Empirical Studies on Bugs

Overview

- General introduction of empirical studies in SE
- Paper presentation and discussion
  - How Do Fixes Become Bugs?[2]
  - Programmers’ Build Errors: Case Study (at Google) [3]
- Threats to Validity
Empirical Research [1]

- Research using empirical evidence. It is a way of gaining knowledge by means of direct or indirect observation or experience.
- Empirical evidence can be analyzed quantitatively or qualitatively
- Researcher answers empirical questions, which should be clearly defined and answerable with the evidence collected

Empirical Studies in SE

- To understand how developers build or maintain software by observing various software artifacts or monitoring software runtime behaviors
- Can be conducted with manual inspection or automatic tools
- May achieve various research goals:
  - identify software change patterns
  - reveal relations between symptoms and root causes
  - shed light on new technique design and impl.
Characteristics of Empirical Studies

- Cool algorithm design or intensive programming is NOT always required
  - Sometimes only manual inspection and eyeball checking is done
- “Interesting Research Questions” is the key contribution
  - Questions haven’t been asked or answered nicely
  - Questions whose answers can provide actionable advice to tool builders or users

How Do Fixes Become Bugs? [2]
A Comprehensive Characteristic Study on Incorrect Fixes in Commercial and Open Source Operating System
Problem Statement

• Bug fixes can be wrong, and thus fail to fix the bugs or introduce new bugs
• Incorrect bugs fixes can cause serious problems
  – Trend Micro lose $8 million for a buggy patch
• Various reasons can cause buggy fixes
  – Time pressure, lack of knowledge
• What is the characteristic of incorrect bug fixes?

Research Questions

• How significant is the problem of incorrect fixes?
  – % of incorrect fixes, severity of caused problems
• What types of bugs are difficult to fix correctly?
• What are the common mistakes made in bug fixes?
Research Questions

• What aspects in the development process are correlated to the correctness of bug fixing?
  – Does fixers and reviewers' knowledge is highly correlated to incorrect fixes?

Methodology

• Four software projects under study
  – One commercial OS, three open source OS (FreeBSD, Linux, OpenSolaris)
• Randomly select a target number, e.g., 2000, of bug fixes, and only examine the 970 post-release bugs
• Find incorrect fixes
  – identify bug fixing changes
  – identify incorrect ones
• Measure code knowledge
Finding Bug Fixing Changes

• The commercial OS and OpenSolaris well maintain the link between a fix and a bug report
• For the other two OS, if the commit message or bug report contains words like “the bug is fixed by change 0a134fad”, the bug is associated with the change 0a134fad

Finding Incorrect Bug Fixes

• Step 1: Automatic filtering
  – If two bug fixes have source code overlap, the latter one is to correct the former one
  – If the bug report of Bug A contains any reference to Bug B, the fix for Bug B is incorrect
• Step 2: manual examination and some discussion with developers
Measuring Code Knowledge

- Heuristic: calculate knowledge of developers for code based on their contribution for the code base

\[ K_{File_d,F,v} = \frac{\text{The LoC written by } d \text{ for } F \text{ at } v}{\text{The total LoC in } F \text{ at } v} \]

\[ K_{Func_d,f,F,v} = \frac{\text{The LoC written by } d \text{ for } f \text{ in } F \text{ at } v}{\text{The total LoC in the } f \text{ at } v} \]

RQ1: How significant is the problem of incorrect fixes?

<table>
<thead>
<tr>
<th>App</th>
<th># of post-release bug fixes</th>
<th># of incorrect fixes</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>189</td>
<td>39</td>
<td>20.6%±3.0%</td>
</tr>
<tr>
<td>B</td>
<td>309</td>
<td>46</td>
<td>14.8%±2.9%</td>
</tr>
<tr>
<td>C</td>
<td>267</td>
<td>41</td>
<td>15.3%±2.6%</td>
</tr>
<tr>
<td>D</td>
<td>205</td>
<td>50</td>
<td>24.4%±3.7%</td>
</tr>
</tbody>
</table>

- 14.8–24.4% bug fixes are incorrect. Since the fixes should be applied by a lot of customers and users, this ratio can have significant impact.
Impact of the Incorrect Bug Fixes

- 14% introduce crash
- 8.4% cause system to hang
- 15.4% lead to data corruption or data loss
- 5.6% cause security problem
- 7% degrade the performance
- 45.1% introduce incorrect functionality

RQ2: What types of bugs are difficult to fix correctly?

