Debugging

CS 695 / SWE 699: Programming Tools

Fall 2023

Today

- Part 1 Lecture(~45 mins)
  - 10 min break
- Part 2: Tech Talks (30 mins)
  - Two tech talks
- Part 3: In-Class Activity(1 hour)

Logistics

- HW 4 due today
- HW 4 presentations on 12/6 during final exam period

Overview

- Process and Challenges of Debugging
- Types of Debugging Tools

Steps in fixing bugs

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

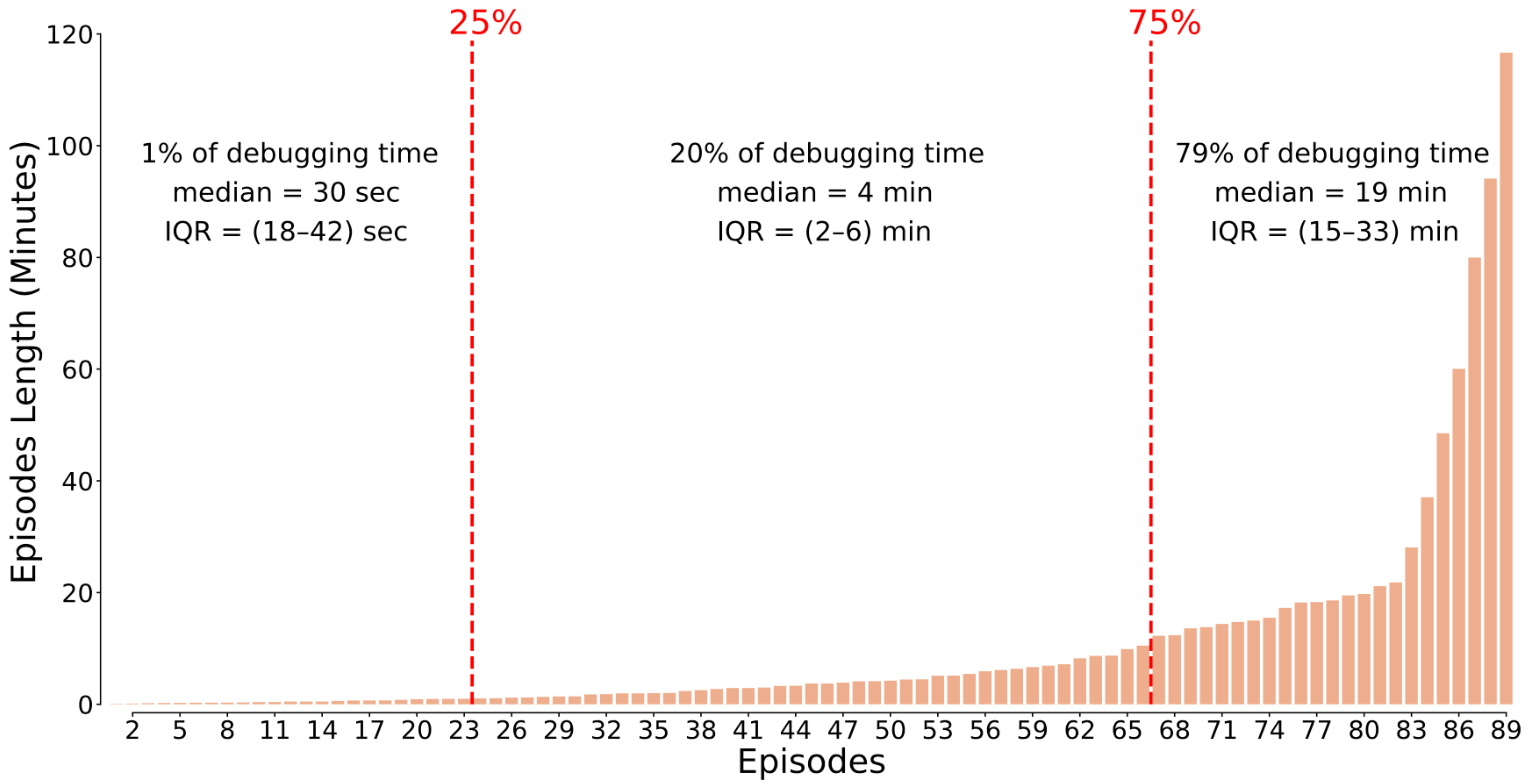


Fig. 2 Debugging episode length, from shortest to longest

Alaboudi, A., LaToza, T.D. What constitutes debugging? An exploratory study of debugging episodes. *Empir Software Eng* **28**, 117 (2023). https://doi.org/10.1007/s10664-023-10352-5

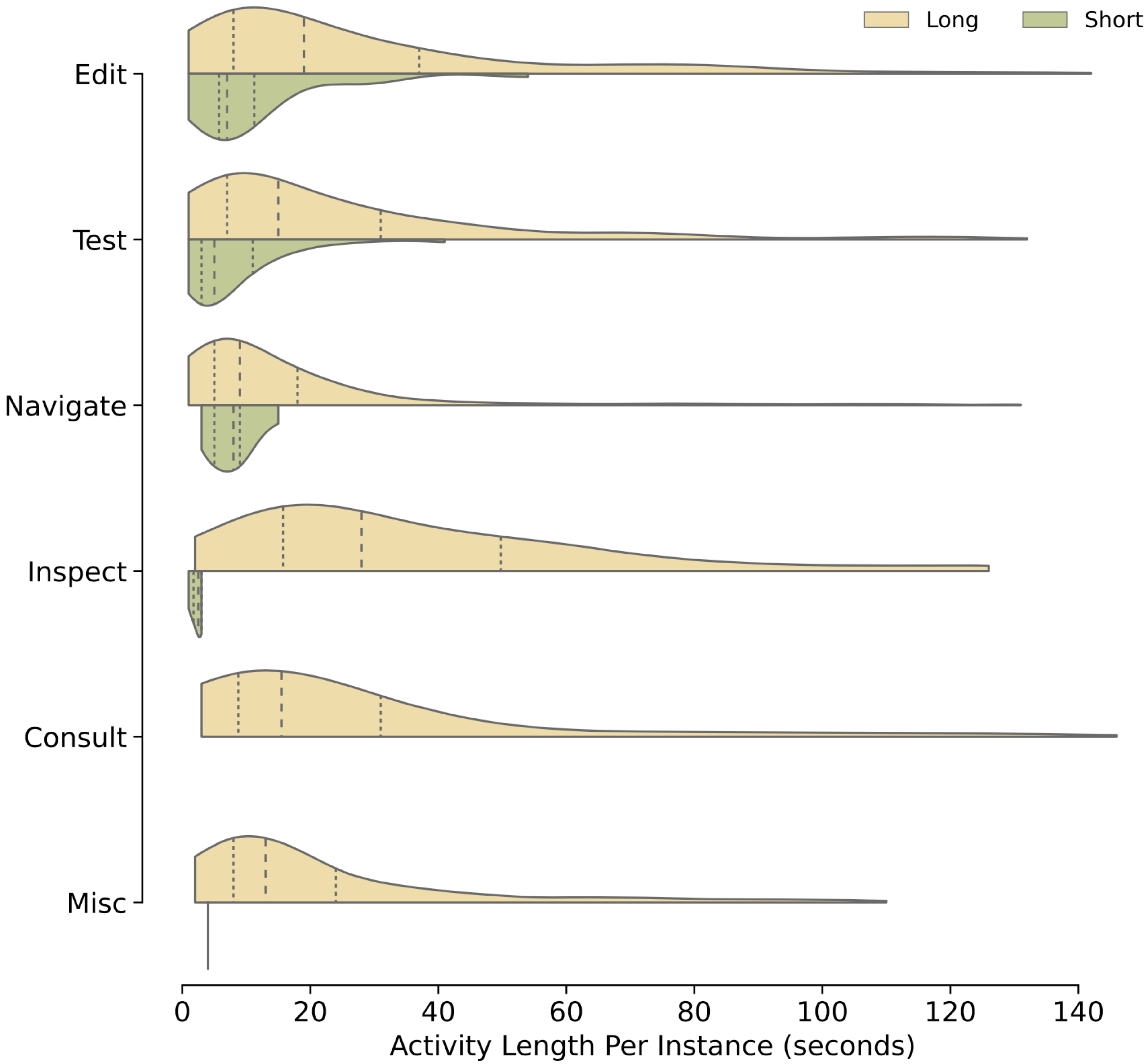


Fig. 3 The distribution of the time developers spent on each activity instance in long and short debugging episodes

