Debugging

SWE 795, Spring 2017

Software Engineering Environments

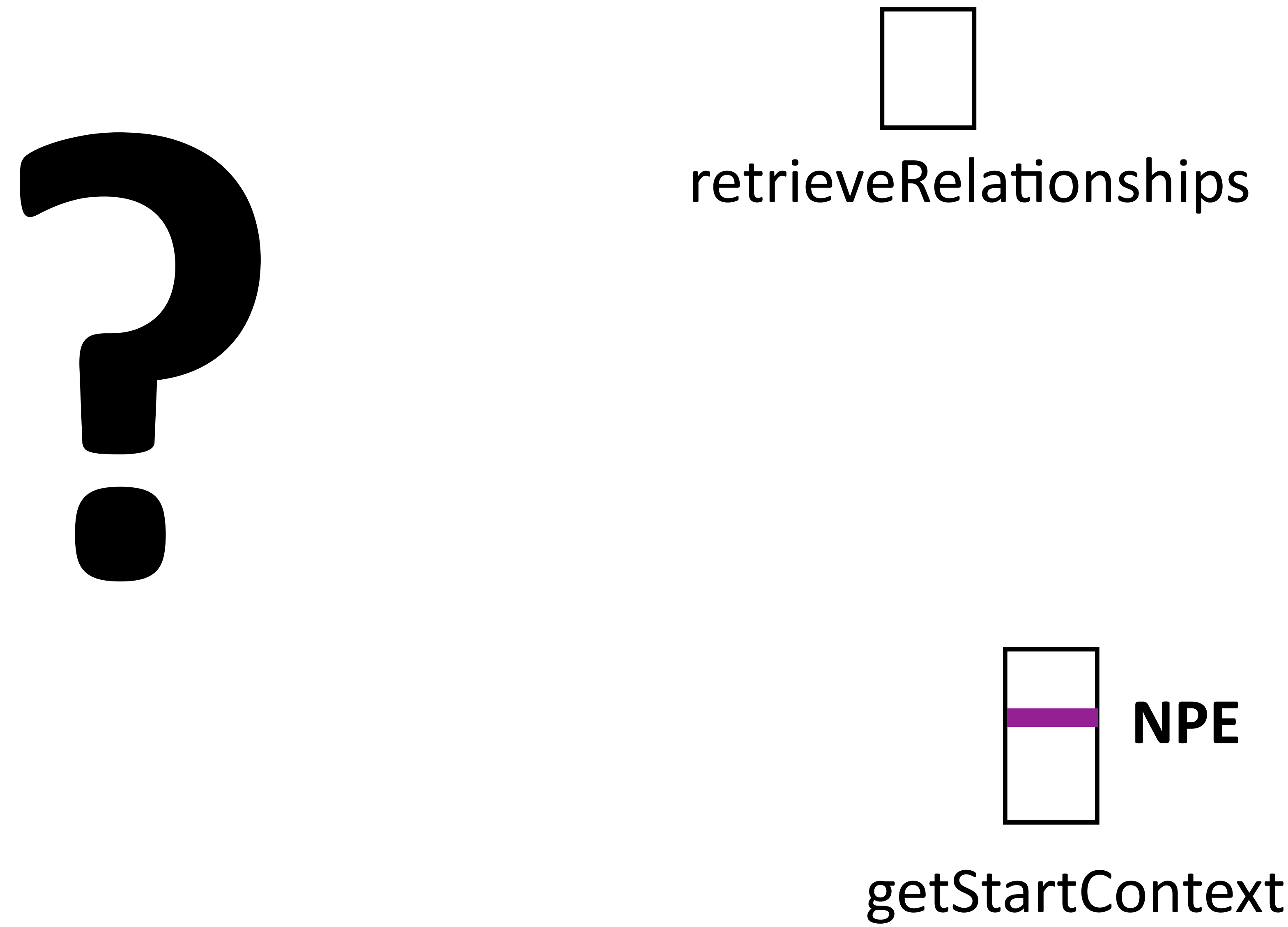
GEORGE MASON UNIVERSITY

Today

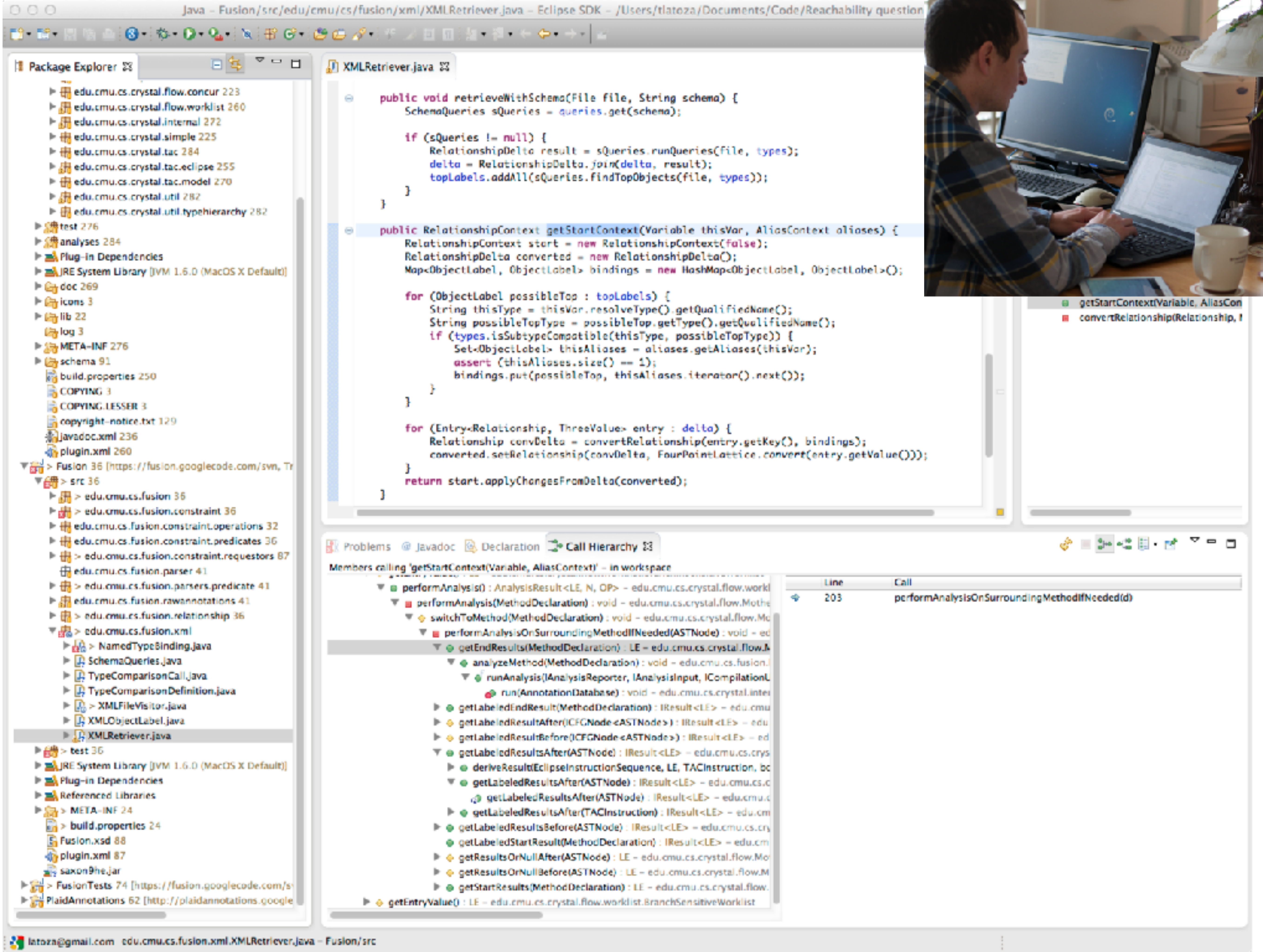
- Part 1 (Lecture)(~45 mins)
  - Debugging
