NPRG075

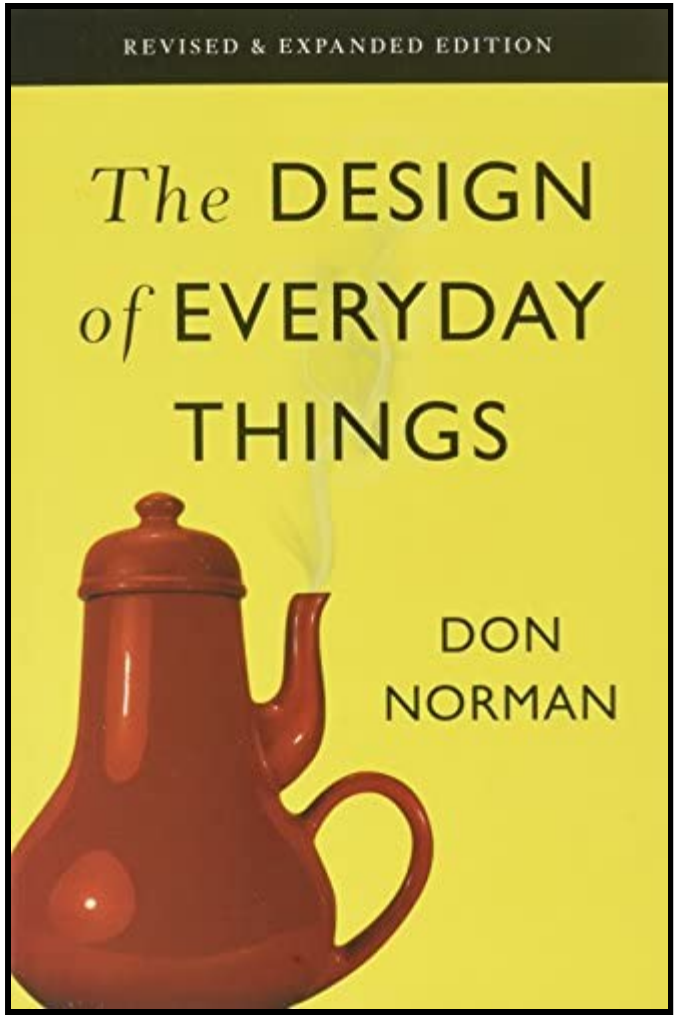
Human-centric language design

Tomáš Petříček, 204 (2nd floor)
petricek@d3s.mff.cuni.cz
https://tomasp.net | @tomaspetricek

Lectures: Tuesday 12:20, S6
https://d3s.mff.cuni.cz/teaching/nprg075

Research methods

Human-computer interaction



HCI perspective

Are programming languages user interfaces?

The means by which the user and a computer system interact (...)

Shifts focus on users and interaction

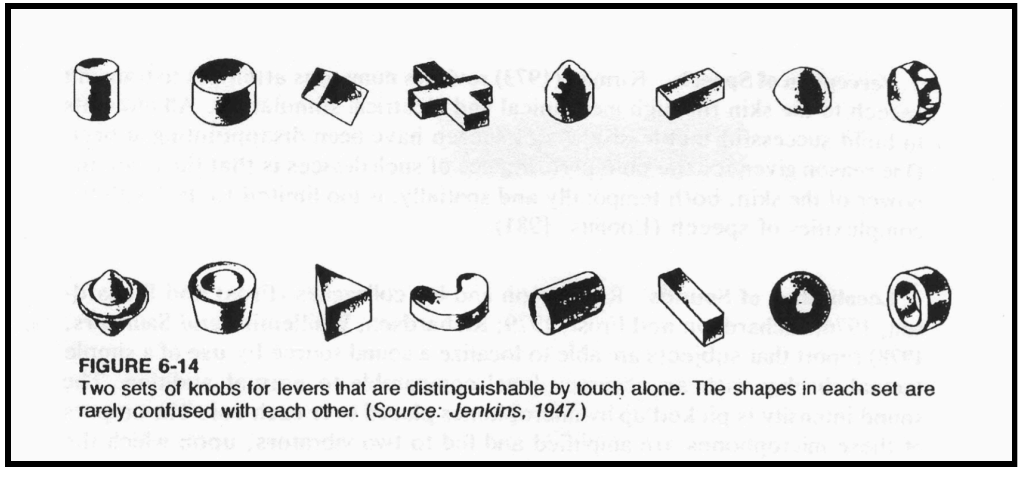


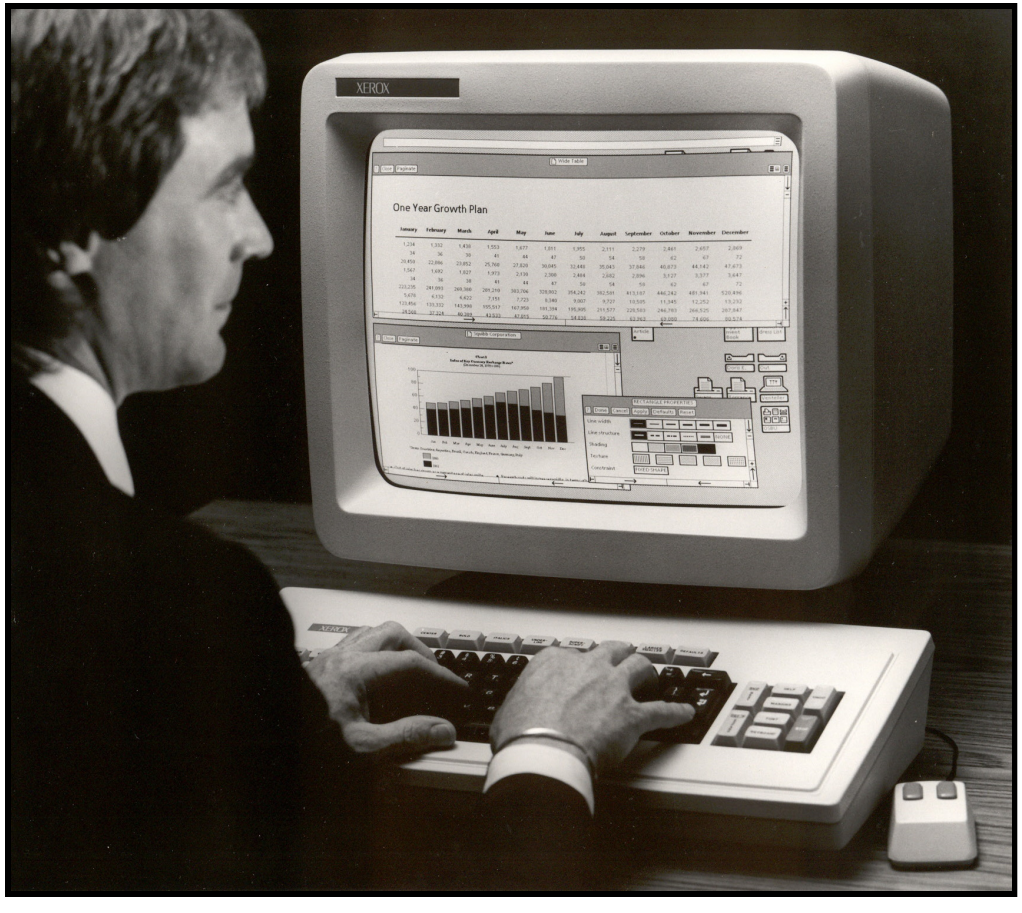
FIGURE 6-14
Two sets of knobs for levers that are distinguishable by touch alone. The shapes in each set are rarely confused with each other. (*Source: Jenkins, 1947.*)

