Evidence Engineering

Audris Mockus
University of Tennessee and Avaya Labs Research
audris@{utk.edu,avaya.com}

[2015-02-20]

How we got here: selected memories

- 70's giant systems
  - Thousands of people, single site, decades
- 90's embracing offshoring and Open Source unproven curiosities
  - Distributed development
- 10's SAS, mobile
  - Small teams, data-driven decisions, incremental releases

Selected Lessons

- Distributed development can be effective
  - Explicit, merit-based, reviews, 20% core, 80% periphery, order of mag. more active users; VCS, ITS, mailing lists
- Offshoring: initially hard
  - Need decades for project experience: productivity + centrality = fluency
  - Difficult with 99% entry-level experience
  - In 2009 predicted a complete transfer of SE expertise from US to India over next decade
- What happened?
  - FLOSS is now a critical public infrastructure
  - Distributed development/small teams are pervasive
  - Much expertise (banking/telecom) has transferred
  - How to foster new forms of development here?

Historic shift in SE focus

- $$$ of computing ↓ 1B times over half century
  - 1950: squeeze max from cycle/bit
    - Domains: Numeric
    - Problems: mathematically defined
    - Success: easy to measure (MFLOPs)
  - 1970: improving developer productivity
    - Domain: Banking, Communications, Operations
    - Problems: well articulated — business automation
    - Success: harder to measure
  - 2000: engineering experiences and behaviors
    - Domain: every part of life
    - Problems: "decision support" — better experience
    - Success: very hard to measure (experience/sales/retention)

Evidence Engineering

Premise

- Current software systems involve
  - Groups of actors (workers/customers/developers)
  - Workflow: relevant data to relevant actors at the right time
  - Operational data: traces of work/play
    - Not a carefully designed measurement system

Definition (Evidence)

Accurate data, properly interpreted

Definition (Evidence Engineering)

Methods to produce accurate and actionable evidence from operational data

Related terms

Definition (Evidence-Based Software Engineering)

Is concerned with determining what works, when and where, in terms of software engineering practice, tools and standards

Definition (Empirical Software Engineering)

Is a field of research that emphasizes the use of empirical studies of all kinds to accumulate knowledge.

Metaphor

Example (Piloting a Drone)

- Feed sensor data to the pilot
- Wrong data/misinterpretation $\rightarrow$ crash
- Accuracy is relatively well defined

In most cases accuracy is not easy to measure or even define

- Domains
  - Search
  - Entertainment
  - Social life
  - Software development
- How to navigate without crashing?

What are main challenges?

- Operational data are treacherous - unlike experimental data
  - Multiple contexts
  - Missing events
  - Incorrect, filtered, or tampered with
- Continuously changing
  - Systems and practices are evolving
- Challenges measuring or defining accuracy
- Potential for misinterpretation

OD: Multi-context, Missing, and Wrong

- Example issues with commits in VCS
  - Context:
    - Why: merge/push/branch, fix/enhance/license
    - What: e.g, code, documentation, build, binaries
    - Practice: e.g., centralized vs distributed
  - Missing: e.g., private VCS, no links to defect
  - Incorrect: tangled, incorrect comments
  - Filtered: small projects, import from CVS
  - Tampered with: `git rebase`

EE for software development tools

- Blurring separation between developer tools and end-user software (DevOps)
  - Collect and integrate developer, integration, and end user activity traces
  - Model development, build, test, deployment, and operations process
  - Produce accurate and actionable evidence for relevant parties on where/what/how to do

Goals and Method of Evidence Engineering

Goals

- Create methods that increase the integrity of evidence

Method

- Discover by studying existing approaches
  - Ratings, workflow systems, crowd-sourcing (e.g., Google maps)
- Suitable techniques from other domains
  - Software engineering, databases, statistics, HCI, ...
- Create novel approaches

Software Engineering

Traditional approaches

- Focus on (potentially) no longer relevant aspects, e.g.,
  - Count bugs
  - Reduce costs
  - Refine process

Evidence Engineering

Potentially relevant practices

- Evaluate outcomes directly:
  - Resilience: e.g., chaos monkey
  - Sales, retention, crashes per visit
- Inform design decisions, shorten cycles
  - DevOps
  - AB testing
- Involve other actors into design, e.g., users
  - Issues and other workflow
  - Content creation
  - Increase data accuracy

Evidence Engineering

Potentially relevant methods

- Sensitivity analysis
- Bayesian models and methods
- Text analysis, e.g., link artifacts

Primary tasks in EE

- Recover context: segment
- Identify if missing
- Obtain missing values
- Determine the quality of observation
- Correct data

