A LEARNING-BASED METHOD FOR DETECTING DEFECTIVE CLASSES IN OBJECT-ORIENTED SYSTEMS

Cagil Biray
Ericsson R&D Turkey

Assoc. Prof. Feza Buzluca
Istanbul Technical University

10th Testing: Academic and Industrial Conference
Practice and Research Techniques (TAIC PART)
Agenda

• INTRODUCTION
• HYPOTHESIS & OBSERVATIONS
• DEFECT DETECTION APPROACH
• CREATING THE DATASET
• CONSTRUCTING THE DETECTION MODEL
• EXPERIMENTAL RESULTS
• CONCLUSION
• Q&A
INTRODUCTION
SOFTWARE DESIGN QUALITY

• Definition:
"capability of software product to satisfy stated and implied needs when used under specified conditions."

• How to assess the quality of software?
  – Understandability, maintainability, modifiability, flexibility, testability...

• Poorly designed classes include structural design defects.
SOFTWARE DESIGN DEFECTS

• Structural defects are not detectable during compile-time or run-time.
• They reduce the quality of software as a cause the following problems:
  – Reduce the flexibility of software
  – Vulnerable to introduction of new errors
  – Reduce the reusability.
OBJECTIVE

• Our main objective is to predict structurally defective classes of software.

• Two important benefits:
  – Helps testers to focus on faulty modules,
    ✓ Saves testing time.
  – Developers can refactor classes to correct design defects,
    ✓ Reduces probability of errors.
    ✓ Reduces the maintenance costs in future releases.
HYPOTHESIS & OBSERVATIONS
HYPOTHESIS

• Structurally defective classes mostly have following properties:
  – High class complexity, high coupling, low internal cohesion, inappropriate position in inheritance hierarchy.

• How to measure these properties?
  – Software design metrics

*Various metric types, distributions and different minimum/maximum values*...
MAIN OBSERVATIONS

• Structurally defective classes tend to generate most of the errors in tests, but healthy classes are also involved in some bug reports.

• Defective classes may not generate errors if they are not changed; errors arise after modifications.

• Healthy classes are not changed frequently and if they are modified they generate errors very rarely.
DEFECT DETECTION APPROACH
THE SOURCE PROJECTS

• 2 long-standing projects developed by Ericsson Turkey.
  – *Project A*: 6-years development, 810 classes.
  – *Project B*: 4-years development, 790 classes.

• Release reports of each project is analyzed.
  ➢ Determine the reasons for changes
    • *Is it a bug?*
    • *Is it a change request (CR)?*
THE PROPOSED DEFECT DETECTION APPROACH

• A learning-based method for defect prediction: Learn from history, predict the future.
  – Rule-based methods, machine-learning algorithms, detection-strategies...

• How to construct dataset? (instances-attributes-labels)
  – Metric collection: iPlasma, ckjm tool.
  – Class labels: defective/healthy?

• How to create a learning model?
  – Decision trees.
    • J48 algorithm.
2 long-standing projects developed by Ericsson Turkey.

Project A: 6-years development, 810 classes.

Project B: 4-years development, 790 classes.
USING RELEASES FOR TRAINING AND EVALUATION

• We constructed the training set examining classes from 46 successive releases of the Project A.
• Applied model to test release of same project.
• Observed errors and changes in classes for 49 consecutive releases.
• Also, applied same model to a test release from Project B.
• Evaluated the performance of our method observing 49 releases of Project B.
USING RELEASES FOR TRAINING AND EVALUATION (cont’d)

• $x = 46$ consecutive releases (training set)
• $y = 49$ consecutive releases (observation releases)
CREATING THE DATASET
Creating the Dataset

- Several releases of a project are examined to gather bug fix/CR information for each class.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>WMC</th>
<th>CBO</th>
<th>NOM</th>
<th>LOC</th>
<th>LCOM</th>
<th>DIT</th>
<th>WOC</th>
<th>HIT</th>
<th>.....</th>
<th>LABEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 1</td>
<td>53</td>
<td>39</td>
<td>16</td>
<td>288</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>Class 2</td>
<td>180</td>
<td>68</td>
<td>45</td>
<td>1051</td>
<td>107</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>Class 3</td>
<td>108</td>
<td>69</td>
<td>30</td>
<td>717</td>
<td>1313</td>
<td>0</td>
<td>0,49</td>
<td>3</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>.....</td>
<td>128</td>
<td>8</td>
<td>74</td>
<td>597</td>
<td>694</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>Class n</td>
<td>95</td>
<td>40</td>
<td>22</td>
<td>453</td>
<td>2399</td>
<td>0</td>
<td>0,6</td>
<td>1</td>
<td>...</td>
<td>1</td>
</tr>
</tbody>
</table>
PARAMETERS of CLASS LABELING

• **ErrC (Error Count):** The total number of bug fixes which are made on a class in the observed training releases.

\[ ErrC_c = \sum_{i=1}^{x} e_{c,i} \]

• **CR (Change Request) Count:** The total number changes in the class made because of CRs of the customer.

\[ CR\ count_c = \sum_{i=1}^{p} r_{c,i} \]
PARAMETERS of CLASS LABELING (cont’ d)

- **ChC (Change Count):** The total number of changes in a class during the training releases.

  \[ ChC_c = \text{Err}C_c + \text{CR count}_c \]

- **EF (Error Frequency):** The ratio between error count and change count of a class.

  \[ EF_c \% = \frac{\text{Err}C_c}{\text{ChC}_c} * 100 \]
THRESHOLD SELECTION

Training Set:
• Structural defective classes tend to change at least 5 times and their EFs are higher than 0.25.

\[ ChC \geq 5 \]
\[ EF \geq 0.25 \]

✓ \( t_1 \) is used for \( ChC \), \( t_2 \) is used for \( EF \).
• **Thresholds** are determined with the help of development team and experimental results.

