
- apply the dataflow equation for *RDout*

until *RDin* and *RDout* do not change anymore

- Instance of a fixpoint analysis

Statement Dependencies

Definition (Control Dependency)

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

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

"The outcome of `A` can influence whether `B` is executed"

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
  mid = (low + high)/2;
  if ( target < array[ mid ] ) {
      high = mid - 1;
  } else if ( target > array[ mid ] ) {
      low = mid + 1;
  } else {
      return mid;
  }
}
return -1;
```

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
  mid = (low + high)/2;
  if ( target < array[ mid ] ) {
      high = mid - 1;
  } else if ( target > array[ mid ] ) {
      low = mid + 1;
  } else {
      return mid;
  }
}
return -1;
```

`mid` is data-dependent on this statement

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
  mid = (low + high)/2;
  if ( target < array[ mid ] ) {
      high = mid - 1;
  } else if ( target > array[ mid ] ) {
      low = mid + 1;
  } else {
      return mid;
  }
}
return -1;
```

`mid` is control-dependent on the while statement

Computing Backward Dependencies

Definition (Backward Dependency)

Statement $\texttt{B}$ is backward dependent on statement $\texttt{A}$ iff

There is a sequence of statements $\texttt{A} = \texttt{A}_1, \texttt{A}_2, \ldots, \texttt{A}_\texttt{n} = \texttt{B}$ such that:

1. for all i, $\texttt{A}_{i+1}$ is control dependent or data dependent on $\texttt{A}_i$
2. there is at least one i with $\texttt{A}_{i+1}$ being data dependent on $\texttt{A}_i$

"The outcome of A can influence the program state in B"

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
  mid = (low + high)/2;
  if ( target < array[ mid ] ) {
      high = mid - 1;
  } else if ( target > array[ mid ] ) {
      low = mid + 1;
  } else {
      return mid;
  }
}
return -1;
```

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
  mid = (low + high)/2;
  if ( target < array[ mid ] ) {
      high = mid - 1;
  } else if ( target > array[ mid ] ) {
      low = mid + 1;
  } else {
      return mid;
  }
}
return -1;
```

mid is backward-dependent on data- and control- dependent statement

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
   mid = (low + high)/2;
   if ( target < array[ mid ] ) {
         high = mid - 1;
   } else if ( target > array[ mid ] ) {
         low = mid + 1;
   } else {
         return mid;
   }
}
return -1;
```

mid is backward-dependent on data- and control- dependent statement

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
  mid = (low + high)/2;
  if ( target < array[ mid ] ) {
      high = mid - 1;
  } else if ( target > array[ mid ] ) {
      low = mid + 1;
  } else {
      return mid;
  }
}
return -1;
```

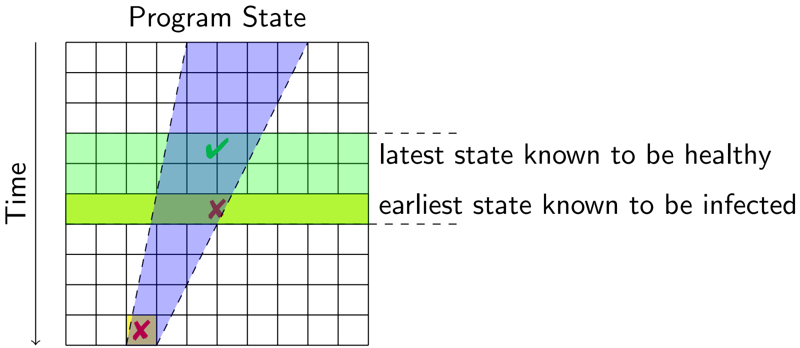
Backward-dependent statements for first execution of loop body

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
    mid = (low + high)/2;
    if ( target < array[ mid ] ) {
        high = mid - 1;
    } else if ( target > array[ mid ] ) {
        low = mid + 1;
    } else {
        return mid;
    }
}
return -1;
```

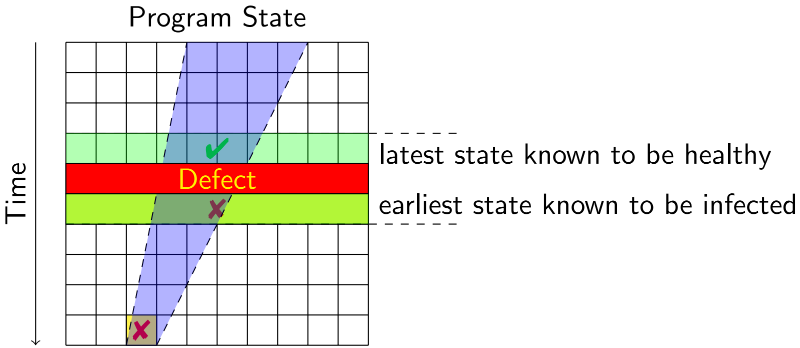
Backward-dependent statements for repeated execution of loop body

