Empirical Methods in Software Research: Which Method Should I Use?

Walter F. Tichy
Why do we need empirical methods in software research?

- There are simply too many tools and methods available for an individual or a software organization to try them all out in order to select the best one(s).

- However, the choice is critical for practitioners. Without data, there is no choice than to fall back on trends, fashion, opinions, personal preferences, prejudice, hearsay, salespersons, consultants, gurus.

- **Empirical studies** investigate, whether differences in software technologies actually exist, with respect to cost, reliability, maintainability, usability, ease of learning, etc.
Empirical studies have become an active area in software research.
The discussion at recent REFSQs have confirmed the strong need for empirical validation of the effectiveness for our RE methods by case studies and experiments, but the literature to date, including that of the REFSQ series, could show more of this validation. This lack is assumed to be at least partly due to the difficulties of

- bringing academics and practitioners together to pursue empirical studies and
- finding and persuading the participation of a sufficient number of suitable subjects for experiments.

Therefore, REFSQ 2012 will offer two events in its empirical track:

1. **Empirical Fair (EF):** Practitioners can propose studies that their organizations would like to have conducted, and researchers can propose studies that they would like to conduct in industry. The EF is a meeting point to match the demand and supply of empirical studies among researchers and practitioners.

2. **Empirical Studies at REFSQ (ESR):** Practitioners and academics will be given the opportunity to conduct a small number of empirical studies during REFSQ 2012 itself. The goals of this opportunity, besides that of permitting the conduct of some studies, are to raise awareness for the necessity and benefits of empirical studies and to show that participating in them is not dangerous to one’s health.
Invariant questions:

1. How to produce **software better** (faster, cheaper)?

2. How to produce **better software** (more reliable, more usable, more maintainable, etc.)?

3. How to show that 1. or 2. have been achieved?
The controlled, randomized experiment

- **Vary Independent variables**
- **Control confounding variables**
- **Observe dependent variables**
Example: Experiment about Pair Programming

- 295 professional consultants (!)
- split into 99 single programmers and 98 pairs
- coming from 29 consultant companies in Norway, Sweden and GB
  - Accenture
  - Cap Gemini
  - Oracle
  - and others
- Participants were compensated for 5 hours work time.
- Cost for that alone: € 250.000

Results for Pair Programming

- Duration: -8%
- Cost: 84%
- Correctness: 7%

Significant difference: $p \leq 0.0001$
Difference according to programmer competence

- Duration:
  - Beginner: 5%
  - Intermediate: -9%
  - Expert: -28%

- Cost:
  - Beginner: 111%
  - Intermediate: 43%
  - Expert: 83%

- Correctness:
  - Beginner: 73%
  - Intermediate: 4%
  - Expert: -8%
Results

- Large study, with almost 300 professional subjects
  - Generalizability is excellent.
- Distinguishes competence and sw complexity
  - PP is effective for beginners, especially when the sw is complex.
  - PP is ineffective for experts (without PP experience).
  - Recommendation: use pair programming for beginners

- Many studies use students as subjects. Have results with student subjects any relevance for professionals?
Some results from Experiments

- Inspections help find software defects early.
- Design patterns work as advertised.
- Inheritance depth is a poor predictor for maintenance effort.
- Pair programming only works for beginners.
- Pair programming can be replaced with single programmers and inspections (for beginners).
- Test-first is not better than test-last.
- UML does not help in maintenance tasks.

Note: these are all experiments about software processes, not about tools (other than the last).
Pros and Cons of Experiments?

**Advantages:**
- Establishes cause-effect relationship
- Experimental method is well developed (methods, statistics)

**Disadvantages:**
- Expensive
- Professional participants are hard to get, even if you pay for their time
- Experiments take time (about 1 experiment per year per PhD student)
- Negative results are the rule
- Only feasible, if tools/methods are easy to learn

Suppose you are developing a new sw technique. You are busy improving it. Experiment is much too expensive and time-consuming for each improvement step. How can we make progress more quickly?
Alternative: Ex post facto Studies: Analyse Software Repositories

- Look for correlations in software repositories including bug histories
- Example: Can software metrics predict fault-prone components?

