Course Overview and Study Design

SWE 795, Spring 2017
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 \textbf{not} go into technical details of approaches
  • Focus on \textbf{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. Analyzing data
3. Information needs
4. Debugging
5. Crosscutting concerns
6. Mental models
7. Software visualization
8. Editing code
9. Preventing defects
10. 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
• Part 3: Discussion of readings
  • Discussant introduces paper with brief 5 min summary
  • Discussant moderates class discussion
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 every other week (6 papers in total)
  • 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 2/7
Project

• The homework in this course will be in the form of a project. All project work will occur in two person groups. Rather than creating a written report, each HW assignment be 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: Study of Current Practice (100 points)
• HW2: Tool Sketch (100 points)
• HW3: Tool Prototype (250 points)
• HW4: Tool Evaluation (100 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.
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

Transcripts

Interested. 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 $status$ 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>Question</th>
<th>Time (mins)</th>
<th>Activity</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 \textit{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 \textbf{search} for error, found it, but many callers</td>
</tr>
<tr>
<td>debugger</td>
<td>\textbf{Stepped} in debugger to find something relevant</td>
</tr>
<tr>
<td>static call traversal</td>
<td>Statically \textbf{traversed} calls to explore</td>
</tr>
<tr>
<td>debugger</td>
<td>Went back to \textbf{stepping} debugger to inspect values 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 $e$)
### 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>Question</th>
<th>Time (mins)</th>
<th>Activity</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**
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);

    return getFoldLevel(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
### 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: \(\sim 90\%\) 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?
How do developers debug?

• 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
- ...

(Expensive) evaluation studies

- lab study
- field deployment

Generate tool

- designs
- scenarios
- mockups

(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

| Sources 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 debug 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 debug 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 |
|purpose of this code? | 1 1 5 56 24 29 | intuition 5 code 2 debug 2 tools 2 spec 1 docs 1 |
|illy related to this code? | 0 1 7 66 27 27 | tools 8 intuition 2 email 1 |
|name 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 |
|s are part of this submission? | 0 2 3 61 7 3 | tools 2 memory 2 |
|ordinate this with this other code? | 1 1 4 75 28 30 | docs 2 code 1 coworker 1 |
|ill 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 |
|ission 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

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

Overall structure
- 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

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

Computation
- Mathematics 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”

<table>
<thead>
<tr>
<th>Activity</th>
<th>0%</th>
<th>1%</th>
<th>2%</th>
<th>3%</th>
<th>4%</th>
<th>5%</th>
<th>6%</th>
<th>7%</th>
<th>8%</th>
<th>9%</th>
<th>10%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examining source code</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Examining source code check-in comments and diffs</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Examining high level views of source code (UML diagrams, class hierarchies, call graphs, …)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Examining the code and looking at the results</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Running the code and examining it with a debugger</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Using debug or trace statements</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Other</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

Tools for understanding code

- Profiler
- Other
- SQL editor
- Diff tool
- Other debugger
- Source Insight
- Visual Studio debugger
- Visual Studio editor
- Other
- Notepad
- Edit

% of time understanding code

Effectiveness (1=Low, 7=High)

- Strongly disagree
- Disagree
- Somewhat disagree
- Neutral
- Somewhat agree
- Agree
- Strongly agree

This technique was effective for understanding existing code

Examining source code
Examining source code check-in comments and diffs
Examining high level views of source code (UML diagrams, class hierarchies, call graphs, …)
Examining 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**
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, ...)
Cheap evaluation studies

• You have a tool idea
  with scenarios of how it would be used
  and mockups of what it would look like

• You could spend 2 yrs building a static analysis to
  implement tool
  But is this the right tool? Would it really help?
  Which features are most important to implement?

• Solution: cheap evaluation studies
  Evaluate the mockup before you build the tool!
  Tool isn’t helpful: come up with new idea
  Users have problems using tool: fix the problems
(Some) types of cheap evaluation studies

- **Empirical studies** (w/ users)
  - Paper prototyping
    - Do tasks on paper mockups of real tool
    - Simulate tool on paper
  - Wizard of oz
    - Simulate tool by computing results by hand
- **Analytical techniques** (no users)
  - Heuristic evaluation / cognitive dimensions
    - Assess tool for good usability design
  - Cognitive walkthrough
    - Simulate actions needed to complete task
Paper prototyping

• Build paper **mockup** of tool before building real version
  May be rough sketch or realistic screenshots
• Experimenter **simulates** tool by adding / changing papers
  May have cutouts for menus, scrolling, screen objects
• Good for checking if user
  Understands interface **terminology**
  Commands users want **match** actual commands
  Able to understand what tool does
  Whether **information** provided by tool helps
• Challenges - must **anticipate** commands used
  Iteratively add commands from previous participants
  Prompt users to try it a different way
• Challenges:
  Must anticipate user questions beforehand

• Hard to do when many possible questions developers could ask
Wizard of oz

- Participant believes (or pretends) to interact with real tool
  - Experimenter simulates (behind the curtain) tool
  - Computes data used by tool by hand

