Debugging

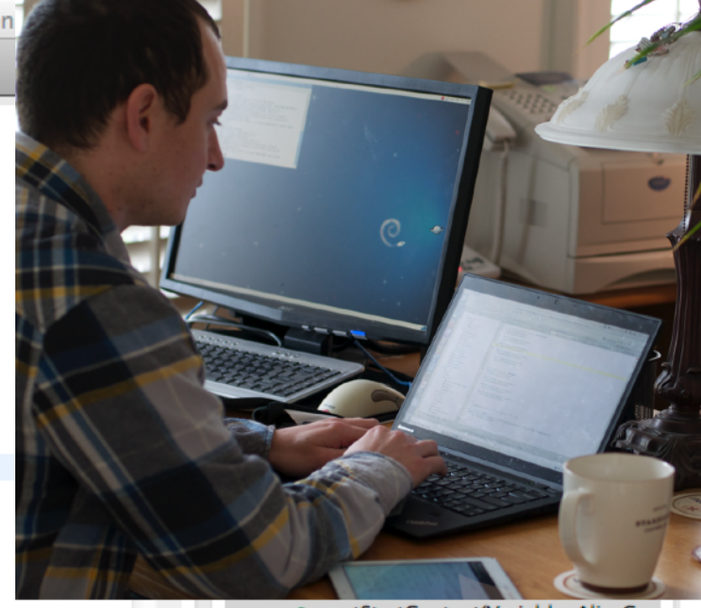
SWE 795, Fall 2019

Software Engineering Environments

Today

- Part 1 (Lecture)(~80 mins)
  - Debugging
- Break!
- Part 3 (Discussion)(45 mins)
  - Discussion of readings

Java – Fusion/src/edu/cmu/cs/fusion/xml/XMLRetriever.java – Eclipse SDK – /Users/tlatoza/Documents/Code/Reachability question



Package Explorer

- edu.cmu.cs.crystal.flow.concur 223
- edu.cmu.cs.crystal.flow.worklist 260
- edu.cmu.cs.crystal.internal 272
- edu.cmu.cs.crystal.simple 225
- edu.cmu.cs.crystal.tac 284
- edu.cmu.cs.crystal.tac.eclipse 255
- edu.cmu.cs.crystal.tac.model 270
- edu.cmu.cs.crystal.util 282
- edu.cmu.cs.crystal.util.typehierarchy 282
- test 276
- analyses 284
- Plug-in Dependencies
- JRE System Library [JVM 1.6.0 (MacOS X Default)]
- doc 269
- icons 3
- lib 22
- log 3
- META-INF 276
- schema 91
- build.properties 250
- COPYING 3
- COPYING.LESSER 3
- copyright-notice.txt 129
- javadoc.xml 236
- plugin.xml 260
- > Fusion 36 [https://fusion.googlecode.com/svn, Tr
  - > src 36
    - > edu.cmu.cs.fusion 36
    - > edu.cmu.cs.fusion.constraint 36
    - edu.cmu.cs.fusion.constraint.operations 32
    - edu.cmu.cs.fusion.constraint.predicates 36
    - > edu.cmu.cs.fusion.constraint.requestors 87
    - edu.cmu.cs.fusion.parser 41
    - > edu.cmu.cs.fusion.parsers.predicate 41
    - edu.cmu.cs.fusion.rawannotations 41
    - > edu.cmu.cs.fusion.relationship 36
    - > edu.cmu.cs.fusion.xml
      - > NamedTypeBinding.java
      - SchemaQueries.java
      - TypeComparisonCall.java
      - TypeComparisonDefinition.java
      - > XMLFileVisitor.java
      - XMLObjectLabel.java
      - XMLRetriever.java
  - > test 36
  - JRE System Library [JVM 1.6.0 (MacOS X Default)]
  - Plug-in Dependencies
  - Referenced Libraries
  - > META-INF 24
  - > build.properties 24
  - Fusion.xsd 88
  - plugin.xml 87
  - saxon9he.jar
- > FusionTests 74 [https://fusion.googlecode.com/s
- PlaidAnnotations 62 [http://plaidannotations.google

XMLRetriever.java

```
    public void retrieveWithSchema(File file, String schema) {
        SchemaQueries sQueries = queries.get(schema);

        if (sQueries != null) {
            RelationshipDelta result = sQueries.runQueries(file, types);
            delta = RelationshipDelta.join(delta, result);
            topLabels.addAll(sQueries.findTopObjects(file, types));
        }
    }

    public RelationshipContext getStartContext(Variable thisVar, AliasContext aliases) {
        RelationshipContext start = new RelationshipContext(false);
        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 (types.isSubtypeCompatible(thisType, possibleTopType)) {
                Set<ObjectLabel> thisAliases = aliases.getAliases(thisVar);
                assert (thisAliases.size() == 1);
                bindings.put(possibleTop, thisAliases.iterator().next());
            }
        }

        for (Entry<Relationship, ThreeValue> entry : delta) {
            Relationship convDelta = convertRelationship(entry.getKey(), bindings);
            converted.setRelationship(convDelta, FourPointLattice.convert(entry.getValue()));
        }
        return start.applyChangesFromDelta(converted);
    }
```

