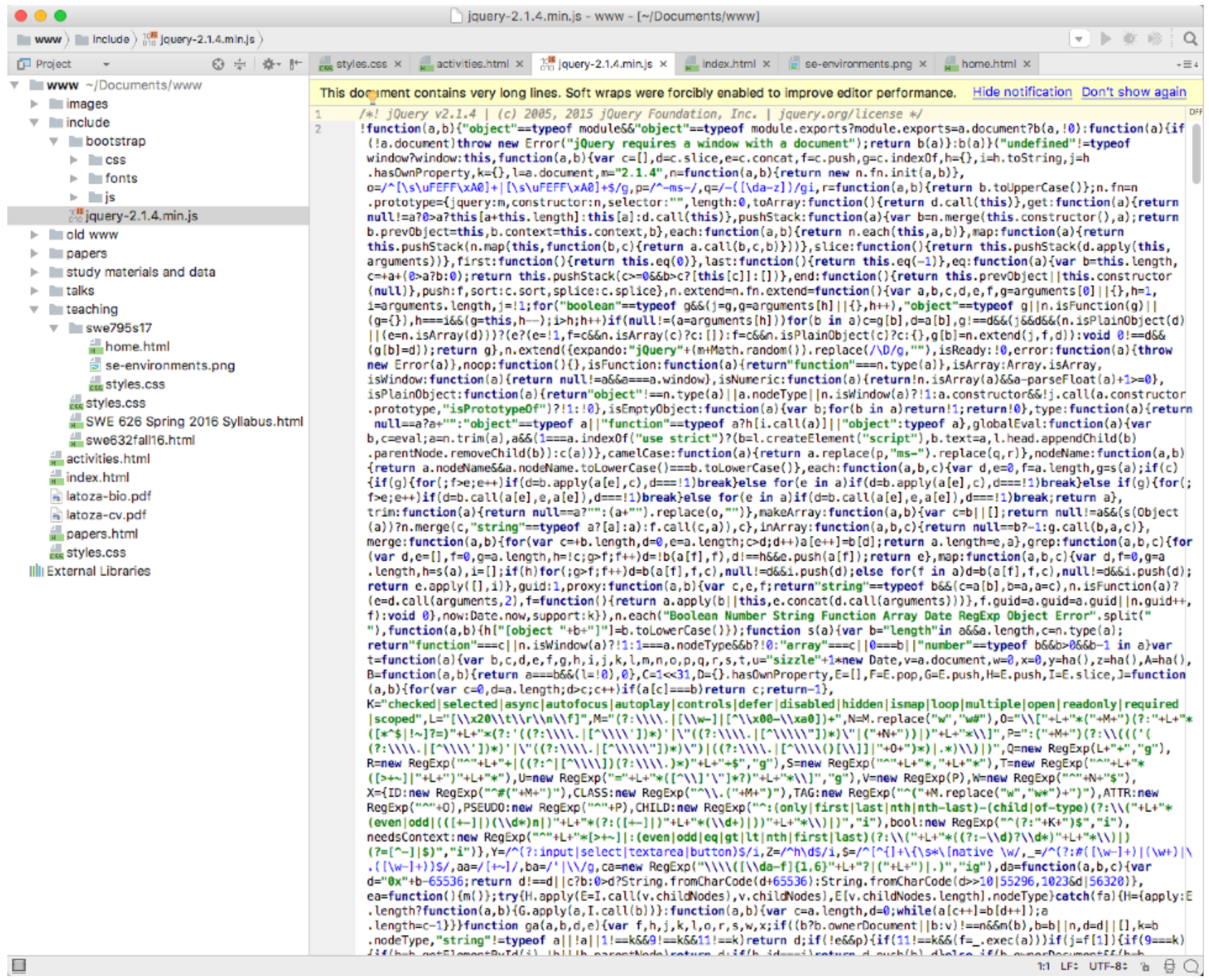
Course Overview and Study Design

CS 695 / SWE 699, Fall 2023

Programming Tools

George Mason University

Exercise: A Programming Tool



- What is a feature offered by a development environment?
- How does this help a developer work more effectively?

Examples of programming tools

- Syntax highlighting
- Errors and warnings
- Autocomplete
- Code templates
- Breakpoint debugger
- Logging statements
- Edit and continue
- GUI builder
- Version control
- Refactoring

Programming Tool

- Software that enables software developers to accomplish a software engineering activity.

- Key concepts:
  - Software engineering activity
  - Task
  - Challenge
  - Support

Why study programming tools?

- Programming tools can have important impact on **productivity**
  - e.g., debugging through console.log vs breakpoint debugger
- By understanding real challenges developers face, help to understand where **new tools** might help developers work more quickly
- Gather evidence to **assess** if a tool is helping
  - Will adopting new IDE plugin *x* help you { debug, reuse code, edit code, navigate, … } faster?

Course Goals

- Offer comprehensive overview of research on programming tools
  - Will **not** go into technical details of approaches
  - Focus on **insights** into software development work
- Gain experience with HCI & SE methods for designing programming tools
- Gain experience reading & critically assessing research papers

Topics

1. Developer-Centered Design
    1. Overview & study design
    2. Design process
    3. Problem solving
2. Editing Code
    1. Structured editors: writing code, without the syntax errors
    2. Program transformation: editing code with GUI commands
    3. GUI builders & No Code: generating code with GUI commands
    4. Program synthesis: transforming text into code
3. Understanding Code
    1. Live Programming: working with immediate, real time feedback
    2. Computational Notebooks: seeing a computation, step by step
    3. Reusing code - external APIs
    4. Navigating code - getting around and reading internal code
    5. Software visualization - diagrams and pictures that explain code
4. Fixing Code
    1. Detecting defects
    2. Debugging

Class format

- Part 1: Lecture: Survey of a type of programming tool
- Part 2: In-Class Activity
- Part 3: Tech Talks

Course Staff

- Prof. Thomas LaToza
  - Office hour: ENGR 4431
    Wed 3:00 - 4:30pm or by appointment
  - Areas of research: software engineering, human-computer interaction, programming tools
  - 15 years experience designing programming tools

TA: Emad Aghayi

Course Readings

- Will have 2 readings a week
  - Responsible for reading both readings and responding to a prompt on Piazza.

Project

- The homework in this course will be in the form of a project. All project work will occur in groups of up to four people.
- HW0: Project Proposal (20 points)
- HW1: Review of Literature (100 points)
- HW2: Study of Current Practice (200 points)
- HW3: Tool Sketch (130 points)
- HW4: Tool Prototype (250 points)

HW0: Project Proposal

Tech Talk

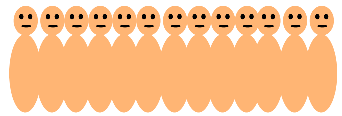
Grades

- Responses to readings: 20%
- Tech Talk: 10%
- Project: 70%

Example: Developing a programming tool

Observations of developers in the field

Participants



17 professional developers

Tasks

~90 minutes
picked one of **their** own coding tasks involving unfamiliar code

Transcripts

Interesting. This looks like, this looks like the code is approximately the same but it's refactored. But the other code is.

Changed what flags it's ???

He added a new flag that I don't care about. He just renamed a couple things.