Nagappan, Ball, Zeller: Mining Metrics to Predict Component Failures, ICSE 2006

High level description

1. Collect input data

![Diagram showing Bug Database, Version Database, and Code]

2. Map post-release failures to defects in entities

![Diagram showing entities with temperature indicators]

3. Predict failure probability for new entities

![Diagram showing entity, predictor, failure probability]

Source: Nagappan
Projects researched

- Internet Explorer 6
- IIS Server
- Windows Process Messaging
- DirectX
- NetMeeting

>1,000,000 Lines of Code

Quelle: Nagappan
<table>
<thead>
<tr>
<th>Metric</th>
<th>Correlation with f0's metric across all functions in module M</th>
<th>Max</th>
<th>0.514</th>
<th>0.585</th>
<th>0.496</th>
<th>0.509</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td># executable lines in f0</td>
<td>Total</td>
<td>0.131</td>
<td>0.709</td>
<td>0.797</td>
<td>0.187</td>
</tr>
<tr>
<td>Parameters</td>
<td># parameters in f0</td>
<td>Max</td>
<td>-0.344</td>
<td>0.372</td>
<td>0.547</td>
<td>0.015</td>
</tr>
<tr>
<td>Arcs</td>
<td># arcs in f0's control flow graph</td>
<td>Total</td>
<td>0.116</td>
<td>0.689</td>
<td>0.790</td>
<td>0.152</td>
</tr>
<tr>
<td>Blocks</td>
<td># basic blocks in f0's control flow graph</td>
<td>Max</td>
<td>-0.209</td>
<td>0.376</td>
<td>0.587</td>
<td>0.527</td>
</tr>
<tr>
<td>ReadCoupling</td>
<td># global variables read in f0</td>
<td>Total</td>
<td>0.127</td>
<td>0.679</td>
<td>0.803</td>
<td>0.158</td>
</tr>
<tr>
<td>WriteCoupling</td>
<td># global variables written in f0</td>
<td>Max</td>
<td>-0.172</td>
<td>0.676</td>
<td>0.756</td>
<td>0.277</td>
</tr>
<tr>
<td>AddrTakenCoupling</td>
<td># global variables whose address is taken in f0</td>
<td>Total</td>
<td>0.128</td>
<td>0.629</td>
<td>0.629</td>
<td>0.230</td>
</tr>
<tr>
<td>ProcCoupling</td>
<td># functions that access a global variable written in f0</td>
<td>Max</td>
<td>-0.063</td>
<td>0.614</td>
<td>0.496</td>
<td>0.024</td>
</tr>
<tr>
<td>Total</td>
<td>0.043</td>
<td>0.562</td>
<td>0.579</td>
<td>0.000</td>
<td>0.443</td>
<td></td>
</tr>
<tr>
<td>FanIn</td>
<td># functions calling f0</td>
<td>Max</td>
<td>0.043</td>
<td>0.562</td>
<td>0.579</td>
<td>0.000</td>
</tr>
<tr>
<td>Total</td>
<td>0.043</td>
<td>0.562</td>
<td>0.579</td>
<td>0.000</td>
<td>0.443</td>
<td></td>
</tr>
<tr>
<td>FanOut</td>
<td># functions called by f0</td>
<td>Max</td>
<td>-0.197</td>
<td>0.360</td>
<td>0.613</td>
<td>0.345</td>
</tr>
<tr>
<td>Total</td>
<td>0.056</td>
<td>0.651</td>
<td>0.776</td>
<td>0.046</td>
<td>0.506</td>
<td></td>
</tr>
<tr>
<td>Complexity</td>
<td>McCabe's cyclomatic complexity of f0</td>
<td>Max</td>
<td>-0.200</td>
<td>0.363</td>
<td>0.594</td>
<td>0.451</td>
</tr>
<tr>
<td>Total</td>
<td>0.112</td>
<td>0.680</td>
<td>0.801</td>
<td>0.165</td>
<td>0.529</td>
<td></td>
</tr>
</tbody>
</table>
Metrics and their Correlation with Post-Release Defects

