Course Overview and Study Design

SWE 795, Fall 2019
Software Engineering Environments
Exercise: Modern Development Environments

- What is a feature offered by a development environment?
- How does this help developers work better?
Examples of features

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

• An application that enables software developers to accomplish a software engineering activity.

• Key concepts:
  • Software engineering activity
  • Task
  • Challenge
  • Support
Why study software engineering environments?

- Development environments 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. Overview & conducting studies
2. Design process
3. Problem solving
4. Programming as communication
5. Debugging
6. Crosscutting concerns
7. Software visualization
8. Editing code
9. Preventing defects
10. Code reuse
11. Program synthesis
12. Software analytics
13. Crowdsourcing
14. End-user software engineering
Class format

• Part 1: Lecture
  • Overview of a specific topic
• Part 2: In-Class Activity or Project Presentations
• Part 3: Discussion of readings
  • Discussant introduces paper with brief 5 min summary
  • Discussant moderates class discussion
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
Course Readings

• Will have 3 readings a week
  • Responsible for reading all 3 papers and responding to a prompt on Piazza.
  • Also responsible for serving as discussant for one paper approximately every 3 weeks (see signup)
  • Discussant responsible for 5 min presentation summarizing paper and leading 10 mins of class discussion about paper
• Will have sign up for discussants for class meetings starting with 9/16
The homework in this course will be in the form of a project. All project work will occur in two person groups. HW0 will be submitted as a short written report and a meeting with the course instructor. HW1 - HW4 will take the form of an in-class presentation, where all groups members will give a 10-min presentation on their work.

- HW0: Project Proposal (50 points)
- HW1: Review of Literature (100 points)
- HW2: Study of Current Practice (100 points)
- HW3: Tool Sketch (100 points)
- HW4: Tool Prototype (250 points)
HW0: Project Proposal

• The project proposal should describe a specific aspect of software development that your project will focus on.
• The project proposal should clearly identify a specific challenge software developers experience in programming work, including a scenario describing a situation a developer might face.
• The project proposal should also include (1) a brief description of the type of study you will perform to understand this challenge better and (2) an initial idea of how a tool might address this challenge.
• Expect to be about a page
• Due Tuesday, Sept 3, 12:00pm
• Will schedule meeting to discuss your proposal.
Course grade

• Paper responses: 20%
• Paper discussant: 20%
• Project: 60%
Example: Developing a programming tool
Observations of developers in the field

Participants

17 professional developers

Tasks

picked one of their own coding tasks involving unfamiliar code

~90 minutes

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)
## Longest activities related to control flow questions

### 4 out of the 5 longest investigation activities

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

### 5 out of the 5 longest debugging activities

<table>
<thead>
<tr>
<th>Primary question</th>
<th>Time (mins)</th>
<th>Related control flow question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where is method ( m ) generating an error?</td>
<td>66</td>
<td>Search downstream from ( m ) for <strong>error</strong> text</td>
</tr>
<tr>
<td>What resources are being acquired to cause this deadlock?</td>
<td>51</td>
<td>Search downstream for <strong>acquire</strong> method calls</td>
</tr>
<tr>
<td>“When they have this attribute, they must use it somewhere to generate the content, so where is it?”</td>
<td>35</td>
<td>Search downstream for <strong>reads</strong> of attribute</td>
</tr>
<tr>
<td>“What [is] the test doing which is different from what my app is doing?”</td>
<td>30</td>
<td><strong>Compare</strong> test traces to app traces</td>
</tr>
<tr>
<td>How are these thread pools interacting?</td>
<td>19</td>
<td>Search downstream for <strong>calls</strong> into thread pools</td>
</tr>
</tbody>
</table>
Longest debugging activity

Where is method \( m \) generating an error?