Systematic Discovery of Defects



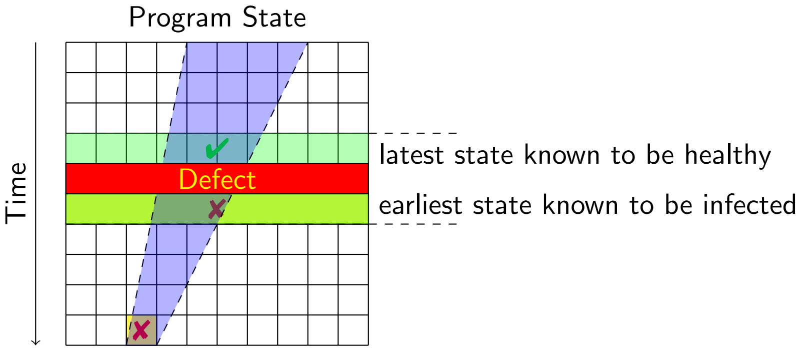
- Separate healthy from infected states
- Separate relevant from irrelevant states

Systematic Discovery of Defects



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

Systematic Discovery of Defects



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

Algorithm: Systematic Discovery of Defects

Invariant: $\mathcal{I}$ is a set of locations (variable set V and statement S) such that each $v \in V$ is infected after executing S.

1. Initialize $\mathcal{I} :=$ {infected location reported by failure}
2. Choose and remove an infected location $L = (V, S)$ from $\mathcal{I}$
3. Let $\mathcal{C} := \emptyset$ accumulate a set of candidates
4. For each statement S' that may contain origin of defect: S backwards depends on S' in one step in execution path
   - 4.1 Let V' be the set of variables that is written in S' and infected
   - 4.2 If $V' \neq \emptyset$ let $\mathcal{C} := \mathcal{C} \cup \{(V', S')\}$
5. If $\mathcal{C} \neq \emptyset$ (there are infected predecessors):
   - 5.1 Let $\mathcal{I} := \mathcal{I} \cup \mathcal{C}$
   - 5.2 Goto 2.
6. L depends only on healthy locations, it must be the infection site!

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
  mid = (low + high)/2;
  if ( target < array[ mid ] ) {
      high = mid - 1;
  } else if ( target > array[ mid ] ) {
      low = mid + 1;
  } else {
      return mid;
  }
}
return -1;
```

`mid` is infected, `mid==low==high==2`

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
  mid = (low + high)/2;
  if ( target < array[ mid ] ) {
      high = mid - 1;
  } else if ( target > array[ mid ] ) {
      low = mid + 1;
  } else {
      return mid;
  }
}
return -1;
```

Look for origins of `low` and `high`

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
  mid = (low + high)/2;
  if ( target < array[ mid ] ) {
      high = mid - 1;
  } else if ( target > array[ mid ] ) {
      low = mid + 1;
  } else {
      return mid;
  }
}
return -1;
```

`low` was changed in previous loop execution, value `low==1` seems healthy

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
  mid = (low + high)/2;
  if ( target < array[ mid ] ) {
      high = mid - 1;
  } else if ( target > array[ mid ] ) {
      low = mid + 1;
  } else {
      return mid;
  }
}
return -1;
```

high==2 set at start (if-branch not taken when target not found), infecte

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
  mid = (low + high)/2;
  if ( target < array[ mid ] ) {
      high = mid - 1;
  } else if ( target > array[ mid ] ) {
      low = mid + 1;
  } else {
      return mid;
  }
}
return -1;
```

high does not depend on any other location—found infection site!

Example

```
int low = 0;
int high = array.length - 1;
int mid;
while ( low <= high ) {
  mid = (low + high)/2;
  if ( target < array[ mid ] ) {
      high = mid - 1;
  } else if ( target > array[ mid ] ) {
      low = mid + 1;
  } else {
      return mid;
  }
}
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 ∞ many paths?

Literature for this Lecture

Essential

Zeller Why Programs Fail: A Guide to Systematic Debugging, Morgan Kaufmann, 2005
Chapters 7, 8, 9