<table>
<thead>
<tr>
<th>Per-class metrics — correlation with maximum and sum of metric across all classes $C$ in a module $M$</th>
<th>Max</th>
<th>0.244</th>
<th>0.589</th>
<th>0.534</th>
<th>0.100</th>
<th>0.283</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassMethods # methods in $C$ (private / public / protected)</td>
<td>Max</td>
<td>0.244</td>
<td>0.589</td>
<td>0.534</td>
<td>0.100</td>
<td>0.283</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>0.520</td>
<td>0.630</td>
<td>0.581</td>
<td>0.094</td>
<td>0.469</td>
</tr>
<tr>
<td>InheritanceDepth # of superclasses of $C$</td>
<td>Max</td>
<td>0.428</td>
<td>0.546</td>
<td>0.303</td>
<td>0.131</td>
<td>0.323</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>0.432</td>
<td>0.606</td>
<td>0.496</td>
<td>0.111</td>
<td>0.425</td>
</tr>
<tr>
<td>ClassCoupling # of classes coupled with $C$ (e.g. as attribute / parameter / return types)</td>
<td>Max</td>
<td>0.501</td>
<td>0.634</td>
<td>0.466</td>
<td>-0.303</td>
<td>0.264</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>0.547</td>
<td>0.598</td>
<td>0.592</td>
<td>-0.158</td>
<td>0.383</td>
</tr>
<tr>
<td>SubClasses # of direct subclasses of $C$</td>
<td>Max</td>
<td>0.196</td>
<td>0.502</td>
<td>0.582</td>
<td>-0.207</td>
<td>0.387</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>0.265</td>
<td>0.560</td>
<td>0.566</td>
<td>-0.170</td>
<td>0.387</td>
</tr>
</tbody>
</table>

Quelle: Nagappan
Do metrics correlate with failures?

<table>
<thead>
<tr>
<th>Project</th>
<th>Metrics correlated w/ failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>#Classes and 5 derived</td>
</tr>
<tr>
<td>B</td>
<td>almost all</td>
</tr>
<tr>
<td>C</td>
<td>all except MaxInheritanceDepth</td>
</tr>
<tr>
<td>D</td>
<td>only #Lines (software was refactored if metrics indicated a problem)</td>
</tr>
<tr>
<td>E</td>
<td>#Functions, #Arcs, Complexity</td>
</tr>
</tbody>
</table>
Do metrics correlate with failures?

<table>
<thead>
<tr>
<th>Project</th>
<th>Metrics correlated w/ failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>#Classes and 5 derived</td>
</tr>
<tr>
<td>B</td>
<td>almost all</td>
</tr>
<tr>
<td>C</td>
<td>all except MaxInheritanceDepth</td>
</tr>
<tr>
<td>D</td>
<td>only #Lines</td>
</tr>
<tr>
<td>E</td>
<td>#Functions, #Arcs, Complexity</td>
</tr>
</tbody>
</table>

**YES**

Given enough data for a project, a predictor for this project can be built.

*Quelle: Nagappan*
Is there a set of metrics that fits all projects?

<table>
<thead>
<tr>
<th>Project</th>
<th>Metrics correlated w/ failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>#Classes and 5 derived</td>
</tr>
<tr>
<td>B</td>
<td>almost all</td>
</tr>
<tr>
<td>C</td>
<td>all except <em>MaxInheritanceDepth</em></td>
</tr>
<tr>
<td>D</td>
<td>only #Lines</td>
</tr>
<tr>
<td>E</td>
<td>#Functions, #Arcs, Complexity</td>
</tr>
</tbody>
</table>
Is there a set of metrics that fits all projects?