- getStartContext(Variable, AliasCon
- convertRelationship(Relationship, I

Problems | Javadoc | Declaration | Call Hierarchy

Members calling 'getStartContext(Variable, AliasContext)' – in workspace

- performAnalysis() : AnalysisResult<LE, N, OP> – edu.cmu.cs.crystal.flow.workl
  - performAnalysis(MethodDeclaration) : void – edu.cmu.cs.crystal.flow.Mothe
    - switchToMethod(MethodDeclaration) : void – edu.cmu.cs.crystal.flow.Mc
      - performAnalysisOnSurroundingMethodIfNeeded(ASTNode) : void – ec
        - getEndResults(MethodDeclaration) : LE – edu.cmu.cs.crystal.flow.M
          - analyzeMethod(MethodDeclaration) : void – edu.cmu.cs.fusion.
            - runAnalysis(IAnalysisReporter, IAnalysisInput, ICompilationU
              - run(AnnotationDatabase) : void – edu.cmu.cs.crystal.inter
        - getLabeledEndResult(MethodDeclaration) : IResult<LE> – edu.cmu
        - getLabeledResultAfter(ICFGNode<ASTNode>) : IResult<LE> – edu
        - getLabeledResultBefore(ICFGNode<ASTNode>) : IResult<LE> – ed
        - getLabeledResultsAfter(ASTNode) : IResult<LE> – edu.cmu.cs.crys
          - deriveResult(EclipseInstructionSequence, LE, TACInstruction, bc
          - getLabeledResultsAfter(ASTNode) : IResult<LE> – edu.cmu.cs.c
            - getLabeledResultsAfter(ASTNode) : IResult<LE> – edu.cmu.c
          - getLabeledResultsAfter(TACInstruction) : IResult<LE> – edu.cm
        - getLabeledResultsBefore(ASTNode) : IResult<LE> – edu.cmu.cs.cry
        - getLabeledStartResult(MethodDeclaration) : IResult<LE> – edu.cm
        - getResultsOrNullAfter(ASTNode) : LE – edu.cmu.cs.crystal.flow.Mo
        - getResultsOrNullBefore(ASTNode) : LE – edu.cmu.cs.crystal.flow.M
        - getStartResults(MethodDeclaration) : LE – edu.cmu.cs.crystal.flow.
- getEntryValue() : LE – edu.cmu.cs.crystal.flow.worklist.BranchSensitiveWorklist

	Line	Call
➪	203	performAnalysisOnSurroundingMethodIfNeeded(d)

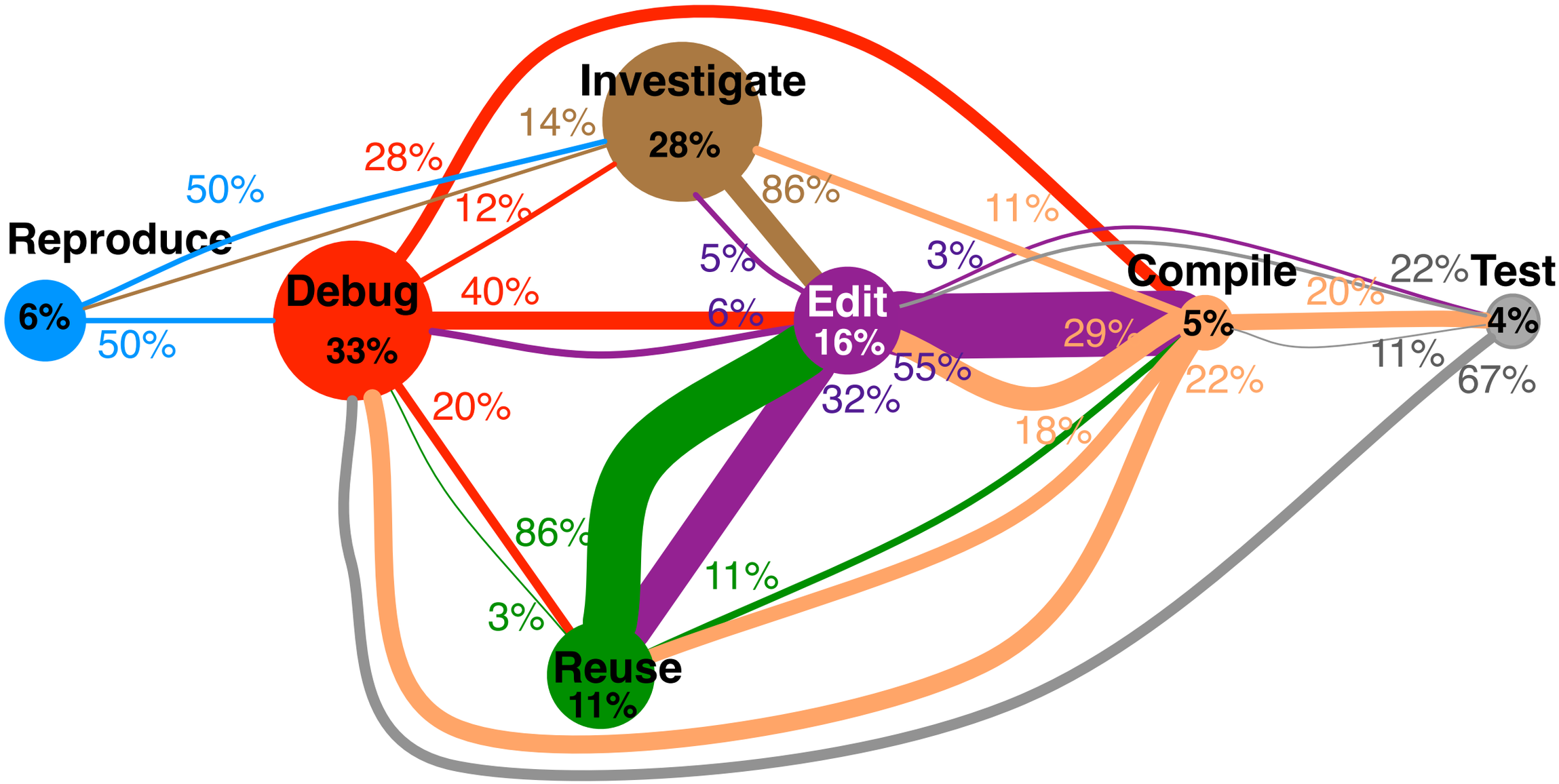
latoza@gmail.com edu.cmu.cs.fusion.xml.XMLRetriever.java – Fusion/src

Steps in fixing bugs

- Reproduce the problem
- Find cause of defect
- Investigate fix
- Implement fix
- Test fix