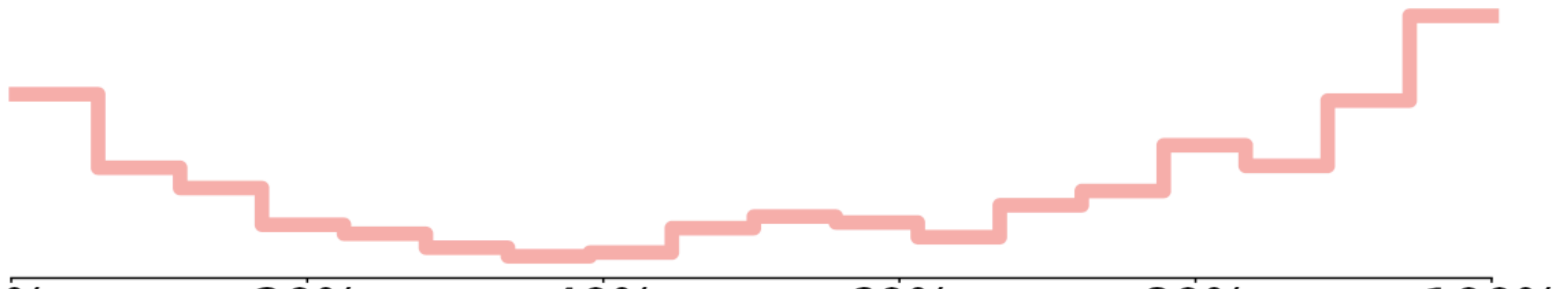
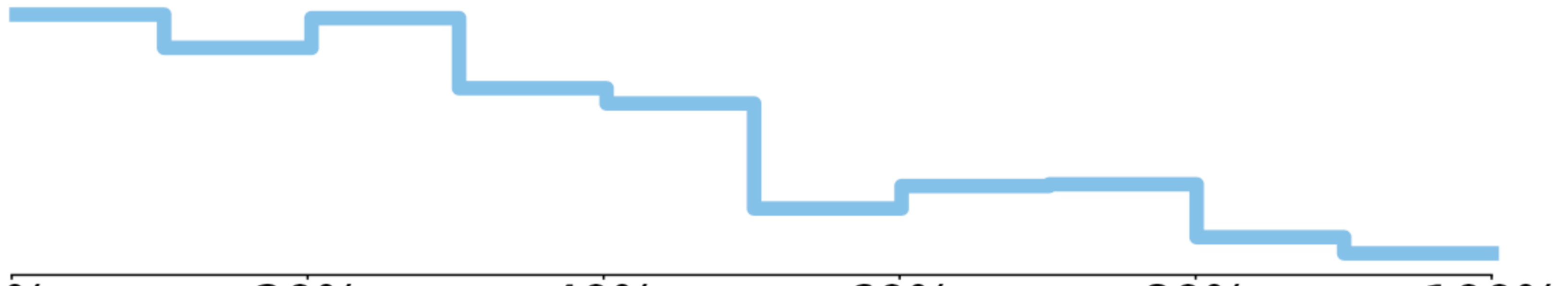
Alaboudi, A., LaToza, T.D. What constitutes debugging? An exploratory study of debugging episodes. *Empir Software Eng* **28**, 117 (2023). https://doi.org/10.1007/s10664-023-10352-5

Table 7 The distribution of debugging activities per episode. % of episode time is the fraction of time of the episodes that the activity occupied

Activities	Instances Per Episode			% of Episode Time		
	Median (IQR)	Min	Max	Median (IQR)	Min	Max
Edit	3 (1-9)	0	67	41% (26-54%)	7%	97%
Test	3 (2-7)	0	32	29% (18-43%)	2%	100%
Navigate	3 (0-6)	0	109	15% (9-22%)	1%	50%
Inspect	0 (0-1)	0	26	14% (8-29%)	1%	58%
Consult	0 (0-1)	0	16	9% (4-18%)	0.4%	59%
Miscellaneous	0 (0-1)	0	35	4% (2-9%)	1%	26%

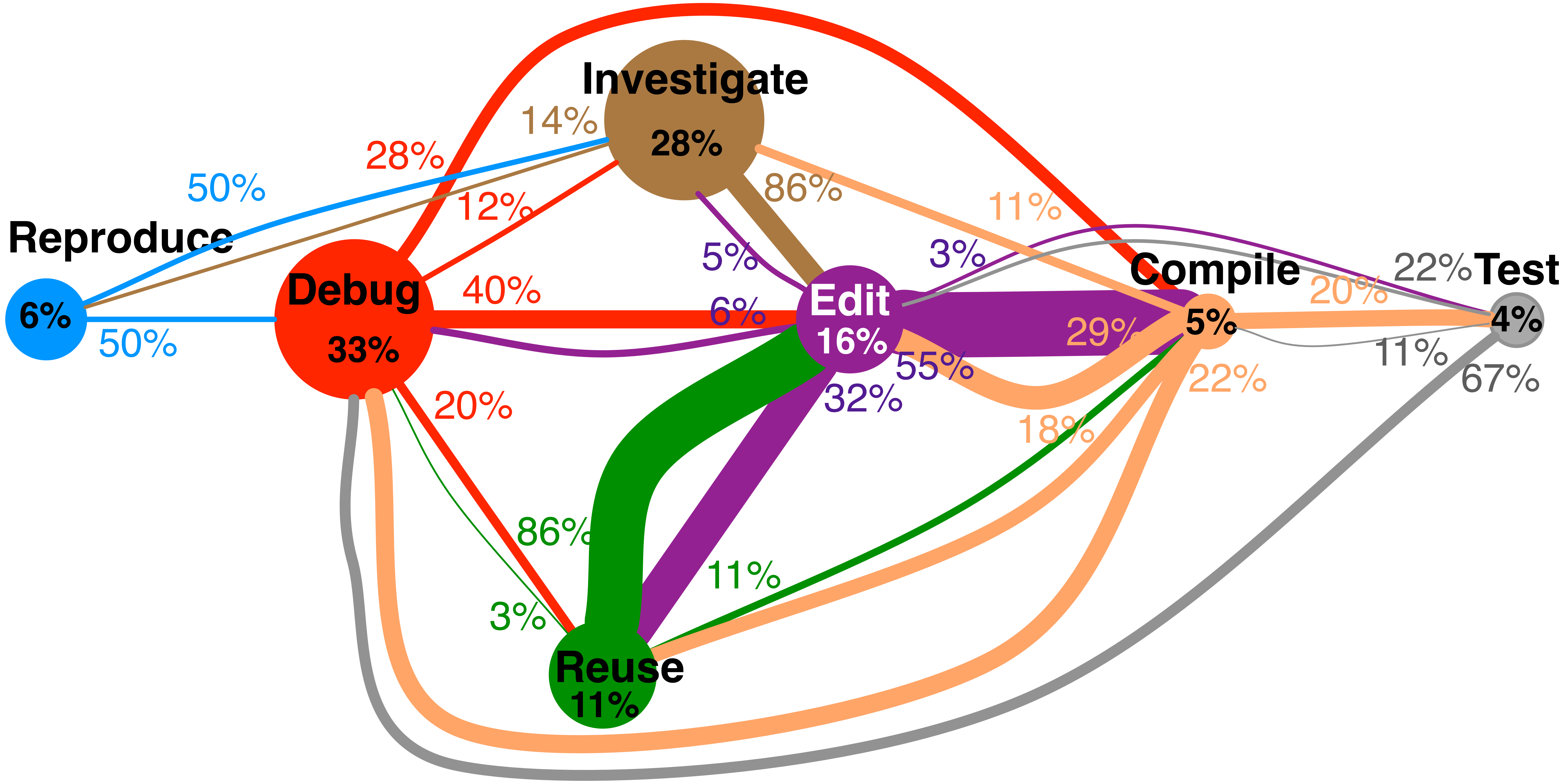
Alaboudi, A., LaToza, T.D. What constitutes debugging? An exploratory study of debugging episodes. *Empir Software Eng* **28**, 117 (2023). https://doi.org/10.1007/s10664-023-10352-5

Table 8 The distribution of frequency (count) of activities throughout the debugging episodes

Activities	Distribution of Frequency Across Episodes Time
Edit	
Test	
Navigate	
Inspect	
Consult	
Miscellaneous	

Alaboudi, A., LaToza, T.D. What constitutes debugging? An exploratory study of debugging episodes. *Empir Software Eng* **28**, 117 (2023). https://doi.org/10.1007/s10664-023-10352-5

Edit / Debug Cycle

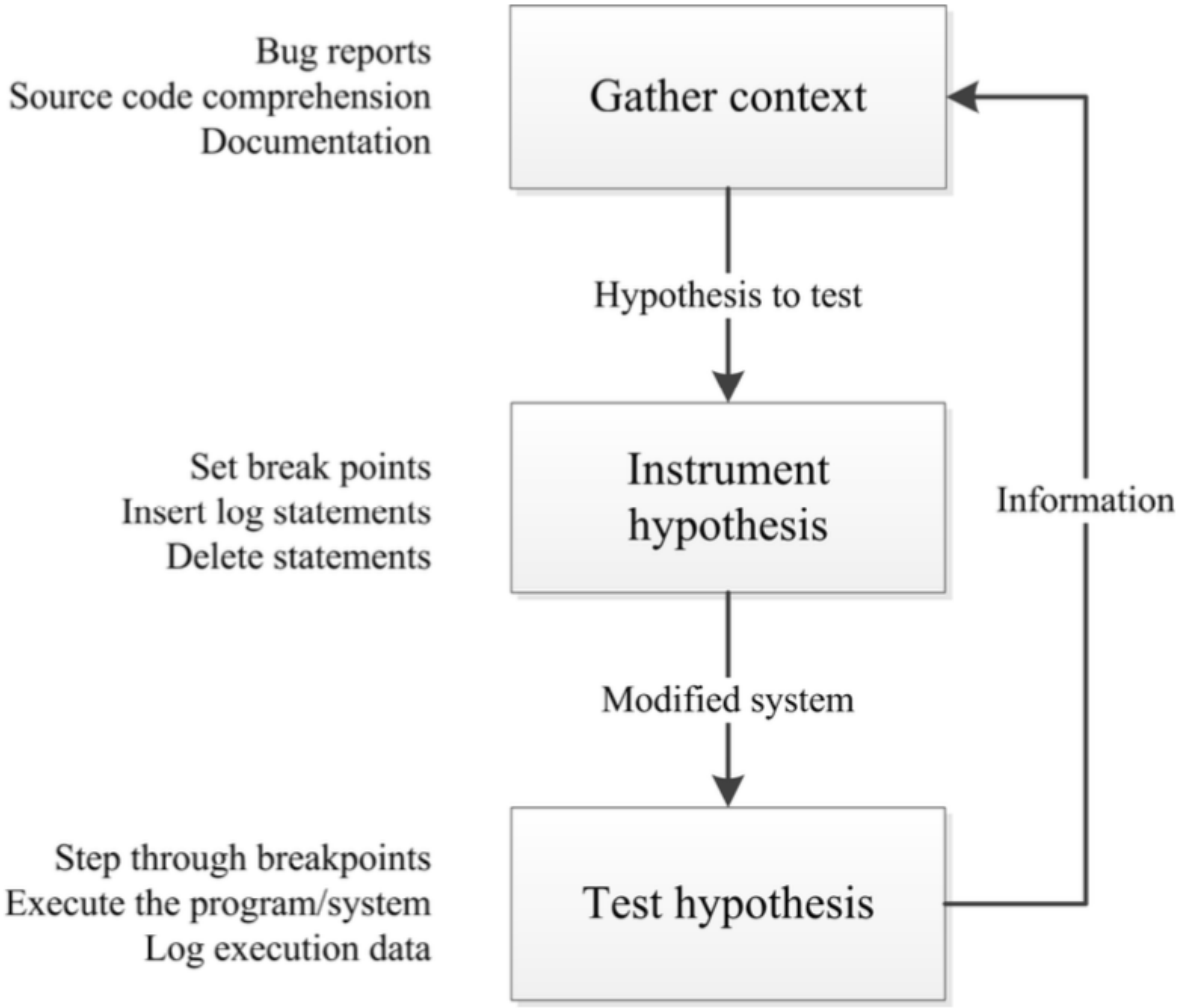


Circle size: % of time **Edge thickness:** % of transitions observed

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.

