Reflections on ICSE 2020

42nd International Conference on Software Engineering
(July 7th-11th, 2020)

Seminar: Beiträge zum Software Engineering

Franz Zieris
zieris@inf.fu-berlin.de
Three Themes

1. **IDTIMWYTIM**
   (I don't think it means what you think it means)

   Researchers blur terminology
   - "Root Cause"
   - "Case Study"
   - "Methodological Triangulation"

2. **Popular Subject – Standard Tool – False Assumption**

   Empirical invalidation of (tacit) assumptions which (a lot) of research built on

3. **Big Data in the Way of Qualitative Analysis**

   How come that hand-crafted qualitative codes can miraculously applied to thousands of source code repositories?
Theme 1: IDTIMWYTIM

Case 1.1 "Root Cause"
What is a "Root Cause"?
What is "Root Cause Analysis"?
Case 1.1 "Root Cause"

• Example 1.1a: "Causal Testing: Understanding Defects’ Root Causes" (by Johnson et al.)

[Distinguished Artifacts Award]
– For failing test case: Tool generates a pair of minimally different, failing/passing test cases
– Motivation: "finding the relevant line is often not enough to help fix the bug [...] developers need help identifying and understanding the root cause of buggy behavior." (60x "root cause")
Case 1.1 "Root Cause"

- **Example 1.1b:** "Towards Understanding and Fixing Upstream Merge Induced Conflicts in Divergent Forks: An industrial Case Study" (by Sung et al.)
  - For failing merge commit: Tool searches for upstream renames
  - Motivation: "Root-causing the upstream commit responsible for merge conflict [...] is non-trivial when the commit history of the upstream consists of several thousand commits." (9x root cause)
Case 1.2 "Case Study"
What is a "Case Study"?
Case 1.2 "Case Study"

• Example 1.2a: "Why reinventing the wheels? An empirical study on library reuse and re-implementation" (by Xu et al.) [J-1st]

+ Why-Question: "goal is to understand why developers switch from their self-implemented code to an external library with the same functionality and the other way around."

+ Section 2 "Case Study Design", Section 3 "Case Study Results", Section 4.2 "Threats to Validity" citing Yin (2002)

— Data collection: surveys to three populations ("Android", "Python", "Open"), i.e., no cases
Case 1.2 "Case Study"

• The classic "case study" ambiguity, Examples 1.2b to 1.2q:
  – 17 papers from ICSE main conference speak of "a case study" they conducted and mean "demonstration of our tool"
    • (excluding Journal-First presentations, not part of proceedings)

• Only one was a hybrid:
  – "Improving the Effectiveness of Traceability Link Recovery using Hierarchical Bayesian Networks" (Moran et al.)
  – Contains an "industrial case study" w/ online-survey and semi-structured interviews
Theme 1: IDTIMWYTIM

Case 1.3 "Methodological Triangulation"

What is a "Triangulation"?

What types are there?
Case 1.3 "Methodological Triangulation"

- Example 1.3a: "One Size Does Not Fit All: A Grounded Theory and Online Survey Study of Developer Preferences for Security Warning Types" (Danilova et al.):
  
  "In addition to the [26 semi-structured face-to-face] interviews [with students and professional developers], we conducted one focus group with seven academic researchers using the same guideline. This was done in order to ensure data reliability and validity by using multiple data collection methods (methodological triangulation)."
Theme 2: Subject-Tool-Assumption
Theme 2: Subject-Tool-Assumption

• General Structure:
  – Research of S routinely uses T which (possibly tacitly) assumes that A is true or useful.
  – Empirical evidence shows that A is not true.

• This is how research communities are supposed to work.
  – But it's nice to see the effects in action.
  – Are you ready?
Case 2.1 "git diff"

- Paper: "How different are different diff algorithms in Git?" (Nugroho et al.) [J-1st]

- The pattern:
  - Research in MSR routinely uses the SZZ algorithm to identify defect-relevant commits (which practically involves standard "git diff") which assumes that the standard git diff behavior is useful.

- The finding:
  - Default "git diff" uses "Myers" algorithm; but: "Histogram"!
    - Histogram patches on source code are easier to understand
      - (My guess: resembles more closely the edit activity/intention of the author)
    - Histogram patches have different #changed lines and #chunks!
  - But: all 52 analyzed studies use the default "git diff" for different purposes (get patches, collect metrics, SZZ)
    - Algorithm choice has a significant effect on the outcome of all three
      - SZZ: between 6% and 13% different bug-fix commits
Case 2.2 "JIT Side-Channel"

• Paper: "JVM Fuzzing for JIT-Induced Side-Channel Detection" (Brennen et al.)

• The pattern:
  – Research on *side-channel attacks* routinely uses the *Blazer collection with 14 "side-channel safe" programs* which assumes that *JVM Just-in-time compilation is deactivated* is true in real life.

• The finding:
  – Just-in-time compilation is *not* always deactivated in real life.
  – The 14 programs are indeed safe without JIT
    • but *13 are unsafe* with JIT enabled
Case 2.3 "Noisy CI-Breakage Data"

• Paper: "Studying the Impact of Noises in Build Breakage Data" (Ghaleb et al.)