- Part 2 (HW1 Presentations)(30 mins)
- Break!

- Part 3 (Discussion)(60 mins)
  - Discussion of readings

Example



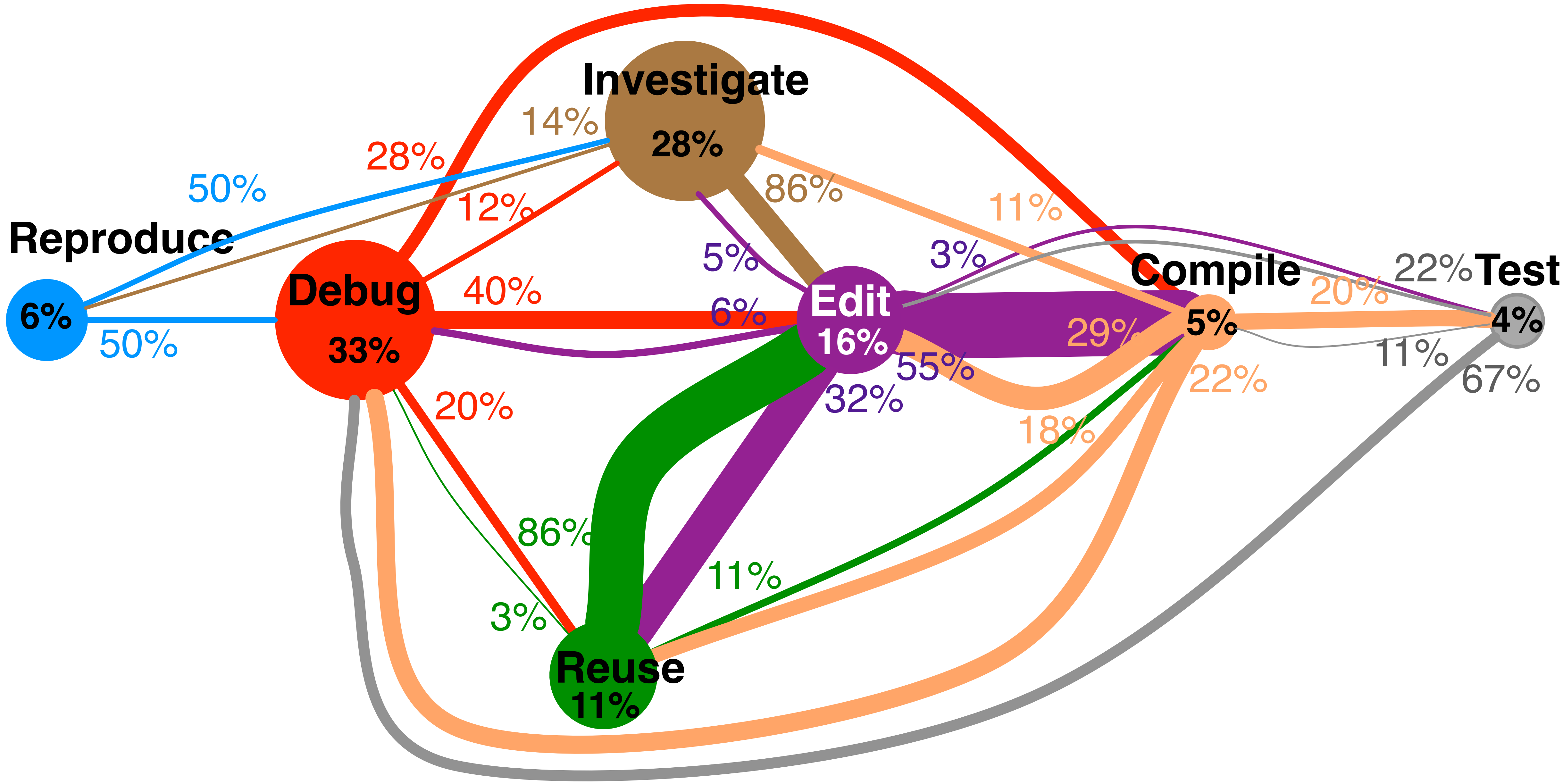
LaToza and Myers. Developers ask reachability questions. ICSE 2010.