Rapidly found method \( m \) implementing command
Unsure where it generated error

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static call traversal</td>
<td>Statically traversed calls looking for something that would generate error</td>
</tr>
<tr>
<td>debugger</td>
<td>Tried debugger</td>
</tr>
<tr>
<td>grep</td>
<td>Did string search for error, found it, but many callers</td>
</tr>
<tr>
<td>debugger</td>
<td>Stepped in debugger to find something relevant</td>
</tr>
<tr>
<td>static call traversal</td>
<td>Statically traversed calls to explore</td>
</tr>
<tr>
<td>debugger</td>
<td>Went back to stepping debugger to inspect values</td>
</tr>
<tr>
<td></td>
<td>Found the answer</td>
</tr>
</tbody>
</table>

(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 $\cap$ statements matching search criteria
### Longest activities related to reachability questions

**4 out of the 5 longest investigation activities**

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

**5 out of the 5 longest debugging activities**

<table>
<thead>
<tr>
<th>Primary question</th>
<th>Time (mins)</th>
<th>Related reachability question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where is method ( m ) generating an error?</td>
<td>66</td>
<td>Search downstream from ( m ) for <strong>error</strong> text</td>
</tr>
<tr>
<td>What resources are being acquired to cause this deadlock?</td>
<td>51</td>
<td>Search downstream for <strong>acquire</strong> method calls</td>
</tr>
<tr>
<td>“When they have this attribute, they must use it somewhere to generate the content, so where is it?”</td>
<td>35</td>
<td>Search downstream for <strong>reads</strong> of attribute</td>
</tr>
<tr>
<td>“What [is] the test doing which is different from what my app is doing?”</td>
<td>30</td>
<td><strong>Compare</strong> test traces to app traces</td>
</tr>
<tr>
<td>How are these thread pools interacting?</td>
<td>19</td>
<td>Search downstream for <strong>calls</strong> into thread pools</td>
</tr>
</tbody>
</table>
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 questions**
```java
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

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

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

Method

12 developers  
15 minutes to answer **reachability** question \( \times 6 \)

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())?
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

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**: \( F = m \times a \)
- 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

**Evaluation studies**
- (Expensive)
- lab study
- field deployment

**Implement tool**

**Evaluation studies**
- (Cheap)
- heuristic evaluation
- paper prototypes
- participatory design
- ...

LaToza
GMU SWE 795 Spring 2019
(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, ...)

GMU SWE 795 Spring 2019
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

<table>
<thead>
<tr>
<th>Sources I depend on changed?</th>
<th>code</th>
<th>1</th>
<th>9</th>
<th>41</th>
<th>15</th>
<th>15</th>
<th>tools</th>
<th>12</th>
<th>coworker</th>
<th>6</th>
<th>email</th>
<th>4</th>
<th>br</th>
<th>2</th>
<th>code</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>What have caused this behavior?</td>
<td>code</td>
<td>0</td>
<td>2</td>
<td>17</td>
<td>73</td>
<td>20</td>
<td>22</td>
<td>coworker</td>
<td>5</td>
<td>intuition</td>
<td>4</td>
<td>log</td>
<td>4</td>
<td>br</td>
<td>4</td>
<td>debug</td>
</tr>
<tr>
<td>this data structure or function?</td>
<td>code</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>71</td>
<td>20</td>
<td>29</td>
<td>docs</td>
<td>11</td>
<td>code</td>
<td>5</td>
<td>coworker</td>
<td>4</td>
<td>spec</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>code implemented this way?</td>
<td>code</td>
<td>0</td>
<td>2</td>
<td>21</td>
<td>61</td>
<td>37</td>
<td>39</td>
<td>-----</td>
<td>code</td>
<td>4</td>
<td>intuition</td>
<td>4</td>
<td>history</td>
<td>3</td>
<td>coworker</td>
<td>2</td>
</tr>
<tr>
<td>n worth fixing?</td>
<td>code</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>44</td>
<td>10</td>
<td>20</td>
<td>-----</td>
<td>coworker</td>
<td>12</td>
<td>email</td>
<td>2</td>
<td>br</td>
<td>1</td>
<td>intuition</td>
<td>1</td>
</tr>
<tr>
<td>implications of this change?</td>
<td>code</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>85</td>
