Using exploratory & evaluation studies

Thomas LaToza

05-899D: Human Aspects of Software Development (HASD)

Spring, 2011
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
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

Implement tool

Generate tool
- designs
- scenarios
- mockups

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

...
(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
Looked for **questions** developers asked

<table>
<thead>
<tr>
<th>information type</th>
<th>search times</th>
<th>% agreed info is...</th>
<th>frequency and outcome of searches</th>
<th>frequency of sources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>mid</td>
<td>max</td>
<td>import.</td>
</tr>
<tr>
<td>s1 Did I make any mistakes in my new code?</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>59</td>
</tr>
<tr>
<td>a2 What have my coworkers been doing?</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>17</td>
</tr>
<tr>
<td>u3 What code caused this program state?</td>
<td>0</td>
<td>2</td>
<td>21</td>
<td>90</td>
</tr>
<tr>
<td>r2 In what situations does this failure occur?</td>
<td>0</td>
<td>2</td>
<td>49</td>
<td>80</td>
</tr>
<tr>
<td>d2 What is the program <strong>supposed</strong> to do?</td>
<td>0</td>
<td>1</td>
<td>21</td>
<td>93</td>
</tr>
<tr>
<td>a1 How have resources I depend on changed?</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>41</td>
</tr>
<tr>
<td>u1 What code <strong>could</strong> have caused this behavior?</td>
<td>0</td>
<td>2</td>
<td>17</td>
<td>73</td>
</tr>
<tr>
<td>c2 How do I use this data structure or function?</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>71</td>
</tr>
<tr>
<td>d3 Why was this code implemented this way?</td>
<td>0</td>
<td>2</td>
<td>21</td>
<td>61</td>
</tr>
<tr>
<td>b3 Is this problem worth fixing?</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>44</td>
</tr>
<tr>
<td>d4 What are the implications of this change?</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>85</td>
</tr>
<tr>
<td>d1 What is the purpose of this code?</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>56</td>
</tr>
<tr>
<td>u2 What's statically related to this code?</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>66</td>
</tr>
<tr>
<td>b1 Is this a legitimate problem?</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>49</td>
</tr>
<tr>
<td>s2 Did I follow my team's conventions?</td>
<td>0</td>
<td>7</td>
<td>25</td>
<td>41</td>
</tr>
<tr>
<td>r1 What does the failure look like?</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>88</td>
</tr>
<tr>
<td>s3 Which changes are part of this submission?</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>61</td>
</tr>
<tr>
<td>c3 How I can coordinate this with this other code?</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>75</td>
</tr>
<tr>
<td>b2 How difficult will this problem be to fix?</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>41</td>
</tr>
<tr>
<td>c1 What can be used to implement this behavior?</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>61</td>
</tr>
<tr>
<td>a3 What information was relevant to my task?</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>59</td>
</tr>
</tbody>
</table>
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.
<table>
<thead>
<tr>
<th>Programming style</th>
<th>Overall structure</th>
<th>Modifying state</th>
</tr>
</thead>
<tbody>
<tr>
<td>54% Production rules/events</td>
<td>45% Player or end-user</td>
<td>61% Behaviors built into objects</td>
</tr>
<tr>
<td>18% Constraints</td>
<td>34% Programmer</td>
<td>20% Direct modification</td>
</tr>
<tr>
<td>16% Other (declarative)</td>
<td>20% Other (third-person)</td>
<td>18% Other</td>
</tr>
<tr>
<td>12% Imperative</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>67% Boolean conjunction</td>
<td>63% Boolean disjunction</td>
<td>66% Sequencing</td>
</tr>
<tr>
<td>29% Sequencing</td>
<td>24% To clarify or restate a prior item</td>
<td>32% “Consequently” or “in that case”</td>
</tr>
<tr>
<td></td>
<td>8% “Otherwise”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5% Other</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations on multiple objects</th>
<th>Complex conditionals</th>
<th>Control structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>95% Set/subset specification</td>
<td>37% Set of mutually exclusive rules</td>
<td>37% Implicit</td>
</tr>
<tr>
<td>5% Loops or iteration</td>
<td>27% General case, with exceptions</td>
<td>20% Explicit</td>
</tr>
<tr>
<td></td>
<td>23% Complex boolean expression</td>
<td>7% Other</td>
</tr>
<tr>
<td></td>
<td>14% Other (additional uses of exceptions)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Remembering state</th>
<th>Mathematical operations</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>56% Present tense for past event</td>
<td>59% Natural language style — incomplete</td>
<td>97% Expect continuous motion</td>
</tr>
<tr>
<td>19% “After”</td>
<td>40% Natural language style — complete</td>
<td></td>
</tr>
<tr>
<td>11% State variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6% Discuss future events</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5% Past tense for past event</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tracking progress</th>
<th>Randomness</th>
</tr>
</thead>
<tbody>
<tr>
<td>85% Implicit</td>
<td>47% Precision</td>
</tr>
<tr>
<td>14% Maintain a state</td>
<td>20% Uncertainty without using “random”</td>
</tr>
<tr>
<td></td>
<td>18% Precision with hedging</td>
</tr>
<tr>
<td></td>
<td>15% Other</td>
</tr>
</tbody>
</table>
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”

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

Examine source code
- Examine source code check-in comments and diffs
- Examine high-level views of source code (UML diagram, class hierarchy, 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

| Percent | 0% | 1% | 2% | 3% | 4% | 5% | 6% | 7% | 8% | 9% | 10%
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>1%</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>2%</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>3%</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>4%</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>5%</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>6%</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>7%</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>8%</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>9%</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>10%</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

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

40. This technique was effective for understanding existing code

Examine source code
- Examine source code check-in comments and diffs
- Examine high-level views of source code (UML diagram, class hierarchy, 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

Effectiveness (1=Low, 7=High)

Tools for understanding code

- Visual Studio debugger
- Visual Studio editor
- Source Insight
- Other debugger
- Emacs
- Notepad
- SQL editor
- V1
- Other
- Diff tool
- Profiler
- SlickEdit
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?
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.
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, ...)

(Some) types of exploratory studies
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
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 \(\rightarrow\) 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
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
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
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</th>
<th># successful</th>
<th>time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>task 1</th>
<th>task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td># of unique source files viewed per minute mean</td>
<td>1.8</td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td>control</td>
<td></td>
</tr>
<tr>
<td>range of files viewed median distance to key function mean</td>
<td>8 - 39</td>
<td>10 - 66</td>
</tr>
<tr>
<td></td>
<td>control</td>
<td>2.2</td>
</tr>
<tr>
<td># why did questions (median, range) mean</td>
<td>2, 1-4</td>
<td>4, 1-8</td>
</tr>
<tr>
<td></td>
<td>control</td>
<td></td>
</tr>
<tr>
<td># why didn't questions (median, range) median # debugger steps taken median # text searches</td>
<td>0, 0-0</td>
<td>9</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
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
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
Universities have an **Institutional Review Board** (IRB) responsible for ensuring human subjects treated ethically

Before conducting a human subjects study

- Must complete human subjects **training** (first time only)
- Submit an application to IRB for **approval** (2 - ??? weeks approval time)

During a study

- Must administer “**informed consent**” describing procedures of study and any risks to participants

For more information


Questions?