- Original example
  - Voice user interface
  - Experimenter translates speech to text

- Advantages
  - High fidelity - user can use actual tool before it’s built

- Disadvantages
  - Requires working GUI, unlike paper prototypes
Types of prototypes

- **Paper**
  - “Low fidelity prototyping”
  - Often surprisingly effective
  - Experimenter plays the computer
  - Drawn on paper → drawn on computer

- **“Wizard of Oz”**
  - User’s computer is “slave” to experimenter’s computer
    - Experimenter provides the computer’s output
  - “Pay no attention to that man behind the curtain”
  - Especially for AI and other hard-to-implement systems

- **Implemented Prototype**
  - Visual Basic
  - Adobe (MacroMind) Flash and Director
  - Visio
  - PowerPoint
  - Web tools (even for non-web UIs)
    - Html
    - Scripting
  - (no database)

- **Real system**

- **Better if sketchier for early design**
  - Use paper or “sketchy” tools, not real widgets
  - People focus on wrong issues: colors, alignment, names
  - Rather than overall structure and fundamental design
Heuristic evaluation [Nielsen]

• Multiple evaluators use dimensions to identify usability problems
  Evaluators aggregate problems & clarify
• 1. Visibility of system **status** - keep users informed
• 2. **Match** between system & real world
  Speak users language, follow real world conventions
• 3. User control & **freedom** - undo, redo, don’t force down paths
• 4. **Consistency** & standards
  Words, situations, actions should mean same in similar situations
• 5. **Error** prevention - prevent illegal actions
  E.g., gray out or remove buttons user can’t use
Heuristic evaluation [Nielsen]

• 6. **Recognition** rather than recall - impt for infreq commands
   Select commands to perform rather than remember command
   Recognition: menus  Recall: command line interface

• 7. Flexibility & **efficiency** of use - make frequent actions fast
   Eg., keyboard accelerators, macros

• 8. Aesthetic & **minimalist** design - remove irrelevant information
   More clutter = harder to do visual search

• 9. Help users recognize, diagnose, & recover from **errors**
   Error message in language user understands
   Precisely indicate problem, suggest solution

• 10. **Help** & documentation
   Easy to search, task focused, concrete steps to take
   Always available
Cognitive dimensions of notations [Green & Blackwell]

- Dimensions for structuring assessment based on experience
- **Visibility** & juxtaposability
  - What is difficult to see or find?
  - If need to compare or combine parts, can see at same time?
- **Viscosity** - how hard is it to change?
- **Diffuseness** - brief or long winded?
- Hard **mental** operations - what requires most mental effort?
- **Error** proneness - are there common mistakes that irritate?
- Closeness of **mapping** - how close is notation to what is described?
- Role **expressiveness** - are parts easy to interpret?
Cognitive dimensions of notations [Green & Blackwell]

- **Hidden dependencies**
  Are changes to one part which affect others apparent? Do some actions cause dependencies to freeze?

- **Progressive evaluation** - can see progress, stop and check work? Can you try out partially completed versions?

- **Provisionality** - can sketch or try things out when playing with ideas?

- **Premature commitment** - are actions only possible in a specific order? Do users have enough information to choose correct actions?

- **Consistency** - do parts with similar meaning look similar? Are parts that are the same shown in different ways?

- **Secondary notation** - is it possible to write notes to yourself?

- **Abstraction management** - can you define your own elements?
Cognitive walkthrough

• Determine the correct **sequence** of actions to perform task
  Build mockups (screenshot) of each step
• For each step, write analysis:
  • 1. Will user try to **achieve** correct effect?
      Will user have the correct goal?
  • 2. Will user **notice** correct action is available?
      Will user be likely to see the control?
  • 3. Will user **associate** correct action w/ effect trying to achieve?
      After users find control, will they associate with desired effect?
  • 4. If correct action performed, will user see progress to solution?
      Will users understand the feedback?
Exercise: What study(s) would you use?

How would you design a study(s) in these situations?

1. You’re designing a tool for a new notation for visualizing software.

2. You’re designing a specification language for finding bugs.
(Some) types of cheap evaluation studies

- **Empirical studies** *(w/ users)*
  - Paper prototyping
    - Do tasks on paper mockups of real tool
    - Simulate tool on paper
  - Wizard of oz
    - Simulate tool by computing results by hand
- **Analytical techniques** *(no users)*
  - Heuristic evaluation / cognitive dimensions
    - Assess tool for good usability design
  - Cognitive walkthrough
    - Simulate actions needed to complete task
Evaluation studies

- You’ve built a tool
  You want to write a paper claiming it’s useful.
  You want to get a company to try it out.