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 - ed
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.cr
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

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



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.

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)

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

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 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
  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.

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

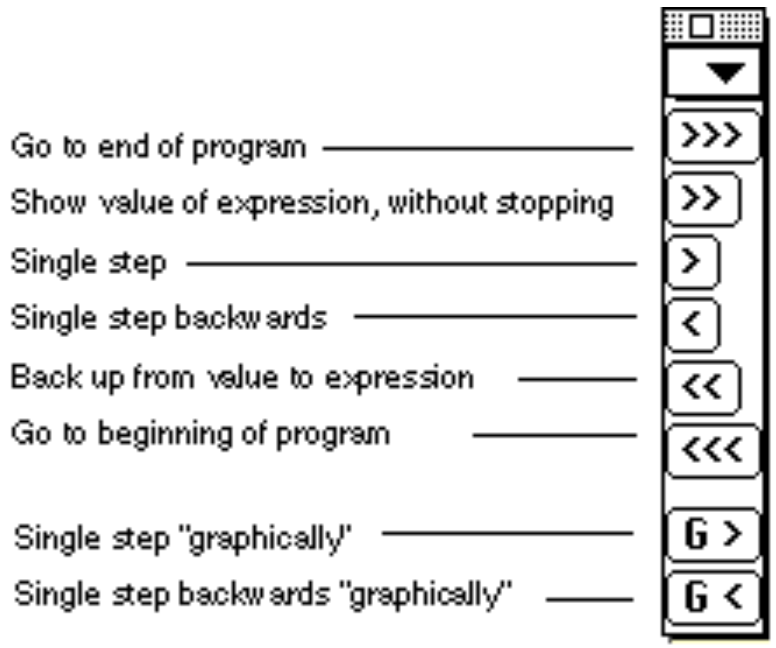
- What if developers could debug forwards **and** backwards?

Record & replay execution

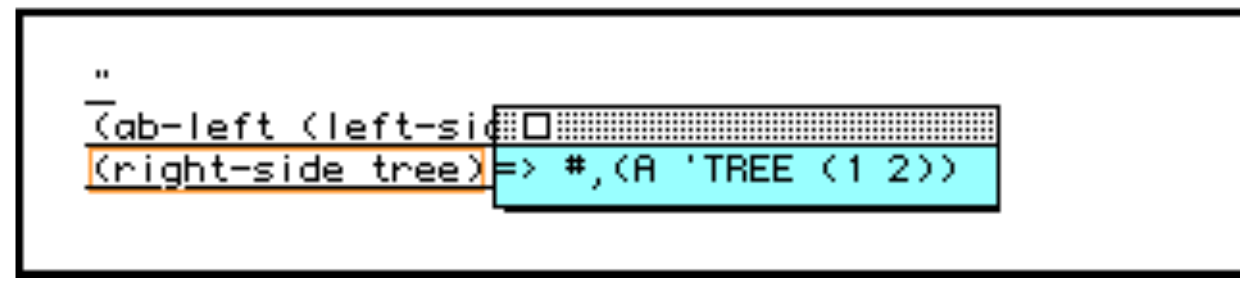
- Record execution, step backwards / forwards through execution
  Biggest challenge - performance slowdown from logging - focus of most papers
- Example systems focused on user interactions
  - Retrace - on exception, backup several statements & start logging
    M. V. Zelkowitz. 1973. Reversible execution. *Commun. ACM* 16, 9 (September 1973), 566.
  - ZStep94 - backwards / forwards stepping, find code which rendered graphics
    Henry Lieberman and Christopher Fry. 1995. Bridging the gulf between code and behavior in programming. In Proceedings of the SIG
    conference on Human factors in computing systems (CHI '95), 480-486.
  - Omniscient debugging - backwards / forwards stepping, step through writes to a variable
    Bill Lewis. Debugging backwards in time. In Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003), October 2003.
  - WhyLine - ask questions about output, traverse dynamic control & data dependencies, ask why didn't questions
    Andrew J. Ko and Brad A. Myers. 2010. Extracting and answering why and why not questions about Java program output. ACM Trans. Softw. Eng. Methodol. 20, 2, Article 4 (September 2010), 36 pages.

ZStep94

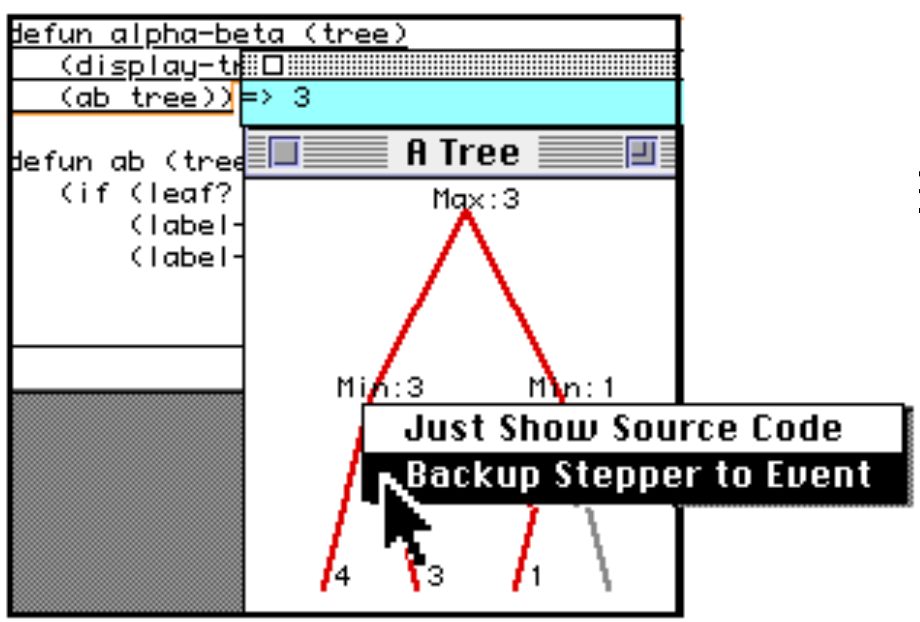
- Forwards / backwards stepping through execution events