Debugging hypotheses matter

- Developers with a correct hypothesis early in the debugging process
  - Spent **30% less time** fixing the fault
  - **>5x** more likely to succeed
- No evidence industrial programming experience or more knowledge of related technologies associated with better hypotheses performance.
- **No evidence that providing potential fault locations helps debugging.**
- Providing generalized debugging hypotheses
  - **> 16x more likely** to successfully fix a fault

Variables	Odds ratio	SE β	Wald	Sig. (p)
Correct hypothesis	5.28	0.67	2.45	**0.01**
Years of Experience	1.08	0.06	1.36	0.17
Technology knowledge	2.08	0.43	1.66	0.09
Debugging task 2	2.43	0.87	1.02	0.30
Debugging task 3	8.43	0.98	2.15	**0.03**
Fault locations	1.37	0.75	0.42	0.67
Years of Experience	1.12	0.06	1.68	0.09
Technology knowledge	2.08	0.46	1.57	0.11
Debugging task 2	2.93	0.98	1.09	0.27
Debugging task 3	12.35	1.05	2.38	**0.01**
Generalized hypotheses	16.33	1.21	2.29	**0.02**
Years of Experience	1.32	0.12	2.20	**0.02**
Technology knowledge	1.25	0.58	0.38	0.69
Debugging task 2	0.15	1.32	-1.42	0.15
Debugging task 3	11.28	1.36	1.78	0.07

A. Alaboudi and T. D. LaToza, "Using Hypotheses as a Debugging Aid," *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Dunedin, New Zealand, 2020, pp. 1-9, doi: 10.1109/VL/HCC50065.2020.9127273.

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.

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.

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.

Information needs in debugging

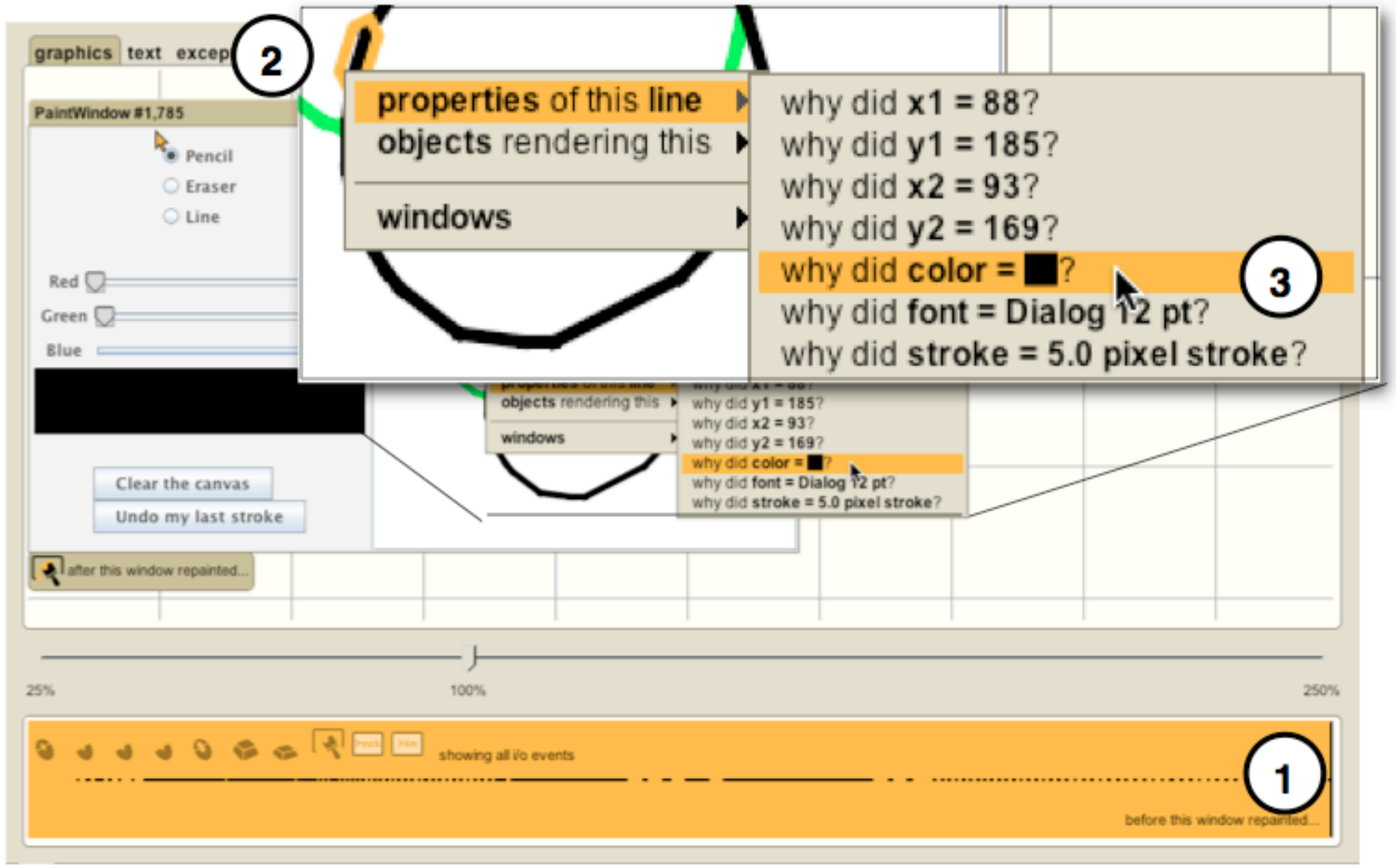
* *How did this **runtime state** occur? (12)*
  data, memory corruption, race conditions, hangs, crashes, failed API calls, test failures, null pointers
* *Where was this **variable** last changed? (1)*
* *Why **didn't** this happen? (3)*

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)*

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

Which team should I assign this bug to?

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

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

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
- **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
- **Hypothesis-based** debugging: identify potentially relevant hypotheses and gather evidence from execution to test

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

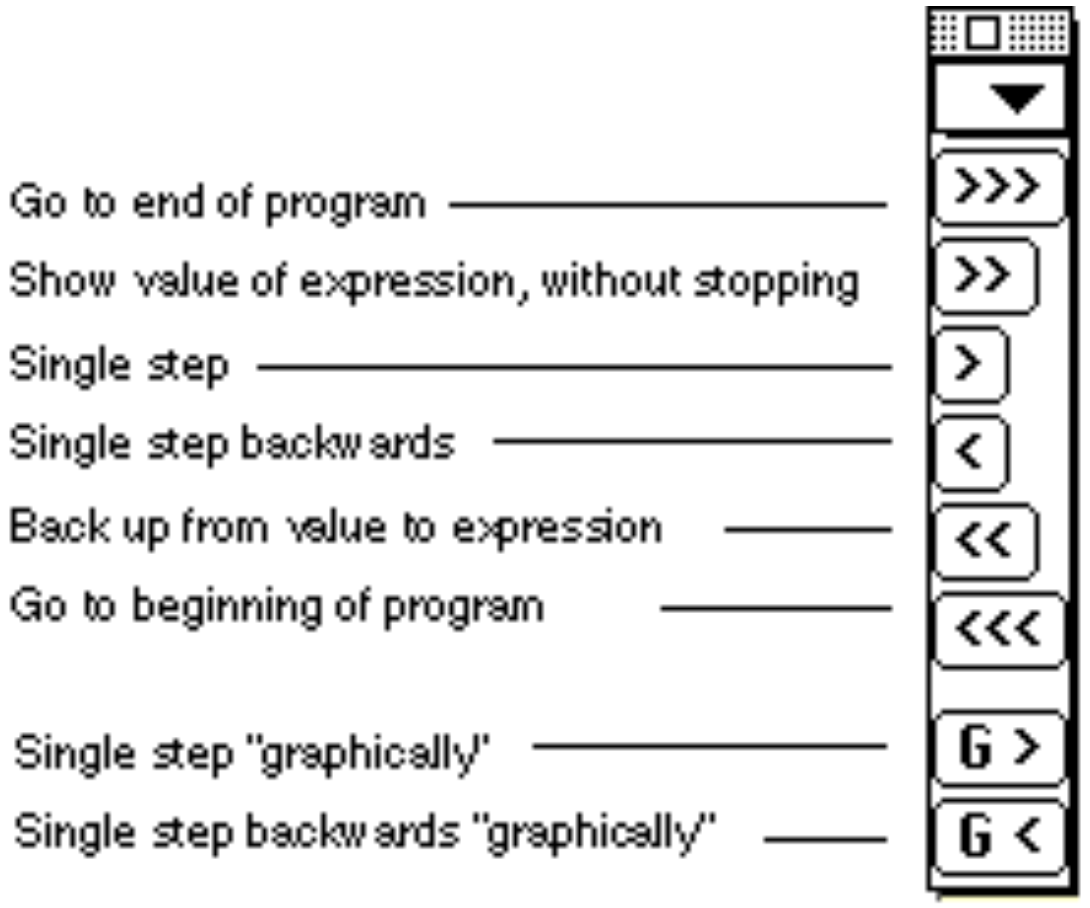
- Debugging in a debugger is hard
  - Forces developer to guess which methods to step into
  - Forces developers to guess which values to instrument
  - Changing guess requires reproing failure again
    - Can be time consuming