- Will focus on **finding cause of defect** today

Edit / Debug Cycle

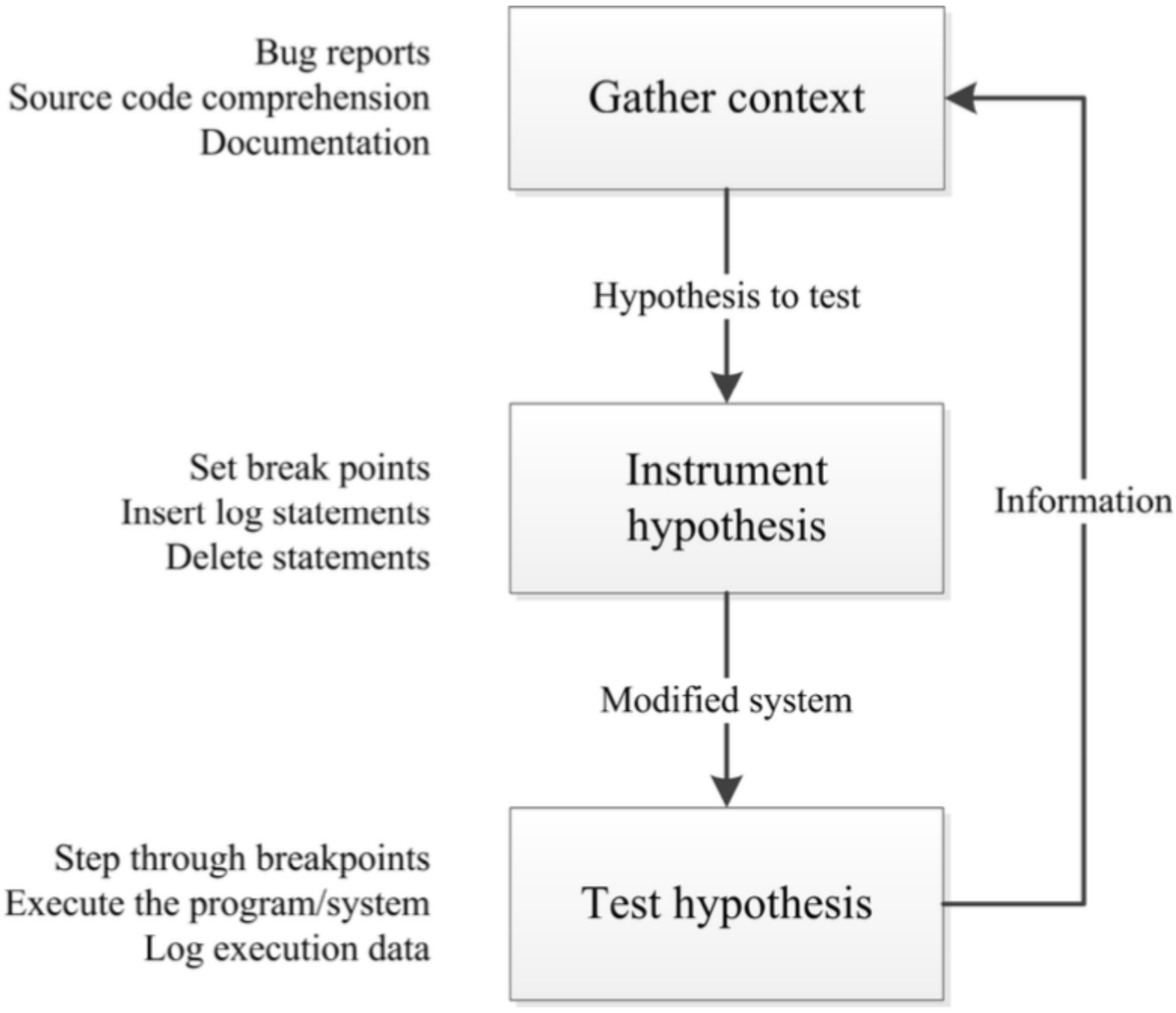


Circle size: % of time **Edge thickness:** % of transitions observed

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

LaToza and Myers. Developers ask reachability questions. ICSE 2010.

Debugging process model



L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline and G. Venolia, "Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers," *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Baltimore, MD, 2013, pp. 383-392.

Formulate & test hypotheses

- Use knowledge & data so far to formulate hypothesis about why bug happened
  cogitation, 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

Eisenstadt, M. Tales of Debugging from the Front Lines. Proc. Empirical Studies of Programmers, Ablex Publishing, Norwood, NJ, 1993, 86-112.

Resources for testing hypotheses

# subjects	Hypothesis instrumentation methods
7	Inserting breakpoints and watch variables
4	Inserting log statements
2	Removing irrelevant code
2	Tweaking - modifying existing code

# subjects	Hypothesis testing and comparison methods
7	Stepping in the debugger
4	Comparing against examples
2	Comparing against an oracle
1	Analyzing network packets
1	Backtracking
1	Printing out hard copies of code

L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline and G. Venolia, "Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers," *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Baltimore, MD, 2013, pp. 383-392.

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
  - May involve finding location (fault localization) as well as explanation.

Resources used in debugging

# subjects	Resources used in debugging
15	Debugger tools
14	Bug information
12	Communication with others
9	Internet resources
7	Custom code/manual debugging data
6	System state information (variables, packets)
5	Searching the source repository
4	Code browsers
3	Printed publications
2	Production health/status/monitoring systems
2	Build information
1	Personal library of technical tidbits
1	Shared internal development team resources
1	Product documentation

L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline and G. Venolia, "Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers," *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Baltimore, MD, 2013, pp. 383-392.

Information needs in debugging

- *How did this **runtime state** occur? (12)*

  data, memory corruption, race conditions, hangs, crashes, failed API calls, test failures, null pointers