Human factors

Equipment interaction incidents by trained users in World War II

Design equipment to minimize potential for problems

Lab testing and experimental psychology

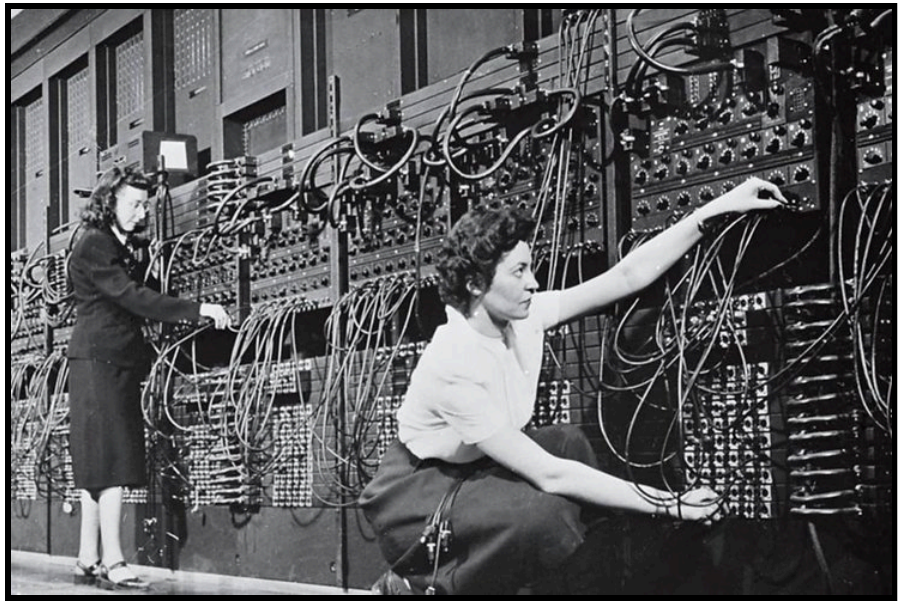


Desktop metaphor

Created in the 1970s at Xerox

Metaphor as a design principle

Move from solving problems to building new interfaces



Human actors

Improve how users work

Study actual users in real workplace environment

Assumptions and methods

Ethnography, prototyping, participatory design

Knowledge has limited generalizability

Research methods

What to study and how

 What is the most effective way of doing X?

 What mistakes programmers make and why?

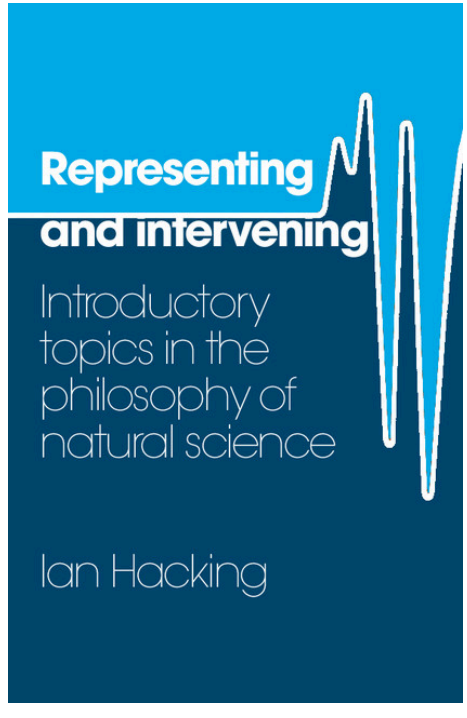
 Can we solve X and Y in a unified way?

 Do systems enable new user experiences?

Methodological bias

Hierarchy in science

- Theoreticians over experimentalists
- Everyone knows Einstein's equation
- Nobody Michelson–Morley experiment

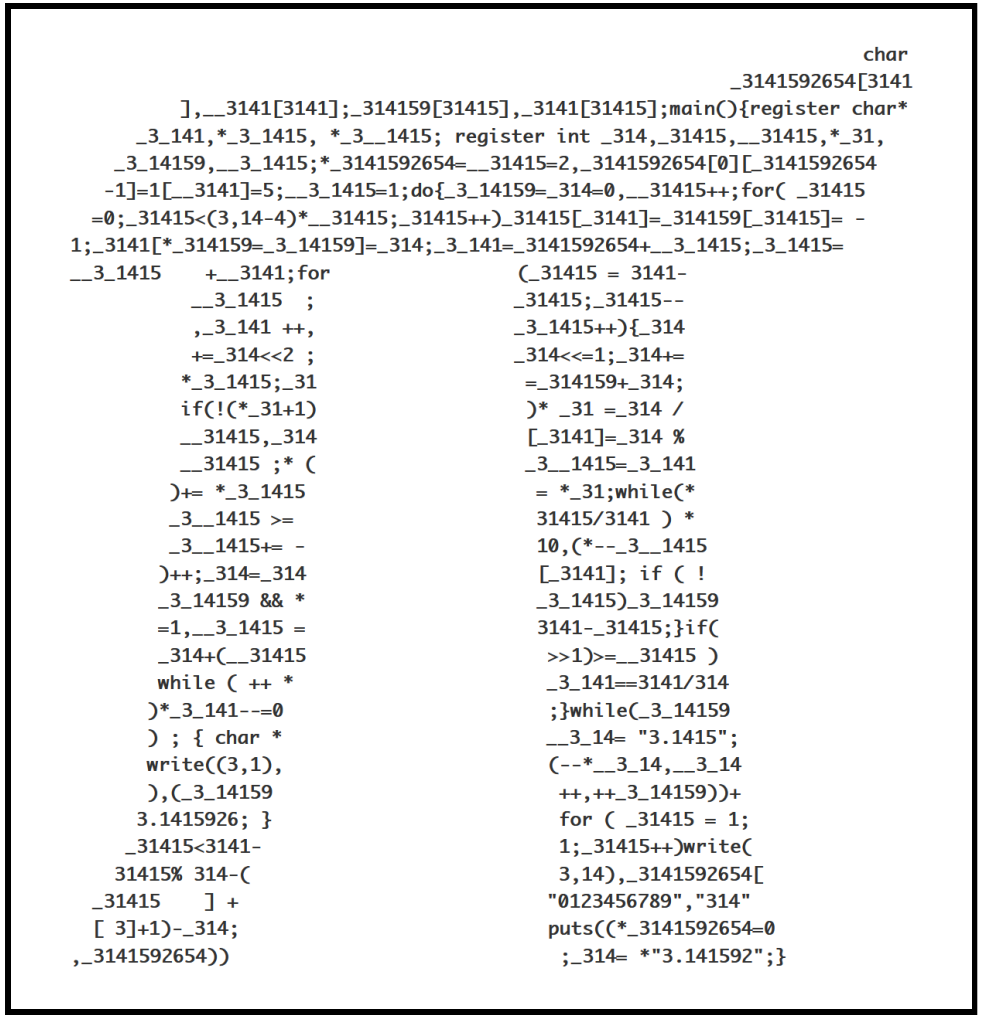


Biases in computing

- Proofs are the most fundamental!
- Can we measure something objective?
- Running a rigorous user experiment?
- All other evaluation is "too soft"!

Controlled experiments

Evidence-based language design



```
                                                                  char
                                                              _3141592654[3141
        ],__3141[3141];_314159[31415],_3141[31415];main(){register char*
      _3_141,*_3_1415, *_3__1415; register int _314,_31415,__31415,*_31,
    _3_14159,__3_1415;*_3141592654=__31415=2,_3141592654[0][_3141592654
   -1]=1[__3141]=5;__3_1415=1;do{_3_14159=_314=0,__31415++;for( _31415
  =0;_31415<(3,14-4)*__31415;_31415++)_31415[_3141]=_314159[_31415]= -
1;_3141[*_314159=_3_14159]=_314;_3_141=_3141592654+__3_1415;_3_1415=
__3_1415    +__3141;for               (_31415 = 3141-
          __3_1415  ;                _31415;_31415--
          ,_3_141 ++,                _3_1415++){_314
          +=_314<<2 ;                _314<<=1;_314+=
         *_3_1415;_31                 =_314159+_314;
         if(!(*_31+1)                 )* _31 =_314 /
         __31415,_314                 [_3141]=_314 %
         __31415 ;* (                 _3__1415=_3_141
        )+= *_3_1415                   = *_31;while(*
        _3__1415 >=                    31415/3141 ) *
        _3__1415+= -                   10,(*--_3__1415
       )++;_314=_314                   [_3141]; if ( !
       _3_14159 && *                   _3_1415)_3_14159
       =1,__3_1415 =                   3141-_31415;}if(
       _314+(__31415                    >>1)>=__31415 )
       while ( ++ *                     _3_141==3141/314
      )*_3_141--=0                      ;}while(_3_14159
      ) ; { char *                      __3_14= "3.1415";
      write((3,1),                      (--*__3_14,__3_14
      ),(_3_14159                        ++,++_3_14159))+
     3.1415926; }                        for ( _31415 = 1;
    _31415<3141-                         1;_31415++)write(
   31415% 314-(                          3,14),_3141592654[
  _31415    ] +                        "0123456789","314"
  [ 3]+1)-_314;                        puts((*_3141592654=0
,_3141592654))                          ;_314= *"3.141592";}
```

Evidence-based language design

For each language feature, determine the best option experimentally

How to make user studies as rigorous as possible?

Randomized controlled trials

Gold standard in medicine

- Compare treatments or with placebo
- Random allocation of participants
- Blinding and study pre-registration

Limitations of RCTs

- Very hard to do properly
- Answers only very limited questions
- Even this may not be rigorous enough!