- 970 bug fixes are classified into three categories: memory bugs, concurrency bugs, and semantic bugs

<table>
<thead>
<tr>
<th>App</th>
<th>Concurrency</th>
<th>Memory</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4/13 (31%)</td>
<td>3/17 (18%)</td>
<td>32/159 (20%)</td>
</tr>
<tr>
<td>B</td>
<td>9/21 (43%)</td>
<td>5/44 (13%)</td>
<td>32/244 (13%)</td>
</tr>
<tr>
<td>C</td>
<td>7/19 (37%)</td>
<td>6/43 (14%)</td>
<td>28/205 (14%)</td>
</tr>
<tr>
<td>D</td>
<td>10/23 (44%)</td>
<td>5/30 (17%)</td>
<td>35/152 (23%)</td>
</tr>
<tr>
<td>Overall</td>
<td>30/76 (39%)</td>
<td>19/134 (14%)</td>
<td>127/760 (17%)</td>
</tr>
</tbody>
</table>

- Developers and testers should be more cautious when fixing concurrency bugs
The Most Observed Detailed Bug Types

<table>
<thead>
<tr>
<th>Bug types and their percentages</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>data race</td>
<td>33%</td>
</tr>
<tr>
<td>buffer overflow</td>
<td>8%</td>
</tr>
<tr>
<td>uninitialized read</td>
<td>4%</td>
</tr>
<tr>
<td>deadlock</td>
<td>29%</td>
</tr>
<tr>
<td>memory leak</td>
<td>6%</td>
</tr>
<tr>
<td>null pointer deref</td>
<td>4%</td>
</tr>
</tbody>
</table>

• Q: Are these bugs generally difficult to fix? Or is this observation unique to incorrect bug fixes?

RQ3: What are the common mistakes made in bug fixes?

• data races
  – When adding locks, some methods should not be put into the critical section

• deadlock
  – When reversing the order of locks or dropping some locks, all execution paths should be tested
Common Mistakes in Bug Fixing

• buffer overflow
  – Increase buffer size or allocate larger buffer without checking

• memory leak
  – Fixing memory leak can introduce dangling pointer or null pointer dereference

• semantic bugs
  – Wrongly changed conditions

RQ4: Correlation between developers’ knowledge and incorrectness

• The more knowledge a fixer has, the fewer incorrect bug fixes he commits
Conclusion

• This paper presents one of the most comprehensive characteristic studies on incorrect bug fixes
• It studies the common patterns of mistakes made in incorrect fixes
• 27% incorrect bug fixes are made by developers who haven’t contributed a single line to the file under fix

Programmers’ Build Errors: A Case Study (at Google) [3]
Problem Statement

- Building is important when developers use compilers, linkers, build files and scripts to assemble their code into executable units
- Slow and failed builds can affect programmer productivity
- Keeping the build process fast and understanding when and how it fails is essential to improve productivity

Google’s Build Environment

- Google’s cloud-based centralized build system
- BUILD files specify the build configuration
  - the list of targets (tests) to build
  - the sources and dependent targets to compile
  - the compilers to use
- Developers may build with the IDE’s own compiler during development, but must ensure the code successfully builds with the centralized system
Research Questions

• How often do builds fail?
  – By reporting the failure-to-success build ratio, the findings help us understand the potential impact of changes

• Why do builds fail?
  – The findings can motivate or prioritize new automatic checking

• How long does it take to fix broken builds?
  – The findings highlight hard-to-fix errors

Methodology

• The build system saves all build logs describing
  – The result of each build (succeeded or failed)
  – The compilation errors

• Examine build logs in nine months for Java and C++ projects

• Focus on builds done by standard developers
  – Active developers without requesting automatic builds
How to Count Build Errors?

- Two ways investigated
  - Count-all: # of all error messages in one build
  - Count-distinct: # all unique error kinds in one build
- The second one is better
  - As one fault, such as missing import, can cause a lot of name resolution errors, the error should be counted only once

How to Calculate Resolution Time?

- If a developer has a series of failure builds before a successful one, and the failure only contains one error
  - Resolution time = Time_{success} - Time_{1stFailure}
RQ1: How often do builds fail?

- The failure ratio is approximately normally distributed
- The median percentage is 38.4% for C++ and 28.5% for Java

RQ2: Why do builds fail?