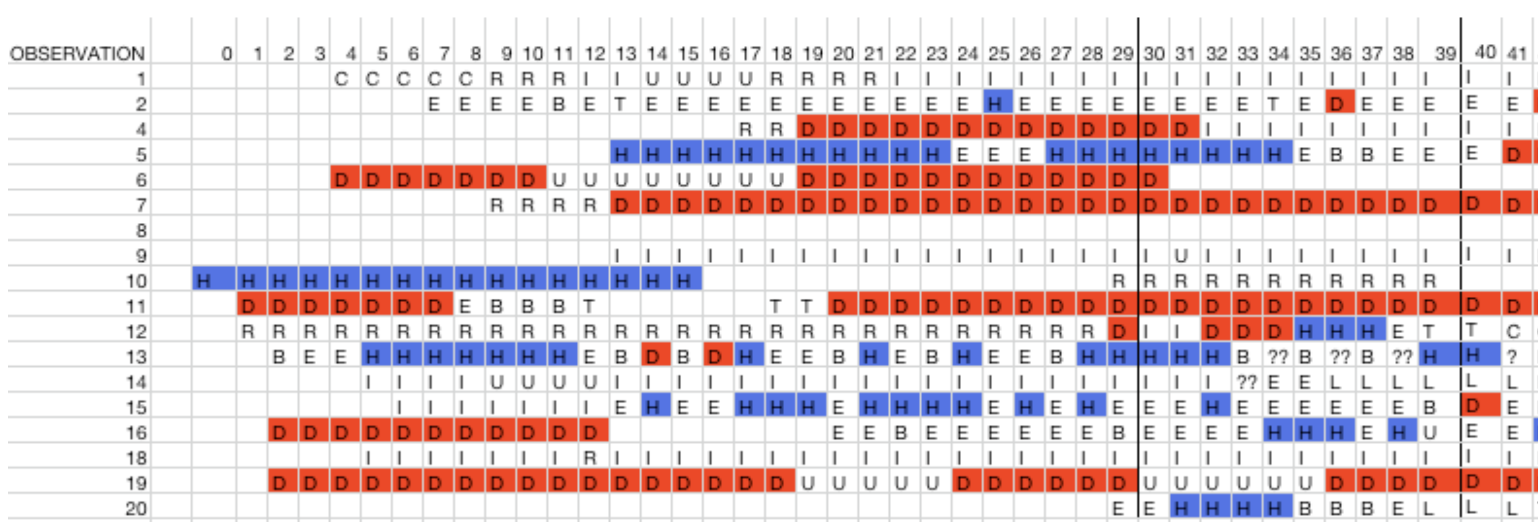
Well.

So the change seemed to have changed some of the way these things are registered,

but I didn't see anything that talked at all about whether the app is running or whether the app is booted. So it seems like, this was useless to me.

(annotated with observer notes about goals and actions) (386 pages)

Activities



Longest activities related to control flow questions

4 out of the 5 longest investigation activities

Primary question	**Time** (mins)	**Related control flow question**
How is this data structure being mutated in this code?	83	Search downstream for **writes** to data structure
"Where [is] the code assuming that the tables are already there?"	53	**Compare** behaviors when tables are or are not loaded
How [does] application state change when *m* is called denoting startup completion?	50	Find field **writes** caused by m
"Is [there] another reason why *status* could be non-zero?"	11	Find statements through which values **flow** into status

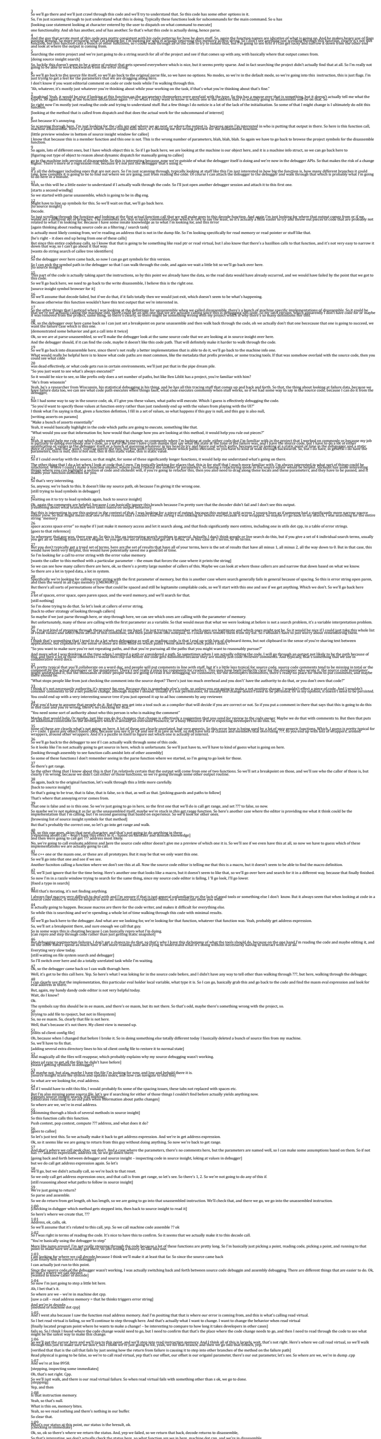
5 out of the 5 longest debugging activities

Where is method *m* generating an error?	66	Search downstream from *m* for **error** text
What resources are being acquired to cause this deadlock?	51	Search downstream for **acquire** method calls
"When they have this attribute, they must use it somewhere to generate the content, so where is it?"	35	Search downstream for **reads** of attribute
"What [is] the test doing which is different from what my app is doing?"	30	**Compare** test traces to app traces
How are these thread pools interacting?	19	Search downstream for **calls** into thread pools

Longest debugging activity

Where is method *m* generating an error?

	Rapidly found method *m* implementing command Unsure **where** it generated error
static call traversal	Statically traversed calls looking for something that would generate error
debugger	Tried debugger
grep	Did string **search** for error, found it, but many callers
debugger	**Stepped** in debugger to find something relevant
static call traversal	Statically **traversed** calls to explore
debugger	Went back to **stepping** debugger to inspect values Found the answer

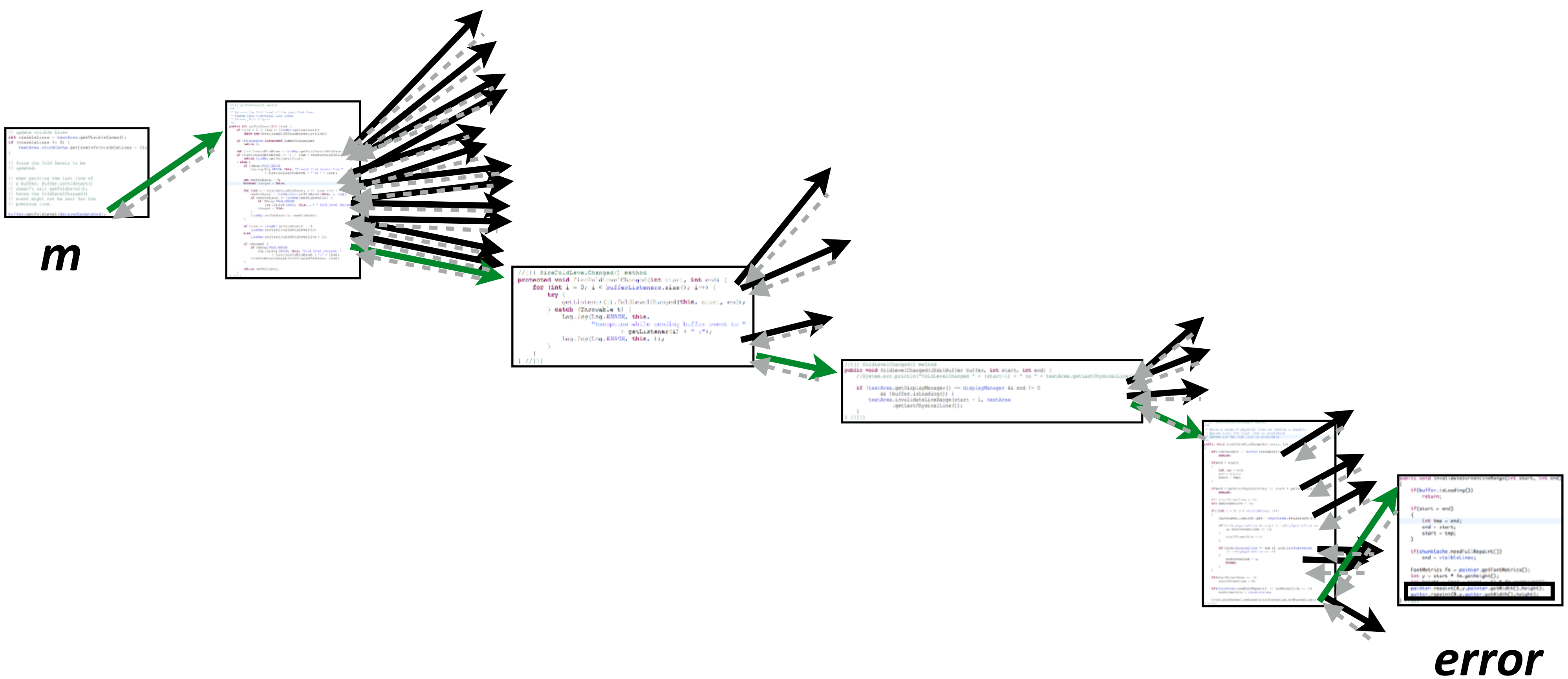


(66 minutes)

Why was this question so hard to answer?