Case study: Perl vs. Randomo

```
action Main
 number x = z(1, 100, 3)
end

action z(integer a, integer b,
  integer c) returns number
 number d = 0.0
 number e = 0.0
 integer i = a
 repeat b - a times
  if i mod c = 0 then
   d = d + 1
  end
  else then
   e = e + 1
  end
  i = i + 1
 end
 if d > e then
  return d
 end
 else then
  return e
 end
end
```

(a) Quorum

```
$x = &z(1, 100, 3);

sub z{
 $a = $_[0];
 $b = $_[1];
 $c = $_[2];
 $d = 0.0;
 $e = 0.0;
 for ($i = $a; $i <= $b; $i++){
  if ($i % $c == 0) {
   $d = $d + 1;
  }
  else {
   $e = $e + 1;
  }
 }
 if ($d > $e) {
  $d;
 }
 else {
  $e;
 }
}
```

(b) Perl

```
^ Main {
 ~ x \ z(1, 100, 3)
}

^ z(@ a % @ b % @ c) | ~ {
 ~ d \ 0.0
 ~ e \ 0.0
 @ i \ a
 # (b - a) {
  : i ; c ! 0 {
   d \ d + 1
  }
  , {
   e \ e + 1
  }
  i \ i + 1
 }
 : d ` e {
  _ d
 }
 , {
  _ e
 }
}
```

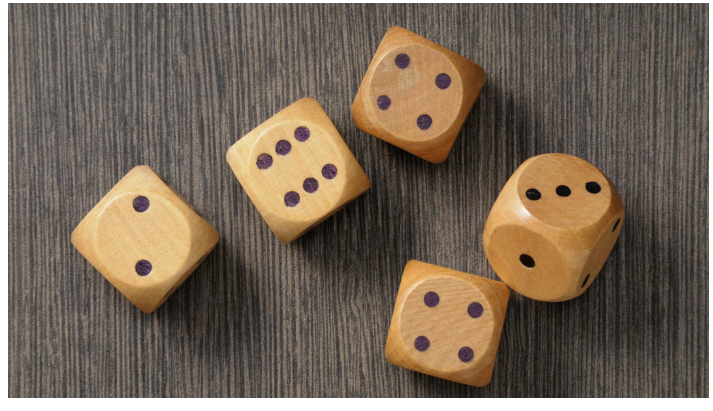
(c) Randomo

An Empirical Investigation into Programming Language Syntax (Steffik, Siebert, 2013)

Getting it right

Study setup

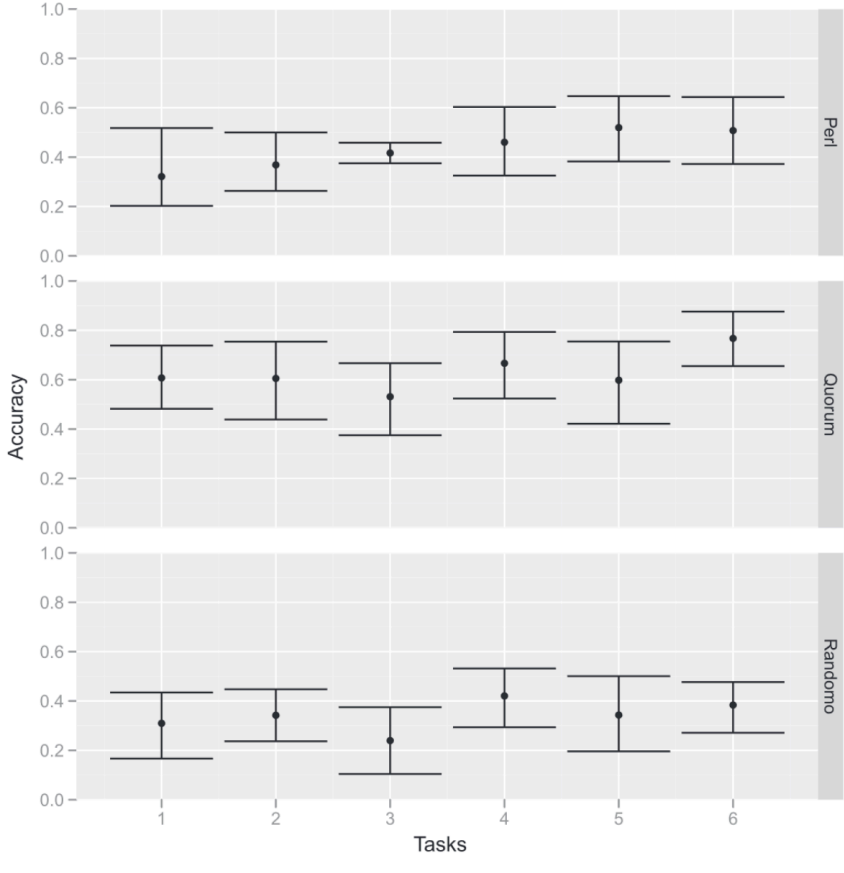
- Copy and modify code sample
- Never programmed before
- Age, gender, language balance



Statistical evaluation

- Verified manual rating of accuracy
- Mauchly's sphericity test
- Repeated-measures ANOVA test

Perl vs. Randomo



While users of Quorum were able to program statistically significantly more accurately than users of Perl ($p = .047$), and users of Randomo ($p = .004$), Perl users were not able to program significantly more accurately than Randomo users ($p = .458$).

Experiments

Studying languages experimentally

- ≠ Typing discipline, syntax, errors, inheritance
- Compare two structurally similar alternatives
- Study participants with similar backgrounds
- Does not help with fundamentally new designs

Empirical studies

Software repository analysis

Software repository analysis

Study existing codebases

- Lots of projects on GitHub
- Commit history, bug reports, etc.

What can we study?

- What leads to fewer bugs?
- How OSS contributors behave
- How code gets duplicated and reused?
- Code quality and code structure

A Large Scale Study of Programming Languages and Code Quality in Github

Baishakhi Ray, Daryl Posnett, Vladimir Filkov, Premkumar Devanbu
{bairay@, dposnett@, filkov@cs., devanbu@cs.}ucdavis.edu
Department of Computer Science, University of California, Davis, CA, 95616, USA

ABSTRACT

What is the effect of programming languages on software quality? This question has been a topic of much debate for a very long time. In this study, we gather a very large data set from GitHub (729 projects, 80 Million SLOC, 29,000 authors, 1.5 million commits, in 17 languages) in an attempt to shed some empirical light on this question. This reasonably large sample size allows us to use a mixed-methods approach, combining multiple regression modeling with visualization and text analytics, to study the effect of language features such as static *v.s.* dynamic typing, strong *v.s.* weak typing on software quality. By triangulating findings from different methods, and controlling for confounding effects such as team size, project size, and project history, we report that language design does have a significant, but modest effect on software quality. Most notably, it does appear that strong typing is modestly better than weak typing, and among functional languages, static typing is also somewhat better than dynamic typing. We also find that functional languages are somewhat better than procedural languages. It is worth noting that these modest effects arising from language design are overwhelmingly dominated by the process factors such as project size, team size, and commit size. However, we hasten to caution the reader that even these modest effects might quite possibly be due to other, intangible process factors, *e.g.*, the preference of certain personality types for functional, static and strongly typed languages.

Categories and Subject Descriptors

D.3.3 [**PROGRAMMING LANGUAGES**]: [Language Constructs and Features]

General Terms

Measurement, Experimentation, Languages

Keywords

programming language, type system, bug fix, code quality, empirical research, regression analysis, software domain

FSE'14 November 16âĂŞ22, 2014, Hong Kong, China

1. INTRODUCTION

A variety of debates ensue during discussions whether a given programming language is "the right tool for the job". While some of these debates may appear to be tinged with an almost religious fervor, most people would agree that a programming language can impact not only the coding process, but also the properties of the resulting artifact.

Advocates of strong static typing argue that type inference will catch software bugs early. Advocates of dynamic typing may argue that rather than spend a lot of time correcting annoying static type errors arising from sound, conservative static type checking algorithms in compilers, it's better to rely on strong dynamic typing to catch errors as and when they arise. These debates, however, have largely been of the armchair variety; usually the evidence offered in support of one position or the other tends to be anecdotal.

Empirical evidence for the existence of associations between code quality programming language choice, language properties, and usage domains, could help developers make more informed choices.