• The pattern:
  – Research on continuous integration routinely uses the Travis CI logs which assumes that all Travis failures are equal is true in real life.

• The finding:
  – Logs contain 55% (!) noise in form of three types of breakage: environmental breakage, cascading (inherited) breakage, allowed breakages (e.g., experimental builds)
  – Repeated analyses:
    • Previously unsupported correlations: time of day, day of week, filetype
    • Previously inverse correlations: merge commits
    • Previously clear relationship: developer's commit frequency
Theme 3: Qualitative Coding 4 Big Data
Theme 3: Qualitative Coding 4 Big Data

• Example 3a: "How Has Forking Changed in the Last 20 Years? A Study of Hard Forks on GitHub" (Zhou et al.)
  – Distinguishes hard forks (like OpenOffice -> LibreOffice) and social forks (like a gazillion forks on GitHub)
  – Considered relationships between original/fork repositories
    • "card sorting [of] printed cards with commit history graphs of 100 randomly selected hard forks. [...] we manually built a classifier that detects the forks for each identified pattern."
    • repeat for 100 yet-unclassified forks
    • until 97.7% of 15,306 forks were classified → 15 patterns
Rather formulaic, huh?

From Interviews: Technically speaking, hard forks are often created with a social intention

Their idea: Creating a bot that warns of hard fork

But: Maybe the distinction of hard/social fork does not make sense?
Theme 3: Qualitative Coding 4 Big Data

• Example 3b: "Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts" (Rahman et al.)
  – Messages of 1,448 defect-related commits -> eight categories
  – Practitioner Agreement
  – Boolean string-based classifiers

Table 2: Rules to Detect Defect Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional</td>
<td>hasDefect(x.sen) ∧ hasCond(x.sen.deps)</td>
</tr>
<tr>
<td>Configuration</td>
<td>hasDefect(x.sen) ∧ (hasStorConf(x.sen.deps) ∨ hasFileConf(x.sen.deps) ∨ hasNetConf(x.sen.deps) ∨ hasUserConf(x.sen.deps) ∨ hasCachConf(x.sen.deps) ∨ dataChanged(x.diff))</td>
</tr>
<tr>
<td>Data</td>
<td>hasDefect(x.sen) ∧ (hasDepe(x.sen.deps) ∨ changedInclude(x.diff))</td>
</tr>
<tr>
<td>Dependency</td>
<td>hasDefect(x.sen) ∧ (hasDoc(x.sen.deps) ∨ changedComment(x.diff))</td>
</tr>
<tr>
<td>Idempotency</td>
<td>hasDefect(x.sen) ∧ hasIdent(x.sen.deps)</td>
</tr>
<tr>
<td>Security</td>
<td>hasDefect(x.sen) ∧ hasSecu(x.sen.deps)</td>
</tr>
<tr>
<td>Service</td>
<td>hasDefect(x.sen) ∧ hasServ(x.sen.deps)</td>
</tr>
<tr>
<td>Syntax</td>
<td>hasDefect(x.sen) ∧ hasSynt(x.sen.deps)</td>
</tr>
</tbody>
</table>

Table 3: String Patterns Used for Functions in Rules

<table>
<thead>
<tr>
<th>Function</th>
<th>String Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasDefect()</td>
<td>'error', 'bug', 'fix', 'issue', 'mistake', 'incorrect', 'fault', 'defect', 'flaw'</td>
</tr>
<tr>
<td>hasCond()</td>
<td>'logic', 'condition', 'boolean'</td>
</tr>
<tr>
<td>hasStorConf()</td>
<td>'sql', 'db', 'databases'</td>
</tr>
<tr>
<td>hasFileConf()</td>
<td>'file', 'permission'</td>
</tr>
<tr>
<td>hasNetConf()</td>
<td>'network', 'ip', 'address', 'port', 'tcp', 'dhcp'</td>
</tr>
<tr>
<td>hasUserConf()</td>
<td>'user', 'username', 'password'</td>
</tr>
<tr>
<td>hasCachConf()</td>
<td>'each'</td>
</tr>
<tr>
<td>hasDepe()</td>
<td>'require', 'depend', 'related', 'order', 'sync', 'compart', 'ensur', 'inherit'</td>
</tr>
<tr>
<td>hasDoc()</td>
<td>'doc', 'comment', 'spec', 'license', 'copyright', 'normal', 'header', 'readme'</td>
</tr>
<tr>
<td>hasIdent()</td>
<td>'identifier'</td>
</tr>
<tr>
<td>hasSecu()</td>
<td>'vulnerability', 'ssl', 'secret', 'authentic', 'password', 'secure', 'cve'</td>
</tr>
<tr>
<td>hasServ()</td>
<td>'service', 'server'</td>
</tr>
<tr>
<td>hasSynt()</td>
<td>'compile', 'limit', 'warn', 'typo', 'spell', 'indent', 'regex', 'variable', 'whitespace'</td>
</tr>
</tbody>
</table>
Our defect categories can correlate. Correlating categories are detectable: if rules for multiple categories are satisfied then ACID will report an ECM belonging to multiple categories. ECMs that tested positive for two categories were 1.01%, 0.05%, 1.72%, and 0.82%, respectively, for GitHub, Mozilla, Openstack, and Wikimedia.