Hard to pick the **control flow path** that leads from starting point to target

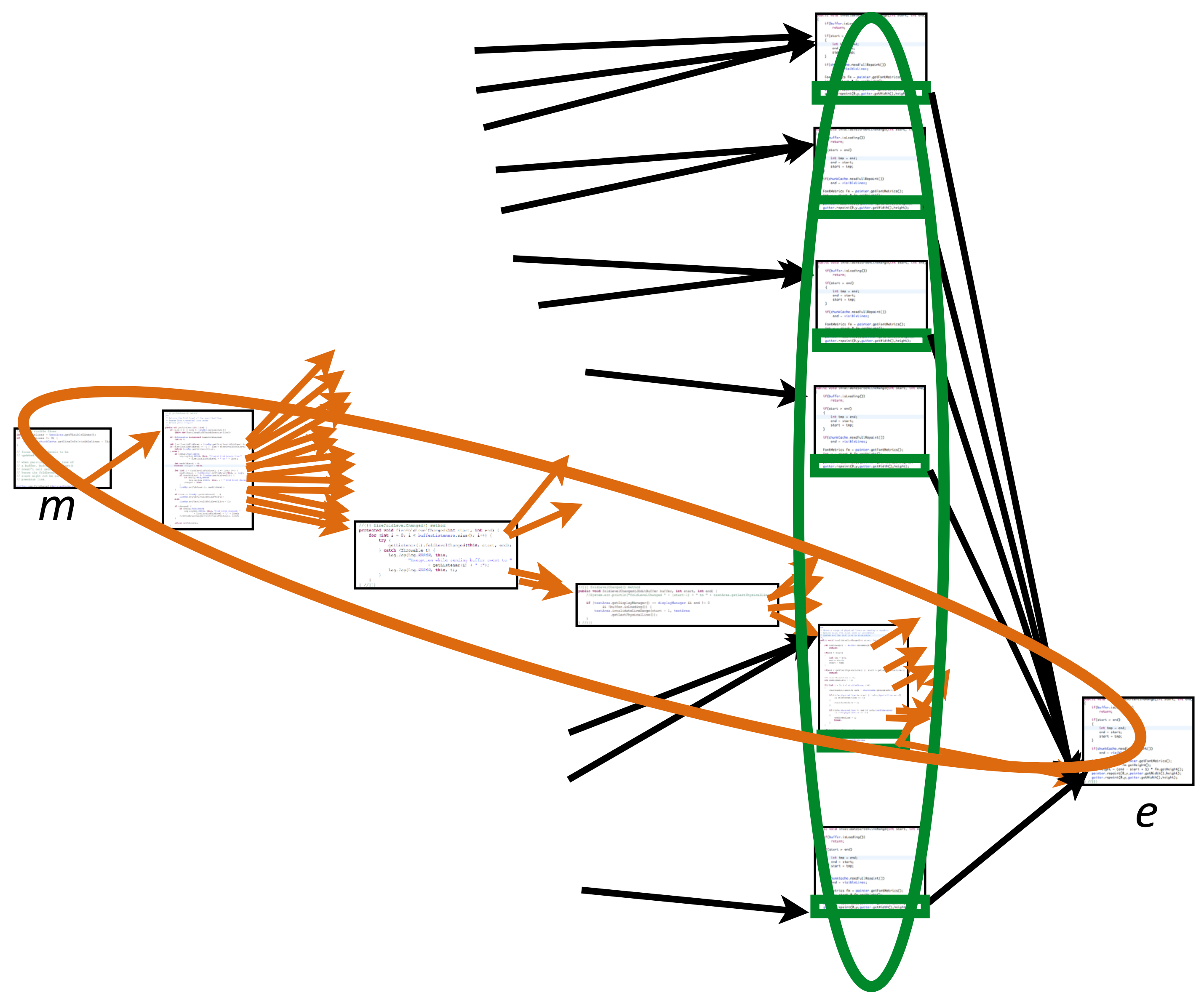
Guess and check: which path leads to the target?



Reachability question: example

Where is method *m* generating an error?

A search along **feasible paths downstream** or **upstream** from a statement (*m*) for **target statements** matching **search criteria (calls to method e)**



feasible paths ∩ **statements matching search criteria**

Longest activities related to reachability questions

4 out of the 5 longest investigation activities

Primary question	**Time** (mins)	**Related reachability question**
How is this data structure being mutated in this code?	83	Search downstream for **writes** to data structure
"Where [is] the code assuming that the tables are already there?"	53	**Compare** behaviors when tables are or are not loaded
How [does] application state change when *m* is called denoting startup completion?	50	Find field **writes** caused by m
"Is [there] another reason why *status* could be non-zero?"	11	Find statements through which values **flow** into status

5 out of the 5 longest debugging activities

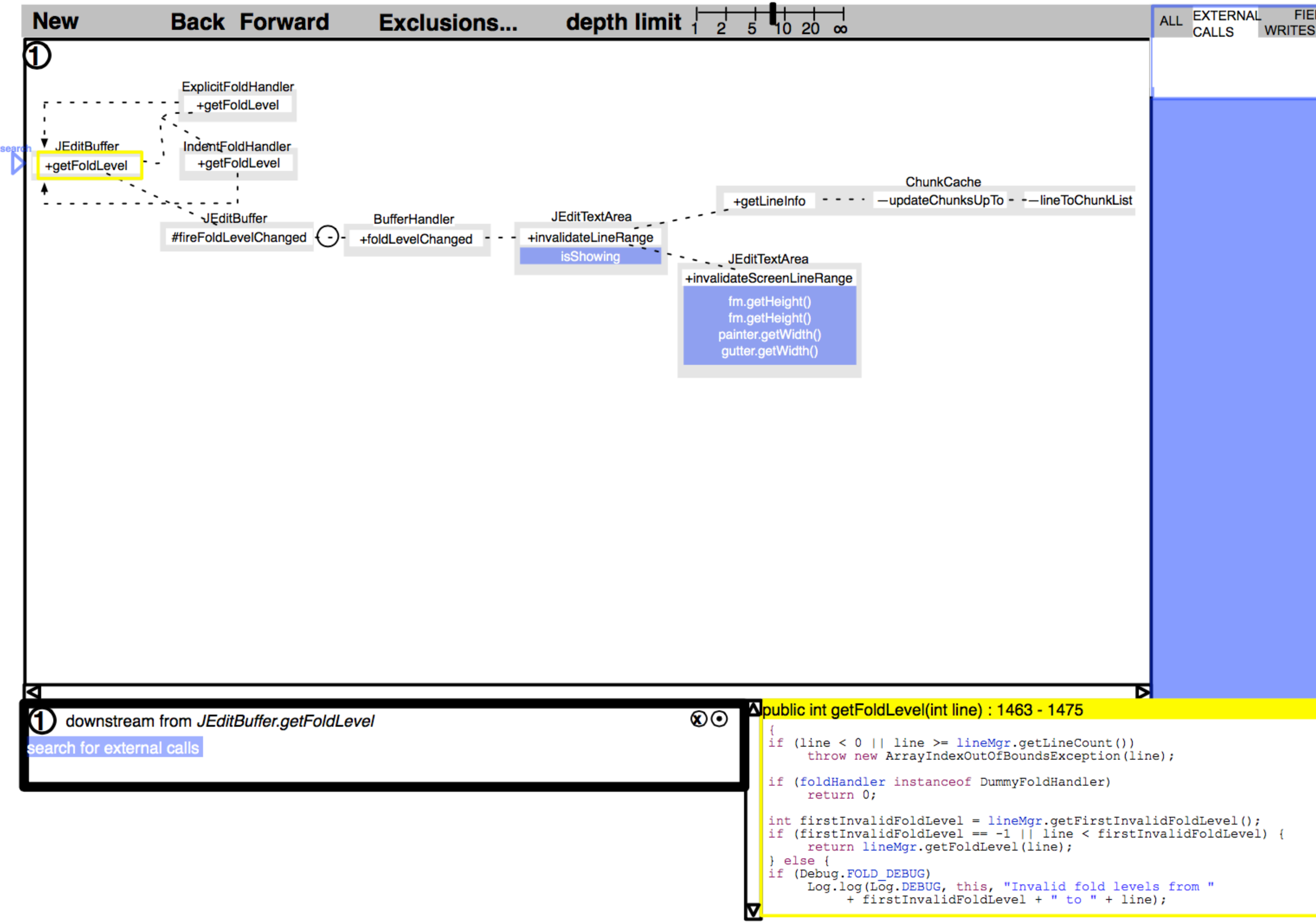
Where is method *m* generating an error?	66	Search downstream from *m* for **error** text
What resources are being acquired to cause this deadlock?	51	Search downstream for **acquire** method calls
"When they have this attribute, they must use it somewhere to generate the content, so where is it?"	35	Search downstream for **reads** of attribute
"What [is] the test doing which is different from what my app is doing?"	30	**Compare** test traces to app traces
How are these thread pools interacting?	19	Search downstream for **calls** into thread pools