- Normalize the errors
- Cluster them to report the top five errors, which constitute 80% of all errors
### Categories of Error Messages - Java

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependency</td>
<td>cannot.resolve</td>
<td>cannot find symbol</td>
</tr>
<tr>
<td></td>
<td>doesn't.exist</td>
<td>package does not exist</td>
</tr>
<tr>
<td></td>
<td>strict</td>
<td>Google-specific dependency check</td>
</tr>
<tr>
<td>Type mismatch</td>
<td>can't.apply.symbol</td>
<td>method called with incorrect argument list</td>
</tr>
<tr>
<td></td>
<td>can't.apply.symbol.i</td>
<td>method called with incorrect argument list</td>
</tr>
<tr>
<td></td>
<td>incompatible.types</td>
<td>method called with incorrect argument list</td>
</tr>
<tr>
<td></td>
<td>unchecked</td>
<td>method called with incorrect argument list</td>
</tr>
<tr>
<td></td>
<td>can't.apply.symbol.l</td>
<td>unchecked conversion from rawtype to parameterized type</td>
</tr>
<tr>
<td></td>
<td>rawtypes</td>
<td>use of raw type rather than parameterized type</td>
</tr>
<tr>
<td></td>
<td>operator.cant.be.applied</td>
<td>wrong types for operator</td>
</tr>
<tr>
<td>Syntax</td>
<td>expected</td>
<td>certain keyword or symbol was expected and not found</td>
</tr>
<tr>
<td></td>
<td>illegal.start.of.expr</td>
<td>illegal start of expression</td>
</tr>
<tr>
<td></td>
<td>notstmt</td>
<td>statement expected but not found</td>
</tr>
<tr>
<td></td>
<td>missing.ret.stat</td>
<td>missing return</td>
</tr>
<tr>
<td></td>
<td>illegal.start.of.type</td>
<td>illegal start of type</td>
</tr>
<tr>
<td></td>
<td>expected3</td>
<td>certain keyword or symbol was expected and not found</td>
</tr>
<tr>
<td>Semantic</td>
<td>method.does.not.override.superclass</td>
<td>method has @Override annotation but does not override a superclass or interface method</td>
</tr>
<tr>
<td></td>
<td>does.not.override.abstract</td>
<td>concrete class must override abstract method in superclass or interface</td>
</tr>
<tr>
<td></td>
<td>non-static.cant.be.ref</td>
<td>non-static cannot be referenced from static context</td>
</tr>
<tr>
<td></td>
<td>report.access</td>
<td>access disallowed by visibility modifier</td>
</tr>
<tr>
<td></td>
<td>static.imp.only.classes.and.interfaces</td>
<td>cannot static import this symbol is not public and cannot be accessed</td>
</tr>
<tr>
<td></td>
<td>not-def.public.cant.access</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>unreported.exception.need.to.catch.or</td>
<td>checked exception must be caught or thrown</td>
</tr>
<tr>
<td></td>
<td>throw</td>
<td>symbol already defined</td>
</tr>
<tr>
<td></td>
<td>already.declared</td>
<td>variable might not have been initialized</td>
</tr>
</tbody>
</table>