ACID = automated categorizer for infrastructure-as-code defects
ECM = enhanced commit message (commit msg + ref. bug report)

Answer to RQ3: Configuration data is the most dominant defect category. Our identified defect categories can correlate, for example, ECMs that tested positive for two categories were 0.05%~8.01% across four datasets.
Theme 3: Qualitative Coding 4 Big Data

- Example 3c: "Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language?" (Shrestha et al.)
  - 15 language pairs, 30 StackOverflow posts for each: "qualitative coding" = labelling as revealing correct/incorrect assumptions regarding the post's target language
    - Only statistically significant differences (p<0.1):
      - <Kotlin, Java> harder than <Python, C++>,
      - <Java, C#>, or <PHP, Java>
    - Unordered pairs? What is an "assumption"? What about half-right assumptions? What about multiple assumptions in one post?
  - Paper: "We discussed disagreements on whether a post was correct or incorrect: if there was still disagreement, it was reconciled by the first author. [...] We used instances of correct and incorrect assumptions as evidence of cross-language interference and facilitation."
    - Q&A @ ICSE: "[Questions with both correct and incorrect assumptions] were a bit tricky. [...] We tried to reconcile it between the co-authors [...] if we could not agree at all, we kind of just dropped that one."
A Java developer is learning Kotlin. They ask if the following Kotlin expression can be simplified:

```kotlin
```

The developer suspects their declaration is more verbose than it should be, given their knowledge of local variable type inference in Java. They assume the declaration can be simplified:

```kotlin
val boundsBuilder = LatLngBounds.Builder()
```

This is an example of facilitation—the accepted answer confirms that the developer can simplify the expression because Kotlin supports type inference, allowing for the explicit type declaration to be removed.
BONUS
Bonus: My Pet Peeve: "Axial Coding"

We performed qualitative data analysis using a sentence-by-sentence approach in a semi-exploratory fashion. We applied selective coding [40] based on the constructs associated with the research question (i.e., positive and negative emotions, as well as strategies for coping with the latter). We identified 29 sentences discussing positive emotions triggers, 41 for negative emotions triggers, and 47 for coping strategies. Subsequently, two researchers coded each sentence following an open coding approach [40]. During a meeting, the researchers reconciled their codes in a single one. We obtained 23 codes—eight reasons for positive emotions, eight for negative ones, and seven strategies for dealing with negative emotions. These codes were then grouped to form relationships and themes captured by applying axial coding [40]. Three themes emerge: self refers to the developers’ dimension, social refers to peers and collaborators, and solution refers to issues with artifacts, design, and implementation of the task.

• Terms
  – Open, axial, and selective coding → Grounded Theory as formulated by Strauss & Corbin (1990), a 270-page book

• Here:
  – Selective coding first, based on constructs of research question
    • Not the purpose of selective coding
    • Not the time for selective coding
  – Axial coding as grouping
    • Not the purpose of axial coding

• Maybe [40] can help?
  – [40] is a 17-page journal article from 1986
  – written by neither Strauss nor Corbin
  – which does not distinguish different ways of coding
    • Coding is characterized as: "generate a set of concepts [through the] analysis [...] of incidents included in [a] set of notes"; neither "too specific" nor "too general"

"Recognizing Developers’ Emotions while Programming" (Girardi et al.)
References

• Johnson et al. (2020): Causal Testing: Understanding Defects’ Root Causes (ICSE’20)
• Sung et al. (2020): Towards Understanding and Fixing Upstream Merge Induced Conflicts in Divergent Forks: An Industrial Case Study (ICSE-SEIP’20)
• Xu et al. (2020): Why reinventing the wheels? An empirical study on library reuse and re-implementation (EMSE, Vol. 25)
• Moran et al. (2020): Improving the Effectiveness of Traceability Link Recovery using Hierarchical Bayesian Networks (ICSE’20)
• Danilova et al. (2020): One Size Does Not Fit All: A Grounded Theory and Online Survey Study of Developer Preferences for Security Warning Types (ICSE’20, no preprint available)
• Nugroho et al. (2020): How different are different diff algorithms in Git? Use --histogram for code changes (EMSE, Vol. 25)
• Brennen et al. (2020): JVM Fuzzing for JIT-Induced Side-Channel Detection (ICSE’20)
• Ghaleb et al. (2019): Studying the Impact of Noises in Build Breakage Data (TSE, future volume)
• Zhou et al. (2020): How Has Forking Changed in the Last 20 Years? A Study of Hard Forks on GitHub (ICSE’20)
• Rahman et al. (2020): Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts (ICSE’20)
• Girardi et al. (2020): Recognizing Developers’ Emotions while Programming (ICSE’20)