Debugging

SWE 795, Spring 2017
Software Engineering Environments
Today

• Part 1 (Lecture)(~45 mins)
  • Debugging
• Part 2 (HW1 Presentations)(30 mins)
• Break!

• Part 3 (Discussion)(60 mins)
  • Discussion of readings
Example

？

retrieveRelationships

NPE

getStartContext

LaToza and Myers. Developers ask reachability questions. ICSE 2010.
```java
public void retrieveWithSchema(File file, String schema) {
    SchemaQueries sQueries = queries.getSchema()
    if (sQueries != null) {
        RelationshipDelta result = sQueries.runQueries(file, types);
        delta = RelationshipDelta.joinWithDelta(result);
        topLabels.addAll(sQueries.findTopObjects(file, types));
    }
}

public RelationshipContext getContext(Variable thisVar, AliasContext aliases) {
    RelationshipContext start = new RelationshipContext(Variable);
    RelationshipDelta converted = new RelationshipDelta();
    Map<ObjectLabel, ObjectLabel> bindings = new HashMap<ObjectLabel, ObjectLabel>();
    for (ObjectLabel possibleTop : topLabels) {
        String thisType = thisVar.resolveType().getQualifiedName();
        String possibleTopType = possibleTop.getType().getQualifiedName();
        if (possibleTopType != null) {
            Set<ObjectLabel> thisAliases = aliases.getAliases(thisVar);
            bindings.put(possibleTop, thisAliases.iterator().next());
        }
    }
    for (Entry<Relationship, ThreeValue<entry : delta>) {
        Relationship convertDelta = convertRelationship(thisVar, key, bindings);
        converted.addEntries(convertDelta, FourPointwise<entry, convert(entry.getValueValue());
    }
    return start.applyChangesFromDelta(converted);
}
```
Definitions

• Error - discrepancy between actual behavior of system and intended behavior

• Failure - incorrect output value, exception, etc.; an error that has become observable

• Fault - lines in code which are incorrect

• Debugging: determining the cause of a failure by localizing its location to a fault
  • More formally: **fault localization**
Edit / Debug Cycle

For tasks in code in your own codebase that you haven’t seen recently

LaToza and Myers. Developers ask reachability questions. ICSE 2010.
Steps in fixing bugs

• Reproduce the problem
• Fault localization
• Investigate fix
• Implement fix
• Test fix

• Will focus on fault localization today
Supporting debugging

• Why is it so challenging to go from failure to fault?
  • It may be unclear where behavior is implemented in code
  • Fault may occur far away from failure
  • How to find connection?
• Understanding why failure occurred may be challenging
What makes hard bugs hard to debug?

- Cause / effect chasm - symptom far removed from the root cause (15 instances)
  - timing / synchronization problems
  - intermittent / inconsistent / infrequent bugs
  - materialize many iterations after root cause
  - uncertain connection to hardware / compiler / configuration

- Inapplicable tools (12 instances)
  - Heisenbugs - bug disappears when using debugging tool
  - long run to replicate - debugging tool slows down long run even more
  - stealth bug - bug consumes evidence to detect bug
  - context - configuration / memory makes it impossible to use tool

- What you see if probably illusory (7 instances)
  - misreads something in code or in runtime observations

- Faulty assumption (6)

- Spaghetti code (3)

Traditional debugging techniques

• Stepping in debugger
• Logging - insert print statements or wrap particular suspect functions
• Dump & diff - use diff tool to compare logging data between executions
• Conditional breakpoints
• Profiling tool - detect memory leaks, illegal memory references

Debugging Strategies

• Strategies
  • Gather execution trace data
  • Formulate & test hypotheses
  • Traverse control & data dependencies backwards (slicing)
Formulate & test hypotheses

- Use knowledge & data so far to formulate hypothesis about why bug happened
cognition, meditation, observation, inspection, contemplation, hand-simulation,
gestation, rumination, dedication, inspiration, articulation

- Recognize cliche
  seen a similar bug before

- Controlled experiments - test hypotheses by gathering data

Some debugging techniques

- Record & **replay** execution (omniscient debuggers)
- Find temporary objects that aren’t garbage collected (Jinsight)
- Find shortest retro steps (delta debugging)
- **Differentiate** faulty from unfaulty executions (statistical debugging)
- Traverse control & data **dependencies** backwards (static slicers, dynamic slicers)
- Connected separated events by searching across control flow (Reacher)
- **Recommend** fixes other developers made for same error [See Crowdsourcing Lecture]
Record & replay execution

- Debugging in a debugger is hard
  - Forces developer to guess which methods to step into
  - Forces developers to guess which values to instrument
  - Changing guess requires reproing failure again
  - Can be time consuming

- What if developers could debug forwards and backwards?
Record & replay execution

- Record execution, step backwards / forwards through execution
  Biggest challenge - performance slowdown from logging - focus of most papers
- Example systems focused on user interactions
  - Retrace - on exception, backup several statements & start logging
  - ZStep94 - backwards / forwards stepping, find code which rendered graphics
  - Omniscient debugging - backwards / forwards stepping, step through writes to a variable
  - WhyLine - ask questions about output, traverse dynamic control & data dependencies, ask why didn’t questions
ZStep94

• Forwards / backwards stepping through execution events

  Go to end of program
  Show value of expression, without stopping
  Single step
  Single step backwards
  Back up from value to expression
  Go to beginning of program
  Single step "graphically"
  Single step backwards "graphically"

• Select graphical output, find code that drew it

  See value of selected variables

  "\[ \text{ab-left (left-side tree)} \rightarrow *, (A \ 'TREE \ (4 \ 3)) \]

  \[ \text{right-side tree)} \rightarrow *, (A \ 'TREE \ (1 \ 2)) \]