### Categories of Error Messages - C++

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependency</td>
<td>undeclared.var.use</td>
<td>use of an undeclared identifier</td>
</tr>
<tr>
<td></td>
<td>no.member</td>
<td>no member (e.g. class variable) in the given class</td>
</tr>
<tr>
<td></td>
<td>undeclared.var.use.suggest</td>
<td>use of an undeclared identifier, with a suggested fix</td>
</tr>
<tr>
<td></td>
<td>unknown.type.name.suggest</td>
<td>unknown type name, with a suggested fix</td>
</tr>
<tr>
<td></td>
<td>unknown.type.name</td>
<td>use of an undeclared identifier, with a suggested fix</td>
</tr>
<tr>
<td></td>
<td>no.member.overloaded.arrow</td>
<td>no member (e.g. class variable) in the given class</td>
</tr>
<tr>
<td></td>
<td>member.decl.does.not.match</td>
<td>out of line definition does not match any declaration</td>
</tr>
<tr>
<td></td>
<td>type.name.nested.not.match</td>
<td>could not find header file</td>
</tr>
<tr>
<td></td>
<td>incomplete.member.access</td>
<td>no type with given name within the specified class or namespace</td>
</tr>
<tr>
<td>Type mismatch</td>
<td>ov1.no.viable.function.in.call</td>
<td>member access into incomplete type</td>
</tr>
<tr>
<td></td>
<td>typecheck.nonviable.condition</td>
<td>calling an undeclared version of an overloaded function</td>
</tr>
<tr>
<td></td>
<td>ov1.no.viable.function.in_init</td>
<td>cannot convert</td>
</tr>
<tr>
<td></td>
<td>init.conversion.failed</td>
<td>no matching constructor for the initialization expression</td>
</tr>
<tr>
<td></td>
<td>ov1.no.viable_member.function.in_call</td>
<td>cannot initialize with the passed-in value</td>
</tr>
<tr>
<td></td>
<td>typecheck.member.reference.suggestion</td>
<td>no matching member function for the call</td>
</tr>
<tr>
<td></td>
<td>typecheck.invalid_operands</td>
<td>using '&lt;:' instead of '::', with a suggested fix</td>
</tr>
<tr>
<td></td>
<td>typecheck.call.too.few_args</td>
<td>invalid operands to binary expression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>too few arguments to a function call</td>
</tr>
<tr>
<td>Syntax</td>
<td>expected.expression</td>
<td>expected an expression</td>
</tr>
<tr>
<td></td>
<td>expected.rparen</td>
<td>expected right parenthesis</td>
</tr>
<tr>
<td></td>
<td>expected.unqualified.id</td>
<td>expected an identifier</td>
</tr>
<tr>
<td></td>
<td>bound.member.function</td>
<td>reference to a non-static member function</td>
</tr>
<tr>
<td>Semantic</td>
<td>invoke</td>
<td>accessing a protected or private member</td>
</tr>
<tr>
<td>Other</td>
<td>non.virtual.dtor</td>
<td>virtual functions but no virtual destructor</td>
</tr>
</tbody>
</table>
Why Do These Errors Happen?

• Dependency
  – missing dependency between source code components due to significant usage of IDE

• Type mismatch
  – assign a variable to an incompatible type or call a function with wrong argument types

• Syntax

• Semantics
  – violating class access rule or unimplemented abstract methods

• Q1: Why do the last three happen?

RQ3: How long does it take to fix builds? - Java

• Top 5 expensive errors (freq. * median resolution time)
  – cant.resolve, doesnt.exist, strict, cant.apply.symbol, cant.apply.symbol.1

• Error with most resolution time: does.not.override.abstract
RQ3: How long does it take to fix builds? - C++

• Top 5 expensive errors (freq. * median resolution time)
  – undeclared_var_use,
    undeclared_var_use_suggest, no_member,
    ovl_no_viable_function_in_init,
    typecheck_nonviable_condition
• Error with most resolution time:
  expected_rparen

Implications

• 10% of the error types account for 90% of the build failures
• Better tools to resolve dependency errors have the greatest potential payoff
Threats to Validity

Man prefers to believe what he prefers to be true.

-- Francis Bacon

Threats to Validity

- Is the investigator's conclusion correct?
- Try to identify the factors which make your conclusion incorrect
External & Internal Validity

• External validity
  – The degree to which the results of an empirical investigation can be generalized to and across individuals, settings, and times
    • “Is the conclusion generalizable?”

• Internal validity
  – The degree to which a causal conclusion based on a study is warranted
    • “Is the experiment done correctly?”

Threats to External Validity

• Aptitude
  – If a medicine is effective for sample patients, will it also be effective for non-volunteers or all other people?

• Situation
  – time, location, scope and extent of measurement
Threats to External Validity

• Pre-test effects
  – The cause-effect relationship can be found when pre-tests are carried out

• Post-test effects
  – The cause-effect relationship can only be found when post-tests are carried out

• ...

Examples

• The empirical study is performed within a single company with particular processes, constraints, resources, and tools

• The empirical study is done on operating system software/open source projects
Threats to Internal Validity

- **Confounding**
  - Changes in the observation may be related to multiple variables
- **Selection bias**
  - Samples should be chosen without bias
- **Instrument change**
  - The measurement may affect the result
- **John Henry effect**
  - John Henry was a worker who outperformed a machine under an experimental setting because he was aware that his performance was compared with that of a machine.

Examples

- The execution time reading may significantly affect the measured execution time
- The causal-effect relationship between bugs and bad variable names may be affected by factors like complexity of functionality, maturity of developers, and types of bugs
Importance of Threat Identification

- Help researchers decide how to propose research questions and do experiments in a plausible way
- Help people understand limitation of the research
- It is OK that you can’t avoid all threats. However, you should try your best to make your results representative and meaningful

Reference

[3] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, Robert Bowdidge, Programmers’ Build Errors: A Case Study (at Google), ICSE’ 14