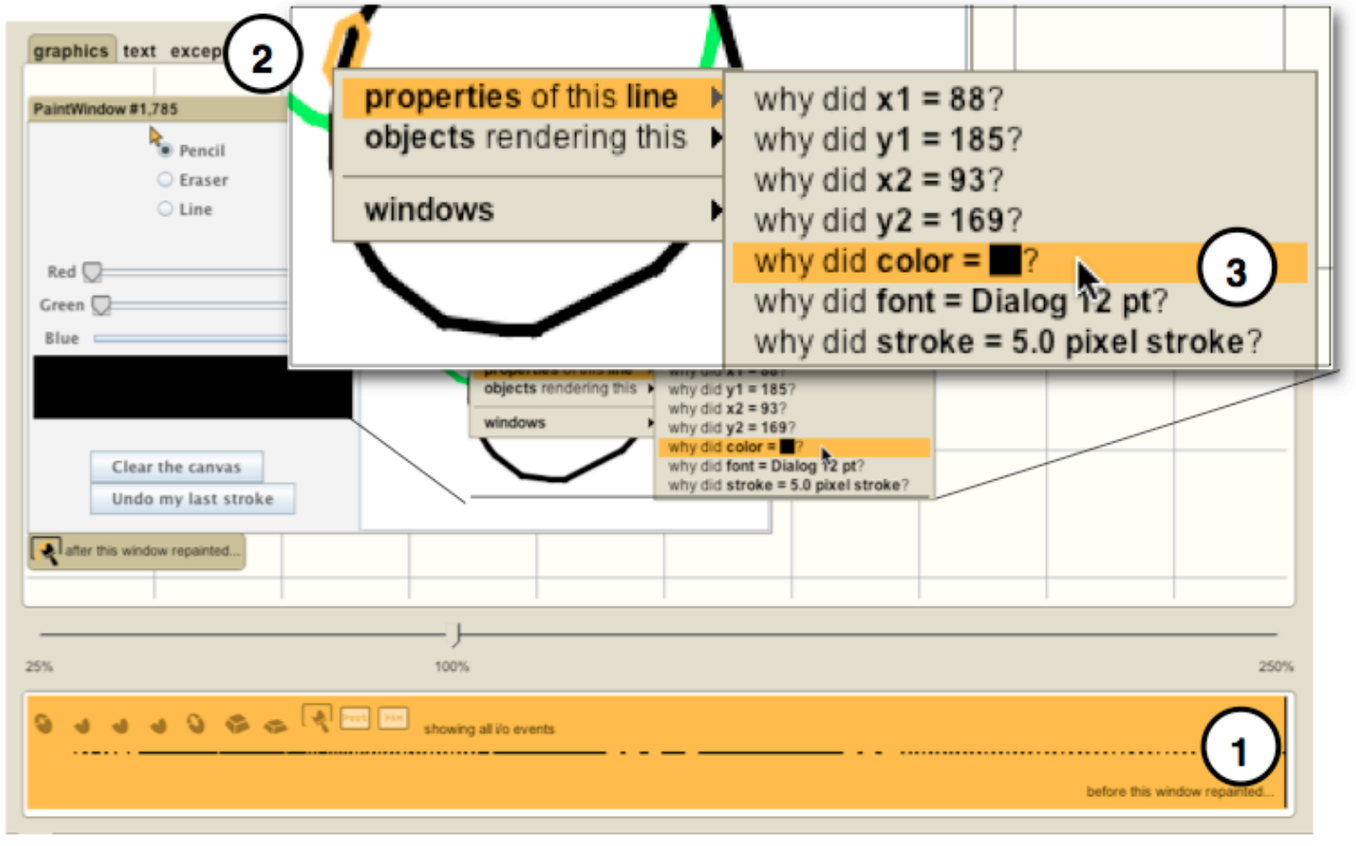
- *Where was this **variable** last changed? (1)*
- *Why **didn't** this happen? (3)*

omniscient debuggers

Record execution history
Provide interactions for browsing or searching

WhyLine

directly supports all 3 questions in some situations



LaToza and Myers. Hard-to-answer questions about code. PLATEAU 2010.

* *How do I debug this bug in this **environment**?(3)*

* *In what **circumstances** does this bug occur? (3)*

statistical debugging [1]

*-Sample execution traces on **user** computers*
*-Find **correlations** between crashes and predicates*

No need to reproduce environment on developer computer

Examine correlations between crashes and predicates

[1] Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I. 2003. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*.

- *How is this object **different** from that object? (1)*

- *What runtime state **changed** when this executed? (2)*

- *Which **team's** component caused this bug? (1)*

Which team should I assign this bug to?

Information needs in debugging

- What code could have caused this behavior?
- What's statically related to this code?
- What code cause this program state?

A: Why did I get gibberish? Storing field, given PPack, what is an MPField? I have no idea what this data structure contains. SPSField? I suspect SPS is just busted.

Andrew J. Ko, Robert DeLine, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. In Proceedings of the 29th international conference on Software Engineering (ICSE '07). IEEE Computer Society, Washington, DC, USA, 344-353. DOI: https://doi.org/10.1109/ICSE.2007.45

Activity

- What's the hardest debugging bug you've ever debugged?
- What made it hard?

What makes debugging hard?

# subjects	Debugging challenges
11	Environmental challenges
7	Multithreaded/multicore
6	Information quality
6	Communication challenges
6	Unable to reproduce failures consistently
4	Debugging process challenges

# subjects	Debugging challenges
6	Capture and replay of production events
3	More contextual information in runtime logs/stack traces
3	Integrating data from different sources
3	Bi-directional debugger
3	Debugging tool training
3	Multithreaded support
2	Automatic breakpoints upon entry into a class
2	Automated log analysis
2	Program context
2	Visually showing the execution trace

L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline and G. Venolia, "Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers," *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Baltimore, MD, 2013, pp. 383-392.

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)

Eisenstadt, M. Tales of Debugging from the Front Lines. Proc. Empirical Studies of Programmers, Ablex Publishing, Norwood, NJ, 1993, 86-112.

What makes hard bugs hard to debug?

Table 6 Root cause of the hardest bug (number of answers given).

memory	parallel	vendor	design	init	variable
42	53	41	82	9	3
lexical	ambiguous	user	unknown	other	
1	6	5	29	32	

Table 7 Most useful technique to find the hardest bug (number of answers given).

stepping	wrapper	printf	log diff	breakpoints	tool
54	5	33	12	38	15
reading	expert	experiments	not fixed	other	
41	4	58	31	12	

Table 8 Main difficulty source for hardest bug (number of answers given).

distance	tools	output	assumption	bad code	unknown	other
87	47	1	33	38	35	62

Activity

- What is a strategy you've used to debug a defect?

Some debugging strategies

- Backwards: Find statement that generated incorrect output, follow data and control dependencies backwards to find incorrect line of code
- Forwards: Find event that triggered incorrect behavior, follow control flow forward until incorrect state reached
- Input manipulation: Edit inputs, observe differences in output
- Blackbox debugging: Find documentation, code examples to understand correct use of API

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

Eisenstadt, M. Tales of Debugging from the Front Lines. Proc. Empirical Studies of Programmers, Ablex Publishing, Norwood, NJ, 1993, 86-112.

