Predicting Fault-Prone Modules Based on Metrics Transitions

Yoshiki Higo, Kenji Murao, Shinji Kusumoto, Katsuro Inoue
{higo,k-murao,kusumoto,inoue}@ist.osaka-u.ac.jp
Outline

• Background
• Preliminaries
  – Software Metrics
  – Version Control System
• Proposal
  – Predict fault-prone modules
• Case Study
• Conclusion
Background

• It is becoming more and more difficult for developers to devote their energies to all modules of a developing system
  – Larger and more complex
  – Faster time to market

• It is important to identify modules that hinder software development and maintenance, and we should concentrate on such modules
  – Manual identification requires much costs depending on the size of the target software

  **Automatic identification is essential for efficient software development and maintenance**
Preliminaries -Software Metrics-

- Measures for evaluating various attributes of software
- There are many software metrics

- CK metrics suite is one of the most widely used metrics
  - CK metrics suite evaluates complexities of OO systems from
    - Inheritance (DIT, NOC)
    - Coupling between classes (RFC, CBO)
    - Complexity within each class (WMC, LCOM)
  - CK metrics suite is a good indicator to predict fault-prone classes[1]

Preliminaries - Version Control System -

• Tool for efficiently developing and maintaining software systems with many other developers

• Every developer
  1. gets a copy of the software from the repository (checkout)
  2. modifies the copy
  3. sends the modified copy to the repository (commit)

• The repository contains various data
  – Modified code of every commitment
  – Developer names of every commitment
  – Commitment time of every commitment
  – Log messages of every commitment
Motivation

- Software Metrics evaluate the latest (or the past) software product
  - They represent the states of the software at the version
- How the software evolved is an important attribute of the software
Motivation -example-

• In the latest version, the complexity of a certain module is high
  – The complexity of the module is stable at high through multiple versions?
  – The complexity is getting higher according to development progress?
  – The complexity is up and down through the development?

• The stability of metrics is an indicator of maintainability
  – If the complexity is stable, the module may not be problematic
  – If the complexity is unstable, big changes may be added repeatedly
Proposal: Metrics Constancy

• Metrics Constancy (MC) is proposed for identifying problematic modules
  – MC evaluates the changeability of the metrics of each module
• MC is calculated using the following statistical tools
  – Entropy
  – Normalized Entropy
  – Quartile Deviation
  – Quartile Dispersion Coefficient
  – Hamming Distance
  – Euclidean Distance
  – Mahalanobis Distance
Entropy

• An indicator to represent the degree of uncertainty
• Given that MC is uncertainty of metrics, Entropy can be used as a measure of MC

\[ H = - \sum p_i \log p_i \quad (p_i \text{ is probability}) \]

- **m1**: 5 changes, value 2: 4 times, value 3: 1 time
  \[ H = -(\frac{4}{5} \log \frac{4}{5} + \frac{1}{5} \log \frac{1}{5}) \approx 0.72 \]

- **m2**: 5 changes, value 1,2,3: 1 time, value 4: 2 times
  \[ H = -(3 \times \frac{1}{5} \log \frac{1}{5} + \frac{2}{5} \log \frac{2}{5}) \approx 1.9 \]

- **m3**: 3 changes, value 1,3,4: 1 time
  \[ H = -(3 \times \frac{1}{3} \log \frac{1}{3}) \approx 1.6 \]
Calculating MC from Entropy

- MC of module $i$ is calculated using the following formula
  \[ MC(i) = \sum_{MT} H \]
  - $MT$ is a set of used metrics

- The more unstable the metrics of module $i$ are, the greater $MC(i)$ is
Procedure for calculating MC

• STEP1: Retrieves snapshots
  – A snapshot is a set of source files just after at least one source file in the repository was updated by a commitment

• STEP2: Measures metrics from all of the snapshots
  – It is necessary to select appropriate software metrics fitting for the purpose
    • If the unit of modules is class, class metrics should be used
    • If we focus on the coupling/cohesion of the target software, coupling/cohesion metrics should be used