Overall findings

- Found that developers can construct **incorrect** mental models of control flow, leading them to insert **defects**

- Found that the **longest** investigation & debugging activities involved a single primary question about control flow

- Found evidence for an underlying cause of these difficulties
  Challenges answering **reachability question**s


New
Back
Forward
Exclusions...
depth limit
1 2 5 10 20 ∞
ALL
EXTERNAL CALLS
FIE
WRITES
ExplicitFoldHandler
+getFoldLevel
JEditBuffer
+getFoldLevel
IndentFoldHandler
+getFoldLevel
ChunkCache
+getLineInfo
—updateChunksUpTo
—lineToChunkList
JEditBuffer
#fireFoldLevelChanged
BufferHandler
+foldLevelChanged
JEditTextArea
+invalidateLineRange
isShowing
JEditTextArea
+invalidateScreenLineRange
fm.getHeight()
fm.getHeight()
painter.getWidth()
gutter.getWidth()
downstream from JEditBuffer.getFoldLevel
search for external calls
public int getFoldLevel(int line) : 1463 - 1475
{
if (line < 0 || line >= lineMgr.getLineCount())
throw new ArrayIndexOutOfBoundsException(line);
if (foldHandler instanceof DummyFoldHandler)
return 0;
int firstInvalidFoldLevel = lineMgr.getFirstInvalidFoldLevel();
if (firstInvalidFoldLevel == -1 || line < firstInvalidFoldLevel) {
return lineMgr.getFoldLevel(line);
} else {
if (Debug.FOLD_DEBUG)
Log.log(Log.DEBUG, this, "Invalid fold levels from "
+ firstInvalidFoldLevel + " to " + line);

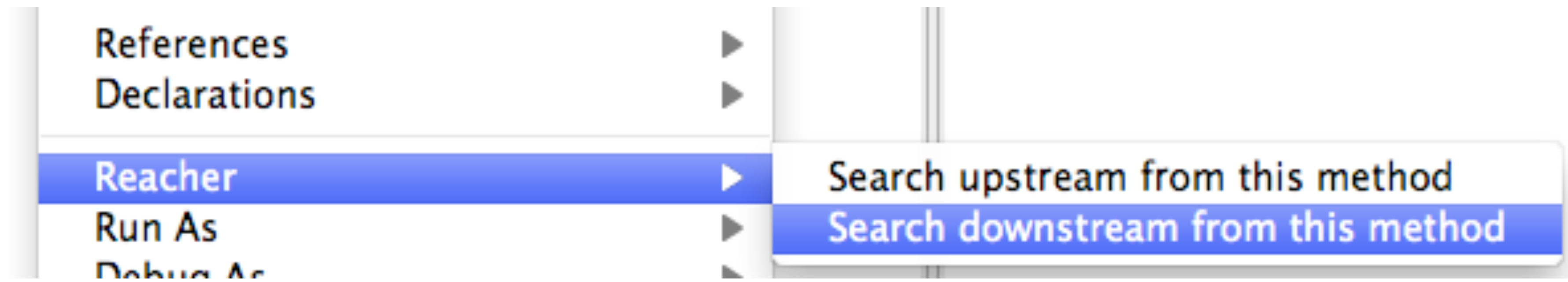
Paper prototype study

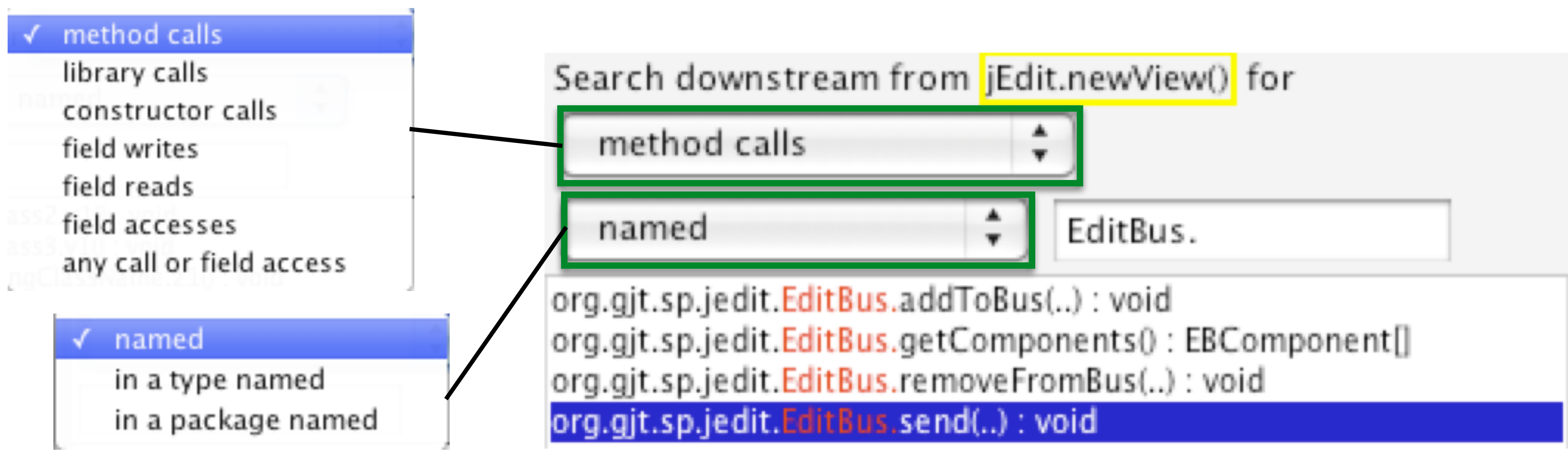
- Built mockups of interface for task from lab study
- Asked 1 participant to complete lab study task with Eclipse & mockup of Reacher
  - Paper overlay of Reacher commands on monitor
  - Experimenter opened appropriate view
- Asked to think aloud, screen capture + audio recording

Study results

- Used Reacher to explore code, unable to complete task
- Barriers discovered
  - Wanted to see methods before or after, not on path to origin or destination
  - Switching between downstream and upstream confusing, particularly search cursor
  - Found horizontal orientation confusing, as unlike debugger call stacks
  - Wanted to know when a path might execute

Step 2: Find statements matching search criteria





Examples of observed reachability questions Reacher supports	Steps to use Reacher
What resources are being acquired to cause this deadlock?	Search downstream for each method which might acquire a resource, pinning results to keep them visible
When they have this attribute, they must use it somewhere to generate the content, so where is it?	Search downstream for a field read of the attribute
How are these thread pools interacting?	Search downstream for the thread pool class
How is data structure *struct* being mutated in this code (between *o* and *d*)?	Search downstream for *struct* class, scoping search to matching type names and searching for field writes.
How [does] application state change when *m* is called denoting startup completion?	Search downstream from *m* for all field writes

Step 3: Help developers understand paths and stay oriented

Goal: help developers reason about control flow by summarizing statements along paths in **compact** visualization

Challenges:
control flow paths can be

complex

long

repetitive

developers get lost and disoriented navigating code

Approach:

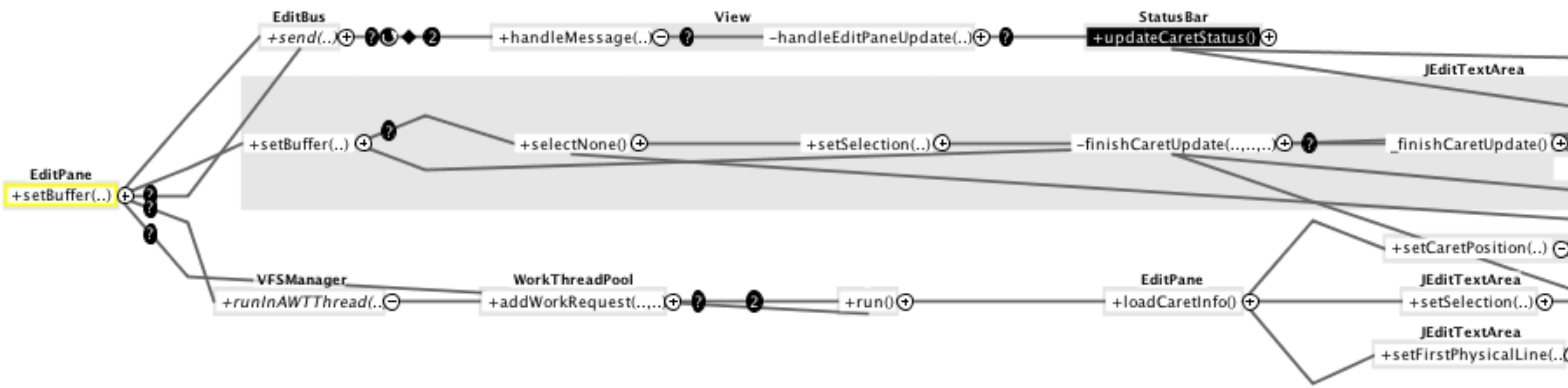
visually encode properties of path

hide paths by default

coalesce similar paths

use visualization to support navigation

Example



EditPane
+setBuffer(..)
EditBus
+send(..)
+handleMessage(..)
View
-handleEditPaneUpdate(..)
StatusBar
+updateCaretStatus()
JEditTextArea
+setBuffer(..)
+selectNone()
+setSelection(..)
-finishCaretUpdate(..,..,..)
_finishCaretUpdate()
+setCaretPosition(..)
VFSManager
+runInAWTThread(..
WorkThreadPool
+addWorkRequest(..,..
+run()
EditPane
+loadCaretInfo()
JEditTextArea
+setSelection(..)
JEditTextArea
+setFirstPhysicalLine(..

Evaluation

Does REACHER enable developers to answer reachability questions faster or more successfully?

Method

12 developers

15 minutes to answer **reachability** question x **6**

Eclipse only on 3 tasks

Eclipse w/ REACHER on 3 tasks

(order counterbalanced)

Tasks

Based on developer questions in lab study.

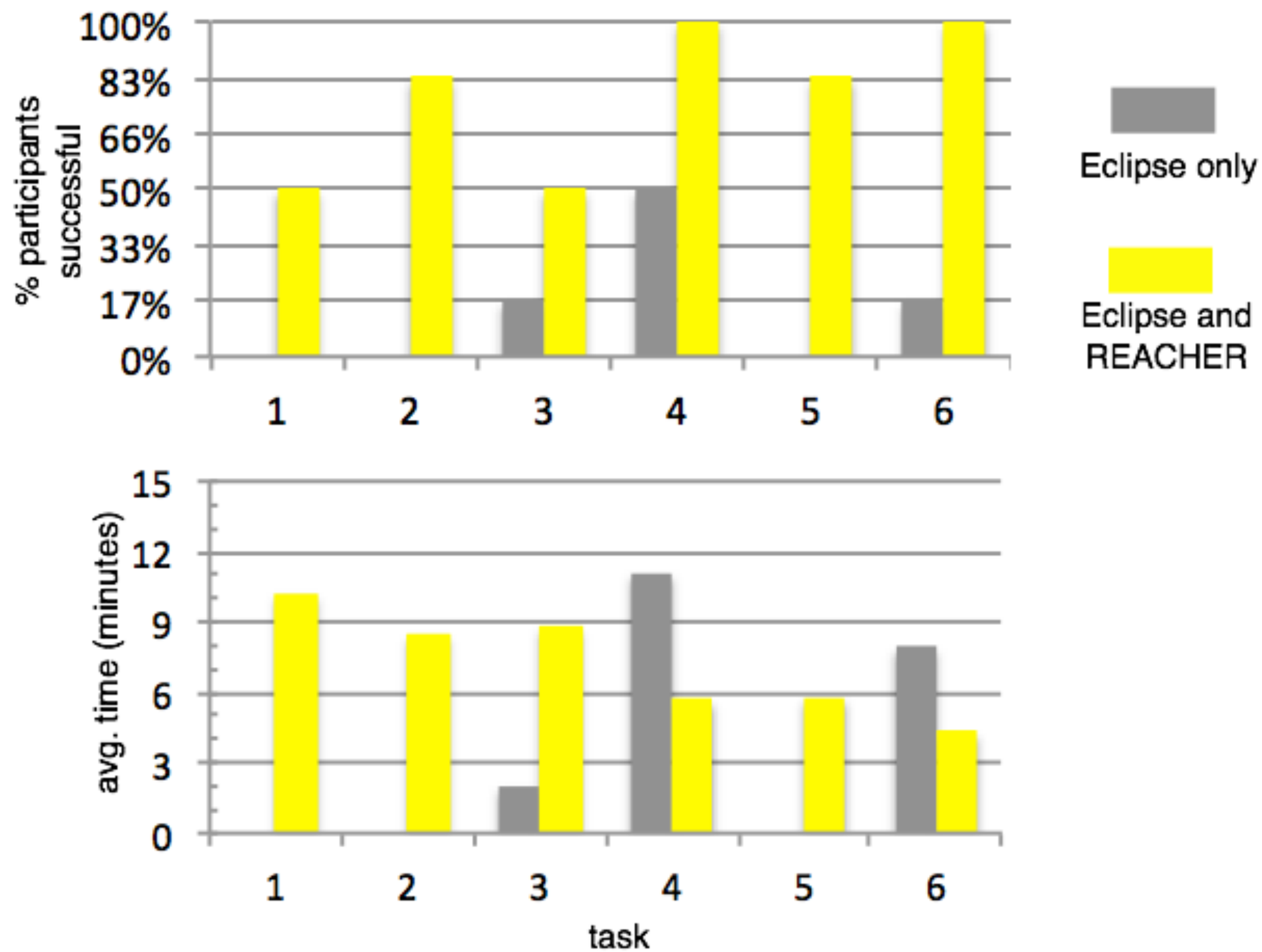
Example:

When a new view is created in jEdit.newView(View), what messages, in what order, may be sent on the EditBus (EditBus.send())?

Results

Developers with REACHER were **5.6** times more **successful** than those working with Eclipse only.

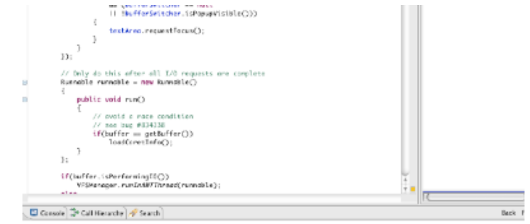
(not enough successful to compare time)



Task time includes only participants that succeeded.

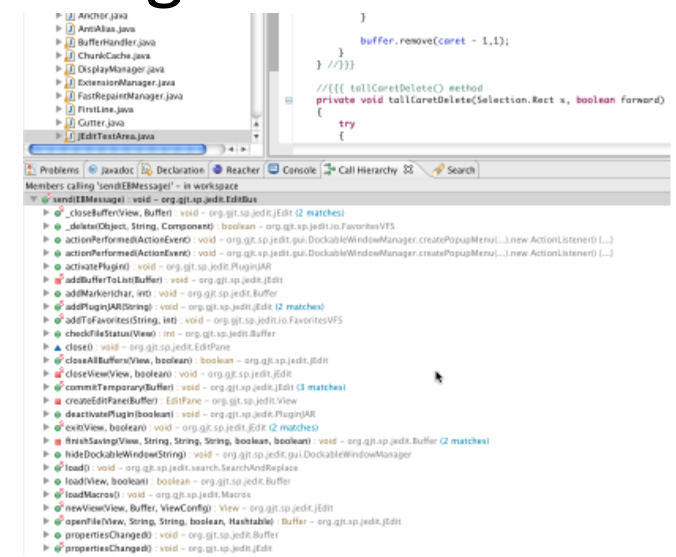
REACHER helped developers stay oriented

Participants with **REACHER** used it to jump between methods.



"It seems pretty cool if you can navigate your way around a complex graph."