See value of selected variables



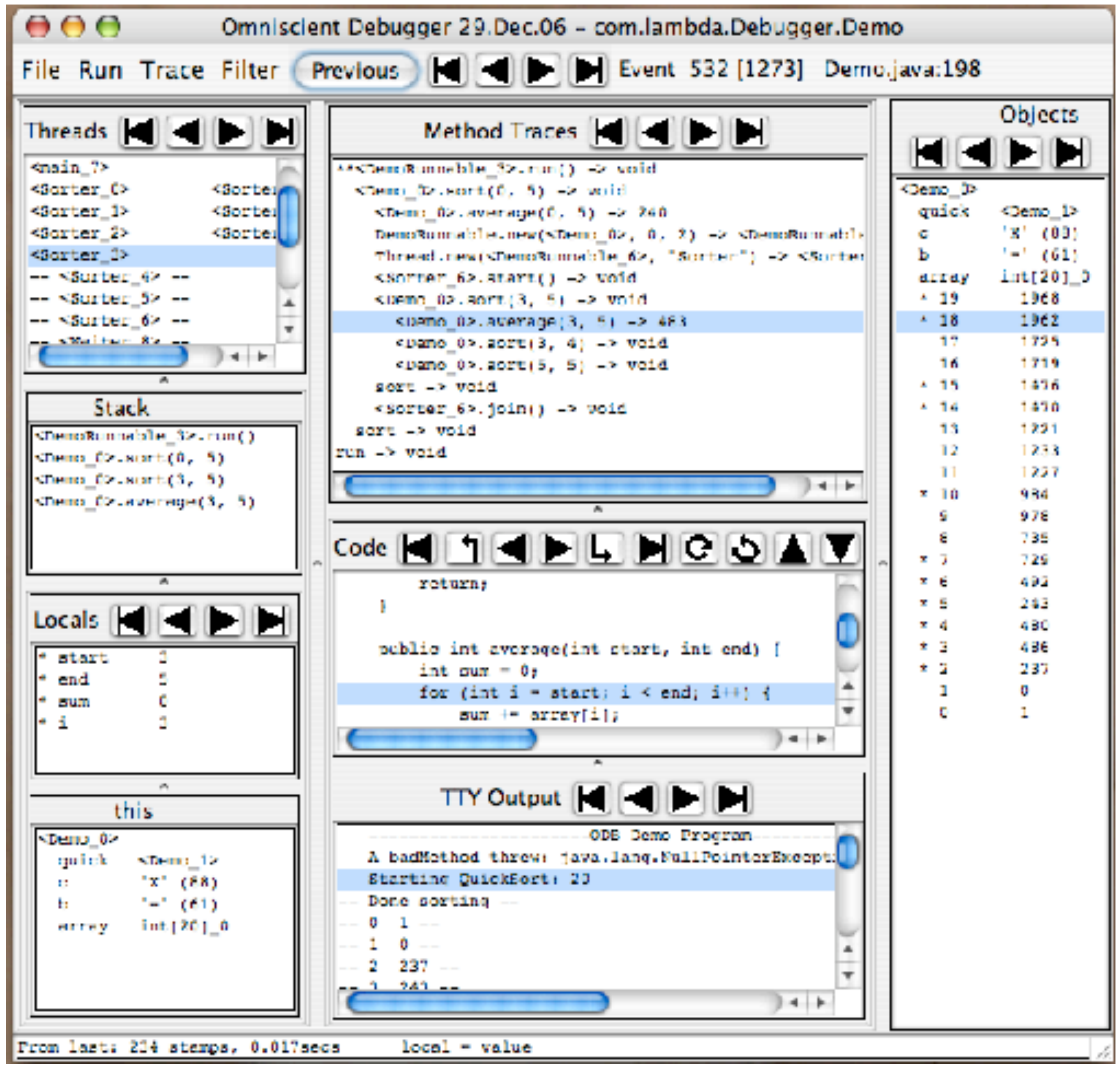
- Select g ... that drew it



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.