<td>44</td>
<td>49</td>
<td>-----</td>
<td>coworker</td>
<td>13</td>
<td>log</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Purpose of this code?</td>
<td>code</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>56</td>
<td>24</td>
<td>29</td>
<td>-----</td>
<td>intuition</td>
<td>5</td>
<td>code</td>
<td>2</td>
<td>debug</td>
<td>2</td>
<td>tools</td>
<td>2</td>
</tr>
<tr>
<td>What is closely related to this code?</td>
<td>code</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>66</td>
<td>27</td>
<td>27</td>
<td>-----</td>
<td>tools</td>
<td>8</td>
<td>intuition</td>
<td>2</td>
<td>email</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>state problem?</td>
<td>code</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>49</td>
<td>17</td>
<td>34</td>
<td>-----</td>
<td>br</td>
<td>5</td>
<td>coworker</td>
<td>1</td>
<td>log</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>What team's conventions?</td>
<td>code</td>
<td>0</td>
<td>7</td>
<td>25</td>
<td>41</td>
<td>10</td>
<td>15</td>
<td>-----</td>
<td>docs</td>
<td>2</td>
<td>tools</td>
<td>2</td>
<td>memory</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Debug failure look like?</td>
<td>code</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>88</td>
<td>24</td>
<td>23</td>
<td>-----</td>
<td>br</td>
<td>3</td>
<td>screenshot</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Are these part of this submission?</td>
<td>code</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>61</td>
<td>7</td>
<td>5</td>
<td>-----</td>
<td>tools</td>
<td>2</td>
<td>memory</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Order this with this other code?</td>
<td>code</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>75</td>
<td>28</td>
<td>30</td>
<td>-----</td>
<td>docs</td>
<td>2</td>
<td>code</td>
<td>1</td>
<td>coworker</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>will this problem be to fix?</td>
<td>code</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>41</td>
<td>15</td>
<td>32</td>
<td>-----</td>
<td>code</td>
<td>1</td>
<td>coworker</td>
<td>1</td>
<td>screenshot</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Are used to implement this behavior?</td>
<td>code</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>61</td>
<td>27</td>
<td>22</td>
<td>-----</td>
<td>memory</td>
<td>1</td>
<td>docs</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaction was relevant to my task?</td>
<td>code</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>59</td>
<td>15</td>
<td>13</td>
<td>-----</td>
<td>memory</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
Grade school students asked to describe in prose how PacMan would work in each of several scenarios

- Usually Pacman moves like this.
- Now let’s say we add a wall.
- Pacman moves like this.
- Not like this.

Do this: Write a statement that summarizes how I (as the computer) should move Pacman in relation to the presence or absence of other things.
### 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)

### Overall structure
- 61% Behaviors built into objects
- 20% Direct modification
- 18% Other

### Pictures
- 67% Yes

### AND
- 67% Boolean conjunction
- 29% Sequencing

### OR
- 63% Boolean disjunction
- 24% To clarify or restate a prior item
- 8% “Otherwise”
- 5% Other

### Keywords

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

### Loopy constructs
- 73% Implicit
- 20% Explicit
- 7% Other

### Complex conditionals
- 59% Natural language style — incomplete
- 40% Natural language style — complete

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

### Motions
- 97% Expect continuous motion

### Computation
- 48% Insert first then reposition others
- 26% Insert without making space
- 17% Make space then insert
- 8% Other

### Randomness
- 47% Precision
- 20% Uncertainty without using “random”
- 18% Precision with hedging
- 15% Other

### Insertion into a data structure
- 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
• 104 respondents at Microsoft rated
  % of time on different activities
  Tool use frequency & effectiveness
  Severity of 13 “problems”
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: Form Project Groups
Activity: Identify Programming Challenges

• Form groups of 2
• Open a development environment
• Based on your past experience, brainstorm programming challenges