When **not** using REACHER, participants often reported being lost and confused.



"Where am I? I'm so lost."

"These call stacks are horrible."

"There was a call to it here somewhere, but I don't remember the path."

"I'm just too lost."

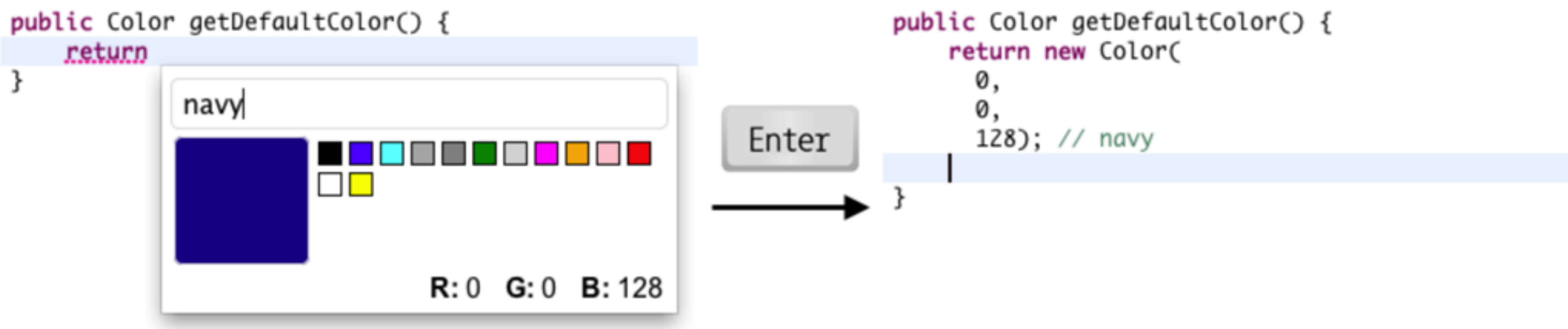
Participants reported that they liked working with REACHER.

"I like it a lot. It seems like an easy way to navigate the code. And the view maps to more of how I think of the call hierarchy."

"Reacher was my hero. … It's a lot more fun to use and look at."

"You don't have to think as much."

Shorter Example: Active Code Completion



Cyrus Omar, Young Seok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active code completion. International Conference on Software Engineering, 859-869.

Studies of software development

Why do studies?

- What tasks are most **important** (time consuming, error prone, frequent, ...)?
  (exploratory studies) (potential usefulness of tool)

- Are these claimed productivity benefits **real**?
  (evaluation studies)

- **Know** the user!
  (You may or may not be a typical developer)

Build a tool, clearly it's [not] useful!

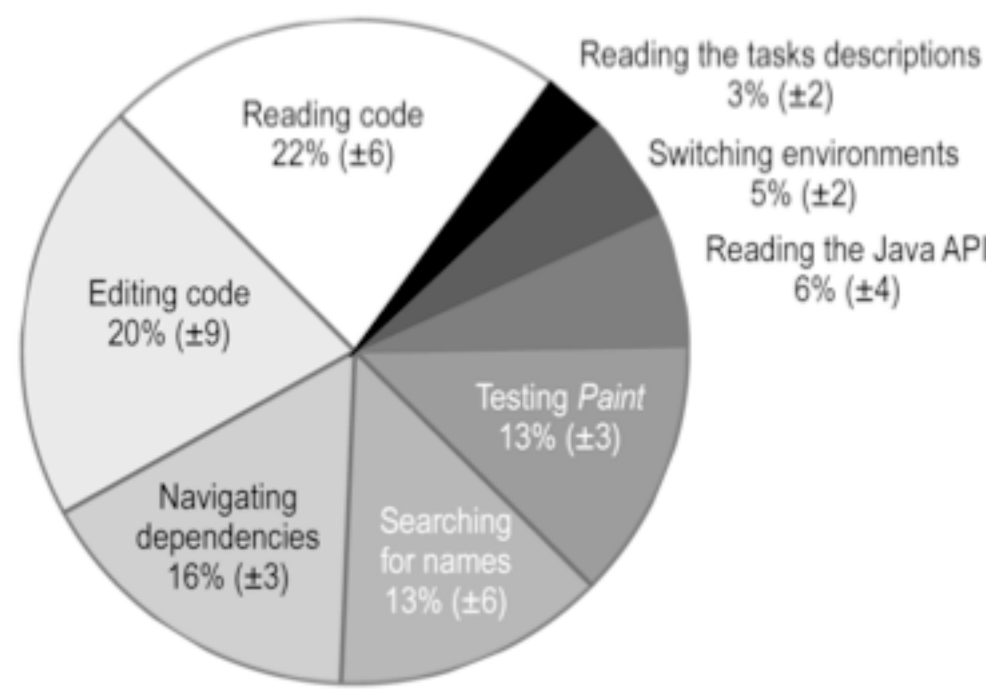
- 80s SigChi bulletin: **~90%** of evaluative studies found no benefits of tool
- A study of 3 code exploration tools found **no benefits**
  [de Alwis+ ICPC07]
- How do you convince real developers to **adopt** tool?
  Studies can provide evidence!

Why not just ask developers?

- Estimates are biased (time, difficulty)
- More likely to remember very hardest problems
  They are hard, but not necessarily typical

- Example of data from study [Ko, Aung, Myers ICSE05]

- 22% of time developers copied too much or too little code



Goal: Theories of developer activity

- A **model** describing the **strategy** by which developers **frequently** do an **activity** that describes **problems** that can be **addressed** ("design implications") through a better designed tool, language, or process that more effectively supports this strategy.

Exercise - How do developers debug?

Some debugging strategies

- by having the computer fix the bug for them.
- by inspecting values, stepping, and setting breakpoints in debugger
- by adding and inspecting logging statements
- by hypothesizing about what they did wrong and testing these hypotheses.
- by asking why and why didn't questions.
- by following {static, dynamic, thin} slices.
- by searching along control flow for statements matching search criteria
- by using information scent to forage for relevant statements.
- by asking their teammates about the right way to do something.
- by checking documentation or forums to see if they correctly made API calls.
- by checking which unit tests failed and which passed.
- by writing type annotations and type checking ("well typed programs never go wrong")

Exercise - what would you like to know about these theories?

Studies provide evidence for or against theories

- Do developers actually do it?
  Or would developers do it given better tools?
- How frequently? In what situations?
- What factors influence use? How do these vary for different developers, companies, domains, expertise levels, tools, or languages?
- How long does it take?
- Are developers successful? What problems occur?
- What are the implications for design? How hard is it to build a tool that solves the problems developers experience? How frequently would it help?

A single study will not answer all these questions

- But thinking about these questions helps to
  - -set scope
  - -describe limitations of study
  - -pick population to recruit participants from
  - -plan followup complementary studies

Analytical vs. empirical generalizability

Empirical: The angle of the incline significantly affects the speed an object rolls down the incline!

-depends on similarity between situations

-need to sample lots of similar situations

-comes from purely quantitative measurements

Analytical: F = m * a

-depends on theory's ability to predict in other situations

-describes a mechanism by which something happens

-building such models requires not just testing an effect, but understanding situations where effect occurs (often qualitative data)

Empirical vs. analytical generalizability in HASD

- **Empirical:** developers using statically typed languages are significantly more productive than those using dynamically typed languages.

- **Analytical**: static type checking changes how developers work by [...]

- Is the question, "Does Java, SML, or Perl lead to better developer productivity even answerable?"

Types of studies

Exploratory studies

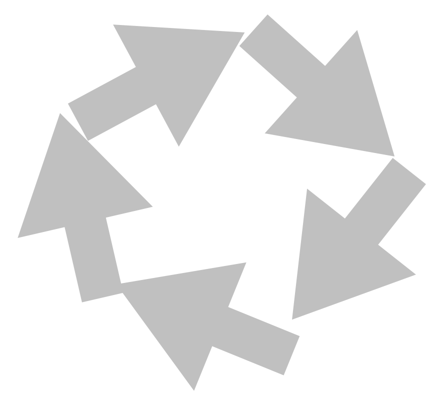
survey
indirect observation
contextual inquiry
...

Models

questions
information needs
use of time
....

Generate tool designs

scenarios
mockups



(Expensive) evaluation studies

lab study
field deployment

(Cheap) evaluation studies

heuristic evaluation
paper prototypes
participatory design
...