Given the number of other factors that influence software engineering outcomes, obtaining such evidence, however, is a challenging task. Considering software quality, for example, there are a number of well-known influential factors, including source code size [8], the number of developers [29, 3], and age/maturity [13]. These factors are known to have a strong influence on software quality, and indeed, such process factors can effectively predict defect localities [25].

One approach to teasing out just the effect of language properties, even in the face of such daunting confounds, is to do a *controlled experiment*. Some recent works have conducted experiments in controlled settings with tasks of limited scope, with students, using languages with static or dynamic typing (based on experimental treatment setting) [11, ?,15]. While type of controlled study is "*El Camino Real*" to solid empirical evidence,another opportunity has recently arisen, thanks to the large number of open source projects collected in software forges such as GitHub.

GitHub contains many projects in multiple languages. These projects vary a great deal across size, age, and number of developers. Each project repository provides a historical record from which we extract project data including the contribution history, project size, authorship, and defect repair. We use this data to determine the effects of language features on defect occurrence using a variety of tools. Our approach is best described as mixed-methods, or triangulation [7] approach. A quantitative (multiple regression) study is further examined using mixed methods: text analysis, clustering, and visualization. The observations from the mixed methods largely confirm the findings of the quantitative study.

Does strong typing matter?

Large scale corpus study

"[It] appear[s] that "strong typing is modestly better than weak typing, and among functional languages, static typing is also somewhat better than dynamic typing.""

The Register®
Biting the hand that feeds IT

Software

Boffins debunk study claiming certain languages (cough, C, PHP, JS...) lead to more buggy code than others

Hard evidence that some coding lingo encourage flaws remains elusive

By Thomas Claburn in San Francisco 30 Jan 2019 at 21:45 154 SHARE ▼

Tempting through it may be to believe that certain programming languages promote errors, recent research finds little if any evidence of that.

A scholarly paper, "A Large Scale Study of Programming Languages and Code Quality in Github," presented at the 2014 Foundations of Software Engineering (FSE) conference, made that claim that some computer languages show higher levels of buggy code, setting off a firestorm of developer comment.

Does strong typing matter?

Attempt to reproduce the study mostly failed

"I believe [it does] in my heart of hearts, but it's kind of an impossible experiment to run."

Repository analysis

How to and limitations

 Lots of code on GitHub is useless

 Focus on somewhat sensible projects!

 Many hidden factors to account for

 Avoid comparing apples and oranges

 Studying semantics and runtime is hard

Usability evaluation

Considered harmful

Cultural adoption

(Greenberg et al. 2008)

"Usability evaluation is appropriate for settings with well-known tasks and outcomes. They fail to consider how novel systems will evolve and be adopted by a culture over time."

Tricky to evaluate

Early designs

- Purely explorative sketches
- Getting the right design vs. Getting the design right

Cultural adoption

- Hard to imagine future uses
- First radio and automobiles
- Memex, Sketchpad and oNLine System

Evaluating user interface research

(Olsen, 2007)

Lively research field in the 1970s and 1980s

Ubiquitous computing challenges the classic desktop metaphor

Increasing number of non-expert programmers!

User interfaces

New system and languages


Reduce time to create new solutions

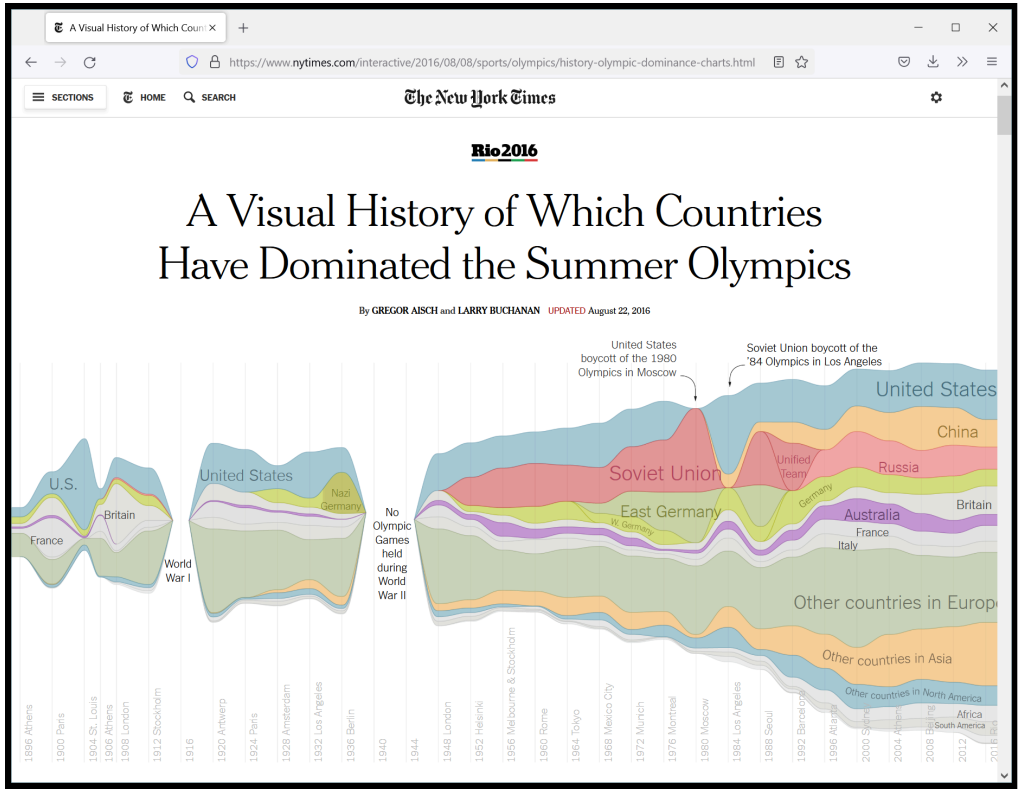

Least resistance to good solutions


Lowering skills barrier of users


Power in common unified infrastructure

Simplifying programming

Data exploration tools



Programming for data journalists

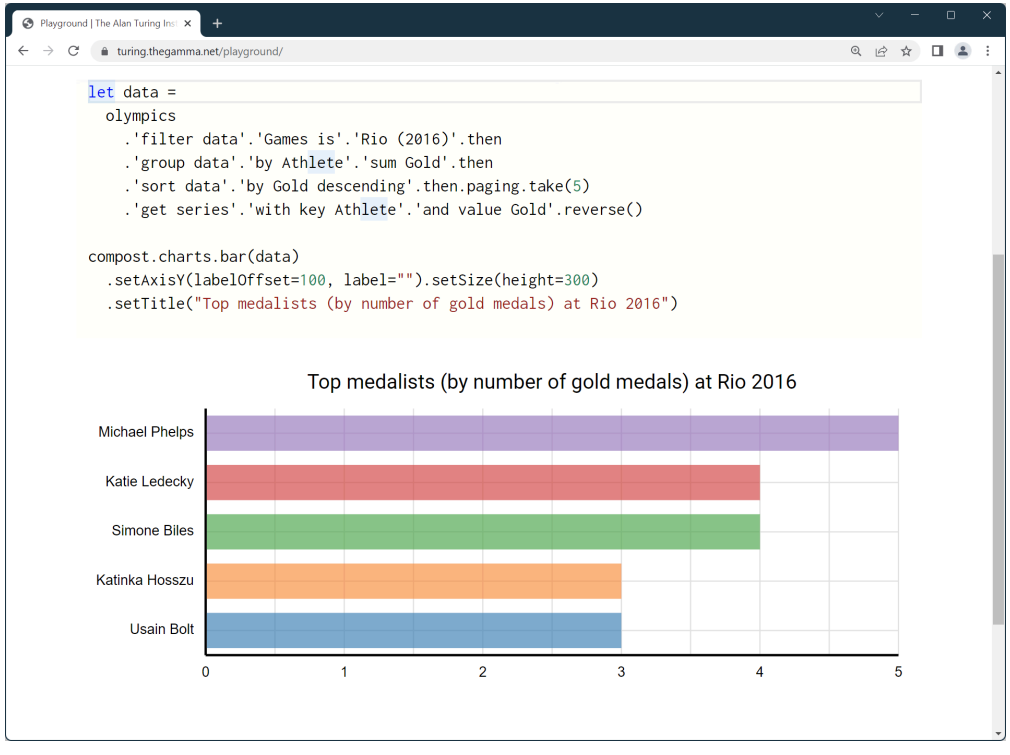
Data transformations using various online data sources

Too hard for Excel, too complex in Python or R

Getting it right is very time-consuming!

Demo

Data exploration in The Gamma



Evaluating The Gamma

Can non-experts actually use it?

Is it better than spreadsheets?