- Solution: run an evaluation study
  Cheap evaluation study
  (Less cheap, but more convincing) evaluation study
(Some) types of evaluation studies

- (Cheap) evaluation studies
- **Lab** experiments - controlled experiment between tools
  Measure differences of your tool w/ competitors
  Strongest quantitative evidence
- **Field** deployments
  Users try your tool in their own work
  Data: usefulness perceptions, how use tool
  Usually more qualitative
Lab studies

• Users complete **tasks** using your tool or competitors
  Within subjects design - all participants use both
  Between subjects design - participants use one

• Typical **measures** - time, bugs, quality, user perception
  Also measures from exploratory observations (think-aloud)
  More detailed measures = better understand results

• Advantages - controlled **experiment**! (few confounds)

• Disadvantages - lower **external** validity
  Users still learning how to use tool, unfamiliar with code
  Benefits may require longer task
Ko & Myers CHI09

- **20** masters students did two 30 minute tasks
- Used **tutorial** to teach the tool to users
- Tasks: **debug** 2 real bug reports from ArgoUML  
  Diagnose problem & write change recommendation
- **Measured** time, success, code exploration, perception

### Results

<table>
<thead>
<tr>
<th>Task 1</th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td># successful</td>
<td>time (min)</td>
</tr>
<tr>
<td>whyline</td>
<td>control</td>
</tr>
<tr>
<td><img src="chart1.png" alt="Bar chart" /></td>
<td><img src="chart2.png" alt="Bar chart" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>Task 1</th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>whyline</td>
<td>control</td>
<td>whyline</td>
</tr>
<tr>
<td># of unique source files viewed per minute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>1.8</td>
<td>1.3</td>
</tr>
<tr>
<td>σ²</td>
<td>1.4</td>
<td>0.8</td>
</tr>
<tr>
<td>range of files viewed</td>
<td>median</td>
<td>mean</td>
</tr>
<tr>
<td>8 – 39</td>
<td>10 – 66</td>
<td>16 – 72</td>
</tr>
<tr>
<td>distance to key function</td>
<td>median</td>
<td>mean</td>
</tr>
<tr>
<td>2.2</td>
<td>3.4</td>
<td>3.6</td>
</tr>
<tr>
<td># why did questions (median, range)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2, 1–4</td>
<td>-</td>
<td>4, 1–8</td>
</tr>
<tr>
<td># why didn’t questions (median, range)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0, 0–0</td>
<td>-</td>
<td>0, 0–2</td>
</tr>
<tr>
<td>median # debugger steps taken</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>median # text searches</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.5</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>
Field deployments

• Generally **not** controlled comparison
  Can’t directly compare your tool against others
  Different tasks, users, code

• Give your tool to developers. See how **they** use it

• **Data** collection: interviews, logging data, observations

• **Qualitative** measures
  - **Perception**: do they like the tool?
  - **Use frequency**: how often do they use it?
  - **Uses**: how do they use it? what questions? tasks? why?
  - **Wishes**: what else would they like to use it for?

• **Quantitative** comparison possible but hard
Cherubini, Venolia, & DeLine VL/HCC07

- Build large code map to be used for meetings & discussions
- Hypotheses: could be used for
  1. understanding new features in code
  2. reengineering parts of the code
  3. transferring knowledge to new developers
- Field deployment of map for 1 month
- Only 2 newcomers used it!
  Too many or too few details for discussions
  Sometimes wrong information (call graph vs inheritance)
  Layout was static & couldn’t be changed
- Developers instead made extensive use of whiteboard
Designing an evaluation study

• 1. What is your research question? What do you want to learn?
   Write a paper abstract with your ideal results
• 2. What type of study will you conduct?
• 3. Who will participate? Undergrads, graduate students, professionals?
   Closer to your target population is better
   Where will you recruit them from?
   What incentive to participate: $$, class credit, friends, ...
• 4. What tasks will they perform?
   Tasks should demonstrate tool’s benefits.
• 5. What data will you collect?
   think aloud, post task interviews, ...
   screen, audio, video recording
• [6. Get Institutional Review Board (IRB) approval]
Learning a new tool

• Study participants will not know how to use your tool.
• Solution: tutorial of your tool
• What to cover:
  Important features, commands of tool
  What visualizations, notations mean
  What questions does tool let user answer?
  Example task done with tool
• Use both text & hands on exercises
• Let user ask experimenter questions
Piloting

• Most **important** step in ensuring useful results!
  
• (1) Run study on **small** (1 - 4) number of participants
  
• (2) Fix **problems** with study design
  Was the tool tutorial sufficient?  
  Did tasks use your tool? Enough?  
  Did they understand your questions? (esp surveys)  
  Did you collect the right data?  
  Are your measures correct?

(3) Fix **usability** problems
  Are developers doing the “real” task, or messing with tool?  
  Are users confused by terminology in tool?  
  Do supported commands match commands users expect?

• (4) **Repeat** 1, 2, and 3 until no more (serious) problems
For more information

• **Field observations, ethnography, interviews, artifact studies, qualitative methods**  Michael Quinn Patton. (2002). Qualitative Research & Evaluation Methods. Sage Publications.


Activity: Identify Programming Challenges

- Form groups of 2
- Open a development environment
- Based on your past experience, brainstorm programming challenges
Activity: Form Project Groups