• STEP3: Calculates MC
  – Currently, the 7 MCs are calculated
Case Study: Outline

- **Target:** open source software written in Java
  - FreeMind, JHotDraw, HelpSetMaker
- **Module:** class (≡ source file)
- **Used Metrics:** CK Metrics, LOC

<table>
<thead>
<tr>
<th>Software</th>
<th>FreeMind</th>
<th>JHotDraw</th>
<th>HelpSetMaker</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Developers</td>
<td>12</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td># of snapshots</td>
<td>104</td>
<td>196</td>
<td>260</td>
</tr>
<tr>
<td># first source files</td>
<td>67</td>
<td>144</td>
<td>14</td>
</tr>
<tr>
<td># last source files</td>
<td>80</td>
<td>484</td>
<td>36</td>
</tr>
<tr>
<td>First total LOC</td>
<td>3,882</td>
<td>12,781</td>
<td>797</td>
</tr>
<tr>
<td>Last total LOC</td>
<td>14,076</td>
<td>60,430</td>
<td>9,167</td>
</tr>
</tbody>
</table>
Case Study: Procedure

1. Divides snapshots into anterior set (1/3) and posterior set (2/3)

2. Calculates MCs from the anterior set
   – Metrics of the last version in the anterior set were used for comparison

3. Identifies bug fixes from the posterior set
   – Commitments including both `bug’’ and `fix’’ in their log messages were regarded as bug fixes

4. Sorts the target classes in the order of MCs and raw metrics values
   – Also, bug coverage is calculated based on the orders
Case Study: Results (FreeMind)

• MCs could identify fault-prone classes more precisely than raw metrics
  – RED: MCs
  – BLUE: raw metrics

• At top 20% files
  – MCs: 94-100% bugs
  – Raw: 30-80% bugs
Case Study: Results (Other software)

- For all of the 3 software, MCs could identify fault-prone classes more precisely than raw metrics
Case study: different breakpoints

- In this case study, we used 3 breakpoints
  - 1/4, 1/3, 1/2

- The previous graphs are the results in case that anterior set is 1/3
Case study: Results (different breakpoints)

- MC’s identifications are good on all of the breakpoints
Case study: Results (different breakpoints)

<table>
<thead>
<tr>
<th>Top 20% files bug coverage, FreeMind</th>
<th>1/4</th>
<th>1/3</th>
<th>1/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCs</td>
<td>94-100%</td>
<td>94-100%</td>
<td>97-100%</td>
</tr>
<tr>
<td>Raw metrics</td>
<td>22-72%</td>
<td>30-80%</td>
<td>22-86%</td>
</tr>
</tbody>
</table>

- MC’s bug coverage is very high for all of the breakpoints
- Raw metrics are not suited for predicting far future
Discussion

• MCs are good indicators for identifying fault-prone modules as well as CK metrics
  – FreeMind
    • MCs: 95-100% bugs
    • Raw: 30-80% bugs
  – JHotDraw
    • MCs: 44-59% bugs
    • Raw: 10-48% bugs
  – HelpSetMaker
    • MCs: 60-75% bugs
    • Raw: 28-63% bugs

• Calculating MCs required much more time than measuring raw metrics from a single version
  – FreeMind
    • MCs: 28 minutes
    • Raw: 1 minute
  – JHotDraw
    • MCs: 40 minutes
    • Raw: 1 minutes
  – HelpSetMaker
    • MCs: 18 minutes
    • Raw: 1 minutes
Conclusion

• Metrics Constancy (MC), which is an indicator for predicting problematic modules, was proposed.

• MCs were compared with raw CK metrics.
  – The case study showed that MCs could identify fault-prone modules more precisely than raw CK metrics.

• In the future, we are going to
  – conduct more case studies on software written in other programming languages (e.g., C++, C#)
  – compare MCs with other identification methods.