• 2 thresholds for class labeling in training set:
  – $t_1$ is used for $ChC$, $t_2$ is used for $EF$.

  $tag_c = \text{Defective, if } (ChC_c \geq t_1 \text{ and } EF_c \geq t_2)$
An Example: Defective Class

ChC > 3 & EF > 0.25

<table>
<thead>
<tr>
<th>Class No.</th>
<th>Is a Bug?</th>
<th>Is a CR?</th>
<th>Error Count (ErrC)</th>
<th>CR Count</th>
<th>Change Count (ChC)</th>
<th>Error Frequency (EF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>YES</td>
<td>NO</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1/1</td>
</tr>
<tr>
<td>1</td>
<td>NO</td>
<td>YES</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1/2</td>
</tr>
<tr>
<td>1</td>
<td>YES</td>
<td>NO</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2/3</td>
</tr>
<tr>
<td>1</td>
<td>YES</td>
<td>NO</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>3/4</td>
</tr>
</tbody>
</table>
An Example: Healthy Class

### Class Release Report

<table>
<thead>
<tr>
<th>Class No.</th>
<th>Is a Bug?</th>
<th>Is a CR?</th>
<th>Error Count (ErrC)</th>
<th>CR Count</th>
<th>Change Count (ChC)</th>
<th>Error Frequency (EF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>YES</td>
<td>NO</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1/1</td>
</tr>
<tr>
<td>1</td>
<td>NO</td>
<td>YES</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1/2</td>
</tr>
<tr>
<td>1</td>
<td>NO</td>
<td>YES</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1/3</td>
</tr>
<tr>
<td>1</td>
<td>NO</td>
<td>YES</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>1/4</td>
</tr>
</tbody>
</table>

ChC > 3 & EF > 0.25

What about 0/0 error frequencies?
RARELY & UNCHANGED CLASSES

• Not correct to tag them as "healthy".
• The common characteristic of high-EF classes: **complexity metric** (WMC) value is high.

\[
tag_c = \begin{cases} 
\text{Defective, if } (ChC_c \geq t_1 \text{ and } EF_c \geq t_2), \\
\text{Defective, if } ((ChC_c < t_1 \text{ or } EF_c < t_2) \text{ and } WMC_c \\ \quad \geq AVG*1.5), \\
\text{Healthy, otherwise.}
\end{cases}
\]
CONSTRUCTING THE DETECTION MODEL
CONSTRUCTING THE DETECTION MODEL

• A classification problem within the concept of machine learning.

• J48 decision-tree learner.
DECISION TREE ANALYSIS

- J48 algorithm selects metrics strongly related to defect-proneness of the classes.
EXPERIMENTAL RESULTS
CREATING THE TRAINING SET

<table>
<thead>
<tr>
<th>Expression</th>
<th>Quantity</th>
<th>Classification Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChC ≥ 5 and EF ≥ 0.25</td>
<td>45</td>
<td>Defective</td>
</tr>
<tr>
<td>(ChC &lt; 5 or EF &lt; 0.25) and WMC_c ≥ AVG(WMC_{dc})*1.5</td>
<td>2</td>
<td>Defective</td>
</tr>
<tr>
<td>(ChC &lt; 5 or EF &lt; 0.25) and WMC_c &lt; AVG(WMC_{dc})*1.5</td>
<td>200</td>
<td>Healthy</td>
</tr>
</tbody>
</table>

- **247** classes, **23** object-oriented metrics and **defective/healthy class tags** in data set.
- **J48 classifier** algorithm selected 5 metrics: \textit{CBO, LCOM, WOC, HIT and NOM.}
RESULTS OF EXPERIMENTS (Project A)

- We applied unseen test release to decision tree model.

- Predictions
  - 53 out of 807: defective
  - 81% of the most defective classes
  - 18 classes with 0/0 EFs: 13 of them are defective.

<table>
<thead>
<tr>
<th>ErrC / ChC = EF</th>
<th>Total # of Defective Classes</th>
<th>Total # of Correctly Detected Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 / 11 = 0.73</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7 / 11 = 0.64</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6 / 12 = 0.5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6 / 10 = 0.6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6 / 7 = 0.86</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5 / 11 = 0.45</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5 / 10 = 0.5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5 / 9 = 0.56</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5 / 8 = 0.63</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5 / 7 = 0.71</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4 / 10 = 0.4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4 / 6 = 0.67</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4 / 5 = 0.8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3 / 7 = 0.43</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3 / 6 = 0.5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3 / 5 = 0.6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2 / 5 = 0.4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
RESULTS OF EXPERIMENTS (Project B)

• Predictions
  – 41 out of 789: defective
  – 83% of the most defective classes.
  – 7 classes with 0/0 EFs: 4 of them are defective.

<table>
<thead>
<tr>
<th>ErrC / ChC = EF</th>
<th>Total # of Defective Classes</th>
<th>Total # of Correctly Detected Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 / 10 = 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9 / 11 = 0.82</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8 / 9 = 0.89</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7 / 7 = 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6 / 8 = 0.75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6 / 7 = 0.86</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5 / 6 = 0.83</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5 / 5 = 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4 / 5 = 0.8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3 / 6 = 0.5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3 / 5 = 0.6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
CONCLUSION
CONCLUSION

• Our proposed approach ensures the early detection of defect-prone classes and provides benefits to the developers and testers.

• Helps testers to focus on faulty modules of software: saves significant proportion of testing time.

• Developers can refactor classes to correct their design defects: reduce the maintenance cost in further releases.
Q&A

Thank you.