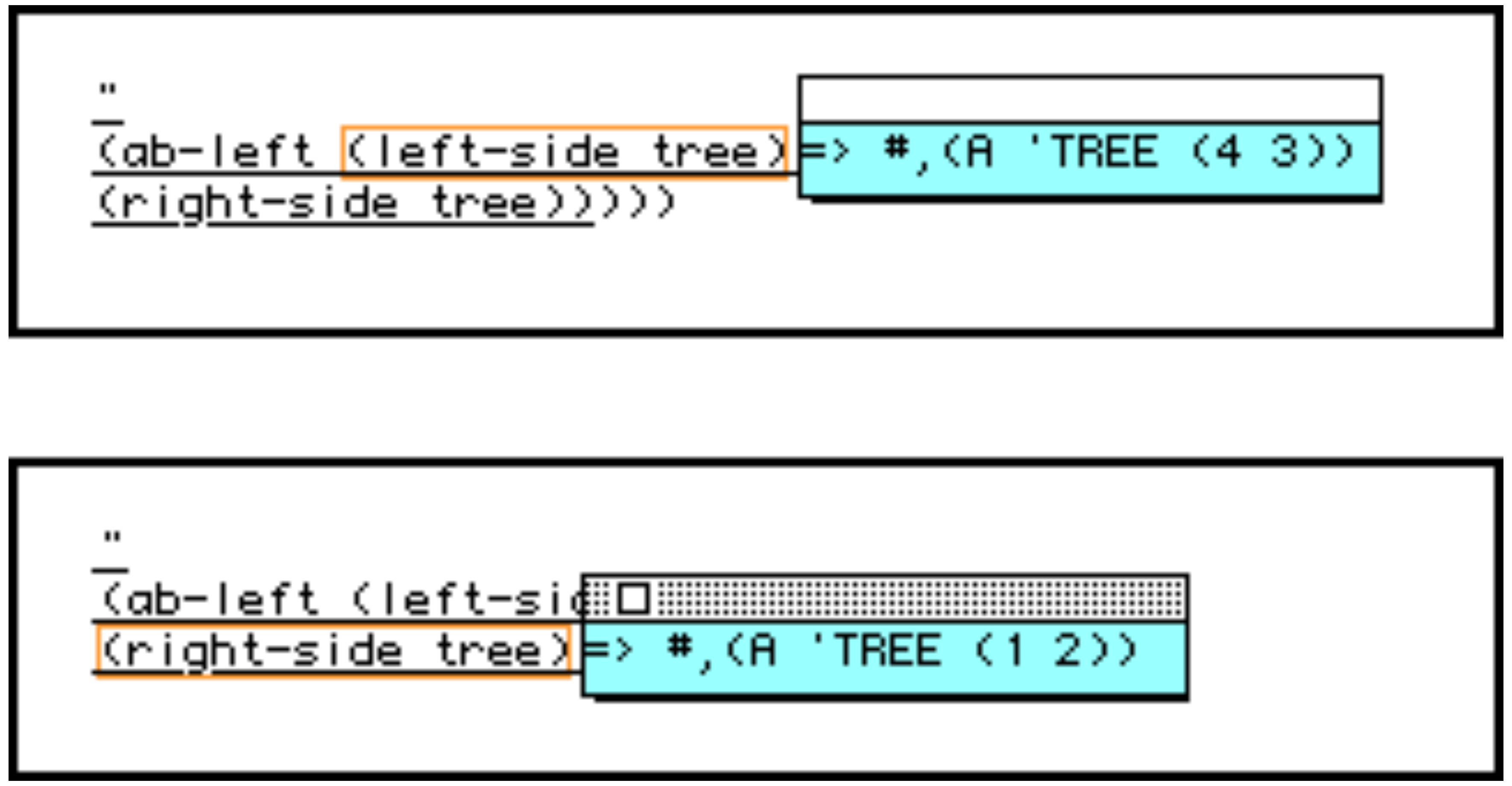
- What if developers could debug forwards **and** backwards?

ZStep94

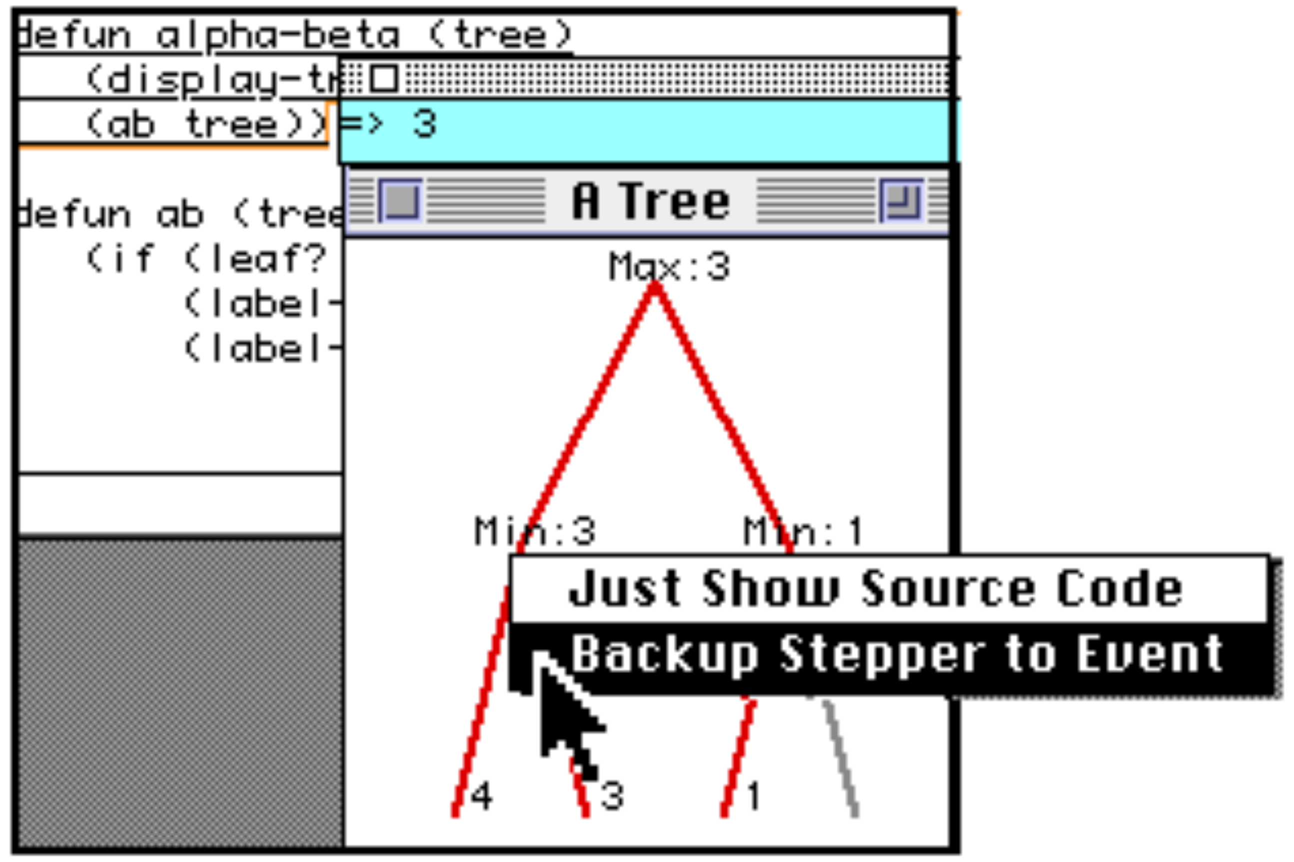
- Forwards / backwards stepping through execution events