What desirable design characteristics does it have?

Case study: The Gamma

Evaluating programming systems

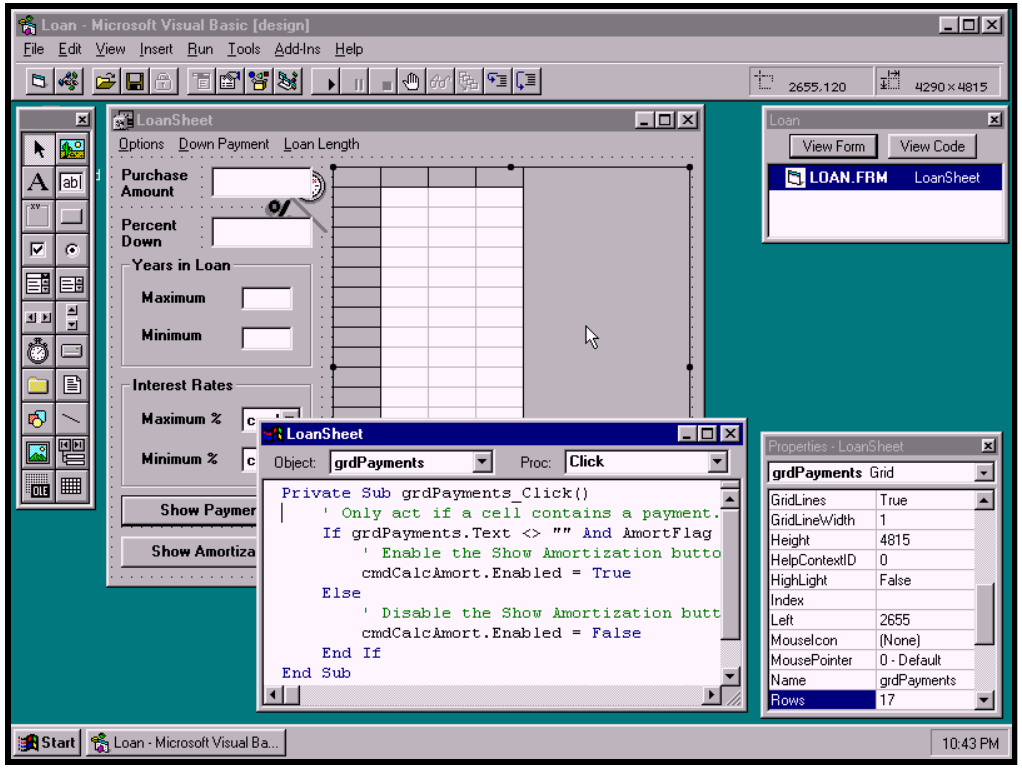
- Programming tool for journalists
- Olsen's framework for UI systems
- tinyurl.com/nprg075-ui

Design questions

- What possible claims can we make?
- What evaluation errors to avoid?

Methods review

Evaluating programming systems



Evaluating HCI toolkits

(Ledo et al., 2018)

Research claims made in publications about UI toolkits, etc.

The same works for languages, libraries, tools, frameworks, ...

Evaluation types

What claims can we make?

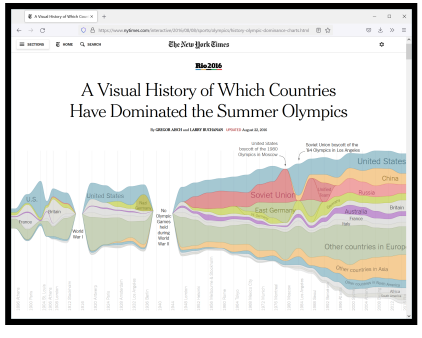
- **Demonstrations** - show what is possible
- **Usage** - study actual system use
- **Performance** - evaluate how well it runs
- **Heuristics** - expert rules of thumb

Demonstrations

Showing a novel example

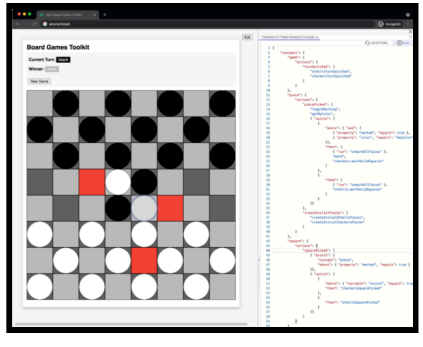
Can do some-thing previously unthinkable

Replicating past examples



System makes previously very hard thing easy

Presenting case studies



Show usability of a system in a range of situations

Demo or Die!

MIT Media Lab paraphrasing of "publish or perish".

Aspen Movie Map
The 1978 precursor of Google Street View

Demo of a radically new technology

Varv: Reprogrammable Interactive Software as a Declarative Data Structure

Marcel Borowski
marcel.borowski@cs.au.dk
Aarhus University
Aarhus, Denmark

Luke Murray
lsmurray@mit.edu
MIT CSAIL
Cambridge, United States

Rolf Bagge
rolf@cavi.au.dk
Aarhus University
Aarhus, Denmark

Janus Bager Kristensen
jbk@cavi.au.dk
Aarhus University
Aarhus, Denmark

Arvind Satyanarayan
arvindsatya@mit.edu
MIT CSAIL
Cambridge, United States

Clemens N. Klokmose
clemens@cs.au.dk
Aarhus University
Aarhus, Denmark

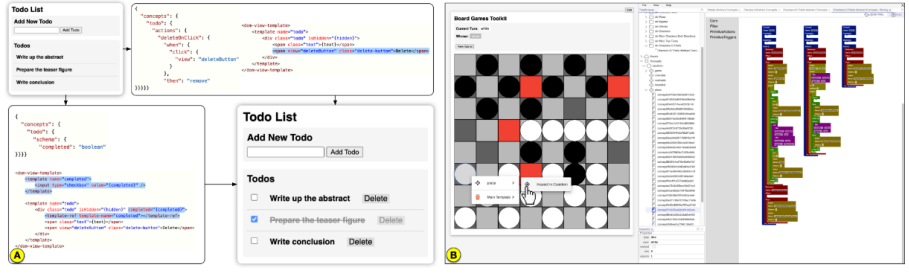


Figure 1: Varv Examples: (a) A todo list web application that is inherently extensible. Here, a basic todo list is extended with the ability to complete and delete todos by adding two new concept definitions and new modified template definitions. (b) A board game toolkit that defines abstractions for board game logic. The games "Checkers" and "Othello" were implemented with the toolkit and then merged into a new "Checkers-O-Thello" game with the addition of a short concept definition. As Varv applications are represented as data structures, higher-level tooling can be developed including a block-based editor (right), an inspector to go from an element in the view to the corresponding template or data (context menu to the left), and a data inspector for live editing application state (middle).

Varv programming system evaluation

(Borowski et al., 2022)

Makes all information visible and modifiable

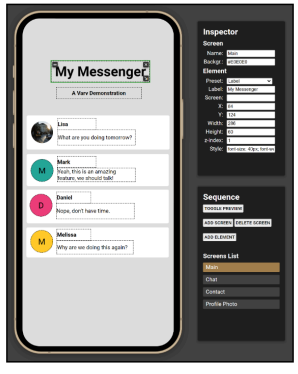
Affects the whole developer workflow

Case studies to illustrate the effects

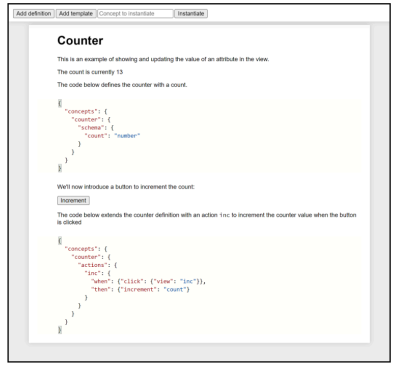
Varv evaluation

Demonstrate workflow

- Two concrete usage scenarios
- Step by step description of work
- Using personas for concreteness



