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?

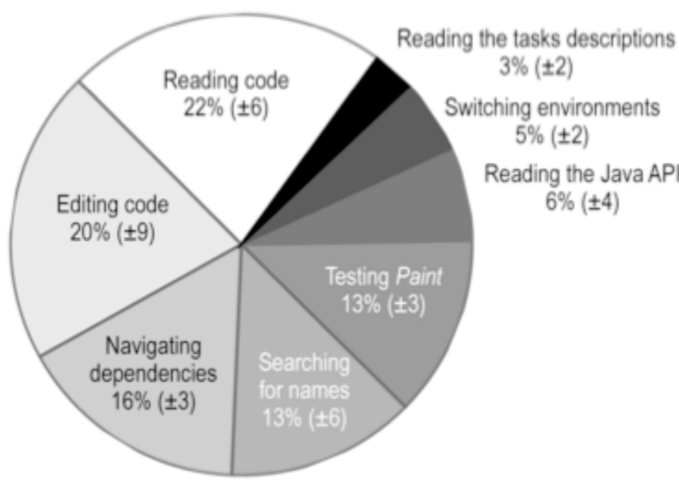
Estimates are biased (time, difficulty)

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



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: $F = m * a$

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

-describes a mechanism by which something happens

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

Empirical vs. analytical generalizability in HASD

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

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

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

Types of studies

Exploratory studies

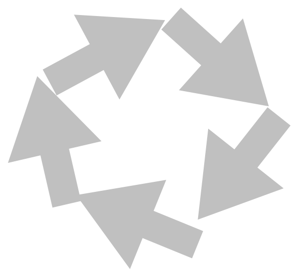
survey
indirect observation
contextual inquiry
...

Models

questions
information needs
use of time
....

Generate tool designs

scenarios
mockups



(Expensive) evaluation studies

lab study
field deployment

(Cheap) evaluation studies

heuristic evaluation
paper prototypes
participatory design
...

Implement tool

(Some) types of exploratory studies

Field observations / ethnography

Observe developers at work in the field

Natural programming

Ask developers to naturally complete a task

Contextual inquiry

Ask questions while developers do work

Surveys

Ask **many** developers specific questions

Interviews

Ask a **few** developers **open-ended** questions

Indirect observations (artifact studies)

Study artifacts (e.g., code, code history, bugs, emails, ...)

Field observations / ethnography

Find software developers

- Pick developers likely to be doing relevant work

Watch developers do **their** work in their office

Ask developers to **think-aloud**

- Stream of consciousness: whatever they are thinking about
- Thoughts, ideas, questions, hypotheses, etc.

Take notes, audio record, or video record

- More is more invasive, but permits detailed analysis
- Audio: can analyze tasks, questions, goals, timing
- Video: can analyze navigation, tool use, strategies
- Notes: high level view of task, interesting observations