See value of selected variables





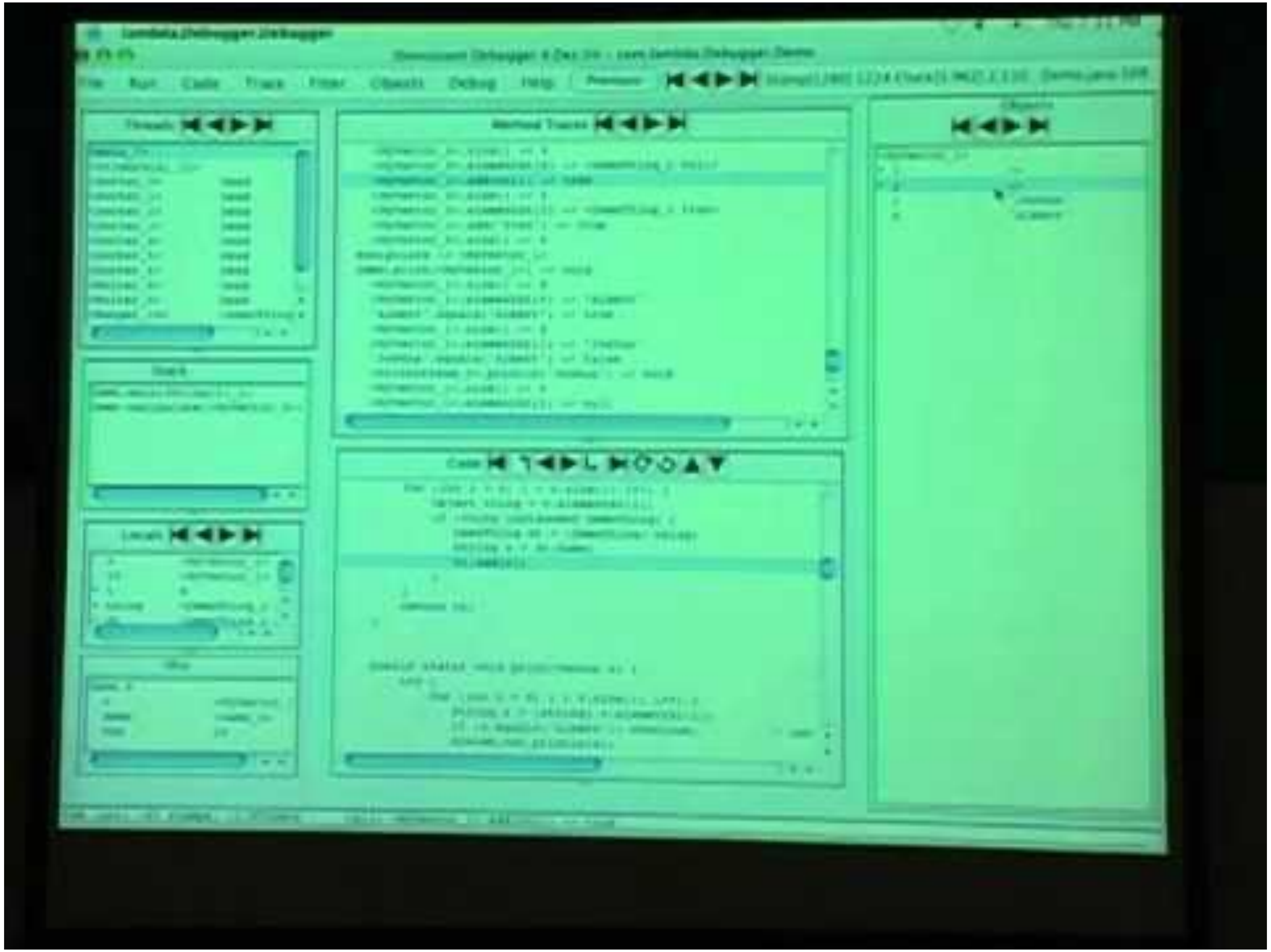
- Select g that drew it
-



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



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

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

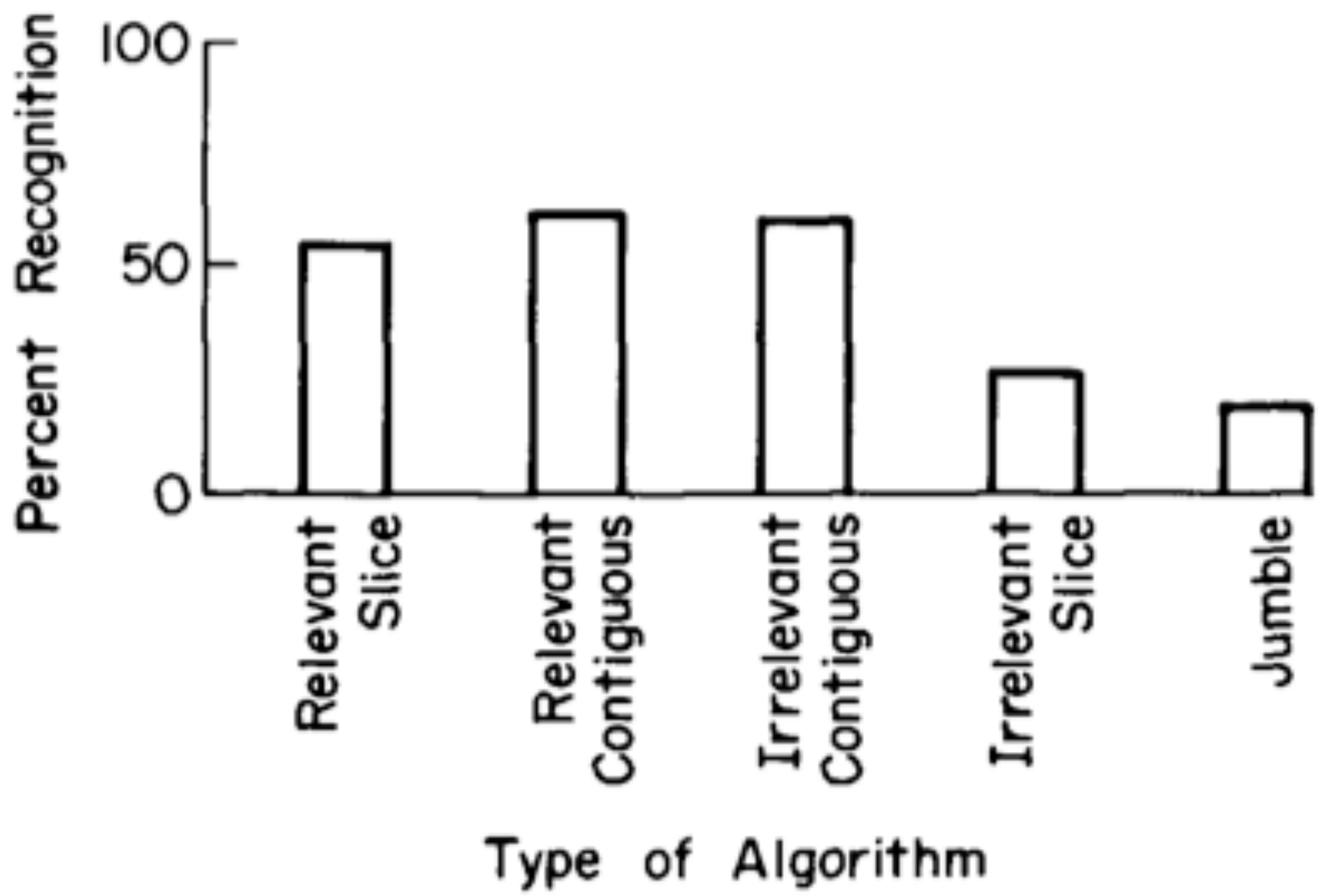
- Originally formulated as **subset** of program

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



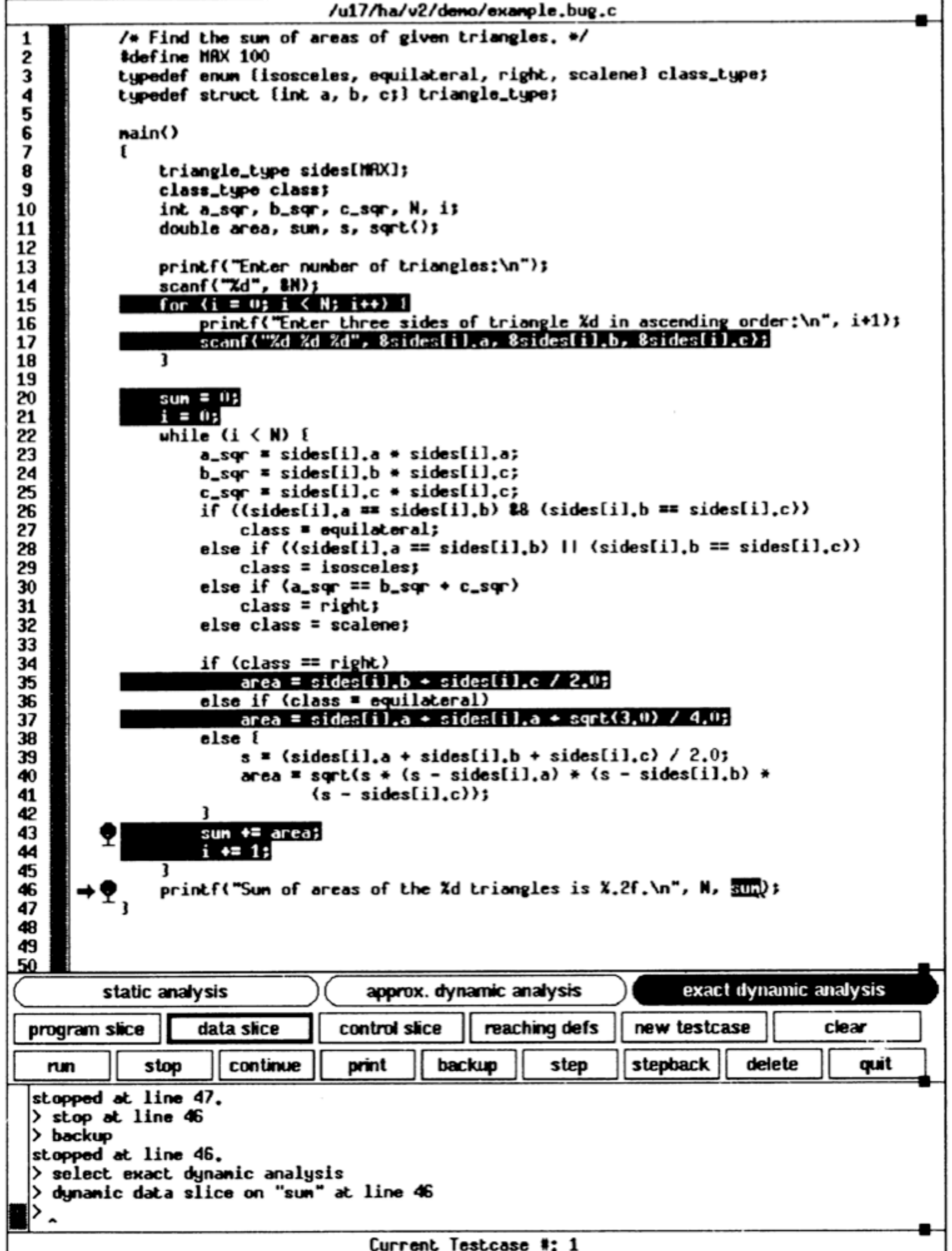
Mark Weiser. 1982. Programmers use slices when debugging. Commun. ACM 25, 7 (July 1982), 446-452.