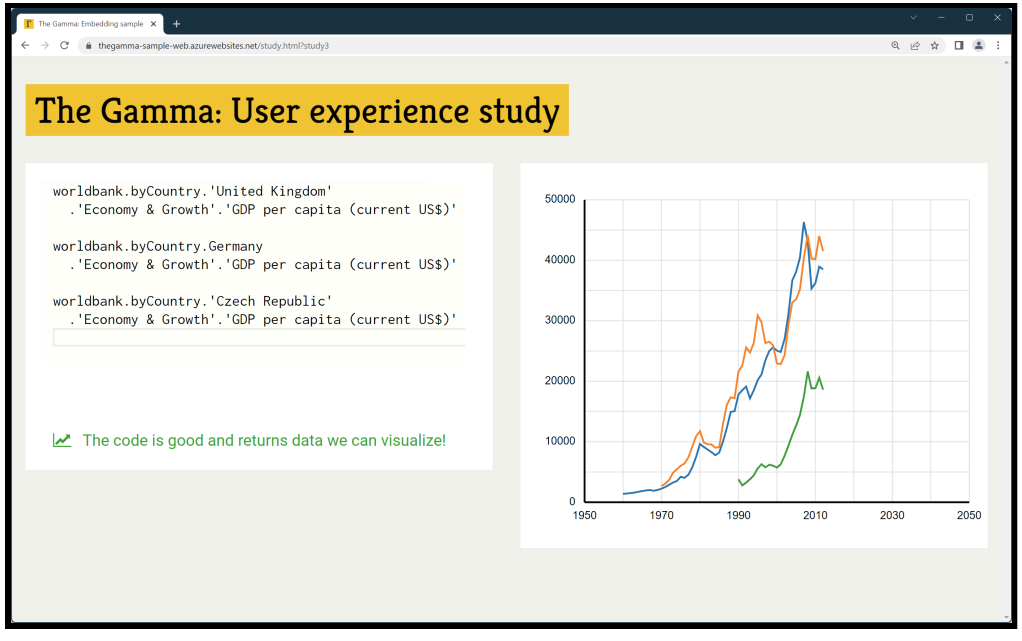
(a) The UI Designer.



(b) The Computational Notebook.

Potential of the system

- Implications of the design
- Debugging, authoring, tools
- Notebooks, blocks, VS Code, etc.



Usage evaluation of The Gamma

(Petricek, 2022)

Can non-programmers really use the system?

Get non-programmers, ask them to try and watch and note!

	Task	*Kind*	*Done*	*Notes*
#1	expenditure	cube	◑	Obtained one of two data series
#2	expenditure	cube	●	Explored furhter data series independently
#3	expenditure	cube	●	Explored further data series independently
#4	expenditure	cube	◕	Completed following a hint to use another member
#5	expenditure	cube	●	Explored further data series independently
#6	worldbank	cube	◕	Completed after a syntax hint about whitespace
#7	worldbank	cube	●	Completed very quickly
#8	worldbank	cube	●	Completed, but needed longer to find correct data
#9	lords	table	◕	Struggled with composition of operations
#10	lords	table	●	Completed very quickly
#11	lords	table	◕	With a hint to avoid operations taking arguments
#12	olympics	table	◕	With a hint to avoid operations taking arguments
#13	olympics	table	◕	With hints about 'then' and operations taking arguments

Table 1. Overview of work completed by individual participants in the study.
The marks denote: ● = completed, ◕ = required some guidance, ◑ = partially completed

The Gamma evaluation

13 participants from business team of a research institute

Asked to complete 1 of 4 different tasks

Evaluated using activity logging, observation and interview

Usage evaluation

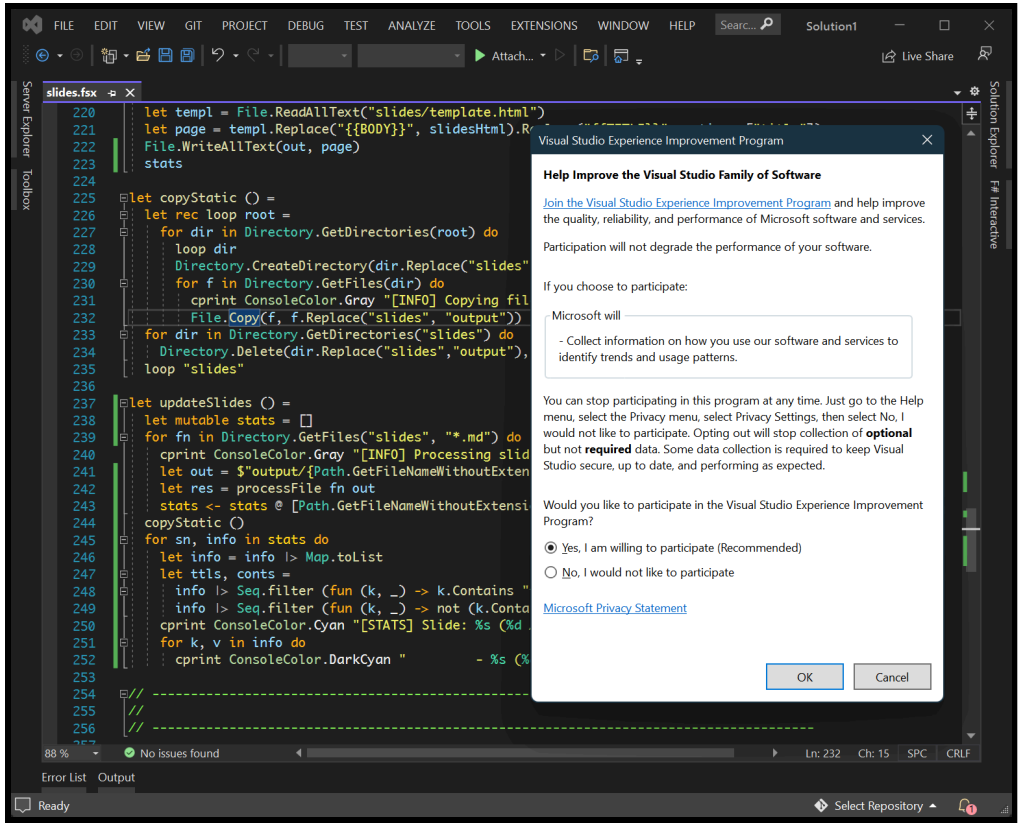
Possible setup

- Complete a given task
- Observe, log & record
- A/B comparison of variants
- In the lab or in the wild

Collecting feedback

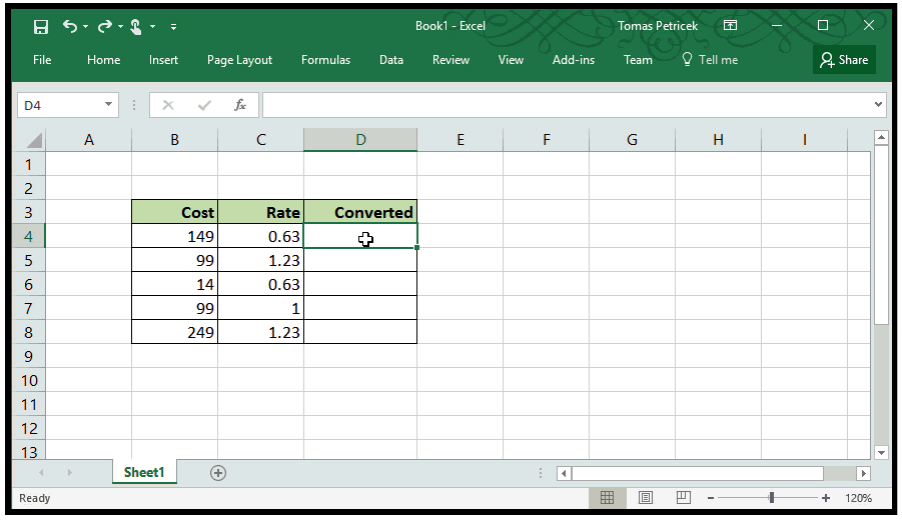
- Complete a questionnaire
- Ask to comment (Think aloud)
- Semi-structured interview afterwards