Implement tool

(Some) types of exploratory studies

- Field observations / ethnography
  Observe developers at work in the field
- Natural programming
  Ask developers to naturally complete a task
- Contextual inquiry
  Ask questions while developers do work
- Surveys
  Ask **many** developers specific questions
- Interviews
  Ask a **few** developers **open-ended** questions
- Indirect observations (artifact studies)
  Study artifacts (e.g., code, code history, bugs, emails, ...)

Field observations / ethnography

- **Find** software developers
  - Pick developers likely to be doing relevant work
- **Watch** developers do **their** work in their office
- Ask developers to **think-aloud**
  - Stream of consciousness: whatever they are thinking about
  - Thoughts, ideas, questions, hypotheses, etc.
- Take notes, audio record, or video record
  - More is more invasive, but permits detailed analysis
  - Audio: can analyze tasks, questions, goals, timing
  - Video: can analyze navigation, tool use, strategies
  - Notes: high level view of task, interesting observations

Ko, DeLine, & Venolia ICSE07

- Observed **17** developers at Microsoft in 90 min sessions
  - Too intrusive to audio or video record
  - Transcribed think-aloud **during** sessions
- Looked for **questions** developers asked

ources I depend on changed?	0	1	9	41	15	15	*tools 12* ***coworker*** *6 email 4* ***br*** *2 code 1*
uld have caused this behavior?	0	2	17	73	20	22	***coworker*** *5 intuition 4 log 4* ***br*** *4 dbug 2 im 1 code 1 spec 1*
this data structure or function?	0	1	14	71	20	29	*docs 11 code 5* ***coworker*** *4 spec 1*
code implemented this way?	0	2	21	61	37	39	*code 4 intuition 4 history 3* ***coworker*** *2 dbug 2 tools 2 comment 1* ***br*** *1*
n worth fixing?	0	2	6	44	10	20	***coworker*** *12 email 2* ***br*** *1 intuition 1*
implications of this change?	0	2	9	85	44	49	***coworker*** *13 log 1*
urpose of this code?	1	1	5	56	24	29	*intuition 5 code 2 dbug 2 tools 2 spec 1 docs 1*
lly related to this code?	0	1	7	66	27	27	*tools 8 intuition 2 email 1*
nate problem?	0	1	2	49	17	34	***br*** *5* ***coworker*** *1 log 1*
y team's conventions?	0	7	25	41	10	15	*docs 2 tools 2 memory 1*
e failure look like?	0	0	2	88	24	23	***br*** *3 screenshot 2*
es are part of this submission?	0	2	3	61	7	5	*tools 2 memory 2*
ordinate this with this other code?	1	1	4	75	28	30	*docs 2 code 1* ***coworker*** *1*
vill this problem be to fix?	2	2	4	41	15	32	*code 1* ***coworker*** *1 screenshot 1*
used to implement this behavior?	2	2	2	61	27	22	*memory 1 docs 1*
tion was relevant to my task?	1	1	1	59	15	13	*memory 2*

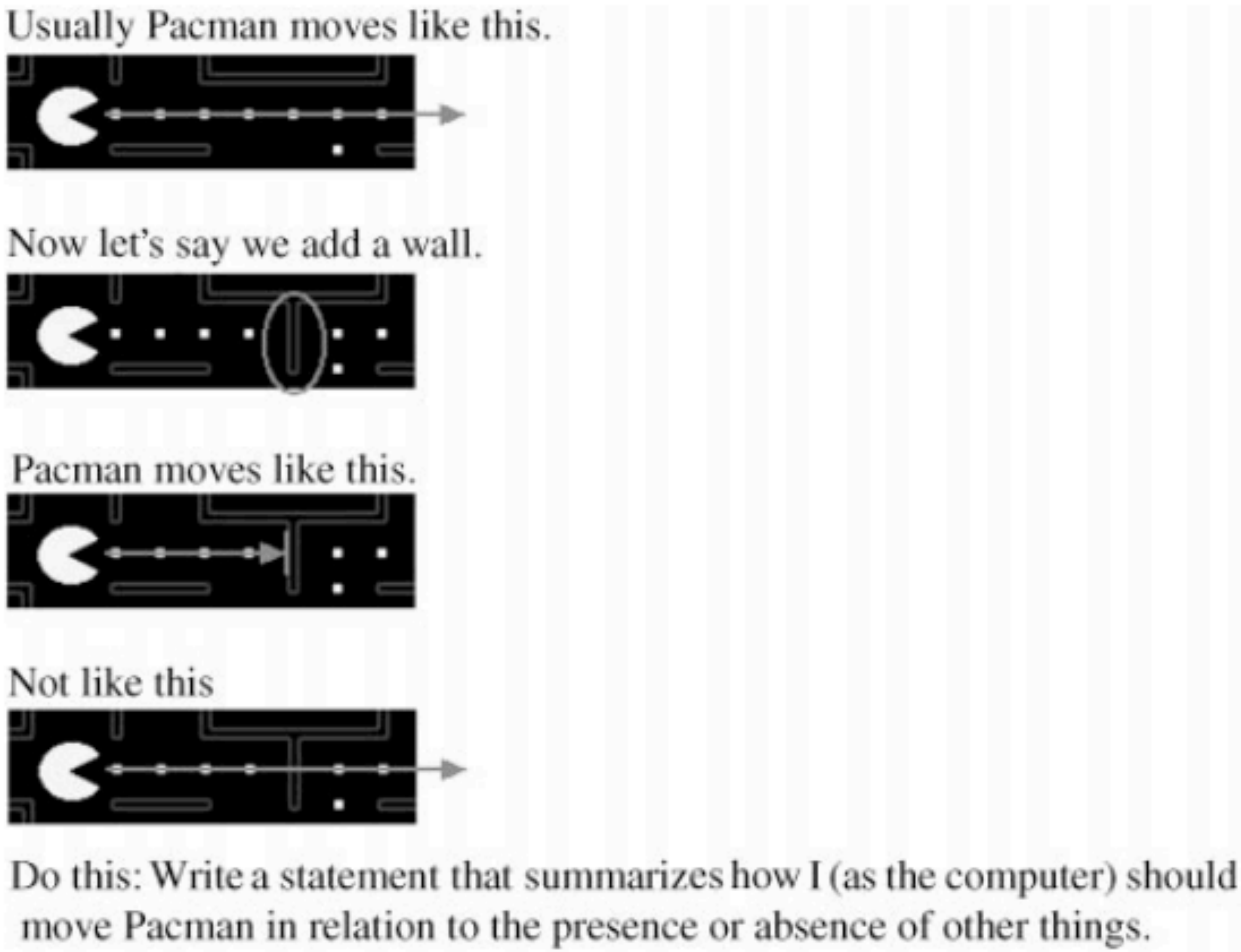
Natural programming

- Design a simple programming task for users
- Ask them to write solution **naturally**
  make up language / APIs / notation of interest
- Analyze use of **language** in solutions

- Advantages:
  elicits the language developers expect to see
  open-ended - no need to pick particular designs
  lets developer design language
- Disadvantages:
  assumes the user's notation is best
  lets developer design notation

Pane, Ratanamahatana, & Myers '01

Grade school students asked to describe in prose how PacMan would work in each of several scenarios



Pane, Ratanamahatana, & Myers IJHCS01

Overall structure

Programming style
54% Production rules/events
18% Constraints
16% Other (declarative)
12% Imperative

Perspective
45% Player or end-user
34% Programmer
20% Other (third-person)

Modifying state
61% Behaviors built into objects
20% Direct modification
18% Other

Pictures
67% Yes

Keywords

AND
67% Boolean conjunction
29% Sequencing

OR
63% Boolean disjunction
24% To clarify or restate a prior item
8% "Otherwise"
5% Other

THEN
66% Sequencing
32% "Consequently" or "in that case"

Control structures

Operations on multiple objects
95% Set/subset specification
5% Loops or iteration

Complex conditionals
37% Set of mutually exclusive rules
27% General case, with exceptions
23% Complex boolean expression
14% Other (additional uses of exceptions)

Looping constructs
73% Implicit
20% Explicit
7% Other

Computation

Remembering state
56% Present tense for past event
19% "After"
11% State variable
6% Discuss future events
5% Past tense for past event

Tracking progress
85% Implicit
14% Maintain a state

Mathematical operations
59% Natural language style — incomplete
40% Natural language style — complete