Debugging tools

- Make breakpoint debuggers **better**
  - Support stepping backwards (omniscient debuggers)
  - Support finding statement that generated incorrect output
- Find **part** of program that generated incorrect output (slicing)
  - Output: subset of program
- Help developers **follow** data and control dependencies backwards
  - Support for navigating control and data dependencies (WhyLine)
- **Compare** execution across different runs to guess locations that might be related (automatic debugging)
  - Output: list of potential fault locations
- **Simplify** input to find a simpler input that still generates failure (delta debugging)
  - Output: simplified input
- Enable developers to directly answer questions about **causality**
  - Support searching across control and data flow (Reacher)
- Help developers understand execution **state**
  - Show visualizations which depict the current contents of memory
- Let developers **experiment**, editing program till they find the right one (live programming)[editing code]

Program analysis building blocks

- Many tools rely on gathering an execution trace
  - Record the value of every expression as it executes (or sometimes at function boundaries)
  - Challenge: scalability
- Other tools use log data
  - Gives developer control over what is being logged
  - More easily scalable, requires developer to control what is logged
- Other tools use test coverage data
  - Which statement executes on each test, test passing or succeeding

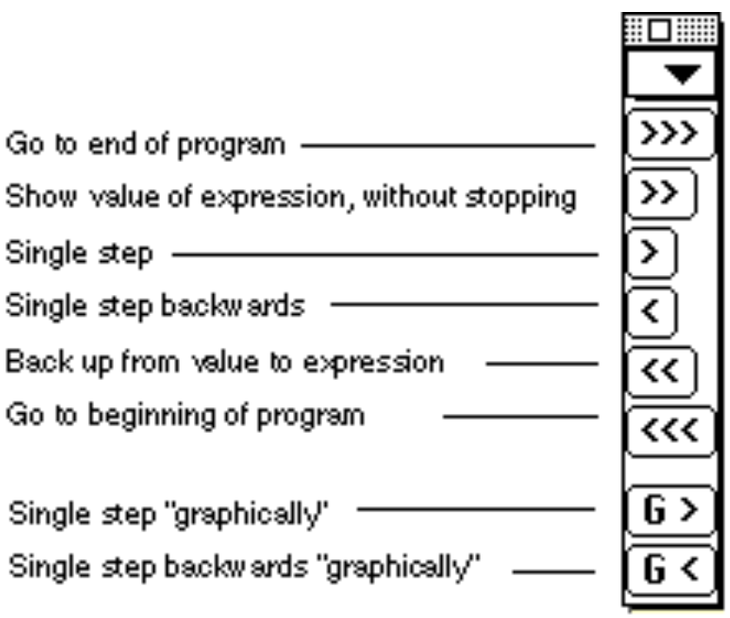
Make breakpoint debugging better

- 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

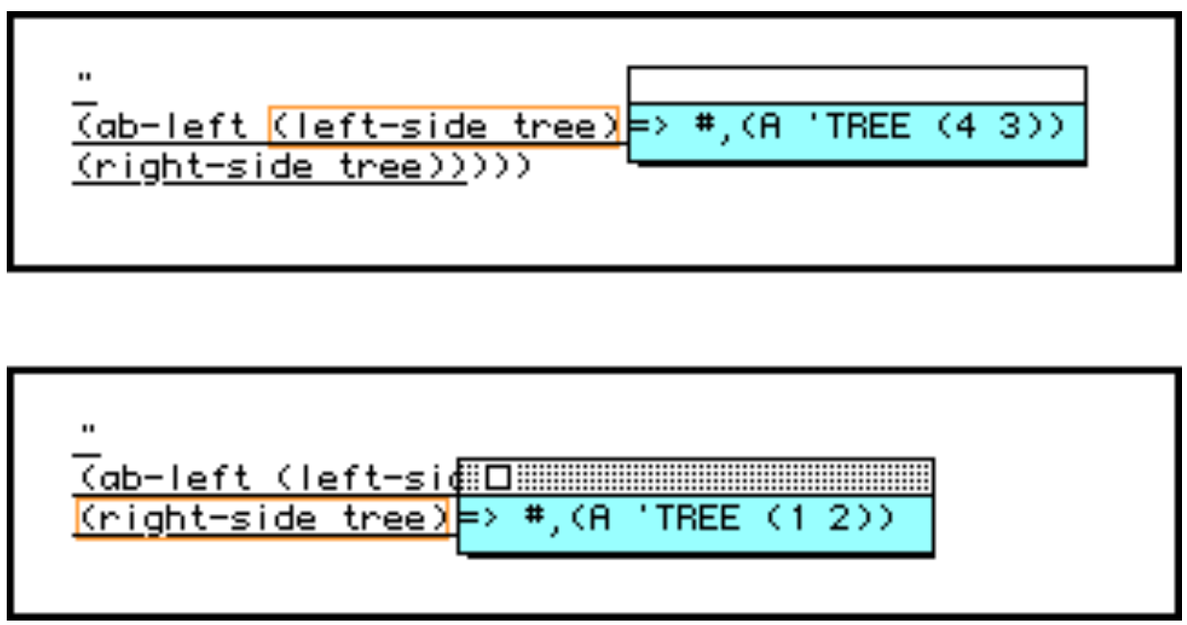
- What if developers could debug forwards **and** backwards?

ZStep94

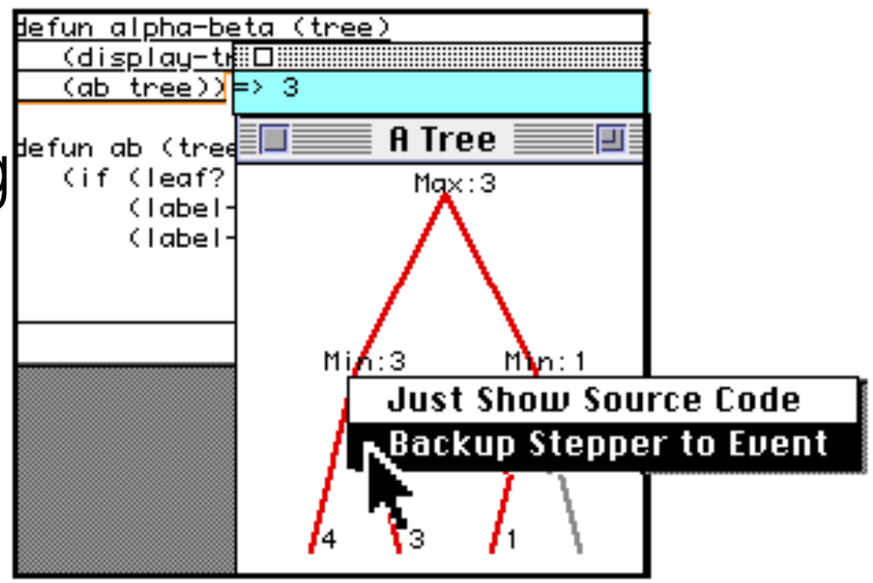
- Forwards / backwards stepping through execution events