Evidence Engineering: novel needs

- Estimate data quality
- Perform data correction
- Perform inverse mapping from traces to reality
  - Theory: data laws
  - Tools: test, debug soundness of evidence production

EE theory: data laws

- Based on the way digital traces are collected
- Based on the physical and economic constraints
- Based on time or location correlations
- Are empirically validated

EE test/debug tools

- Help recover practices
- Develop a map from OD to reality
  - Filter, segment, integrate, and model

Summary

- Quality of data is key determinant of software success

Summary

- Quality of data is key determinant of software success
- The main challenge of EE is OD
  - No two events have the same context
    - Observables represent a mix of platonic concepts
  - Not everything is observed
  - Data are often incorrect

Summary

- Quality of data is key determinant of software success
- The main challenge of EE is OD
  - No two events have the same context
    - Observables represent a mix of platonic concepts
  - Not everything is observed
  - Data are often incorrect
- EE changes the goals of research
  - Model practices of using operational systems
  - Establish data laws
    - Use other sources, experiment, ...
  - Use data laws to
    - Recover the context
    - Correct data
    - Impute missing information
  - Bundle with existing operational support systems
    :noexport:

References

D. Atkins, T. Ball, T. Graves, and A. Mockus.

Using version control data to evaluate the impact of software tools: A case study of the version editor.
IEEE Transactions on Software Engineering, 28(7):625–637, July 2002.

James Herbsleb and Audris Mockus.

Formulation and preliminary test of an empirical theory of coordination in software engineering.
In *2003 International Conference on Foundations of Software Engineering*, Helsinki, Finland, October 2003. ACM Press.

Audris Mockus.

Amassing and indexing a large sample of version control systems: towards the census of public source code history.
In *6th IEEE Working Conference on Mining Software Repositories*, May 16–17 2009.

Audris Mockus.

Succession: Measuring transfer of code and developer productivity.
In *2009 International Conference on Software Engineering*, Vancouver, CA, May 12–22 2009. ACM Press.

Audris Mockus.

Engineering big data solutions.
In *ICSE'14 FOSE*, 2014.

Audris Mockus, Roy T. Fielding, and James Herbsleb.

Two case studies of open source software development: Apache and Mozilla.
ACM Transactions on Software Engineering and Methodology, 11(3):1–38, July 2002.

Audris Mockus, Randy Hackbarth, and John Palframan.

Risky files: An approach to focus quality improvement effort.
In *9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2013.

The costs of computing have decreased a billion times over half a century. The focus of software engineering has consequently transformed from trying to squeeze as much as possible from every compute cycle and of every bit of memory, to improving developer productivity, and, as of late, to engineering user experiences and behaviors.

As computing becomes a commodity, software is omnipresent in all parts of life and, therefore, it either helps end users make decisions or it makes decisions for them. Because most users are not able to understand software systems or articulating their needs, software systems have both to collect massive amounts of operational data related to user activities and to analyze and use that data to provide user experiences that lead to desired outcomes, e.g., increasing sales revenue or the quality of software (if the user happens to be a software developer). It no longer suffices to deliver software that requires, for example, an entry field for a specific piece of data. Instead, the software has to ensure that users can and will enter the relevant data or it has to obtain the data by observing user behavior. Moreover, the software has to ensure that the resulting data reflects the intended quantities, and that the quality of that data is sufficient to make important decisions either automatically of with human support. Such software is engineered to provide accurate and actionable evidence and, therefore, it requires novel approaches to design, implement, test, and operate it. The criteria for success demand much more than performing a well-defined task according to specification. Software has to provide evidence that is both accurate and also leads to the intended user behavior.

In contexts where the desired user behaviors are relatively well defined, some existing software systems achieve these goals through detailed measurement of behavior and massive use of AB testing (in which two samples of users provided slightly different versions of software in order to estimate the effect these differences have on user activity). It is not clear if and how these approaches could generalize to the setting where the desired behaviors are less clearly defined or vary among users.

As operation and measurement are becoming increasingly a part of software development, the separation between the software tools and end-user software are increasingly blurred. Similarly, the measurement associated with testing and use of software is increasingly becoming an integral part of the software delivered to users.

Software engineering needs to catch up with these realities by adjusting the topics of its study. Software construction, development, build, delivery, and operation will become increasingly critical tools and an integral part of the software system. Simply concerning ourselves with architectures and languages to support scalable computation and storage will not be enough. Software systems will have to produce compelling evidence, not simply store or push bits around. Software engineering will, therefore, need to become evidence engineering.