Ko, DeLine, & Venolia ICSE07

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

Looked for **questions** developers asked

	information type	search times min	mid	max	% agreed info is... import.	unavail.	inacc.	frequency and outcome of searches (acquired, deferred, gave up, beyond obs.)	frequency of sources (br = bug report, dbug = debugger)
s1	Did I make any mistakes in my new code?	0	1	6	59	7	12		*dbug* 10 *compile* 26 *intuition* 6 *unit test* 4
a2	What have my coworkers been doing?	0	1	11	17	10	10		**coworker** 20 *email* 13 *tool* 4 *bug alert* 4 *im* 2
u3	What code caused this program state?	0	2	21	90	49	32		*dbug* 16 **br** 3 *intuition* 3 *log* 3 *tools* 3 *code* 2 **coworker** 1
r2	In what situations does this failure occur?	0	2	49	80	32	20		**br** 8 **coworker** 8 *inference* 5 *tools* 3 *dbug* 2 *comment* 1
d2	What is the program *supposed* to do?	0	1	21	93	29	29		*spec* 13 **coworker** 9 *docs* 5 *email* 1
a1	How have resources I depend on changed?	0	1	9	41	15	15		*tools* 12 **coworker** 6 *email* 4 **br** 2 *code* 1
u1	What code *could* have caused this behavior?	0	2	17	73	20	22		**coworker** 5 *intuition* 4 *log* 4 **br** 4 *dbug* 2 *im* 1 *code* 1 *spec* 1
c2	How do I use this data structure or function?	0	1	14	71	20	29		*docs* 11 *code* 5 **coworker** 4 *spec* 1
d3	Why was this code implemented this way?	0	2	21	61	37	39		*code* 4 *intuition* 4 *history* 3 **coworker** 2 *dbug* 2 *tools* 2 *comment* 1 **br** 1
b3	Is this problem worth fixing?	0	2	6	44	10	20		**coworker** 12 *email* 2 **br** 1 *intuition* 1
d4	What are the implications of this change?	0	2	9	85	44	49		**coworker** 13 *log* 1
d1	What is the *purpose* of this code?	1	1	5	56	24	29		*intuition* 5 *code* 2 *dbug* 2 *tools* 2 *spec* 1 *docs* 1
u2	What's statically related to this code?	0	1	7	66	27	27		*tools* 8 *intuition* 2 *email* 1
b1	Is this a legitimate problem?	0	1	2	49	17	34		**br** 5 **coworker** 1 *log* 1
s2	Did I follow my team's conventions?	0	7	25	41	10	15		*docs* 2 *tools* 2 *memory* 1
r1	What does the failure look like?	0	0	2	88	24	23		**br** 3 *screenshot* 2
s3	Which changes are part of this submission?	0	2	3	61	7	5		*tools* 2 *memory* 2
c3	How I can coordinate this with this other code?	1	1	4	75	28	30		*docs* 2 *code* 1 **coworker** 1
b2	How difficult will this problem be to fix?	2	2	4	41	15	32		*code* 1 **coworker** 1 *screenshot* 1
c1	What can be used to implement this behavior?	2	2	2	61	27	22		*memory* 1 *docs* 1
a3	What information was relevant to my task?	1	1	1	59	15	13		*memory* 2

Natural programming

Design a simple programming task for users

Ask them to write solution **naturally**

make up language / APIs / notation of interest

Analyze use of **language** in solutions

Advantages:

elicits the language developers expect to see

open-ended - no need to pick particular designs

lets developer design language

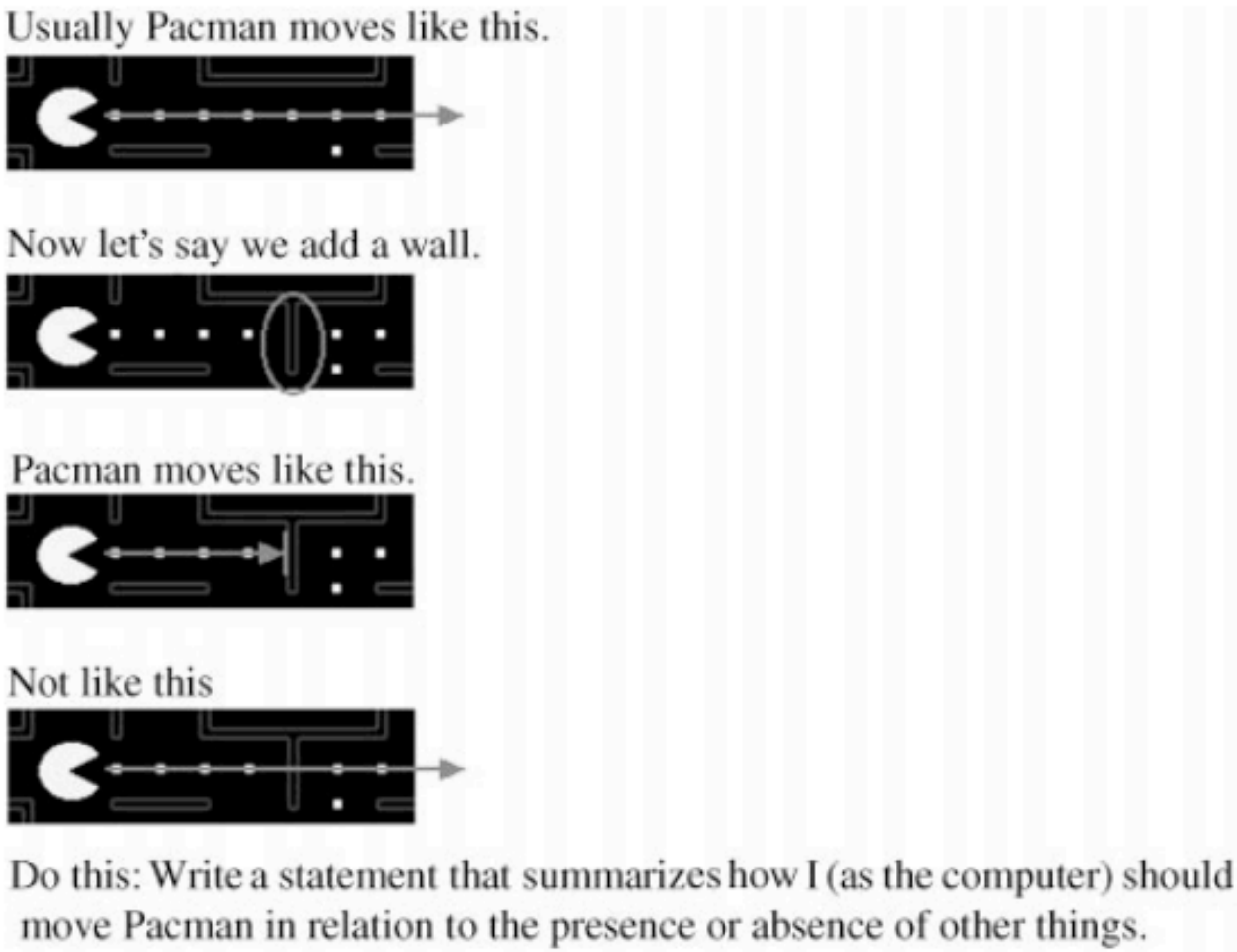
Disadvantages:

assumes the user's notation is best

lets developer design notation

Pane, Ratanamahatana, & Myers ‘01

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



Pane, Ratanamahatana, & Myers IJHCS01

Overall structure

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

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

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

Pictures
67% Yes

Keywords

AND
67% Boolean conjunction
29% Sequencing

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

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

Control structures

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

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

Looping constructs
73% Implicit
20% Explicit
7% Other

Computation

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

Tracking progress
85% Implicit
14% Maintain a state

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

Motions
97% Expect continuous motion

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

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

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

Surveys

Can reach **many** (100s, 1000s) developers

- Websites to run surveys (e.g., SurveyMonkey)

Find **participants** (usually mailing lists)

Prepare multiple choice & free response **questions**

- Multiple choice: faster, standardized response
- Free response: more time, more detail, open-ended

Background & **demographics** questions

- E.g., experience, time in team, state of project, ...

Study questions

Open comments

La Toza, Venolia, & DeLine ICSE06

104 respondents at Microsoft rated

- % of time on different activities
- Tool use frequency & effectiveness
- Severity of 13 "problems"