Demo: http://web.media.mit.edu/~lieber/Lieberary/ZStep/ZStep.mov

Omniscient debugger


Find temporary objects that aren’t garbage collected

Find shortest repro steps

- Long sequence of steps uncovered by tester triggers a bug.
- Which of these steps are causing the bug?
- Complex input - which part of input is responsible for bug?
- Example - 10,700 Mozilla bugs (11/20/2000)

Find shortest repro steps

- dadmin algorithm sketch:
  1. Decompose input into pieces
  2. Run tests on pieces
  3. If there’s a piece that still fails, go back to 1 on piece

  Otherwise, found locally minimal smallest input

---

<table>
<thead>
<tr>
<th>Step</th>
<th>Test case</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\Delta_1$</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>$\Delta_2$</td>
<td>$\times$</td>
</tr>
<tr>
<td>3</td>
<td>$\Delta_1$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>4</td>
<td>$\Delta_2$</td>
<td>$\times$</td>
</tr>
<tr>
<td>5</td>
<td>$\Delta_1$</td>
<td>$\times$</td>
</tr>
<tr>
<td>Result</td>
<td>. . . . . . . . 7 .</td>
<td>Done</td>
</tr>
</tbody>
</table>

Compare faulty & unfaulty execution traces

- Idea: bugs caused by executing buggy statements
  Find buggy statements executed **mostly** on failing tests (color red)

```c
mid() {
    int x, y, z, m;
    read("Enter 3 numbers:", x, y, z);
    if (y < z) {
        if (x < y) {
            m = y;
        } else if (x < z) {
            m = y;
        } else {
            m = y;
        }
    } else if (x > z) {
        m = y;
    } else {
        m = x;
    }
    print("Middle number is:", m);
}
```

**Test Cases**

<table>
<thead>
<tr>
<th>Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.5</td>
</tr>
<tr>
<td>12.3</td>
</tr>
<tr>
<td>32.4</td>
</tr>
<tr>
<td>53.5</td>
</tr>
<tr>
<td>53.4</td>
</tr>
<tr>
<td>21.3</td>
</tr>
</tbody>
</table>

```
suspiciousness(e) = 1 - hue(e) = \\
= \frac{\text{failed(e)}}{\text{totalfailed}} \\
= \frac{\text{passed(e)}}{\text{totalpassed} + \text{failed(e)}}
```
Compare faulty & unfaulty execution traces

Tarantula - frequency of failing runs relative to passing runs (“suspiciousness”)
Union: \((U \text{ passing_tests}) - \text{failing_test}\)
Intersection: intersect passed test statements, subtract failing tests statements
Nearest neighbor (NN): failing_test - most_similar_passing_test
Cause transition (CT): find smallest memory difference

Compare faulty & unfaulty execution traces

User hits bug and program crashes
Program (e.g. Microsoft Watson) logs stack trace
Stack trace sent to developers
Tool classifies trace into bug buckets

Problems
WAY too many bug reports => way too many open bugs
=> can’t spend a lot of time examining all of them
Mozilla has 35,622 open bugs plus 81,168 duplicates (in 2004)

Stack trace not good bug predictor for some systems (e.g. event based systems)
⇒ bugs may be in multiple buckets or multiple bugs in single bucket

Stack trace may not have enough information to debug
⇒ hard to find the problem to fix

Compare faulty & unfaulty execution traces

- Program runs on user computer
  - Crashes or exhibits bug (failure)
  - Exits without exhibiting bug (success)
- Counters count # times predicates hit
  - Counters sent back to developer for failing and successful runs
- Statistical debugging finds predicates that predict bugs
  - 100,000s to millions of predicates for small applications
  - Finds the best bug predicting predicates amongst these
- Problems to solve
  - Reports shouldn’t overuse network bandwidth (esp ~2003)
  - Logging shouldn’t kill performance
  - Interesting predicates need to be logged (fair sampling)
  - Find good bug predictors from runs
  - Handle multiple bugs in failure runs

Compare faulty & unfaulty execution traces

• Predictor of what statements are related to a bug:
  \[
  \text{Fail}(P) - \text{Context}(P) = \Pr(\text{Crash} | P \text{ observed to be true}) - \Pr(\text{Crash} | P \text{ observed at all})
  \]

• Example of a “likelihood ratio test”

• Comparing two hypotheses
  1. Null Hypothesis: \(\text{Fail}(P) \leq \text{Context}(P)\)
     \[\alpha \leq \beta\]
  2. Alternative Hypothesis: \(\text{Fail}(P) > \text{Context}(P)\)
     \[\alpha > \beta\]

Traverse dependencies

- **Slice**
  - Subset of the program that is responsible for computing the value of a variable at a program point

- **Backwards slice**
  - Transitive closure of all statements that have a control or data dependency

- Originally formulated as *subset* of program
- Later formulations emphasize ability to traverse control & data dependencies (e.g., WhyLine)
Traverse control & data dependencies backwards

- BEGIN
  READ(X, Y)
  TOTAL := 0.0
  SUM := 0.0
  IF X <= 1
      THEN SUM := Y
  ELSE BEGIN
      READ(Z)
      TOTAL := X * Y
  END
  WRITE(TOTAL, SUM)
END

- (Static) slice - subset of the program that produces the same variable values at a program point
- Slice on variable Z at 12

Participants performed 3 debugging tasks on short code snippets

Asked to recognize code snippets afterwards

Slicers debug faster

• Students debugging 100 LOC C++ programs
• Students given
  Programming environment
  Hardcopy input, wrong output, correct output
  Files with program & input
• Compared students instructed to slice against everyone else
  Excluding students who naturally use slicing strategy
• Slicers debug significantly faster (65.29 minutes vs. 30.16 minutes)

Dynamic slicing