See value of selected variables



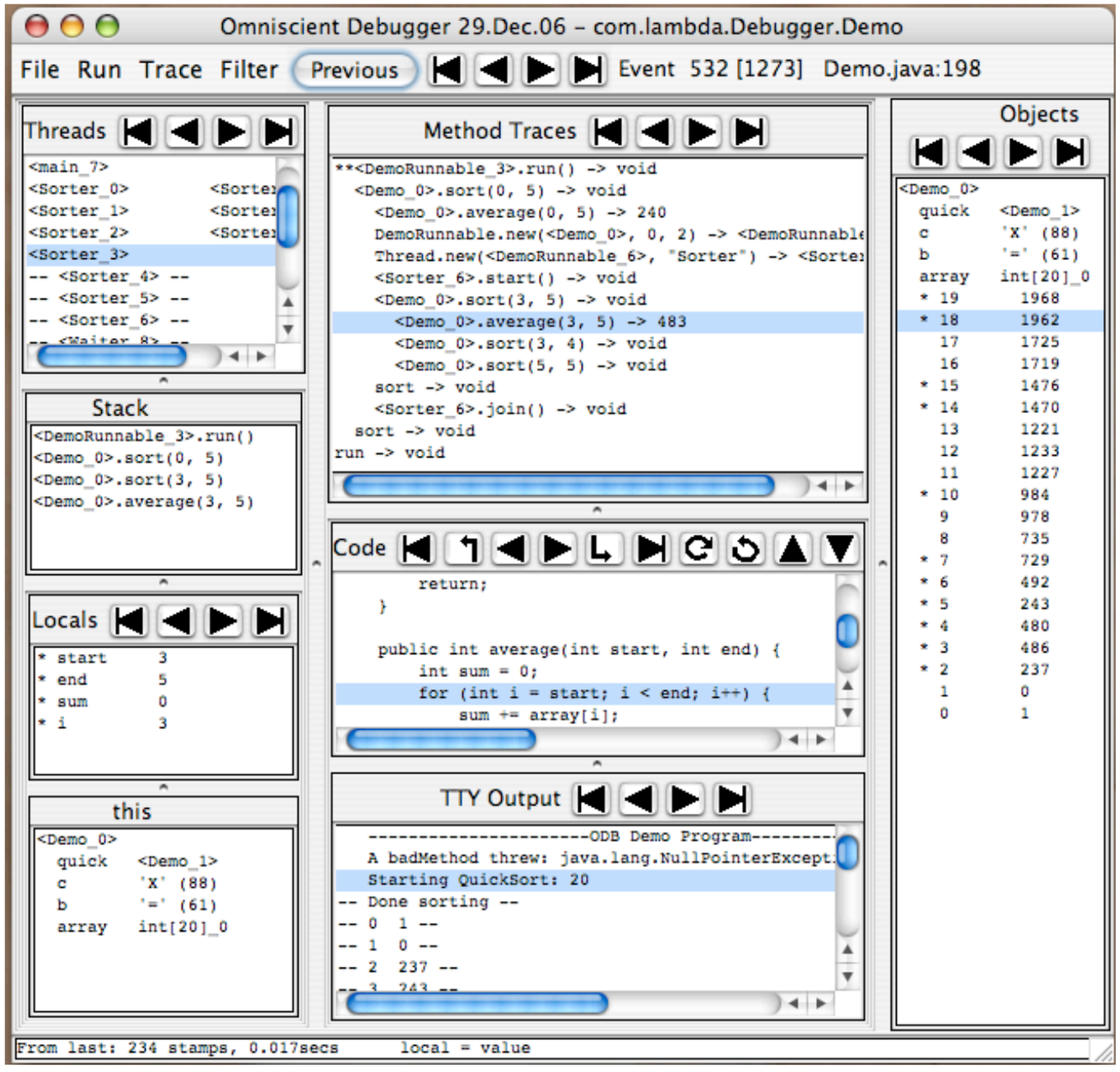
- Select g e that drew it
-



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

Henry Lieberman and Christopher Fry. 1995. Bridging the gulf between code and behavior in programming. In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '95), 480-486.

Omniscient debugger



Demo / talk: http://video.google.com/videoplay?docid=3897010229726822034#

Bill Lewis. Debugging backwards in time. In Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003), October 2003.