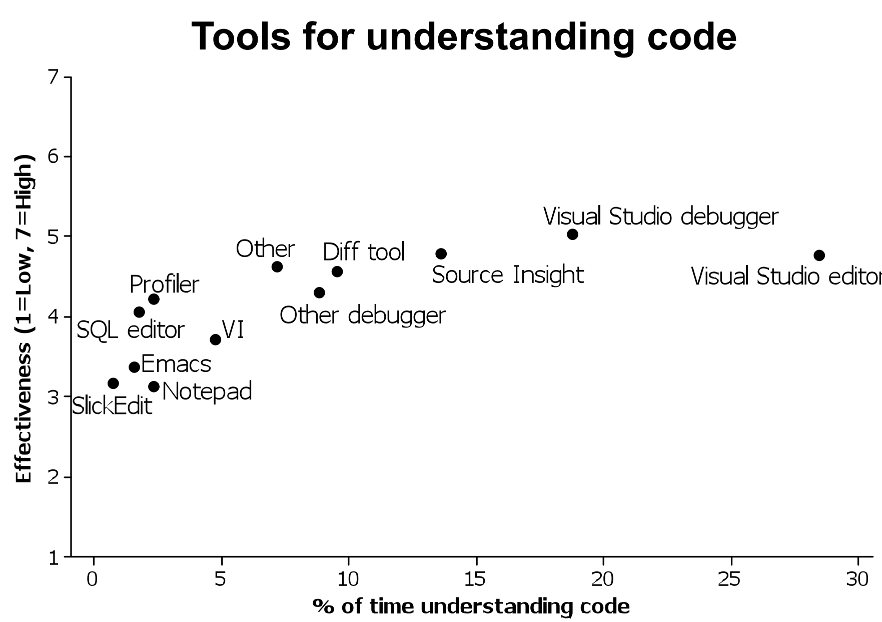
38. Of the time I spent understanding existing code last week, the percent of time I spent

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

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

40. This technique was effective for understanding existing code

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



Semi-structured interviews

Develop a list of focus areas
- Sets of questions related to topics

Prompt developer with question on focus areas
- Let developer talk at length
- Follow to lead discussion towards interesting topics

Manage time
- Move to next topic to ensure all topics covered

Contextual inquiry [Beyer & Holtzblatt]

Interview **while** doing field observations

Learn about environment, work, tasks, culture, breakdowns

Principles of contextual inquiry

- **Context** - understand work in natural environment
  - Ask to see current work being done
  - Seek concrete data - ask to show work, not tell
  - **Bad**: usually, generally **Good**: Here's how I, Let me show you
- **Partnership** - close collaboration with user
  - Not interviewer, interviewee! User is the expert.
  - Not host / guest. Be nosy - ask questions.
- **Interpretation** - make sense of work activity
  - Rephrase, ask for examples, question terms & concepts
- **Focus** - perspective that defines questions of interest

Read Beyer & Holtzblatt book before attempting this study

Indirect observations

Indirect record of developer activity

Examples of **artifacts** (where to get it)

- Code (open source software (OSS) codebases)
- Code changes (CVS / subversion repositories)
- Bugs (bug tracking software)
- Emails (project mailing lists, help lists for APIs)

Collect data from instrumented tool (e.g., code navigation)

Advantages:

- **Lots** of data, easy to obtain
- Code, not developer activity

Disadvantages:

- Can't observe developer **activity**

Malayeri & Aldrich, ESOP09

Gathering data for usefulness of language feature

Structure of study

1. Make **hypotheses** about how code would benefit.
2. Use program analysis to measure **frequency** of idioms in corpus of codebases.
3. Have **evidence** that code would be **different** with approach.
4. **Argue** that different code would make developers more productive.

Example of research questions / hypotheses

1. Does the body of a method only use subset of parameters?
    - Structural types could make more general
    - Are there common types used repeatedly?
2. How many methods throw unsupported operation exception?