Collecting usage data in the wild



Widely used to understand use of commercial systems

What language or editor features are used, performance, project profiles

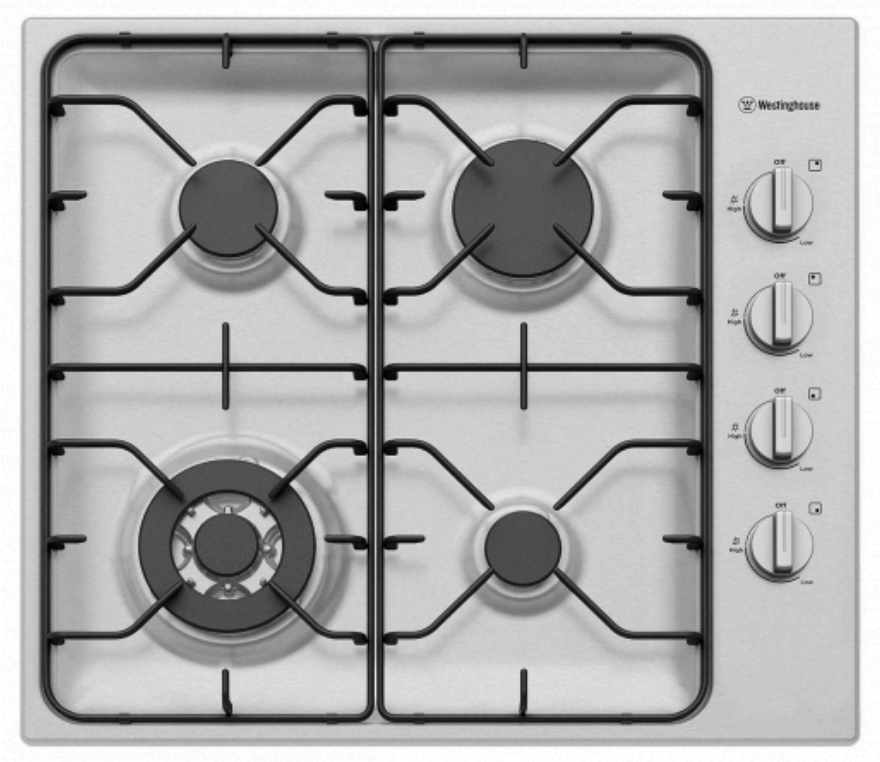


Studying how real users use the system

Interviews and observation studies of real work

Two ways of learning complex formulas

Percolation vs. experts



Heuristics

Rules of thumb for evaluating designs written by experts

Evaluation without direct human involvement!

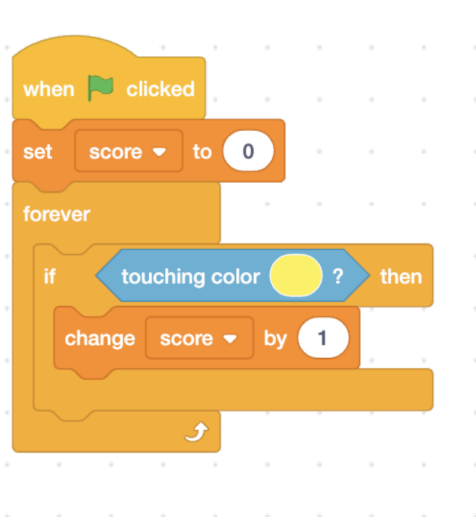
Example: Match between system and the real world

Olsen's criteria for user interface systems

Heuristic evaluation

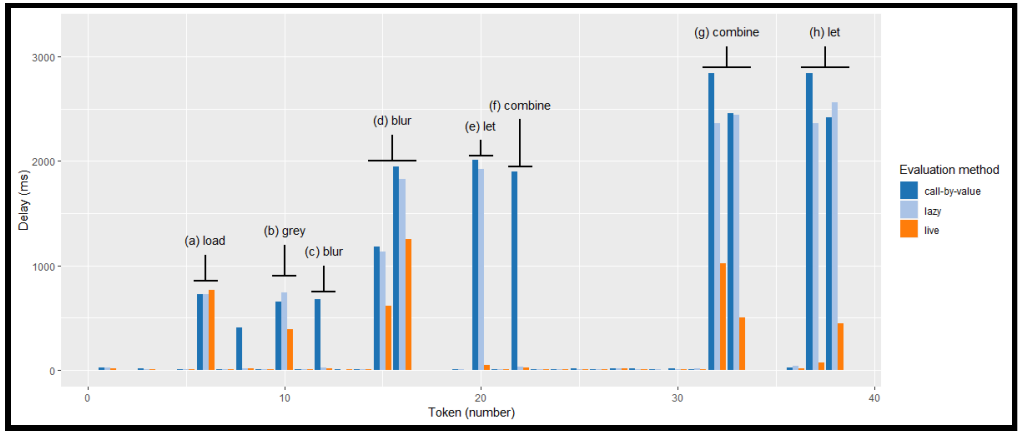
Nielsen's usability heuristics

- Characteristics of a good interface
- General usability guidelines
- Consistency, visibility of state, ...



Cognitive dimensions of notation

- Heuristics for assessing notations
- Broad-brush understandable evaluation
- Viscosity, visibility, abstraction, ...

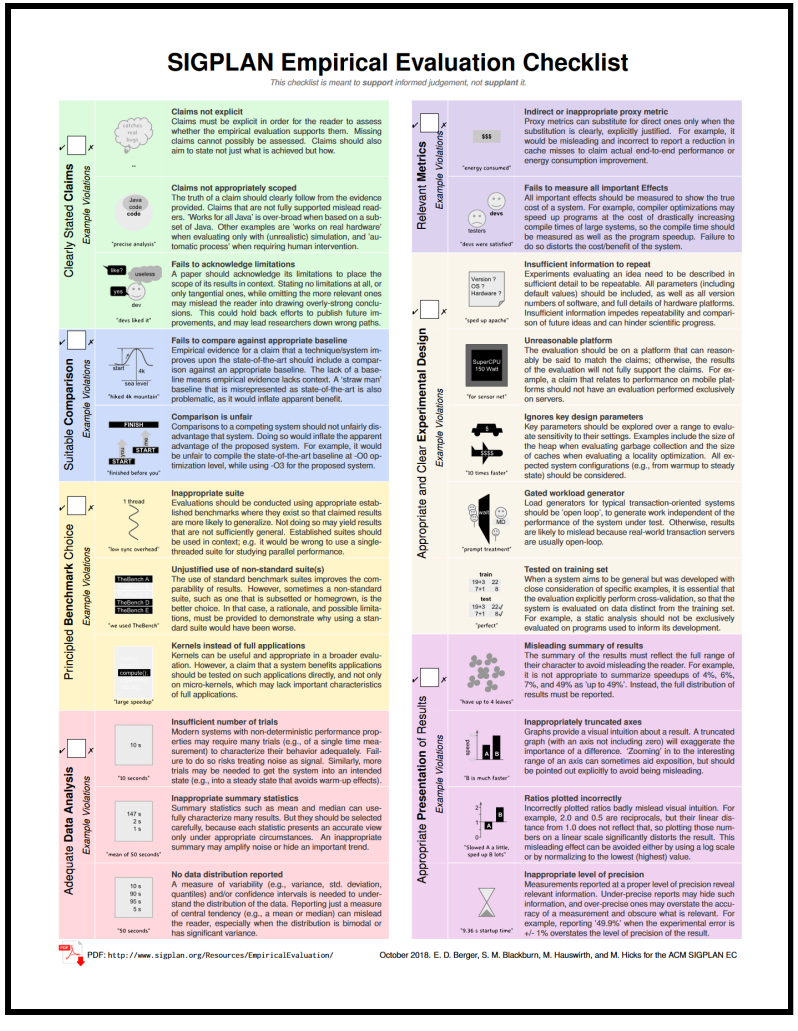


Technical performance

Baseline or improves over state of the art

Efficiency, lines of code

Not about usability, but an easy thing to show



SIGPLAN Empirical Evaluation Checklist

*This checklist is meant to **support** informed judgement, not **supplant** it.*

Clearly Stated Claims — *Example Violations*

Claims not explicit
Claims must be explicit in order for the reader to assess whether the empirical evaluation supports them. Missing claims cannot possibly be assessed. Claims should also aim to state not just what is achieved but how.

Claims not appropriately scoped
The truth of a claim should clearly follow from the evidence provided. Claims that are not fully supported mislead readers. 'Works for all Java' is over-broad when based on a subset of Java. Other examples are 'works on real hardware' when evaluating only with (unrealistic) simulation, and 'automatic process' when requiring human intervention.

Fails to acknowledge limitations
A paper should acknowledge its limitations to place the scope of its results in context. Stating no limitations at all, or only tangential ones, while omitting the more relevant ones may mislead the reader into drawing overly-strong conclusions. This could hold back efforts to publish future improvements, and may lead researchers down wrong paths.

Suitable Comparison — *Example Violations*

Fails to compare against appropriate baseline
Empirical evidence for a claim that a technique/system improves upon the state-of-the-art should include a comparison against an appropriate baseline. The lack of a baseline means empirical evidence lacks context. A 'straw man' baseline that is misrepresented as state-of-the-art is also problematic, as it would inflate apparent benefit.

Comparison is unfair
Comparisons to a competing system should not unfairly disadvantage that system. Doing so would inflate the apparent advantage of the proposed system. For example, it would be unfair to compile the state-of-the-art baseline at -O0 optimization level, while using -O3 for the proposed system.