<table>
<thead>
<tr>
<th>Project</th>
<th>Metrics correlated w/ failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>#Classes and 5 derived</td>
</tr>
<tr>
<td>B</td>
<td>Almost all</td>
</tr>
<tr>
<td>C</td>
<td>Except MaxInheritanceDepth</td>
</tr>
<tr>
<td>D</td>
<td>Only #Lines</td>
</tr>
<tr>
<td>E</td>
<td>#Functions, #Arcs, Complexity</td>
</tr>
</tbody>
</table>

Quelle: Nagappan
Pros and Cons of SW Repositories

Advantages
- Large data sets available, even open source
- Automate analysis
- Quantitative results
- Don’t need to deal with, or search for, human subjects.

Disadvantages
- You only get correlations, no cause-effect relationship
- Can only analyze what is there. If a new technique has not been used, then there is no data to analyze.
- So it is useless for untried tools and methods
Analysis of software repositories

Vary independent variables

Control confounding variables

Observe dependent variables
What to Do?

- How can the empirical community contribute useful insights that demonstrably improve software engineering?
- And do so faster than it has in the past?
- Note: “More money” is the wrong answer.
Recommendation: Use Benchmarks!

- Benchmarks are sets of problems with a quality metric for solutions (or gold standard solutions)
  - Independent teams apply their automated “solvers” to the problem and the quality of the solutions can be compared.
  - Benchmarks have a tremendous advantage over experiments with human subjects: they can be repeated as often as necessary, usually at moderate cost.
  - Setting up a benchmark is usually not for free: data has to be collected, benchmark programs have to be prepared.
  - However, this cost can be amortized over many trials and provides a basis for comparison.
  - Over time, the benchmark must evolve (become harder, more general, avoid overfitting.)
Benchmarks have been extremely successful in driving research

- **Computer architecture**: Various benchmarks have been used for decades in order to compare processor performance.
  - The Standard Performance Evaluation Corporation (SPEC) publishes benchmarks to evaluate a range of performance criteria (CPU, Web server, Mail Server, AppServer, power consumption, etc.)
  - Benchmarks combined with simulation have made computer architecture research quantitative.
  - Every performance feature must be substantiated on relevant benchmarks.
Autonomic vehicles: DARPA Grand Challenge

Google’s autonomic vehicle

2007 DARPA Urban Challenge

2004, 2005 DARPA Grand Challenge
Where Benchmarks Rule:

- **Databases:** Transaction Processing Performance Council (TPC)
- **Speech recognition:** large databases of speech samples are used in competitions to determine the best speech recognizer
  - Here, the issue is not speed, but error rate.
- **Speech translation:** same idea.

In all of these cases, benchmarks resulted in swift and substantial progress. The winning techniques were quickly adopted by other teams and improved upon. How could we achieve comparable progress in software research?
Software research could use more benchmarks

- Benchmarks apply to any tool that automates an aspect of software engineering.
- Share the work on developing a wider range of meaningful and challenging benchmarks, so
  - The work is spread over several teams
  - better tools can be built,
  - we know which techniques work best,
  - progress accelerates.

- Some examples of SE benchmarks follow.
Example 1: Data Race Detection

- Data races (unsynchronized accesses to shared variables) are a common defect in parallel programs.
- They are difficult to find.
- Current race detectors are impractical
  - They produce thousands to millions of false alarms.
  - Programmers are overwhelmed.

Why false positives?
- Ad-hoc, programmer-defined synchronizations
- Unknown synchronization libraries
- Detectors cannot reason about these, causing many false positives

Contribution: how to handle user-defined synchronization and unknown synchronization libraries, reducing false positives.
What is a Data Race?

- Two or more concurrent accesses to a shared location, at least one of them a write.

