What do we know about code change?
A Survey of Version Control Studies of Software Change

Thomas LaToza
2/17/2005

What makes good research in software engineering?

Version control studies

- Instrument with which to study SE
  - Particularly how code changes

- Advantages
  - LOTS of data
  - >1000s of checkins
  - >1000s of files
  - REAL projects over LONG periods of time (up to 20 years)
  - CHEAP to collect (copy the repository)
  - But probably still need to talk to people to get background data

- Disadvantages
  - High granularity
    - What was the developer doing before checkin?
    - No experimental control (correlational)
    - => Messy
  - Might not have all of the variables of interest
    - What process were they using?
    - Development environment
    - How much up front design?
    - What expertise do developers have?
Data in a Version Control Study

- Modification request logs
  - Data about what they are doing
  - Time
  - Reason
  - Developer identity
- Checkins - version control system
  - # / size of components (functions, classes, files, directories, packages, ...)
  - # / type of connectors (call dependencies, ...)
  - Location of changed lines
  - Developer identity
  - Time & date
- Bug logs
  - Cause
  - Where fix is localized to
  - Time to fix
  - Developer identity
- Other
  - Programming language
  - Type of application

Questions answered by major version control studies

- Prehistory – reasons for change
- How is change distributed over time and structure?
- What causes bugs?
- Does code decay?
- How is OSS different?

Why does code change?

- Not an empirical study! – just influential
- Definitive classification of types of maintenance
  - Data based, more specific ones more recently proposed
- Corrective – fixing bugs
  - Processing failure
  - Performance failure
  - Implementation failure
- Adaptive – change in context [central problem]
  - Change in data environment
  - Change in processing environment
- Perfective – “maintenance for the sake of maintenance”
  - Processing inefficiency
  - Performance enhancement
  - Maintainability

How is change distributed over time and structure?

- OS/360 over 12 years, 20 releases
- Doesn’t use a version control system (?)
  - Data collected by hand at each release
- “Laws” of program evolution
  - 1. Law of continuing change
    - A system that is used undergoes continuing change until it is judged more cost effective to recreate it
  - 2. Law of increasing entropy
    - Unstructuredness increases with time without spending time structuring it
    - Growth trend looks stochastic and self regulates

The Dimensions of Maintenance E.B. Swanson, ICSE (1976)

A model of Large Program Development LA Belady & MM Lehman, IBM Systems Journal (1976)
How is change distributed over time and structure?

- Replicates on 2 other systems
  - Necessary to argue that properties are general, not unique to OS/360 development
  - Confirms or does not disconfirm laws
- OS/360
  - "Executive system"
    - Algol like language
    - For inhouse customers
    - 10% size of OS/360
  - "Transaction system"
    - Assembly language
    - Maintenance period data, different developers than construction
    - Limited number of customers
    - 50% size of OS/360

Program Evolution and Its Impact on Software Engineering
MM Lehman & F. N. Parr. ICSE (1976)

How is change distributed over time and structure?

- 5 business systems; total of 621 modules
- Support for Pareto principle
  - 27% of modules received 80% changes
- Adaptive changes - 26% modules received 80%
  - Modules whose functionality is strategic
- Corrective changes - 17% modules received 80%
  - Higher complexity, larger, older
- Perfective changes - 14% modules received 80%

Determinants of Software Maintenance Profiles: An Empirical Investigation

How is change distributed over time and structure?

<table>
<thead>
<tr>
<th>SYSTEM NAME</th>
<th>SYSTEM NAME</th>
<th>FROM TO</th>
<th>SIZE</th>
<th>VSTLM</th>
<th>PCT</th>
<th>NO. OF COLLABORS</th>
<th>NO. OF COLLABORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS/360</td>
<td>Operating System</td>
<td>10/76</td>
<td>10/77</td>
<td>1126</td>
<td>1200</td>
<td>22</td>
<td>Many</td>
</tr>
<tr>
<td>OMEGA</td>
<td>Operating System</td>
<td>10/74</td>
<td>11/74</td>
<td>333</td>
<td>388</td>
<td>9</td>
<td>Many</td>
</tr>
<tr>
<td>DOS</td>
<td>Operating System</td>
<td>8/73</td>
<td>6/76</td>
<td>646</td>
<td>246</td>
<td>31</td>
<td>Many</td>
</tr>
<tr>
<td>B</td>
<td>Operating System</td>
<td>11/74</td>
<td>11/75</td>
<td>1728</td>
<td>2437</td>
<td>6</td>
<td>Many</td>
</tr>
<tr>
<td>EXECUTIVE</td>
<td>Systems Support</td>
<td>11/75</td>
<td>11/76</td>
<td>657</td>
<td>967</td>
<td>12</td>
<td>2 Sales</td>
</tr>
<tr>
<td>BO</td>
<td>Application</td>
<td>11/75</td>
<td>11/75</td>
<td>42</td>
<td>86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>Application</td>
<td>12/74</td>
<td>13/74</td>
<td>3483</td>
<td>46</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

*The first year of data collection on DOS is not shown.

- Results
  - Generally confirms laws 1 and 2
  - Disconfirms law 3 – growth not smooth

An Examination of Evolution Dynamics
M. J. Lawrence, ICSE (1982)

What causes bugs?

- 90 KLOC mostly Fortran satellite mission program
- # functions changed for bug fixes
  - 89% - 1, 6% - 2, 2% - 3, 4% - 4, 0.5% - 5
- Bug causes recorded by bug fixer
  - Requirements misinterpreted – 12%
  - Functional spec incorrect or misinterpreted – 36%
  - Design error w/ multiple functions – 4%
  - Design error w/ single function – 22%
  - Clerical – 12%
  - Incorrect bug fix – 8%
- Previously modified functions had even higher percentage of functional spec bugs
  - Specifications decay with changes?
- Larger functions, fewer bugs per LOC

Software Errors and Complexity – An Empirical Investigation
Does code decay?

- Data - 1.5 MLOC telephone switch (5ESS) subsystem
  - C/C++, few hundred developers, ~130,000 changes
  - Predict 1,500 bug fixes
- Code decay = any change produces many bugs
- Code decays
  - # changes and age independently predict future bugs
  - Best predictor is # changes with each change contribution decaying with time (weighted time damped model)
- # developers who had changed a function doesn’t predict # bugs
- Relevance relation poor predictor of bugs
- Aside – complexity measures are worthless
  - Predictiveness of bugs nearly completely captured by LOC
  - Cyclomatic complexity, # functions, #variables, # operators, others
  - Coupling measure was more predictive

Does Code Decay? Assessing the evidence from change management data
Stephen Eick, Todd Graves, Alan Karr, J.S. Marron, Audris Mockus, TSE (2001)

Does code decay?

- Data – entire 5ESS – 100 MLOC + 100 M lines of make / header files
  - 5,000 directories; 10,000 developers over 15 years
- Code decays
  - Time to implement changes increases
    - Less structured, more work to do
  - Number of files touched in a change increases with time
  - Effectiveness of modularity decreases over time

Predicting Fault Incidence Using Software Change History

How does Open Source software change?

- Data - Mozilla and Apache
  - Not pure version control study, as lots of background info
  - Core developers - >= 80% new functionality
    - No larger than 10 to 15 people with only informal communication
    - Too difficult to coordinate
    - If larger, use explicit process, code ownership, or inspections
  - Other project members
    - Order of magnitude more developers fix bugs
    - Another order of magnitude more report bugs
  - Developers are users
    - Low defect densities achieved because of domain expertise from use

Two Case Studies of Open Source Software Development: Apache and Mozilla
Cumulative contributions for 5 Mozilla releases

Questions?