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

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

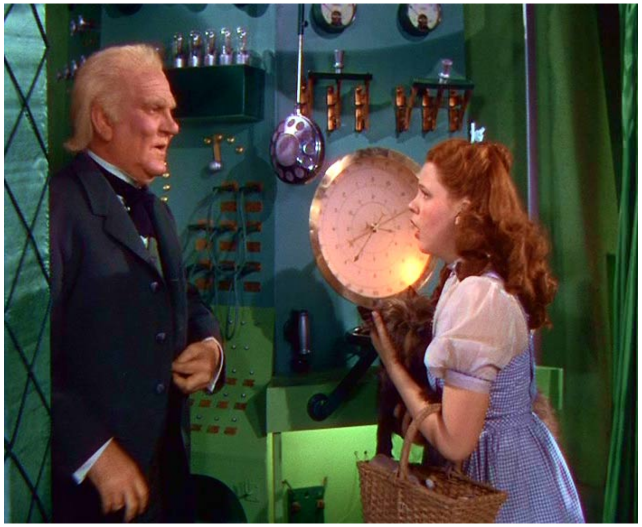
Disadvantages

- Requires **working** GUI, unlike paper prototypes

Types of prototypes

Increasing fidelity

- **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

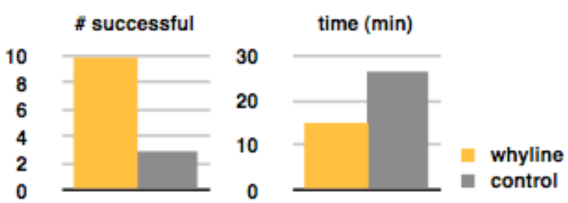
Used **tutorial** to teach the tool to users

Tasks: **debug** 2 real bug reports from ArgoUML
Diagnose problem & write change recommendation

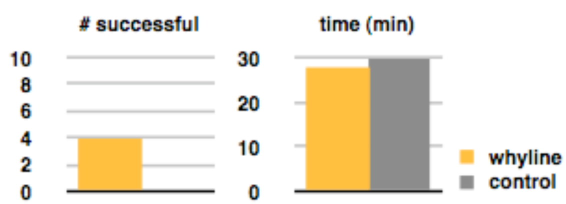
Measured time, success, code exploration, perception

Results

Task 1



Task 2



		task 1		task 2	
		whyline	*control*	*whyline*	*control*
# of unique source files viewed per minute	*mean*	1.8	13.3	1	0.6
	σ^2	1.4	0.8	0.5	0.4
range of files viewed		8 – 39	10 – 66	16 – 72	6 – 44
median distance to key function	*mean*	2.2	3.4	3.6	3.3
	σ^2	0.6	0.5	0.5	0.5
# *why did* questions (median, range)		2, 1–4	–	4, 1–8	–
# *why didn't* questions (median, range)		0, 0–0	–	0, 0–2	–
median # debugger steps taken		–	9	–	14.5
median # text searches		0.5	7	1	8

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 develoers

Field deployment of map for 1 month

Only **2** newcomers used it!

- Too many or too few details for discusssions
- 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

IRB Approval

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

See http://www.cmu.edu/osp/regulatory-compliance/human-subjects.html

For more information

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

Natural programming John F. Pane, Chotirat "Ann" Ratanamahatana, and Brad A. Myers, "Studying the language and structure in non-programmers solutions to programming problems", *International Journal of Human-Computer Studies (IJHCS)*. Special Issue on Empirical Studies of Programmers, vol. 54, no. 2, February 2001, pp. 237-264.

Contextual inquiry Beyer, H. and Holtzblatt, K. 1997. Contextual Design: Defining Customer-Centered Systems. Morgan Kaufman.

Quantitative methods, experiment design, surveys Robert Rosenthal & Ralph Rosnow. (2007). Essentials of Behavioral Research: Methods and Data Analysis. McGraw-Hill.

Qualitative methods applied to SE Carolyn B. Seaman. 1999. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Trans. Softw. Eng.* 25, 4 (July 1999), 557-572.

Wizard of Oz David Maulsby, Saul Greenberg and Richard Mander. “Prototyping an Intelligent Agent through Wizard of Oz,” *Human Factors in Computing Systems,* Proceedings INTERCHI'93. Amsterdam, The Netherlands, Apr, 1993. pp. 277-284.

Sketching and Prototyping Bill Buxton. 2007. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Heuristic evaluation *Nielsen, J., Enhancing the explanatory power of usability heuristics, CHI'94 Conference Proceedings, (1994).*

Cognitive walkthrough C. Wharton et al. "The cognitive walkthrough method: a practitioner's guide" in J. Nielsen & R. Mack "Usability Inspection Methods" pp. 105-140.

Cognitive dimensions of notations Thomas R. G. Green, Marian Petre. (1996). Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. J. Vis. Lang. Comput. 7(2): 131-174.

Questions?