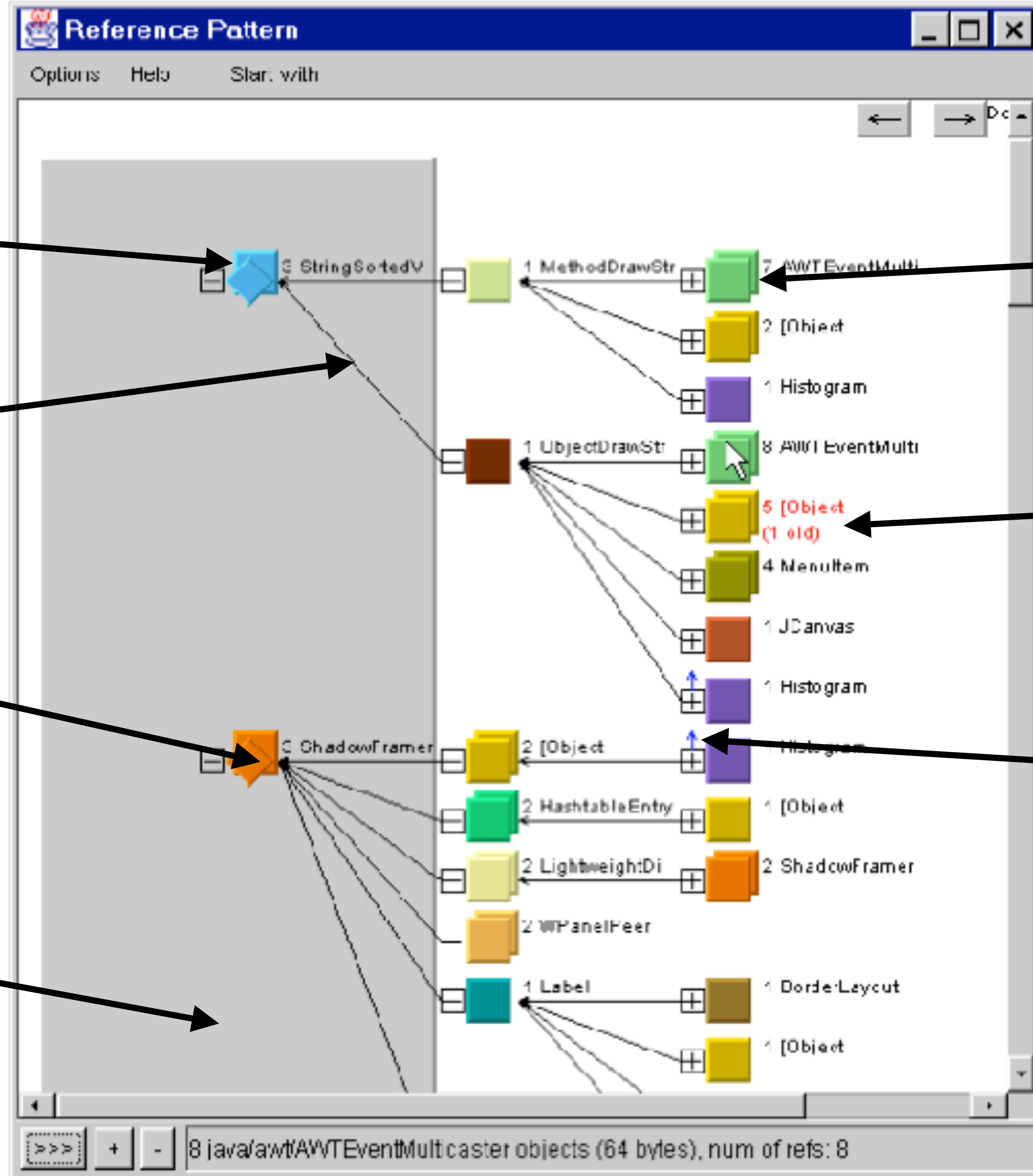
Find temporary objects that aren't garbage collected

Objects of a specific class with same reference tree

Reference

Class object (static members)

Starting set (can be objects created after temporary event that didn't collected)



Multiple instances

Old object (created before temporary event)

Class occurs more than once in diagram

Wim De Pauw and Gary Sevitsky. Visualizing Reference Patterns for Solving Memory Leaks in Java. In European Conference on Object-Oriented Programming (ECOOP), pages 116–134,1999.