Thread 1

X = 0
X++

Thread 2

T = X

T=0 or T=1?
Ad-hoc (User-defined) Synchronization

- Synchronization constructs implemented for performance reasons

- Ad-hoc synchronizations are widely used
  - 12 - 31 in SPLASH-2 and 32 - 329 in PARSEC 2.0

/*Initially \texttt{FLAG} is zero */

\begin{minipage}{0.4\textwidth}
Thread 1
\begin{itemize}
\item DATA++
\item \texttt{FLAG} = 1
\item ...
\end{itemize}
\end{minipage}
\begin{minipage}{0.6\textwidth}
Thread 2
\begin{itemize}
\item ...
\item while(\texttt{FLAG} == 0)
\item --do nothing
\item DATA--
\end{itemize}
\end{minipage}
Test Suite – data-race-test

- 120 different test cases (2-16 Threads)
  - Test cases are racy or race-free programs (using Pthread)
    - Includes difficult cases
  - Spinning read loop detection of up to 7 basic blocks
    - 24 false positives and one false negative are removed
  - Removing information about Pthread library (unknown library)
    - Only one false positive more

<table>
<thead>
<tr>
<th>Tools</th>
<th>False alarms</th>
<th>Missed races</th>
<th>Failed cases</th>
<th>Correctly analyzed cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helgrind+ lib</td>
<td>32</td>
<td>8</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>Helgrind+ lib+spin(7)</td>
<td>8</td>
<td>7</td>
<td>15</td>
<td>105</td>
</tr>
<tr>
<td>Helgrind+ nolib+spin(7)</td>
<td>9</td>
<td>7</td>
<td>16</td>
<td>104</td>
</tr>
<tr>
<td>DRD</td>
<td>13</td>
<td>20</td>
<td>33</td>
<td>87</td>
</tr>
</tbody>
</table>
Exmple 2: Auto-Parallelization Benchmark

- To test automatic parallelizers, we construct a benchmark
  - sequential implementations
  - hand-parallelized implementation

- We test auto-future detection, pipelines, master/worker and other patterns
  - Is all parallelization potential found?
  - Were correct transformations steps performed?
  - Were concurrency bugs introduced?
  - What speed-up was achieved?
Example 3: NLRP-Bench
A Benchmark for Requirements Processing
Online at http://nlrp.ipd.kit.edu

- Sample Requirements Specs:
  - ITrust Medical Care
  - Pacemaker
  - Elevator
  - Steam Boiler
  - Ambulance Dispatching System
  - Movie Theatre
  - Kuchenrezept
  - Ludo
  - Problemmelder
  - Pflichtenheft Handyverträge
- RE UTS Coincidence Matrix in the ATLAS Muon Spectrometer
- Quasar Fraunhofer Türsteuergerät
- German Health Professional Card and Security Module Card
- ERS ACME - University Library Information System
- Racing
- Timbered House
- Whois Protocol
- Display Management System
- Cable TV Package Purchase
- DaimlerChrysler Demonstrator: Instrument Cluster
A Grand Challenge:
Programming in ordinary language

Benchmarks would be good for evaluation.
But where to get them?
Answer: Create Animations and let subjects describe them in their own words. Then use the stories as input to the generator.
Subjects are shown the video and tell the story

- 10 different animations so far,
- 90 stories, which are the benchmark for AliceNLP.

The astronaut says, "That's one small step for a man...". As he says this, the alien is moving on his wheels toward him. The astronaut continues, "...one...giant leap for...". He stops as he sees the alien moving towards [...]
Conclusions

- I think the use of benchmarks in software research is not as high as it could be.
- All areas of SE could benefit: requirements, design, implementation, testing, maintenance.
- With realistic benchmarks, one gets reliable and testable results.
- Benchmarks accelerate progress: they eliminate inferior choices quickly, help concentrate on the challenges.
- Share the work of preparing benchmarks.
- With a concentrated effort in benchmarking, we might speed up tool research dramatically.
- When tool progress has been made, check usability with human subjects (the expensive experiment).
“If you are not keeping score, you’re just practicing.”

Vince Lombardi
Berühmter US Football Trainer
Barcelona gegen Manchester United: Wer spielt besser?