Associating incorrect output with responsible code

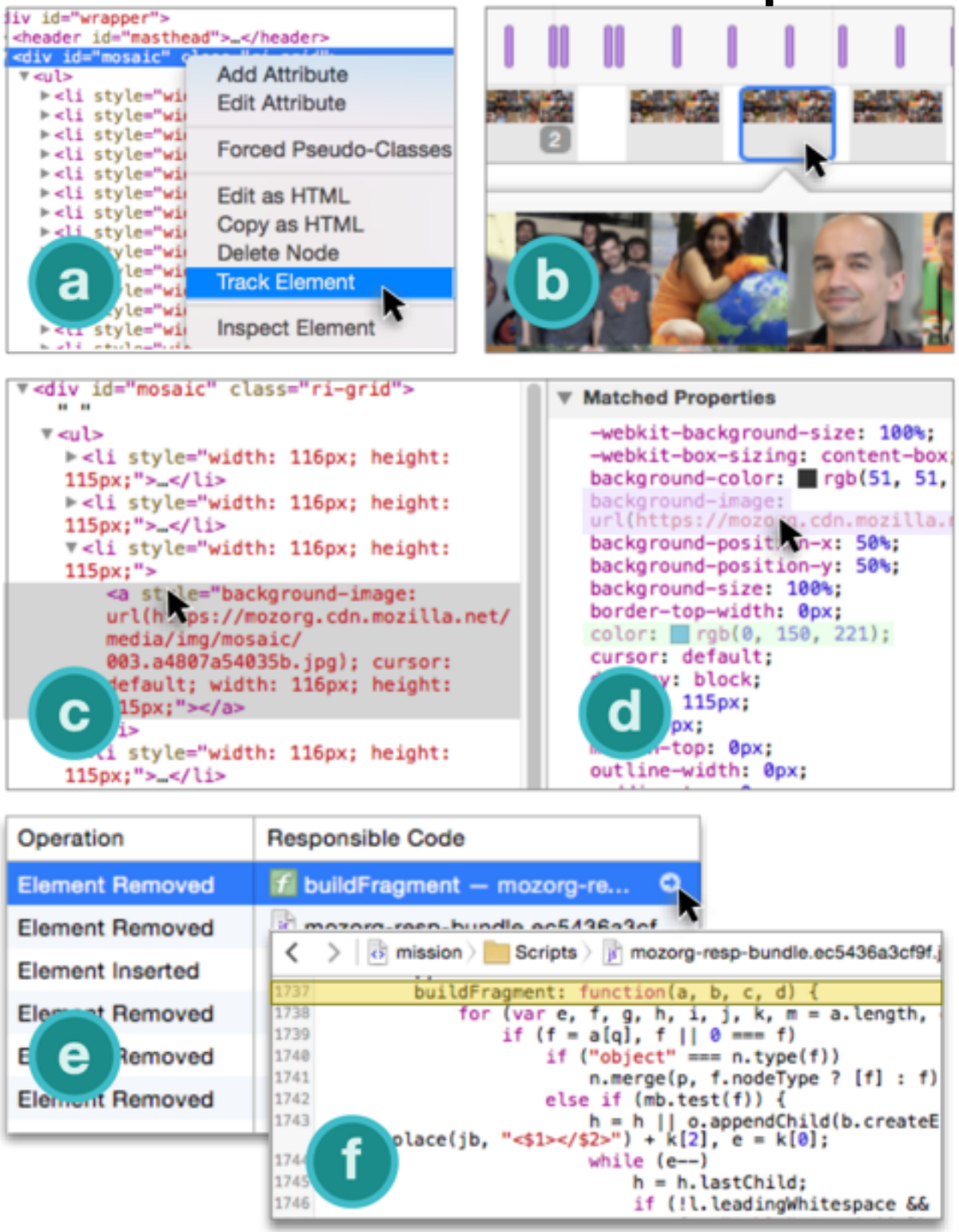


Figure 2. Susan first uses the Web Inspector to go from the mosaic's visual output to its DOM elements. Then, she uses Scry to track changes to the mosaic element (a), select different visual states to inspect (b), and see the DOM tree (c) and CSS styles (d) that produced each visual state. To jump to the code that implements interactive behaviors, Susan uses Scry to compare two states and then selects a single style property difference (d). Scry shows the mutation operations indirectly responsible for causing the property difference (e), and Susan can jump to JavaScript code (f) that performed each mutation operation.

Brian Burg, Andrew J. Ko, and Michael D. Ernst. 2015. Explaining Visual Changes in Web Interfaces. In Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15). ACM, New York, NY, USA, 259-268. DOI: https://doi.org/10.1145/2807442.2807473

Find part of the program that caused incorrect output

- 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

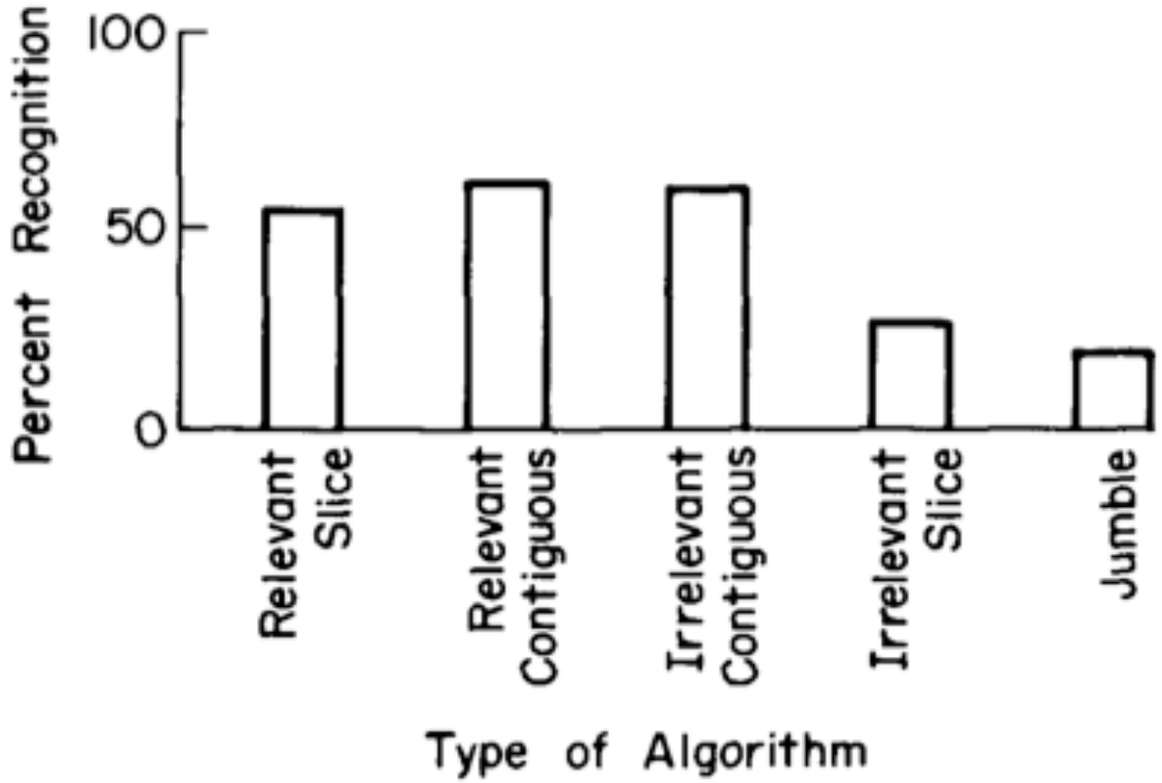
- Originally formulated as **subset** of program

Early evidence for slicing

- 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 progra 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



Mark Weiser. 1982. Programmers use slices when debugging. Commun. ACM 25, 7 (July 1982), 446-452.

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)

Francel M. A. and S. Rugaber (2001). The Value of Slicing While Debugging, *Science of Computer Programming*, 40(2–3), 151–169.