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)

```
<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

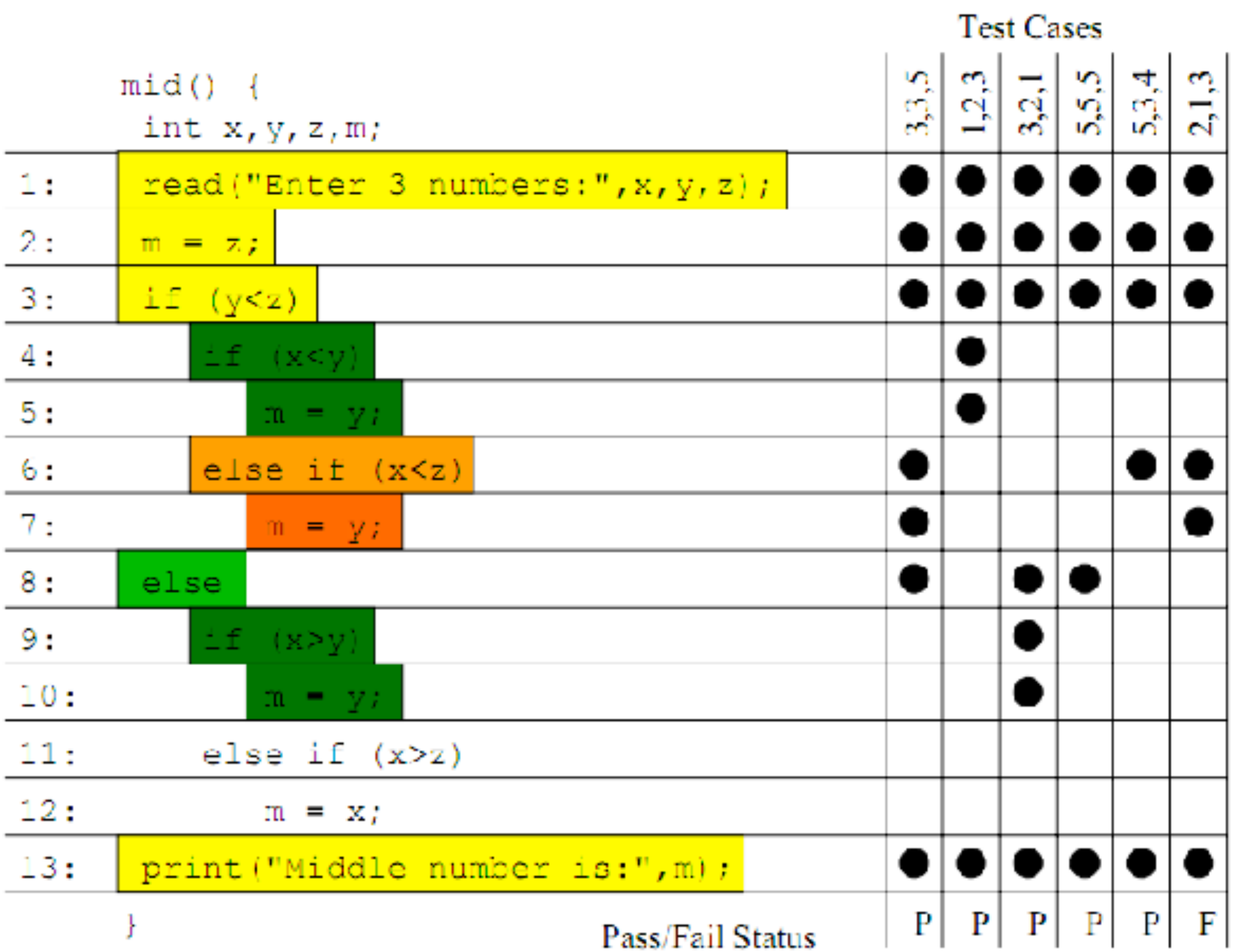
- ddmin algorithm sketch:
- 1. Decompose input into pieces
  2. Run tests on pieces
  3. If there's a piece that still fails, go back to 1 on piece
     Otherwise, found locally minimal smallest input

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	.		

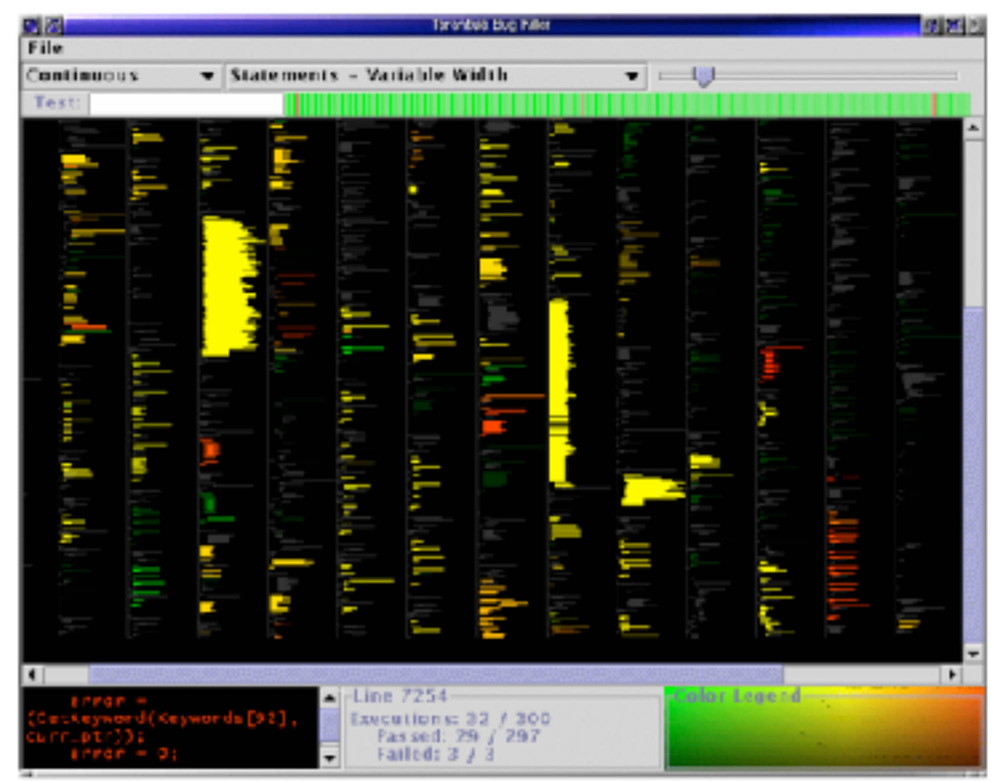
Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. IEEE Transactions on Software Engineering 28(2), February 2002, pp. 183-200.

Compare faulty & unfaulty execution traces

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

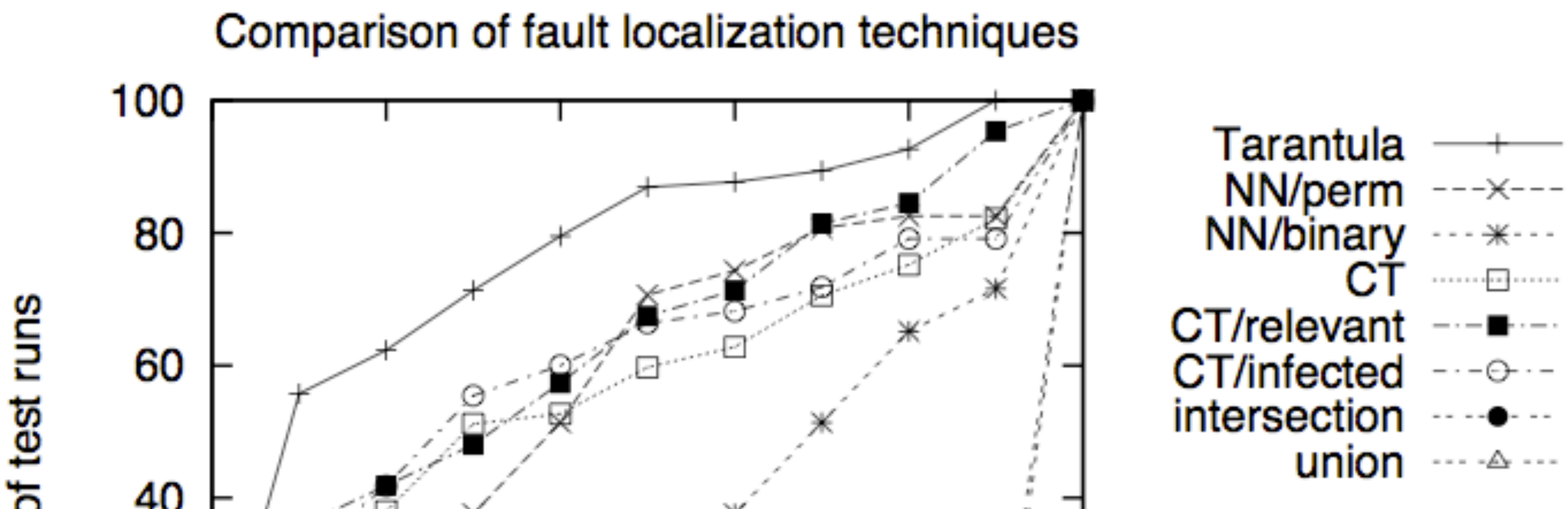


$$suspiciousness(e) = 1 - hue(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}}$$