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)

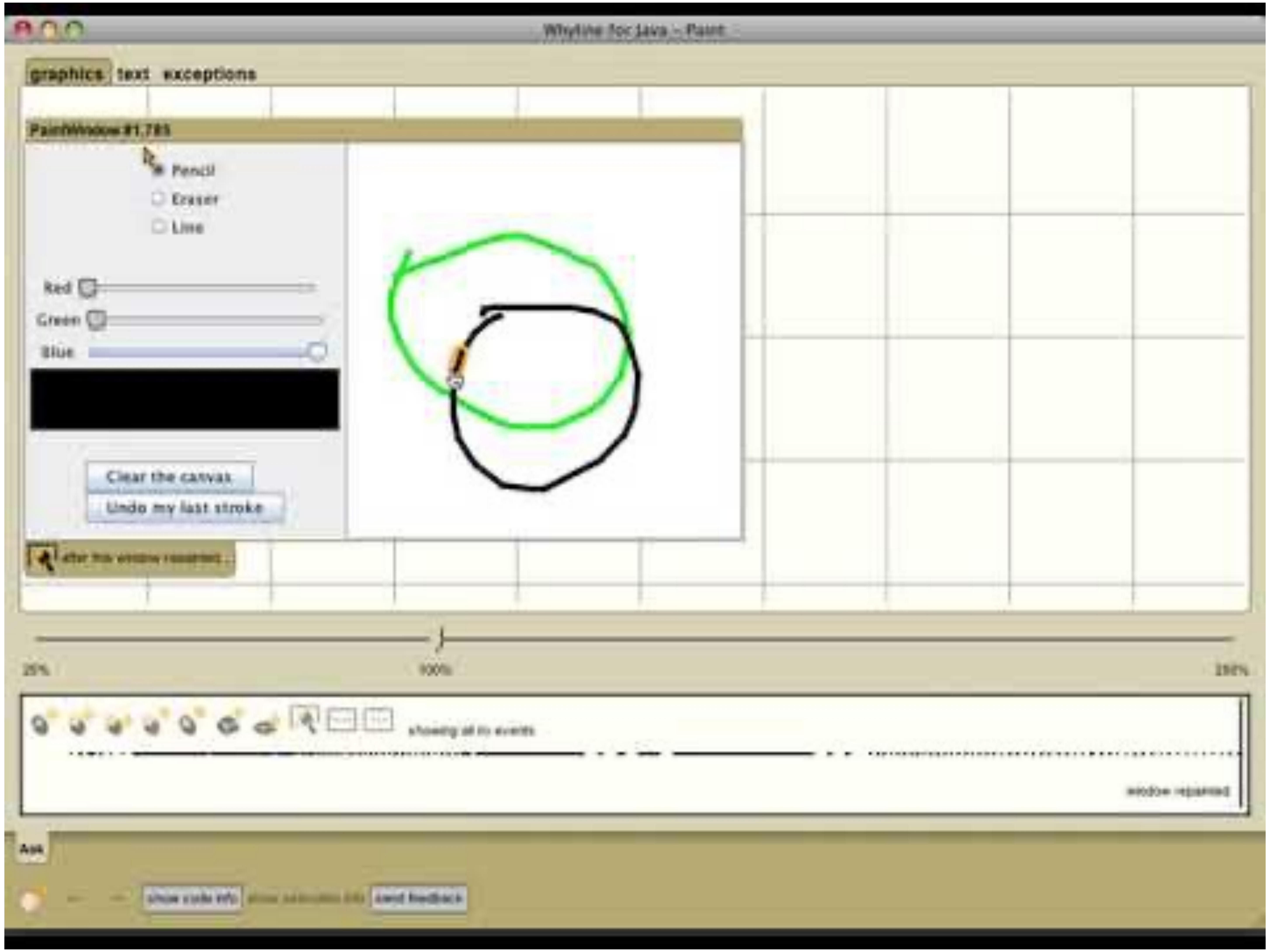
Francel M. A. and S. Rugaber (2001). The Value of Slicing While Debugging. *Science of Computer Programming*, 40(2-3), 151-169.

Dynamic slicing



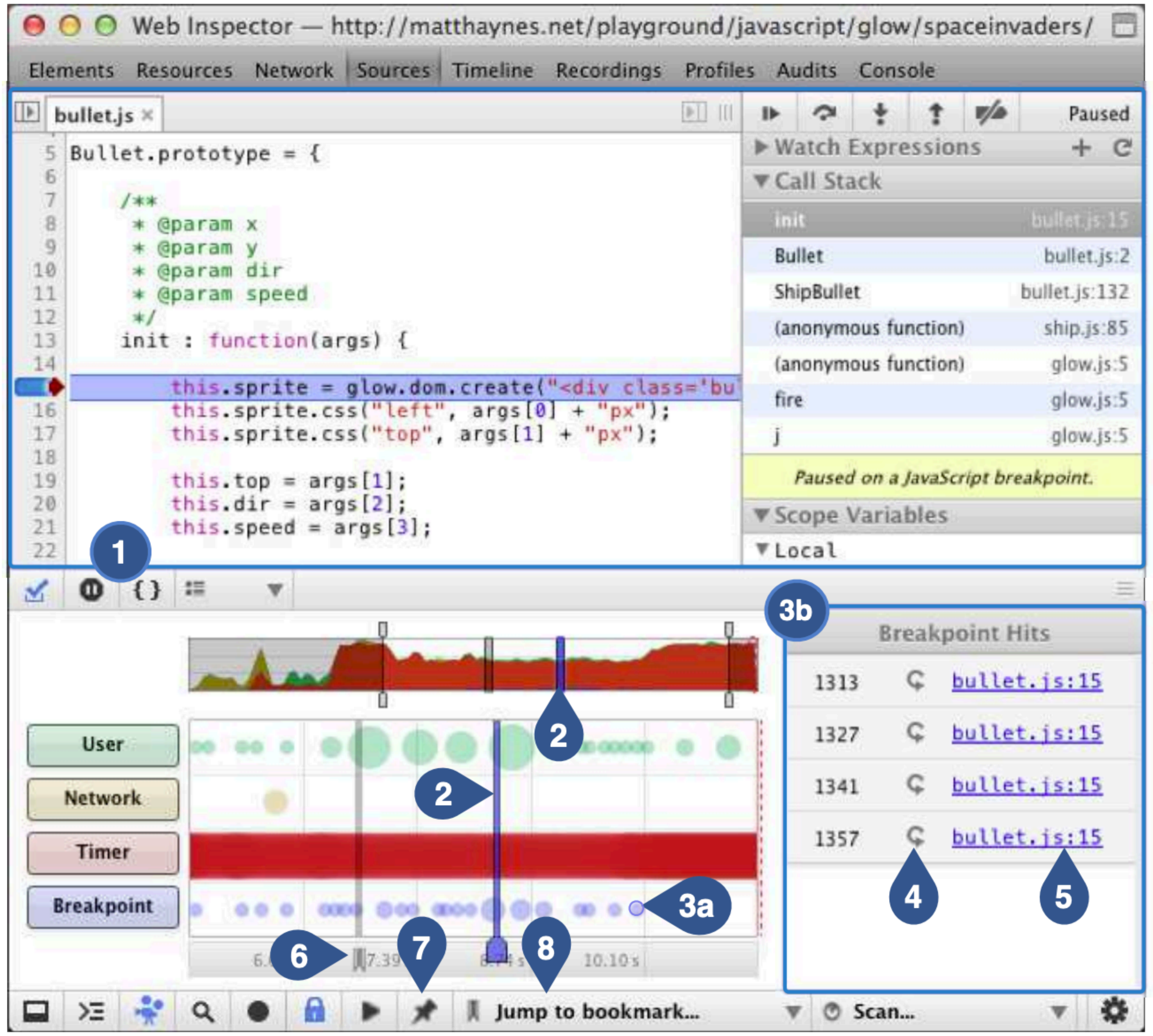
Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. 1993. [Debugging with dynamic slicing and backtracking](). Softw. Pract. Exper. 23, 6 (June 1993), 589-616.

Associating incorrect output with responsible code



Amy J. Ko and Brad A. Myers. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In Proceedings of the 30th international conference on Software engineering (ICSE '08). Association for Computing Machinery, New York, NY, USA, 301–310. https://doi.org/10.1145/1368088.1368130

Record / Replay for Web Apps



Demo: https://dl.acm.org/doi/10.1145/2501988.2502050

BJ Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. 2013. Interactive record/replay for web application debugging. Symposium on User interface software and technology (UIST '13). 473–484. https://doi.org/10.1145/2501988.2502050