Dynamic slicing

/u17/ha/v2/demo/example.bug.c

```
1   /* Find the sum of areas of given triangles. */
2   #define MAX 100
3   typedef enum {isosceles, equilateral, right, scalene} class_type;
4   typedef struct {int a, b, c;} triangle_type;
5
6   main()
7   {
8       triangle_type sides[MAX];
9       class_type class;
10      int a_sqr, b_sqr, c_sqr, N, i;
11      double area, sum, s, sqrt();
12
13      printf("Enter number of triangles:\n");
14      scanf("%d", &N);
15      for (i = 0; i < N; i++) {
16          printf("Enter three sides of triangle %d in ascending order:\n", i+1);
17          scanf("%d %d %d", &sides[i].a, &sides[i].b, &sides[i].c);
18      }
19
20      sum = 0;
21      i = 0;
22      while (i < N) {
23          a_sqr = sides[i].a * sides[i].a;
24          b_sqr = sides[i].b * sides[i].c;
25          c_sqr = sides[i].c * sides[i].c;
26          if ((sides[i].a == sides[i].b) && (sides[i].b == sides[i].c))
27              class = equilateral;
28          else if ((sides[i].a == sides[i].b) || (sides[i].b == sides[i].c))
29              class = isosceles;
30          else if (a_sqr == b_sqr + c_sqr)
31              class = right;
32          else class = scalene;
33
34          if (class == right)
35              area = sides[i].b * sides[i].c / 2.0;
36          else if (class = equilateral)
37              area = sides[i].a * sides[i].a * sqrt(3.0) / 4.0;
38          else {
39              s = (sides[i].a + sides[i].b + sides[i].c) / 2.0;
40              area = sqrt(s * (s - sides[i].a) * (s - sides[i].b) *
41                      (s - sides[i].c));
42          }
43          sum += area;
44          i += 1;
45      }
46      printf("Sum of areas of the %d triangles is %.2f.\n", N, sum);
47  }
48
49
50
```

static analysis | approx. dynamic analysis | exact dynamic analysis

program slice | data slice | control slice | reaching defs | new testcase | clear

run | stop | continue | print | backup | step | stepback | delete | quit

```
stopped at line 47.
> stop at line 46
> backup
stopped at line 46.
> select exact dynamic analysis
> dynamic data slice on "sum" at line 46
>
```

Current Testcase #: 1

Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. 1993. Debugging with dynamic slicing and backtracking. Softw. Pract. Exper. 23, 6 (June 1993), 589-616.

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)

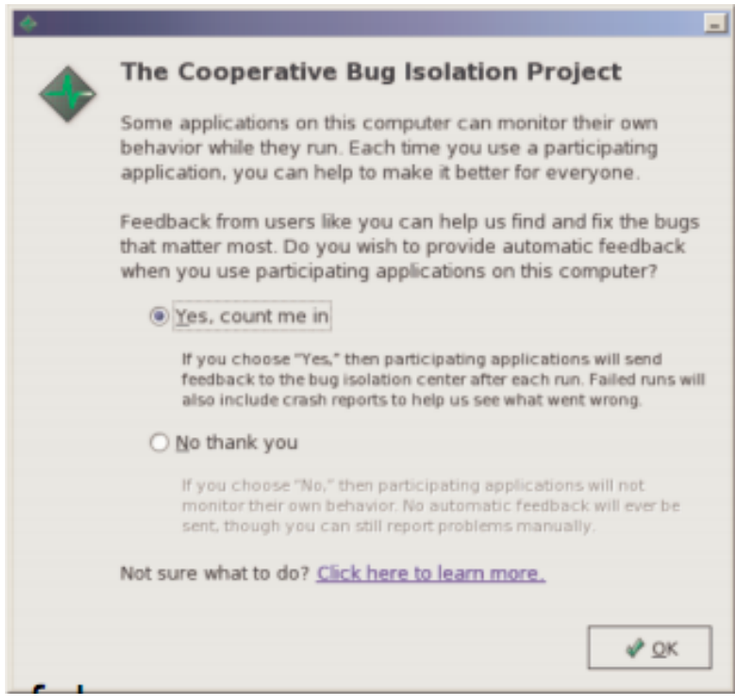
Stack trace not good bug predictor for some systems (e.g. event based systems)
⇒ bugs may be in multiple buckets or multiple bugs in single bucket

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

Ben Liblit. (2005). Cooperative bug isolation. Dissertation, UC Berkeley.

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



Ben Liblit. (2005). Cooperative bug isolation. Dissertation, UC Berkeley.

Compare faulty & unfaulty execution traces

- Predictor of what statements are related to a bug:

  Fail(P) - Context(P)

  Pr(Crash | P observed to be true) - Pr(Crash | P observed at all)

- Example of a “likelihood ratio test”
  - Comparing two hypotheses
  - 1. Null Hypothesis: Fail(P) <= Context(P)

    Alpha <= Beta
  - 2. Alternative Hypothesis: Fail(P) > Context(P)

    Alpha > Beta

Ben Liblit. (2005). Cooperative bug isolation. Dissertation, UC Berkeley.

Simplify failure inducing input

- 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)

```
<td align=left valign=top>
<SELECT NAME="op_sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows 95">Windows 95<OPTION VALUE="Windows
98">Windows 98<OPTION VALUE="Windows ME">Windows ME<OPTION VALUE="Windows 2000">Windows 2000<OPTION VALUE="Windows
NT">Windows NT<OPTION VALUE="Mac System 7">Mac System 7<OPTION VALUE="Mac System 7.5">Mac System 7.5<OPTION VALUE="Mac
System 7.6.1">Mac System 7.6.1<OPTION VALUE="Mac System 8.0">Mac System 8.0<OPTION VALUE="Mac System 8.5">Mac System
8.5<OPTION VALUE="Mac System 8.6">Mac System 8.6<OPTION VALUE="Mac System 9.x">Mac System 9.x<OPTION VALUE="MacOS X">MacOS
X<OPTION VALUE="Linux">Linux<OPTION VALUE="BSDI">BSDI<OPTION VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION
VALUE="OpenBSD">OpenBSD<OPTION VALUE="AIX">AIX<OPTION VALUE="BeOS">BeOS<OPTION VALUE="HP-UX">HP-UX<OPTION
VALUE="IRIX">IRIX<OPTION VALUE="Neutrino">Neutrino<OPTION VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION
VALUE="OSF/1">OSF/1<OPTION VALUE="Solaris">Solaris<OPTION VALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION
VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug_severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION VALUE="major">major<OPTION
VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```

Fig. 1. Printing this HTML page makes Mozilla crash (excerpt)

Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. IEEE Transactions on Software Engineering 28(2), February 2002, pp. 183-200.

Find shortest repro steps

- ddmin 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
-

Step	Test case									*test*	
1	Δ_1	1	2	3	4	.	.	.	.	?	
2	Δ_2	.	.	.	.	5	6	7	8	✘	
3	Δ_1	.	.	.	.	5	6	.	.	✔	
4	Δ_2	.	.	.	.	.	.	7	8	✘	
5	Δ_1	.	.	.	.	.	.	7	.	✘	Done
Result		.	.	.	.	.	.	7	.		

Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. IEEE Transactions on Software Engineering 28(2), February 2002, pp. 183-200.