Principled Benchmark Choice — *Example Violations*

Inappropriate suite
Evaluations should be conducted using appropriate established benchmarks where they exist so that claimed results are more likely to generalize. Not doing so may yield results that are not sufficiently general. Established suites should be used in context; e.g. it would be wrong to use a single-threaded suite for studying parallel performance.

Unjustified use of non-standard suite(s)
The use of standard benchmark suites improves the comparability of results. However, sometimes a non-standard suite, such as one that is subsetted or homegrown, is the better choice. In that case, a rationale, and possible limitations, must be provided to demonstrate why using a standard suite would have been worse.

Kernels instead of full applications
Kernels can be useful and appropriate in a broader evaluation. However, a claim that a system benefits applications should be tested on such applications directly, and not only on micro-kernels, which may lack important characteristics of full applications.

Adequate Data Analysis — *Example Violations*

Insufficient number of trials
Modern systems with non-deterministic performance properties may require many trials (e.g., of a single time measurement) to characterize their behavior adequately. Failure to do so risks treating noise as signal. Similarly, more trials may be needed to get the system into an intended state (e.g., into a steady state that avoids warm-up effects).

Inappropriate summary statistics
Summary statistics such as mean and median can usefully characterize many results. But they should be selected carefully, because each statistic presents an accurate view only under appropriate circumstances. An inappropriate summary may amplify noise or hide an important trend.

No data distribution reported
A measure of variability (e.g., variance, std. deviation, quantiles) and/or confidence intervals is needed to understand the distribution of the data. Reporting just a measure of central tendency (e.g., a mean or median) can mislead the reader, especially when the distribution is bimodal or has significant variance.

Relevant Metrics — *Example Violations*

Indirect or inappropriate proxy metric
Proxy metrics can substitute for direct ones only when the substitution is clearly, explicitly justified. For example, it would be misleading and incorrect to report a reduction in cache misses to claim actual end-to-end performance or energy consumption improvement.

Fails to measure all important Effects
All important effects should be measured to show the true cost of a system. For example, compiler optimizations may speed up programs at the cost of drastically increasing compile times of large systems, so the compile time should be measured as well as the program speedup. Failure to do so distorts the cost/benefit of the system.

Appropriate and Clear Experimental Design — *Example Violations*

Insufficient information to repeat
Experiments evaluating an idea need to be described in sufficient detail to be repeatable. All parameters (including default values) should be included, as well as all version numbers of software, and full details of hardware platforms. Insufficient information impedes repeatability and comparison of future ideas and can hinder scientific progress.

Unreasonable platform
The evaluation should be on a platform that can reasonably be said to match the claims; otherwise, the results of the evaluation will not fully support the claims. For example, a claim that relates to performance on mobile platforms should not have an evaluation performed exclusively on servers.

Ignores key design parameters
Key parameters should be explored over a range to evaluate sensitivity to their settings. Examples include the size of the heap when evaluating garbage collection and the size of caches when evaluating a locality optimization. All expected system configurations (e.g., from warmup to steady state) should be considered.

Gated workload generator
Load generators for typical transaction-oriented systems should be 'open loop', to generate work independent of the performance of the system under test. Otherwise, results are likely to mislead because real-world transaction servers are usually open-loop.

Tested on training set
When a system aims to be general but was developed with close consideration of specific examples, it is essential that the evaluation explicitly perform cross-validation, so that the system is evaluated on data distinct from the training set. For example, a static analysis should not be exclusively evaluated on programs used to inform its development.

Appropriate Presentation of Results — *Example Violations*

Misleading summary of results
The summary of the results must reflect the full range of their character to avoid misleading the reader. For example, it is not appropriate to summarize speedups of 4%, 6%, 7%, and 49% as 'up to 49%'. Instead, the full distribution of results must be reported.

Inappropriately truncated axes
Graphs provide a visual intuition about a result. A truncated graph (with an axis not including zero) will exaggerate the importance of a difference. 'Zooming' in to the interesting range of an axis can sometimes aid exposition, but should be pointed out explicitly to avoid being misleading.

Ratios plotted incorrectly
Incorrectly plotted ratios badly mislead visual intuition. For example, 2.0 and 0.5 are reciprocals, but their linear distance from 1.0 does not reflect that, so plotting those numbers on a linear scale significantly distorts the result. This misleading effect can be avoided either by using a log scale or by normalizing to the lowest (highest) value.

Inappropriate level of precision
Measurements reported at a proper level of precision reveal relevant information. Under-precise reports may hide such information, and over-precise ones may overstate the accuracy of a measurement and obscure what is relevant. For example, reporting '49.9%' when the experimental error is +/- 1% overstates the level of precision of the result.

PDF: http://www.sigplan.org/Resources/EmpiricalEvaluation/

October 2018. E. D. Berger, S. M. Blackburn, M. Hauswirth, and M. Hicks for the ACM SIGPLAN EC

Technical performance

Getting it right

Claims, comparison, benchmarks, metrices, setup, presentation

See SIGPLAN Empirical Evaluation Checklist

Conclusions

Usability and evaluation

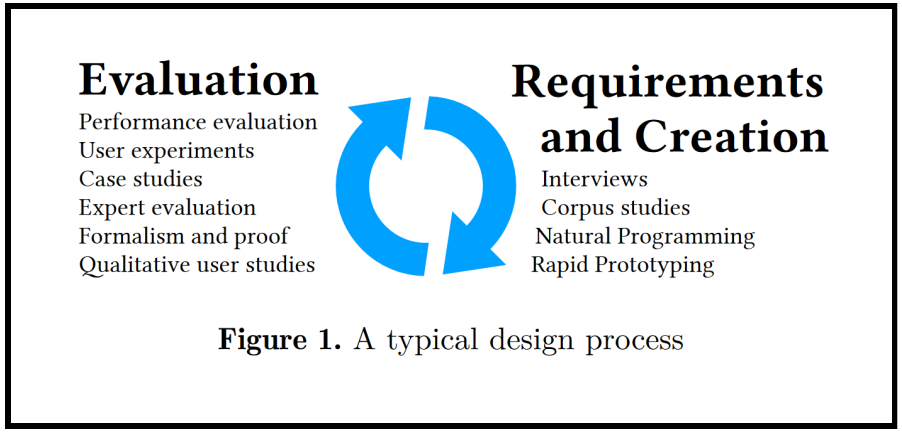


Figure 1. A typical design process

Usability evaluation

Evaluating and comparing with existing systems

Evaluating usability can inspire new designs

The danger is designing with focus just on effective evaluability

Reading

Reactive programming

- Introduction to RxJS concepts
- Available at: https://www.learnrxjs.io/learn-rxjs/concepts/rxjs-primer

Why read this

- Widely used practical library!
- But what exactly is going on?
- Does it always behave "intuitively"?

Conclusions

Human-centric language design

- Evaluation methods from the HCI field
- Controlled experiments, empirical studies
- Demos, usage, heuristics & performance

Tomáš Petříček, 204 (2nd floor)
petricek@d3s.mff.cuni.cz
https://tomasp.net | @tomaspetricek
https://d3s.mff.cuni.cz/teaching/nprg075

References (1/2)

Methodology

- Greenberg, S. and Buxton, B. (2008) Usability Evaluation Considered Harmful (Some of the Time), CHI
- Ledo et al. (2018). Evaluation Strategies for HCI Toolkit Research
- Olsen (2007). Evaluating User Interface Systems Research. UIST
- Arnold, K. (2005). Programmers are People, Too, ACM Queue

Heuristics

- Nielsen, J. (1994). 10 Usability Heuristics for User Interface Design. Norman-Nielsen Group
- Blackwell, A., Green, T. (2002). Notational Systems – the Cognitive Dimensions of Notations framework. (Chapter)
- Berger, E. et al. (2022). SIGPLAN Empirical Evaluation Checklist. ACM SIGPLAN

Examples

- Steffik, A. et al. (2013). An Empirical Investigation into Programming Language Syntax. ACM
- Ray, B. et al. (2014) A Large Scale Study of Programming Languages and Code Quality in Github, FSE
- Berger, E. et al. (2019) On the Impact of Programming Languages on Code Quality, ACM
- Borowski, M. et al. (2022). Varv: Reprogrammable Interactive Sofware as a Declarative Data Structure. CHI
- Petricek, T. (2022). The Gamma: Programmatic Data Exploration for Non-programmers. VL/HCC
- Sarkar, A, et al. (2018). How do people learn to use spreadsheets?

Books

- Norman, D. (1988). The Design of Everyday Things, Basic Books
- Hacking, I. (1983). Representing and Intervening