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

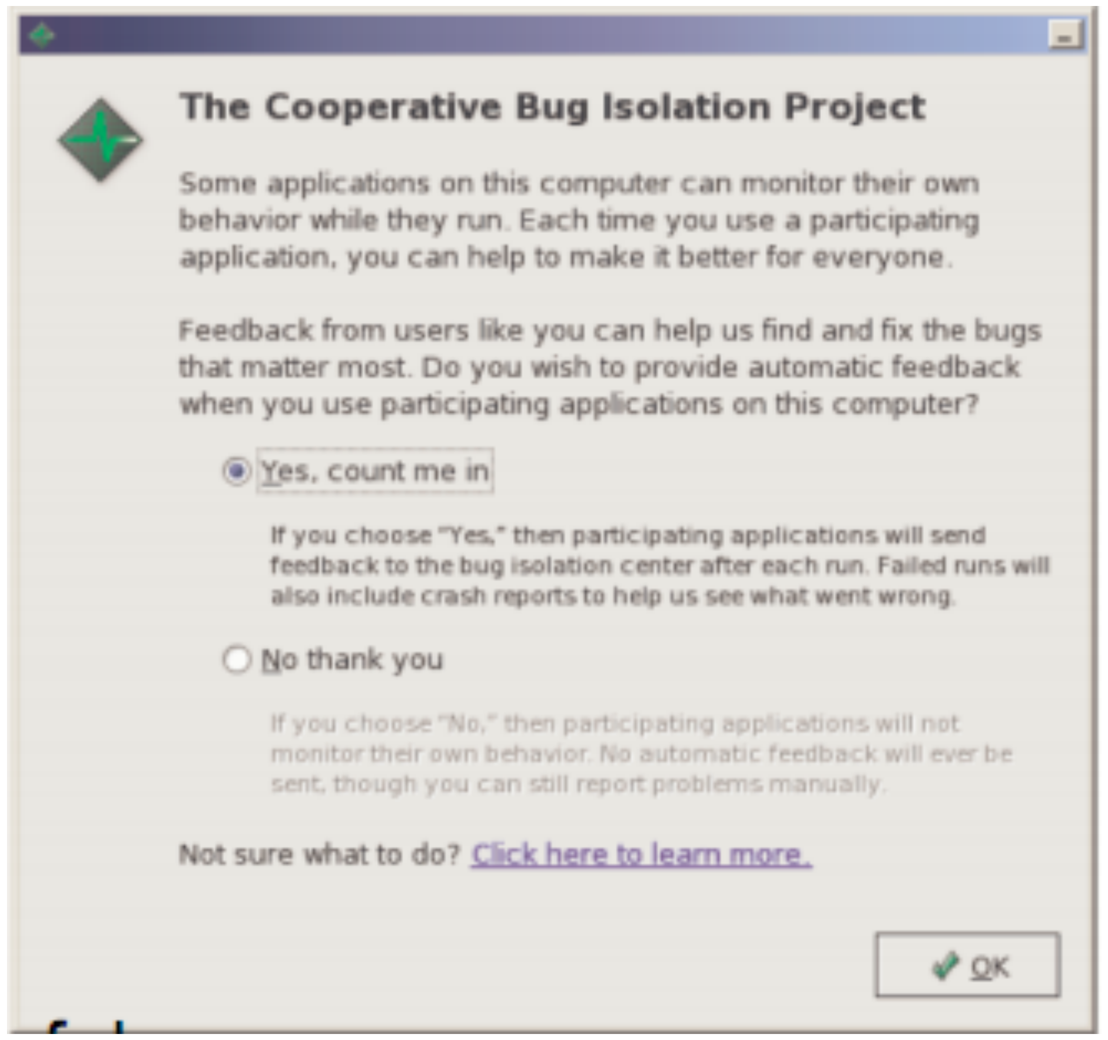
Problems

- 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.

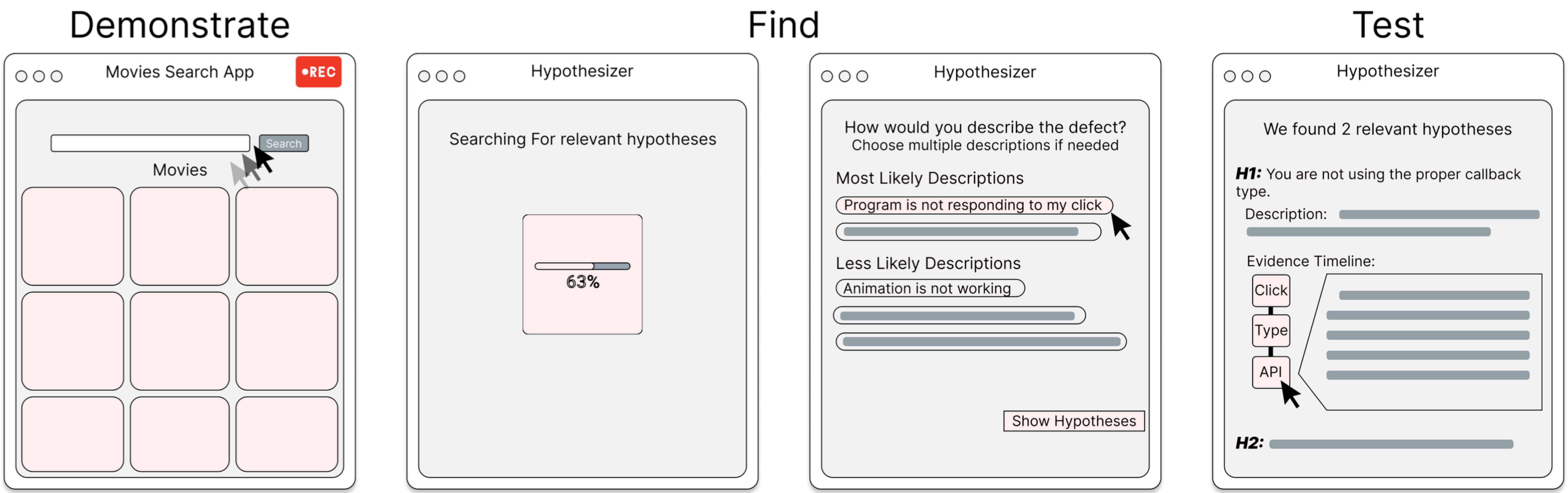
Find shortest repro steps

- 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
-

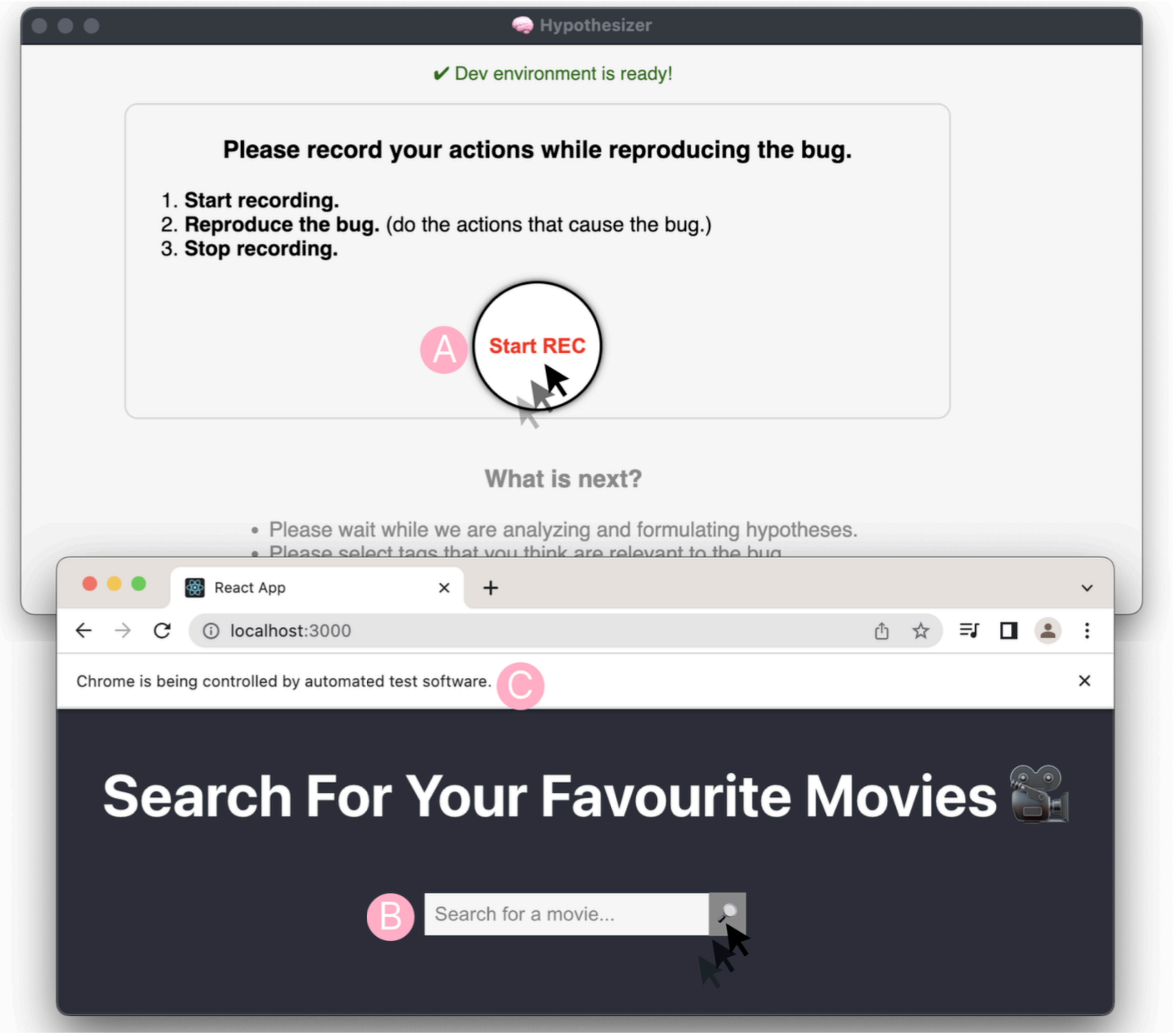
Step	Test case									*test*	
1	Δ_1	1	2	3	4	.	.	.	.	**?**	
2	Δ_2	.	.	.	.	5	6	7	8	✘	
3	Δ_1	.	.	.	.	5	6	.	.	✔	
4	Δ_2	.	.	.	.	.	.	7	8	✘	
5	Δ_1	.	.	.	.	.	.	7	.	✘	Done
Result		.	.	.	.	.	.	7	.		