Motions
97% Expect continuous motion

Randomness
47% Precision
20% Uncertainty without using "random"
18% Precision with hedging
15% Other

Insertion into a data structure
48% Insert first then reposition others
26% Insert without making space
17% Make space then insert
8% Other

Sorted insertion
43% Incorrect method
28% Correct non-general method
18% Correct general method

Surveys

- Can reach **many** (100s, 1000s) developers
  - Websites to run surveys (e.g., SurveyMonkey)
- Find **participants** (usually mailing lists)
- Prepare multiple choice & free response **questions**
  - Multiple choice: faster, standardized response
  - Free response: more time, more detail, open-ended
- Background & **demographics** questions
  - E.g., experience, time in team, state of project, ...
- Study questions
- Open comments

LaToza, Venolia, & DeLine ICSE06

- 104 respondents at Microsoft rated
  - % of time on different activities
  - Tool use frequency & effectiveness
  - Severity of 13 “problems”

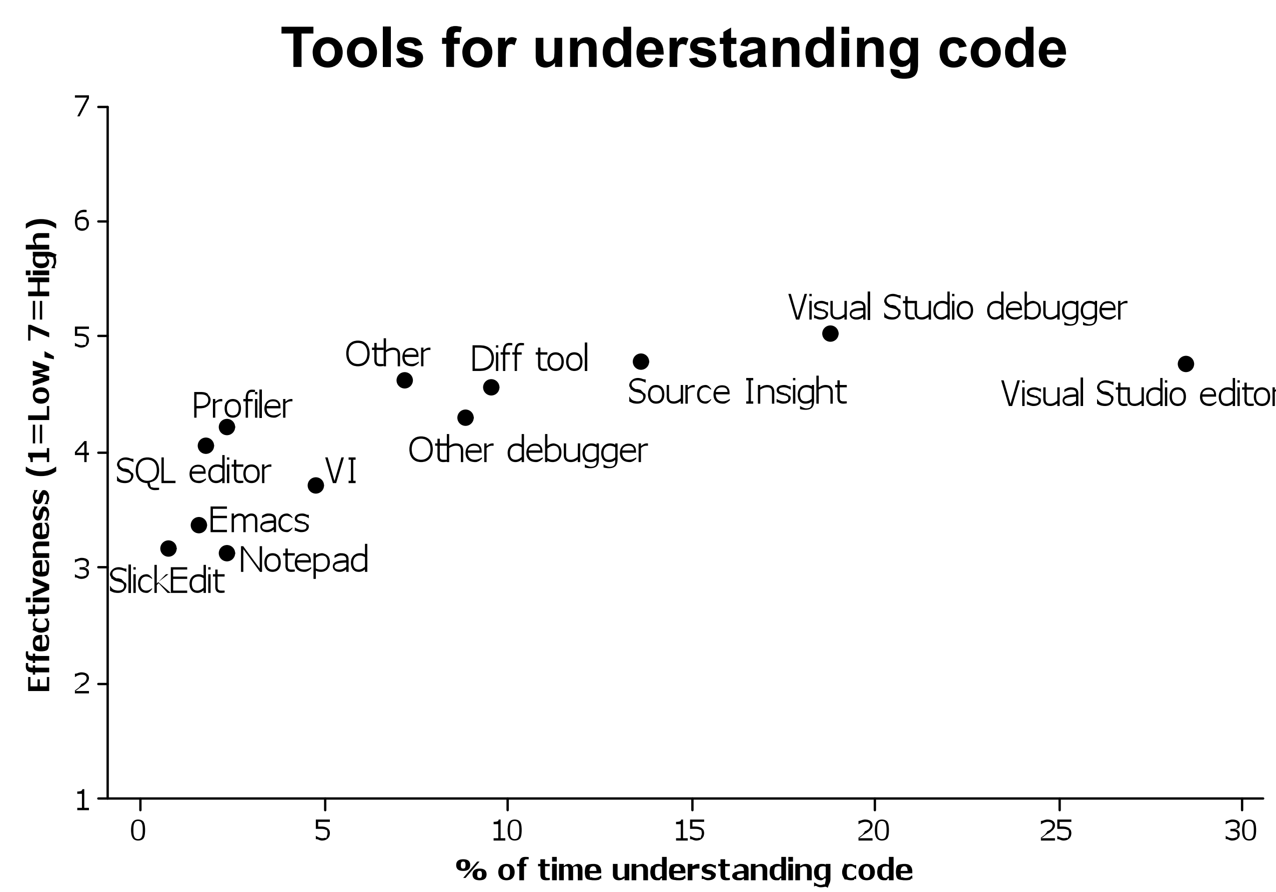
38. Of the time I spent understanding existing code last week, the percent of time I spent

	0%	1%	2%	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Examining source code														
Examining source code check-in comments and diffs														
Examining high-level views of source code (UML diagrams, class hierarchies, call graphs, …)														
Running the code and looking at the results														
Running the code and examining it with a debugger														
Using debug or trace statements														
Other														

39. Other techniques used last week (if you answered “other” above)
(Max Characters: 256)

40. This technique was effective for understanding existing code

	Strongly agree	Agree	Somewhat agree	Neutral	Somewhat disagree	Disagree	Strongly disagree	Didn't use
Examining source code								
Examining source code check-in comments and diffs								
Examining high-level views of source code (UML diagrams, class hierarchies, call graphs, …)								
Running the code and looking at the results								
Running the code and examining it with a debugger								
Using debug or trace statements								
Other (same as above)								
All techniques I used, taken together								



Semi-structured interviews

- Develop a list of focus areas
  - Sets of questions related to topics
- Prompt developer with question on focus areas
  - Let developer talk at length
  - Follow to lead discussion towards interesting topics
- Manage time
  - Move to next topic to ensure all topics covered

Contextual inquiry [Beyer & Holtzblatt]

- Interview **while** doing field observations
- Learn about environment, work, tasks, culture, breakdowns
- Principles of contextual inquiry
  - **Context** - understand work in natural environment
    - Ask to see current work being done
    - Seek concrete data - ask to show work, not tell
    - **Bad**: usually, generally **Good**: Here's how I, Let me show you
  - **Partnership** - close collaboration with user
    - Not interviewer, interviewee! User is the expert.
    - Not host / guest. Be nosy - ask questions.
  - **Interpretation** - make sense of work activity
    - Rephrase, ask for examples, question terms & concepts
  - **Focus** - perspective that defines questions of interest
- Read Beyer & Holtzblatt book before attempting this study

Indirect observations

- **Indirect** record of developer activity
- Examples of **artifacts** (where to get it)
  - Code (open source software (OSS) codebases)
  - Code changes (CVS / subversion repositories)
  - Bugs (bug tracking software)
  - Emails (project mailing lists, help lists for APIs)
- Collect data from instrumented tool (e.g., code navigation)
- Advantages:
  - **Lots** of data, easy to obtain
  - Code, not developer activity
- Disadvantages:
  - Can’t observe developer **activity**

Malayeri & Aldrich, ESOP09

- Gathering data for usefulness of language feature
- Structure of study
  1. Make **hypotheses** about how code would benefit.
  2. Use program analysis to measure **frequency** of idioms in corpus of codebases.
  3. Have **evidence** that code would be **different** with approach.
  4. **Argue** that different code would make developers more productive.
- Example of research questions / hypotheses
- 1. Does the body of a method only use subset of parameters?
  Structural types could make more general
  Are there common types used repeatedly?
- 2. How many methods throw unsupported operation exception?
  Structural supertypes would apply

Exercise: What study(s) would you use?

How would you use studies in these situations?

1. You'd like to design a tool to help web developers more easily reuse code.
2. You'd like to help developers better prioritize which bugs should be fixed.

(Some) types of exploratory studies

- Field observations / ethnography
  Observe developers at work in the field
- Surveys
  Ask **many** developers specific questions
- Interviews
  Ask a **few** developers **open-ended** questions
- Contextual inquiry
  Ask **questions** while developers do work
- Indirect observations (artifact studies)
  Study artifacts (e.g., code, code history, bugs, emails, ...)

Activity: Identify Programming Challenges

- Form groups of 4
- Based on your past experience, brainstorm programming challenges
    - Try to be specific: what's the user's goal, and what makes it hard?

Activity: Form Project Groups

Questions? Come talk to me!