Role of change history in empirical studies of software

Audris Mockus
http://www.bell-labs.com/~audris
Outline

- Motivation and goals
  - Why changes are made?
    - how to obtain the purpose
  - Why changes are hard?
    - how to obtain change effort
- Implications
Motivation

Example
- Real, 20 year old, huge switching product
  - large proportion of changes are enhancements

Advantages of change history data
- ubiquitous - most products have it
- massive - far larger than any survey
- complete - all parts of software are recorded
- unbiased - no Hawthorne effect
- uniform over time
Goal

- Design tools and methods that do not compromise advantages of change history data, i.e.,
  1. Are uniformly applicable
  2. Minimize human involvement
  3. Use existing data
  4. Complete - characterize all parts of software
Great, but can it be done?

- Change history contains
  - who changed, when and what was changed
- But is it possible to obtain:
  - why?
  - how difficult?
- Two studies on
  - purpose (with L. Votta)
  - effort (with T. Graves)
How Code Evolves

- By adding and deleting line blocks

before:  
```c
// initialize
int i=0;  int i=0;
while (i++)
    read (x);
```

after:  
```c
// initialize
int i=0;  int i=0;
while (i++ < N)
    read (x);  read(x);
```

- one line deleted
- two lines added
- two lines unchanged
Any VCS Records:

- Change - added and deleted lines
  - Who - login, organization
  - When - date and time
  - Description - line of text
- Available data:
  - ~100M lines, ~4M changes, ~5K logins, 30Gb
  - ~30 products (select one)
Why code is changed?

- Primary reasons for maintenance activities
  - corrective: fix faults
  - adaptive: add features
- How those reasons relate to:
  - interval, effort, quality
  - developer, size
  - location, time
How to obtain the purpose?

- Look for bug/new field
  - may not be there, unreliable, only two values
- Ask developers
  - too much overhead - small coverage
- Read change abstracts
  - great idea - but 2M abstracts
- Let computer read abstracts
  - but how?
Algorithm

- Use change description line
  - extract frequent keywords
    - classify keywords (fix, new, add, etc.)
      - discover new types
        • perfective - code cleanup
        • inspection - code inspection suggestions
    - verify on sample abstracts
  - keyword -> purpose of the change
  - iterate
Example keywords

<table>
<thead>
<tr>
<th>Adaptive:</th>
<th>Corrective:</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, new, create,</td>
<td>fix, bug, error,</td>
</tr>
<tr>
<td>initial coding, modify, update</td>
<td>problem, incorrect, must, needs</td>
</tr>
<tr>
<td>Perfective:</td>
<td>Inspection:</td>
</tr>
<tr>
<td>cleanup, remove,</td>
<td>code review, inspection, rework, walkthrough</td>
</tr>
<tr>
<td>clear, unneeded,</td>
<td>clear, unneeded, flex name</td>
</tr>
<tr>
<td>flex name</td>
<td></td>
</tr>
</tbody>
</table>
Proportions

Why:
- add new functionality - 45%
  - fix faults (bug) - 34%
  - cleanup/restructure - 4%
  - code inspection - 5%
- unclassified - 12%
Is it right?

- Survey:
  - 2+5 developers (>9 years experience)
  - 20+150 changes (< 2 years old)
    - ~ equal numbers for different types
    - small percent of all changes developers did

- Questions
  - Type: bug, new, cleanup, inspection
  - Difficulty: Easy, Medium, Hard
Results

- Unclassified changes are mostly bug fixes
  - Almost perfect match
- Inspection changes are easiest to detect

<table>
<thead>
<tr>
<th>Dev./Auto</th>
<th>Corrective</th>
<th>Adaptive</th>
<th>Perfective</th>
<th>Inspection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corrective</td>
<td>35</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Adaptive</td>
<td>11</td>
<td>23</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Perfective</td>
<td>10</td>
<td>8</td>
<td>27</td>
<td>9</td>
</tr>
<tr>
<td>Inspection</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>21</td>
</tr>
</tbody>
</table>
Will it work elsewhere?

- Other Product
  - 2 X size and five years older
    - different functionality
    - different organization

- Tool
  - the same classification (no manual input)

- Results
  - very similar purpose profiles
Effort Estimation

How difficult a change was?
What makes changes difficult?
Where difficult changes are?
Why change effort

- detect key factors that affect effort
  - in a larger project change type, size, and developer are aggregated over many changes and their effects cannot be detected
How to get effort?

- ask developers
  - small coverage, large effort
- use developer reported monthly effort
  - divide among changes made that month
- simplification
  - developers report similar effort every month
  - hence reported effort can be replaced by 1
Algorithm

- Specify factors that might contribute to effort
  - Use reported effort (unit monthly effort if reported effort unavailable) to estimate contributions from each factor
  - Use cross-validation to determine significance of each factor
Example

\[ \text{ChangeEffort} \sim \text{Purpose} + \text{Size} + \text{Login} \]
\[ + \text{Decay} + \text{FileType} + \text{otherFactors} \]

- Choose factors that may affect effort
  - base factors: purpose, size, developer
    - test factors: e.g., complexity, decay, ...

- Result
  - the value and significance of each factor
  - e.g. effort for a similar change \( \uparrow \) 20%/year
### Example Factor Estimates

11 developers from 5ESS OA

<table>
<thead>
<tr>
<th>Factor</th>
<th>Effect</th>
<th>Significant</th>
</tr>
</thead>
<tbody>
<tr>
<td>purpose</td>
<td>bug change takes twice more effort than new change</td>
<td>yes</td>
</tr>
<tr>
<td>size</td>
<td>effort is proportional to: #delta, #files, #lines</td>
<td>yes</td>
</tr>
<tr>
<td>login</td>
<td>effort to make a similar change can vary 3 times across logins</td>
<td>no</td>
</tr>
<tr>
<td>decay</td>
<td>making a similar change takes 5-25% more effort each year</td>
<td>yes</td>
</tr>
<tr>
<td>sdl versus c</td>
<td>no effect</td>
<td>no</td>
</tr>
</tbody>
</table>
Applications

- SoftChange system - prototype tool
  - Other applications
    - monitoring (where/when code decays)
      - expertise locator (who is the best match)
    - process/tool evaluation (is there any effect)
      - Version Editor and process capability studies
    - benchmarking - 4 projects
More results

- Assessing code decay
  - Predicting fault potential
  - Complexity of parallel changes
  - How legacy organizations cope with changing business environment
Summary

- Change history is invaluable
  - automatically it can be enhanced with
    - purpose
    - effort
  - Cost drivers can then be determined
Summary

- Change history is invaluable
  - automatically it can be enhanced with
    - purpose
    - effort
  - Cost drivers can then be determined
- Don’t forget change history in your next study!
SoftChange: highlights

- ECMS/SABLIME + SCCS interface
  - Summarization (5ESS ~ 30Gb data)
    - developer, size, time, interval, #files, #delta
  - Financial Support System (FSS) interface
    - person, monthly effort
- Reliable automatic MR classification
  - bug, new, code improvement
- Change effort estimation
Architecture

Access Engines

Summary Engine

User Interface

Analysis Engines $\lambda \sim \Sigma \mu$