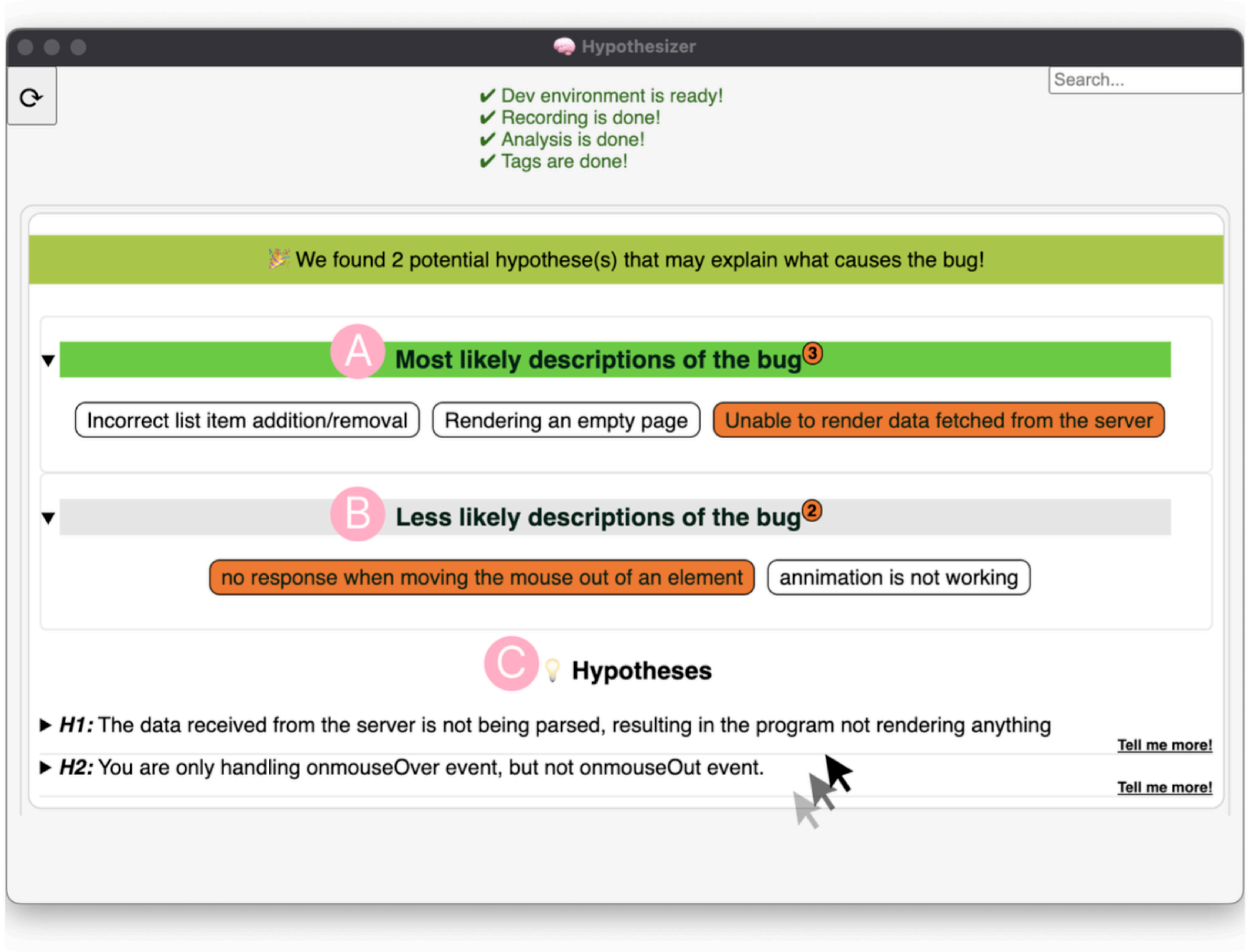
Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. IEEE Transactions on Software Engineering 28(2), February 2002, pp. 183-200.

Hypothesis-Based Debuggers



Abdulaziz Alaboudi and Thomas D. Latoza. 2023. Hypothesizer: A Hypothesis-Based Debugger to Find and Test Debugging Hypotheses. In Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23). Association for Computing Machinery, New York, NY, USA, Article 73, 1–14. https://doi.org/10.1145/3586183.3606781


Hypothesizer
✔ Dev environment is ready!
Please record your actions while reproducing the bug.
1. Start recording.
2. Reproduce the bug. (do the actions that cause the bug.)
3. Stop recording.
A
Start REC
What is next?
Please wait while we are analyzing and formulating hypotheses.
React App
localhost:3000
Chrome is being controlled by automated test software.
C
Search For Your Favourite Movies
B
Search for a movie...


Hypothesizer
Search...
✔ Dev environment is ready!
✔ Recording is done!
✔ Analysis is done!
✔ Tags are done!
We found 2 potential hypothese(s) that may explain what causes the bug!
A
Most likely descriptions of the bug 3
Incorrect list item addition/removal
Rendering an empty page
Unable to render data fetched from the server
B
Less likely descriptions of the bug 2
no response when moving the mouse out of an element
annimation is not working
C
Hypotheses
H1: The data received from the server is not being parsed, resulting in the program not rendering anything
Tell me more!
H2: You are only handling onmouseOver event, but not onmouseOut event.
Tell me more!

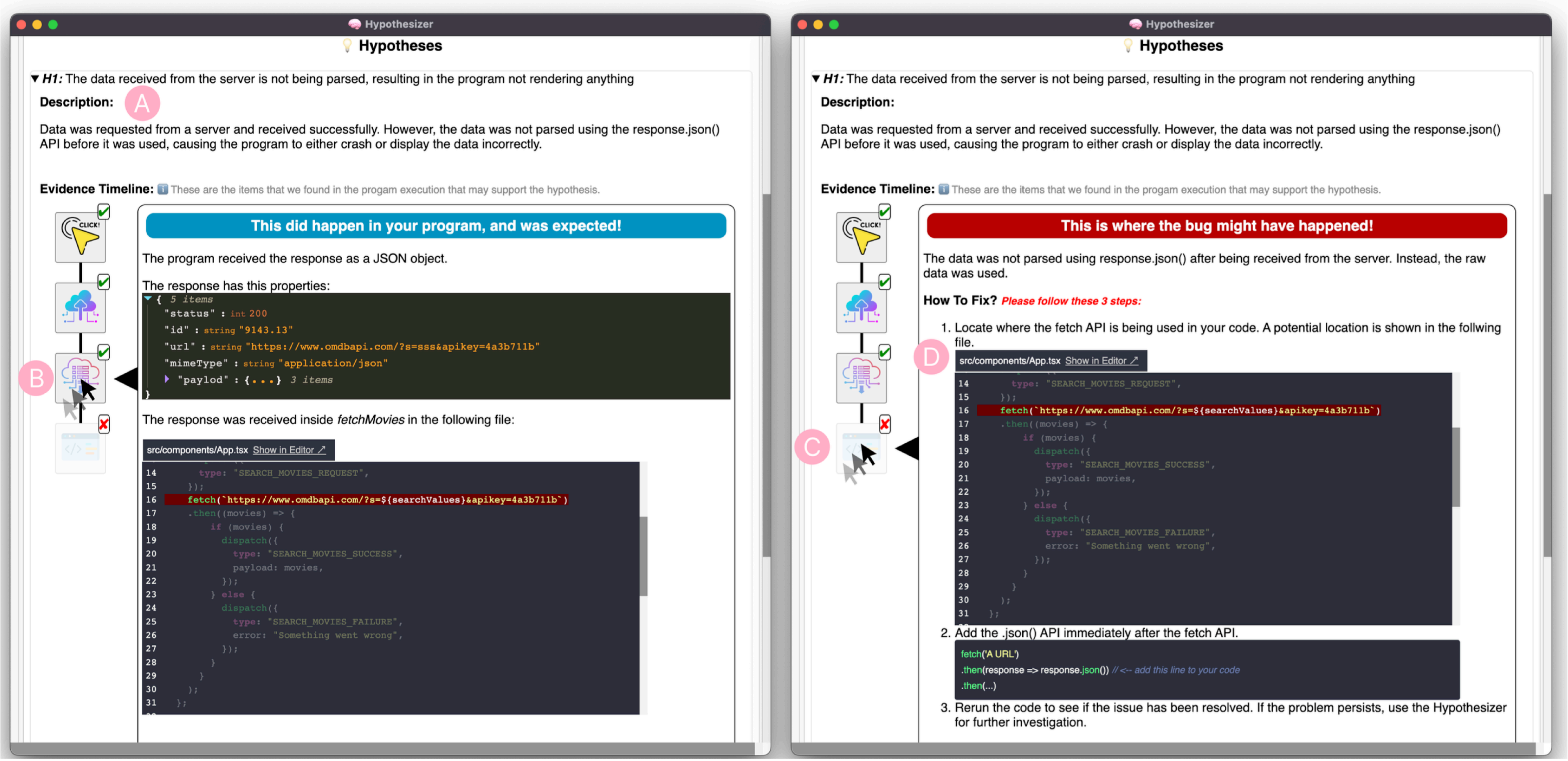


Figure 5: Developers test a hypothesis by expanding it, revealing an extended description(A) and timeline view summarizing evidence (B) confirming (indicated with a ✔) or contradicting the hypothesis. Evidence items may also indicate that they contain a potential starting point for a fix (indicated with a ✘)(C), with step-by-step instructions for implementing a fix (D).

10 min break

Tech Talks

In-Class Activity

- In groups of 2 or 3, try out replay.io
  - Find a sample frontend JavaScript codebase that you can run it on (e.g., your 695 project)
  - Download and setup the tool, run it on your codebase
  - Use it to try to understand a behavior in the web application codebase
  - Write a reflection on your experiences using the tool:
    - How did it help in understanding application behavior?
    - How did tool change your approach or strategy to working with execution behavior?
    - What did you like most about the tool?
    - What's hardest to use about the tool? What information would you like to see that it doesn't currently provide?
- Submission
  - Submit a pdf with your reflection through Blackboard. 1 submission per group. Due 7:10pm today.