James A. Jones and Mary Jean Harrold. 2005. [Empirical evaluation of the tarantula automatic fault-localization technique](). In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05). ACM, New York, NY, USA 273-282.

Compare faulty & unfaulty execution traces



Siemens suite of fault localization programs

Program	Faulty Versions	Procedures	LOC	Test Cases	Description
print_tokens	7	20	472	4056	lexical analyzer
print_tokens2	10	21	399	4071	lexical analyzer
replace	32	21	512	5542	pattern replacement
schedule	9	18	292	2650	priority scheduler
schedule2	10	16	301	2680	priority scheduler
tcas	41	8	141	1578	altitude separation
tot_info	23	16	440	1054	information measure

- Tarantula - frequency of failing runs relative to passing runs ("suspicousness")
  Union: (U passing_tests) - 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

James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05). ACM, New York, NY, USA 273-282.

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)

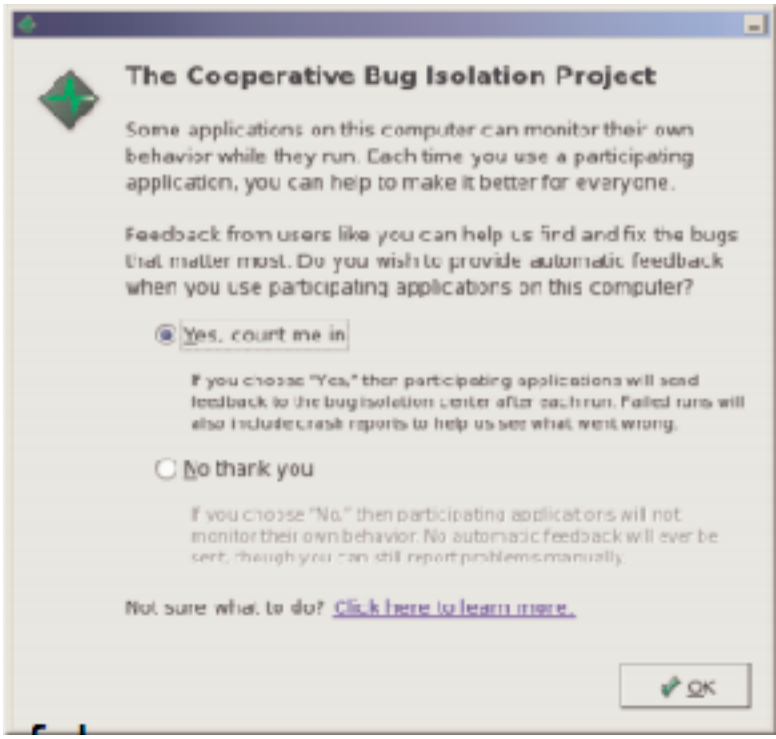
- Stack trace not good bug predictor for some systems (e.g. event based systems)
  ⇒ bugs may be in multiple buckets or multiple bugs in single bucket

- 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.

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

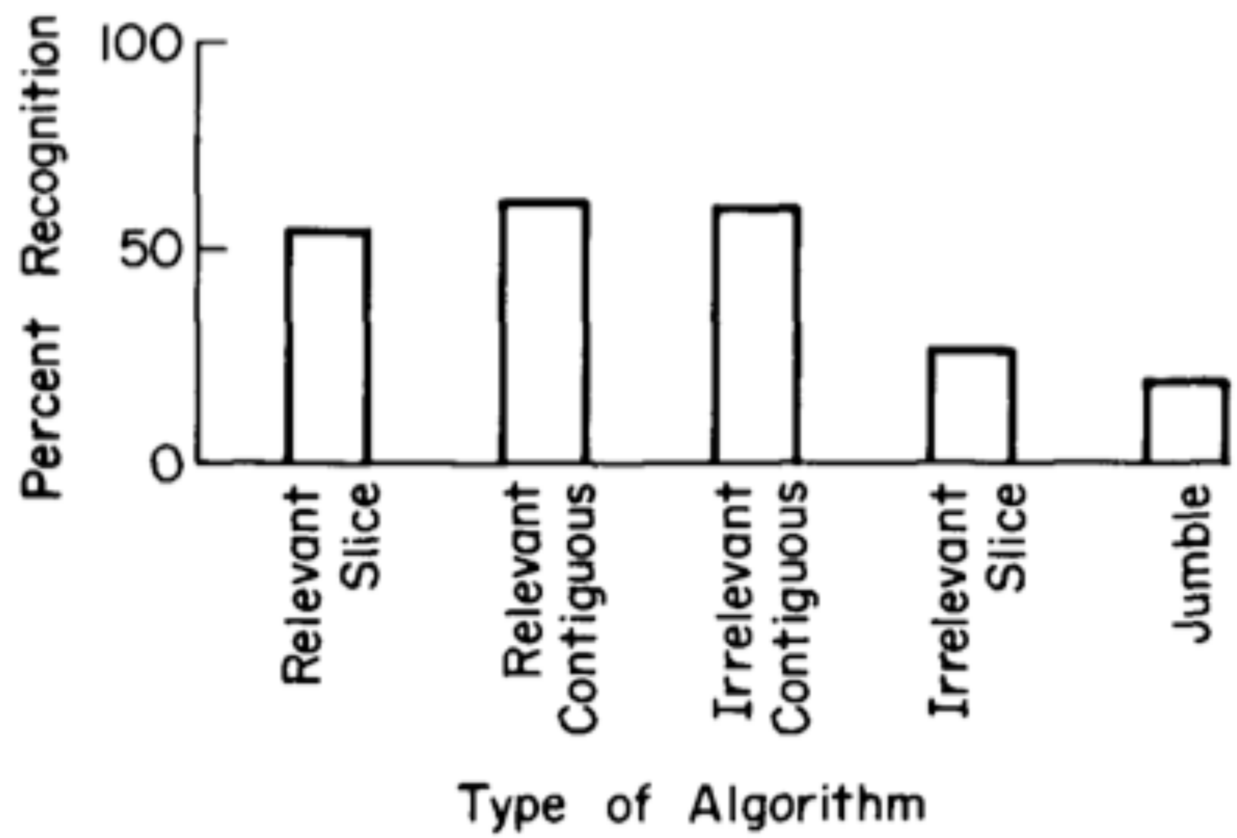
- Originally formulated as **subset** of program
- Later formulations emphasize ability to **traverse** control & data dependencies (e.g., WhyLine)

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 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.

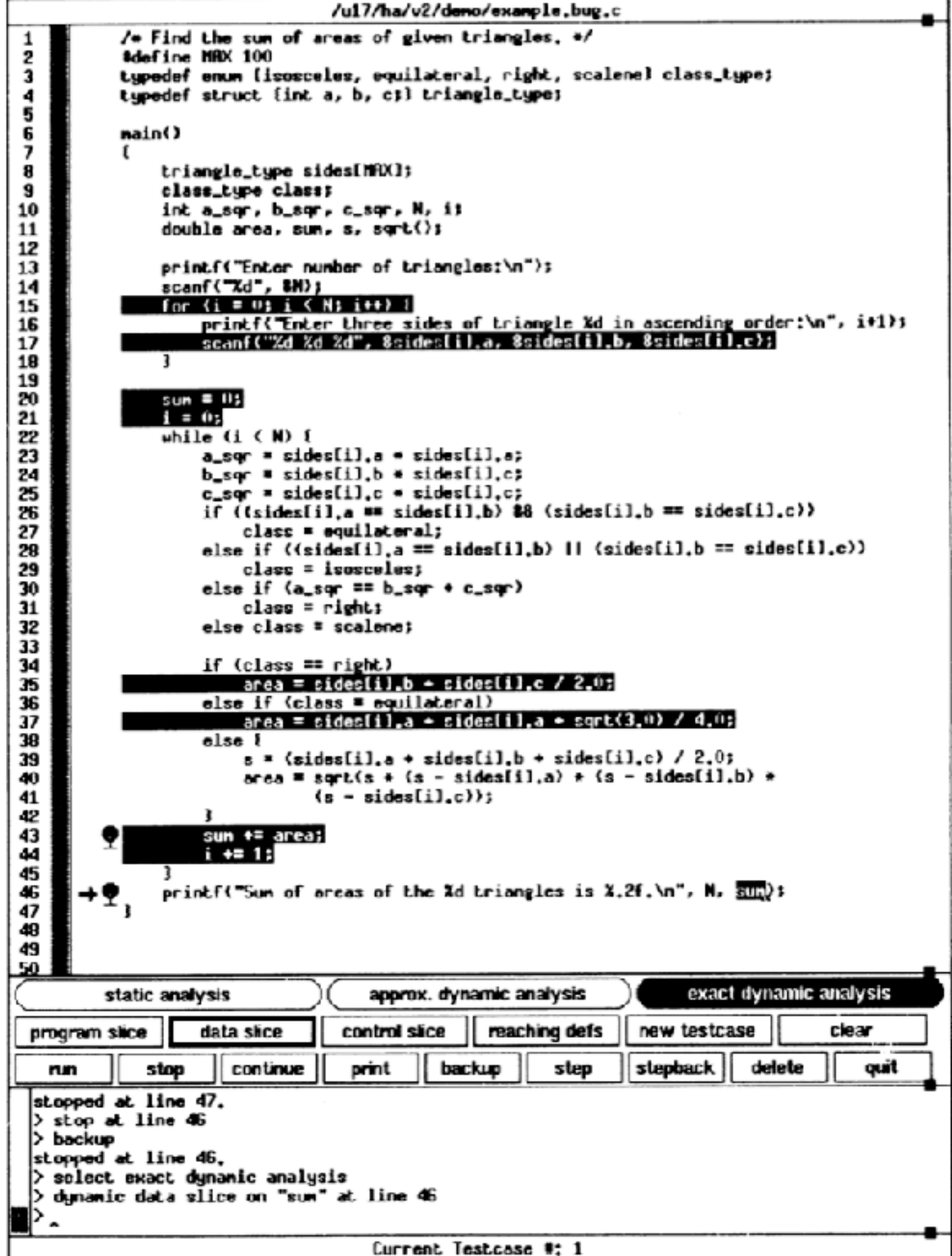
Slicers debug faster

- 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



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.